**Data Annotations in ASP.NET MVC**

**Why do we need Data Annotation Attributes in ASP.NET MVC?**

Now a day’s, it’s a challenging job for a web developer to validate the user input for any Web application. As web developers, we not only validate the business logic at the client-side that is in the browser, but also we need to validate the business logic running on the Server. That means as a developer we need to validate the business logic both at the client-side as well as server-side.

The client-side validation of the business logic gives the users immediate feedback on the information they entered into a web page and which is an expected feature in today’s web applications. Along the same line, the server-side validation logic is in place because we never trust the information coming from the network.

**What are the Validations?**

In simple terms, we can say that validations are nothing but some rules set by the developer on the input fields of a web page so as to satisfy the business rules for that particular input field in order have to maintain the proper data in a system. There are two types of validations:

1. **Server-side Validations**
2. **Client Side Validations**

While doing validations, as a developer we need to take care of not only the proper validation but also ensure that the validation meets the business rule as per the requirement. From the security point of view, it is also possible that some hackers may bypass client-side validation and insert some vulnerable data into the server.

When we talk about the validation in the ASP.NET MVC framework, we primarily focus on validating the model value. That means has the user provided a required value? Is the value in the required range? Is the value in a proper format etc.?

**In ASP.NET MVC web applications, we can do the following three types of validations:**

1. HTML validation / JavaScript validation (i.e. Client-Side Validation)
2. ASP.NET MVC Model validation (i.e. Server-side Validation)
3. Database validation (i.e. Server-side Validation)

Among the above three, the most secure validation is the ASP.NET MVC model validation. In HTML/JavaScript validation, the validation can break easily by disabling the javascript in the client machine, but the model validation can’t break.

ASP.NET MVC Framework provides a concept called Data Annotation which is used for model validation. It’s inherited from System.ComponentModel.DataAnnotations assembly.

That means ASP.NET MVC Framework uses Data Annotation attributes to implement model validations. The Data Annotation Attributes include built-in validation attributes for different validation rules, which can be applied to the model class properties.

The ASP.NET MVC framework will automatically enforce the validation rules and then display proper validation messages in the view if validation fails.

**The System.ComponentModel.DataAnnotations assembly has many built-in validation attributes, for example:**

1. **Required**
2. **Range**
3. **RegularExpression,**
4. **Compare**
5. **StringLength**
6. **Data type**

Along with the above build-in validation attributes, there are also many data types the user can select to validate the input. Using this data type attribute, the user can validate the exact data type as in the following:

1. Credit Card number
2. Currency
3. Custom
4. Date
5. DateTime
6. Duration
7. Email Address
8. HTML
9. Image URL
10. Multiline text
11. Password
12. Phone number
13. Postal Code
14. Upload

**Let’s create an empty ASP.NET MVC application with the name DataAnnotationInMVC.**

Next, add the Employee.cs file to the Models folder and copy-paste the following code

**namespace** *DataAnnotationInMVC.Models*

**{**

**public** **class** Employee

**{**

**public** Guid EmployeeId

**{**

**get**;

**set**;

**}**

**public** string FirstName

**{**

**get**;

**set**;

**}**

**public** string LastName

**{**

**get**;

**set**;

**}**

**public** **int** Age

**{**

**get**;

**set**;

**}**

**public** DateTime DateOfBirth

**{**

**get**;

**set**;

**}**

**public** string Address

**{**

**get**;

**set**;

**}**

**public** string Mobile

**{**

**get**;

**set**;

**}**

**public** string PostalCode

**{**

**get**;

**set**;

**}**

**public** string EmailId

**{**

**get**;

**set**;

**}**

**public** string UserName

**{**

**get**;

**set**;

**}**

**public** string Password

**{**

**get**;

**set**;

**}**

**public** string ConfirmPassword

**{**

**get**;

**set**;

**}**

**public** string URL

**{**

**get**;

**set**;

**}**

**}**

**}**

**Adding EmployeeContoller**

Now Add EmployeeContoller to the Controllers folder and copy-paste the following code. In the below controller, we have created three action methods i.e. Index and Create action method with Get and Post HTTP Verb.

**using** *DataAnnotationInMVC.Models;*

**namespace** *DataAnnotationInMVC.Controllers*

**{**

**public** **class** EmployeeController : Controller

**{**

**public** ActionResult Index**()**

**{**

**return** View**()**;

**}**

**public** ActionResult Create**()**

**{**

**return** View**()**;

**}**

**[**HttpPost**]**

**public** ActionResult Create**(**Employee employee**)**

**{**

**try**

**{**

**if** **(**ModelState.IsValid**)**

**{**

**return** RedirectToAction**(**"Index"**)**;

**}**

**return** View**()**;

**}**

**catch**

**{**

**return** View**()**;

**}**

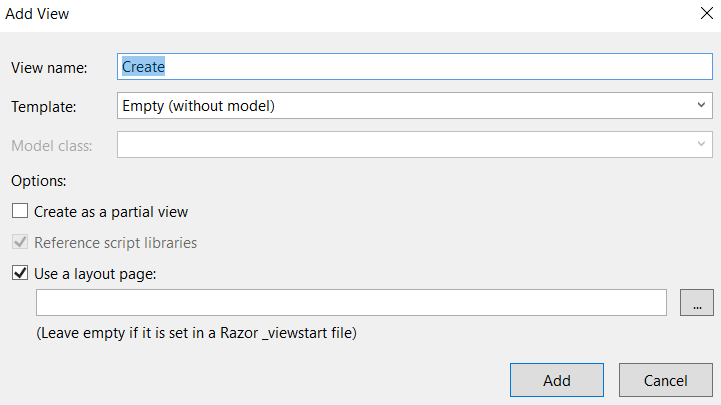
**}**

**}**

**}**

**Create the Create.cshtml view**

Right-click on the Create action method of the Home Controller class from the context menu and click on Add View. Once you click on the Add View option then it will show you the below popup.



Then click on the Add button which will create the**Create.cshtml** view in Home Folder which is inside the Views folder. Copy and paste the following code in the **Create.cshtml** view

@model DataAnnotationInMVC.Models.Employee

@{

ViewBag.Title = "Create Employee";

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

}

**<h2>**Create Employee**</h2>**

@using (Html.BeginForm())

{

@Html.AntiForgeryToken()

**<div** class="form-horizontal"**>**

**<hr/>**

@Html.ValidationSummary(true, "", new { @class = "text-danger" })

**<div** class="form-group"**>**

@Html.LabelFor(model => model.FirstName, htmlAttributes: new { @class = "control-label col-md-2" })

**<div** class="col-md-10"**>**

@Html.EditorFor(model => model.FirstName, new { htmlAttributes = new { @class = "form-control" } })

@Html.ValidationMessageFor(model => model.FirstName, "", new { @class = "text-danger" })

**</div>**

**</div>**

**<div** class="form-group"**>**

@Html.LabelFor(model => model.LastName, htmlAttributes: new { @class = "control-label col-md-2" })

**<div** class="col-md-10"**>**

@Html.EditorFor(model => model.LastName, new { htmlAttributes = new { @class = "form-control" } })

@Html.ValidationMessageFor(model => model.LastName, "", new { @class = "text-danger" })

**</div>**

**</div>**

**<div** class="form-group"**>**

@Html.LabelFor(model => model.Age, htmlAttributes: new { @class = "control-label col-md-2" })

**<div** class="col-md-10"**>**

@Html.EditorFor(model => model.Age, new { htmlAttributes = new { @class = "form-control" } })

@Html.ValidationMessageFor(model => model.Age, "", new { @class = "text-danger" })

**</div>**

**</div>**

**<div** class="form-group"**>**

@Html.LabelFor(model => model.DateOfBirth, htmlAttributes: new { @class = "control-label col-md-2" })

**<div** class="col-md-10"**>**

@Html.EditorFor(model => model.DateOfBirth, new { htmlAttributes = new { @class = "form-control" } })

@Html.ValidationMessageFor(model => model.DateOfBirth, "", new { @class = "text-danger" })

**</div>**

**</div>**

**<div** class="form-group"**>**

@Html.LabelFor(model => model.Address, htmlAttributes: new { @class = "control-label col-md-2" })

**<div** class="col-md-10"**>**

@Html.EditorFor(model => model.Address, new { htmlAttributes = new { @class = "form-control" } })

@Html.ValidationMessageFor(model => model.Address, "", new { @class = "text-danger" })

**</div>**

**</div>**

**<div** class="form-group"**>**

@Html.LabelFor(model => model.Mobile, htmlAttributes: new { @class = "control-label col-md-2" })

**<div** class="col-md-10"**>**

@Html.EditorFor(model => model.Mobile, new { htmlAttributes = new { @class = "form-control" } })

@Html.ValidationMessageFor(model => model.Mobile, "", new { @class = "text-danger" })

**</div>**

**</div>**

**<div** class="form-group"**>**

@Html.LabelFor(model => model.PostalCode, htmlAttributes: new { @class = "control-label col-md-2" })

**<div** class="col-md-10"**>**

@Html.EditorFor(model => model.PostalCode, new { htmlAttributes = new { @class = "form-control" } })

@Html.ValidationMessageFor(model => model.PostalCode, "", new { @class = "text-danger" })

**</div>**

**</div>**

**<div** class="form-group"**>**

@Html.LabelFor(model => model.EmailId, htmlAttributes: new { @class = "control-label col-md-2" })

**<div** class="col-md-10"**>**

@Html.EditorFor(model => model.EmailId, new { htmlAttributes = new { @class = "form-control" } })

@Html.ValidationMessageFor(model => model.EmailId, "", new { @class = "text-danger" })

**</div>**

**</div>**

**<div** class="form-group"**>**

@Html.LabelFor(model => model.UserName, htmlAttributes: new { @class = "control-label col-md-2" })

**<div** class="col-md-10"**>**

@Html.EditorFor(model => model.UserName, new { htmlAttributes = new { @class = "form-control" } })

@Html.ValidationMessageFor(model => model.UserName, "", new { @class = "text-danger" })

**</div>**

**</div>**

**<div** class="form-group"**>**

@Html.LabelFor(model => model.Password, htmlAttributes: new { @class = "control-label col-md-2" })

**<div** class="col-md-10"**>**

@Html.EditorFor(model => model.Password, new { htmlAttributes = new { @class = "form-control" } })

@Html.ValidationMessageFor(model => model.Password, "", new { @class = "text-danger" })

**</div>**

**</div>**

**<div** class="form-group"**>**

@Html.LabelFor(model => model.ConfirmPassword, htmlAttributes: new { @class = "control-label col-md-2" })

**<div** class="col-md-10"**>**

@Html.EditorFor(model => model.ConfirmPassword, new { htmlAttributes = new { @class = "form-control" } })

@Html.ValidationMessageFor(model => model.ConfirmPassword, "", new { @class = "text-danger" })

**</div>**

**</div>**

**<div** class="form-group"**>**

@Html.LabelFor(model => model.URL, htmlAttributes: new { @class = "control-label col-md-2" })

**<div** class="col-md-10"**>**

@Html.EditorFor(model => model.URL, new { htmlAttributes = new { @class = "form-control" } })

@Html.ValidationMessageFor(model => model.URL, "", new { @class = "text-danger" })

**</div>**

**</div>**

**<div** class="form-group"**>**

**<div** class="col-md-offset-2 col-md-10"**>**

**<input** type="submit" value="Create" class="btn btn-danger" **/>**

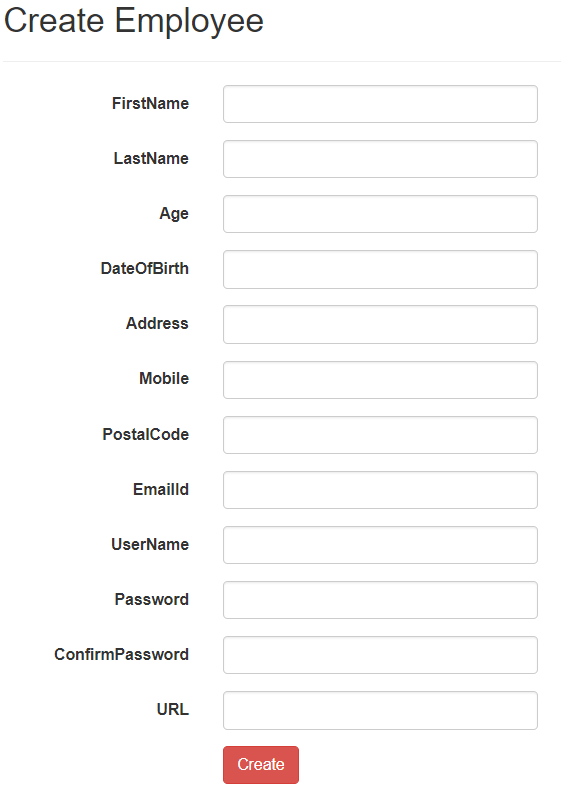
**</div>**

**</div>**

**</div>**

}

We have added Create.cshtml view to create the Action Method when we run the application and go to the “**/Employee/Create**” URL it will look as shown below:



**Enabling Client-Side Validation in ASP.NET MVC Application:**

The Validation attributes in ASP.NET MVC Framework provide both the client-side and server-side validation. There are 2 simple steps to enable the client-side validation in the ASP.NET MVC application. In the First step, we need to Enable ClientValidation and UnobtrusiveJavaScript in the web.config file. Add the following two keys within the appSettings section of your web config file.

**<appSettings>**

**<add** key="ClientValidationEnabled" value="true" **/>**

**<add** key="UnobtrusiveJavaScriptEnabled" value="true" **/>**

**</appSettings>**

In the second step, we need to include the references to the following javascript files. Add the following javascript files in sequence in the **\_Layout.cshtml** view which is inside the shared folder as our **create.cshtml** view uses **\_Layout.cshtml** view

**<script** src="~/Scripts/jquery-1.10.2.js"**></script>**

**<script** src="~/Scripts/jquery.validate.js"**></script>**

**<script** src="~/Scripts/jquery.validate.unobtrusive.js"**></script>**

In real-time we need to include the references to the javascript files in the master page which avoids the need to reference them on each and every view where we need the validation. **The order in which the script files are referenced is also important.**jquery.validate is dependent on jquery and /jquery.validate.unobtrusive is dependent on jquery.validate, so they should be referenced in the above order. Otherwise, client-side validation will not work as expected. In short, JavaScript is parsed “top-down”, so all dependencies need to be referenced before the dependent reference.

With the above 2 changes, validation should now happen on the client-side without a round trip to the server. If the client disables javascript in the browser, then client-side validation does not work, but the server-side validation will continue to work as expected.

**Required and StringLength Attribute in MVC**

**Required and StringLength Attribute in ASP.NET MVC**

**Note:**[**Data annotations**](https://dotnettutorials.net/lesson/data-annotation-attributes-mvc/) are the attributes that we can find in the **System.ComponentModel.DataAnnotations** namespace. These attributes provide Server-side validation as well as client-side validation.

**Required Attribute in ASP.NET MVC:**

let’s understand Required Attribute with one example. Our business requirement is that the First name and Last Name of an employee can’t be empty. That means we will force the Employee to give the first name and last name; we can achieve this very easily in the ASP.NET MVC application by decorating the FirstName and LastName properties of the Employee Model with the **Required** data annotation attribute as shown below. The Required attribute makes the model property as required.

**[**Required**]**

**public** string FirstName

**{**

**get**;

**set**;

**}**

**[**Required**]**

**public** string LastName

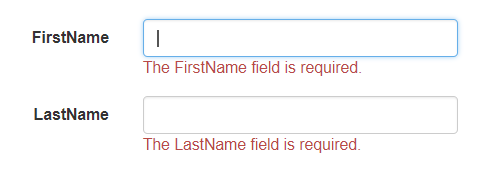
**{**

**get**;

**set**;

**}**

The Required attribute raises a validation error, if either the property value is null or empty. The built-in required validation attributes provide both client-side and server-side validation like other built-in validation attributes. When we submit the page without providing the FirstName and LastName of an employee, we will get the error message as shown in the below image.



With the required attributes in place, if someone tries to submit the page without providing the FirstName and LastName values, then it will give you the default error as shown in the above image. But if you want to provide some user-defined error message when validation fails, then you can use the other overloaded version of the Required attribute which accepts ErrorMessage as an input parameter as shown below.

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

**public** string FirstName

**{**

**get**;

**set**;

**}**

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

**public** string LastName

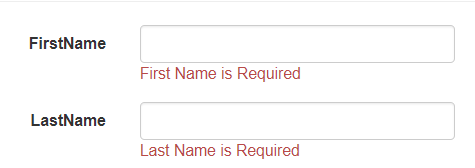
**{**

**get**;

**set**;

**}**

With the above changes, if someone tries to submit the page without providing the FirstName and LastName values, then it will give the user-defined error message as shown in the below image.



**StringLength Attribute in ASP.NET MVC**

In our last example, we are forcing the user to enter his first name and last but what happens if he enters a name with enormous length? For example, our business requirement is that the employee’s LastName should not be greater than 30 characters that mean we need to set a maximum of 30 characters that can be entered for the employee the last name. We can achieve this very easily using the **StringLength** data annotation attribute in the ASP.NET MVC application. To achieve this we need to decorate the LastName property with the StringLength attribute as shown below.

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

**[**StringLength**(**30**)]**

**public** string LastName

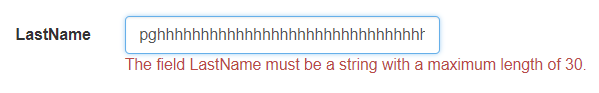
**{**

**get**;

**set**;

**}**

When we run the application, and when we enter more than 30 characters in the LastName textbox, then we will get the below default error message:



**Note:** We can apply multiple validation attributes on a single property. The MinimumLength is an optional named parameter that is used to specify the minimum length for a string. We can also specify the user-defined error message as shown below. Here, we specify the minimum length as 4 and the ErrorMessage as the Last name should be between 4 and 30 characters.

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

**[**StringLength**(**30, MinimumLength = 4,

ErrorMessage = "Last name should be between 4 and 30 characters"**)]**

**public** string LastName

**{**

**get**;

**set**;

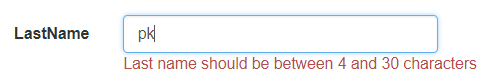
**}**

In the above example, we have decorated the **“**LastName**”**property with the **StringLength** attribute and then specified the **Minimum** and **Maximum** length of the model properties. We also used the **[Required]** attribute. So, at this point, the LastName property is required and should be between 4 and 30 characters.

**Points to remember about StringLength Attribute in MVC:**

1. **[StringLength]** attribute is present in **System.ComponentModel.DataAnnotations** namespace.
2. **[StringLength]** attribute verifies that a string is of a certain length, but does not enforce that the property is **REQUIRED**. If you want to enforce that the property is required then use the **[Required]** attribute.

Now run the application and check everything is working as expected as shown below.



**Regular Expression Attribute in ASP.NET MVC**

**Regular Expression Attribute in ASP.NET MVC**

**What is Regular Expression Attribute in ASP.NET MVC?**

The Regular Expression Attribute is generally used for pattern matching validations in ASP.NET MVC applications.

**Let’s understand the Regular expression attribute with an example.**

Suppose, we need to validate the Email ID of an employee, then we can achieve this very easily in the ASP.NET MVC application by using Regular expression attributes as shown below:

**[**Required**(**ErrorMessage = "Email id is required"**)]**

**[**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 = "Please enter a valid email address"**)]**

**public** string EmailId

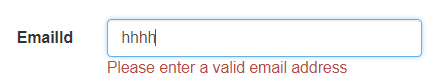
**{**

**get**;

**set**;

**}**

In the above example, we are applying both **Required** and **RegularExpression** attributes to the EmailID model property which ensures that the Email Id is a required field and along with it will validate the email id field value as shown below. When a user tries to enter an invalid Email ID and submit the page, then he will get the validation error message as shown below:



**Another Example:**

**Here is the requirement for validating Name property**

1. UserName can contain the first and last names with a single space.
2. The last name is optional. If the last name is not present, then there shouldn’t be any space after the first name.
3. Only upper and lower case alphabets are allowed.

This requirement can achieve very easily in ASP.NET MVC using **RegularExpressionAttribute**. In the **Employee.cs** class file, decorate the UserName property with the **RegularExpression** attribute as shown below.

**[**RegularExpression**(**@"^(([A-za-z]+[\s]{1}[A-za-z]+)|([A-Za-z]+))$"**)]**

**public** string UserName

**{**

**get**;

**set**;

**}**

Notice that, we are passing regular expression string to the attribute constructor. The Regular Expression Attribute is great for pattern matching and ensures that the value for the UserName property is in the format that we want. Also, notice that we are using a verbatim literal (@ symbol) string, as we don’t want escape sequences to be processed.

Run the application and see the UserName field is work as expected. Understanding and writing regular expressions is beyond the scope of this article. If you are interested in learning to write regular expressions, here is a link from MSDN

[**http://msdn.microsoft.com/en-us/library/az24scfc.aspx**](http://msdn.microsoft.com/en-us/library/az24scfc.aspx)

The following website is very helpful, for writing and testing regular expressions. This website also contains commonly used regular expressions. In fact, I have picked up the regular expression for validating the UserName property from here.

[**http://gskinner.com/RegExr/**](http://gskinner.com/RegExr/)

**Range Attribute in ASP.NET MVC**

**Range Attribute in ASP.NET MVC Application**

[**Data Annotations in ASP.NET MVC**](https://dotnettutorials.net/lesson/data-annotation-attributes-mvc/)

[**Required and StringLength Attribute**](https://dotnettutorials.net/lesson/required-attribute-asp-dot-net-mvc/)

[**Regular Expression Attribute**](https://dotnettutorials.net/lesson/regular-expression-attribute-mvc/)

**Range Attribute in ASP.NET MVC Application:**

Notice that we don’t have validation on **Age field**. That means if you enter **5000** as the age and click on the Save button, then the data also gets saved. As we know an employee having 5000 years as the age is not possible. So, let’s validate the Age field, and enforce users to enter a value between 25 and 60. We can achieve this very easily by using the **RangeAttribute**in ASP.NET MVC Application. The Range attributes specify the minimum and maximum constraints for a numerical number, as shown below:

**[**Range**(**25, 60, ErrorMessage = "Age must be between 25 and 60"**)]**

**public** **int** Age

**{**

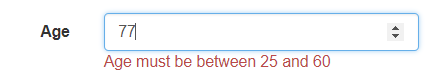
**get**;

**set**;

**}**

In the above example, we set the age of the employee will be between 25 to 60 years to pass the validation. Here, the first parameter of the attribute is the minimum value and the second parameter is the maximum value and the third parameter is the error message that we want to show when the validation failed.

When a user tries to enter the age which is not between 25 and 60 and click on the submit button, then he will get the validation error message as shown below:



At this point, we should not be able to enter any values outside the range of 25 and 60 for the Age field. The**Range**attribute in ASP.NET MVC can also be used to validate the DateTime fields.

**Range Attribute with DateTime fields in ASP.NET MVC Application.**

Notice that, we are passing DateTime as the type and specifying the **minimum** and **maximum** values for DateOfBirth. We are also using the **DisplayFormat** attribute so that the only date part of **DateTime** is displayed in the view.

**[**Range**(**typeof**(**DateTime**)**, "01-01-1970", "01-01-2005",

ErrorMessage = "Date of Birth Must be between 01-01-1970 and 01-01-2005"**)]**

**[**DisplayFormat**(**DataFormatString = "{0:d}", ApplyFormatInEditMode = **true)]**

**public** DateTime DateOfBirth

**{**

**get**;

**set**;

**}**

At this point, we should not be able to enter any values outside the range of **“01/01/1970”** and **“01/01/2005”** for the DateOfBirth field. However, when the Range attribute is used with DateTime fields, the client-side validation does not work as expected. We will discuss this in a later session.

**MinLength and MaxLength Attribute in ASP.NET MVC Application**

These two attributes are used to specify the Minimum Length and Maximum Length of a property. For example, if we want to restrict the Employee Address as 5 as the minimum length and 25 as the maximum length then we can decorate the address property with the MinLength and Maxlength attribute as shown below

**[**MinLength**(**5, ErrorMessage = "The Address must be at least 5 characters"**)]**

**[**MaxLength**(**25, ErrorMessage = "The Address cannot be more than 25 characters"**)]**

**public** string Address

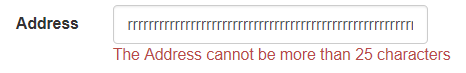
**{**

**get**;

**set**;

**}**

When we submit the page by entering more than 25 characters, it will give the error as shown below.



**Custom Validation Attribute in MVC**

**Custom Validation Attribute in MVC**

**[**Range**(**typeof**(**DateTime**)**, "01-01-1970", "01-01-2005",

ErrorMessage = "Date of Birth Must be between 01-01-1970 and 01-01-2005"**)]**

**[**DisplayFormat**(**DataFormatString = "{0:d}", ApplyFormatInEditMode = **true)]**

**public** DateTime DateOfBirth

**{**

**get**;

**set**;

**}**

But, let’s say, we want the end date to be today’s date instead of the hardcode “**01/01/2005**” value. To achieve this we would be tempted to use **DateTime.Now.ToShortDateString**() as shown below.

**[**Range**(**typeof**(**DateTime**)**, "01-01-1970", DateTime.Now.ToShortDateString**()**,

ErrorMessage = "Date of Birth Must be between 01-01-1970 and Current Date"**)]**

**[**DisplayFormat**(**DataFormatString = "{0:d}", ApplyFormatInEditMode = **true)]**

**public** DateTime DateOfBirth

**{**

**get**;

**set**;

**}**

At this point, if you compile your application, you will get the error saying – **An attribute argument must be a constant expression, typeof expression or array creation expression of an attribute parameter type.**

**To fix this error, we can create a custom DateRangeAttribute. Here are the steps**

1. Right-click on the project name in solution explorer, and add **“Common”** folder.
2. Then Right-click on the **“Common”** folder and add a class file with the name **DateRangeAttribute.cs**
3. Copy and paste the following code in **DateRangeAttribute.cs** class file.

**using** *System.ComponentModel.DataAnnotations;*

**namespace** *DataAnnotationInMVC.Common*

**{**

**public** **class** DateRangeAttribute : RangeAttribute

**{**

**public** DateRangeAttribute**(**string minimumValue**)**

: **base(**typeof**(**DateTime**)**, minimumValue, DateTime.Now.ToShortDateString**())**

**{**

**}**

**}**

**}**

Finally, decorate the **“DateOfBirth”** property with our custom **DateRangeAttribute** as shown below. Notice that, we are only passing the minimum date value. The maximum date value will be today’s date. Please note, DateRangeAttribute is present in MVCDemo.Common namespace.

**[**DateRange**(**"01/01/2000", ErrorMessage = "Date of Birth Must be between 01-01-1970 and Current Date"**)]**

**[**DisplayFormat**(**DataFormatString = "{0:d}", ApplyFormatInEditMode = **true)]**

**public** DateTime DateOfBirth

**{**

**get**;

**set**;

**}**

**Another example of creating a custom validation attribute in ASP.NET MVC.**

Let’s say our business rules have changed, and the DateOfBirth property should allow any valid date that is <= Today’s Date. This means, there is no minimum value restriction and the maximum value should be less than or equal to Today’s date. To achieve this, let’s add another custom validation attribute. Here are the steps

1. Right-click on the “**Common**” folder and add a class file with the name **CurrentDateAttribute.cs**
2. Copy and paste the following code in **CurrentDateAttribute.cs** class file.

**using** *System.ComponentModel.DataAnnotations;*

**namespace** *DataAnnotationInMVC.Common*

**{**

**public** **class** CurrentDateAttribute : ValidationAttribute

**{**

**public** **override** **bool** IsValid**(object** **value)**

**{**

DateTime dateTime = Convert.ToDateTime**(value)**;

**return** dateTime **<**= DateTime.Now;

**}**

**}**

**}**

Decorate the **“**DateOfBirth**“** property with our custom **CurrentDateAttribute** as shown below.

**[**CurrentDate**]**

**[**DisplayFormat**(**DataFormatString = "{0:d}", ApplyFormatInEditMode = **true)]**

**public** DateTime DateOfBirth

**{**

**get**;

**set**;

**}**

Please note that the validation error message can be customized using the named parameter **“ErrorMessage”** as shown below.

**[**CurrentDate**(**ErrorMessage = "Hire Date must be less than or equal to Today's Date"**)]**

**[**DisplayFormat**(**DataFormatString = "{0:d}", ApplyFormatInEditMode = **true)]**

**public** DateTime DateOfBirth

**{**

**get**;

**set**;

**}**

# DataType and Compare Attributes in ASP.NET MVC

## ****DataType and Compare Attributes in ASP.NET MVC Application****

In this article, I am going to discuss **DataType and Compare Attributes in ASP.NET MVC Application** with Examples. Please read our previous article where we discussed how to create [**Custom Validation Attribute in** **ASP.NET** **MVC**](https://dotnettutorials.net/lesson/custom-validation-attribute-mvc/). At the end of this article, you will understand the need and use of Compare Attribute in the ASP.NET MVC Application.

#### ****DataType Attribute in ASP.NET MVC****:

DataType Attribute in ASP.NET MVC Framework enables 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 an Email address, URL, or password. There are various data types that include Currency, Date, Time, Password and MultilineText, etc. Let’s see some of the examples of using the DataType attribute.

**[**DataType**(**DataType.PostalCode, ErrorMessage = "Please Enter a valid PIN/ZIP Code"**)]**

**public** string PostalCode

**{**

**get**;

**set**;

**}**

**[**DataType**(**DataType.Url, ErrorMessage = "Please Enter a valid URL"**)]**

**public** string URL

**{**

**get**;

**set**;

**}**

**[**DataType**(**DataType.Password**)]**

**public** string Password

**{**

**get**;

**set**;

**}**

**[**DataType**(**DataType.PhoneNumber, ErrorMessage = "Please Enter a valid Phone Number"**)]**

**public** string Mobile

**{**

**get**;

**set**;

**}**

##### ****Password:****

When we set the DataType as a password, we will see the password field in a non-readable format. Let’s add Password Attribute to Password and Confirm Password property of employee model as shown below

**[**DataType**(**DataType.Password**)]**

**[**Required**(**ErrorMessage = "Password is Required"**)]**

**public** string Password

**{**

**get**;

**set**;

**}**

**[**DataType**(**DataType.Password**)]**

**[**Required**(**ErrorMessage = "Confirm Password is Required"**)]**

**public** string ConfirmPassword

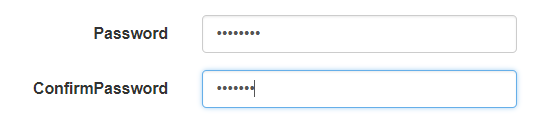
**{**

**get**;

**set**;

**}**

When we run the application and enter the data into the password and confirm password textbox, it will be shown as below.



#### ****Compare**** ****Attribute in ASP.NET MVC Application****:

Compare Attribute in ASP.NET MVC Framework is used to compare 2 properties of a model that have the same value. Comparing email addresses and passwords is the common use case of Compare attribute. Let’s understand using Compare attribute with an example. For example, to ensure that the user has typed the correct password we must use Password and ConfirmPassword of employee model as shown below.

**[**DataType**(**DataType.Password**)]**1

**[**Required**(**ErrorMessage = "Password is Required"**)]**

**public** string Password

**{**

**get**;

**set**;

**}**

**[**DataType**(**DataType.Password**)]**

**[**Required**(**ErrorMessage = "Confirm Password is Required"**)]**

**[**Compare**(**"Password", ErrorMessage = "Password and Confirm Password do not match"**]**

**public** string ConfirmPassword

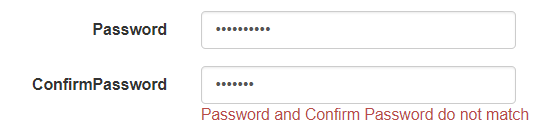
**{**

**get**;

**set**;

**}**

If both Password and Confirm Password are not the same, the user will get the model validation error, as shown below:



**Validation Message and Validation Summary in ASP.NET MVC**

**Validation Message and Validation Summary in ASP.NET MVC**

1. **ValidationMessage**
2. **ValidationMessageFor**
3. **ValidationSummary**

**ValidationMessage Attribute in ASP.NET MVC:**

The Html.ValidationMessage() is an extension method, that is a loosely typed method. It displays a validation message if an error exists for the specified field in the ModelStateDictionary object. The Validation Method Signature is given below:  
**MvcHtmlString ValidateMessage(string modelName, string validationMessage, object htmlAttributes)**

Consider the following ValidationMessage example.  
**@Html.EditorFor(model => model.FirstName, new { htmlAttributes = new { @class = “form-control” } })**  
**@Html.ValidationMessage(“FirstName”, “”, new { @class = “text-danger” })**

In the above example, the first parameter in the ValidationMessage method is a property name for which we want to show the error message e.g. Employee First Name. The second parameter is for the custom error messages and the third parameter is for html attributes like css, style, etc.

The ValidationMessage() method will only display an error if you have configured the DataAnnotations attribute to the specified property in the model class.

**The above code will generate following html.**

**<**input **class**="form-control text-box single-line"

data-val="true"

data-val-required="First Name is Required"

id="FirstName"

name="FirstName"

type="text"

**value**=""**>**

**<**span **class**="field-validation-valid text-danger"

data-valmsg-**for**="FirstName"

data-valmsg-replace="true"**><**/span**>**

Now, when the user submits a form without entering a FirstName, then ASP.NET MVC uses a data- attribute of HTML5 for the validation and validation message will be injected, when the validation error occurs, as shown below.

<span class=”text-danger field-validation-error”  
data-valmsg-for=”FirstName”  
data-valmsg-replace=”true”>  
<span for=”FirstName” generated=”true” class=””>First Name is Required</span>  
</span>

**ValidationMessageFor Attribute in ASP.NET MVC:**

The Html.ValidationMessageFor() is a strongly typed extension method. It displays a validation message if an error exists for the specified field in the ModelStateDictionary object. The validationMessageFor signature is given below.

**MvcHtmlString ValidateMessage(Expression<Func<dynamic, TProperty>> expression, string validationMessage, object htmlAttributes)**

**Consider the following ValidationMessageFor() example.**

**@Html.EditorFor(model => model.FirstName, new { htmlAttributes = new { @class = “form-control” } })**  
**@Html.ValidationMessageFor(model => model.FirstName, “”, new { @class = “text-danger” })**

In the above example, the first parameter in ValidationMessageFor method is a lambda expression to specify a property for which we want to show the error message. The second parameter is for custom error message and the third parameter is for html attributes like css, style etc.

The ValidationMessageFor() method will only display an error if you have configured the DataAnnotations attribute to the specifed property in the model class.

**The above code will generate following html.**

**<**input **class**="form-control text-box single-line"

data-val="true"

data-val-required="First Name is Required"

id="FirstName"

name="FirstName"

type="text"

**value**=""**>**

**<**span **class**="field-validation-valid text-danger"

data-valmsg-**for**="FirstName"

data-valmsg-replace="true"**><**/span**>**

Now, when the user submits the form without entering the FirstName then ASP.NET MVC uses the data- attribute of HTML5 for the validation and the validation message will be injected when validation error occurs, as shown below.

<span class=”text-danger field-validation-error”  
data-valmsg-for=”FirstName”  
data-valmsg-replace=”true”>  
<span for=”FirstName” generated=”true” class=””>First Name is Required</span>  
</span>

**ValidationSummary Attribute in ASP.NET MVC Application:**

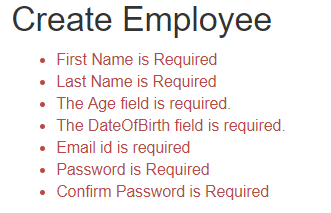
The ValidationSummary helper method generates an unordered list (ul element) of validation messages that are in the ModelStateDictionary object. The ValidationSummary can be used to display all the error messages for all the fields. It can also be used to display custom error messages.

**Displaying all validation errors at one place using validation summary HTML helper**

To display all errors at one place, use ValidationSummary() HTML helper.

**@Html.ValidationSummary(false, “Please fix the following errors and then submit the form”)**

The following figure shows how ValidationSummary displays the error messages. Now run the application and submit the form without filling the data. It will show the errors as shown below



**Remote Validations in ASP.NET MVC**

**Remote Validations in ASP.NET MVC**

**What is Remote Validation in ASP.NET MVC Application?**

Sometimes, to check if a field value is valid or not, we may need to make a database call. A classic example of this is the Gmail user registration page. To register a user, we need a unique username. So, to check, if the username is not taken already, we have to make a call to the server and check the database table. RemoteAttribute is useful in situations like this. So in this article, I am going to discuss how to use Remote Validations in ASP.NET MVC Application.

**Example:** When a user provides a username that already exists, the associated validation error message should be displayed immediately as shown below.



**Step 1: Create Users table**

**Create** **table** Users

(

[UserID] int primary key identity,

[FullName] nvarchar(50),

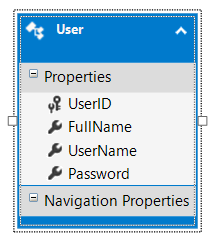
[UserName] nvarchar(50),

[Password] nvarchar(50)

)

**Step2:** Create one ASP.NET MVC application with the name “RemoteValidationInMVC”

**Step3:**Create an ado.net entity data model using table Users. Save and build the solution.



**It will create the following model**

**namespace** *RemoteValidationInMVC.Models*

**{**

**public** **partial** **class** User

**{**

**public** **int** UserID **{** **get**; **set**; **}**

**public** string FullName **{** **get**; **set**; **}**

**public** string UserName **{** **get**; **set**; **}**

**public** string Password **{** **get**; **set**; **}**

**}**

**}**

**Step4: Add HomeController with the following settings**

1. Template = MVC5 controller with views, using Entity Framework
2. Model Class = User (RemoteValidationInMVC.Models)
3. Data context class = UserDBContext (RemoteValidationInMVC.Models)
4. Controller Name = HomeController

**Step5:**

Copy and paste the following function in HomeController. This is the method that gets called to perform the remote validation. An AJAX request is issued to this method. If this method returns true, validation succeeds, else validation fails and the form is prevented from being submitted. The parameter name (UserName) must match the field name on the view. If they don’t match, the model binder will not be able to bind the value with the parameter, and validation may not work as expected.

**[**HttpPost**]**

**public** JsonResult IsUserNameAvailable**(**string UserName**)**

**{**

**return** Json**(**!db.Users.Any**(**x =**>** x.UserName == UserName**)**,

JsonRequestBehavior.AllowGet**)**;

**}**

**Step6:**

Right-click on the Models folder and a class file with the name **PUser.cs**. Copy and paste the following code. Notice that the name of the method (IsUserNameAvailable) and the controller name (Home) and the error message are passed as arguments to Remote Attribute. Remote is the attribute for validation in Data Annotation, which is used in model class to validate records instantly.

**using** *System.Web.Mvc;*

**using** *System.ComponentModel.DataAnnotations;*

**namespace** *RemoteValidationInMVC.Models*

**{**

**[**MetadataType**(**typeof**(**UserMetaData**))]**

**public** **partial** **class** User

**{**

**}**

**public** **class** UserMetaData

**{**

**[**Remote**(**"IsUserNameAvailable", "Home",HttpMethod ="POST", ErrorMessage = "UserName already in use."**)]**

**public** string UserName **{** **get**; **set**; **}**

**}**

**}**

In the above example, we have defined a few properties of the Remote attribute to work on remote validation properly, let’s know them in brief.

1. **IsUserNameAvailable**: This is the JsonResult method which checks the details from the database and returns true or false.
2. **Home**: This is the MVC Controller name and inside that, the IsUserNameAvailable JsonResult method is defined to check the details from the database.
3. **HttpMethod**: This is the HttpMethod type which is called on Remote attribute e.g. Get, Put, Post. This is optional to define.
4. **ErrorMessage**: This is used to show the message on the client-side.

There are many optional properties of the Remote attribute which are used as per the validation requirements.

**Step7:**

Include references to the following CSS and script files in **Create.cshtml** view. jquery.validateand jquery.validate.unobtrusivescript files are required for remote validation to work.

**<link** href="~/Content/Site.css" rel="stylesheet" type="text/css" **/>**

**<script** src="~/Scripts/jquery-1.7.1.min.js" type="text/javascript"**></script>**

**<script** src="~/Scripts/jquery.validate.min.js" type="text/javascript"**></script>**

**<script** src="~/Scripts/jquery.validate.unobtrusive.js" type="text/javascript"**></script>**

**Step8:**

**Make sure ClientValidation and UnobtrusiveJavaScript are enabled in web.config**

**<add key=”ClientValidationEnabled” value=”true” />**  
**<add key=”UnobtrusiveJavaScriptEnabled” value=”true” />**

**Remote Validation in ASP.NET MVC when JavaScript is Disabled**

**Remote Validation in ASP.NET MVC when JavaScript is Disabled**

**Remote validation in MVC when javascript is disabled:**

The Remoteattribute only works when JavaScript is enabled. If the end-user disables JavaScript on his/her machine then the validation does not work. This is because RemoteAttribute requires JavaScript to make an asynchronous AJAX call to the server-side validation method. As a result, the user will be able to submit the form, bypassing the validation in place. This is why it is always important to have server-side validation.

To make server-side validation work, when JavaScript is disabled, there are 2 ways

1. **Add model validation error dynamically in the controller action method**
2. **Create a custom remote attribute and override IsValid() method**

**Add model validation error dynamically in the controller action method**

Modify the Create action method that is decorated with the [HttpPost] attribute as shown below.

**[**HttpPost**]**

**[**ValidateAntiForgeryToken**]**

**public** ActionResult Create**([**Bind**(**Include = "UserID,FullName,UserName,Password"**)]** User user**)**

**{**

// Check if the UserName already exists, and if it does, add Model validation error

**if** **(**db.Users.Any**(**x =**>** x.UserName == user.UserName**))**

**{**

ModelState.AddModelError**(**"UserName", "UserName already in use"**)**;

**}**

**if** **(**ModelState.IsValid**)**

**{**

db.Users.Add**(**user**)**;

db.SaveChanges**()**;

**return** RedirectToAction**(**"Index"**)**;

**}**

**return** View**(**user**)**;

**}**

At this point, disable JavaScript in the browser, and test your application. Notice that, we don’t get client-side validation, but when we submit the form server-side validation still prevents the user from submitting the form, if there are validation errors.

However, delegating the responsibility of performing validation, to a controller action method violates the separation of concerns within MVC. Ideally, all validation logic should be in the Model. Using validation attributes in models should be the preferred method for validation.

**Creating a Custom Remote Attribute**

**Step1:**Right-click on the project name in solution explorer and a folder with the name “Common”

**Step2:** Right-click on the “Common” folder, you have just added and add a class file with the name RemoteClientServer.cs

**Step3:** Copy and paste the following code.

**namespace** *RemoteValidationInMVC.Common*

**{**

**public** **class** RemoteClientServerAttribute : RemoteAttribute

**{**

**protected** **override** ValidationResult IsValid**(object** **value**, ValidationContext validationContext**)**

**{**

// Get the controller using reflection

Type controller = Assembly.GetExecutingAssembly**()**.GetTypes**()**

.FirstOrDefault**(**type =**>** type.Name.ToLower**()** == string.Format**(**"{0}Controller",

this.RouteData**[**"controller"**]**.ToString**())**.ToLower**())**;

**if** **(**controller != **null)**

**{**

// Get the action method that has validation logic

MethodInfo action = controller.GetMethods**()**

.FirstOrDefault**(**method =**>** method.Name.ToLower**()** ==

this.RouteData**[**"action"**]**.ToString**()**.ToLower**())**;

**if** **(**action != **null)**

**{**

// Create an instance of the controller class

**object** instance = Activator.CreateInstance**(**controller**)**;

// Invoke the action method that has validation logic

**object** response = action.Invoke**(**instance, new **object[]** **{** **value** **})**;

**if** **(**response **is** JsonResult**)**

**{**

**object** jsonData = **((**JsonResult**)**response**)**.Data;

**if** **(**jsonData **is** **bool)**

**{**

**return** **(bool)**jsonData ? ValidationResult.Success :

new ValidationResult**(**this.ErrorMessage**)**;

**}**

**}**

**}**

**}**

**return** ValidationResult.Success;

// If you want the validation to fail, create an instance of ValidationResult

// return new ValidationResult(base.ErrorMessageString);

**}**

**public** RemoteClientServerAttribute**(**string routeName**)**

: **base(**routeName**)**

**{**

**}**

**public** RemoteClientServerAttribute**(**string action, string controller**)**

: **base(**action, controller**)**

**{**

**}**

**public** RemoteClientServerAttribute**(**string action, string controller,

string areaName**)** : **base(**action, controller, areaName**)**

**{}}}**

**Step4:** Open **“User.cs”** file, that is present in **“Models”**folder. Decorate **“UserName”**property with RemoteClientServerAttribute.

**using** *System.ComponentModel.DataAnnotations;*

**using** *RemoteValidationInMVC.Common;*

**namespace** *RemoteValidationInMVC.Models*

**{**

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

**public** **partial** **class** User

**{**

**}**

**public** **class** UserMetadata

**{**

**[**RemoteClientServer**(**"IsUserNameAvailable", "Home",

ErrorMessage = "UserName already in use"**)]**

**public** string UserName **{** **get**; **set**; **}**

**}**

**}**

Disable JavaScript in the browser, and test your application. Notice that, we don’t get client-side validation, but when you submit the form server-side validation still prevents the user from submitting the form, if there are validation errors.