**Data Annotations & Validations In ASP.Net MVC**

**Validation Of Data**

* In an ASP.NET MVC application, users interact with the application by Submitting a form provided by the application as a view.
* We as a developer need to ensure that any data sent by a user to the application is valid.
* We can validate user data by including validation logic in your application.
* The validation logic should first analyze whether or not the user specified data is valid and as expected by the application.
* The validation logic should accordingly provide feedback to the user so that the user can identify and rectify any invalid input before they resubmit the data.
* The MVC Framework implements a validation workflow to validate user data.
* The validation workflow starts when a user specified data arrives in the server.

**Validation Process**

* Iirst checks whether the request is some form of attack, such as Cross Site Scripting (CSS).
* If the process identifies a request as an attack, it throws an exception.
* Then, the process checks the request data against the validation requirements of a application.
* If all data meets the validation requirements, the process forwards the request to the application for further processing.
* If one or more data does not meet the validation requirements, the process sends back a validation error message as a response.

In an ASP.NET MVC application, you can manually validate data in the controller action that accepts user data for some kind of processing.

To manually validate data, you can implement simple validation routines (Logics / Conditions), such as a routine (Logics / Conditions) to check that the length of a password submitted through a registration form exceeds more than seven characters.

Below code snippet shows a manual validation implemented in an action method:

public class HomeController : Controller {

Public ActionResult Index() {

return View();

}

[HttpPost]

Public ActionResult Index(User model) {

String modelPassword = model.password;

if (modelPassword.Length< 7) {

return View();

}

else {

/\*Implement registration process\*/

return Content("You have successfully registered");

}

}

}

**In Above code,**

* Creates a HomeController controller class with two Index() action methods.
* The first Index() method returns the corresponding view.
* When a user submits a form displayed by the view the HttpPost version of the action method receives a User model object that represents the user submitted data.
* This action method validates whether or not the length of the password property is greater than seven.
* If the password length is less than seven, the action method returns back the view, else the action method returns a success message.
* You can also use regular expression while performing manual validations.
* For example, you can use a regular expression to check whether the user submitted e-mail id is in the correct format, such as example@abc.com.

Below code snippet shows a manual validation implemented in an action method using regular expression

public class HomeController : Controller {

public ActionResult Index() {

return View();

}

[HttpPost]

public ActionResult Index(User model) {

string modelEmailId = model.email;

string regexPattern=@"[A-Za-z0-9.\_%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4}";

if (System.Text.RegularExpressions.Regex.IsMatch(modelEmailId,

regexPattern)) {

/\*Implement registration process\*/

return Content("You have successfully registered"); }

else {

return View(); }

}

}

**In Above Code,**

* Uses the regexPattern variable to store a regular expression that specifies a valid email id format.
* Then, uses the IsMatch() method of the System.Text.RegularExpressions.Regex class to validate whether or not the email property is in valid format. If the value of the email property is valid, the action method returns back the view else the action method returns a success message.

**Data Annotations In ASP.Net MVC**

* The MVC Framework provides several data annotations that you can apply as attributes to a model.
* These data annotations implement tasks that are commonly required across applications.
* Some of the important annotations that you can use in the models of an ASP.NET MVC application are as follows:
  + Required
  + StringLength
  + RegularExpression
  + Range
  + Compare
  + DisplayName
  + ReadOnly
  + DataType
  + ScaffoldColumn

**Required Data Annotation**

* The Required data annotation specifies that the property, with which this annotation is associated, is a required property.
* This attribute raises a validation error if the property value is null or empty.
* Following is the syntax of the Required data annotation:

[Required]

public string <property\_name>;

where,

* property\_name: Is the name of a model property.

Following code snippet shows using the Required annotation in the properties of a User model:

public class User {

public long Id { get; set; }

[Required]

public string Name { get; set; }

[Required]

public string Password { get; set; }

[Required]

public string ReenterPassword { get; set; }

[Required]

public int Age { get; set; }

[Required]

public string Email { get; set; }

}

In Above code, the Required attribute is specified for the Name, Password, ReenterPassword, Age, and Email properties of the User model.

* Once you use the Required attributes in the model properties, you should:
  + Use the Html.ValidationSummary() helper method passing true as the parameter
  + This method is used to display validation messages on the page.
  + Use the Html.ValidationMessageFor() helper method passing the lambda expression to associate the method with a model property for each field.
  + This method returns the validation error message for the associated property as HTML.

Following code shows the view that uses helper methods to display validation messages:

@model MVCModelDemo.Models.User @{

ViewBag.Title = “User Form Validation”;

}

<h2>User Form</h2>

@using (Html.BeginForm()) {

@Html.ValidationSummary(true)

<div>

@Html.LabelFor(model =>model.Name)

</div>

<div>

@Html.EditorFor(model =>model.Name)

@Html.ValidationMessageFor(model =>model.Name)

</div>

<div>

@Html.LabelFor(model =>model.Password)

</div>

<div>

@Html.EditorFor(model =>model.Password)

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

</div>

<div>

@Html.LabelFor(model =>model.ReenterPassword)

</div>

<div>

@Html.EditorFor(model =>model.ReenterPassword)

@Html.ValidationMessageFor(model =>model.ReenterPassword)

</div>

<div>

@Html.LabelFor(model =>model.Age)

</div>

@Html.EditorFor(model =>model.Age)

@Html.ValidationMessageFor(model =>model.Age)

<div>

@Html.LabelFor(model =>model.Email)

</div>

<div>

@Html.EditorFor(model =>model.Email)

@Html.ValidationMessageFor(model =>model.Email)

</div>

<p>

<input type=”submit” value=”Submit” />

</p>

}

**In Above Code,**

* The Html.ValidationSummary(true) helper method displays validation messages as a list.
* Then, for each UI field, the Html.ValidationMessageFor() helper method is used to validate the corresponding property of the model.

**Custom Message In Required Data Annotation**

The syntax to specify a custom validation message for the Required annotation is as follows:

[Required(ErrorMessage = <error-message>)]

where,

error-message: Is the custom error message that you want to display.

Following code snippet shows a User model with the Required annotations with custom validation messages applied to its properties:

public class User {

public long Id { get; set; }

[Required(ErrorMessage = "Please enter your name.")]

public string Name { get; set; }

[Required(ErrorMessage = "Please enter your password.")]

public string Password { get; set; }

[Required(ErrorMessage = "Please re-enter your password.")]

public string ReenterPassword { get; set; }

[Required (ErrorMessage = "Please enter your age.")]

public int Age { get; set; }

[Required(ErrorMessage = "Please enter your Email-ID.")]

public string Email { get; set; }

}

**Validation Message And Validation Summary Helper Methods Of Html In ASP.Net MVC**

**Validation Message**

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.

**Validation Summary**

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.

**Source Code Of Validation Message And Validation Summary**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text.RegularExpressions;

using System.Web;

using System.Web.Mvc;

namespace ValidationMessage\_Summary.Controllers

{

public class HomeController : Controller

{

string EmailPattern = "^([0-9a-zA-Z]([-\\.\\w]\*[0-9a-zA-Z])\*@([0-9a-zA-Z][-\\w]\*[0-9a-zA-Z]\\.)+[a-zA-Z]{2,9})$";

// GET: Home

public ActionResult Index()

{

return View();

}

[HttpPost]

public ActionResult Index(string fullName, string Age, string Email)

{

if(fullName.Equals("") == true)

{

ModelState.AddModelError("fullName","Full Name is required !!");

ViewData["fullNameError"] = "\*";

}

if (Age.Equals("") == true)

{

ModelState.AddModelError("Age", "Age is required !!");

ViewData["AgeError"] = "\*";

}

if (Email.Equals("") == true)

{

ModelState.AddModelError("Email", "Email is required !!");

ViewData["EmailError"] = "\*";

}

else

{

if(Regex.IsMatch(Email,EmailPattern) == false)

{

ModelState.AddModelError("Email", "Invalid Email !!");

ViewData["EmailError"] = "\*";

}

}

if(ModelState.IsValid == true)

{

ViewData["SuccessMessage"] = "<script>alert('Data has been Submitted !!')</script>";

ModelState.Clear();

}

return View();

}

}

}

**Index.cshtml Code**

@{

ViewBag.Title = "Index";

}

<h2>Index</h2>

<p>

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

</p>

@using (Html.BeginForm())

{

<p>Enter Name: @Html.TextBox("fullName","",new { @class = "form-control"})

<b style="color:red;">@ViewData["fullNameError"]</b>

@\*@Html.ValidationMessage("fullName",new { @class = "text-danger" })\*@

</p>

<p>Enter Age: @Html.TextBox("Age", "", new { @class = "form-control" })

<b style="color:red;">@ViewData["AgeError"]</b>

@\*@Html.ValidationMessage("Age", new { @class = "text-danger" })\*@

</p>

<p>Enter Email: @Html.TextBox("Email", "", new { @class = "form-control" })

<b style="color:red;">@ViewData["EmailError"]</b>

@\*@Html.ValidationMessage("Email",new { @class = "text-danger" })\*@

</p>

<input type="submit" value="Submit" class="btn btn-info" />

@Html.Raw(ViewData["SuccessMessage"])

}

**StringLength Data Annotation In MVC**

* You can use the StringLength annotation to specify the minimum and maximum lengths of a string field.
* Whenever a user specified values for these fields are out of this specified range a validation error message is displayed

Following is the syntax for StringLength annotation:

[StringLength(<max\_length>, MinimumLength= <min\_length>)]

Where,

* max\_length: Is an integer value that specifies the maximum allowed length.
* min\_length: Is an integer value that specifies the minimum allowed length.

Following code shows a User model with the StringLength annotations applied to its Password and ReenterPassword properties:

public class User { public long Id { get; set; }

[Required(ErrorMessage = "Please enter your name.")]

public string Name { get; set; }

[StringLength(9, MinimumLength = 4)]

[DataType(DataType.Password)]

public string Password { get; set; }

[StringLength(9, MinimumLength = 4)]

[DataType(DataType.Password)]

public string ReenterPassword { get; set; }

[Required (ErrorMessage = "Please enter your age.")]

public int Age { get; set; }

[Required(ErrorMessage = "Please enter your Email-ID.")]

public string Email { get; set; } }

In this code, the [StringLength] annotation specifies that the maximum length of the Password and ReenterPassword properties is set to 9 and the minimum length is set to 4.

Whenever a user specified values for these fields are out of this specified range a validation error message is displayed

**Regular Expression Data Annotation In ASP.Net MVC**

* You can use the RegularExpression annotation to accept user input in a specific format.
* This annotation allows you to match a text string with a search pattern that contains one or more character literals, operators, or constructs.

Following is the syntax for RegularExpression annotation:

[RegularExpression(<pattern>)]

where,

<pattern> : is the specified format according to which you want user input.

Following code shows a User model with the RegularExpression annotations applied to its Email property:

public class User {

public long Id { get; set; }

[Required(ErrorMessage = "Please enter your name.")]

public string Name { get; set; }

[StringLength(9, MinimumLength = 4)]

public string Password { get; set; }

[StringLength(9, MinimumLength = 4)]

public string ReenterPassword { get; set; }

[Required (ErrorMessage = "Please enter your age.")]

publicint Age { get; set; }

[RegularExpression(@"[A-Za-z0-9.\_%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4}",

ErrorMessage = "Email is not valid.")]

public string Email { get; set; }

}

**In Above Code,**

* The regular expression A-Za-z0-9.\_%+-]+@[A-Za-z0-9.-]+.[A-Zaz]{2,4} defines the format of an e-mail address. It is divided in three parts where:
  + Part 1:[A-Za-z0-9.\_%+-]+
  + Part 2:[A-Za-z0-9.-]+
  + Part 3:[A-Za-z]{2,4}
* The first part of the regular expression specifies the characters and it ranges within square brackets that can appear.
* Then, the + sign between the first and second part indicates that these parts can consist of one or more characters of the types specified within the square brackets preceding the + sign.
* The third part contains {2,4} at the end indicating that this part can include 2-4 characters.

**Range Data Annotation In ASP.Net MVC**

You can use the Range annotation to specify the minimum and maximum constraints for a numeric value.

Following is the syntax of the Range annotation:

[Range (<minimum\_range>, <maximum\_range>)]

**Where,**

* minimum\_range: Is a numeric value that specifies the minimum value for the range.
* maximum\_range: Is a numeric value that specifies the maximum value for the range.

Following code shows a User model with the Range annotations applied to its Age property:

public class User {

public long Id { get; set; }

[Required(ErrorMessage = "Please enter your name.")]

public string Name { get; set; }

[StringLength(9, MinimumLength = 4)]

public string Password { get; set; }

[StringLength(9, MinimumLength = 4)]

public string ReenterPassword { get; set; }

[Range(18, 60, ErrorMessage = "The age should be between 18 and 60.")]

public int Age { get; set; }

[RegularExpression(@"[A-Za-z0-9.\_%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4}",

ErrorMessage = "Email is not valid.")]

public string Email { get; set; }

}

This code shows using the Range annotations to the Age property.

**Compare Data Annotation In MVC**

* You can use the Compare annotation to compare values in two fields.
* The Compare annotation ensures that the two properties on a model object have the same value.

Following code snippet shows a User model with the Compare annotation applied to its ReenterPassword property:

public class User {

public long Id { get; set; }

[Required(ErrorMessage = "Please enter your name.")]

public string Name { get; set; }

[StringLength(9, MinimumLength = 4)]

public string Password { get; set; }

[StringLength(9, MinimumLength = 4)]

[Compare("Password", ErrorMessage = "The specified passwords do not

match with the Password field.")]

public string ReenterPassword { get; set; }

[Range(18, 60, ErrorMessage = "The age should be between 18 and 60.")]

public int Age { get; set; }

[RegularExpression(@"[A-Za-z0-9.\_%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4}",

ErrorMessage = "Email is not valid.")]

public string Email { get; set; }

}

This code uses the Compare annotation to check that the ReenterPassword field contains the same values as that of the Password field

**DisplayName Data Annotation In MVC**

* When you use the @Html.LabelFor() helper method in a strongly typed view, the method displays a label with a text whose value is the corresponding property name.
* You can also explicitly state the text that the @Html.LabelFor() method should display using the DisplayName annotation on the model property.

Following is the syntax for DisplayName annotation:

[DisplayName(<text>)]

**where,**

text: Is the text to be displayed for the property.

**Following code shows a User model with the DisplayName annotation applied to its Name, ReenterPassword, and Email properties:**

public class User {

public long Id { get; set; }

[DisplayName("User Name")]

public string Name { get; set; }

public string Password { get; set; }

[DisplayName("Re-enter Password")]

public string ReenterPassword { get; set; }

public int Age { get; set; }

[DisplayName("Email- ID")]

public string Email { get; set; }

}

This code applies the DisplayName annotation to the Name, ReenterPassword, and Email properties.

**ReadOnly Data Annotation In MVC**

* We can use the ReadOnly annotation to display read-only fields on a form.
* We can use this annotation to instruct the default model binder not to set the specific property with a new value from the request.

Following is the syntax for ReadOnly annotation:

[ReadOnly(<boolean\_value>)]

**where,**

boolean\_value: Is a boolean value which can be either true or false. A true value indicates that the default model binder should not set a new value for the property. A false value indicates that the default model binder should set a new value for the property based on the request.

Following code snippet shows how to use the ReadOnly annotation:

[ReadOnly(true)]

public intDiscountedAmount { get; set; }

}

In this code uses the ReadOnly annotation passing true as the value for the DiscountedAmount property

**DataType Data Annotation In MVC**

* You can use the DataType annotation to provide information about the specific purpose of a property at run time.

Following is the syntax for DataType annotation:

[DataType(DataType.<value>)]

**where,**

value: Is a member of DataType enumeration.

Few DataType Annotation Values:

* Password
* MultilineText
* Date
* Time

Following code shows applying the DataType annotation to the  
Password and Address properties of a model:

[DataType(DataType.Password)]

public string Password { get; set; }

[DataType(DataType.MultilineText)]

public string Address { get; set; }

**In Above code,**

* The DataType annotation is first applied to the Password property. This instructs the view to display password field masks the user entered password.
* Next, the DataType annotation is applied to the Address property. This instructs the view to display a multi-line text area for the Address property

**ScaffoldColumn Data Annotation**

* When you use scaffolding using the Create template, the view by default will create UI fields for all the properties of the model.
* However, you might need to ensure that the template does not create UI fields for certain properties.
* In such scenarios, you can use the ScaffoldColumn annotation passing a false value.
* Following code shows using the ScaffoldColumn annotation:

[ScaffoldColumn (false)]

public long Id { get; set; }

In this code, the ScaffoldColumn annotation with a false value instructs the Create scaffolding template not to create a UI field in the view for the Id property.

**The ModelState Validation**

* ModelState is a class in the System.Web.Mvc namespace that encapsulate the state of model binding in an application.
* ModelState object stores the values that a user submits to update a model and any validation errors.
* You can check whether the state of a model is valid by using the ModelState.IsValid property
* If the model binding succeeds, the ModelState.IsValid property returns true and you can accordingly perform the required function with the model data.

Below code shows how to use the ModelState.IsValid property

publicActionResult Index(User model)

{

if (ModelState.IsValid)

{

/\*Perform required function with the model data\*/

return Content("You have successfully registered");

}

else

return View();

}

In Above code, the ModelState.IsValid is used to check the state of the User model.

* If the property returns true, a success message is displayed.
* Then the view is returned so that the user can enter valid values.

**Source Code Of ALL Data Annotations In One Project**

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.ComponentModel.DataAnnotations;

using System.Linq;

using System.Web;

namespace DataAnnotation.Models

{

public class Employee

{

[DisplayName("ID")]

[Required(ErrorMessage = "Id is mandatory")]

public int EmployeeId { get; set; }

[DisplayName("NAME")]

[Required(ErrorMessage = "Name is mandatory")]

[StringLength(20,MinimumLength = 5,ErrorMessage = "Name should be in between 5 and 20")]

public string EmployeeName { get; set; }

[DisplayName("AGE")]

[Required(ErrorMessage = "Age is mandatory")]

[Range(20,60,ErrorMessage = "Age should be in the range of 20 and 60")]

public int? EmployeeAge { get; set; }

[DisplayName("GENDER")]

[Required(ErrorMessage = "Gender is mandatory")]

public string EmployeeGender { get; set; }

[DisplayName("EMAIL")]

[Required(ErrorMessage = "Email is mandatory")]

[RegularExpression("^([0-9a-zA-Z]([-\\.\\w]\*[0-9a-zA-Z])\*@([0-9a-zA-Z][-\\w]\*[0-9a-zA-Z]\\.)+[a-zA-Z]{2,9})$",ErrorMessage = "Invalid Email")]

public string EmployeeEmail { get; set; }

[DisplayName("PASSWORD")]

[Required(ErrorMessage = "Employee Password is mandatory")]

[RegularExpression(@"(?=^.{8,}$)((?=.\*\d)|(?=.\*\W+))(?![.\n])(?=.\*[A-Z])(?=.\*[a-z]).\*$",ErrorMessage = "UpperCase, LowerCase, Numbers, Symbols, 8 Characters")]

[DataType(DataType.Password)]

public string EmpPassword { get; set; }

[DisplayName("CONFIRM PASSWORD")]

[Required(ErrorMessage = "Employee Confirm Password is mandatory")]

[Compare("EmpPassword",ErrorMessage = "Password is not identical")]

[DataType(DataType.Password)]

public string EmpConfirmPassword { get; set; }

[DisplayName("ORGANIZATION NAME")]

[ReadOnly(true)]

public string EmpOgranizationName { get; set; }

[DisplayName("ADDRESS")]

[Required(ErrorMessage = "Address is mandatory")]

[DataType(DataType.MultilineText)]

public string EmpAddress { get; set; }

[DisplayName("JOINING DATE")]

[Required(ErrorMessage = "Date is mandatory")]

[DataType(DataType.Date)]

public string EmpJoiningDate { get; set; }

[DisplayName("JOINING TIME")]

[Required(ErrorMessage = "Time is mandatory")]

[DataType(DataType.Time)]

public string EmpJoiningTime { get; set; }

}

}

**HomeController Source Code**

using DataAnnotation.Models;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

namespace DataAnnotation.Controllers

{

public class HomeController : Controller

{

// GET: Home

public ActionResult Index()

{

return View();

}

[HttpPost]

public ActionResult Index(Employee e)

{

if(ModelState.IsValid == true)

{

ViewData["SuccessMessage"] = "<script>alert('Data has been submitted !!')</script>";

ModelState.Clear();

}

return View();

}

}

}

**Index.cshtml Source Code**

@model DataAnnotation.Models.Employee

@{

ViewBag.Title = "Index";

}

<h2>Index</h2>

@using (Html.BeginForm())

{

<p>@Html.LabelFor(model => model.EmployeeId) &nbsp; @Html.TextBoxFor(model => model.EmployeeId,"",new {@class = "form-control" })

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

</p>

<p>@Html.LabelFor(model => model.EmployeeName) &nbsp; @Html.TextBoxFor(model => model.EmployeeName, new { @class = "form-control" })

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

</p>

<p>@Html.LabelFor(model => model.EmployeeAge) &nbsp; @Html.TextBoxFor(model => model.EmployeeAge, new { @class = "form-control" })

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

</p>

<p>@Html.LabelFor(model => model.EmployeeGender) &nbsp; @Html.TextBoxFor(model => model.EmployeeGender, new { @class = "form-control" })

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

</p>

<p>@Html.LabelFor(model => model.EmployeeEmail) &nbsp; @Html.TextBoxFor(model => model.EmployeeEmail, new { @class = "form-control" })

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

</p>

<p>

@Html.LabelFor(model => model.EmpPassword) &nbsp; @Html.EditorFor(model => model.EmpPassword, new { htmlAttributes = new { @class = "form-control" } })

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

</p>

<p>

@Html.LabelFor(model => model.EmpConfirmPassword) &nbsp; @Html.EditorFor(model => model.EmpConfirmPassword, new { htmlAttributes = new { @class = "form-control" } })

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

</p>

<p>

@Html.LabelFor(model => model.EmpOgranizationName) &nbsp; @Html.TextBoxFor(model => model.EmpOgranizationName, new { @class = "form-control", @Value = "ABC ORGANIZATION", @readonly = true })

</p>

<p>

@Html.LabelFor(model => model.EmpAddress) &nbsp; @Html.EditorFor(model => model.EmpAddress, new { htmlAttributes = new { @class = "form-control" } })

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

</p>

<p>

@Html.LabelFor(model => model.EmpJoiningDate) &nbsp; @Html.EditorFor(model => model.EmpJoiningDate, new { htmlAttributes = new { @class = "form-control" } })

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

</p>

<p>

@Html.LabelFor(model => model.EmpJoiningTime) &nbsp; @Html.EditorFor(model => model.EmpJoiningTime, new { htmlAttributes = new { @class = "form-control" } })

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

</p>

<input type="submit" value="Submit" class="btn btn-info" />

@Html.Raw(ViewData["SuccessMessage"])

}