# ASP.NET MVC - Helpers

In ASP.Net web forms, developers are using the toolbox for adding controls on any particular page. However, in ASP.NET MVC application there is no toolbox available to drag and drop HTML controls on the view. In ASP.NET MVC application, if you want to create a view it should contain HTML code. So those developers who are new to MVC especially with web forms background finds this a little hard.

To overcome this problem, ASP.NET MVC provides HtmlHelper class which contains different methods that help you create HTML controls programmatically. All HtmlHelper methods generate HTML and return the result as a string. The final HTML is generated at runtime by these functions. The HtmlHelper class is designed to generate UI and it should not be used in controllers or models.

There are different types of helper methods.

* **Createinputs** − Creates inputs for text boxes and buttons.
* **Createlinks** − Creates links that are based on information from the routing tables.
* **Createforms** − Create form tags that can post back to our action, or to post back to an action on a different controller.

Given below is the list of methods in HtmlHelper class.

This HTML is a property that we inherit from the ViewPage base class. So, it's available in all of our views and it returns an instance of a type called HTML Helper.

Let’s take a look at a simple example in which we will enable the user to edit the employee. Hence, this edit action will be using significant numbers of different HTML Helpers.

As you can see that there are many helper methods used. So, here “HTML.BeginForm” writes an opening Form Tag. It also ensures that the method is going to be “Post”, when the user clicks on the “Save” button.

Html.BeginForm is very useful, because it enables you to change the URL, change the method, etc.

@HTML.HiddenFor”, which emits the hidden field.

MVC Framework is smart enough to figure out that this ID field is mentioned in the model class and hence it needs to be prevented from getting edited, that is why it is marked as hidden.

The Html.LabelFor HTML Helper creates the labels on the screen. The Html.ValidationMessageFor helper displays proper error message if anything is wrongly entered while making the change.

We also need to change the Edit action for POST because once you update the employee then it will call this action.

// POST: Employee/Edit/5

[HttpPost]

public ActionResult Edit(int id, FormCollection collection){

try{

var employee = empList.Single(m => m.ID == id);

if (TryUpdateModel(employee)){

//To Do:- database code

return RedirectToAction("Index");

}

return View(employee);

}catch{

return View();

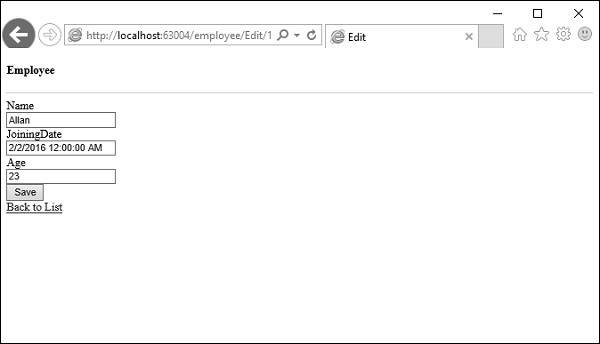
}

}

Let’s run this application and request for the following URL **http://localhost:63004/employee**. You will receive the following output.



Click on the edit link on any particular employee, let’s say click on Allan edit link. You will see the following view.



Let’s change the age from 23 to 29 and click ‘Save’ button, then you will see the updated age on the Index View.



Using the HTML Helper class, we can create HTML Controls programmatically. HTML Helpers are used in View to render HTML content. HTML Helpers (mostly) is a method that returns a string. It is not mandatory to use HTML Helper classes for building an ASP.NET MVC application. We can build an ASP.NET MVC application without using them, but HTML Helpers helps in the rapid development of a view. HTML Helpers are more lightweight as compared to ASP.NET Web Form controls as they do not use ViewState and do not have event models.

HTML Helpers are categorized into three types:

1. Inline HTML Helpers
2. Built-in HTML Helpers
3. Custom HTML Helpers

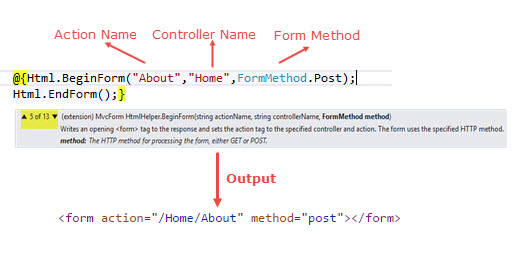
Built-in HTML Helpers are further divided into three categories:

1. Standard HTML Helpers
2. Strongly Typed HTML Helpers
3. Templated HTML Helpers

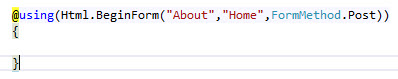
**1. Standard HTML Helpers**

Standard HTML Helpers are used to render the most common type of HTML controls like TextBox, DropDown, Radio buttons, Checkbox etc. Extension methods of HTML Helper classes have several overloaded versions. We can use any one according to our requirement. Let’s see some of the Standard HTML Helpers:

**Form:** For creating the Form element, we can use BeginForm() and EndForm() extension method.

[](https://4.bp.blogspot.com/-IvYw2Rinblk/V0NFzsOuQuI/AAAAAAAABIE/3HfNMxEj5UILXtrp0dReNx2Bu9_3bFR6QCLcB/s1600/image1.jpg)

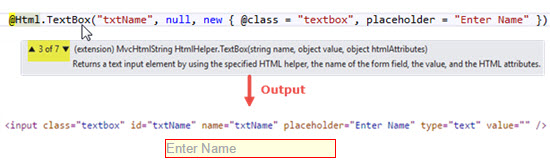
The BeginForm helper implements the IDisposable interface, which enables us to use the **using** keyword.

[](https://1.bp.blogspot.com/-O2ryd5F3Ekw/V0NGTeFsdWI/AAAAAAAABII/VeQoCHtCiCUosi7B2MqbCO3qAIeXVwc9gCLcB/s1600/image2.jpg)

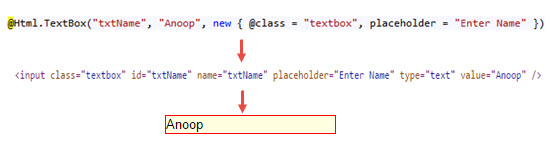
**Label:** The Label method of HTML helper can use for generating label element. Label extension method have 6 overloaded versions.

[](https://2.bp.blogspot.com/-bsJv_0Al_0M/V0NGvWEXL9I/AAAAAAAABIM/p65BvtXflUgZF3oOoAT-x7vTyXQEHUVhQCLcB/s1600/image3.jpg)

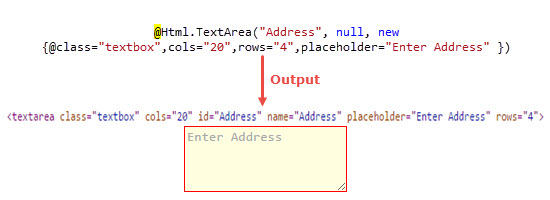
**TextBox:**TheTextBox Helper method renders a textbox in View that has a specified name. We can also add attributes like class, placeholder etc. with the help of overloaded method in which we have to pass objects of HTML Attributes.

[](https://4.bp.blogspot.com/-5mIkH3KfUQo/V0NHIsyRQ-I/AAAAAAAABIY/1DW4fs80pHww064BTNVnD6ddN8gYbpXEQCLcB/s1600/image4.jpg)

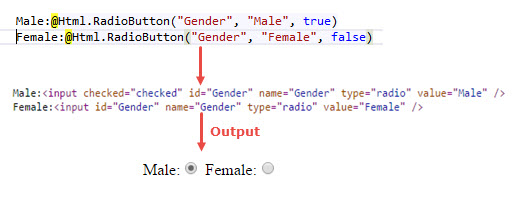
We can also set the value In Textbox by passing a value in the TextBox extension method.

[](https://1.bp.blogspot.com/-dGNPrgRfj28/V0NHEnsupkI/AAAAAAAABIU/iu2vvYOpEP4V3yMCQL9yyj9qMZ-lbu2mACKgB/s1600/image5.jpg)

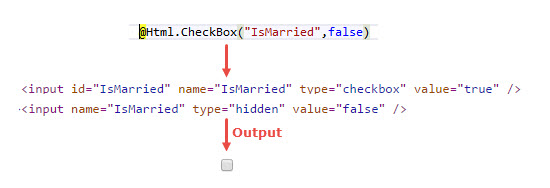
**TextArea:** The TextArea Method renders <textarea> element on view.

[](https://3.bp.blogspot.com/-J5dofJu26IU/V0NHtRWXn3I/AAAAAAAABIk/3bL2Ap0PCLYj6tZzbPvW0i8TLmNawZdkACLcB/s1600/image6.jpg)

**RadioButton:**RadioButton can be rendered in the view using the RadioButton Helper method. In the simplest form, RadioButton Helper method takes three parameters i.e. name of the control, the value, and the Boolean value for selecting the value initially.

[](https://3.bp.blogspot.com/-GO-qLPICecg/V0NIAfB9lgI/AAAAAAAABIs/6-PBM3UC1l07BOKrdmy_QMa4F2WB90ntQCLcB/s1600/image7.jpg)

**CheckBox:** The CheckBox helper method renders a checkbox and has the name and id that you specify.

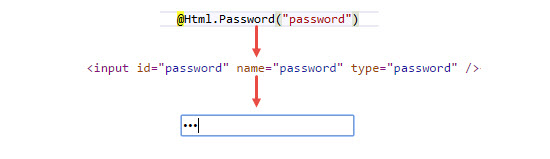
[](https://3.bp.blogspot.com/-XOPReLgKm08/V0NIUPIoVqI/AAAAAAAABIw/sFD97fvzlzkFf-bFVwhIm4DYK5zuBCfdQCLcB/s1600/image8.jpg)

In the above example, did you notice an additional input element? In case if you unchecked the checkbox or checkbox value not selected then you will get the value from the hidden field.

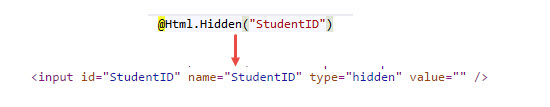
**DropDownList:** The DropDownList helper renders a drop down list.

[](https://3.bp.blogspot.com/-ECDNqwqWQlc/V0NIniNnO0I/AAAAAAAABI4/T7aS7bHsCKUfoYqdItiqXbtVD-XwUSmCQCLcB/s1600/image9.jpg)

**Password:** The Password Helper method renders the input type as password.

[](https://3.bp.blogspot.com/-MMFvbE4ie2Y/V0NI_OFicjI/AAAAAAAABI8/TAs3aiEW5zwSuZv48OJrZAw2WUz81kDxwCLcB/s1600/image10.jpg)

**Hidden:** The Hidden Helper method renders a Hidden field.

[](https://4.bp.blogspot.com/-U1U90JEX-a8/V0NJWblYkVI/AAAAAAAABJE/tChfwKd6VkU4rZjLttmrfZCbMGUdUj0RQCLcB/s1600/image11.jpg)

**2. Strongly Typed Helper Method**

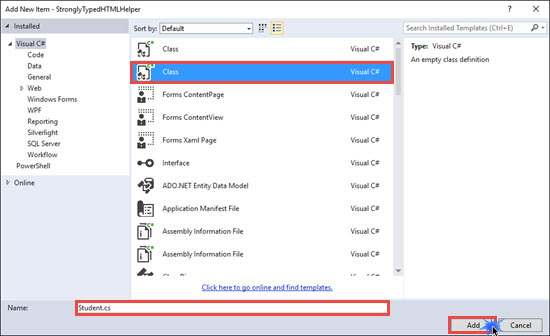
Just like Standard Helper, we have several strongly typed methods.

Html.TextBoxFor(), Html.TextAreaFor(), Html.DropDownListFor(), Html.CheckboxFor(), Html.RadioButtonFor(), Html.ListBoxFor(), Html.PasswordFor(), Html.HiddenFor(), Html.LabelFor(), etc.

Strongly Typed Helper requires lambda expressions.To use Strongly Typed Helper method, we first have to make Strongly Typed View.

**Let’s create a simple example:**

Create an empty MVC Application and add a Student class in the model.

[](https://3.bp.blogspot.com/-kK_wFTNh1tY/V0NJzsusT_I/AAAAAAAABJM/7lF47K12jDwwVk-ECZI5CvsX0HV2nLMxACLcB/s1600/image12.jpg)

public int RollNo { get ; set ; }

public string Name { get ; set ; }

public string Gender { get ; set ; }

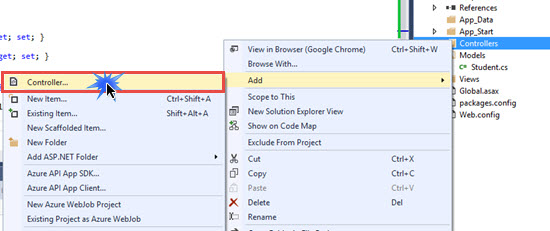
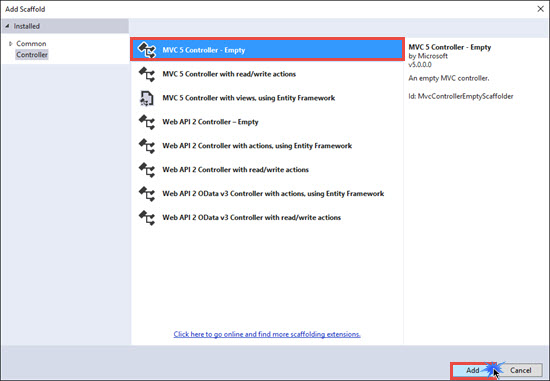
public string City { get ; set ; }

[ DataType ( DataType .MultilineText)]

public string Address { get ; set ; }

In the Student class, I have added DataType attribute on the Address property. I have added this attribute for showing a Templated HTML Helper example along with a strongly Typed Helper method example. DataTypeAttribute class is present in the System.ComponentModel.DataAnnotations namespace.

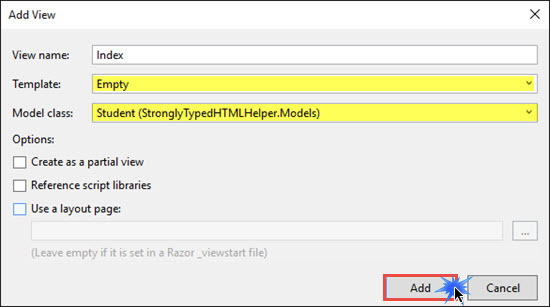
Now Add an empty controller and Name it as HomeController.

[](https://2.bp.blogspot.com/-aPuOhQDRfLI/V0NKwlg1eTI/AAAAAAAABJY/fOU8BMq-mTI5_8vRkaFkxvDzdUGMEe31wCLcB/s1600/image13.jpg)[](https://1.bp.blogspot.com/-4yTa8s-WMVE/V0NLARA-ZDI/AAAAAAAABJc/wOUChi4nhW8yQ6SSBuwYM5gGQ7CB0fHYwCLcB/s1600/image14.jpg)

public ActionResult Index()

public ActionResult Index( Student stud)

Right click on the Index Action method for the get request and click on Add View. Select Student Model class and Select empty template. Click on Add.

[](https://3.bp.blogspot.com/-KifpcPBpuzU/V0NLwwbPcPI/AAAAAAAABJo/1GZW3v1gnNUQxGZsOBSxsC0cpGSEJ5TrwCKgB/s1600/image15.jpg)

Go to View and add the following code. I am not going to use any scaffolding template for this example.

@model StronglyTypedHTMLHelper.Models. Student

< meta name ="viewport" content ="width=device-width" />

@ using (@Html.BeginForm( "Index" , "Home" , FormMethod .Post)){

< tr >< td > @ Html.LabelFor(m=>Model.RollNo) </ td >< td > @ Html.TextBoxFor(m=>Model.RollNo) </ td ></ tr >

< tr >< td > @ Html.LabelFor(m => Model.Name) </ td >< td > @ Html.TextBoxFor(m => Model.Name) </ td ></ tr >

< tr >< td > @ Html.LabelFor(m => Model.Gender) </ td >< td > Male: @ Html.RadioButtonFor(m => m.Gender, "Male" , new { @checked= "checked" })Female: @ Html.RadioButtonFor(m => m.Gender, "Female" , true ) </ td ></ tr >

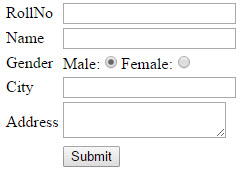
< tr >< td > @ Html.LabelFor(m => Model.City) </ td >< td > @ Html.TextBoxFor(m => Model.City) </ td ></ tr >

< tr >< td > @ Html.LabelFor(m => Model.Address) </ td >< td > @ Html.EditorFor(m => Model.Address) </ td ></ tr >

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

In the above code, we have bounded helpers with the model property using lambda expressions. EditorFor is a Tempate HTML Helper which will generate HTML elements based upon the data type of the Address property of the Student class.

**Preview:**

[](https://3.bp.blogspot.com/-3WTLY9czBpg/V0NNeESeVNI/AAAAAAAABJ0/7DatjGBcaGAcDfKQ76yNo3svtEJvMU-ggCLcB/s1600/image16.jpg)

Fill the form and click on submit. The request comes to Index Action method with [HTTPPost] attribute, from here we can get all posted values from the form and we can use that student object for any kind of Data Operations.

The following is the list of Html Helper controls.

* Html.Beginform
* Html.EndForm
* Html.Label
* Html.TextBox
* Html.TextArea
* Html.Password
* Html.DropDownList
* Html.CheckBox
* Html.RedioButton
* Html.ListBox
* Html.Hidden

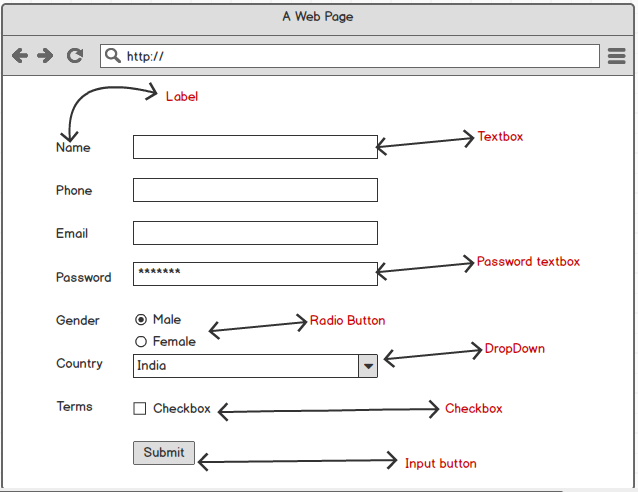
Below are Strongly Type Html Helper methods, this will allow us to check compile time errors. We get Model's Property intelligence at Runtime.

* Html.LabelFor
* Html.TextBoxFor
* Html.TextAreaFor
* Html.DropDownListFor
* Html.CheckBoxFor
* Html.RadioButtonFor
* Html.ListBoxFor
* Html.HiddenFor

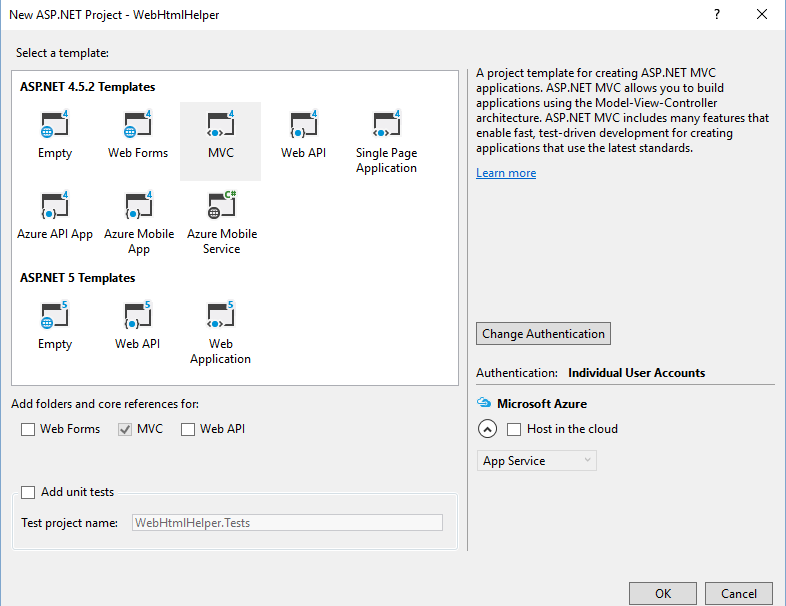
Here we have Label, Textbox, Password, Rediobutton, DropDown and Checkbox.

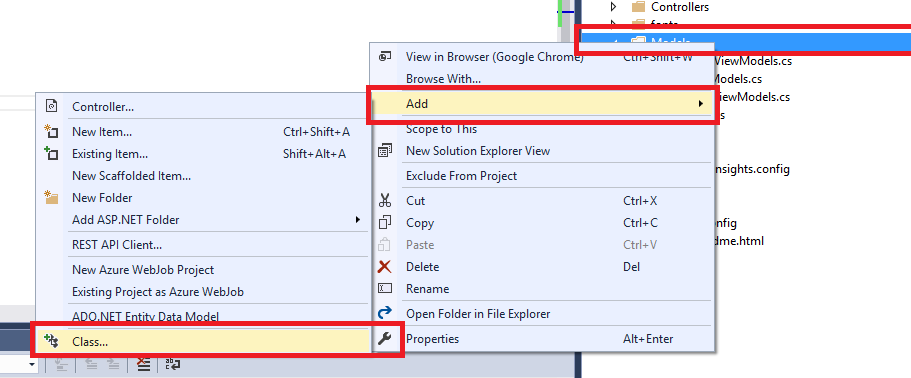
In [ASP.NET](http://www.c-sharpcorner.com/technologies/asp-dot-net-programming) Application we have Server Controls, so we just need to drag and drop from the Toolbox and design the page easily, but here in MVC Application we don't have Server Control so either we can use html or Html Helper Class Controls.

One more important thing here is in MVC there is no ViewState, so managing controls value here is challenging.



Create a new MVC Project, go to Files - > New Project -> ASP.NET Web Application and Select MVC Template. Give a name and Click on Ok.

   
  
Right click on Model folder and add new Class 'Register.cs'.



Create a property for the Register class as follows.

1. **public** **class** Register
2. {
4. **public** **string** Name { **get**; **set**; }
5. **public** **string** Phone { **get**; **set**; }
6. **public** **string** Email { **get**; **set**; }
7. **public** **string** Gender { **get**; **set**; }
8. **public** **string** Password { **get**; **set**; }
9. **public** **string** Country { **get**; **set**; }
10. **public** **bool** Terms { **get**; **set**; }
12. }

Let's use HomeController and it's view 'Index.cshtml'

In the html part let's start the with **Html.BeginForm.**That is the main container, we will place all controls inside the tag.

1. @using (Html.BeginForm("Index", "Home", FormMethod.Post))
2. {
3. }

Here in Html.Beginform we are passing View name (Index), Controller name (Home) and declare formMethod = Post Method.   
  
If you are not passing any parameter like View name and Controller name then it will automatically call the current controller action method having [HttpPost] attribute.

Inside the Form Let's Start with  **Html.Lable :**  @Html.Label("UserName").  
  
This will create a Lable Named UserName, so for Creating Label we will use @Html.Lable

Next, We need Textbox for the UserName, for that we use **Html.TextBox** same as @Html.TextBox("UserName").

This will Create a TextBox Named UserName.

1. @using (Html.BeginForm("Index", "Home", FormMethod.Post))
2. {
3. <br />
5. @Html.Label("UserName")
6. @Html.TextBox("UserName")
7. <br />
8. }

Same way we will design UserPhone, UserEmail etc.

1. @using (Html.BeginForm("Index", "Home", FormMethod.Post))
2. {
3. <br />
5. @Html.Label("UserName")
6. @Html.TextBox("UserName")
7. <br />
9. @Html.Label("UserPhone")
10. @Html.TextBox("UserPhone")
11. <br />
12. @Html.Label("UserEmail")
13. @Html.TextBox("UserEmail")
14. <br />
15. }

For Password Type Textbox We can use **Html.Password** : @Html.Password("UserPassword")

1. @Html.Label("UserPassword")
2. @Html.Password("UserPassword")
3. <br />

For Gender (Male/Female) we can use RadioButton and for that you can use **Html.RadioButton** as follows

1. @Html.Label("M")
2. @Html.RadioButton("M", new { value = "Male" })
3. @Html.Label("F")
4. @Html.RadioButton("F", new { value = "Female" })
6. <br />

For Country selection we can use dropdown and for that we have Html.DropDownList.  
  
Here we have to create a list for the country for that I have added below code outside of BeginForm tag.

1. @{
3. var MyList = new List<SelectListItem>(){
4. new SelectListItem(){Value="1",Text="India"},
5. new SelectListItem(){Value="2",Text="UK"}
7. };
8. }

Now Use this MyList in your **Html.DropDownList** as follows.

1. @Html.Label("UserCountry")
3. @Html.DropDownList("UserCountry", MyList)
5. <br />

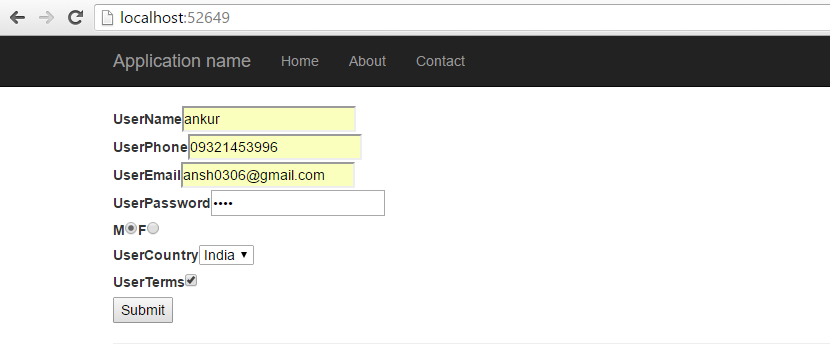
 Same way for Checkbox we use **Html.CheckBox**

1. <br />
2. @Html.Label("UserTerms")
3. @Html.CheckBox("UserTerms")
4. <br />

 Add Submit button in the view as follows.

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

Now run your application. It will display your html helper contols like label, textbox, rediobutton, etc.



Below is the view of html which shows Render of controls. Here you can see the id and name property of control id="UserName" and Name = "UserName".

1. <form action="/" method="post">
2. <br />
3. <label **for**="UserName">UserName</label>
4. <input id="UserName" name="UserName" type="text" value="" />
5. <br />
6. <label **for**="UserPhone">UserPhone</label>
7. <input id="UserPhone" name="UserPhone" type="text" value="" />
8. <br />
9. <label **for**="UserEmail">UserEmail</label>
10. <input id="UserEmail" name="UserEmail" type="text" value="" />
11. <br />
12. <label **for**="UserPassword">UserPassword</label>
13. <input id="UserPassword" name="UserPassword" type="password" />
14. <br />
15. <label **for**="M">M</label>
16. <input id="M" name="M" type="radio" value="{ value = Male }" />
17. <label **for**="F">F</label>
18. <input id="F" name="F" type="radio" value="{ value = Female }" />
19. <br />
20. <label **for**="UserCountry">UserCountry</label>
21. <select id="UserCountry" name="UserCountry">
22. <option value="1">India</option>
23. <option value="2">UK</option>
24. </select>
25. <br />
26. <label **for**="UserTerms">UserTerms</label>
27. <input id="UserTerms" name="UserTerms" type="checkbox" value="true" />
28. <input name="UserTerms" type="hidden" value="false" />
29. <br />
30. <input type="submit" value="Submit" />
31. </form>

Same thing but we will bind this with our model. I have already created the model class which is 'Register.cs'

Go to your HomeController and  add Model reference.

1. **using** WebHelper.Models;

 and in the Index method write below code.

 I have added [HrtpPost] Attributes here, and pass object r of Register class.

1. [HttpPost]
2. **public** ActionResult Index(Register r)
3. {
4. **return** View();
5. }

Now, I have designed the same form by binding it through Model property using Html Helper methods.

For Label we can use **Html.LabelFor**and assign it with the property 'Name', 'Email', 'Phone' Etc.  
  
By writing **m => m.Name**we are binding label with the model property 'Name'.

Same way for Textbox we can use **Html.TextBoxFor**  and assign it  with the property 'Name', 'Email', 'Phone' Etc

**Html.PasswordFor**for Password field, **Html.RadioButtonFor**for Password field,**Html.DropDownListFor** for dropdown field, **Html.CheckBoxFor** for checkbox field etc.

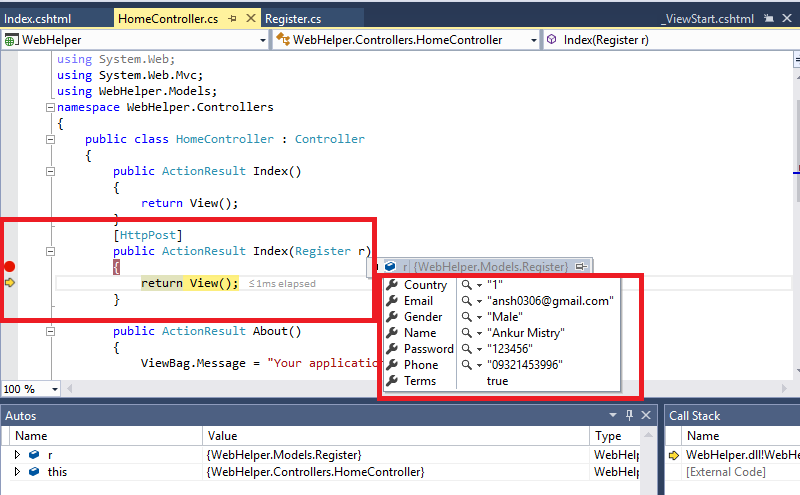
Check below html code for the same form.

1. @model WebHelper.Models.Register
2. @{
3. var MyList = **new** List
4. <SelectListItem>(){
5. **new** SelectListItem(){Value="1",Text="India"},
6. **new** SelectListItem(){Value="2",Text="UK"}
8. };
9. }
11. @**using** (Html.BeginForm("Index", "Home", FormMethod.Post))
12. {

15. @Html.LabelFor(m => m.Name)
16. @Html.TextBoxFor(m => m.Name)
18. <br />
20. @Html.LabelFor(m => m.Phone)
21. @Html.TextBoxFor(m => m.Phone)
23. <br />
24. @Html.LabelFor(m => m.Email)
25. @Html.TextBoxFor(m => m.Email)
27. <br />
28. @Html.LabelFor(m => m.Password)
29. @Html.PasswordFor(m => m.Password)
31. <br />
32. @Html.Label("Male")
33. @Html.RadioButtonFor(m => m.Gender, "Male", **new** { value = "Male" })
34. @Html.Label("Female")
35. @Html.RadioButtonFor(m => m.Gender, "Female", **new** { value = "Female" })

38. <br />
39. @Html.LabelFor(m => m.Country)
40. @Html.DropDownListFor(m => m.Country, MyList)
42. <br />
43. @Html.LabelFor(m => m.Terms)
44. @Html.CheckBoxFor(m => m.Terms)
46. <br />
47. <input type="submit" value="Submit" />
48. }

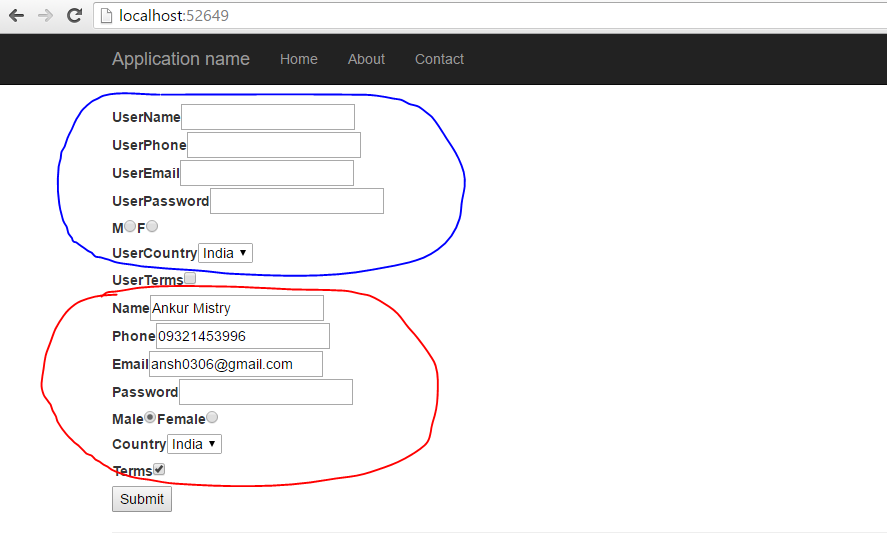
Now, run the application, fill the information and set breakpoint in controller, shown as per below screen.



Here you can see that I got all the values of the form controls in my controller class.

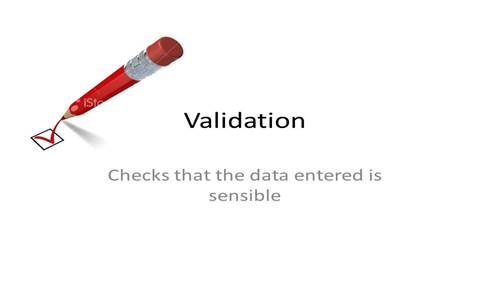
 Now if you want to persist the values of the controls you can pass the object in View for example return View(r) here I am passing object 'r'.

1. [HttpPost]
2. **public** ActionResult Index(Register r)
3. {
4. **return** View(r);
5. }



Model Binder will bind the control's values with model values.

# Data Annotations And Validation In MVC

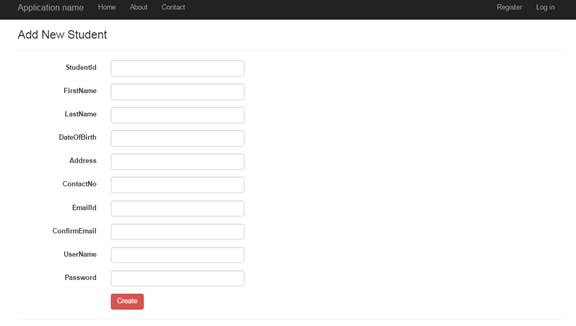
Validating user input has always been a challenging task for the Web developers. We not only want validation of logic executing in the browser, but we also must validate the logic running on the Server. The client side logic gives the users instant feedback on the information they entered into a Web page and is an expected feature in today’s applications. Meanwhile, the server validation logic is in place because you should never trust information arriving from the network.   
  
  
  
  
When we talk about the validation in MVC framework context, we primarily focus on validating the model value. Has the user provided a required value? Is the value in range etc.?

In this article, we will discuss all the predefined data annotation validation listed in *System.ComponentModel.DataAnnotations*namespace.  
  
**Prerequisite**  
For easy understanding of this article, you should have minimal ASP.Net MVC knowledge. At least, you should be aware about controller view models etc. in MVC and how to create a basic application using ASP.Net MVC.  
  
**Overview**  
For this article, we create an Application i.e ASP.NET MVC Application and named it as *DataAnnotationsValidations* (you can download the source code for better understanding) and we are using Student Model Class that contains student relation information in which we are going to validate using Data Annotation.

1. **public** **class** StudentModel
2. {
3. **public** Guid StudentId
4. {
5. **get**;
6. **set**;
7. }
8. **public** **string** FirstName
9. {
10. **get**;
11. **set**;
12. }
13. **public** **string** LastName
14. {
15. **get**;
16. **set**;
17. }
18. **public** DateTime DateOfBirth
19. {
20. **get**;
21. **set**;
22. }
23. **public** **string** Address
24. {
25. **get**;
26. **set**;
27. }
28. **public** **string** ContactNo
29. {
30. **get**;
31. **set**;
32. }
33. **public** **string** EmailId
34. {
35. **get**;
36. **set**;
37. }
38. **public** **string** ConfirmEmail
39. {
40. **get**;
41. **set**;
42. }
43. **public** **string** UserName
44. {
45. **get**;
46. **set**;
47. }
48. **public** **string** Password
49. {
50. **get**;
51. **set**;
52. }
53. }

We have added a student Controller and added Post action method to add a new student. In this Post Action method, we will apply and test the data annotation validation.  
  
**Student Controller**

1. **using** System.Web.Mvc;
2. **using** DataAnnotationsValidations.Models;
4. **namespace** DataAnnotationsValidations.Controllers
5. {
6. **public** **class** StudentController: Controller
7. {
8. // GET: Student
9. **public** ActionResult Index()
10. {
11. **return** View();
12. }
14. // GET: Student/Create
15. **public** ActionResult Create()
16. {
17. **return** View();
18. }
20. // POST: Student/Create
21. [HttpPost]
22. **public** ActionResult Create(StudentModel student)
23. {
24. **try**
25. {
26. **if** (ModelState.IsValid)
27. {
29. **return** RedirectToAction("Index");
30. }
31. **return** View();
32. } **catch**
33. {
34. **return** View();
35. }
36. }
38. }
39. }

We have added a student view to create Action Method, when we run that view  it will look like the one, shown below:  
  
  
  
**Create Student View**

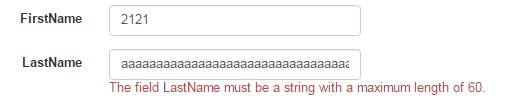
1. @model DataAnnotationsValidations.Models.StudentModel
3. @{
4. ViewBag.Title = "Add Student";
5. }
7. <h3>Add New Student</h3>
9. @using (Html.BeginForm())
10. {
11. @Html.AntiForgeryToken()
13. <div class="form-horizontal">
14. <hr />
15. @Html.ValidationSummary(true, "", new { @class = "text-danger" })
16. <div class="form-group">
17. @Html.LabelFor(model => model.StudentId, htmlAttributes: new { @class = "control-label col-md-2" })
18. <div class="col-md-10">
19. @Html.EditorFor(model => model.StudentId, new { htmlAttributes = new { @class = "form-control" } })
20. @Html.ValidationMessageFor(model => model.StudentId, "", new { @class = "text-danger" })
21. </div>
22. </div>
24. <div class="form-group">
25. @Html.LabelFor(model => model.FirstName, htmlAttributes: new { @class = "control-label col-md-2" })
26. <div class="col-md-10">
27. @Html.EditorFor(model => model.FirstName, new { htmlAttributes = new { @class = "form-control" } })
28. @Html.ValidationMessageFor(model => model.FirstName, "", new { @class = "text-danger" })
29. </div>
30. </div>
32. <div class="form-group">
33. @Html.LabelFor(model => model.LastName, htmlAttributes: new { @class = "control-label col-md-2" })
34. <div class="col-md-10">
35. @Html.EditorFor(model => model.LastName, new { htmlAttributes = new { @class = "form-control" } })
36. @Html.ValidationMessageFor(model => model.LastName, "", new { @class = "text-danger" })
37. </div>
38. </div>
40. <div class="form-group">
41. @Html.LabelFor(model => model.DateOfBirth, htmlAttributes: new { @class = "control-label col-md-2" })
42. <div class="col-md-10">
43. @Html.EditorFor(model => model.DateOfBirth, new { htmlAttributes = new { @class = "form-control" } })
44. @Html.ValidationMessageFor(model => model.DateOfBirth, "", new { @class = "text-danger" })
45. </div>
46. </div>
48. <div class="form-group">
49. @Html.LabelFor(model => model.Address, htmlAttributes: new { @class = "control-label col-md-2" })
50. <div class="col-md-10">
51. @Html.EditorFor(model => model.Address, new { htmlAttributes = new { @class = "form-control" } })
52. @Html.ValidationMessageFor(model => model.Address, "", new { @class = "text-danger" })
53. </div>
54. </div>
56. <div class="form-group">
57. @Html.LabelFor(model => model.ContactNo, htmlAttributes: new { @class = "control-label col-md-2" })
58. <div class="col-md-10">
59. @Html.EditorFor(model => model.ContactNo, new { htmlAttributes = new { @class = "form-control" } })
60. @Html.ValidationMessageFor(model => model.ContactNo, "", new { @class = "text-danger" })
61. </div>
62. </div>
64. <div class="form-group">
65. @Html.LabelFor(model => model.EmailId, htmlAttributes: new { @class = "control-label col-md-2" })
66. <div class="col-md-10">
67. @Html.EditorFor(model => model.EmailId, new { htmlAttributes = new { @class = "form-control" } })
68. @Html.ValidationMessageFor(model => model.EmailId, "", new { @class = "text-danger" })
69. </div>
70. </div>
72. <div class="form-group">
73. @Html.LabelFor(model => model.ConfirmEmail, htmlAttributes: new { @class = "control-label col-md-2" })
74. <div class="col-md-10">
75. @Html.EditorFor(model => model.ConfirmEmail, new { htmlAttributes = new { @class = "form-control" } })
76. @Html.ValidationMessageFor(model => model.ConfirmEmail, "", new { @class = "text-danger" })
77. </div>
78. </div>
80. <div class="form-group">
81. @Html.LabelFor(model => model.UserName, htmlAttributes: new { @class = "control-label col-md-2" })
82. <div class="col-md-10">
83. @Html.EditorFor(model => model.UserName, new { htmlAttributes = new { @class = "form-control" } })
84. @Html.ValidationMessageFor(model => model.UserName, "", new { @class = "text-danger" })
85. </div>
86. </div>
88. <div class="form-group">
89. @Html.LabelFor(model => model.Password, htmlAttributes: new { @class = "control-label col-md-2" })
90. <div class="col-md-10">
91. @Html.EditorFor(model => model.Password, new { htmlAttributes = new { @class = "form-control" } })
92. @Html.ValidationMessageFor(model => model.Password, "", new { @class = "text-danger" })
93. </div>
94. </div>
96. <div class="form-group">
97. <div class="col-md-offset-2 col-md-10">
98. <input type="submit" value="Create" class="btn btn-danger" />
99. </div>
100. </div>
101. </div>
102. }

This is the initial set up and we need to run this data annotation validation project.  
  
Now, we are going to discuss the validation available in data annotation only by one.   
  
**Note**Data annotations are the attributes that we can find in the *System.ComponentModel.DataAnnotations*namespace. These attributes provide Server side validation and the framework also supports client side validation.  
  
**Required**  
We will force the student to give their first name and last name, we can decorate the *FirstName*and *LastName*properties of the student Model with *required*attributes.

1. [Required]
2. **public** **string** FirstName
3. {
4. **get**;
5. **set**;
6. }
7. [Required]
8. **public** **string** LastName
9. {
10. **get**;
11. **set**;
12. }

When we run the page without giving the *FirstName*and *LastName*, we will get error message, as shown below:  
  
  
  
These attributes raise a validation error, if either property value is null or empty. Like all the built in validation attributes the required attributes deliver both client side and server side validation.  
  
With these attributes in place, if someone tries to submit the page without proving the *FirstName*and *LastName*, they will see the default error, shown above.   
  
Attributes based on validation ensures that our client and server side validation rules are kept in synchronization because they have been declared in one place.  
  
**StringLength**  
We are forcing the user to enter his name but what happens if he enters a name with enormous length? For student *LastName*, we will set *maximum 60*characters that can be entered. Hence, for doing that, we must decorate *LastName*with *StringLength*attributes.

1. [Required]
2. [StringLength(60)]
3. **public** **string** LastName
4. {
5. **get**;
6. **set**;
7. }

When we run the page after entering *LastName*more than *60*characters, we will get the error message, as shown below:  
  
  
  
We notice that we can attack multiple validation attributes on the single properties. *MinimumLength*is an optional named parameter we can use to specify the minimum length for a string.

1. [Required]
2. [StringLength(60, MinimumLength = 4)]
3. **public** **string** LastName
4. {
5. **get**;
6. **set**;
7. }

Here, last name requires a minimum of four character words.  
  
**RegularExpression**  
Regular expressions are an efficient means to enforce the shape and contains a string value. Suppose, we need to validate the Email ID of the student without sending the Email and Regular expression attributes will do it for us in the way, shown below:

1. [RegularExpression(@ "\A(?:[a-z0-9!#$%&'\*+/=?^\_`{|}~-]+(?:\.[a-z0-9!#$%&'\*+/=?^\_`{|}~-]+)\*@(?:[a-z0-9](?:[a-z0-9-]\*[a-z0-9])?\.)+[a-z0-9](?:[a-z0-9-]\*[a-z0-9])?)\Z")]
2. **public** **string** EmailId
3. {
4. **get**;
5. **set**;
6. }

When a user tries to enter an invalid Email ID and submit the page, he will get validation error, as shown below:  
  
RegularExpression   
  
We will see how to display user friendly error message in a later part of this article.  
  
**Range**  
The Range attributes specifies the minimum and maximum constrains for a numerical number, as shown below:

1. [Range(18, 30)]
2. **public** **int** Age
3. {
4. **get**;
5. **set**;
6. }

In this example, the age for student will be between 10 to 30 years to pass the validation. Here, the first parameter of the attribute is the minimum value and second part is the maximum value. The Range attributes will work with an integer, double and another overloaded versions will take Type as parameters, as shown below:

1. [Range(**typeof**(**double**),"00.00","100.49")]

**Compare**  
Compare to ensure two properties of a model has same value. For example, Email ID and confirm email ID.

1. [RegularExpression(@ "\A(?:[a-z0-9!#$%&'\*+/=?^\_`{|}~-]+(?:\.[a-z0-9!#$%&'\*+/=?^\_`{|}~-]+)\*@(?:[a-z0-9](?:[a-z0-9-]\*[a-z0-9])?\.)+[a-z0-9](?:[a-z0-9-]\*[a-z0-9])?)\Z")]
2. **public** **string** EmailId
3. {
4. **get**;
5. **set**;
6. }
7. [Compare("EmailId")]
8. **public** **string** ConfirmEmail
9. {
10. **get**;
11. **set**;
12. }

If both Email ID and confirm Email ID are not same, the user will get model validation error, as shown below:  
  
  
  
**Display**   
  
Display attribute sets the friendly name for the model properties. We can use the Display attributes to fix the name for label for *LastName*field.

1. [Required]
2. [StringLength(60, MinimumLength = 4)]
3. [Display(Name = "Last Name")]
4. **public** **string** LastName
5. {
6. **get**;
7. **set**;
8. }

When we will set the last name display properties then it will look like   
  
  
  
**ScaffoldColumn**   
  
The *ScaffoldColumn*attribute hides the properties from HTML helper such as *EditorForModel*and *DisplayForModel*,

1. [ScaffoldColumn(**false**)]
2. **public** **string** Address
3. {
4. **get**;
5. **set**;
6. }

Address will not be shown in *UI*if *ScaffoldColumn*is false.  
  
**DisplayFormat**  
  
*DisplayFormat*attribute handles formatting option for the properties via named parameters. We can provide an alternative text for display, when property is null. We can also specify data format string, currency symbol etc. Suppose we have student income property for which we want to set currency symbol, which is shown below:

1. [DisplayFormat(ApplyFormatInEditMode = **true**, DataFormatString = "{0:c}")]
2. **public** Decimal Income
3. {
4. **get**;
5. **set**;
6. }

By default, *ApplyFormatInEditMode*is false because *MVC*binder might not to parse the value formatted for display only.   
  
**ReadOnly**  
  
If we decorate an attribute with *ReadyOnly*attribute, the user can not set that property's value.

1. [ReadOnly(**true**)]
2. **public** Decimal Income
3. {
4. **get**;
5. **set**;
6. }

**DataType**   
  
*DataType*attributes enable us to provide the *runtime*information about the specific purpose of the properties. For example, a property of type string can have various scenarios as it might hold Email address, URL or a password. There are various data types that includes *Currency*, *Date*, *Time*, *Password*and *MultilineText*etc.

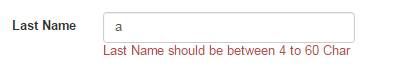
1. [DataType(DataType.Password)]
2. **public** **string** Password
3. {
4. **get**;
5. **set**;
6. }

When we set *DataType*as *password*, we will see the password field in non readable format.  
  
DataType   
  
**Custom Error Message and Localization**  
  
Every validation attribute allows you to pass the name parameter with custom error message. We can configure the default error message with a user friendly error message as we need to pass the custom error message with *ErrorMessage*named attribute shown below:

1. [RegularExpression(@ "\A(?:[a-z0-9!#$%&'\*+/=?^\_`{|}~-]+(?:\.[a-z0-9!#$%&'\*+/=?^\_`{|}~-]+)\*@(?:[a-z0-9](?:[a-z0-9-]\*[a-z0-9])?\.)+[a-z0-9](?:[a-z0-9-]\*[a-z0-9])?)\Z", ErrorMessage = "Invalid Email Id")]
2. **public** **string** EmailId
3. {
4. **get**;
5. **set**;
6. }

If a user will not input the correct Email format, he will get an error, shown below:   
  
  
  
The *ErrorMessage*can also have a single format item in the string. The built in attributes format provide the properties name with a user friendly message.

1. [Required(ErrorMessage="Your {0} is required")]
2. [StringLength(60,MinimumLength =4,ErrorMessage ="{0} should be between 4 to 60 Char")]
3. **public** **string** LastName { **get**; **set**; }

  
  
If we create an Application with globalization and localization features then this string message will be not sufficient. Hence, in this case, we can configure the error message with the resource string.

1. [Required(ErrorMessageResourceType = **typeof**(ErrorMessage), ErrorMessageResourceName = "LastName")]
2. **public** **string** LastName { **get**; **set**; }

In this case, *ErrorMessage*considers a resource file name with *LastName*as a key for an associated error message.  
  
**Validation and Model Binding**  
  
By default, ASP.Net *MVC*framework executes validation logic during model binding. In Controller side, we need to check

1. **if** (ModelState.IsValid)
2. {
3. }

We can also use,

1. **if**(TryUpdateModel(newStudent)){}

We can also check Individual validation, as shown below:

1. **if** (ModelState.IsValidField("LastName") == **false**)
2. **if** (ModelState["LastName"].Errors.Count > 0)

To display the error message in our view, we must need to include following line with all the properties in which we are applying validation.

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