Agenda: Annotations and Validations

- Overview of Data Annotations
- Validation Attributes
- How Validation Works
- Using CustomValidationAttribute
- Model Level Validation using IValidatatableObject
- Developing Custom Unobtrusive Client Side Validation
- Applying Annotations to partial Model class

Data Annotations and Validation Overview

Validating user-input and enforcing business rules/logic are core requirements of most the web applications.

We want to enable this validation to occur on both the server and on the client (via JavaScript).

- Data annotations are attribute classes in the namespace **System.ComponentModel.DataAnnotations**.
- DataAnnotations provide a really easy way to **declaratively** add validation rules to objects and properties with minimal code.
- These annotations are available across various Visual Studio projects including
    - ASP.NET MVC,
    - Web Forms,
    - Silverlight & WPF.
    - For data models like
        - EF models,
        - Linq2SQL models, etc...

  Basically we can use them in any type of project.

We'll implement validation rules on our model object – and *not* within our Controller or our View. The benefit of implementing the rules within model object is that this will ensure that the validation will be enforced via any scenario within our application that uses the model object. This will help ensure that we keep our code DRY and avoid repeating rules in multiple places.

Validation Attributes

Following are the attribute classes under the name space **System**.**ComponentModel.DataAnnotations**

- Required
- Compare
- Range
- ReqularExpression
- StringLength
- MinLength
- MaxLength
- Email
- FileExtensions
- Url
- Remote
- CustomValidation

More Annotation Attributes

Attribute Name	Description
DataType	Specifies the name of an additional type to associate with a data field.
DisplayFormat	Specifies how data fields are displayed and formatted. [DisplayFormat(ConvertEmptyStringToNull = true, NullDisplayText = "[Null]")] [DisplayFormat(DataFormatString="{0:C}")]
HiddenInput	If the HTML element should be hidden – this is set to be true by using the HiddenInput attribute with a **DisplayValue** value of false.
Display	**Name** is text of Label and **Order** is used to set the order of item in the list.

Example: Create an Employee model with Annotations as mentioned below

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.ComponentModel.DataAnnotations;
namespace MvcApplication1.Models
{
   public class Employee
   {
      [Required(ErrorMessage = "User Name is Required")]
      public string UserName { get; set; }

      [Required(ErrorMessage = "Password is Required")]
      [DataType("password")]
      public string Password { get; set; }

      [Required(ErrorMessage = "DateOfBirth is Required")]
      [Display(Name="Date of Birth")]
      [DataType(DataType.Date)]
      [DisplayFormat(DataFormatString="{0:d}", ApplyFormatInEditMode=true)]
      public DateTime DateOfBirth { get; set; }

      [Required(ErrorMessage = "Email is Required")]
      [EmailAddress(ErrorMessage = "Please enter valid Email Id")]
```

```
        public string Email { get; set; }

        [Required(ErrorMessage = "Rating is Required")]
        [Range(1, 10)]
        public int Rating { get; set; }

        [Required(ErrorMessage = "Phone Number is Required")]
        [Display(Name="Phone Number")]
        public int PhoneNo { get; set; }

        [Required(ErrorMessage = "Comments is Required")]
        [DataType(DataType.MultilineText)]
        public string Comments { get; set; }

        [FileExtensions(Extensions = "png,jpg,jpeg,gif")]
        public string Photo { get; set; }
    }
}
```

Once you've setup validation on the model using data annotations, they're automatically consumed by **Html Helpers** in views so the helpers can render the proper HTML output.

For client side Validations include the following in .CSHTML file:

```
@section scripts
{
    @Scripts.Render("~/bundles/jqueryval");
}
```

Example:

Create Employee view

```
@using (Html.BeginForm())
{
    @Html.ValidationSummary(true, "Please correct below errors")
    <fieldset>
        <legend>Employee</legend>
        @Html.EditorForModel()
        <input type="submit" value="Create" />
```

```
</fieldset>
}
OR
@using (Html.BeginForm()) {
   @Html.ValidationSummary(true)
   <fieldset>
      <legend>Employee</legend>
      <div class="editor-label">
         @Html.LabelFor(model => model.UserName)
      </div>
      <div class="editor-field">
         @Html.EditorFor(model => model.UserName)
         @Html.ValidationMessageFor(model => model.UserName) //Shows ErrorMessage
         OR
         Html.ValidateFor(model => model.UserName); //Doesn't Show ErrorMessage – This is Optional
      </div>
      . . .
      </p>
   </fieldset>
}
```

To get the list of all validation errors from the ModelStateObject:

```
string s = "";
foreach (var key in ModelState.Keys)
{
   foreach (var err in ModelState[key].Errors)
   {
      s += err.ErrorMessage;
   }
}
```

How Validation Works

Both client and server side validation work because of a few conventions in your project that match up data annotations and Html Helpers rendered output. Html Helpers in views render HTML elements containing attributes that start with the pattern **data-val-*.** The data-val-* attributes contain error messages, regular expressions, ranges, and other validation information that originates in data annotations.

That means that this decorated code in the model...

```
[Display (Name="Qty Available")]
[Required(ErrorMessage = "The Qty Available field is required.")]
[Range(0,120)]
public int QtyOnHand { get; set; }
```

...combined with this code in the view...

```
<div class="editor-field">
  @Html.EditorFor(model => model.QtyOnHand)
  @Html.ValidationMessageFor(model => model.QtyOnHand)
</div>
```

...turns into this HTML at runtime...

```
<div class="editor-field">
  <input class="text-box single-line"
  data-val="true" data-val-number="The field Qty Available must be a number."
  data-val-range="The field Qty Available must be between 0 and 120."
  data-val-range-max="120" data-val-range-min="0"
  data-val-required="The Qty Available field is required."
  id="QtyOnHand" name="QtyOnHand" type="text" value="12" />
  <span class="field-validation-valid" data-valmsg-for="QtyOnHand" data-valmsg-replace="true"></span>
</div>
```

The tie-in between the data model annotations and the data-val-* attributes should be clear after reading the above code, but it's where the client side validation ties in, might not be so obvious. Open the **\Scripts\jquery.validate.unobtrusive.js** file and search for "data-val". Right away you'll see that the JavaScript uses the data-val-*, input-* and field-* CSS classes to display/hide validation messages on the client. Although you shouldn't modify or need to maintain the built-in .js files; it's worth investigating them to see how things work together in ASP.NET MVC.

```
function onError(error, inputElement) {  // 'this' is the form element
  var container = $(this).find("[data-valmsg-for='" + inputElement[0].name + "']"),
      replace = $.parseJSON(container.attr("data-valmsg-replace")) !== false;

  container.removeClass("field-validation-valid").addClass("field-validation-error");
  error.data("unobtrusiveContainer", container);
  if (replace) {
    container.empty();
    error.removeClass("input-validation-error").appendTo(container);
  }
```

```
    else {
        error.hide();
    }
}
```

Having the onError function tied in by only HTML attributes keeps client side validation unobtrusive, or in other words, out of the way of your view code. Annotations combined with unobtrusive validation make both the view and the output very clean and maintainable. The final result in the browser is fully capable client side validation that falls back to server side validation for browsers without JavaScript enabled. Both way, the same validation happens and the user sees the same error.

Remote Validation

- ASP.NET MVC 3 provides a mechanism that can make a remote server call in order to validate a form field without posting the entire form to the server
- This is useful when you have a field that cannot be validated on the client and is therefore likely to fail validation when the form is submitted.

Add the following method to the controller class

```
public JsonResult IsUserNameAvailable(string userName)
{
    if (userName != "Demo")
        return Json(true, JsonRequestBehavior.AllowGet);
    string suggestedUID = String.Format("{0} is not available.", userName);
    return Json(suggestedUID, JsonRequestBehavior.AllowGet);
}
```

Add the attribute Remote to the Model class Property

```
public class Employee
{
        //Home is name of controller and IsUserNameAvailable is action method.
        [Remote("IsUserNameAvailable", "Home")]
        public string UserName { get; set; }

      //...
}
```

Using CustomValidation Attribute

1. Add the following class to project

```
public class DesignationValidator
{
    public static ValidationResult IsDesignationValid(string designation, ValidationContext context)
    {
        if (string.IsNullOrWhiteSpace(designation))
            return new ValidationResult("Designation cannot be null or white space");
        if (designation.ToLower().Equals("senior") || designation.ToLower().Equals("junior"))
            return ValidationResult.Success;
        return new ValidationResult("Designation can be either senior or junior only");
    }
}
```

Note: The first parameter of the method should be datatype of the property with which we want to attach CustomValidation Attribute.

2. Add the following property to Employee class

```
[CustomValidation(typeof(DesignationValidator), "IsDesignationValid")]
    public string Designation { get; set; }
```

Note: CustomValidation can be provided for Employee class also but in that case the first parameter data type of the method should be "Employee".

Model Level Validation using IValidatableObject

The IValidatableObject interface enables you to perform model-level validation, and enables you to provide validation error messages specific to **the state of the overall model, or between two properties within the model**.

Model has to implement **IValidatableObject** interface and add following method to model class

```
public IEnumerable<ValidationResult> Validate(ValidationContext validationContext)
{
  ValidationResult vr = null;
  if (Designation.ToLower() == "senior" && Rating < 5)
      vr = new ValidationResult("Invalid  designation based on rating");
  return new List<ValidationResult>() { vr };
```

```
}
```

Note: This particular method is only called after all other validations have passed successfully.

Add a post action method in the controller

```
[HttpPost]
public ActionResult Employee(Employee emp)
{
 If (ModelState.IsValid)
   return View();
 return View(emp);
}
```

To turn off client side validation in view

```
@{  Html.EnableClientValidation(false); }
```

Developing Custom Unobtrusive Client Side Validation

Step1: Add a class to project:

```
public class CustomRangeAttribute : ValidationAttribute, IClientValidatable
{
   private const string DefaultErrorMessage = "'{0}' must be a date between {1} and {2}";
   public int Min { get; set; }
   public int Max { get; set; }
   //Constructor with two parameters
   public CustomRangeAttribute(int min, int max)
      : base(DefaultErrorMessage)
   {
      Min = min;
      Max = max;
   }
   public override bool IsValid(object value)
   {
      if (value == null)
      { return true; }
      int n = (int)value;
      return Min <= n && n <= Max;
   }
   public override string FormatErrorMessage(string name)
```

```
    {
        return String.Format(ErrorMessageString, name, Min, Max);
    }
    //Member of IClientValidatable for Server Side Validation.
    public IEnumerable<ModelClientValidationRule> GetClientValidationRules(ModelMetadata metadata,
ControllerContext context)
    {
        return new[] { new ModelClientValidationRangeRule(FormatErrorMessage(metadata.GetDisplayName()), Min,
Max) };
    }
}

public class ModelClientValidationRangeRule : ModelClientValidationRule
{
    public ModelClientValidationRangeRule(string errorMessage, int min, int max)
    {
        ErrorMessage = errorMessage;
        ValidationType = "customrange";
        ValidationParameters["min"] = min.ToString();
        ValidationParameters["max"] = max.ToString();
    }
}
```

Note: The method GetValidationRules returns an array of ModelClientValidationRule instances. Each of these instances represents metadata for a validation rule that is written in JavaScript and will be run in the client. This is purely metadata at this point and the array will get converted into JSON and emitted in the client so that client validation can hook up all the correct rules.

Step2: Facilitate Client Side Validation

Server-side, the validation parameters are written to the rendered HTML as attributes on the form inputs. These attributes are then picked up by some client-side helpers that add the appropriate client-side validation. The RangeDateValidator.js script below contains both the custom validation plugin for jQuery and the plugin for the unobtrusive validation adapters (each section is called out by comments in the script.

Add the following script to View

```
@section scripts
```

```
{
    <script>
        //The adapter to support ASP.NET MVC unobtrusive validation
        $.validator.unobtrusive.adapters.add('customrange', ['min', 'max'], function (options) {
            var params = {
                min: options.params.min,
                max: options.params.max
            };
            options.rules['customrange'] = params;
            if (options.message) {
                options.messages['customrange'] = options.message;
            }
        });

        //Validator function
        $.validator.addMethod('customrange',
        function (value, element, param) {
            if (value == null)
                return true;
            //To Convert string to number for comparison.
            var min = Number(param.min);
            var max = Number(param.max);
            var intValue = Number(value)
            return (intValue >= min && intValue <= max);
        });
    </script>
}
```

Note: You can see that the validator name ("customrange") matches on both client and server, as do the parameters ("min" and "max").

Step 3: Use the attribute in Model class:

```
[CustomRange(18, 60)]
public int Age { get; set; }
```

Step 4: Include in View:

```
<div class="editor-label">
```

```
    @Html.LabelFor(model => model.Age)
</div>
<div class="editor-field">
    @Html.EditorFor(model => model.Age)
    @Html.ValidationMessageFor(model => model.Age)
</div>
```

Using Data Validation Annotators with MetaData class.

```
public class UserProfileMetadata
{
        [Required(ErrorMessage = "FirstName is required")]
        public string FirstName { get; set; }
        [Required(ErrorMessage = "LastName is required")]
        public string LastName { get; set; }
        [Required(ErrorMessage = "UserName is required")]
        public string UserName { get; set; }
        [Required(ErrorMessage = "Password is required")]
        [DataType(DataType.Password)]
        public string Password { get; set; }
        [Email(ErrorMessage = "Please enter a valid Email Address.")]
        public string EmailId { get; set; }
}
[System.ComponentModel.DataAnnotations.MetadataType(typeof(UserProfileMetadata))]
public partial class UserProfile
{  . . . }
```