|  |
| --- |
|  |
| MVC NOTES |
| [**www.enosislearning.com**](http://www.enosislearning.com) |

[Overview of MVC 3](#_Toc430812601)

[MVC Architecture 4](#_Toc430812602)

[Components of MVC(Model-View-Controller) 4](#_Toc430812603)

[Model 4](#_Toc430812604)

[View 4](#_Toc430812605)

[Controller 4](#_Toc430812606)

[Flow of MVC 5](#_Toc430812607)

[Advantages 6](#_Toc430812608)

[Disadvantages: 7](#_Toc430812609)

[History of M.V.C 7](#_Toc430812610)

[MVC File Structure & File Naming Standards 9](#_Toc430812611)

[ASP.NET MVC Execution Life Cycle 9](#_Toc430812612)

[STEP 1: Browser Request 10](#_Toc430812613)

[STEP 2: Job of Global.asax – MVC routing 10](#_Toc430812614)

[STEP 3: Controller and Action methods 11](#_Toc430812615)

[STEP 4: Call to View 11](#_Toc430812616)

[Asp.net MVC Request Life Cycle 11](#_Toc430812617)

[1. Routing 11](#_Toc430812618)

[2. MvcHandler 12](#_Toc430812619)

[3. Controller 12](#_Toc430812620)

[4. View Result 12](#_Toc430812621)

[5. View Engine 12](#_Toc430812622)

[6. View 12](#_Toc430812623)

[View Engine 12](#_Toc430812624)

[Razor 13](#_Toc430812625)

[ASPX 13](#_Toc430812626)

[Web Forms, MVC, Single Page App, Web Pages Comparison 14](#_Toc430812627)

[Web Forms 15](#_Toc430812628)

[MVC 16](#_Toc430812629)

[Single Page App 18](#_Toc430812630)

[Web Pages 19](#_Toc430812631)

[Controller Actions 20](#_Toc430812632)

[**Controller** 20](#_Toc430812633)

[Action Types In MVC 21](#_Toc430812634)

[Controllers and Action Methods in ASP.NET MVC Applications 22](#_Toc430812635)

[UNDERSTANDING MVC CONTROLLER 23](#_Toc430812636)

[GET and POST Request Type 23](#_Toc430812637)

[**Calculate Simple Interest** 24](#_Toc430812638)

[Get the view's data in the Action Method 25](#_Toc430812639)

[1 View-to-Controller 31](#_Toc430812640)

[1.1 Traditional Approach 31](#_Toc430812641)

[1.2 Through FormCollection: 31](#_Toc430812642)

[1.3 Through Parameters: 31](#_Toc430812643)

[1.4 Strongly typed model binding to view: 32](#_Toc430812644)

[2 Controller-to-View 33](#_Toc430812645)

[2.1 ViewData: 33](#_Toc430812646)

[2.2 ViewBag: 33](#_Toc430812647)

[2.3 TempData: 34](#_Toc430812648)

[3 Controller-to-Controller 35](#_Toc430812649)

[**ViewData** 35](#_Toc430812650)

[**ViewBag** 35](#_Toc430812651)

[**TempData** 36](#_Toc430812652)

[**Overview of Approaches in MVC** 38](#_Toc430812653)

[**ViewModel** 38](#_Toc430812654)

[**Partial View** 39](#_Toc430812655)

[**ViewBag** 39](#_Toc430812656)

[**ViewData** 40](#_Toc430812657)

[**TempData** 40](#_Toc430812658)

[**Tuple** 41](#_Toc430812659)

[Using the Standard HTML Helpers 41](#_Toc430812660)

[Rendering Form Elements 43](#_Toc430812661)

[CSS Class 44](#_Toc430812662)

[Example: Using Controls in MVC View Pages. 46](#_Toc430812663)

[Rendering a Form 48](#_Toc430812664)

[Different types of HTML Helpers 49](#_Toc430812665)

[Validation Rules 49](#_Toc430812666)

[**Adding Validation Rules to the Model** 49](#_Toc430812667)

[MVC CLIENT SIDE VALIDATIONS 52](#_Toc430812668)

[Output of Application 55](#_Toc430812669)

[**Approach 1: Manually Add Error to ModelState Object** 57](#_Toc430812670)

[**Run the Application and Test in Various Ways** 59](#_Toc430812671)

[**Approach 2: Specifying Business Rules with Data Annotation** 60](#_Toc430812672)

# Overview of MVC

“M” “V” “C” stands for “MODEL” “VIEW” “CONTROLLER”.

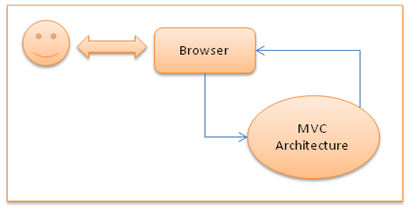
MVC design pattern aims to separate content from presentation and data-processing from content.

ASP.NET MVC is architecture to develop ASP.NET web applications in a different manner than the traditional ASP.NET web development.

Developing ASP.NET MVC application requires Microsoft .NET Framework 3.5 or higher.

**MVC Interaction with Browser**

Like a normal web server interaction, MVC application also accepts requests and responds to the web browser in the same way.



# MVC Architecture

The entire ASP.NET MVC architecture is based on Microsoft .NET Framework and in addition uses LINQ to SQL Server.

## Components of MVC(Model-View-Controller)

## Model

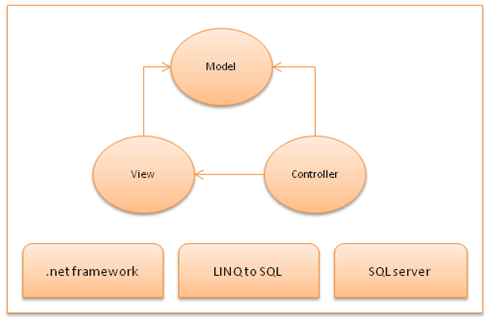
* MVC **model** is basically a C# or VB.NET class
* A **model** is accessible by both **controller** and **view**
* A **model** can be used to pass data from **Controller** to **view**
* A Model can also be used to get data from View to the Controller.
* A **view** can use model to display data in page.

## View

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

## Controller

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



# Flow of MVC

|  |
| --- |
| **Step 1**: Incoming request directed to **Controller.**  View,Model,Presenter,Model |
| **Step 2**: **Controller** processes request and forms a data **Model.**  View,Model,Presenter,Model |
| **Step 3**: **Model** is passed to **View.**  View,Model,Presenter,Model |
| **Step 4**: **View** transforms **Model** into appropriate output format.  View,Model,Presenter,Model |
| **Step 5**: Response is rendered.  View,Model,Presenter,Model |

# Advantages

* Separation of concerns –

One of the advantages of using ASP.NET MVC is it enforces separation of concerns. So there is very less chances of getting things more complex.

|  |  |
| --- | --- |
|  |  |

* Test Driven development and Reusability –

|  |  |
| --- | --- |
| * + In MVC controller is a separate class so automatic testing is possible featuring Test Driven Development.   + Controllers are not bound to any specific view and so can be reused for multiple views. | http://www.codeproject.com/KB/aspnet/528117/MVCTESTING_AND_REUSE.png |

* **Performance - ASP.NET MVC don’t have support for view state**, so there will not be any automatic state management which reduces the page size and so gain the performance.
* **Full control over HTML - ASP.NET MVC doesn’t support server controls,** only option available is using html input controls, so we will be sure about final html rendered at the end.

And **so integration of ASP.NET MVC application with third party JavaScript libraries like jQuery becomes easy**.

* Support for parallel development- **In ASP.NET MVC layers are loosely coupled with each other, so one developer can work on Controller, at the same time other on View and third developer on Model. This is called parallel development.**
* **SEO, URL routing and REST** - Rich routing features lets treat every URL as a resource supporting RESTful interfaces.

http://www.codeproject.com/KB/aspnet/528117/15.JPG

Also user-friendly and readable URL improves SEO.

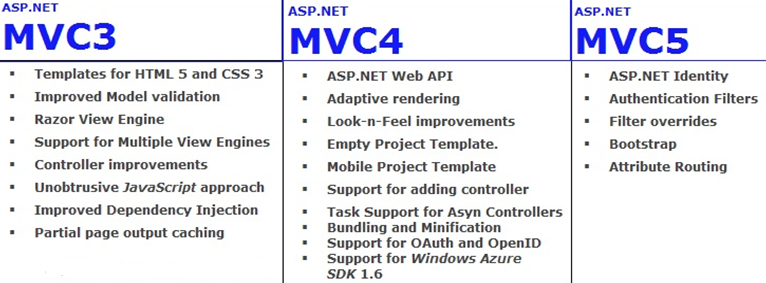
* Extensibility - ASP.NETMVC supports multiple view engines like aspx, razor and if required we can create our own.
* Existing ASP.NET Features – ASP.NET MVC framework is built on top of matured ASP.NET framework and thus provides developer to use many good features such as forms authentication, windows authentication, caching,session etc.

# Disadvantages:

* More learning effort - Absence of event driven programming model and ViewState makes ASP.NET MVC a very difficult framework for developers with no or little experience in web application development.

# History of M.V.C

|  |  |
| --- | --- |
| **ASP.NET MVC 1.0** | The [first official release, dated MIX 2009](http://codeclimber.net.nz/archive/2009/03/18/asp.net-mvc-goes-rtm-before-mix09.aspx), brought all the core features of the framework, which made their way through the years till today:   * Obviously, the MVC concept, with the streamlined processing pipeline and the separation of the processing of the request (in the Controller) and the rendering of the output (in the View) * The concept of routing * Lightweight helpers for rendering HTML tags * Ajax helpers to easily ajaxify links and forms * Automatic binding of posted forms to .NET objects and a kind of model validation |
| **ASP.NET MVC 2** | The following year, same timeframe (MIX 2010), [the second version of the framework came out](http://weblogs.asp.net/scottgu/archive/2010/03/11/asp-net-mvc-2-released.aspx).   * **Model validation based on attributes, both server and client side** * Areas to partition bigger applications * Html Templated helpers, to automatically render edit forms and display pages based on the model and attributes applied on it * Asynchronous Controllers * Lambda based Html Helpers to remove most of the "magic strings" previously needed in html helpers   The inclusion of the model validation and display attributes was also a first step toward some kind of support for a default model inside ASP.NET MVC. |
| **ASP.NET MVC 3** | At the beginning of 2011, less than a year after the previous version, [version 3 was released](http://weblogs.asp.net/scottgu/archive/2011/01/13/announcing-release-of-asp-net-mvc-3-iis-express-sql-ce-4-web-farm-framework-orchard-webmatrix.aspx), together with a bunch of other pretty interesting tools, like NuGet, IIS Express and SQL Server Express.  The new features introduced with this latest version:   * Works only on .NET4 * Unobtrusive JavaScript validation and general better JavaScript love * Remote validation * Dependency Resolver * The new Razor view engine * Global Filters   In the context of extensibility, a few more were introduced:   * Service Locator, DependencyResolver, Filter Provider,IvalidatableObject |
| **ASP.NET MVC Tool Update** | Finally, as tradition, they released something also at MIX 2011: no changes to the library itself, but justan upgrade to the tooling around MVC:   * They integrated the great MvcScaffolding into Visual Studio, to easily create controllers and CRUD views based on the model * Libraries included by default (jQuery, jQuery UI and modernizer for example) are included as NuGet packages so that they can be easily upgraded * Entity Framework CodeFirst is included as default data access library in the default project template * The default template is HTML5 enabled (if a checkbox is selected, views will have the semantic HTML5 tags)   With this last update ASP.NET MVC finally had a default implementation of the M part of the pattern: Entity Framework CodeFirst with automatic creation of the DB schema attributes for data validation and editing, and scaffolding for the automatic generation of CRUD actions and views. |



**ASP.NET MVC 3:**

1.    MVC 3 introduced a popular new features that is called **Razor**.

2.    MVC 3  introduced a new features that is called **bundling**.

3.    MVC 3 Start supporting to multiple view(Razor) engines.

4.    MVC 3 introduced a popular new features that is called ViewBag dynamic property.

5.    MVC 3 introduced a popular new features that is called ActionResults Types.

6.    In MVC 3, added  separates the functionality approach over the JavaScript in the presentation layer  that is called to UX.

7.    MVC 3 start supporting to the **HTML 5** and **CSS 3**.

8.    MVC 3 added enhancement to validate a **model**.

9.    MVC 3 improved the Dependency Injection by using the IDependencyResolver interface.

10.  MVC 3 added output caching for Partial views.

**ASP.NET MVC 4:**

1.    MVC 4 introduced a popular new features that is called webapi  framework. and its work over the HTTP services .

2.    MVC 4 added and improve to **default project templates**.

3.    MVC 4 introduced to empty project template.

4.    MVC 4 introduced to new **mobile**project template.

5.    MVC 4 is going to support and adding **controller**to other project folders.

6.    MVC 4 is going to support the **Asynchronous Controllers**.

7.    MVC 4 introduced to **bundling**and **Magnification**concepts.

8.    MVC 4 introduced to **OAuth** and **OpenID** login with the help of **DotNetOpenAuth** library.

9.    MVC 4 going to support to **Windows Azure** SDK 1.6.

**ASP.NET MVC 5:**

1.    MVC 5 introduced to ASP.NET Identity for authentication and identity management.

2.    ASP.NET Identity is a new Membership provider to handle the authentication and authorization for social networking site just like Google, twitter, face-book etc.

3.    MVC 5 added  to authentication filters for authenticate to users using  third party authentication provider or your custom logic by using **filter override** methods and now we can override to filters on  method and  controller both.

4.    MVC 5 replaced to bootstrap by using the default template.

5.    MVC 5 added new to **Attribute Routing**[Route("Empolyee/{EmpID}")].

Example code look like.

[RoutePrefix("API/Company")]

public class CompanyController : BaseAPIController

{

        [Route("Employee/{EmpId}")]

        public ActionResult GetEmployeeById(string EmpId)

        {

            //TODO: Put your logic here.

            return View();

        }  
}

The **Attribute Routing**work with both **ActionResult** and **Controller** Level.

**Now, In MVC 5**

1. Empty Template
2. Internate Application Template
3. Intranet Application Template
4. ASP.NET Web API Template
5. Mobile Project Template
6. Single Page Application Template
7. Asp.Net Identity
8. Bootstrap in the MVC 5 templates
9. Authentication Filters
10. Filter overrides

**ASP.NET MVC 6:**

1.     MVC 6 added new cloud computing optimization system of MVC , web API, SignalR and entity framework.

2.    The Microsoft  make a bundle of MVC, Web API, WebPages, SignalR , That bundle we called MVC 6.

3.    In MVC 6, Microsoft removed the dependency of system.web.dll from MVC 6  because it's so expensive. Typically  it consume 30K memory per request/response.

4.    Right now, in MVC 6 consume 2K  memory per request response. It's too small memory  consume.

5.    Most of the problem solved using the Roslyn Compiler.

6.    The ASP .Net  vNext used the Roslyn Compiler,  By using Roslyn compiler do not need to compile the application Its  compile automatically the application code.

7.    The .Net vNext is a cross platform and open source.

8.    The .Net vNext has the new project extension **project.json.**Basically project.json contain the all dependency dll of the application.  
  
In MVC 5.1 and 5.2 support to Enum and EnumHelper in  razor views.

## MVC File Structure & File Naming Standards

MVC uses a standard directory structure and file naming standards which are a very important part of MVC application development.

Inside the ROOT directory of the application, there must be 3 directories each for model, view and Controller.

Apart from 3 directories, there must have a ***Global.asax***file in root folder, and a *web.config* like a traditional ASP.NET application.

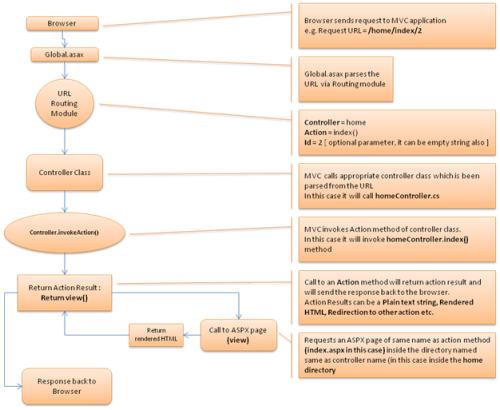
|  |
| --- |
| * **Root**[directory]   + - **Controller** [directory]       * Controller CS files     - **Models** [directory]       * Model CS files     - **Views**[directory]       * View aspx/ascx files   + *Global.asax*   + *Web.config* |

# 

# ASP.NET MVC Execution Life Cycle

Here is how MVC architecture executes the requests to browser and objects interactions with each other.

A step by step process is explained below **[Refer to the figure as given below]**:



## STEP1: Browser Request : Browser request happens with a specific URL. Let’s assume that the user enters URL like: [xyz.com]/home/index/

## STEP 2:Job of Global.asax – MVC routing : The specified URL will first get parsed via application\_start() method inside *Global.asax* file. From the requested URL, it will parse the Controller, Action and ID.

So for [xyz.com]/home/index/:

* Controller = home
* Action = index()
* ID = empty — we have not specified ID in [xyz.com]/home/index/, so it will consider as empty string

## STEP 3: Controller and Action methods

MVC now finds the **home controller** class in controller directory. A controller class contains different action methods,There can be more than one action method, but MVC will only invoke the action method which has been parsed from the URL, its index() in our case.

So something like: homeController.index() will happen inside MVC controller class.

Invoking action method can return plain text string OR rendered HTML by using view.

## STEP 4: Call to View

Invoking view will return view(). A call to view will access the particular ASPX page inside the view directory and generate the rendered HTML from the ASPX and will respond back to the browser.In our case, controller was **home**and action was index(). So calling view() will return a rendered HTML from the ASPX page located at */views/home/index.aspx*.

# Asp.net MVC Request Life Cycle

While programming with Asp.net MVC, you should be aware of the life of an Asp.net MVC request from birth to death. In this article, I am going to expose the Asp.net MVC Request Lifecycle. There are six main steps that happen when you make a request to an Asp.net MVC web applications:

## Routing

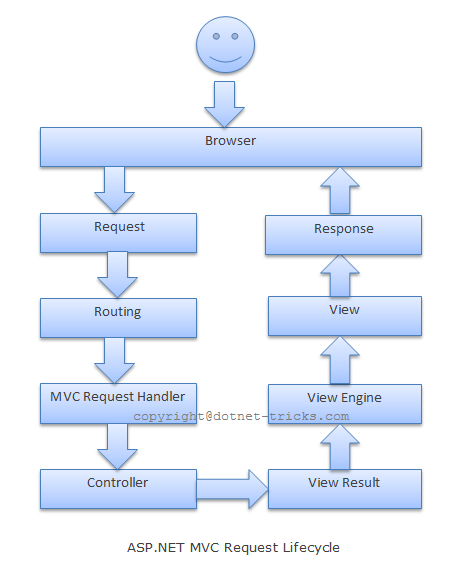
Asp.net Routing is the first step in MVC request cycle. Basically it is a pattern matching system that matches the request’s URL against the registered URL patterns in the Route Table. When a matching pattern found in the Route Table, the Routing engine forwards the request to the appropriate handler for that request. Otherwise, the routing engine returns a 404 HTTP status code against that request.

When application starts at first time, it registers one or more patterns to the Route Table to tell the routing system what to do with any requests that match these patterns. An application has only one Route Table and this is setup in the Global.asax file of the application.

* 1. **publicstaticvoidRegisterRoutes(RouteCollection routes)**
  2. **{**
  3. **routes.IgnoreRoute("{resource}.axd/{\*pathInfo}");**
  4. **routes.MapRoute("Default",*// Route name***
  5. **"{controller}/{action}/{id}",*// URL with parameters***
  6. **new{ controller ="Home", action ="Index", id =UrlParameter.Optional}*// Parameter defaults***
  7. **);**
  8. **}**

## MvcHandler

The handler returned from the RouteTable will always be an MvcHandler and represents a particular RouteContext. Actually, MvcHandler creates a controller, and also executes the controller.

****

## Controller

The controller determines which action method of the controller class to call, and then calls that method

## View Result

The action method receives user input, prepares the appropriate response data, and then executes the result by returning a result type. The result type can be ViewResult, RedirectToRouteResult, RedirectResult, ContentResult, JsonResult, FileResult, and EmptyResult.

## View Engine

Now the ViewResult calls into the current view engine to render the view. Controller is responsible only for preparing the data and decides which view to display by returning a ViewResult instance.

## View

The data rendered by the current view engine is displayed on the view.

# View Engine

What is View Engine? Types of View Engine

View Engine is responsible for rendering the view into html form to the browser. By default, Asp.net MVC support Web Form (ASPX) and Razor View Engine. There are many third party view engines (like Spark, Nhaml etc.) that are also available for Asp.net MVC. Now, Asp.net MVC is open source and can work with other third party view engines like Spark, Nhaml.

Note: All view engines may not support all versions of ASP.NET MVC. Please check the corresponding sites to get the latest updates.

## Razor

The Razor view engine is an advanced view engine from Microsoft, packaged with MVC 3. Razor using an @ character instead of aspx's <% %> and Razor does not require you to explicitly close the code-block, this view engine is parsed intelligently by the run-time to determine what is a presentation element and what is a code element.

This is the default view engine in MVC . Web pages with Razor syntax have the special file extension cshtml (Razor with C#) or vbhtml (Razor with VB).

Sample code is given below,

**<h1>Sample with Razor</h1>**

**<h3>**

**Hello @name, the year is @DateTime.Now.Year**

**</h3>**

**<p>**

**checkout <ahref="/Products/Details/@produtid">this prodcut</a>**

**</p>**

## ASPX

The syntax for writing views with this engine is the same syntax that the ASP.NET Web Forms uses and the file extensions are also taken from ASP.NET Web Form (.*aspx*, .ascx, .master). The coding will give us the memory of legacy ASP style.

This is the default view engine for MVC 1 and MVC 2.

Sample code is given below,

**<h1>Sample with ASPX </h1>**

**<h3>**

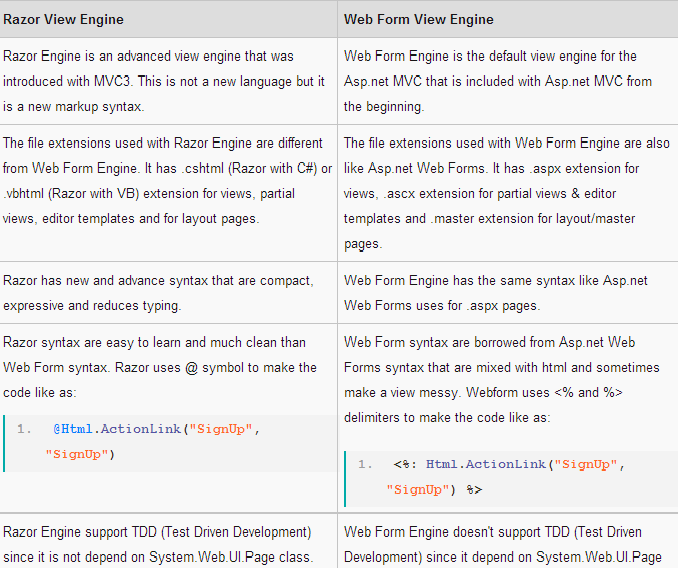
**Hello <%=name %>, the year is <%= DateTime.Now.Year %>**

**</h3>**

**<p>**

**checkout <ahref="/Products/Details/<%=produtid %>">this prodcut</a>**

**</p>**

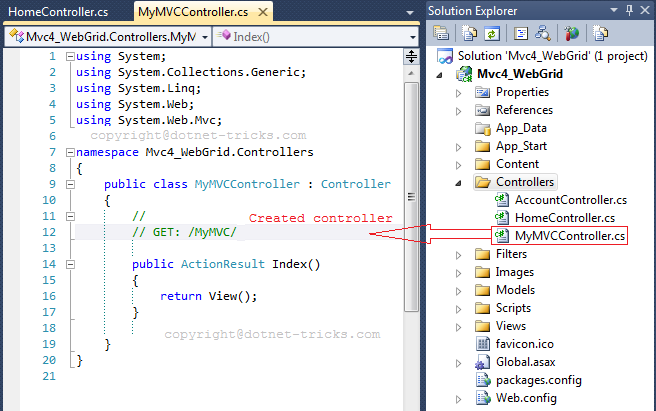


# Controller Actions

Asp.net MVC Controllers are responsible for controlling the flow of the application execution. When you make a request (means request a page) to MVC applications, a controller is responsible for returning the response to that request. A controller can have one or more actions. A controller action can return different types of action results to a particular request.

**Controller**

Basically, the controller Receives input like as form values, query strings values etc. from users via the View and perform required operations on the user's inputs with the help of Model and passing the results back to the View. Actually, controllers are classes that have methods or action results and these actions results are called by the routing system to process a particular request.



Controller actions are methods defined in the controller class and responsible to perform required operations on the user's inputs like as form values, query strings values etc. with the help of Model and passing the results back to the View.

# Action Types In MVC

**Asp.net MVC has the following Built-in Action Results Type and Helper methods:**

|  |  |  |
| --- | --- | --- |
| **Action Result** | **Helper Method** | **Description** |
| [ViewResult](http://msdn.microsoft.com/en-in/library/system.web.mvc.viewresult(v=vs.98).aspx) | [View](http://msdn.microsoft.com/en-in/library/system.web.mvc.controller.view(v=vs.98).aspx) | Renders a view as a Web page. |
| **Renders a specified view to the response stream** | | |
| [PartialViewResult](http://msdn.microsoft.com/en-in/library/system.web.mvc.partialviewresult(v=vs.98).aspx) | [PartialView](http://msdn.microsoft.com/en-in/library/system.web.mvc.controller.partialview(v=vs.98).aspx) | Renders a partial view, which defines a section  Of a view that can be rendered inside another view. |
| **Renders a specified partial view to the response stream** | | |
| [RedirectResult](http://msdn.microsoft.com/en-in/library/system.web.mvc.redirectresult(v=vs.98).aspx) | [Redirect](http://msdn.microsoft.com/en-in/library/system.web.mvc.controller.redirect(v=vs.98).aspx) | Redirects to another action method by using its URL. |
| **Performs an HTTP redirection to a specified URL** | | |
| [RedirectToRouteResult](http://msdn.microsoft.com/en-in/library/system.web.mvc.redirecttorouteresult(v=vs.98).aspx) | [RedirectToAction](http://msdn.microsoft.com/en-in/library/system.web.mvc.controller.redirecttoaction(v=vs.98).aspx)  [RedirectToRoute](http://msdn.microsoft.com/en-in/library/system.web.mvc.controller.redirecttoroute(v=vs.98).aspx) | Redirects to another action method. |
| **Performs an HTTP redirection to a URL that is determined by the routing engine, based on given route data** | | |
| [ContentResult](http://msdn.microsoft.com/en-in/library/system.web.mvc.contentresult(v=vs.98).aspx) | [Content](http://msdn.microsoft.com/en-in/library/system.web.mvc.controller.content(v=vs.98).aspx) | Returns a user-defined content type. |
| **Writes content to the response stream without requiring a view** | | |
| [JsonResult](http://msdn.microsoft.com/en-in/library/system.web.mvc.jsonresult(v=vs.98).aspx) | [Json](http://msdn.microsoft.com/en-in/library/system.web.mvc.controller.json(v=vs.98).aspx) | Returns a serialized JSON object. |
| **Serializes a given ViewData object to JSON format** | | |
| [JavaScriptResult](http://msdn.microsoft.com/en-in/library/system.web.mvc.javascriptresult(v=vs.98).aspx) | [JavaScript](http://msdn.microsoft.com/en-in/library/system.web.mvc.controller.javascript(v=vs.98).aspx) | Returns a script that can be executed on the client. |
| **Returns a piece of JavaScript code that can be executed on the client** | | |
| [FileResult](http://msdn.microsoft.com/en-in/library/system.web.mvc.fileresult(v=vs.98).aspx) | [File](http://msdn.microsoft.com/en-in/library/system.web.mvc.controller.file(v=vs.98).aspx) | Returns binary output to write to the response. |
| **Returns a file to the client** | | |
| [EmptyResult](http://msdn.microsoft.com/en-in/library/system.web.mvc.emptyresult(v=vs.98).aspx) | (None) | Represents a return value that is used if the action  method must return a **null** result (void). |
| **An empty response is returned** | | |

## 

In MVC all actions are performed through controller. Controller assume each method as an action. Each action method has a return type and we call it as action result. There are plenty of action results are available according to need but all action results are inherited by **ActionResult** class. This is the base action result used in controller to perform any type of action. We can return any action type with **ActionResult** class. Let see action results with some example.

It is preferred to make a method private if it is not an action method.

**Action Results**

There is a list of action results type available to be used as return type of a method in controller and **ActionResult** class is the base class for all. It means this type can be used with any method as a result. First of all let us see different types of action results available and then we can discuss each one by one.

|  |  |  |
| --- | --- | --- |
| **Action Result** | **Associated Method** | **Description** |
| ViewResult | View() | Renders a view. |
| PartialViewResult | PartialView() | Renders a partial view. |
| RedirectResult | Redirect(), RedirectPermanent() | Redirects to another action method using URL in method. |
| RedirectToRouteResult | RedirectToAction(), RedirectToActionPermanent(), RedirectToRoute(), RedirectToRoutePermanent() | Redirects to action. |
| ContentResult | Content() | Returns content. |
| FileResult | File() | Returns file or binary output. |
| JsonResult | Json() | Returns Json. |
| JavaScriptResult | JavaScript() | Returns JavaScript code to client. |
| HttpNotFoundResult | HttpNotFound() | Returns an error page showing File not found error message - 404. |
| EmptyResult | None | Returns null or void. |

FileResult is the base class of few more action results like FileContentResult, FilePathResult and FileStreamResult. All these uses File() method to render with different signatures.

Let's discuss each of them one by one. The most commonly used action results are ViewResult, PartialViewResult and JsonResult.

**ViewResult**

Used to renders a view. Below code will render Index view.

public ActionResult Index()  
        {  
            //It will render index view.  
            return View();  
        }

You can also render view by passing view name in **View()** method like below. Below method will render **Login** view.

        public ViewResult CallViewResult()  
        {  
            return View("Login");  
        }

**PartialViewResult**

This is used to renders a partial view. Below method takes int parameter and passing it as model. Also note that I am using **HttpPost** method type so this method cannot be directly called from browser.

        /// <summary>  
        /// This method returns a partial named \_InnerPartial  
        /// </summary>  
        /// <param name="value"></param>  
        /// <returns></returns>  
        [HttpPost]  
        public PartialViewResult CallPartial(int value)  
        {  
            return PartialView("\_InnerPartial", value);  
        }

By default no method type is defined on action method and it can be access as get or post. But if you define method type as HttpPost, it means it is of post type and if you define it as HttpGet then it is of get type.

**RedirectResult**

Used to redirects to another action method using URL in method. Below method will be redirected to given url.

        public RedirectResult CallRedirectResult()  
        {  
            //Redirect to given url  
            return Redirect("/Example/Index");  
        }

You can also permanently redirect to a perticular url with **RedirectPermanent()** method.

        public RedirectResult CallRedirectPermanentResult()  
        {  
            //url to redirect  
             return RedirectPermanent("/Example/Index");  
        }

**RedirectToRouteResult**

Used to redirects to another action. You can use **RedirectToAction()** method to perform this task or you can use **RedirectToRoute()** method. Let's first see **RedirectToAction()** method. We pass action name in this method and it will redirect to that method.

        /// <summary>  
        /// This method will search for route url in route table (RouteConfig.cs) and if found, it will show that route url in url else it will show action name.  
        /// This example will show route name in url as I have defined route for the method CallViewResult()  
        /// </summary>  
        /// <returns></returns>  
        public RedirectToRouteResult CallRedirectToAction()  
        {  
            TempData["mess"] = "Redirected from CallRedirectToAction method...";  
            //I have defined a route 'MyTestRoute' for this action and when CallRedirectToAction() method will execute, it will replace url with that route url.  
            return RedirectToAction("CallViewResult");  
        }

Now let's see an example of **RedirectToRoute()** method. Here we pass route name what we have defined in route table.

        public RedirectToRouteResult CallRedirectToRoute()  
        {  
            TempData["mess"] = "Redirected from RedirectToActionPermanent method...";  
            //put name of route and it will redirect to the url defined for that route.  
            //Below is the route defined in RouteConfig.cs   
            //routes.MapRoute(name: "RouteDefault",url: "MyTestRoute",  
            //   defaults: new { controller = "Example", action = "CallViewResult"});  
            return RedirectToRoute("RouteDefault");  
        }

**ContentResult**

We can return content with this result type. In below example I have created a small HTML template and rendered it with the help of ContentResult.

        public ContentResult CallContentResult()  
        {  
            System.Text.StringBuilder stringBuilder = new System.Text.StringBuilder();  
            stringBuilder.Append("<div style='width:100%;background-color:#ccc; float:left;padding-left:10px;'>");  
            stringBuilder.Append("<h1>This is header</h1>");  
            stringBuilder.Append("<h2>This is second header</h2>");  
            stringBuilder.Append("<h3>This is third header</h3>");  
            stringBuilder.Append("</div>");  
            return Content(stringBuilder.ToString());  
        }

**FileResult**

You can render a file or binary output with this result.

        /// <summary>  
        /// This will render image file  
        /// </summary>  
        /// <returns></returns>  
        public FileResult CallFileResult()  
        {  
            string file=Server.MapPath("~/Files/fileupload.png");  
            return File(file, "image");  
        }

**JsonResult**

Returns JSON result to client.

        public JsonResult CallJsonResult()  
        {  
            return Json(new { Message = "This is JSON data", JsonRequestBehavior.AllowGet });  
        }

**JavaScriptResult**

Returns javascript result to ajax call.

        public JavaScriptResult CallJavaScriptResult()  
        {  
            return JavaScript("alert('hello');");  
        }

**HttpNotFoundResult**

Returns HTTP Not Found error page.

        /// <summary>  
        /// Inherited from HttpStatusCodeResult class whose base class is again ActionResult class. Means HttpNotFoundResult>HttpStatusCodeResult>ActionResult  
        /// </summary>  
        /// <returns></returns>  
        public HttpNotFoundResult CallHttpNotFoundResult()  
        {  
            //Will render error page- Not found with message you are passing  
            return HttpNotFound("I am testing error");  
        }

**EmptyResult**

Returns null result.

        public EmptyResult CallEmptyResult()  
        {  
            //when nothing to return, can use this result.  
            return null;  
        }

## Controllers and Action Methods in ASP.NET MVC Applications

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 base class for all controllers is the [ControllerBase](http://msdn.microsoft.com/en-in/library/system.web.mvc.controllerbase(v=vs.98).aspx) class, which provides general MVC handling. The [Controller](http://msdn.microsoft.com/en-in/library/system.web.mvc.controller(v=vs.98).aspx) class inherits from **ControllerBase** and is the default implement of a controller. The **Controller** class is responsible for the following processing stages:

* Locating the appropriate action method to call and validating that it can be called.
* Getting the values to use as the action method's arguments.
* Handling all errors that might occur during the execution of the action method.
* Providing the default [WebFormViewEngine](http://msdn.microsoft.com/en-in/library/system.web.mvc.webformviewengine(v=vs.98).aspx) class for rendering ASP.NET page types (views).

All controller classes must be named by using the "Controller" suffix. The following example shows the sample controller class, which is namedHomeController. This controller class contains action methods that render view pages.

**[HandleError]**

**publicclass HomeController : Controller**

**{**

**public ActionResult Index()**

**{**

**ViewData["Message"] = "Welcome to ASP.NET MVC!";**

**return View();**

**}**

**public ActionResult About()**

**{**

**return View();**

**}**

**}**

[Marking Public Methods as Non-Action Methods](javascript:void(0))

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-in/library/system.web.mvc.nonactionattribute(v=vs.98).aspx) attribute.

The following example shows a method that is marked with the **NonAction** attribute.

**[NonAction]**

**privatevoid DoSomething()**

**{**

**// Method logic.**

**}**

# UNDERSTANDING MVC CONTROLLER

We use two types of methods to handle our browser request;

* HTTP GET
* HTTP POST

**When we call an action method by a request's URL by the browser then the HTTP GET method will be called but when a request is from a button click event then the HTTP POST method will be called.**

Explaining how to access view input field data in the controller's action method when a HTTP POST request is called.

## GET and POST Request Type

To access view input field data in the controller's action method (POST), we create a "Calculate Simple Interest" application.

This application gets Principle, Rate and Time as user input and generates simple interest.

Create an action method in the CalculateSimpleInterest controller *CalculateSimpleInterestController.cs*) that renders the view on the UI.

public ActionResult SimpleInterest()

{

return View();

}

Create a view to get user input from the UI, so the code is:

**Calculate Simple Interest**

<fieldset>

<legend>Calculate Simple Interest</legend>

@using (Ajax.BeginForm("CalculateSimpleInterestResult","CalculateSimpleInterest",

new AjaxOptions { UpdateTargetId = "divInterestDeatils" }))

{

<div id="divInterestDeatils"></div>

<ol>

<li>

@Html.Label("Amount")

@Html.TextBox("txtAmount")

</li>

<li>

@Html.Label("Rate")

@Html.TextBox("txtRate")

</li>

<li>

@Html.Label("Year")

@Html.TextBox("txtYear")

</li>

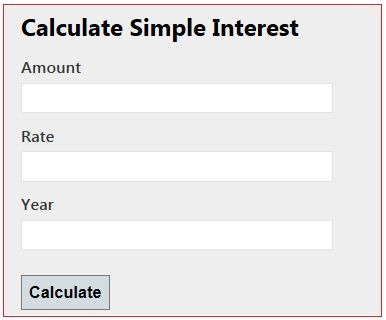
</ol>

<button>Calculate</button>

}

</fieldset>

So now the screen is ready to get input and it shows it as:



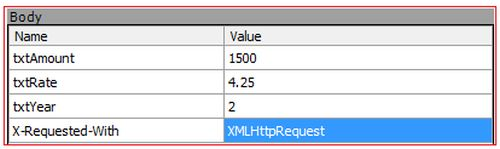
***Figure 1.1 Input screens to calculate simple interest***

## Get the view's data in the Action Method

I will now explain the four ways to get the view's data in the controller action. These are:

1. Using Traditional approach [Request]
2. Using the FormCollection Object
3. Using the Parameters
4. **Strongly type model binding to view**

#### Using Traditional Approach

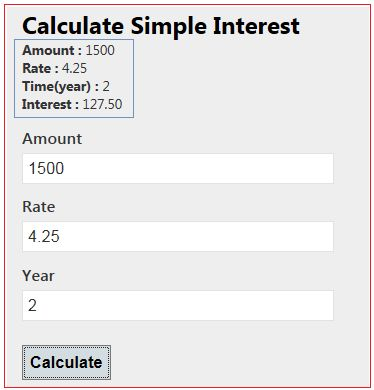
In the traditional approach we use the request object of the HttpRequestBase class. The request object has view input field values in name/value pairs. When we create a submit button then the request type POST is created and calls the POST method.

***Figure 1.2 Requested Data***

We have four data, those are in Name-Value pairs. So we can access these data in a POST method by passing the Name as an indexer in the Request and get values. Our POST method means the controller action that handles the POST request type is:

|  |
| --- |
| [HttpPost]  public ActionResult CalculateSimpleInterestResult()  {  decimal principle = Convert.ToDecimal(Request["txtAmount"].ToString());  decimal rate = Convert.ToDecimal(Request["txtRate"].ToString());  int time = Convert.ToInt32(Request["txtYear"].ToString());  decimal simpleInteresrt = (principle\*time\*rate)/100;  StringBuilder sbInterest = new StringBuilder();  sbInterest.Append("<b>Amount :</b> " + principle+"<br/>");  sbInterest.Append("<b>Rate :</b> " + rate + "<br/>");  sbInterest.Append("<b>Time(year) :</b> " + time + "<br/>");  sbInterest.Append("<b>Interest :</b> " + simpleInteresrt);  return Content(sbInterest.ToString());  } |
|  |

When it executes, we get simple interest as the result as in the following:



***Figure 1.3 Output screen after getting response***

#### Using the FormCollection Object

We can also get post requested data by the FormCollection object. The FormCollection object also has requested data in the name/value collection as the Request object. To get data from the FormCollection object we need to pass it is as a parameter and it has all the input field data submitted on the form.

[HttpPost]

public ActionResult CalculateSimpleInterestResult(FormCollection form)

{

decimal principle = Convert.ToDecimal(form["txtAmount"].ToString());

decimal rate = Convert.ToDecimal(form["txtRate"].ToString());

int time = Convert.ToInt32(form["txtYear"].ToString());

decimal simpleInteresrt = (principle\*time\*rate)/100;

StringBuilder sbInterest = new StringBuilder();

sbInterest.Append("<b>Amount :</b> " + principle+"<br/>");

sbInterest.Append("<b>Rate :</b> " + rate + "<br/>");

sbInterest.Append("<b>Time(year) :</b> " + time + "<br/>");

sbInterest.Append("<b>Interest :</b> " + simpleInteresrt);

return Content(sbInterest.ToString());

}

It also gives the same output as Figure 1.3 shows.

#### Using the Parameters

We can pass all input field names as a parameter to the post action method. The input field name and parameter name should be the same. These parameters have input field values that were entered by the user. So we can access view input field values from these parameters. The input field takes a string value from the user so the parameter should be a string type. There is no need to define a parameter in any specific sequence.

[HttpPost]

public ActionResult CalculateSimpleInterestResult(string txtAmount, string txtRate, string txtYear)

{

decimal principle = Convert.ToDecimal(txtAmount);

decimal rate = Convert.ToDecimal(txtRate);

int time = Convert.ToInt32(txtYear);

decimal simpleInteresrt = (principle\*time\*rate)/100;

StringBuilder sbInterest = new StringBuilder();

sbInterest.Append("<b>Amount :</b> " + principle+"<br/>");

sbInterest.Append("<b>Rate :</b> " + rate + "<br/>");

sbInterest.Append("<b>Time(year) :</b> " + time + "<br/>");

sbInterest.Append("<b>Interest :</b> " + simpleInteresrt);

return Content(sbInterest.ToString());

}

It also gives the same output as Figure 1.3 shows.

In all three approaches above we are parsing the string to a non-string type. If any of the parsing attempts fail then the entire action will fail. We are converting each value to avoid an exception but it also increases the amount of code. So we look at the fourth approach that would reduce the amount of code.

#### Strongly type model binding to view

We bind a model to the view; that is called strongly type model binding.

**Step 1**: Create a Model for Simple Interest.

namespace CalculateSimpleInterest.Models

{

public class SimpleInterestModel

{

public decimal Amount { get; set; }

public decimal Rate { get; set; }

public int Year { get; set; }

}

}

**Step 2**: Create an action method that render a view on the UI.

We are passing an empty model to be bound to the view.

public ActionResult SimpleInterest()

{

SimpleInterestModel model = new SimpleInterestModel();

return View(model);

}

**Step 3**: Create a strongly typed view that has the same screen as in Figure 1.1.

**@model CalculateSimpleInterest.Models.SimpleInterestModel**

**@{**

**ViewBag.Title = "SimpleInterest";**

**}**

**<h2>Calulate Simple Interest</h2>@using (Ajax.BeginForm("CalculateSimpleInterestResult","CalculateSimpleInterest",**

**new AjaxOptions { UpdateTargetId = "divInterestDeatils" }))**

**{**

**<fieldset>**

**<legend>Calulate Simple Interest</legend><div id="div1"></div>**

**<div class="editor-label">**

**@Html.LabelFor(model => model.Amount)**

**</div>**

**<div class="editor-field">**

**@Html.EditorFor(model => model.Amount)**

**</div>**

**<div class="editor-label">**

**@Html.LabelFor(model => model.Rate)**

**</div>**

**<div class="editor-field">**

**@Html.EditorFor(model => model.Rate)**

**</div>**

**<div class="editor-label">**

**@Html.LabelFor(model => model.Year)**

**</div>**

**<div class="editor-field">**

**@Html.EditorFor(model => model.Year)**

**</div>**

**<p>**

**<input type="submit" value="Calculate" />**

**</p>**

**</fieldset>**

**}**

**@section Scripts {**

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

**}**

**Step 4**: Create an action method that handles the POST request and processes the data.

In the action method we pass a model as the parameter. That model has UI input field data. Here we do not need to parse and do not need to write extra code.

|  |
| --- |
| [HttpPost]  public ActionResult CalculateSimpleInterestResult(SimpleInterestModel model)  {  decimal simpleInteresrt = (model.Amount\*model.Year\*model.Rate)/100;  StringBuilder sbInterest = new StringBuilder();  sbInterest.Append("<b>Amount :</b> " + model.Amount+"<br/>");  sbInterest.Append("<b>Rate :</b> " + model.Rate + "<br/>");  sbInterest.Append("<b>Time(year) :</b> " + model.Year + "<br/>");  sbInterest.Append("<b>Interest :</b> " + simpleInteresrt);  return Content(sbInterest.ToString());  } |

# View-to-Controller

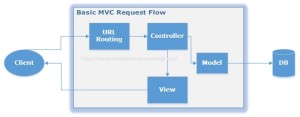
## Traditional Approach

In this approach, we can use the request object of the HttpRequestBase class. This object contains the input field name and values as name-value pairs in case of the form submit.

string strName = Request["txtName"].ToString();

# Controller-to-View

ASP.NET MVC is a framework that facilitates building web applications based on MVC (Model-View-Controller) design pattern. Request coming from client reaches the Controller through URL Rewriting Module. Controller decides which model to use in order to fulfill the request. Further passing the Model data to View which then transforms the Model data and renders response to client as shown in following basic level request flow diagram.

[](http://www.webdevelopmenthelp.net/wp-content/uploads/2014/06/MVC-Request-Flow2.jpg)

**There are three options to pass information from controller to view. They are mentioned below:**

## ViewData:

The ViewData is a Dictionary of objects that are derived from the 'ViewDataDictionary' class and its having keys as string type and a value for respective keys. It contains a null value on each redirection and it needs typecasting for complex data types.

For example: Assign value in controller action like:

ViewData["PersonName"] = "Test Name";

Fetch this ViewData value in View like this:

<h1><%= ViewData["PersonName"] %></h1>

## ViewBag:

ViewBag is just a dynamic wrapper around ViewData and exists only in ASP.NET MVC 3. ViewBag is a dynamic property that takes advantage of the new dynamic features in C# 4.0. It doesn't require typecasting for complex data types. It also contains a null value when redirection occurs.

For example: Assign value in controller action like:

ViewBag.PersonName= "Test Name";

Fetch this ViewData value in View like this:

<h1><%= ViewBag.PersonName %></h1>

**ViewBag Example**

As we discussed earlier that ViewBag and ViewData serves the same purpose but ViewBag is basically a dynamic property (a new C# 4.0 feature) having advantage that it doesn’t have typecasting and null checks.

So, In order to pass data from Controller to View using ViewBag, we will modify our EmployeeController code as follows:

**public class EmployeeController : Controller**

**{**

**// GET: /Employee/**

**public ActionResult Index()**

**{**

**ViewBag.EmployeeName = “Muhammad Hamza”;**

**ViewBag.Company = “Web Development Company”;**

**ViewBag.Address = “Dubai, United Arab Emirates”;**

**return View();**

**}**

**}**

And to get Employee details passed from Controller using ViewBag, View code will be as follows:

**<body>**

**<div>**

**<h1>Employee (ViewBag Data Example)</h1>**

**<div>**

**<b>Employee Name:</b> @ViewBag.EmployeeName<br />**

**<b>Company Name:</b> @ViewBag.Company<br />**

**<b>Address:</b> @ViewBag.Address<br />**

**</div>**

**</div>**

**</body>**

In order to see the above changes in action run the solution, we will find the following output.

[](http://www.webdevelopmenthelp.net/wp-content/uploads/2014/06/ViewBag.jpg)

## TempData:

TempData by default uses the session to store data, so it is nearly same to session only, but it gets cleared out at the end of the next request. It should only be used when the data needs only to persist between two requests. If the data needs to persist longer than that, we should either repopulate the TempData or use the Session directly.

For example:

In Controller : TempData["PersonName"] = "Test Name";

In View:

<h1><%= TempData["PersonName"] %></h1>

# Controller-to-Controller

If it's not private, just pass it as JSON object on the URL using the third parameter in the redirect as given in the below example:

return RedirectToAction("ActionName", "ControllerName", new { userId = id });

If it is private data, we can use TempData - which will be removed at the end of the request after the next request reads it.

Comparison – **View Data, View Bag and TempData to pass data from controller to view and in next request.**

ViewData

1. ViewData is a dictionary object that is derived from ViewDataDictionary class.
2. ViewData is used to pass data from controller to corresponding view.
3. Its life lies only during the current request.
4. If redirection occurs then its value becomes null.
5. It’s required typecasting for complex data type and check for null values to avoid error.

Viewdata is the collection of state that is passed to a view. A view should get all the information it needs from the viewdata. It contains key/value pairs as well as some special properties, such as Model. Viewdata is accessible on the controller as well. The controller is responsible for filling the viewdata dictionary with objects. When a view is executing, it will pull various object from viewdata while it is rendering. If an expected object is not present, then you will see the same type of exception present in any code using a dictionary collection in .Net.

**ViewBag**

1. ViewBag is a dynamic property that takes advantage of the new dynamic features in C# 4.0.
2. Basically it is a wrapper around the ViewData and also used to pass data from controller to corresponding view.
3. Its life also lies only during the current request.
4. If redirection occurs then its value becomes null.
5. It doesn’t required typecasting for complex data type.
6. ViewBag is a property of a Controller class that we generally used to share data between Controller and View and. ViewBag objects is wrapper around ViewData that is used to create dynamic properties for ViewBag.

**TempData**

1. TempData is a dictionary object that is derived from TempDataDictionary class and stored in short lives session.
2. TempData is used to pass data from current request to subsequent request means incase of redirection.
3. Its life is very short and lies only till the target view is fully loaded.
4. It’s required typecasting for complex data type and check for null values to avoid error.
5. It is used to store only one time messages like error messages, validation messages.

TempData is meant to be a very short-lived instance, and you should only use it during the current and the subsequent requests only! Since TempData works this way, you need to know for sure what the next request will be, and redirecting to another view is the only time you can guarantee this. Therefore, the only scenario where using TempData will reliably work is when you are redirecting. This is because a redirect kills the current request (and sends HTTP status code 302 Object Moved to the client), then creates a new request on the server to serve the redirected view.

**ViewData, ViewBag and TempData Current Request Example**

|  |
| --- |
| 1. public class HomeController : Controller 2. { 3. public ActionResult Index() 4. { 5. var emp = new Employee 6. { 7. EmpID=101, 8. Name = "Deepak", 9. Salary = 35000, 10. Address = "Delhi" 11. }; 12. ViewData["emp"] = emp; 13. ViewBag.Employee = emp; 14. TempData["emp"] = emp; 15. return View(); } 16. } 17. @model MyProject.Models.EmpModel; 18. @{ 19. Layout = "~/Views/Shared/\_Layout.cshtml"; 20. ViewBag.Title = "Welcome to Home Page"; 21. var viewDataEmployee = ViewData["emp"] as Employee; //need typcasting 22. var tempDataEmployeet = TempData["emp"] as Employee; //need typcasting 23. } 24. <h2>Welcome to Home Page</h2> 25. <div> 26. <a href="/Employees"> 27. <img src='@Url.Content("\\Content\\Images\\101.jpg")' alt="Best Employee "/> 28. </a> 29. <div> 30. This Year Best Employee is! 31. <h4>@ViewBag.emp.Name</h4> 32. <h3>@viewDataEmployee.Name</h3> 33. <h2>@tempDataEmployee.Name</h2> 34. </div> |

The basic difference between ViewData and ViewBag is that in ViewData instead creating dynamic properties we use properties of Model to transport the Model data in View and in ViewBag we can create dynamic properties without using Model data.

**Overview of Approaches in MVC**

1. ViewModel
2. Partial View
3. ViewBag
4. ViewData
5. TempData
6. Tuple

***An application can use more than one approach based on the requirement at a particular point.*** We need to consider the best one as per the current need.  
Now we will discuss the above approaches, their usages, associated pros and cons in detail.

**ViewModel**

ViewModel is a pattern that allow us to have multiple models as a single class. It aggregates models or contains their properties exactly as per the need of a view. ViewModel should not have methods. It should be a collection of properties needed for a view.

**Typical Usages**

It is the most widely used approach to pass multiple models to View in enterprise applications. It is the standard way you can use when you need to display multiple models in the view (applicable in case of partial view too).

**Advantages**

* ViewModel allows us to render multiple model types in a View as a single model.
* Great intellisense support and compile time error checking on View page.
* ViewModel is good for security purpose also as Views have only what they exactly need. Core domain models are not exposed to user.
* If there is any change in core domain model, you do not need to change anywhere in View code, just you need to modify corresponding ViewModel.
* In this way, ViewModel promotes loose coupling in application.

**Disadvantages**

* ViewModels add another layer between Models and Views so it increases the complexity a little bit. So for small and demo applications, we can use tuple or other ways to keep the things simple for demo.

**Partial View**

A Partial View is a sub-view that you can include in a parent view. In many cases, we have a situation where many Views have shared/common presentation, that common presentation is separated into a Partial View and used in other Views.

**Typical Usages**

This approach is also frequently used in enterprise applications along with ViewModels. It is used where you need to share the same code (Razor and HTML code) in more than one View.

**Advantages**

* It promotes reusability of code (Razor and HTML code) in application.
* It is very helpful for single page applications
* You can use ViewModel approach to Partial View too.
* Using Partial View, you can update a particular area of a View without refreshing the whole page using AJAX.

**Disadvantages**

* If used excessively, then View becomes a just aggregation of Partial Views, so sometime readability is compromised.

**ViewBag**

ViewBag is a dynamic property which comes from ControllerBase class. Internally ViewBag properties are stored as name/value pairs in the dictionary. It takes advantage of the new dynamic features in C# 4.0 so ViewBag doesn’t require typecasting for data types.

**Typical Usages**

ASP.NET MVC 3.0 and latest versions, it is supposed to be used where a property is directly associated to View and do not fit as a model (a model is supposed to be a class encapsulating business data and behaviors). Typical examples of ViewBag are to set the title of a View using ViewBag.Title, or display some message in View as ViewBag.Message, etc.

**Advantages**

* Using ViewBag, we can send data from controller to view with minimal efforts.
* Syntax of ViewBag is better than ViewData. No need to use key.
* ViewBag doesn’t require type casting for data types.

**Disadvantages**

* It is meant for transferring data one-way only that is from controller to view.
* The value of ViewBag persists only during current request. Its value cannot persist between requests so if redirection occurs, then its value becomes null.
* It is certainly a bad practice to overuse ViewBag. It is not recommended in enterprise application, even though sometime it may use to transfer small data.
* No intelligence support and compile-time error checking.
* Technically speaking, it is slow as compared to ViewData but in real world scenarios, the difference is just negligible. (And at first place, micro optimization is bad.)

**ViewData**

ViewData is defined as property (type of ViewDataDictionary class) in ControllerBase class. Values stored in ViewData require typecasting to their datatype in View. The values in ViewData are accessible using a key.

**Advantages**

* Using ViewData, we can send data from controller to view with inbuilt feature using keys.

**Disadvantages**

* It is meant for transferring data one-way only that is from controller to view.
* The value of ViewData persists only during current request. Its value cannot persist between requests so if redirection occurs, then its value becomes null.
* It is certainly a bad practice to overuse ViewData. It is not recommended in enterprise application, even though sometime it may use to show small data.
* Use key syntax, so not as readable as ViewBag which uses property style syntax.
* No intelligence support and compile-time error checking.

**TempData**

TempData is defined as property in ControllerBase class. It is a type of TempDataDictionary class. Values stored in TempData require typecasting to datatype in View. The values in TempData are accessible using a key. It is similar to ViewData but the difference is that it allow us to send and receive the data from one controller to another controller and from one action to another action. It is possible because it internally uses session variables.

**Typical Usages**

Whenever you need to hold some information till subsequent request, it is good to use TempData. It should be used when you need to hold some information like validation messages, error message or some small data which are not having any sensitive information. As it maintains the session to pass value, you should not keep sensitive data in TempData.

**Advantages**

* You can pass value from one action to another action or one controller to another controller.

**Disadvantages**

* It may introduce security risk as described above.
* It requires type casting for data type and check for null values to avoid error.
* No intellisense support in Visual Studio.

**Tuple**

Tuple is a new class introduced in .NET Framework 4.0. It is an ordered sequence, immutable, fixed-size collection of heterogeneous (allows us to group multiple kind of data types) objects.

**Typical Usages**

It may be good for small and demo applications. Tuple is a feature of C# language meant for specific scenarios , but if you are using it in ASP.NET MVC, you should use Tuple only when you do not want to create ViewModel.

**Advantages**

* It provides a way to aggregate models without creating new class (ViewModel)
* A quick remedy and need less coding efforts than ViewModel

**Disadvantages**

* Tuples is fixed size maximum limit of 8 items.
* Value is passes as item1, item2.....It is difficult to identify the arguments just by seeing the code.
* Not a great intellisense support in Visual Studio.

# Using the Standard HTML Helpers

The ASP.NET MVC framework includes a standard set of helpers that you can use to render the most common types of HTML elements. For example, you can use the standard set of helpers to render HTML links and HTML textboxes.

**Rendering Links**

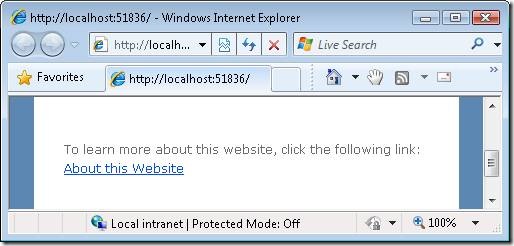
The easiest way to render an HTML link in a view is to use the HTML.ActionLink() helper.

The Html.ActionLink() does not link to a view.

Instead, you use the Html.ActionLink() helper to create a link to a controller action.

For example, the view in Listing 1 includes a link to an action named About (see Figure 1).

Figure 1 – Link rendered by Html.ActionLink() helper



Listing 1 – ViewsHomeAbout.aspx

<%@ Page Language="C#" MasterPageFile="~/Views/Shared/Site.Master" Inherits="System.Web.Mvc.ViewPage" %>

<asp:Content ID="indexContent" ContentPlaceHolderID="MainContent" runat="server">

<p>

To learn more about this website, click the following link:

<%= Html.ActionLink("About this Website", "About" ) %>

</p>

</asp:Content>

In Listing 1, the first parameter passed to the Html.ActionLink() represents the link text and the second parameter represents the name of the controller action.

This Html.ActionLink() helper renders the following HTML:

**<a href=”/Home/About”>About this Website</a>**

*The Html.ActionLink() helper has several overloads and supports several parameters:*

**· linkText** – The label for the link.

**· actionName** – The action that is the target of the link.

**· routeValues** – The set of values passed to the action.

· **controllerName** – The controller that is the target of the link.

· **htmlAttributes** – The set of HTML attributes to add to the link.

· **protocol** – The protocol for the link (for example, https)

· **hostname** – The host name for the link (for example, www.MyWebsite.com)

· **fragment** – The fragment (anchor target) for the link

Notice that you can pass route values from an Html.ActionLink() to a controller action. For example, you might need to pass the Id of a database record that you want to edit. Here’s how you pass an Id parameter to the Edit() action:

[C#]

**<%= Html.ActionLink(“Edit Record”, “Edit”, new {Id=3})**

**When this Html.ActionLink() is rendered to the browser, the following link is created:**

**<a href=”/Home/Edit/3″>Edit Record</a>**

**\*\*\* Begin Note \*\*\***

**Route values are URL encoded automatically. For example, the string “Hello World!” is encoded to “Hello%20World!”.**

**\*\*\* End Note \*\*\***

**Rendering Image Links : Url.Action()**

Unfortunately, you can’t use the Html.ActionLink() helper to render an image link.

Because the Html.ActionLink() helper HTML encodes its link text automatically, you cannot pass an <img> tag to this method and expect the tag to render as an image.

Instead, you need to use the Url.Action() helper to generate the proper link. Here’s how you can generate a delete link with an image:

**Syntax:**

**<a href=”<%= Url.Action(“Delete”) %>”><img src=”../../Content/Delete.png” alt=”Delete” style=”border:0px” /></a>**

The Url.Action() helper supports a set of parameters that are very similar to those supported by the Html.ActionLink() helper.

# Rendering Form Elements

There are several HTML helpers that you can use to render HTML form elements:

1. **BeginForm()**
2. **TextBox()**

**@Html.TextBox("Textbox1", "val")**

**<input id="Textbox1" name="Textbox1" type="text" value="val" />**

**@Html.TextBoxFor(m=>m.Name)**

**Output: <input id="Name" name="Name" type="text" value="Name-val" />**

**Applying Styles**

### CSS Class

1. .inputclass
2. {
3. **width**:100px;
4. **height**:25px;
5. }

**Apply CSS Class to Html Helpers**

Suppose above css class is defined in the external style sheet. Now you want to apply this class to html helpers then we need to add class name by using @class like as :

|  |
| --- |
| @Html.TextBox("Name",**new**{@class="inputclass"})  @Html.TextBoxFor(model => model.Name,**new**{@class="inputclass"}) |

**Apply Inline CSS to Html Helpers**

We can also add inline css to html helpers by using style like as :

|  |
| --- |
| @Html.TextBoxFor(model => model.Name,**new**{ style="width:100px;height:25px"})  @Html.TextBox("Name",**new**{ style="width:100px;height:25px"}) |

1. **TextArea()**

|  |
| --- |
| **@Html.TextArea("Textarea1", "val", 5, 15, null)**  **Output: <textarea cols="15" id="Textarea1" name="Textarea1" rows="5">val</textarea>**  **@Html.TextArea(m=>m.Address , 5, 15, new{}))**  **Output: <textarea cols="15" id="Address" name=" Address " rows="5">Addressvalue</textarea>** |

1. **Password() :**

|  |
| --- |
| **@Html.Password("Password1", "val")**  **Output: <input id="Password1" name="Password1" type="password" value="val" />**  **@Html.PasswordFor(m=>m.Password)**  **Output: <input id="Password" name="Password" type="password"/>** |

1. **HiddenField**

|  |
| --- |
| **@Html.Hidden("Hidden1", "val")**  **Output: <input id="Hidden1" name="Hidden1" type="hidden" value="val" />**  **@Html.HiddenFor(m=>m.UserId)**  **Output: <input id=" UserId" name=" UserId" type="hidden" value="UserId-val" />** |

1. **CheckBox()**

**@Html.CheckBox("chkReading", false) Reading**

|  |
| --- |
| **@Html.CheckBox("Checkbox1", false)**  **Output:< input id="Checkbox1" name="Checkbox1" type="checkbox" value="true" />< input name="myCheckbox" type="hidden" value="false" />**  **@Html.CheckBoxFor(m=>m.IsApproved)**  **Output:< input id="Checkbox1" name="Checkbox1" type="checkbox" value="true" />< input name="myCheckbox" type="hidden" value="false" />** |

1. **RadioButton()**

|  |
| --- |
| **@Html.RadioButton("Radiobutton1", "val", true)**  **Output: <input checked="checked" id="Radiobutton1" name="Radiobutton1" type="radio" value="val" />**  **@Html.RadioButtonFor(m=>m.IsApproved, "val")**  **Output: <input checked="checked" id="Radiobutton1" name="Radiobutton1" type="radio" value="val" />** |

1. **DropDownList()**

|  |
| --- |
| **@Html.DropDownList (“DropDownList1”, new SelectList(new [] {"Male", "Female"}))**  **Output:< select id="DropDownList1" name="DropDownList1">< option>M</option>< option>F</option>< /select>**  **@Html.DropDownListFor(m => m.Gender, new SelectList(new [] {"Male", "Female"}))**  **Output:< select id="Gender" name="Gender">< option>Male</option>< option>Female</option>< /select>** |

1. **EndForm()**
2. **ListBox()**

## Example: Using Controls in MVC View Pages.

**The view renders an HTML page with a simple user registration form (see Figure 2).**

Listing 2 – ViewsCustomerRegister.aspx [C#]

**<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master" Inherits="System.Web.Mvc.ViewPage<MvcApplication1.Models.Customer>" %>**

**<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">**

**<%= Html.ValidationSummary("Create was unsuccessful. Please correct the errors and try again.") %>**

**<% using (Html.BeginForm()) {%>**

**<fieldset>**

**<legend>Register</legend>**

**<p>**

**<label for="FirstName">First Name:</label>**

**<%= Html.TextBox("FirstName") %>**

**<%= Html.ValidationMessage("FirstName", "\*") %>**

**</p>**

**<p>**

**<label for="LastName">Last Name:</label>**

**<%= Html.TextBox("LastName") %>**

**<%= Html.ValidationMessage("LastName", "\*") %>**

**</p>**

**<p>**

**<label for="Password">Password:</label>**

**<%= Html.Password("Password") %>**

**<%= Html.ValidationMessage("Password", "\*") %>**

**</p>**

**<p>**

**<label for="Password">Confirm Password:</label>**

**<%= Html.Password("ConfirmPassword") %>**

**<%= Html.ValidationMessage("ConfirmPassword", "\*") %>**

**</p>**

**<p>**

**<label for="Profile">Profile:</label>**

**<%= Html.TextArea("Profile", new {cols=60, rows=10})%>**

**</p>**

**<p>**

**<%= Html.CheckBox("ReceiveNewsletter") %>**

**<label for="ReceiveNewsletter" style="display:inline">Receive Newsletter?</label>**

**</p>**

**<p>**

**<input type="submit" value="Register" />**

**</p>**

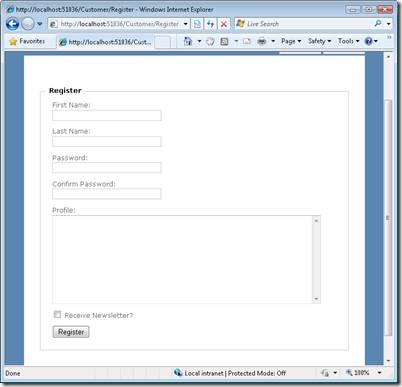
**</fieldset>**

**<% } %>**

**</asp:Content>**

It is worth emphasizing, once again, that these form helpers are simply rendering strings. For example, the Html.TextBox() helper renders a string that includes an “<input>” tag. If you prefer, you could create the view in Listing 2 without using any of these helpers.

Figure 2 – The Register page



# Rendering a Form

[C#]

<% using (Html.BeginForm()) {%>

**… Form Contents …**

<% } %>

The advantage of opening and closing a <form> tag with a using statement is that you won’t accidently forget to close the <form> tag.

**However, if you find this syntax confusing or otherwise disagreeable then you don’t need to use a using statement.**

Instead, you can open the <form> tag with Html.BeginForm() and close the <form> with HTml.EndForm() like this:

[C#]

<% Html.BeginForm(); %>

… Form Contents …

<% Html.EndForm (); %>

**By default, the Html.BeginForm() method renders a form that posts back to the same controller action.**

**In other words, if you retrieved the view by invoking the Customer controller Details () action, then the Html.BeginForm () renders a <form> tag that looks like:**

<form action="/Customer/Details" method="post">

</form>

**If you want to post to another action, or modify any other property of the <form> tag, then you can call the Html.BeginForm() helper with one or more parameters.**

**The Html.BeginForm() helper accepts the following parameters:**

* **routeValues — The set of values passed to the action.**
* **actionName – The action that is the target of the form post.**
* **controllerName – The controller that is the target of the form post.**
* **method – The HTTP method of the form post. The possible values are restricted to POST and GET (You can’t use other HTTP methods within HTML, you must use JavaScript).**
* **htmlAttributes – The set of HTML attributes to add to the form.**

**In HTML Helper is just a method that returns a HTML string. The string can represent any type of content that you want. For example, you can use HTML Helpers to render standard HTML tags like HTML <input>, <button> and <img> tags etc.**

**You can also create your own HTML Helpers to render more complex content such as a menu strip or an HTML table for displaying database data.**

## Different types of HTML Helpers

There are three types of HTML helpers as given below:

1. Inline Html Helpers

These are create in the same view by using the Razor @helper tag. These helpers can be reused only on the same view.

1. @helper ListingItems(string[] items)

2. {

3. <ol>

4. @foreach (string item in items)

5. {

6. <li>@item</li>

7. }

8. </ol>

9. }

10.

11. <h3>Programming Languages:</h3>

12.

13. @ListingItems(new string[] { "C", "C++", "C#" })

14.

15. <h3>Book List:</h3>

16.

17. @ListingItems(new string[] { "How to C", "how to C++", "how to C#" })

1. Built-In Html Helpers

Built-In Html Helpers are extension methods on the HtmlHelper class.

# Validation Rules

**Adding Validation Rules to the Model**

Add a using statement at the top of the file that references the[System.ComponentModel.DataAnnotations](http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.aspx) namespace:

Notice the namespace does not contain System.Web. DataAnnotations provides a built-in set of validation attributes that you can apply declaratively to any class or property.

Now update the Model class to take advantage of the built-in [Required](http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.requiredattribute.aspx), [StringLength](http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.stringlengthattribute.aspx), and [Range](http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.rangeattribute.aspx) validation attributes. Use the following code as an example of where to apply the attributes.

Example of Movie class

publicclassMovie {  
    publicint ID { get; set; }  
  
    [Required]  
    publicstringTitle { get; set; }  
  
    [DataType(DataType.Date)]  
    publicDateTimeReleaseDate { get; set; }  
  
    [Required]  
    publicstringGenre { get; set; }  
  
    [Range(1, 100)]  
    [DataType(DataType.Currency)]  
    publicdecimalPrice { get; set; }  
  
    [StringLength(5)]  
    publicstringRating { get; set; }  
}

Run the application and you will again get the following run time error:

MovieDBContext db = newMovieDBContext();  
  
Movie movie = newMovie();  
movie.Title = "Gone with the Wind";  
movie.Price = 0.0M;  
  
db.Movies.Add(movie);    
db.SaveChanges();        // <= Will throw server side validation exception

Having validation rules automatically enforced by the .NET Framework helps make your application more robust. It also ensures that you can't forget to validate something and inadvertently let bad data into the database.

Lab Example

Here's a complete code listing for the updated *Movie.cs* file:

usingSystem.ComponentModel.DataAnnotations;  
  
namespaceMvcMovie.Models {  
    publicclassMovie {  
        publicint ID { get; set; }  
  
        [Required]  
        publicstringTitle { get; set; }  
  
        [DataType(DataType.Date)]  
        publicDateTimeReleaseDate { get; set; }  
  
        [Required]  
        publicstringGenre { get; set; }  
  
        **[Range(1, 100)]** [DataType(DataType.Currency)]  
        publicdecimalPrice { get; set; }  
  
        [StringLength(5)]  
        publicstringRating { get; set; }  
    }  
  
    publicclassMovieDBContext : DbContext {  
        publicDbSet<Movie>Movies { get; set; }  
    }  
}

**Model with Validation Implementation**

publicclassRegister

{

[Required(ErrorMessage="FirstName Required:")]

[RegularExpression(@"^[a-zA-Z'.\s]{1,40}$",ErrorMessage="Special Characters not allowed")]

[StringLength(50, ErrorMessage = "Less than 50 characters")]

publicstring FirstName { get; set; }

[Required(ErrorMessage="LastName Required:")]

[RegularExpression(@"^[a-zA-Z'.\s]{1,40}$", ErrorMessage = "Special Characters not allowed")]

[StringLength(50, ErrorMessage = "Less than 50 characters")]

publicstring LastName { get; set; }

[Required(ErrorMessage="EmailId Required:")]

[RegularExpression(@"^\w+([-+.']\w+)\*@\w+([-.]\w+)\*\.\w+([-.]\w+)\*$", ErrorMessage = "Email Format is wrong")]

[StringLength(50, ErrorMessage = "Less than 50 characters")]

publicstring EmailId { get; set; }

[Required(ErrorMessage="Password Required:")]

[DataType(DataType.Password)]

[StringLength(30, ErrorMessage = "Less than 30 characters")]

publicstring Password { get; set; }

[Required(ErrorMessage="Confirm Password Required:")]

[DataType(DataType.Password)]

[Compare("Password", ErrorMessage = "Confirm not matched.")]

[StringLength(30, ErrorMessage = "Less than 30 characters")]

publicstring ConfirmPassword { get; set; }

[Required(ErrorMessage="Street Address Required")]

[StringLength(100, ErrorMessage = "Less than 100 characters")]

publicstring StreetAdd1 { get; set; }

[StringLength(100, ErrorMessage = "Less than 100 characters")]

publicstring StreetAdd2 { get; set; }

[Required(ErrorMessage="City Required")]

[RegularExpression(@"^[a-zA-Z'.\s]{1,40}$", ErrorMessage = "Special Characters not allowed")]

[StringLength(50, ErrorMessage = "Less than 50 characters")]

publicstring City { get; set; }

[Required(ErrorMessage="State Required")]

[RegularExpression(@"^[a-zA-Z'.\s]{1,40}$", ErrorMessage = "Special Characters not allowed")]

[StringLength(50, ErrorMessage = "Less than 50 characters")]

publicstring State { get; set; }

[Required(ErrorMessage="ZipCode Required")]

[StringLength(20, ErrorMessage = "Less than 20 characters")]

publicstring Zip { get; set; }

# MVC CLIENT SIDE VALIDATIONS

When you are developing an MVC application in Visual Studio 2012 then the client-side becomes enabled by default, but you can easily enable or disable the writing of the following app setting code snippet in the web.config file.

<configuration>

<appSettings>

<addkey="ClientValidationEnabled"value="true"/>

<addkey="UnobtrusiveJavaScriptEnabled"value="true"/>

</appSettings>

</configuration>

The jQuery validation plug-in takes advantage of the Data Annotation attributes defined in the model, which means that you need to do very little to start using it. Let's create an Employee model (*Employee.cs* class file under the Models folder) that has two properties with Data annotation attributes.

using System.ComponentModel.DataAnnotations;

namespace ClientValidation.Models

{

publicclass Employee

{

[Required(ErrorMessage = "Name is Requirde")]

publicstring Name { get; set; }

[Required(ErrorMessage = "Email is Requirde")]

[RegularExpression(@"^([a-zA-Z0-9\_\-\.]+)@((\[[0-9]{1,3}" +

@"\.[0-9]{1,3}\.[0-9]{1,3}\.)|(([a-zA-Z0-9\-]+\" +

@".)+))([a-zA-Z]{2,4}|[0-9]{1,3})(\]?)$",

ErrorMessage = "Email is not valid")]

publicstring Email { get; set; }

}

}

After that you need to create the controller's action methods. These render views on the UI and bind a model with the view. So let's create a controller with two action methods that handle both request types (GET and POST respectively).

using System.Web.Mvc;

using ClientValidation.Models;

namespace ClientValidation.Controllers

{

publicclass EmployeeController : Controller

{

public ActionResult Index()

{

return View();

}

[AcceptVerbs(HttpVerbs.Post)]

public ActionResult Index(Employee model)

{

if (ModelState.IsValid)

{

ViewBag.Name = model.Name;

ViewBag.Email = model.Email;

}

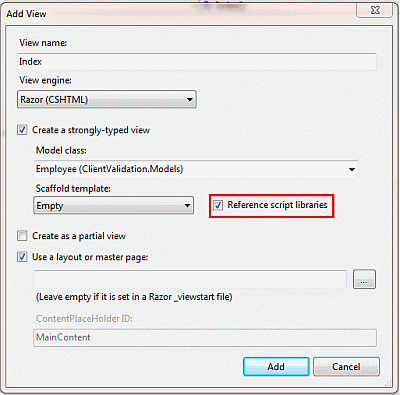
return View(model);

}

}

}

Thereafter create a strong view that the input field binds with the model and create a JavaScript function that uses the HTML Helper extension method FieldIdFor() to get the Id of the input field and shows the input field's value in an alert message.



***Figure 1.1 : Add New View Screen***

Here I checked "Reference script libraries," which means Visual Studio adds these references automatically. Visual Studio adds the following code snippet to the bottom of the view.

@section Scripts

{

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

}

Now click on the "Add" button and the view is created as in the following code snippet:

@model ClientValidation.Models.Employee

@{

ViewBag.Title = "Index";

}

@if (ViewData.ModelState.IsValid)

{

if(@ViewBag.Name != null)

{

**Name : @ViewBag.Name**

**Email : @ViewBag.Email**

}

}

@using (Html.BeginForm()) {

@Html.ValidationSummary(true)

<fieldset>

<legend>Employee</legend>

<divclass="editor-label">

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

</div>

<divclass="editor-field">

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

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

</div>

<divclass="editor-label">

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

</div>

<divclass="editor-field">

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

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

</div>

<p>

<inputtype="submit"value="Create"/>

</p>

</fieldset>

}

@section Scripts

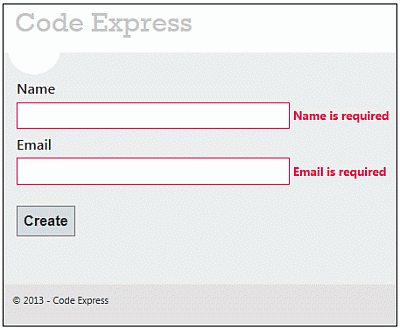
{

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

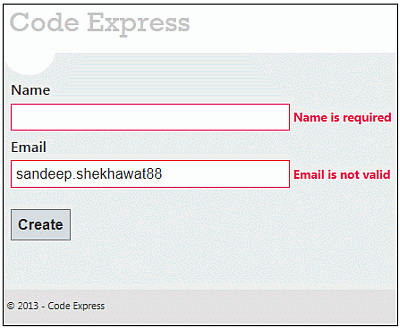
}

Output of Application

1. When all fields are empty:

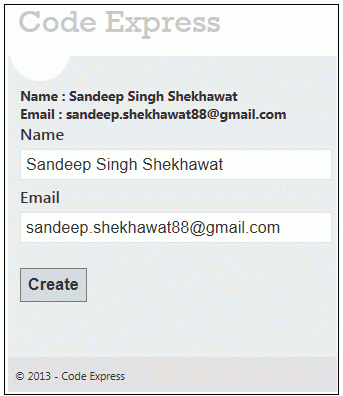


1. When the Name field is empty but Email is not valid:



***Figure 1.3 : Validation Message when Email is not valid***

1. When both fields are valid:



**MVC SERVER SIDE VALIDATIONS**

ASP.NET MVC server-side validation using the Data Annotation API. The ASP.NET MVC Framework validates any data passed to the controller action that is executing. It populates a ModelState object with any validation failures that it finds and passes that object to the controller. Then the controller actions can query the ModelState to discover whether the request is valid and react accordingly.

One is to manually add an error to the ModelState object and another uses the Data Annotation API to validate the model data.

**Approach 1: Manually Add Error to ModelState Object**

I create a User class under the *Models* folder. The User class has two properties "Name" and "Email". The "Name" field has required field validations while the "Email" field has Email validation. So let's see the procedure to implement the validation. Create the User Model as in the following:

namespace ServerValidation.Models

{

publicclass User

{

publicstring Name { get; set; }

publicstring Email { get; set; }

}

}

After that, I create a controller action in User Controller (*UserController.cs* under *Controllers* folder). That action method has logic for the required validation for Name and Email validation on the Email field. I add an error message on ModelState with a key and that message will be shown on the view whenever the data is not to be validated in the model.

using System.Text.RegularExpressions;

namespace ServerValidation.Controllers

{

publicclass UserController : Controller

{

public ActionResult Index()

{

return View();

}

[HttpPost]

public ActionResult Index(ServerValidation.Models.User model)

{

if (string.IsNullOrEmpty(model.Name))

{

ModelState.AddModelError("Name", "Name is required");

}

if (!string.IsNullOrEmpty(model.Email))

{

string emailRegex = @"^([a-zA-Z0-9\_\-\.]+)@((\[[0-9]{1,3}" +

@"\.[0-9]{1,3}\.[0-9]{1,3}\.)|(([a-zA-Z0-9\-]+\" +

@".)+))([a-zA-Z]{2,4}|[0-9]{1,3})(\]?)$";

Regex re = new Regex(emailRegex);

if (!re.IsMatch(model.Email))

{

ModelState.AddModelError("Email", "Email is not valid");

}

}

else

{

ModelState.AddModelError("Email", "Email is required");

}

if (ModelState.IsValid)

{

ViewBag.Name = model.Name;

ViewBag.Email = model.Email;

}

return View(model);

}

}

}

Thereafter, I create a view (*Index.cshtml*) for the user input under the *User* folder.

@model ServerValidation.Models.User

@{

ViewBag.Title = "Index";

}

@using (Html.BeginForm()) {

if (@ViewData.ModelState.IsValid)

{

if(@ViewBag.Name != null)

{

<b>

Name : @ViewBag.Name<br/>

Email : @ViewBag.Email

</b>

}

}

<fieldset>

<legend>User</legend>

<divclass="editor-label">

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

</div>

<divclass="editor-field">

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

@if(!ViewData.ModelState.IsValid)

{

<spanclass="field-validation-error">

@ViewData.ModelState["Name"].Errors[0].ErrorMessage</span>

}

</div>

<divclass="editor-label">

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

</div>

<divclass="editor-field">

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

@if (!ViewData.ModelState.IsValid)

{

<spanclass="field-validation-error">

@ViewData.ModelState["Email"].Errors[0].ErrorMessage</span>

}

</div>

<p>

<inputtype="submit"value="Create"/>

</p>

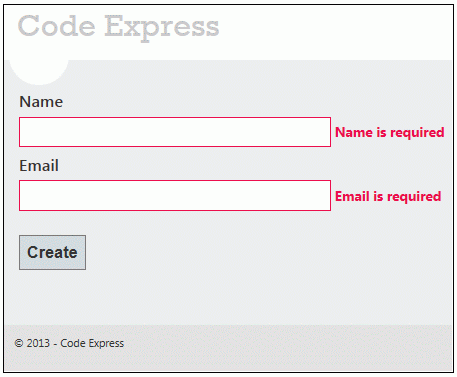
</fieldset>

}

**Run the Application and Test in Various Ways**

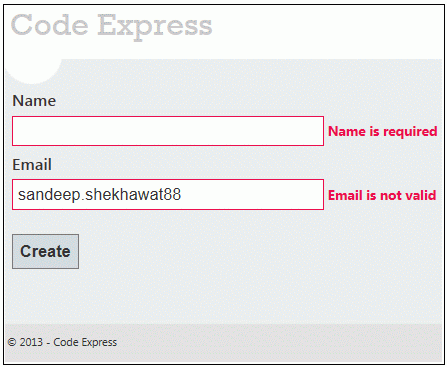
We are going to test various scenarios for testing the validation. Let’s look at each one by one.

1. When all fields are empty:

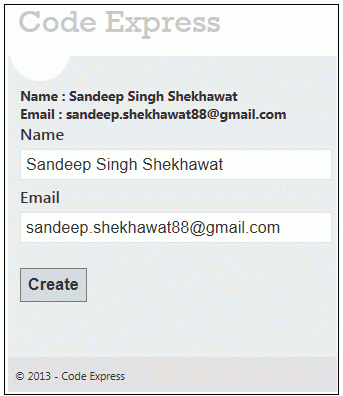


***Figure 1.1: Validation Message when both fields are empty***

1. When the Name field is empty, but Email is not valid:



1. When both fields are valid:



**Approach 2: Specifying Business Rules with Data Annotation**

While the first approach works quite well, it does tend to break the application's separation of concerns. Namely, the controller should not contain business logic such as, the business logic belongs in the model.

Microsoft provides an effective and easy-to-use data validation API called Data Annotation in the core .NET Framework. It provides a set of attributes that we can apply to the data object class properties. These attributes offer a very declarative way to apply validation rules directly to a model.

First, create a model named Student (*Student.cs*) under the *Models* folder and applies Data Annotation attributes on the properties of the Student class.

using System.ComponentModel.DataAnnotations;

namespace ServerValidation.Models

{

publicclass Student

{

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

publicstring Name { get; set; }

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

[RegularExpression(@"^([a-zA-Z0-9\_\-\.]+)@((\[[0-9]{1,3}" +

@"\.[0-9]{1,3}\.[0-9]{1,3}\.)|(([a-zA-Z0-9\-]+\" +

@".)+))([a-zA-Z]{2,4}|[0-9]{1,3})(\]?)$",

ErrorMessage="Email is not valid")]

publicstring Email { get; set; }

}

}

Now, create an action method in the controller (StudentController class under the *Controllers* folder) that returns a view with a model after the post request.

namespace ServerValidation.Controllers

{

publicclass StudentController : Controller

{

public ActionResult Index()

{

return View();

}

[AcceptVerbs(HttpVerbs.Post)]

public ActionResult Index(Student model)

{

if (ModelState.IsValid)

{

ViewBag.Name = model.Name;

ViewBag.Email = model.Email;

}

return View(model);

}

}

}

After that, I created a view (*Index.cshtml*) to get student details and show an error message if the model data is not valid.

@model ServerValidation.Models.Student

@{

ViewBag.Title = "Index";

}

@if (ViewData.ModelState.IsValid)

{

if(@ViewBag.Name != null)

{

<b>

Name : @ViewBag.Name<br/>

Email : @ViewBag.Email

</b>

}

}

@using (Html.BeginForm()) {

@Html.ValidationSummary(true)

<fieldset>

<legend>Student</legend>

<divclass="editor-label">

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

</div>

<divclass="editor-field">

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

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

</div>

<divclass="editor-label">

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

</div>

<divclass="editor-field">

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

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

</div>

<p>

<inputtype="submit"value="Create"/>

</p>

</fieldset>

}

#### Using Data Annotations

The simplest way to implement validation is to put annotation attributes in the model class. As an example, I will put the [Required] attribute in the Name property of the user.

publicpartialclass User

{

publicint UserID { get; set; }

[Required]

publicstring Name { get; set; }

publicstring Address { get; set; }

publicstring Town { get; set; }

publicstring County { get; set; }

publicstring Country { get; set; }

publicstring Email { get; set; }

public DateTime DoB { get; set; }

publicbool IsActive { get; set; }

publicstring UserName { get; set; }

publicstring Password { get; set; }

publicint Rating { get; set; }

}

If the user does not populate the name of the user, UpdateModel will throw an exception, and the required field message will be shown in the view.

#### Using a MetaData class

Sometimes you are not able to modify a model class, especially if it is automatically generated. In that case you can create a MetaData class for the model class, where you will place validation rules. An example is shown in the following listing:

**[MetadataType(typeof(UserMetadata))]**

publicpartialclass User

{

publicclass UserMetadata

{

[StringLength(50), Required]

publicobject Name { get; set; }

[StringLength(5)]

publicobject Email { get; set; }

[Range(0, 10)]

publicobject Rating { get; set; }

}

}

In this case, in the model class is defined that UserMetaData will be the meta data class of the User class where validation rules will be placed. In the metadata class you can put properties named the same way as properties of the model and put annotations there. This is a useful way to implement validation even if you can modify your original model classes but you do not want to mix validation rules and properties. The only prerequisite is that the model class is defined as partial so we can put the second part of the class in a separate file.

#### Custom validation

The third option is to put custom validation rules directly in the controller code. An example of explicit setting of error messages in the controller is shown in the following listing:

ModelState.AddModelError("Country", "This is useless message attached to the 'Country' input");

The method AddModelError takes the ID of the input element where the error label should be shown and the text of the error message. This code is placed in the action method of the controller and it puts a new error that will be associated to the Country input.

Custom HTML Helpers

**Example refer MVC\_Assignments\CreatingHtmlHelperExample\MvcCustomHelperExp**

Custom HTML helpers than that can also be done very easily.

Ways of creating custom helpers.

1. Creating a static method [ static Class, static methods
2. Writing an extension method
3. Using the @helper(razor only) [ Inside the cshtml file]

Let us try to create a simple HTML helper method which will create a marked HTML label.

**Creating a static method**

In this approach we can simply create a static class with static method which will return the HTML string to be used at the time of rendering.

|  |
| --- |
| namespace HTMLHelpersDemo.Helpers  {  public static class MyHTMLHelpers  {  public static IHtmlString LabelWithMark(string content)  {  string htmlString = String.Format("<label><mark>{0}</mark></label>", content);  return new HtmlString(htmlString);  }  }  }  @MyHTMLHelpers.LabelWithMark(“suresh”)  @Html.TextBox(“Name”); |

**Writing an extension method**

In this approach we will write a simple extension method for the built in html helper class. This will enable us to use the same Html object to use our custom helper method.

**public static class MyExtensionMethods**

**{**

**public static IHtmlString LabelWithMark(this HtmlHelper helper, string content)**

**{**

**string htmlString = String.Format("<label><mark>{0}</mark></label>", content);**

**return new HtmlString(htmlString);**

**}**

**}**

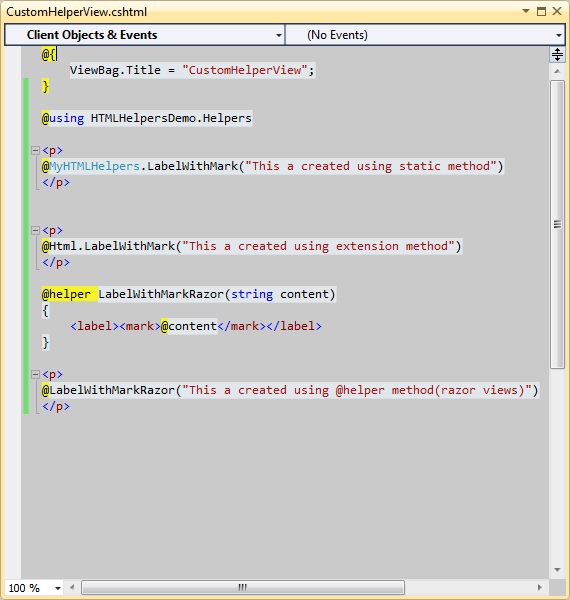
**Using the @helper(razor only)**

This method is pretty specific to razor view engine. Let us see how this can be done. We need to write this method in the view itself.

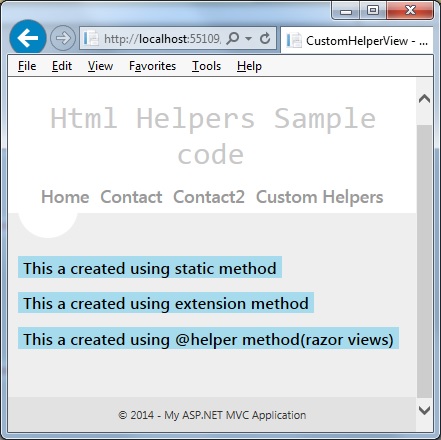
|  |
| --- |
| @helper LabelWithMarkRazor(string content)  {  <label><mark>@content</mark></label>  } |

**Note:** By default these helpers will be available in the view they are defined in. If we want to use the razor created helpers in multiple views then we have to put all of them in one view and put that view file in App\_Code directory. This way they will be usable from all the views.

Now we have the same html helper method written in 3 different ways, Let us create a simple view and try to use these helper methods.



Lets run the application and see the result:



# Layouts

Suppose you are developing an ASP.NET web application and you want to maintain a consistent look and feel across all the pages within your web application.

You then have two options,

the first one is to design the head, body and footer sections on each page. In this approach, you need to write more code on each page so ASP.NET 2.0 introduced "Master Pages" that helps enable this when using *.aspx* based pages or templates.

It is your second option. Razor also supports this concept with a feature called "layouts" that allow you to define a common site template, and then inherit its look and feel across all the views/pages on your web application.

**Create an MVC Application**

I will create an MVC application using Visual Studio 2012.

**Step 1**: Go to "File" -> "New" -> "Project...".

**Step 2**: Choose "ASP.NET MVC 4 Web Application" from the list, then give the application name "LayoutMvcApplication" and set the path in the location input where you want to create the application.

**Step 3**: Now choose the Project Template "Empty" and select "Razor" as the view engine from the dropdown list.

**Create Layout for Application**

A layout is used to provide a consistent look and feel of all pages of a web application so we create a layout for the web application. Let's see the procedure for that.

**Step 1**: Create a "*Content*" folder in the root directory of the web application.

**Step 2**: Create a Style Sheet "*Site.css*" under the *content* folder. That CSS file contains all CSS classes necessary for a consistent web application page design.

**Step 3**: Create a "*Shared*" folder under the "*View*" folder.

**Step 4:** Create a "*\_Layout.cshtml*" file under the "*Shared*" folder. The file "*\_Layout.cshtml*" represents the layout of each page in the application. Right-click on the *Shared* folder in Solution Explorer, then go to "Add" item and click on "View".

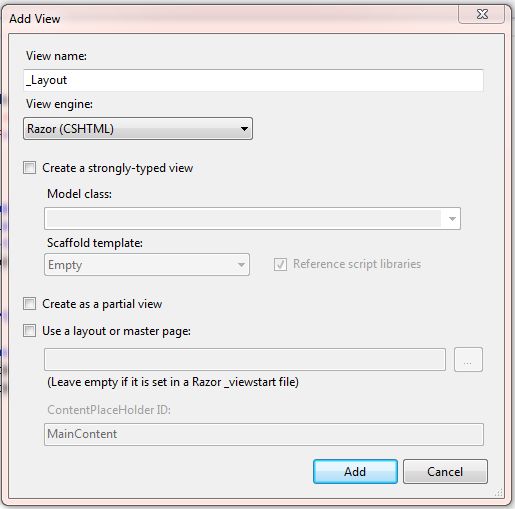


Figure 1.1 Create a layout for application

Now the View has been created. Write the following code in the "*\_Layout.cshtml*" file.

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="utf-8" />  <title>@ViewBag.Title - Code Express</title>  <link href="~/favicon.ico"  rel="shortcut icon" type="image/x-icon" />  <link rel="stylesheet"  href="@Url.Content("~/Content/Site.css")" />  </head>  <body>  <header>  <div class="content-wrapper">  <div class="float-left">  <p class="site-title">  @Html.ActionLink("Code Express",  "Index", "Home")  </p>  </div>  <div class="float-right">  <nav>  <ul id="menu">  <li>@Html.ActionLink("Home", "Index", "Home")</li>  <li>@Html.ActionLink("About", "About", "Home")</li>  </ul>  </nav>  </div>  </div>  </header>  <div id="body">  **@RenderSection("featured", required: false)**  <section class="content-wrapper main-content clear-fix">  **@RenderBody()**  </section>  </div>  <footer>  <div class="content-wrapper">  <div class="float-left">  <p>&copy; @DateTime.Now.Year&acirc;&euro;&ldquo; Code Express</p>  </div>  </div>  </footer>  </body>  </html> |

In this layout, we are using an HTML helper method and some other system defined methods so let's see these methods one by one.

* **RenderBody(): In layout pages, renders the portion of a content page that is not within a named section. It returns the HTML content to render. RenderBody is required, since it's what renders each view.**
* **RenderSection(): RenderSection() is a method of the WebPageBase class. 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 (that can make it easier to track down content errors). It returns the HTML content to render.**

**The \_ViewStart File**

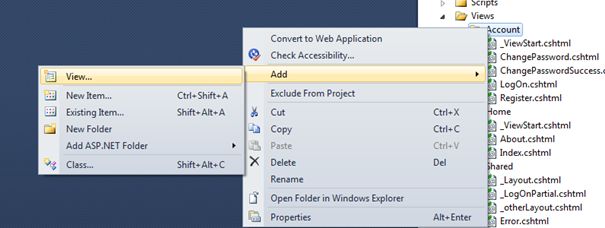
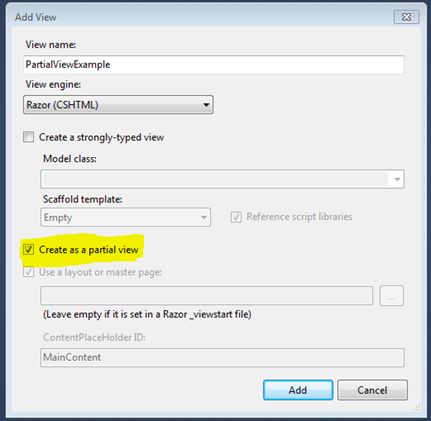
The "*\_ViewStart*" file in the *Views* folder contains the following content:

@{

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

}

**Partial View**  
Partial view is special view which renders a portion of view content.

It is just like a user control web form application. Partial can be reusable in multiple views. It helps us to reduce code duplication. In other word a partial view enables us to render a view within the parent view.  
   
The partial view is instantiated with its own copy of a ViewDataDictionary object which is available with the parent view so that partial view can access the data of the parent view. If we made the change in this data (ViewDataDictionary object), the parent view's data is not affected. Generally the Partial rendering method of the view is used when related data that we want to render in a partial view is part of our model.  
   
**Creating Partial View  
  
To create a partial view, right-click on view -> shared folder and select Add -> View option. In this way we can add a partial view.**  
  
  
   
It is not mandatory to create a partial view in a shared folder but a partial view is mostly used as a reusable component, it is a good practice to put it in the "shared" folder.  
  


Contoller – ActionName ->PartialView

Folder -> Add PartialView

HTML helper has two methods for rendering the partial view: Partial and RenderPartial.

1. <div>
2. **@Html.Partial("PartialViewExample",@model)   //NormalView**
3. </div>
4. <div>
5. @{
6. **Html.RenderPartial("PartialViewExample",@model);   //NormalView**
7. }
8. </div>

**@Html.RenderPartial**  
The result of the RenderPartial method is written directly into the HTTP response, it means that this method used the same TextWriter object as used by the current view. This method returns nothing.  
**@Html.Partial**  
This method renders the view as an HTML-encoded string. We can store the method result in a string variable.  
   
The Html.RenderPartial method writes output directly to the HTTP response stream so it is slightly faster than the Html.Partial method.  
   
Returning a Partial view from the Controller's Action method:

1. public ActionResult PartialViewExample()
2. {
3. return PartialView();
4. }

**Render Partial View Using jQuery**  
Sometimes we need to load a partial view within a model popup at runtime, in this case we can render the partial view using JQuery element's load method.

1. <script type="text/jscript">
2. $('#partialView').load('/shared/PartialViewExample’);
3. </script>

**View Vs Partial View**

|  |  |
| --- | --- |
| **View** | **Partial View** |
| View contains the layout page | Partial view does not contain the layout page |
| \_viewstart page is rendered before any view is rendered | Partial view does not check for a \_viewstart.cshtml. We cannot place any common code for a partial view within the \_viewStart.cshtml page. |
| View may have markup tags like html, body, head, title, meta etc. | The Partial view is specially designed to render within the view and as a result it does not contain any mark up. |
| Partial view is more lightweight than the view.  We can also pass a regular view to the RenderPartial method. | |
| If there is no layout page specified in the view, it can be considered as a partial view. In razor, there is no distinction between views and partial views as in the ASPX view engine (aspx and ascx). | |

# Areas

Areas are functionally independent modules of your ASP.NET MVC application that mimic the folder structure and conventions of MVC. Consider the following scenario:

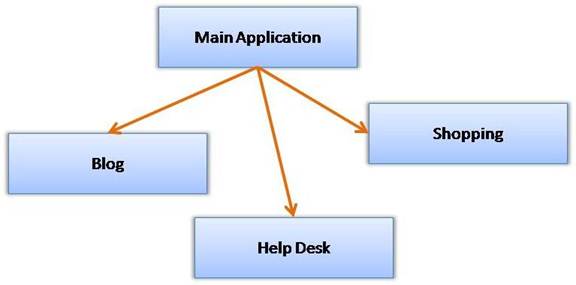


Figure 1: Areas are independent modules of your ASP.NET MVC application

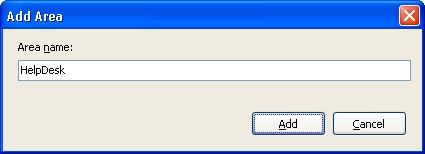
As you can see the main application has three functionally independent modules viz. Blog, Help Desk and Shopping. Without any areas in place, you would have put all the controllers and views in the respective folder of the main application folder. Obviously, you cannot have the same controller name for different modules. For example, you cannot have HomeController for Blog and HomeController for HelpDesk. You either need to put all action methods of both the modules in a single controller or create two controllers with altogether different names (say BlogHomeController and HelpDeskHomeController).

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).

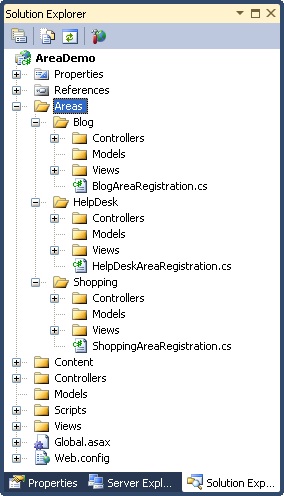
**Adding a New Area**

Before you work with areas, create a new ASP.NET MVC application with ASPX view engine.

To add a new area right click on the application in the Solution Explorer and select the Add > Area shortcut menu option. Doing so will open "Add Area" dialog as shown below:

  
Figure 2: Adding a New Area

You need to enter the Area Name and click the Add button to create the folder structure for that area. The following figure shows the complete folder structure with three Areas - Blog, HelpDesk and Shopping - added.

  
Figure 3: Solution Explorer

As you can see, the main application has Areas subfolders. Inside the Areas subfolders there is a folder for each area you wish to create (Blog, HelpDesk and Shpping in the above example). Each Areaa subfolder has its own set of MVC folders (Models, Views and Controllers). You can treat each area just like the main application and add models, views and controllers in the respective folders.

**Registering Areas with MVC Framework**

**Simply creating area folder structure is not sufficient, you also need to inform ASP.NET MVC framework about the areas you've added. Luckily, this task is automatically done when you add a new area.**

**Notice the area registration class file that was created under each area folder (BlogAreaRegistration.cs, HelpDeskAreaRegistration.cs, etc.).**

**The area registration file contains a class that inherits from AreaRegistration base class.**

The HelpDeskAreaRegistration class is shown below :

1. public class HelpDeskAreaRegistration : AreaRegistration
2. {
3. public override string AreaName
4. {
5. get
6. {
7. return "HelpDesk";
8. }
9. }
11. public override void RegisterArea(AreaRegistrationContext context)
12. {
13. context.MapRoute(
14. "HelpDesk\_default",
15. "HelpDesk/{controller}/{action}/{id}",
16. new { action = "Index", id = UrlParameter.Optional }
17. );
18. }
19. }

As you can see the HelpDeskAreaRegistration class overrides AreaName property and the RegisterArea method. The RegisterArea method registers the route information for the newly added area with MVC framework.

You will have one area registration class file for each area. When are the area registration classes used? If you open Global.asax file, you will have the following code in the Application\_Start event handler :

1. protected void Application\_Start()
2. {
3. **AreaRegistration.RegisterAllAreas();**
4. RegisterGlobalFilters(GlobalFilters.Filters);
5. RegisterRoutes(RouteTable.Routes);
6. }

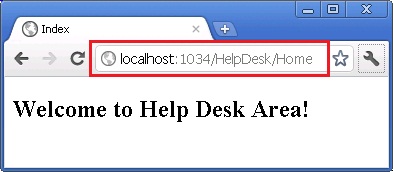
As you can see, there is a call to RegisterAllAreas() static method of AreaRegistration class. Calling RegisterAllAreas() method in turn invokes the RegisterArea() method of all the area registration classes found in the application.

Before going ahead, add HomeController to all the areas and the main application, i.e. four controller classes with name HomeController. All the HomeController classes will look like this :

1. public class HomeController : Controller
2. {
3. public ActionResult Index()
4. {
5. return View();
6. }
7. }

Then right click on each Index() action method and add Index view, i.e. total four Index views in their respective folders.

Run the application and try navigating to different areas. The following figure shows a sample run for HelpDesk area. Notice the URL of the area.



http://localhost/Application/Area/ControllerName/Action

**Linking Between Two 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:

1. <h3>You are inside main area. To navigate to some other area please click on the following links</h3>
2. <%= Html.ActionLink("Main Area", "Index", "Home", **new { area = "" }**, null)%>
3. <br /><br />
4. <%=

**Html.ActionLink("Blog Area", "Index", "Home", new { area = "Blog",id=1 }, null**)

**Html.ActionLink("Blog Area", "Index", "Home", new { area = "Blog",id=1 }, null**)

1. %>
2. <br /><br />
3. <%= Html.ActionLink("Help Desk Area", "Index", "Home", **new { area = "HelpDesk" }**, null)%>
4. <br /><br />
5. <%= Html.ActionLink("Shopping Area", "Index", "Home", **new { area = "Shopping" }**, null)%>

The above fragment is from the Index view of the main area. It renders four hyperlinks using ActionLink(). Notice the fourth parameter that specifies area name. This parameter governs the area whose action method is being linked to.

**Redirecting to an Action Belonging to Another Area**

You can also redirect to some action method of a specific area using RedirectToAction() method. Consider the following action method:

publicActionResult Index()

{

**returnRedirectToAction("Index", "Home", new { Area = "HelpDesk" });**

}

As you can see, the Index() action method of the main area redirects the control to Index() action method of the HelpDesk area.

ASP.NET MVC allows you to organize your application into areas. Each area replicates the MVC structure and can be treated as independent as far as models, views and controllers are concerned. An area is registered with MVC framework using an area registration class and RegisterAllAreas() call in the Global.asax file.

Basically, Routing is a pattern matching system that monitor the incoming request and figure out what to do with that request. At runtime, Routing engine use the Route table for matching the incoming request's URL pattern against the URL patterns defined in the Route table.

You can register one or more URL patterns to the Route table at Application\_Start event.

## How to define route...

|  |
| --- |
| 1. publicstaticvoidRegisterRoutes(RouteCollection routes) 2. { 3. routes.MapRoute( 4. "Default",// Route name 5. "{controller}/{action}/{id}",// Route Pattern 6. new{ controller ="Home", action ="Index", id =UrlParameter.Optional} 7. // Default values for above defined parameters 8. ); 9. } 11. protectedvoidApplication\_Start() 12. { 13. RegisterRoutes(RouteTable.Routes); 14. //To:DO 15. } |

When the routing engine finds a match in the route table for the incoming request's URL,

It forwards the request to the appropriate controller and action. If there is no match in the route table for the incoming request's URL, it returns a 404 HTTP status code.

#### Note

Always remember route name should be unique across the entire application. Route name can’t be duplicate.

## How it works...

In above example we have defined the **Route Pattern {controller}/{action}/{id}** and also provide the default values for controller, action and id parameters.

**Default values means if you will not provide the values for controller or action or id defined in the pattern then these values will be serve by the routing system.**

Suppose your webapplication is running on www.example.com then the urlpattren for you application will be [www.example.com/{controller}/{action}/{id}](http://www.example.com/%7bcontroller%7d/%7baction%7d/%7bid%7d).

Hence you need to provide the controller name followed by action name and id if it is required. If you will not provide any of the value then default values of these parameters will be provided by the routing system. Here is a list of URLs that match and don't match this route pattern.

Matching URLs

|  |  |
| --- | --- |
| Request URL | Parameters |
| http://example.com/ | Controller=Home, action=Index, id=none, **Since default value of controller and action are Home and Index respectively.** |
| http://example.com/Admin | controller=Admin, action=Index, id=none**, Since default value of action is Index** |
| http://example.com/Admin/Product | controller=Admin, action=Product, id=none |
| http://example.com/Admin/Product/1 | controller=Admin, action=Product, id=1 |
| **http://example.com/Admin/Product/SubAdmin/1** | **No Match Found** |
| **http://example.com/Admin/Product/SubAdmin/Add/1** | **No Match Found** |

# Creating Custom Routes (C#)

How to add a custom route to an ASP.NET MVC application. You learn how to modify the default route table in the Global.asax file with a custom route.

For many simple ASP.NET MVC applications, the default route table will work just fine. However, you might discover that you have specialized routing needs. In that case, you can create a custom route.

Imagine, for example, that you are building a blog application. You might want to handle incoming requests that look like this:

/Archive/12-25-2009

When a user enters this request, you want to return the blog entry that corresponds to the date 12/25/2009. In order to handle this type of request, you need to create a custom route.

The Global.asax file in Listing 1 contains a new custom route, named Blog, which handles requests that look like /Archive/entry date.



The order of the routes that you add to the route table is important. Our new custom Blog route is added before the existing Default route. If you reversed the order, then the Default route always will get called instead of the custom route.

The custom Blog route matches any request that starts with /Archive/. So, it matches all of the following URLs:

/Archive/12-25-2009

/Archive/10-6-2004

/Archive/apple

/Archive/NewAction/

The custom route maps the incoming request to a controller named Archive and invokes the Entry() action. When the Entry() method is called, the entry date is passed as a parameter named entryDate.

You can use the Blog custom route with the controller in Listing 2.

**Listing 2 - ArchiveController.cs**

using System;

using System.Web.Mvc;

namespace MvcApplication1.Controllers

{

public class ArchiveController : Controller

{

public string Entry(DateTime entryDate)

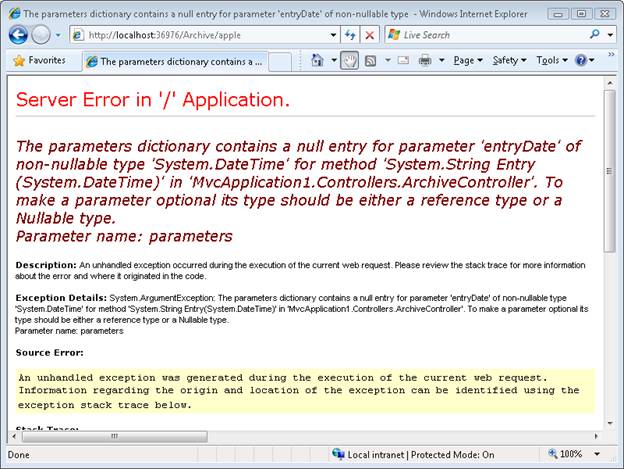
{ return "You requested the entry from " + entryDate.ToString(); }

}

}

Notice that the Entry() method in Listing 2 accepts a parameter of type DateTime. The MVC framework is smart enough to convert the entry date from the URL into a DateTime value automatically. If the entry date parameter from the URL cannot be converted to a DateTime, an error is raised (see Figure 1).

**Figure 1 - Error from converting parameter**

[](http://media-www-asp.azureedge.net/images-legacy/asp.net/images/mvc/23/image001.png)

**Figure 01**: Error from converting parameter ([Click to view full-size image](http://media-www-asp.azureedge.net/images-legacy/asp.net/images/mvc/23/image001.png))

## Summary

The goal of this tutorial was to demonstrate how you can create a custom route. You learned how to add a custom route to the route table in the Global.asax file that represents blog entries. We discussed how to map requests for blog entries to a controller named ArchiveController and a controller action named Entry().

Routing is a pattern matching system that monitor the incoming request and figure out what to do with that request. At runtime, Routing engine use the Route table for matching the incoming request's URL pattern against the URL patterns defined in the Route table. You can register one or more URL patterns to the Route table at Application\_Start event.

How to define route...

1. **publicstaticvoidRegisterRoutes(RouteCollection routes)**
2. **{**
3. **routes.MapRoute(**
4. **"Default", *// Route name***
5. **"{controller}/{action}/{id}", *// Route Pattern***
6. **new { controller = "Home", action = "Index", id = UrlParameter.Optional } *// Default values for above defined parameters***
7. **);**
8. **}**
10. **protectedvoidApplication\_Start()**
11. **{**
12. **RegisterRoutes(RouteTable.Routes);**
13. ***//To:DO***
14. **}**