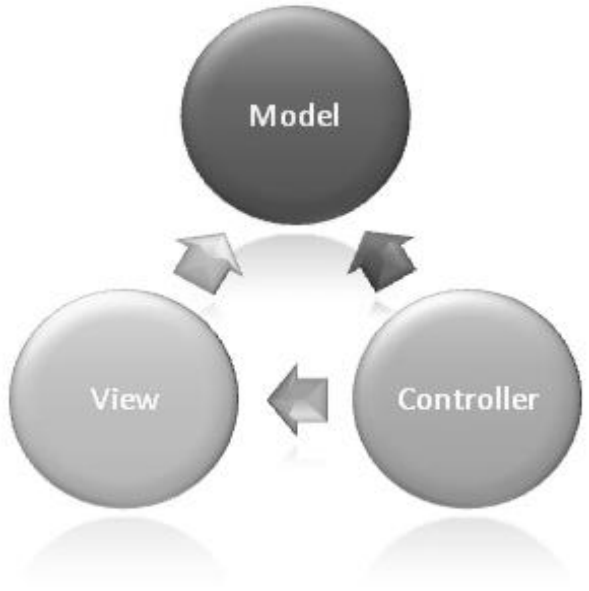
ASP.NET MVC

By

Srikanth Pragada

What is MVC?

- MVC is a framework for building web applications using 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 part 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(Layouts), 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.

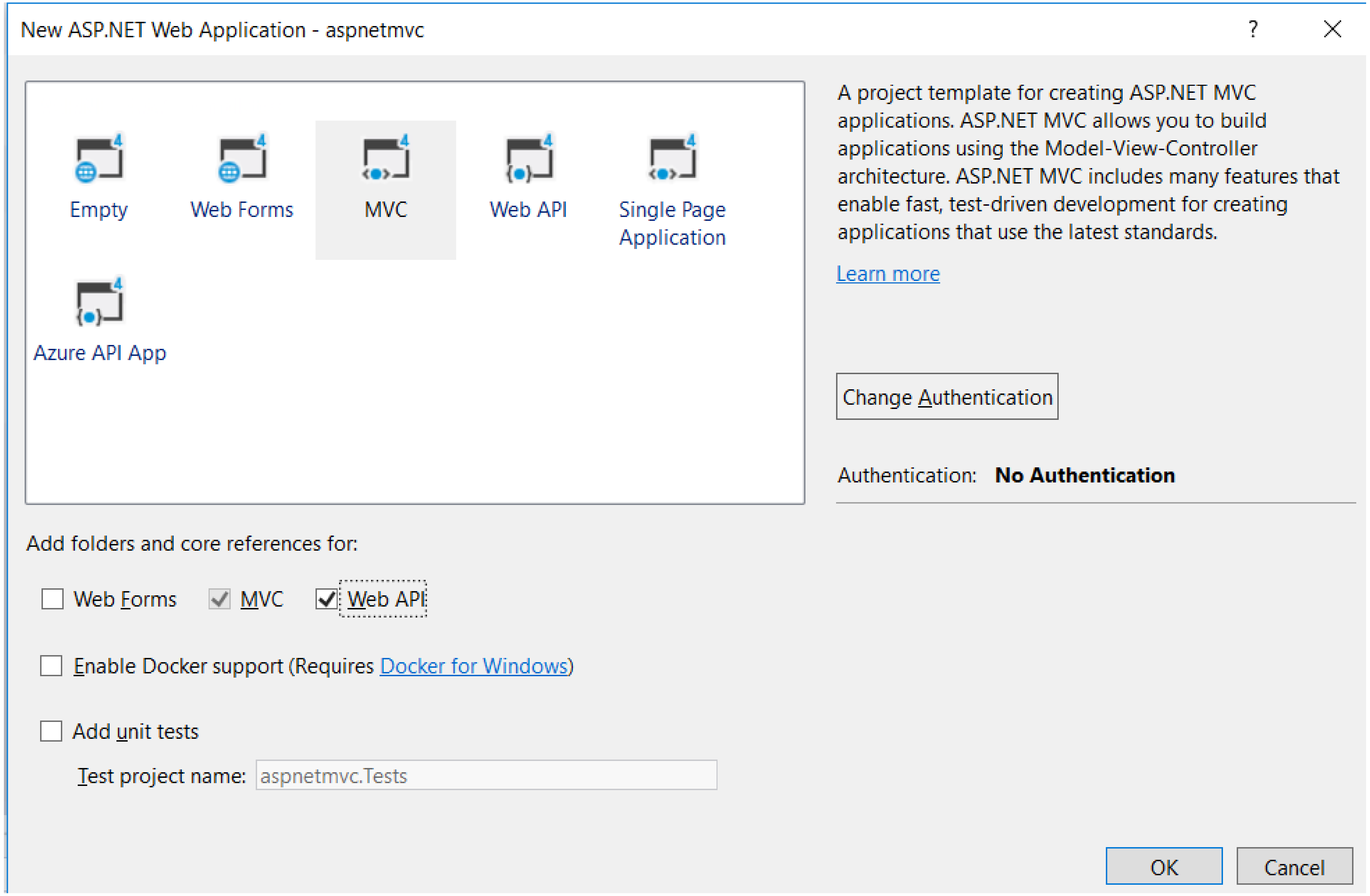
Advantages of MVC-Based Web Application

- It makes it easier to manage complexity by dividing an application into the model, the view, and the controller.
- It does not use viewstate 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

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

Creating ASP.NET MVC Project


New ASP.NET Web Application - aspnetmvc

A project template for creating ASP.NET MVC applications. ASP.NET MVC allows you to build applications using the Model-View-Controller architecture. ASP.NET MVC includes many features that enable fast, test-driven development for creating applications that use the latest standards.

Learn more

Change Authentication

Authentication: **No Authentication**

Add folders and core references for:

- [ ] Web Forms
- [✓] MVC
- [✓] Web API

- [ ] Enable Docker support (Requires Docker for Windows)

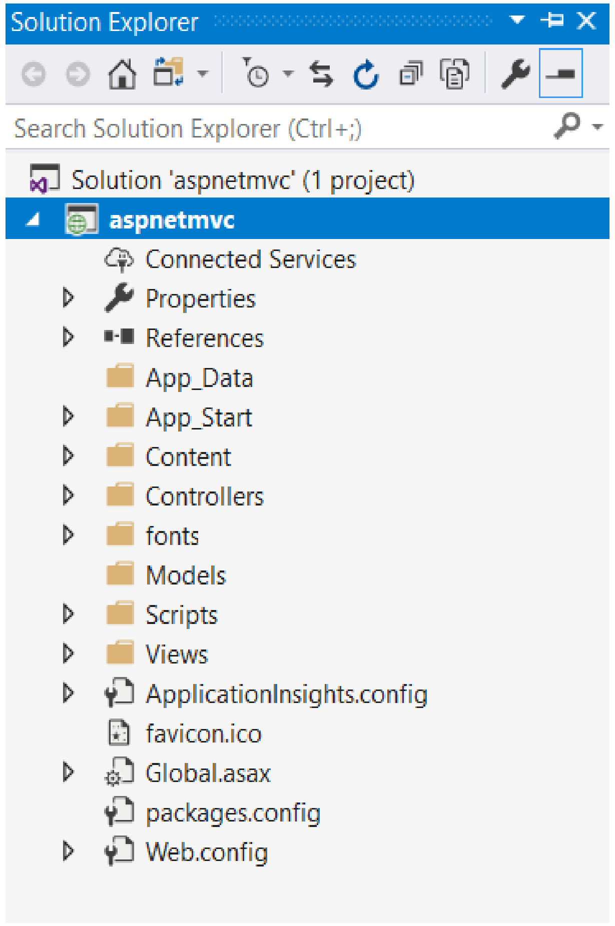
- [ ] Add unit tests

Test project name: aspnetmvc.Tests

OK | Cancel

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.



Solution Explorer

Search Solution Explorer (Ctrl+;)

```
Solution 'aspnetmvc' (1 project)
◢ aspnetmvc
    Connected Services
  ▷ Properties
  ▷ References
    App_Data
  ▷ App_Start
  ▷ Content
  ▷ Controllers
  ▷ fonts
    Models
  ▷ Scripts
  ▷ Views
  ▷ ApplicationInsights.config
    favicon.ico
  ▷ Global.asax
    packages.config
  ▷ Web.config
```

Standard Folders

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.
- ❑ WebApiConfig.cs is used to configure routing related to Web API.
- ❑ Methods in these classes are called from Application_Start() event in Global.Asax.

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.

Controllers Folder

- ❑ The Controllers folder contains the controller classes responsible for handling user input and response.
- ❑ 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.

Standard Folders

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.
- ❑ File _ViewStart.cshtml contains Razor block that is associated with all Razor pages.

Scripts Folder

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

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}
        );
    }
}
```

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

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();
  }
}
```

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

Adding Model

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

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

```
public class HelloController : Controller {
     public ActionResult Greet(String first="Srikanth", String last="Pragada") {
         Person p = new Person {FirstName = first, LastName = last };
         return View(p);
     }
}
```

```
@model demo.Models.Person
@{
    ViewBag.Title = "Greet";
    Layout = null;
}
<h2>Welcome, @Model.FirstName - @Model.LastName</h2>
```

Strongly Typed View

- A view can be associated with Model object.
- When view is associated with a model of specific type, it is called as strongly typed view.
- **@model** statement is used to specify the type of model the view is associated with.
- Inside the view, we use **@Model** to refer to model provided to view.
- Remember @Model (capital M) is used to refer to object but @model (small m) is used with statement.
- Strongly typed views are preferred as they provide auto-list and intellisense when @Model is used.

ViewBag, ViewData and TempData Properties

- ASP.NET MVC offers three options ViewData, ViewBag and TempData for passing data from controller to view.
- 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.

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

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

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

TempData

- TempData is also a dictionary derived from **TempDataDictionary** class and stored in 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 checks 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 is easier to learn if you're a beginner and 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 @ 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.

Razor – Example

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

<!-- Inline expression or variable -->
<p>Message : @myMessage</p>

<!-- Multi-statement block -->
@{
 var greeting = "Welcome to our site!";
 var weekDay = DateTime.Now.DayOfWeek;
 var greetingMessage = greeting + " Today 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 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

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

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.

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

foreach loop

- If you work with a collection or an array, you often use a foreach loop.
- A collection is a group of similar objects, and the foreach loop lets you carry out a task on each item.
- The foreach 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 Helper

- 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.
- In 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.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.
- The first parameter is the link text, and the second parameter is the name of the controller action.

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

$\downarrow$

```
<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

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

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

HTML Form Example – InventoryController.cs

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

```
usingdemo.Models;
usingSystem.Web;
usingSystem.Web.Mvc;
namespace aspnetmvc.Controllers {
  public class InventoryController : Controller
  {
     public ActionResult SellingPrice() {
           Tax t = new Tax();
           return View(t);
     }
     [HttpPost]
     public ActionResult SellingPrice(Tax t) {
           t.SellingPrice = t.Amount + (t.Amount * t.TaxRate / 100);
           return View(t);
     }
  }
}
```

HTML Form Example

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

```
@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>
}
```

Layout file

- Layout file contains common content of multiple Razor pages.
- We define layout file like any other view.
- Layout files are typically prefixed with _ (underscore) as any page with _ prefix cannot be sent to client directly.
- Layout file uses **@RenderBody** to render body of the associated view.
- **@RenderSection** is used to render a section that is defined in view.
- Layout is associated with View using Layout property.
- Set Layout property to null if view is not to be associated with any layout.

_MyLayout.cshtml

```
<!DOCTYPE html>
<html>
 <head>
   <meta name="viewport" content="width=device-width" />
   <title>@ViewBag.Title</title>
 </head>
 <body>
   <h1>Books</h1>
   @RenderBody()
   <hr/>
   <a href="http://www.srikanthtechnologies.com">Srikanth Technologies</a>
 </body>
</html>
```

Info.cshtml

```
@model  demo.Models.Book
@{
   ViewBag.Title = "Book Informaton";
   Layout = "~/Views/_MyLayout.cshtml";
}
Book Title :  @Model.Title
<p></p>
Book Price :  @Model.Price
```

_ViewStart.cshtml file contains content that is treated as if it were part of every view file.

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.

ActionResult

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);
}
```

Different Results

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.GenerateUrl() method.
- ❑ For the RedirectResult it is instead going to use the UrlHelper.GenerateContentUrl() 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 of 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 – Book.cs (Model)

```
using System.ComponentModel.DataAnnotations;
using System.Web;

namespace aspnetmvc.Models
{
    public class Book
    {
        public int Id { get; set; }
        [Required ( ErrorMessage = "Please provide title!")]
        public string Title { get; set; }

        [Range (100,1000, ErrorMessage = "Invalid Price. Must be between 100 and 1000")]
        public decimal Price { get; set; }

        [Required(ErrorMessage = "Please provide subject!")]
        [RegularExpression("^[a-zA-Z]+$", ErrorMessage = "Subject must have only alpha")]
        public string Subject { get; set; }
    }
}
```

Validation Example – BookController.cs (Controller)

```
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
     }
}
```

Validation Example – /shared/_master.cshtml (Layout Page)

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>@ViewBag.Title</title>
</head>
<body>
    <h1>Books Catalog</h1>
    <div class="menu">
        @Html.ActionLink("Add New Book", "Add", "Books")
        @Html.ActionLink("Search", "Search", "Books")
    </div>
    @RenderBody()

    @Scripts.Render("~/bundles/jquery")
    @RenderSection("myscripts", false)
</body>
</html>
```

Validation Example – Add.cshtml (Razor Page)

```
@model mvcdemo.Models.Book
@{Layout = "~/views/shared/_master.cshtml";}
<h2>Add Book</h2>
@Html.ValidationSummary(true)
@using (Html.BeginForm()) {
        <label>Title</label> <br />
        @Html.EditorFor(model => model.Title)
        @Html.ValidationMessageFor(model => model.Title)
        <p></p>
        <label>Price</label> <br />
        @Html.EditorFor(model => model.Price)
        @Html.ValidationMessageFor(model => model.Price)
        <p></p>
        <label>Subject </label> <br />
        @Html.EditorFor(model => model.Subject)
        @Html.ValidationMessageFor(model => model.Subject)
        <p></p>
        <input type="submit" value="Add Book" />
}
@section myscripts {
    @Scripts.Render("~/bundles/jqueryval")
}
```

AjaxHelper

- 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 AjaxHelper class 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

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 Example - AjaxController.cs

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

namespace aspnetmvc.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);
        }
    }
}
```

Ajax Example - 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>
```

Ajax Example - Search.cshtml, _SearchResult.cshtml

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>
}
```

_SearchResult.cshtml

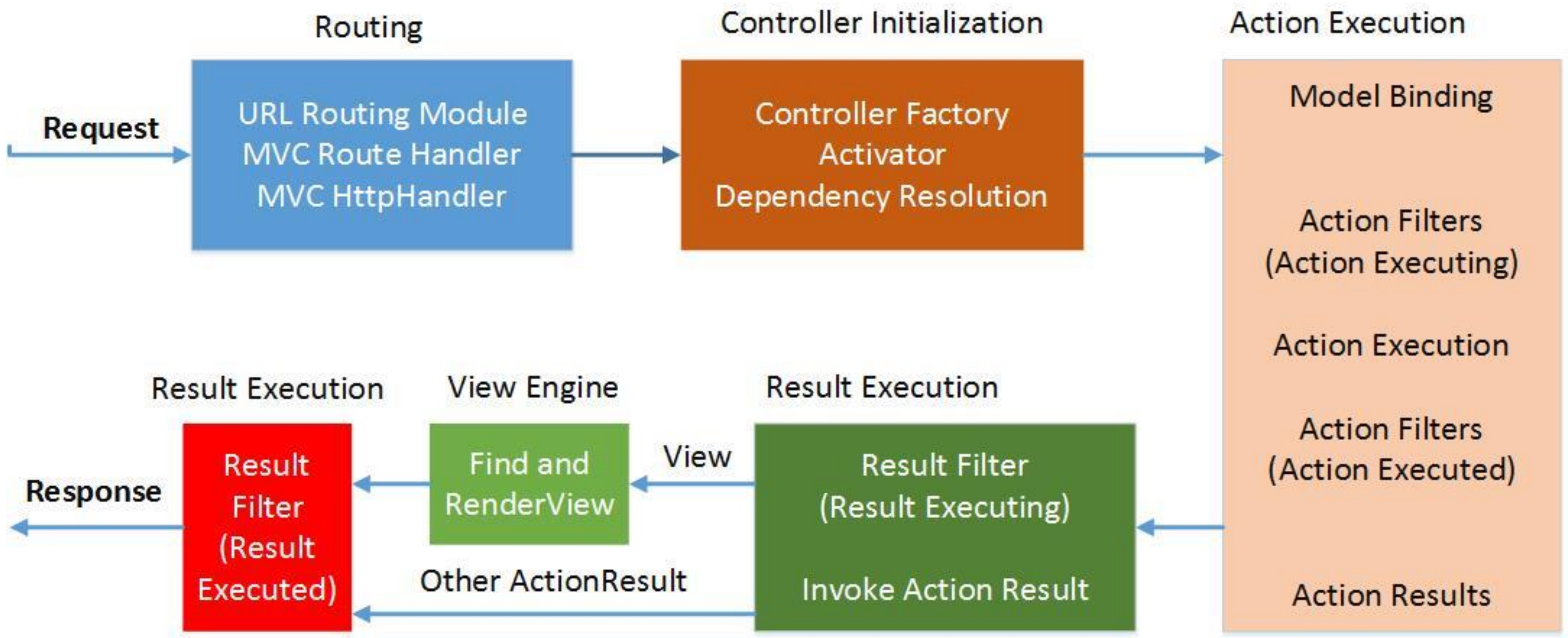
```
@model String

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

MVC Request Life Cycle

- 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.
- 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 a 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.

MVC Request Life Cycle



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.

Filters

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

Filter	Description
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.
ValidateAntiForgeryToken	Used to prevent forgery of a request.

ActionFilterAttribute class implements IActionFilter, IResultFilter interfaces and inherits FilterAttribute class

Creating Filter

- Create an attribute class that derives from **ActionFilterAttribute** and apply the attribute to a controller or an action method.
- Override one or more of the controller's **On<Filter>** methods.
- Register a global filter using the **GlobalFilterCollection** class in **FilterConfig.cs**.
- 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.

Order of Filters

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.

Creating Filter

```
public class LogFilter : System.Web.Mvc.ActionFilterAttribute {
    public override void OnActionExecuting(ActionExecutingContext filterContext) {
       base.OnActionExecuting(filterContext);
       HttpContext.Current.Response.Write("About to execute action : " +
                                 filterContext.ActionDescriptor.ActionName);
    }
}
```

```
public class TestController : Controller {
        public ActionResult Index(){
             return View();
        }
        [LogFilter]
        public ActionResult About() {
            HttpContext.Current.Response.Write("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 **customErrors** page in web.config
- ❑ Use **Application_Error** in Global.asax

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

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

Custom Error Page

```
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>
```

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

HandleError Attribute

We can use **[HandleError]** attribute with Controller or Actions.

```
// TestController.cs
[HandleError]    // Error in this action will display custom error page
public string Error(string input)
{
   int num = Int32.Parse(input);
   return "Testing..";
}
```

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.

Data Caching

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

OutputCache Properties

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.

OutputCache - Example

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");
  }
}
```

web.config

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

Index.cshtml

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

Data Caching – Example – DataCacheController.cs

```
public class DataCacheController : Controller{
 private List<Book> GetBooks(){
   List<Book> books = HttpContext.Cache["books"] as List<Book>;
   if (books == null) {
    BooksContext ctx = new BooksContext();  // EF Context class
    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);
   }
 }
```

Data Caching – Example – Books.cshtml

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

Data Caching – Example – List.cshtml

```
@model IEnumerable<mvcdemo.Models.Book>
@{ Layout = null; }
<!DOCTYPE html>
<html>
<head>
     <title>List of Books</title>
</head>
<body>
    <h1>Titles</h1>
    <ul>
        @foreach (var item in Model)
        {
            <li>@item.Title</li>
        }
    </ul>
    <p></p>
    @ViewBag.Message
</body>
</html>
```