

ASP.NET Web Forms.....	2
Difference between Asp.Net Web Forms and Asp.Net MVC.....	2
Advantages of ASP.Net MVC and Web Forms.....	2
What is MVC and MVC Architecture	3
Asp.Net MVC Flow.....	3
Benefits of ASP.NET MVC flow.....	3
Different version of MVCs and comparison	3
Difference between Razor Engine and Web Form Engine	4
Models	5
ViewModels.....	5
Controller.....	8
Controller Actions.....	8
ActionResult	9
ActionResult Types.....	9
Controller Base class methods:.....	11
Views.....	11
Implementing view in three ways (jquery, aspx, razor)	12
Types of Views.....	12
Difference between ViewData, ViewBag, TempData, Session	13
HTML Helpers	14
Difference between @Html.TextBox and @Html.TextBoxFor	15
Validation	16
Routing.....	24
State Management.....	25

ASP.NET Web Forms

Web Forms is the oldest ASP.NET programming model, with event driven web pages written as a combination of HTML, server controls, and server code.

Web Forms are compiled and executed on the server, which generates the HTML that displays the web pages.

Web Forms comes with hundreds of different web controls and web components to build user-driven web sites with data access.

Difference between Asp.Net Web Forms and Asp.Net MVC

ASP.NET Web Forms	ASP.NET MVC
Asp.Net Web Form follow a traditional event driven development model.	Asp.Net MVC is a lightweight and follow MVC (Model, View, Controller) pattern based development model.
Asp.Net Web Form has server controls.	Asp.Net MVC has html helpers.
Asp.Net Web Form has state management (like as view state, session) techniques.	Asp.Net MVC has no automatic state management techniques.
Asp.Net Web Form has file-based URLs means file name exist in the URLs must have its physically existence.	Asp.Net MVC has route-based URLs means URLs are divided into controllers and actions and moreover it is based on controller not on physical file.
Asp.Net Web Form follows Web Forms Syntax	Asp.Net MVC follow customizable syntax (Razor as default)
In Asp.Net Web Form, Web Forms(ASPX) i.e. views are tightly coupled to Code behind(ASPX.CS) i.e. logic.	In Asp.Net MVC, Views and logic are kept separately.
Asp.Net Web Form has Master Pages for consistent look and feels.	Asp.Net MVC has Layouts for consistent look and feels
Asp.Net Web Form has User Controls for code re-usability.	Asp.Net MVC has Partial Views for code re-usability.
Asp.Net Web Form has built-in data controls and best for rapid development with powerful data access.	Asp.Net MVC is lightweight, provide full control over markup and support many features that allow fast & agile development. Hence it is best for developing interactive web application with latest web standards.
Asp.Net Web Form is not Open Source.	Asp.Net Web MVC is an Open Source.

Advantages of ASP.Net MVC and Web Forms

The main advantages of **ASP.net MVC** are:

- Enables the full control over the rendered HTML.
- Provides clean separation of concerns(SoC).
- Enables Test Driven Development (TDD).

- Easy integration with JavaScript frameworks.
- Following the design of stateless nature of the web.
- RESTful urls that enables SEO.(Search Engine Optimization)
- No ViewState andPostBack events

The main advantage of **ASP.net Web Form** are:

- It provides RAD(Rapid Application Development) development
- Easy development model for developers those coming from winform development.

What is MVC and MVC Architecture

The ASP.NET MVC framework is a lightweight, highly testable presentation framework that (as with Web Forms-based applications) is integrated with existing ASP.NET features, such as master pages and membership-based authentication. The MVC framework is defined in theSystem.Web.Mvc assembly.

What is MVC and MVC architecture from book.

Asp.Net MVC Flow

From book.

Benefits of ASP.NET MVC flow

- Enables the full control over the rendered HTML.
- Provides clean separation of concerns(SoC).
- Enables Test Driven Development (TDD).
- Easy integration with JavaScript frameworks.
- Following the design of stateless nature of the web.
- RESTful urls that enables SEO.(Search Engine Optimization)
- No ViewState andPostBack events

Different version of MVCs and comparison

Category	MVC2	MVC3	MVC4
View Engine	only Web Forms view engine (.aspx)	Razor View Engine (.cshtml for c# and .vbhtml for Visual Basic) and Web Forms view engine (.aspx), support for	also uses Razor View Engine as a default view engine with some new features like condition

		multiple view engines	attribute and 'Tilde slash'
Chart, WebGrid, Crypto, WebImage, WebMail Controls	not available	Available	Available
Syntax	Web Forms view engine syntax: <%=Html code %>	Razor View Engine syntax: @Html code	same Razor View Engine Syntax but with the addition of new features like conditional attribute and 'Tilde Slash'
Objects available for sharing of data between View and Controller	TempData, ViewData	TempData, ViewData, ViewBag	TempData, ViewData, ViewBag
Jquery Support	Good	Better	Better with JQuery Mobile support
Layout Support	Supports only Master Page	Supports not only Master Page but also Layout Page	Supports not only Master Page but also Layout Page
Validation	Client-side Validation and Asynchronous controllers	Unobtrusive Ajax and Client side Validation , JQuery Validation and JSON binding support	Client side validation, JQuery validation and enhanced support for asynchronous methods
Features Added		templates enabled by HTML 5	new features for mobile apps, asp.net web api

- MVC2 supports on web form engine while MVC3 and MVC4 support multiple view engine like web forms, razor etc. MVC4 introduced conditional attributes and Tilde Slash.
- Chart, webgrid, webmail controls are not available in MVC3, while it is available in MVC4.
- Syntax followed in MVC2 is <%: Html.ActionLink("SignUp", "SignUp") %> and in MVC3 and MVC4 it is @Html.ActionLink("SignUp", "SignUp")
- MVC2 only have tempdata and viewdata while mvc3, mvc4 have tempdata, viewdata, viewbag.
- MVC2 supports for Master Page only while MVC3 and MVC4 supports for Master page and layout page.
- MVC3 provides support for Templates enabled by HTML5 while MVC4 supports for mobile apps and web api.

Difference between Razor Engine and Web Form Engine

Razor Engine	Web Form Engine
--------------	-----------------

The namespace for Razor Engine is System.Web.Razor	The namespace for Webform Engine is System.Web.Mvc.WebFormViewEngine
File extension has .cshtml (Razor with C#) or .vbhtml (Razor with VB) extension for views, partial views, editor templates and for layout pages.	File extension has .aspx extension for views, .ascx extension for partial views & editor templates and .master extension for layout/master pages.
Syntax @Html.ActionLink("SignUp", "SignUp")	Syntax <%: Html.ActionLink("SignUp", "SignUp") %>
Razor Engine is little bit slow as compared to Webform Engine.	Web Form Engine is faster than Razor Engine.
Razor Engine, doesn't support design mode in visual studio	Web Form engine support design mode in visual studio
Razor Engine support TDD (Test Driven Development) since it is not depend on System.Web.UI.Page class	Web Form Engine doesn't support TDD (Test Driven Development) since it depend on System.Web.UI.Pageclass which makes the testing complex

Models

The model should contain all of the application business logic, validation logic, and database access logic.

ASP.NET MVC is compatible with any data access technology (for example LINQ to SQL)

All .edmx files, .dbml files etc. are located in the Models folder.

ViewModels

ViewModel contain fields that are represented in the view (for LabelFor,EditorFor,DisplayFor helpers)

ViewModel can have specific validation rules using data annotations or IDataErrorInfo.

ViewModel can have multiple entities or objects from different data models or data source.

In ViewModel put only those fields/data that you want to display on the view/page.

Since view represents the properties of the ViewModel, hence it is easy for rendering and maintenance.

Use a mapper when ViewModel become more complex.

Designing ViewModel

```
1. public class UserLoginViewModel
```

```

2. {
3.     [Required(ErrorMessage = "Please enter your username")]
4.     [Display(Name = "User Name")]
5.     [MaxLength(50)]
6.     public string UserName { get; set; }
7.     [Required(ErrorMessage = "Please enter your password")]
8.     [Display(Name = "Password")]
9.     [MaxLength(50)]
10.    public string Password { get; set; }
11. }

```

Presenting the viewmodel in the view

```

1. @model MyModels.UserLoginViewModel
2. @{
3.     ViewBag.Title = "User Login";
4.     Layout = "~/Views/Shared/_Layout.cshtml";
5. }
6. @using (Html.BeginForm())
7. {
8.     <div class="editor-label">
9.         @Html.LabelFor(m => m.UserName)
10.    </div>
11.    <div class="editor-field">
12.        @Html.TextBoxFor(m => m.UserName)
13.        @Html.ValidationMessageFor(m => m.UserName)
14.    </div>
15.    <div class="editor-label">
16.        @Html.LabelFor(m => m.Password)
17.    </div>
18.    <div class="editor-field">
19.        @Html.PasswordFor(m => m.Password)
20.        @Html.ValidationMessageFor(m => m.Password)
21.    </div>
22.    <p>
23.        <input type="submit" value="Log In" />

```

```
24. </p>
25. </div>
26. }
```

Working with Action

```
1.  public ActionResult Login()
2.  {
3.      return View();
4.  }
5.  [HttpPost]
6.  public ActionResult Login(UserLoginViewModel user)
7.  {
8.      // To acces data using LINQ
9.      DataClassesDataContext mobjentity = new DataClassesDataContext();
10.     if (ModelState.IsValid)
11.     {
12.         try
13.         {
14.             var q = mobjentity.tblUsers.Where(m => m.UserName == user.UserName &&
15.                 m.Password == user.Password).ToList();
16.             if (q.Count > 0)
17.             {
18.                 return RedirectToAction("MyAccount");
19.             }
20.             else
21.             {
22.                 ModelState.AddModelError("", "The user name or password provided is
23.                     incorrect.");
24.             }
25.         }
26.         catch (Exception ex)
27.         {
28.             return View(user);
29.         }
30.     }
31. }
```

```
29. }
```

Controller

- ◆ It is a class
- ◆ Derives from the base `System.Web.Mvc.Controller` class
- ◆ Generates the response to the browser request
- ◆ Locating the appropriate action method to call and validating that it can be called.
- ◆ Getting the values to use as the action method's arguments.
- ◆ Handling all errors that might occur during the execution of the action method.

```
[HandleError]
public class HomeController : Controller
{
    public ActionResult Index()
    {
        ViewData["Message"] = "Welcome to ASP.NET MVC!";

        return View();
    }

    public ActionResult About()
    {
        return View();
    }
}
```

Controller Actions

1. Public method of the Controller class
2. Cannot be overloaded
3. Cannot be a static method
4. Returns action result

```
public ActionResult About()
{
    return View();
}
```


ActionResult

An ActionResult is a return type of a controller method in MVC. Action methods help us to return models to views, file streams, and also redirect to another controller's Action method.

There are many derived ActionResult types in MVC that you may use to return the results of a controller method; these are more specific for a particular view. All Action Result Classes are derived from "ActionResult". In other words ActionResult is an abstract class that has several subtypes.

ActionResult Types

ViewResult: It renders a specified view to the response stream. The "**ViewResult**" class inherits from the abstract class "ViewResultBase" and is used to render a view. **This class contains methods for finding the view to render and also for executing the result.** This class also contains properties that identify the view (view Name) to render for the application. It is in the ViewResultBase abstract base class that we get access to all of our familiar data objects like: TempData, ViewData, and ViewBag.

If you want an action method to result in a rendered view, the action method should return a call to the controller's View helper method. The View helper method passes a ViewResult object to the ASP.NET MVC framework, which calls the object's ExecuteResult method.

```
public ViewResult ViewResultTest()
{
    return View("ViewResultTest");
}
```

PartialViewResult: The PartialViewResult class is inherited from the abstract "ViewResultBase" class and is used to render a partial view. **This class contains methods for finding the partial view to render and also for executing the result.** This class also contains properties that identify the partial view (View Name) to render for the application. 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.

```
public PartialViewResult PartialViewResultTest()
```

```
{  
    return PartialView("PartialViewResult");  
}
```

JsonResult: Represents a class that is used to send JSON-formatted content to the response. The JsonResult object that serializes the specified object to JSON format.

```
public JsonResult JsonResultTest()  
{  
    return Json("Hello My Friend!");  
}
```

ContentResult: The ContentResult may be used to return to an action as plain text.

```
public ActionResult ContentResultTest()  
{  
    return Content("Hello My Friend!");  
}  
  
public ContentResult ContentResultTest()  
{  
    return Content("Hello My Friend!");  
}
```

RedirectResult: It is used to perform an HTTP redirect to a given URL.

```
public ActionResult RedirectResultTest()  
{  
    return Redirect("http://www.google.com/");  
}
```

RedirectToRouteResult: RedirectToResult is used to redirect by using the specified route values dictionary

```
public ActionResult RedirectToRouteResultTest(int? Id)  
{  
    return new RedirectToRouteResult(new System.Web.Routing.RouteValueDictionary(new {  
        controller = "Home", action = "List", Id = new int?() }));  
}
```

JavaScriptResult - Sends JavaScript content to the response.

FileContentResult - Represents a class that is used to send binary file content to the response

```
public ActionResult GetImage(int ID){  
    byte[] imageData = ImageRepository.GetImage(ID);  
    return File(imageData, "image/jpeg", "fileName.jpg");  
}
```

FileStreamResult - Sends binary content to the response by using a Stream instance.

FilePathResult - Sends the contents of a file to the response

EmptyResult: Sends empty response to the response.

Controller Base class methods:

View - Creates a ViewResult object that renders a view to the response.

Redirect - Creates a RedirectResult object that redirects to the specified URL.

RedirectToAction - Redirects to the specified action using the action name

```
{controller}/{action}/{id}  
RedirectToAction("profile", "person", new { personID = Person.personID});
```

Json - Creates a JsonResult object that serializes the specified object to JavaScript Object Notation (JSON) format.

JavaScript - Creates a JavaScriptResult object

Content - Creates a content result object by using a string.

File - Creates a FileContentResult object by using the file contents and file type.

Views

- Most of the Controller Actions return views
- The path to the view is inferred from the name of the controller and the name of the controller action.

\Views\ControllerName\ControllerAction.aspx

- A view is a standard (X)HTML document that can contain scripts.
- script delimiters <% and %> in the views

Implementing view in three ways (jquery, aspx, razor)

MyTutorial Folder

Types of Views

The primary difference is that a **Partial View** is designed to be rendered within other full Views and as a result it will not contain all of the markup that a full View might (*since it is designed to work within other Views*).

For example, a normal View might look like this and contain all of the elements present in a traditional HTML page :

```
<!DOCTYPE html>
<html>
<head>
<meta charset=utf-8 />
<title>Example</title>
</head>
<body>
  <!-- Your Content Here -->
</body>
</html>
```

where as a Partial View might only contain a small section that it needs to render and since it should be displayed within an existing View, it won't require the necessary tags like <head>, <html>, etc. because it is implied that these should already be present when a Partial View is rendered. An example partial View might look like this :

```
<div>
  <!-- Example -->
  <ul>
    <li>List Item</li>
  </ul>
</div>
```

Dynamic View:

- View where we use dynamic instead of any model.
- During runtime Model is binded to the view.
- Intellisense supports is not available while coding.

For example:

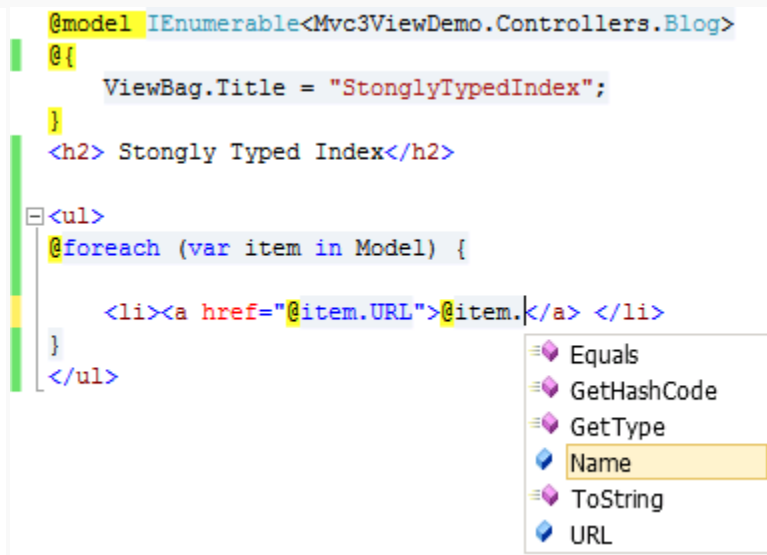
@model dynamic

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

< p >
< ul >
@foreach (var blog in Model) {
    < li >
        < a href="@blog.URL">@blog.Name< /a >
    < /li >
}
< /ul >
< /p >
```

Strongly Typed View:

- View are strongly binded with the Model.
- Intellisense supports is available while coding.



```
@model IEnumerable<Mvc3ViewDemo.Controllers.Blog>
@{
    ViewBag.Title = "StonglyTypedIndex";
}
<h2> Stongly Typed Index</h2>
<ul>
    @foreach (var item in Model) {
        <li><a href="@item.URL">@item.</a> </li>
    }
</ul>
```

The screenshot shows a code editor with a dropdown menu open for the property access in the line `@item.`. The dropdown lists several properties: Equals, GetHashCode, GetType, Name (which is highlighted), ToString, and URL.

Difference between ViewData, ViewBag, TempData, Session

ViewData

- ViewData is a dictionary object that is derived from ViewDataDictionary class.
- ViewData is used to pass data from controller to corresponding view.

- It's life lies only during the current request.
- If redirection occurs then it's value becomes null.
- It's required typecasting for getting data and check for null values to avoid error.

ViewBag

- ViewBag is a dynamic property that takes advantage of the new dynamic features in C# 4.0.
- Basically it is a wrapper around the ViewData and also used to pass data from controller to corresponding view.
- It's life also lies only during the current request.
- If redirection occurs then it's value becomes null.
- It doesn't required typecasting for getting data.

TempData

- TempData is a dictionary object that is derived from TempDataDictionary class and stored in short lives session.
- TempData is used to pass data from current request to subsequent request (means redirecting from one page to another).
- It's life is very short and lies only till the target view is fully loaded.
- It's required typecasting for getting data and check for null values to avoid error.
- It is used to store only one time messages like error messages, validation messages. To persist data with TempData refer this article: Persisting Data with TempData

Session

- Session is also used to pass data within the ASP.NET MVC application and Unlike TempData, it never expires.
- Session is valid for all requests, not for a single redirect.
- It's also required typecasting for getting data and check for null values to avoid error.

HTML Helpers

- Methods which typically return string.
 - Used to generate standard HTML elements
 - textboxes, dropdown lists, links etc.
 - Example: Html.TextBox() method
 - Usage is optional
 - You can create your own HTML Helpers
-
- BeginForm()
 - EndForm()
 - TextArea()
 - TextBox()
 - CheckBox()

- RadioButton()
- ListBox()
- DropDownList()
- Hidden()
- Password()

ASP.NET Syntax C#:

```
<%= Html.ValidationSummary("Create was unsuccessful. Please correct
the errors and try again.") %>
<% using (Html.BeginForm()) {%>
<p>
<label for="FirstName">First Name:</label>
<%= Html.TextBox("FirstName") %>
<%= Html.ValidationMessage("FirstName", "*") %>
</p>
<p>
<label for="LastName">Last Name:</label>
<%= Html.TextBox("LastName") %>
<%= Html.ValidationMessage("LastName", "*") %>
</p>
<p>
<label for="Password">Password:</label>
<%= Html.Password("Password") %>
<%= Html.ValidationMessage("Password", "*") %>
</p>
<p>
<label for="Password">Confirm Password:</label>
<%= Html.Password("ConfirmPassword") %>
<%= Html.ValidationMessage("ConfirmPassword", "*") %>
</p>
<p>
<label for="Profile">Profile:</label>
<%= Html.TextArea("Profile", new {cols=60, rows=10}) %>
</p>
<p>
<%= Html.CheckBox("ReceiveNewsletter") %>
<label for="ReceiveNewsletter" style="display:inline">Receive
Newsletter?</label>
</p>
<p>
<input type="submit" value="Register" />
</p>
<%}%>
```

Difference between @Html.TextBox and @Html.TextBoxFor

Ultimately they both produce the same HTML but [Html.TextBoxFor\(\)](#) is strongly typed where as [Html.TextBox](#) isn't.

```
1: @Html.TextBox("Name")
```

```
2: Html.TextBoxFor(m => m.Name)
```

will both produce

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

Html.TextBox is not strongly typed and it doesn't require a strongly typed view meaning that you can hardcode whatever name you want as first argument and provide it a value:

```
<%= Html.TextBox("foo", "some value") %>
```

You can set some value in the ViewData dictionary inside the controller action and the helper will use this value when rendering the textbox (ViewData["foo"] = "bar").

Html.TextBoxFor is requires a strongly typed view and uses the view model:

```
<%= Html.TextBoxFor(x => x.Foo) %>
```

The helper will use the lambda expression to infer the name and the value of the view model passed to the view.

And because it is a good practice to use strongly typed views and view models you should always use the `Html.TextBoxFor` helper.

Validation

The ASP.NET MVC3 comes with a validation feature that not only supports both server side and client side validation, but also hides all validation details to have a very clean controller code and HTML markup.

Validation Attribute on Model Class

- The **Required** attribute is used on property **UserName**, **Email** and **Password** to mark them as required.
- The **Display** attribute is used on all properties to give them a display name as field label or in error message.
- The **DataType** attribute is used on property **Email** and **Password** to indicate type of property.
- The **ValidationPasswordLength** attribute is a custom validation attribute. I will talk more about it later.
- The **Compare** attribute is used on **ConfirmPassword** to compare **Password** with **ConfirmPassword**.

The general purpose validation attributes are defined in `System.ComponentModel.DataAnnotations` namespace (*System.ComponentModel.DataAnnotations.dll*). This

includes Required attribute, Range attribute, RegularExpression attribute, StringLength attribute, etc. They all inherit from ValidationAttribute base class and override IsValid method to provide their specific validation logic. DisplayAttribute is also in `System.ComponentModel.DataAnnotations` namespace, but it's a display attribute instead of validation attribute. DataTypeAttribute is a validation attribute, but is classified as display attribute in MSDN. FYI, in `System.ComponentModel.DataAnnotations` namespace, there are Data Modeling attributes, AssociationAttribute, KeyAttribute, etc. designed for Entity Framework.


```

public class RegisterModel
{
    [Required]
    [Display(Name = "User name")]
    public string UserName { get; set; }

    [Required]
    [DataType(DataType.EmailAddress)]
    [Display(Name = "Email address")]
    public string Email { get; set; }

    [Required]
    [ValidatePasswordLength]
    [DataType(DataType.Password)]
    [Display(Name = "Password")]
    public string Password { get; set; }

    [DataType(DataType.Password)]
    [Display(Name = "Confirm password")]
    [Compare("Password", ErrorMessage = "The password and confirmation password
        do not match.")]
    public string ConfirmPassword { get; set; }
}

```

Validation attribute provided by ASP.NET MVC is **RemoteAttribute** that uses Ajax call to service side controller action to do validation.

To make validation attribute uses client side validation, it has to implement **IClientValidatable** interface, the **IClientValidatable** has only one method **GetClientValidationRule** that has the following signature:

```

IEnumerable<modelclientvalidationrule> GetClientValidationRules
(ModelMetadata metadata, ControllerContext context);
</modelclientvalidationrule>

```

- **ModelMetadata** is a container for common metadata. It allows classes to utilize model information when doing validation
- **ControllerContext** is a container for HTTP request and other request environment data.
- **ModelClientValidationRule** is a base class for client validation rule that is sent to the browser. There are six built-in validation rules in MVC: **ModelClientValidationEqualToRule**, **ModelClientValidationRemoteRule**, **ModelClientValidationRequiredRule**, **ModelClientValidationRangeRule**, **ModelClientValidationStringLengthRule**, **ModelClientValidationRegexRule**. If you pay attention, you can see all general purpose validation attributes in **System.ComponentModel.DataAnnotations** have a corresponding **ModelClientValidationRule** in here. The ASP.NET MVC creates adapter, e.g. **RequiredAttributeAdapter**, to extend general purpose validation attribute to support ASP.NET MVC validation design

Creating Custom Validation

`ValidatePasswordLengthAttribute` is a custom validation that inherits from `ValidateAttribute` and implements `IClientValidatable`.

```
[AttributeUsage(AttributeTargets.Field | AttributeTargets.Property,
    AllowMultiple = false, Inherited = true)]
public sealed class ValidatePasswordLengthAttribute : ValidationAttribute,
    IClientValidatable
{
    private const string _defaultErrorMessage = "'{0}'
        must be at least {1} characters long.";
    private readonly int _minCharacters =
        Membership.Provider.MinRequiredPasswordLength;

    public ValidatePasswordLengthAttribute()
        : base(_defaultErrorMessage)
    {
    }

    public override string FormatErrorMessage(string name)
    {
        return String.Format(CultureInfo.CurrentCulture, ErrorMessageString,
            name, _minCharacters);
    }

    public override bool IsValid(object value)
    {
        string valueAsString = value as string;
        return (valueAsString != null && valueAsString.Length >= _minCharacters);
    }

    public IEnumerable<ModelClientValidationRule>
    GetClientValidationRules(ModelMetadata metadata, ControllerContext context)
    {
        return new[]{
            new ModelClientValidationStringLengthRule(FormatErrorMessage
                (metadata.GetDisplayName()), _minCharacters, int.MaxValue)
        };
    }
}
</modelclientvalidationrule>
```

Server Side Validation:

The **ModelValidator** calls each validation attribute class to validate the model data based on a given setting. The validation results (**ModelValidationResult**) are stored in **ModelState** to be used in action or view.

```
[HttpPost]
public ActionResult Register(RegisterModel model)
{
    if (ModelState.IsValid)
    {
        // Attempt to register the user
    }
}
```

```

        MembershipCreateStatus createStatus = MembershipService.CreateUser
            (model.UserName, model.Password, model.Email);

        if (createStatus == MembershipCreateStatus.Success)
        {
            FormsService.SignIn(model.UserName, false /* createPersistentCookie */);
            return RedirectToAction("Index", "Home");
        }
        else
        {
            ModelState.AddModelError("",
                AccountValidation.ErrorCodeToString(createStatus));
        }
    }

    // If we got this far, something failed, redisplay form
    ViewBag.PasswordLength = MembershipService.MinPasswordLength;
    return View(model);
}

```

```

@using (Html.BeginForm())
{
    @Html.ValidationSummary()

    Start date (MM/dd/yyyy HH:mm:ss AM/PM) *: @Html.TextBoxFor(x => x.StartDate, new
{ size = 25 })

    Duration (Hours) *: @Html.DropDownListFor(x => x.DurationInHours, new[]{
        new SelectListItem{ Text = "1", Value = "1"},
        new SelectListItem{ Text = "2", Value = "2"},
        new SelectListItem{ Text = "3", Value = "3"},
        new SelectListItem{ Text = "4", Value = "4"},
        new SelectListItem{ Text = "5", Value = "5"}
    }, "Select the duration", new { style = "width:180px" })

    No. of joinees *: @Html.TextBoxFor(x => x.NoOfJoinees, new { size = 5 })

    Drinks? @Html.CheckBoxFor(x => x.Drinks)

    <input type="submit" value="Host the party!" />
}

```

The `@Html.ValidationSummary()` method that we used right below the form declaration in the view helps us to show all the validation errors of a model returned from the controller.

Client Side Validation:

To enable client side validation we just have to do following steps:

5. The client side validation is enabled by default in `web.config`.

```
<appSettings>
```

```
<add key="ClientValidationEnabled" value="true"/>
<add key="UnobtrusiveJavaScriptEnabled" value="true"/>
</appSettings>
```

6. make sure *jquery.validate.min.js* and *jquery.validate.unobtrusive.min.js* are added in the view page also.

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

Now, client side validation is enabled.

The extension methods for **HtmlHelper** class are required for validation: **Validate**, **ValidateFor**, **ValidationMessage**, **ValidationMessageFor**, **ValidationSummary**.

The *jquery.validate.min.js* is standard jQuery validation library.

The *jquery.validate.unobtrusive.min.js* is an ASP.NET MVC client side validation library that is built on top of jQuery validation library. It uses HTML element attributes to store validation information. This design is very clean and intrusive to the UI designer

Remote Validation Attribute in MVC

Add Remote attribute on **UserName** of **LogOnModel**.






```
[Required]
[Display(Name = "User name")]
[Remote("DisallowName", "Account")]
public string UserName { get; set; }
```







The first parameter is **Action** name, the second parameter is **Controller** name.





Create a **DisallowName** action in **AccountController**.








```
public ActionResult DisallowName(string UserName)
{
    if (UserName != "Bin")
    {
        return Json(true, JsonRequestBehavior.AllowGet);
    }

    return Json(string.Format("{0} is invalid", UserName),
        JsonRequestBehavior.AllowGet);
}
```

	Class	Description
	AssociatedMetadataTypeTypeDescriptionProvider	Extends the metadata information for a class by adding attributes and property information that is defined in an associated class.
	AssociationAttribute	Specifies that an entity member represents a data relationship, such as a foreign key relationship.
	BindableTypeAttribute	Specifies whether a type is typically used for binding.
	CompareAttribute	Provides an attribute that compares two properties.
	ConcurrencyCheckAttribute	Specifies that a property participates in optimistic concurrency checks.
	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.
	DisplayColumnAttribute	Specifies the column that is displayed in the referred table as a foreign-key column.
	DisplayFormatAttribute	Specifies how data fields are displayed and formatted by ASP.NET Dynamic Data.
	EditableAttribute	Indicates whether a data field is editable.
	EmailAddressAttribute	Validates an email address.
	EnumDataTypeAttribute	Enables a .NET Framework enumeration to be mapped to a data column.
	FileExtensionsAttribute	Validates file name extensions.
	FilterUIHintAttribute	Represents an attribute that is used to specify the filtering behavior for a column.
	KeyAttribute	Denotes one or more properties that uniquely identify an entity.
	MaxLengthAttribute	Specifies the maximum length of array or string data allowed in a property.

	MetadataTypeAttribute	Specifies the metadata class to associate with a data model class.
	MinLengthAttribute	Specifies the minimum length of array or 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.
	ScaffoldColumnAttribute	Specifies whether a class or data column uses scaffolding.
	ScaffoldTableAttribute	Specifies whether a class or data table uses scaffolding.
	StringLengthAttribute	Specifies the minimum and maximum length of characters that are allowed in a data field.

	TimestampAttribute	Specifies the data type of the column as a row version.
	UIHintAttribute	Specifies the template or user control that Dynamic Data uses to display a data field.
	UrlAttribute	Provides URL validation.
	ValidationAttribute	Serves as the base class for all validation attributes.
	ValidationContext	Describes the context in which a validation check is performed.
	ValidationException	Represents the exception that occurs during validation of a data field when the ValidationAttribute class is used.
	ValidationResult	Represents a container for the results of a validation request.
	Validator	Defines a helper class that can be used to validate objects, properties, and methods when it is included in their associated ValidationAttribute attributes.

Routing

The Routing module is responsible for mapping incoming browser requests to particular MVC controller actions.

Two places to setup:

7. Web.config file
8. Global.asax file

Web.config

There are four sections in the configuration file that are relevant to routing: the system.web.httpModules section, the system.web.httpHandlers section, the system.webserver.modules section, and the system.webserver.handlers section.

Global.asax

A route table is created in the application's Global.asax file. The Global.asax file is a special file that contains event handlers for ASP.NET application lifecycle events. The route table is created during the Application Start event.

```
public static void RegisterRoutes(RouteCollection routes)
{
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
    routes.MapRoute(
        "Blog",                      // Route name
        "Archive/{entryDate}",       // URL with parameters
        new { controller = "Archive", action = "Entry" } // Parameter defaults
    );
    routes.MapRoute(
        "Default",                   // Route name
        "{controller}/{action}/{id}", // URL with parameters
        new { controller = "Home", action = "Index", id = "" } // Parameter defaults
    );
}
```

State Management

ViewData, ViewBag, TempData, Application, Session