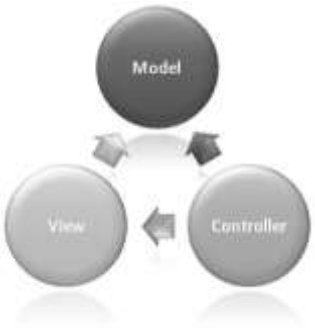
What is MVC?

- MVC is a framework for building web applications using a MVC (Model View Controller) design.
- The **Model** is the part of the application that handles the logic for the application data. Often model objects retrieve data (and store data) from a database.
- The **View** is the parts of the application that handles the display of the data. Most often the views are created from the model data.
- The **Controller** is the part of the application that handles user interaction. Typically controllers read data from a view, control user input, and send input data to the model.



What is ASP.NET MVC?

- The MVC programming model is a lighter alternative to traditional ASP.NET (Web Forms). It is a lightweight, highly testable framework, integrated with all existing ASP.NET features, such as Master Pages, Security, and Authentication.
- The Razor syntax gives you all the power of ASP.NET, but is using a simplified syntax that's easier to learn if you're a beginner, and makes you more productive if you're an expert.
- When the server reads the page, it runs the Razor code first, before it sends the HTML page to the browser.

Advantages of an MVC-Based Web Application

- It makes it easier to manage complexity by dividing an application into the model, the view, and the controller.
- It does not use view state or server-based forms. This makes the MVC framework ideal for developers who want full control over the behavior of an application.
- It uses a Front Controller pattern that processes Web application requests through a single controller. This enables you to design an application that supports a rich routing infrastructure.
- It provides better support for test-driven development (TDD).
- It works well for Web applications that are supported by large teams of developers and for Web designers who need a high degree of control over the application behavior.

Creating Asp.Net MVC Project

Take following steps to create a simple Asp.Net MVC project.

1. Select **File**->**NewProject**
2. Select **Templates** ->**Visual C# -> Web**
3. Select **ASP.NET Web Application** project template
4. Enter name of the application in **Name** textbox
5. Select **Location** where project is to be created. Use Browse button if required to select folder
6. Click on **OK**.
7. When Select a template dialog is displayed, select **MVC**.
8. Select **Change Authentication** button and select **No Authentication** radio button. Click on Ok to create a project with default settings.

Standard Folders

- When you create an ASP.NET MVC application, the following structure is created by Visual Studio.
- The folder names are same in all MVC applications. The MVC framework is based on default naming.
- Controllers are in the Controllers folder, Views are in the Views folder, and Models are in the Models folder.
- Standard naming reduces the amount of code, and makes it easier for developers to understand MVC projects.

App_Start folder

- This folder contains classes used to configure application.
- RouteConfig.cs is used to configure routing using RouteCollection.
- FilterConfig.cs is used to register global filters.
- BundleConfig.cs is used to add ScriptBundles and StyleBundles to BundleCollection.
- Method in these classes are called from Application_Start() event in Global.Asax

The Content Folder

- The Content folder is used for static files like style sheets (css files), icons and images.
- Visual Studio also adds a standard style sheet file to the project - Site.css and other style sheets in the content folder.

The Controllers Folder

- The Controllers folder contains the controller classes responsible for handling user input and responses.
- MVC requires the name of all controller files to end with "Controller".

The Models Folder

- The Models folder contains the classes that represent the application models.
- Models hold and manipulate application data.

The Views Folder

- The Views folder stores the HTML files related to the display of the application (the user interfaces).
- The Views folder contains one folder for each controller.
- The **Shared** folder is used to store views shared between controllers (master pages and layout pages).
- _Layout.cshtml is used as the default layout for views.

The Scripts Folder

- The Scripts folder stores the JavaScript files of the application.
- By default Visual Studio fills this folder with standard bootsrap, jQuery and modernizer files

_ViewStart.cshtml

- This code is automatically added to all views displayed by the application.
- The _ViewStart file in the Shared folder (inside the Views folder) contains the following content.

```
@{
      Layout = "~/Views/Shared/_Layout.cshtml";
}
```

HomeController.cs

- By default HomeController is created with the following code.
- It contains three actions – Index, About and Contact, which will invoke the corresponding .cshtml files in Views\Home folder.

```
public class HomeController : Controller
{
 public ActionResult Index()
 {
   return View();
 }
 public ActionResult About()
 {
   ViewBag.Message = "Your application description page.";
   return View();
 }
 public ActionResult Contact()
 {
   ViewBag.Message = "Your contact page.";
   return View();
 }
}
```

Running Application

- When you run ASP.NET MVC application, it displays Index.cshtml as Home is the default controller and Index is default action. When action **Index**() returns ActionResult using View() method control goes to Index.cshtml in Views/Home folder.
- You can invoke other actions using /Controller/Action in URL.
- To invoke About action in Home controller, we need to invoke /Home/About

Routing

- To handle MVC URLs, the ASP.NET platform uses the routing system.
- When processing an incoming request, the job of the routing system is to match the URL that has been requested to a pattern and extract values from the URL for the segment variables defined in the pattern
- The segment variables are expressed using braces ({}).
- The routing system has two functions:
  - Examine an incoming URL and figure out for which controller and action the request is intended.
  - Generate outgoing URLs. These are the URLs that appear in the HTML rendered from views so that a specific action will be invoked when the user clicks the link (at which point, it has become an incoming URL again).

```
public class RouteConfig {
    public static void RegisterRoutes(RouteCollection routes) {
        routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
        routes.MapRoute(name: "Default",
            url: "{controller}/{action}/{id}",
            defaults: new { controller = "Home", action = "Index",
                            id = UrlParameter.Optional }
        );
    }
}
```

Application_Start event is used to call RegisterRoutes() method as shown below:

```
public class MvcApplication : System.Web.HttpApplication {
    protected void Application_Start() {
           // other code
           RouteConfig.RegisterRoutes(RouteTable.Routes);
           // other code
    }
}
```

Convention vs. Attribute Routing

- Attribute routing is one of the major additions to MVC 5.
- Attribute routing is disabled by default.
- It is enabled by MapMvcAttributeRoutes() method as follows.

```
public static void RegisterRoutes(RouteCollection routes) {
     routes.MapMvcAttributeRoutes();
     routes.MapRoute("Default", "{controller}/{action}/{id}",
                      new { controller = "Home", action = "Index",
                            id = UrlParameter.Optional });

}
```

Route attribute

- This attribute is used to specify the pattern used to select an action method.
- It is applied at action as well as controller

```
[Route("Test")]
public ActionResult Index() {

}

[Route("Books/Add/{title}")]
public string Create(string title) {
}
```

Note: When Route attribute is used, the method is not considered by convention routing.

Controller

- A controller class name must end with "Controller"
- Controller action methods respond to requests that are sent to the controller.
- Action methods must be public.
- Action methods cannot be static.
- Action methods cannot have unbounded generic type parameters. An unbounded generic type parameter has an empty parameter list. An unbounded generic type is also known as an open generic type.
- Action methods cannot be overloaded based on parameters. Action methods can be overloaded when they are disambiguated with attributes such as AcceptVerbsAttribute.

Adding Controller

Add a new controller using the following steps:

1. Select Controllers folder in Solution Explorer.
2. Right click and select Add option from popup menu then select Controller.
3. In the next window, select MVC 5 Controller – Empty option and click on Add button.
4. Enter controller name – HelloController and click on Add button.
5. An empty controller is created with Index action. Modify Index() as follows.

```
public class HelloController : Controller
{
   // GET: /Hello/
  public ActionResult Index()
  {
     ViewBag.Message = "Hello World!";
     return View();
 }
}
```

Right click on Index action and select Add View option and click on Add button to crate view – Views/Hello/Index.cshtml.

```
@{
    ViewBag.Title = "Index";
    Layout = null;
}
<h2> @ViewBag.Message</h2>
```

Run project and invoke Index action in Hello controller using /Hello/Index

Adding Model

- Model represents data that is stored and transferred from controller to view.
- Model is typically a POCO.

Models/Person.cs

```
public class Person
{
       public string FirstName { get; set; }
       public string LastName { get; set; }
}
```

Controllers/HelloController.cs

```
public class HelloController : Controller
{
     // GET: /Hello/
     public ActionResult Index()
     {
          ViewBag.Message = "Hello World!";
          return View();
     }
     public ActionResult Greet(String first="Srikanth", String last="Pragada")
     {
         Person p = new Person {FirstName = first, LastName = last };
          return View(p);
     }
}
```

View/Hello/Greet.cshtml

```
@model demo.Models.Person

@{
    ViewBag.Title = "Greet";
    Layout = null;
}

<h2>Welcome, @Model.FirstName @Model.LastName</h2>
```

ViewBag, ViewData and TempData Properties

- ASP.NET MVC offers us three options ViewData, ViewBag and TempData for passing data from controller to view and in next request.
- They are properties of **ControllerBase** class
- ViewBag is actually just a wrapper around the ViewData object, and its whole purpose is to let you use dynamics to access the data instead of using keys.
- ViewBag is **DynamicViewDataDictionary** with the ViewData as its data

Similarities between ViewBag & ViewData :

- Helps to maintain data when you move from controller to view.
- Used to pass data from controller to corresponding view.
- Short life means value becomes null when redirection occurs. This is because their goal is to provide a way to communicate between controllers and views. It's a communication mechanism within the server call.

Difference between ViewBag & ViewData

- ViewData is a dictionary of objects that is derived from *ViewDataDictionary* class and accessible using strings as keys.
- ViewBag is a dynamic property that takes advantage of the new dynamic features in C# 4.0.
- ViewData requires typecasting for complex data type and check for null values to avoid error.
- ViewBag doesn't require typecasting for complex data type.

```
publicActionResult Index()
{
   ViewBag.Name = "Srikanth Pragada ";
   return View();
}

publicActionResult Index()
{
   ViewData["Name"] = "Srikanth Pragada";
   return View();
}
```

In View:

Collapse | Copy Code

```
@ViewBag.Name
@ViewData["Name"]
```

TempData

- TempData is also a dictionary derived from *TempDataDictionary* class and stored in short lives session and it is a string key and object value.
- TempData helps to maintain data when you move from one controller to other controller or from one action to other action.
- In other words when you redirect, "Tempdata" helps to maintain data between those redirects.
- It internally uses Session variables.
- It requires typecasting for complex data type and check for null values to avoid error.
- Generally used to store only one time messages like error messages, validation messages.

Razor – View Engine

- The Razor syntax gives you all the power of ASP.NET, but using a simplified syntax that's easier to learn if you're a beginner and that makes you more productive if you're an expert.
- Razor code blocks are enclosed in @{ ... }
- Inline expressions (variables and functions) start with @
- Code statements end with semicolon
- Variables are declared with the var keyword. You can store values in a variable, including strings, numbers, and dates, etc. You can insert variable values directly in a page using @.
- Strings are enclosed with quotation marks
- C# code is case sensitive
- C# files have the extension .cshtml
- By default content emitted using a @ block is automatically HTML encoded to better protect against XSS attack scenarios.
- Razor supports consistent look and feel across all pages in website using "layout pages" – which allow you to define a common site template, and then inherit its look and feel across all the views/pages on your site.

```
<!-- Single statement block -->
@{ var myMessage = "Hello World"; }

<!-- Inline expression or variable -->
<p>The value of name is: @name</p>

<!-- Multi-statement block -->
@{
 var greeting = "Welcome to our site!";
 var weekDay = DateTime.Now.DayOfWeek;
 var greetingMessage = greeting + " Here in Huston it is: " + weekDay;
}
<p>The greeting is: @greetingMessage</p>
```

if and else Conditions

An important feature of dynamic web pages is that you can determine what to do is based on conditions.The common way to do this is with the if ... else statements:

```
@{
var txt = "";
if(DateTime.Now.Hour> 12)
  {txt = "Good Evening";}
else
  {txt = "Good Morning";}
}
```

Reading User Input

Another important feature of dynamic web pages is that you can read user input.Input is read by the Request[] function, and posting (input) is tested by the IsPost condition:

```
@{ var totalMessage = "";
if(IsPost){
    var num1 = Request["text1"];
    var num2 = Request["text2"];
    var total = num1.AsInt() + num2.AsInt();
    totalMessage = "Total = " + total;
}
}
```

Data Types

Below is a list of common data types:

Type	Examples
int	103, 12, 5168
float	3.14, 3.4e38
decimal	1037.196543
bool	true, false
string	"Hello W3Schools", "John"

Operators

An operator tells ASP.NET what kind of command to perform in an expression.The C# language supports many operators. Below is a list of common operators:

Operator	Description
+ - * /	Math operators used in numerical expressions.
=	Assignment. Assigns the value on the right side of a statement to the object on the left side.
==	Equality. Returns true if the values are equal. (Notice the distinction between the =operator and the ==operator.)
!=	Inequality. Returns true if the values are not equal.
< > <= >=	Less-than, greater-than, less-than-or-equal, and greater-than-or-equal.
+	Concatenation, which is used to join strings. ASP.NET knows the difference between this operator and the addition operator based on the data type of the expression.
+= -=	The increment and decrement operators, which add and subtract 1 (respectively) from a variable.
.	Dot. Used to distinguish objects and their properties and methods.
()	Parentheses. Used to group expressions and to pass parameters to methods.
[]	Brackets. Used for accessing values in arrays or collections.
!	Not. Reverses a true value to false and vice versa. Typically used as a shorthand way to test for false (that is, for not true).
&& ||	Logical AND and OR, which are used to link conditions together.

Converting Data Types

- Converting from one data type to another is sometimes useful.The most common example is to convert string input to another type, such as an integer or a date.
- As a rule, user input comes as strings, even if the user entered a number. Therefore, numeric input values must be converted to numbers before they can be used in calculations.
- Below is a list of common conversion methods:

Method	Description	Example
AsInt() IsInt()	Converts a string to an integer.	if (myString.IsInt()) {myInt=myString.AsInt();}
AsFloat() IsFloat()	Converts a string to a floating-point number.	if (myString.IsFloat()) {myFloat=myString.AsFloat();}
AsDecimal() IsDecimal()	Converts a string to a decimal number.	if (myString.IsDecimal()) {myDec=myString.AsDecimal();}
AsDateTime() IsDateTime()	Converts a string to an ASP.NET DateTime type.	myString="10/10/2012"; myDate=myString.AsDateTime();
AsBool() IsBool()	Converts a string to a Boolean.	myString="True"; myBool=myString.AsBool();
ToString()	Converts any data type to a string.	myInt=1234; myString=myInt.ToString();

for Loop

If you need to run the same statements repeatedly, you can program a loop. If you know how many times you want to loop, you can use a for loop. This kind of loop is especially useful for counting up or counting down:

```
@for(vari = 10; i< 21; i++)
    {<p>Line @i</p>}
```

foreach Loop

If you work with a collection or an array, you often use a for each loop.A collection is a group of similar objects, and the for each loop lets you carry out a task on each item. The for each loop walks through a collection until it is finished.The example below walks through the ASP.NET Request.ServerVariables collection.

```
<ul>
@foreach (var x in Request.ServerVariables)
    {<li>@x</li>}
</ul>
```

HTML Helpers

- With MVC, HTML helpers are much like traditional ASP.NET Web Form controls.
- Just like web form controls in ASP.NET, HTML helpers are used to modify HTML.
- But HTML helpers are more lightweight. Unlike Web Form controls, an HTML helper does not have an event model and a view state.
- In most cases, an HTML helper is just a method that returns a string.
- With MVC, you can create your own helpers, or use the built in HTML helpers.
- HTML Helpers are methods that can be invoked within code-blocks, and which encapsulate generating HTML.
- These are implemented using pure code, typically as extension methods.

HTML Links – HTML.ActionLink()

The easiest way to render an HTML link is to use the HTML.ActionLink() helper.With MVC, the Html.ActionLink() does not link to a view. It creates a link to a controller action.

```
@Html.ActionLink("About this Website", "About")
```

The first parameter is the link text, and the second parameter is the name of the controller action.TheHtml.ActionLink() helper above, outputs the following HTML:

```
<a href="/Home/About">About this Website</a>
```

```
@Html.ActionLink("Edit Record", "Edit", new {Id=3})
```

Property	Description
.linkText	The link text (label)
.actionName	The target action
.routeValues	The values passed to the action
.controllerName	The target controller
.htmlAttributes	The set of attributes to the link
.protocol	The link protocol
.hostname	The host name for the link
.fragment	The anchor target for the link

HTML Form Elements

The following HTML helpers can be used to render (modify and output) HTML form elements:

- BeginForm()
- EndForm()
- TextArea()
- TextBox()
- CheckBox()
- RadioButton()
- ListBox()
- DropDownList()
- Hidden()
- Password()

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

HTML Form Example

The following example shows how to use HTML form and different actions for GET and POST Methods.

Controllers/InventoryController.cs

```
usingdemo.Models;
usingSystem.Web;
usingSystem.Web.Mvc;
namespacedemo.Controllers
{
  public class InventoryController : Controller
  {
      public ActionResultSellingPrice()
      {
             Tax t = new Tax();
          return View(t);
      }

      [HttpPost]
      publicActionResultSellingPrice(Tax t)
      {
          t.SellingPrice = t.Amount + (t.Amount * t.TaxRate / 100);
          return View(t);
      }
  }
}
```

Models/Tax.cs

```
public class Tax
{
 public double Amount { get; set; }
 public double TaxRate { get; set; }
 public double SellingPrice { get; set; }
}
```

Views/Inventory/SellingPrice.cshtml

```
@model demo.Models.Tax
@{
   ViewBag.Title = "SellingPrice";
}
<h2>Selling Price Calculation</h2>
@using (Html.BeginForm("SellingPrice", "Inventory", FormMethod.Post))
{
  @Html.Label("Base Price : ")  <br />
  @Html.TextBox("Amount", @Model.Amount) <p />
  @Html.Label("Tax Rate:")  <br />
  @Html.TextBox("TaxRate", @Model.TaxRate) <p />
  <input type="submit" value="Calculate Selling Price" />
  <p />
}

@if (Model.SellingPrice> 0)
{
    <h3>Selling Price :  @Model.SellingPrice</h3>
}
```

ActionResult

- The action result is a very generic return value for an action. This is because it is the abstract base class for other types of actions. It is actually a very simple class having only one method that needs implementing.
- More precisely, you can design an action method to return any .NET type, including primitive and complex types.
- You can also return void. If the action method is void, the actual type being processed is EmptyResult.
- If the type is any .NET type that doesn't inherit ActionResult, the actual response is encapsulated in a ContentResult type.

Action Result	Helper Method	Description
ViewResult	View	Renders a view as a Web page.
PartialViewResult	PartialView	Renders a partial view, which defines a section of a view that can be rendered inside another view.
RedirectResult	Redirect	Redirects to another action method by using its URL.
RedirectToRouteResult	RedirectToAction or RedirectToRoute	Redirects to another action method.
ContentResult	Content	Returns a user-defined content type.
JsonResult	Json	Returns a serialized JSON object.
JavaScriptResult	JavaScript	Returns a script that can be executed on the client.
HttpStatusCodeResult	(None)	Returns a specific HTTP response code and description.
HttpUnauthorizedResult	(None)	Returns the result of an unauthorized HTTP request.
HttpNotFoundResult	HttpNotFound	Indicates the requested resource was not found.
FileResult	File	Returns binary output to write to the response.
FileContentResult	Controller.File(Byte[], String) or Controller.File(Byte[], String, String)	Sends the contents of a binary file to the response.
FilePathResult	Controller.File(String, String) or Controller.File(String, String, String)	Sends the contents of a file to the response.
FileStreamResult	Controller.File(Stream, String) or Controller.File(Stream, String, String)	Sends binary content to the response through a stream.
EmptyResult	(None)	Represents a return value that is used if the action method must return a null result (void).

ViewResultBase, ViewResult, and PartialViewResult

- The ViewResult is the most common concrete type you will be returning as a controller action.
- It has an abstract base class called ViewResultBase, which it shares with PartialViewResult.
- It is in the ViewResultBase abstract base class that we get access to all of our familiar data objects like: TempData, ViewData, and ViewBag.
- PartialViews are not common as action results.
- PartialViews are not the primary thing being displayed to the user, which is the View. The partial view is usually a widget or something else on the page. It's usually not the primary content the user sees.
- The following is the common return syntax, and it means that you're returning a ViewResult.

```
return View();
```

That is actually a call to the base Controller.View() method, which is just going to call through with some defaults.

```
protected internal ViewResult View(){
    return View(null, null, null);
}
```

The beauty of ASP.NET MVC is actually in its simplicity though, because all that really did was create our ViewResult for us. If we take a look at the method that is being called you can see that we're just taking a little shortcut and keeping our action clean of this code we would otherwise repeat every time we wanted a ViewResult.

ContentResult

- The content result lets you define whatever content you wish to return.
- You can specify the content type, the encoding, and the content. This gives you control to have the system give whatever response you want.
- This is a good result to use when you need a lot of control over what you're returning and it's not one of the standards.
- Its ExecuteResult override is extremely simple.

JsonResult

- Represents a class that is used to send JSON-formatted content to the response.
- It also has hardcoded its ContentType, but what makes it a bit more complex is that it uses a hardcoded JavaScriptSerializer to serialize the JSON data before writing it directly to the response.

RedirectResult and RedirectToRouteResult

- These are a little bit more complex, but both are ways of redirecting.
- Each one can either be a permanent or temporary redirect and they both just use the Redirect methods on the Response object.
- For redirecting to a route, it is going to generate a URL to the route using the UrlHelper'sGenerateUrl method. For the RedirectResult it is instead going to use the UrlHelpersGenerateContentUrl method.
- Either of these two is useful, and both will maintain your TempData if you need to pass something along with the redirect, all you have to do is put it in TempData.

Validating Data using Data Annotations

- Validation is centralized in model.
- Validation rules are specified using Attributes called DataAnnotations.
- HTML Helpers read these attributes and generate data dash attributes in HTML.
- jQuery is used to process data dash attributes in client and provide validation.

Class	Description
CompareAttribute	Provides an attribute that compares two properties.
CreditCardAttribute	Specifies that a data field value is a credit card number.
CustomValidationAttribute	Specifies a custom validation method that is used to validate a property or class instance.
DataTypeAttribute	Specifies the name of an additional type to associate with a data field.
DisplayAttribute	Provides a general-purpose attribute that lets you specify localizable strings for types and members of entity partial classes.
DisplayFormatAttribute	Specifies how data fields are displayed and formatted by ASP.NET Dynamic Data.
EmailAddressAttribute	Validates an email address.
MaxLengthAttribute	Specifies the maximum length of array or string data allowed in a property.
MinLengthAttribute	Specifies the minimum length of array of string data allowed in a property.
PhoneAttribute	Specifies that a data field value is a well-formed phone number using a regular expression for phone numbers.
RangeAttribute	Specifies the numeric range constraints for the value of a data field.
RegularExpressionAttribute	Specifies that a data field value in ASP.NET Dynamic Data must match the specified regular expression.
RequiredAttribute	Specifies that a data field value is required.
StringLengthAttribute	Specifies the minimum and maximum length of characters that are allowed in a data field.
UrlAttribute	Provides URL validation.

Validation Example

This example shows how to use validation attributes and data access using LINQ.

Models/Book.cs

```
using System;
usingSystem.Collections.Generic;
usingSystem.ComponentModel.DataAnnotations;
usingSystem.Data.Linq.Mapping;
usingSystem.Linq;
usingSystem.Web;
usingSystem.Web.Mvc;
namespace catalog.Models
{
    public class Book
    {

        public int Id { get; set; }

        [Required ( ErrorMessage = "Title is required!")]
        [StringLength (50, MinimumLength = 4,
           ErrorMessage="Invalid Title. It must be between 4 and 50 characters")]
        public string Title { get; set; }


        [Required ( ErrorMessage = "Author(s) name(s) required")]
        public string Authors { get; set; }


        [Range (100,10000, ErrorMessage = "Price must be between 100 and 10000")]
        public decimal Price { get; set; }

        public string Publisher { get; set; }

        public string HardBound { get; set; }
        public static List<SelectListItem> Publishers
        {
         get
            {
                return new List<SelectListItem>
               {
                new SelectListItem { Text = "Willy", Value =  "Willy"  },
                new SelectListItem { Text = "Sams", Value =  "Sams"  },
                new SelectListItem { Text = "Manning", Value =  "Manning"  }
               };
            }
        }
    }
}
```

Controllers/BookController.cs

```
public class BookController : Controller
{
      public ActionResult Add ()
      {
           Book b = new Book();
          return View(b);
      }

      [HttpPost]
      public ActionResult Add(Book b)
      {
          if (ModelState.IsValid)
          {
                // process data
             return RedirectToAction("Index");
          }
          else
             return View(b);   // redisplay the view with errors
     }
}
```

Views/Shared/_Layout.cshtml

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width" />
  <title>@ViewBag.Title</title>
  @Styles.Render("~/Content/css")
  @Scripts.Render("~/bundles/modernizr")
</head>
<body>
   <div class="header">Books Catalog</div>
   <div class="links">
     @Html.ActionLink("Home", "Index","Book")
      
     @Html.ActionLink("Add", "Add","Book")
      
     @Html.ActionLink("Search", "Search","Book")
   </div>

   @RenderBody()

   @Scripts.Render("~/bundles/jquery")
   @RenderSection("scripts", required: false)

   <script src='@Url.Content("~/Scripts/jquery.validate.js")'></script>
   <script
     src='@Url.Content("~/Scripts/jquery.validate.unobtrusive.js")'></script>
   <scriptsrc='@Url.Content("~/Scripts/jquery.unobtrusive-ajax.js")'></script>
</body>
</html>
```

RemoteAttribute

- Provides an attribute that uses the jQuery validation plug-in remote validator.
- In order for this to work, we need to make jQuery validation and jQuery Ajax also available.

```
public RemoteAttribute(
      string action,
      string controller
)
```

```
public class UserController : Controller
{
   // Used by remote validation
   public ActionResult ValidateEmail(string email)
   {
     if (!email.EndsWith("gmail.com"))
              return Json("Invalid Email. It must be gmail account!",
                                  JsonRequestBehavior.AllowGet);
     return Json("true", JsonRequestBehavior.AllowGet);
   }
}
```

```
public class User
{
   [System.Web.Mvc.Remote("ValidateEmail", "User")]
   [Required(ErrorMessage = "Email is required!")]
   public string Email { get; set; }
}
```

AjaxHelper class

- Represents support for rendering HTML in AJAX scenarios within a view.
- The AjaxHelper class includes methods that provide client-side functionality in ASP.NET AJAX in MVC applications, such as creating asynchronous forms and rendering links.
- The AjaxHelper class supports asynchronous partial-page updates.
- Extensions to the AjaxHelperclass are in the System.Web.Mvc.Ajax namespace.
- Helper methods and extensions are called using the Ajax property of the view, which is an instance of the AjaxHelper class. For example, to generate a link to a controller action method, you can call the ActionLink method in your view using the syntax -
  <%= Ajax.ActionLink("ActionName") %>

ActionLink Method

- Returns an anchor element that contains the URL to the specified action method; when the action link is clicked, the action method is invoked asynchronously by using JavaScript.
- This method renders an anchor element. When the user clicks the link, MVC asynchronously invokes the specified action method via an HTTP POST request.
- The response of that action method can be used to update a specified DOM element, depending on which AjaxOptions are specified.

```
ActionLink(String, String, AjaxOptions)
ActionLink(String, String, Object, AjaxOptions)
ActionLink(String, String, RouteValueDictionary, AjaxOptions)
```

AjaxForm Method

The form is submitted asynchronously by using JavaScript.

```
BeginForm(AjaxOptions)
BeginForm(String, AjaxOptions)
BeginForm(String, Object, AjaxOptions)
BeginForm(String, RouteValueDictionary, AjaxOptions)
```

AjaxOptions Object

Provides settings through which we can configure Ajax calls.

Name	Description
Confirm	Gets or sets the message to display in a confirmation window before a request is submitted.
HttpMethod	Gets or sets the HTTP request method ("Get" or "Post").
InsertionMode	Gets or sets the mode that specifies how to insert the response into the target DOM element.
LoadingElementDuration	Gets or sets a value, in milliseconds, that controls the duration of the animation when showing or hiding the loading element.
LoadingElementId	Gets or sets the id attribute of an HTML element that is displayed while the Ajax function is loading.
OnBegin	Gets or sets the name of the JavaScript function to call immediately before the page is updated.
OnComplete	Gets or sets the JavaScript function to call when response data has been instantiated but before the page is updated.
OnFailure	Gets or sets the JavaScript function to call if the page update fails.
OnSuccess	Gets or sets the JavaScript function to call after the page is successfully updated.
UpdateTargetId	Gets or sets the ID of the DOM element to update by using the response from the server.

Url	Gets or sets the URL to make the request to.

Ajax Examples

The following examples show how to use Ajax Helper like ActionLink and BeginForm.

AjaxController.cs

```
using System;
usingSystem.Web;
usingSystem.Web.Mvc;

namespace demo.Controllers
{
    public class AjaxController : Controller
    {
        public ActionResult Now()
        {
             return View();
        }
        public string DateAndTime()
        {
             return DateTime.Now.ToString());
        }
        public ActionResult Search()
        {
             return View();
        }
        public ActionResult SearchResult(string title)
        {
          return PartialView("_SearchResult", "You are searching for "  + title);
        }
    }
}
```

Views/Ajax/Now.cshtml

```
@{
   ViewBag.Title = "Now";
}

<script src='@Url.Content("~/Scripts/jquery.1.10.2.js")'></script>
<script src='@Url.Content("~/Scripts/jquery.unobtrusive-ajax.js")'></script>

<h2>Now</h2>
@Ajax.ActionLink("Get Date and Time", "DateAndTime", "Ajax",
       new AjaxOptions { UpdateTargetId = "datetime"})
<p />
<h2 id="datetime"></h2>
```

Views/Ajax/Search.cshtml

```
@{
   ViewBag.Title = "Search";
}
<script src='@Url.Content("~/Scripts/jquery.1.10.2.js")'></script>
<script src='@Url.Content("~/Scripts/jquery.unobtrusive-ajax.js")'></script>

<h2>Search</h2>

@using (Ajax.BeginForm("SearchResult",
         new AjaxOptions { UpdateTargetId = "searchresult"}))
{
<span>Title :</span>
<input type="text" name="title" />
<input type="submit" value="Search" />
<div id="searchresult"></div>
}
```

Views/Ajax/_searchresult.cshtml

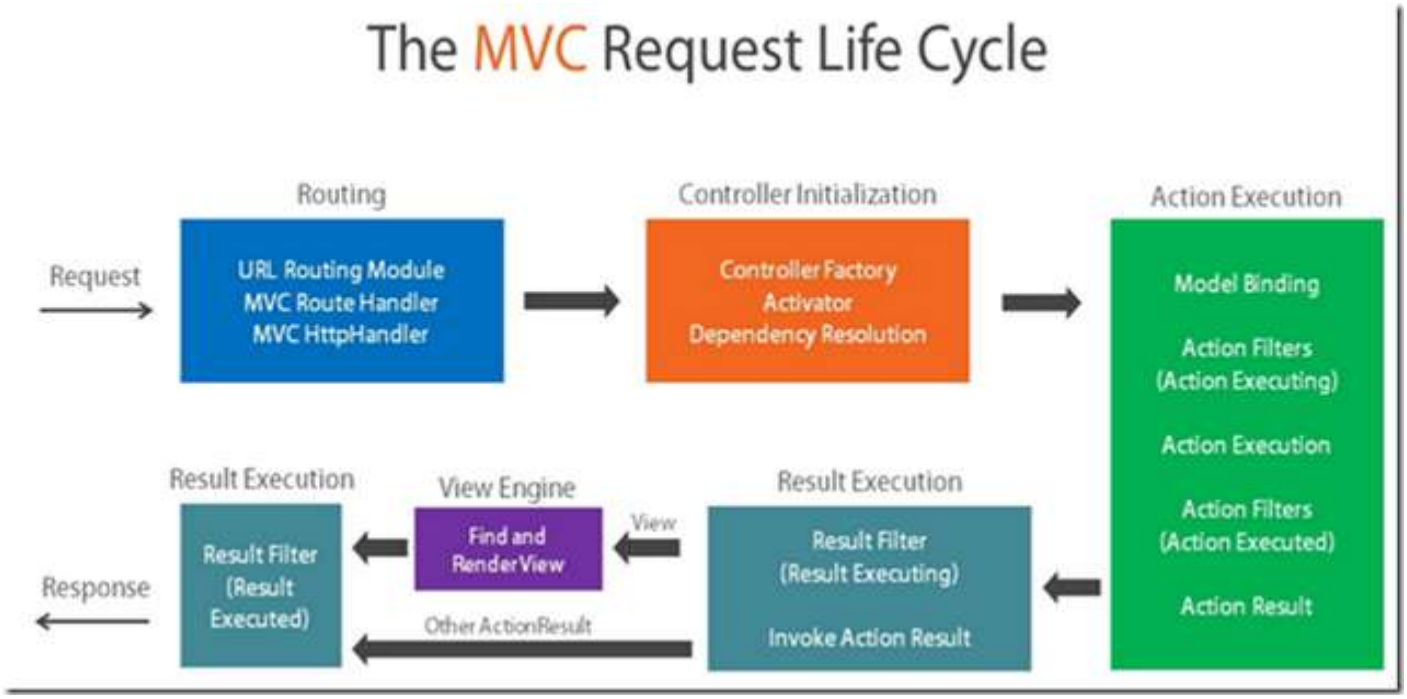
```
@model String

<h2>@Model</h2>
```

MVC Request Life Cycle

The following are important steps in request processing of MVC request.

- The incoming request from IIS pipeline is handed over to URL Routing module which analyses the request and looks up Routing table to figure out which controller the incoming request maps to.
- An instance **MVCRouteHandler** executes and brings instance of MVCHttpHandler.
- Controller factory is used to create an instance of controller.
- Dependency injection takes place at this time.
- After action is selected, Authentication and Authorization filters come into picture.
- Corresponding action is executed by ActionInvoker.
- Before that model binding, which maps http request parameters to method parameters, takes place.
- After model binding, action filters are invoked and then action execution. First OnActionExecuting then action and lastly OnActionExecuted.
- Then Result execution starts. If action results in view then a view engine is used to render response. If result is not view then it is sent as-it-is to http response.
- Before result is processed, Result Executing (OnResultExecuting) is called and then after that Result Executed (OnResultExecuted) is called.



Filters

Sometimes you want to perform logic either before an action method is called or after an action method runs. To support this, ASP.NET MVC provides filters. Filters are custom classes that provide both a declarative and programmatic means to add pre-action and post-action behavior to controller action methods.

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

- Authorization filters. These implement IAuthorizationFilter and make security decisions about whether to execute an action method, such as performing authentication or validating properties of the request. The AuthorizeAttribute class and the RequireHttpsAttribute class are examples of an authorization filter. Authorization filters run before any other filter.
- Action filters. These implement IActionFilter and wrap the action method execution. The IActionFilter interface declares two methods : OnActionExecuting and OnActionExecuted. OnActionExecuting runs before the action method. OnActionExecuted runs after the action method and can perform additional processing, such as providing extra data to the action method, inspecting the return value, or canceling execution of the action method.
- Result filters. These implement IResultFilter and wrap execution of the ActionResult object. IResultFilter declares two methods: OnResultExecuting and OnResultExecuted. OnResultExecuting runs before the ActionResult object is executed. OnResultExecuted runs after the result and can perform additional processing of the result, such as modifying the HTTP response. The OutputCacheAttribute class is one example of a result filter.
- Exception filters. These implement IExceptionFilter and execute if there is an unhandled exception thrown during the execution of the ASP.NET MVC pipeline. Exception filters can be used for tasks such as logging or displaying an error page. The HandleErrorAttribute class is one example of an exception filter.

IAuthorizationFilter	OnAuthorization
IActionFilter	OnActionExecuting and OnActionExecuted
IResultFilter	OnResultExecuting and OnResultExecuted
IExceptionFilter	OnException

Filters Provided in ASP.NET MVC

ASP.NET MVC includes the following filters, which are implemented as attributes. The filters can be applied at the action method, controller, or application level.

AuthorizeAttribute

Restricts access by authentication and optionally authorization. It implements IAuthorizationFilter

HandleErrorAttribute

Specifies how to handle an exception that is thrown by an action method. It implements IExceptionFilter

OutputCacheAttribute

Provides output caching. Extends ActionFilterAttribute, which is derived from FilterAttribute and implements IActionFilter and IResultFilter

RequireHttpsAttribute

Forces unsecured HTTP requests to be resent over HTTPS. It implements IAuthorizationFilter

Creating Filter

You can create a filter in the following ways:

- Override one or more of the controller's On<Filter> methods.
- Create an attribute class that derives from ActionFilterAttribute and apply the attribute to a controller or an action method.
- Register a filter with the filter provider (the FilterProviders class).
- Register a global filter using the GlobalFilterCollection class.

Note: You can use the filter attribute declaratively with action methods or controllers. If the attribute marks a controller, the action filter applies to all action methods in that controller.

Filter Order

Filters run in the following order:

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

For example, authorization filters run first and exception filters run last. Within each filter type, the Order value specifies the run order.

Within each filter type and order, the Scope enumeration value specifies the order for filters. This enumeration defines the following filter scope values (in the order in which they run):

1. First
2. Global
3. Controller
4. Action
5. Last

For example, an OnActionExecuting(ActionExecutingContext) filter that has the Order property set to zero and filter scope set to First runs before an action filter that has the Order property set to zero and filter scope set to Action.

Because exception filters run in reverse order, an exception filter that has the Order property set to zero and filter scope set to First runs after an action filter that has the Order property set to zero and filter scope set to Action.

The execution order of filters that have the same type, order, and scope is undefined.

```
public class LogFilter : ActionFilterAttribute
{
    public override void OnActionExecuting(ActionExecutingContext filterContext)
    {
       base.OnActionExecuting(filterContext);
       Debug.WriteLine("About to execute action : " +
                         filterContext.ActionDescriptor.ActionName);

    }
}
```

```
public class TestController : Controller
    {
        public ActionResult Index()
        {
             return View();
        }

        [LogFilter]
        public ActionResult About()
        {
            Debug.WriteLine("About execution");
            return View();
        }
}
```

Error Handling in ASP.NET MVC

The following are possible ways to handle exceptions in ASP.NET MVC.

- Using try and catch blocks
- Override OnException method in Controller and send model to error page
- Use HandleError attribute and enable customError page in web.config
- Use Application_Error in Global.aspx

Override OnException method in Controller

In this method, we override OnException method in Controller class to handle errors related to Actions in the controller.

```
protected override void OnException(ExceptionContext filterContext)
{
     Exception ex = filterContext.Exception;
     filterContext.ExceptionHandled = true;

     filterContext.Result = newViewResult()
     {
       ViewName = "Error",
       ViewData = new ViewDataDictionary(ex.Message)
     };
}

// Action
public string Error(string input)
{
     int num = Int32.Parse(input);
     return num.ToString();
}
```

Use HandleErrorAttribute with controller

In this method, we use [HandleError] attribute with Controller or Actions and create required entries in web.config so that we redirect to specified file when error occurs.

Error.cshtml

```
@{
    Layout = null;
}
<!DOCTYPE html>
<html>
<head>
<meta name="viewport" content="width=device-width" />
<title>Error</title>
</head>
<body>
<h2>Sorry! Error [@Model]</h2>
</body>
</html>
```

TestController.cs

```
[HandleError]
public string Error(string input)
{
   int num = Int32.Parse(input);
   return "Testing..";
}
```

web.config

```
<system.web>
    <customErrors defaultRedirect="error.cshtml" mode="On" />
</system.web>
```

Using Application_Error

It is possible to handle all unhandled events of controllers in Error event of Application objects**.**

```
protected void Application_Error(object sender, EventArgs e)
{
  Exception exception = Server.GetLastError();
  Server.ClearError();
  Response.Redirect("/Test/HandleError");
}
```

Caching

- Caching enables reuse of data or output without having to go through the whole process to generate output or data.
- Caching comes in two modes – Output and Data.

Output Caching

- Caches the output produced by an action and reuses it for given amount of time.
- It is implemented using [OutputCache] attribute.
- OutputCache attribute properties can be used to fine-tune caching.

Property	Meaning
CacheProfile	Gets or sets the cache profile name.
Duration	Gets or sets the cache duration, in seconds.
Location	Gets or sets the location.
SqlDependency	Gets or sets the SQL dependency.
VaryByContentEncoding	Gets or sets the vary-by-content encoding.
VaryByCustom	Gets or sets the vary-by-custom value.
VaryByHeader	Gets or sets the vary-by-header value.
VaryByParam	Gets or sets the vary-by-param value.

CacheController.cs

```
public class CacheController : Controller
{
  [OutputCache( Duration = 60, VaryByParam = "*")]
  public ActionResult Index()
  {
    ViewBag.Message = DateTime.Now.ToString();
    return View();
  }

  [OutputCache( CacheProfile ="Cache2Mins")]
  public ActionResult Now()
  {
    ViewBag.Message = DateTime.Now.ToString();
    return View("Index");
  }
}
```

Index.cshtml

```
<body>
     <h2>@ViewBag.Message</h2>
</body>
```

Web.config

```
<system.web>
    <caching>
      <outputCacheSettings>
        <outputCacheProfiles>
          <add name="Cache2Mins" duration="120" varyByParam="none"/>
        </outputCacheProfiles>
      </outputCacheSettings>
    </caching>
</system.web>
```

Data Caching

- Allows any arbitrary data to be cached and reused.
- Implemented using Cache object

CacheController.cs

```
public class CacheController : Controller
{
   private List<Book> GetBooks()
   {
     BooksContext ctx = new BooksContext();
     List<Book> books = HttpContext.Cache["books"] as List<Book>;
     if (books == null) {
         books = ctx.Books.ToList();
         ViewBag.Message = "Cache Created!";
         HttpContext.Cache.Insert("books", books , null,
                        DateTime.Now.AddMinutes(2), TimeSpan.Zero);
     }
     else
         ViewBag.Message = "Cache Being Used!";
         return books;
     }

     public ActionResult Books()
     {
         var books =  GetBooks();
         return View(books);
     }
     public ActionResult List()
     {
         var books = GetBooks();
         return View(books);
     }
 }
```

Books.cshtml

```
@model IEnumerable<mvcdemo.Models.Book>
@{
    Layout = null;
}

<!DOCTYPE html>
<html>
 <head>
    <meta name="viewport" content="width=device-width" />
    <title>Books</title>
 </head>
 <body>
    <h1>Books</h1>
    <table class="table" width="100%">
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Title)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Price)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Subject)
            </th>
        </tr>

        @foreach (var item in Model) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.Title)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Price)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Subject)
            </td>
       </tr>
    }
    </table>
    <p></p>
    @ViewBag.Message
  </body>
</html>
```

List.cshtml

```
@model IEnumerable<mvcdemo.Models.Book>
@{
    Layout = null;
}

<!DOCTYPE html>
<html>
<head>
     <title>List</title>
</head>
<body>
    <h1>Titles</h1>
    <ul>
        @foreach (var item in Model)
        {
            <li>@item.Title</li>
        }
    </ul>
    <p></p>
    @ViewBag.Message
</body>
</html>
```