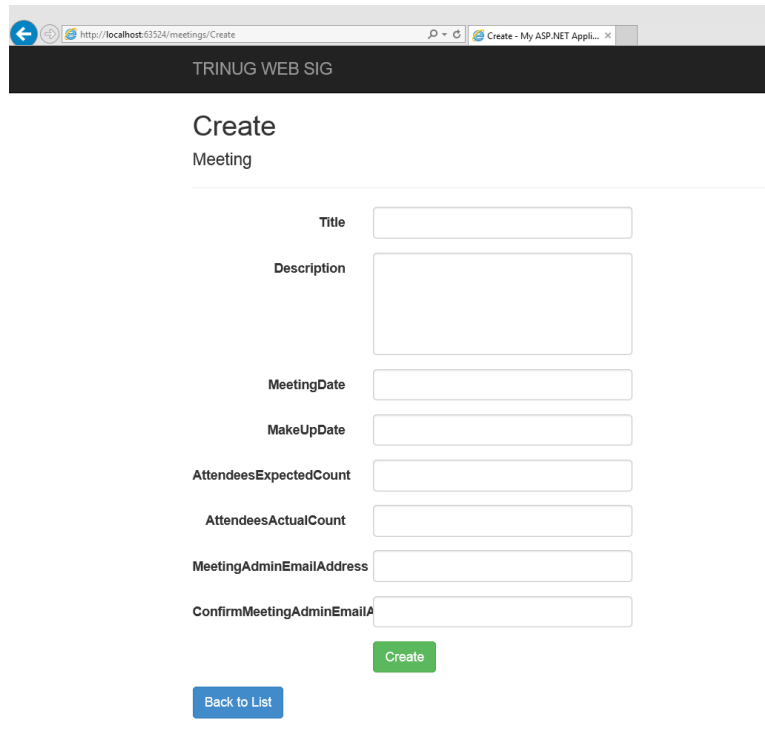


Lab 2 – Working With Data Validation

Step 1: Opening the existing solution and reviewing the contents

The beginning Lab 2 solution is an extension of the completed Lab 1 solution. A few new properties have been added to the Meeting model, specifically the MakeUpDate, AttendeesExpectedCount, AttendeesActualCount, MeetingAdminEmailAddress and ConfirmMeetingAdminEmailAddress. Controls for these new properties have been added to the Create and Edit views.



The screenshot shows a web browser window with the address bar displaying 'http://localhost:63334/meetings/Create'. The page has a dark header with 'TRINUG WEB SIG'. Below the header, the page title is 'Create Meeting'. The form contains several input fields: 'Title', 'Description', 'MeetingDate', 'MakeUpDate', 'AttendeesExpectedCount', 'AttendeesActualCount', 'MeetingAdminEmailAddress', and 'ConfirmMeetingAdminEmailAddress'. At the bottom of the form, there is a green 'Create' button and a blue 'Back to List' button.

The Meeting repository has also been updated to reflect the new add model properties.

Run the project to ensure everything is working properly. You should be able to create a blank meeting record.

Step 2: Marking Model Properties [Required]

The [Required] data annotation specifies that the model property cannot be left empty. In this step we'll add [Required] to a number of model properties.

```
[Required]  
public string Title { get; set; }
```

Optionally you can add your own custom validation error message like this:

```
[Required(ErrorMessage = "No, you can't leave without a title!")]
```

Open up the Meeting model and mark the following fields as [Required]:

```
Title: [Required]  
Description: [Required]  
AttendeesExpectedCount: [Required]
```

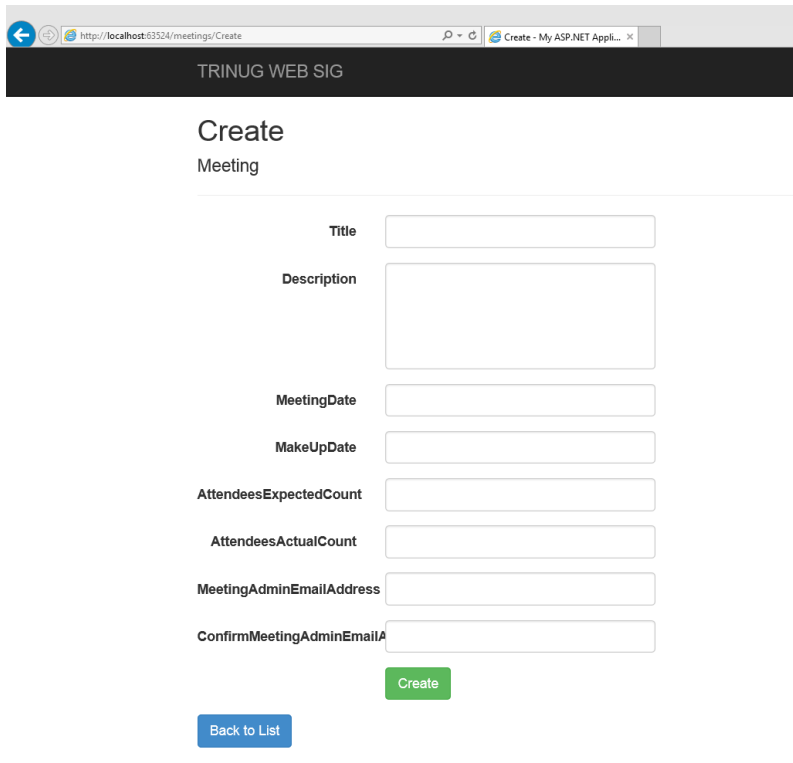
```
MeetingAdminEmailAddress: [Required]
ConfirmMeetingAdminEmailAddress: [Required]
```

Extra: Add a custom validation message to one or more of the model properties.

Note: Before you can run the application you're going to need to add the following using statement:

```
using System.ComponentModel.DataAnnotations;
```

Run the project and create a new meeting. Try to save it immediately and you should see the [Required] validation you just added working.



The screenshot shows a web browser window with the URL `http://localhost:63324/meetings/Create`. The page title is "TRINUG WEB SIG". The main heading is "Create Meeting". Below the heading is a form with the following fields: "Title", "Description", "MeetingDate", "MakeUpDate", "AttendeesExpectedCount", "AttendeesActualCount", "MeetingAdminEmailAddress", and "ConfirmMeetingAdminEmailA". At the bottom of the form are two buttons: a green "Create" button and a blue "Back to List" button.

It didn't work did it? It allowed you to add a blank record again, right? Well that's not good, not good at all. Let's fix that by making sure we're validating the model on the server. We need to check the `ModelState.IsValid` flag to see if the model properties pass all the validation attributes.

Revise your Meetings controller code to look like this:

```
// POST: Meetings/Create
[HttpPost]
public ActionResult Create(Meeting meeting)
{
    if (ModelState.IsValid)
    {
        try
        {
            //Take the new meeting which has been posted and save it.
            meetingRepository.Add(meeting);
            return RedirectToAction("Index");
        }
        catch
        {
            //If there is an exception return the form view.
        }
    }
}
```

```

        return View(meeting);
    }
}
else
{
    return View(meeting);
}
}

```

Now try to run the project again and try to create a blank meeting. You should see something like this now.

TRINUG WEB SIG

Create Meeting

Title
The Title field is required.

Description
The Description field is required.

MeetingDate

MakeUpDate

AttendeesExpectedCount
The AttendeesExpectedCount field is required.

AttendeesActualCount
The AttendeesActualCount field is required.

MeetingAdminEmailAddress
The MeetingAdminEmailAddress field is required.

ConfirmMeetingAdminEmailAddress
The ConfirmMeetingAdminEmailAddress field is required.

Now we're cooking with propane. 😊

Another thing you might want to consider at this point is pre-populating model properties when a new meeting is created. Perhaps you would like to set the initial meeting date to a week from the time the meeting record is created, the make-up meeting date to a week after the initial meeting date and setting the expected & actual attendee counts both equal to zero. To accomplish this you need to instantiate a new meeting object, assign property values and then send that new meeting object to the view.

Add the following constructor code to the Meeting class (it doesn't matter if you put it at the top or the bottom of the class code):

```

public Meeting()
{
    MeetingDate = DateTime.Now.AddDays(14);
    MakeUpDate = DateTime.Now.AddDays(21);
    AttendeesExpectedCount = 0;
    AttendeesActualCount = 0;
}

```

Here's what it'll look like if you put it at the bottom of the class.

```
[Required]
6 references
public string MeetingAdminEmailAddress { get; set; }

[Required]
6 references
public string ConfirmMeetingAdminEmailAddress { get; set; }

8 references | 0/1 passing
public Meeting()
{
    MeetingDate = DateTime.Now.AddDays(14);
    MakeUpDate = DateTime.Now.AddDays(21);
    AttendeesExpectedCount = 0;
    AttendeesActualCount = 0;
}
```

We're almost done. The last thing we have to do is create a Meeting object to send down to the view. We will do this in the Create action (GET) in the Meeting controller. Revise the code to look like this:

```
var meeting = new Meeting();
return View(meeting);
```

It should look like this in Visual Studio:

```
// GET: Meetings/Create
0 references
public ActionResult Create()
{
    //Return the view which contains the form to enter a new meeting.
    var meeting = new Meeting();
    return View(meeting);
}

// POST: Meetings/Create
[HttpPost]
0 references
public ActionResult Create(Meeting meeting)
{
    if (ModelState.IsValid)
    {
        try
        {
            //Take the new meeting which has been posted and save it.
            meetingRepository.Add(meeting);
            return RedirectToAction("Index");
        }
        catch
        {
            //If there is an exception return the form view.
            return View(meeting);
        }
    }
    else
    {
        return View(meeting);
    }
}
```

Run the project and create a new meeting. You should see something like this:

TRINUG WEB SIG

Create Meeting

Title

Description

Meeting Date

Make Up Date

Expected Attendance

Actual Attendance

Meeting Admin Email

Confirm Email Address

© 2014 @RobZelt @JMDuffy

Let's continue adding more validation attributes to the model.

Step 3: Specifying a [StringLength] for Model Properties

The [StringLength] attribute specifies the maximum (and optionally the minimum) number of characters required for a model property.

```
[StringLength(350)]  
public string Description { get; set; }
```

Optionally you can specify a minimum length and/or an error message to display if the string length is invalid.

```
[StringLength(350, MinimumLength = 5, ErrorMessage = "Ya gotta gimme somethin'. I need between  
5 and 350 characters!")]
```

Open up the Meeting model and mark the following fields with these string lengths:

```
Title: [StringLength(80)]
```

```
Description: [StringLength(350, MinimumLength = 5, ErrorMessage = "Ya  
gotta gimme somethin'. I need between 5 and 350 characters!")]
```

```
MeetingAdminEmailAddress: [StringLength(150, ErrorMessage = "150  
characters ain't enough for you huh?")]
```

```
ConfirmMeetingAdminEmailAddress: [StringLength(150, ErrorMessage =  
"150 characters ain't enough for you huh?")]
```

Run the project and create a new meeting. Populate the title and description with text long enough to violate the StringLength attribute. You should see something like this now.

http://localhost:63524/meetings/Create

TRINUG WEB SIG

Create Meeting

Title

The field Title must be a string with a maximum length of 80.

Description

Ya gotta gimme somethin'. I need between 5 and 350 characters!

MeetingDate

MakeUpDate

AttendeesExpectedCount

AttendeesActualCount

MeetingAdminEmailAddress

ConfirmMeetingAdminEmailAddress

© 2014 @RobZelt @JMDuffy

Step 4: Specifying a [Display] for Model Properties

The [Display] attribute specifies the text to be used for the on-screen label describing the model property.

```
[Display(Name = "Make Up Date")]
public DateTime? MakeUpDate { get; set; }
```

Open up the Meeting model and mark the following fields with these [Display] attribute settings:

```
MeetingDate: [Display(Name = "Meeting Date")]
MakeUpDate: [Display(Name = "Make Up Date")]
AttendeesExpectedCount: [Display(Name = "Expected Attendance")]
AttendeesActualCount: [Display(Name = "Actual Attendance")]
MeetingAdminEmailAddress: [Display(Name = "Meeting Admin Email")]
ConfirmMeetingAdminEmailAddress: [Display(Name = "Confirm Email Address")]
```

Run the project and create a new meeting. You should see the new labels you specified for the fields above. You should see something like this now.

TRINUG WEB SIG

Create Meeting

Title

Description

Meeting Date

Make Up Date

Expected Attendance

Actual Attendance

Meeting Admin Email

Confirm Email Address

© 2014 @RobZelt
@JMDuffy

I don't care what people say, things are starting to shape up. 😊

Step 5: Specifying a [Range] for Model Properties

The [Range] attribute specifies the acceptable high and low values for a numeric model property.

```
[Range(0, 200)]
public int? AttendeesActualCount { get; set; }
```

Optionally you can specify an error message to display if the value entered violates the range constraints.

```
[Range(0, 200, ErrorMessage = "I have an idea. Why don't you enter a value between 0 and 200.")]
```

Open up the Meeting model and set low and high range values for the following fields with these [Range] attribute settings:

```
AttendeesExpectedCount: [Range(0, 200)]
AttendeesActualCount: [Range(0, 200)]
```

Save and run the project. Create a new meeting and enter values for the attendee count fields that violate the range constraints. Click Create and you should see something like this:

TRINUG WEB SIG

Create Meeting

Title: Sample Meeting

Description: This is the description.

Meeting Date: 07-15-2014

Make Up Date: 07-22-2014

Expected Attendance: 400
The field Expected Attendance must be between 0 and 200.

Actual Attendance: 500
The field Actual Attendance must be between 0 and 200.

Meeting Admin Email: gpugh@nc.rr.com

Confirm Email Address: gpugh@nc.rr.com

Create

Back to List

© 2014 @RobZett @JMDuffy

Step 6: Specifying a [DisplayFormat] for Model Properties

The [DisplayFormat] attribute specifies how the data should look when displayed on a page. We will use it to format the dates on the form.

```
[DisplayFormat(DataFormatString = "{0:MM-dd-yyyy}")]  
public DateTime MeetingDate { get; set; }
```

Optionally you can add the ApplyFormatInEditMode to use the display format on data entry pages.

```
[DisplayFormat(DataFormatString = "{0:MM-dd-yyyy}", ApplyFormatInEditMode = true)]
```

Open up the Meeting model and set the [DisplayFormat] attribute for the following model properties:

```
MeetingDate: [DisplayFormat(DataFormatString = "{0:MM-dd-yyyy}",  
ApplyFormatInEditMode = true)]  
MakeUpDate: [DisplayFormat(DataFormatString = "{0:MM-dd-yyyy}",  
ApplyFormatInEditMode = true)]
```

Save and run the project. The date fields on your form should look like this:

TRINUG WEB SIG

Create

Meeting

Title

Description

Meeting Date

Make Up Date

Expected Attendance

Actual Attendance

Meeting Admin Email

Confirm Email Address

© 2014 @RobZeit
@JMDuffy

Step 7: Specifying a [DataType] for Model Properties

The [DataType] attribute specifies the data type for a model property. This is useful because the browser can pick up on the data type and display an appropriate control (like a date picker for a date).

```
[DataType(DataType.Date)]  
public DateTime? MakeUpDate { get; set; }
```

Open up the Meeting model and delete (or comment out) the DisplayFormat attribute you added in the previous step and add the DateType attribute for the two date fields.

```
MeetingDate: [DataType(DataType.Date)]  
MakeUpDate: [DataType(DataType.Date)]
```

Things should be looking like this in your model now.

```

[Required]
[StringLength(80)]
7 references
public string Title { get; set; }

[Required]
[StringLength(350, MinimumLength = 5, ErrorMessage = "Ya gotta gimme somethin'. I need between 5 and 350 characters!")]
6 references
public string Description { get; set; }

[DataType(DataType.Date)]
[Display(Name = "Meeting Date")]
7 references
public DateTime? MeetingDate { get; set; }
[DataType(DataType.Date)]
[Display(Name = "Make Up Date")]
1 reference
public DateTime? MakeUpDate { get; set; }

[Required]
[Display(Name = "Expected Attendance")]
7 references
public int AttendeesExpectedCount { get; set; }

[Display(Name = "Actual Attendance")]
7 references
public int AttendeesActualCount { get; set; }

[Required]
[StringLength(150, ErrorMessage = "150 characters ain't enough for you huh?")]
[Display(Name = "Meeting Admin Email")]
6 references
public string MeetingAdminEmailAddress { get; set; }

[Required]
[StringLength(150, ErrorMessage = "150 characters ain't enough for you huh?")]
[Display(Name = "Confirm Email Address")]
6 references
public string ConfirmMeetingAdminEmailAddress { get; set; }

8 references
public Meeting()
{
    MeetingDate = DateTime.Now.AddDays(14);
    MakeUpDate = DateTime.Now.AddDays(21);
    AttendeesExpectedCount = 0;
    AttendeesActualCount = 0;
}
}
}

```

Save and run the project and create a new meeting. If you're using Internet Explorer you won't notice much of a difference. If you're using Google Chrome you notice that the date fields now have a data picker associated with them.

Step 8: Specifying a [RegularExpression] for Model Properties

The [RegularExpression] attribute specifies the regular expression string used to validate the model property. We will use it to validate the two email address model properties.

```

[RegularExpression(@"^([a-zA-Z0-9_\-\.]+)@((\[[0-9]{1,3}\. [0-9]{1,3}\. [0-9]{1,3}\.))|(([a-zA-Z0-9\-\ ]+\.))+)([a-zA-Z]{2,4}|[0-9]{1,3})(\ )?$", ErrorMessage = "Please enter a valid e-mail address")]
public string MeetingAdminEmailAddress { get; set; }

```

Tip: We're using an email model property to illustrate the [RegularExpression] attribute but you could also just use the [EmailAddress] attribute instead.

Open up the Meeting model and set the [RegularExpression] attribute for the following model properties: (NO, we're not going to make you type all that! You'll find a RegexCode.txt file in the beginning lab sample you can copy and paste the regular expression from.)

```
MeetingAdminEmailAddress: [RegularExpression(@"^([a-zA-Z0-9_\-\.\.])@((\[[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.)|((([a-zA-Z0-9\-\.]+\.)+))([a-zA-Z]{2,4}|[0-9]{1,3})(\)?)$")]
```

Save and run the project. Create a new meeting, populate the fields and enter invalid email addresses. You should see something like this:

The screenshot shows a web browser window with the URL `http://localhost:63524/meetings/Create`. The page title is "TRINUG WEB SIG". Below the title is a heading "Create Meeting". The form contains the following fields:

- Title:** Sample title
- Description:** This is sample text.
- Meeting Date:** 7/15/2014
- Make Up Date:** 7/22/2014
- Expected Attendance:** 0
- Actual Attendance:** 0
- Meeting Admin Email:** Jim (with error message: "Please enter a valid e-mail address")
- Confirm Email Address:** Chuck (with error message: "Please enter a valid e-mail address")

At the bottom of the form are two buttons: "Create" (green) and "Back to List" (blue). Two red arrows point from the left towards the "Meeting Admin Email" and "Confirm Email Address" fields.

© 2014 @RobZelt
@JMDuffy

Step 9: Utilizing the [Compare] attribute for Model Properties

The [Compare] attribute determines if two model properties contain the same information. It is often used to insure that password and confirm password fields match.

```
[Compare("MeetingAdminEmailAddress", ErrorMessage = "Email addresses must match.")]
public string ConfirmMeetingAdminEmailAddress { get; set; }
```

Optionally you can specify the error message displayed when the two values do not match.

```
[Compare("MeetingAdminEmailAddress", ErrorMessage = "Email addreses must match.")]
```

Open up the Meeting model and add the [Compare] attribute to the ConfirmAdminEmailAddress property.

```
[Compare("MeetingAdminEmailAddress", ErrorMessage = "Email addresses
must match.")]
```

Save and run the project. Create a new meeting, populate the fields and different email addresses in the two email fields. You should see something like this:

TRINUG WEB SIG

Create Meeting

Title

Description

Meeting Date

Make Up Date

Expected Attendance

Actual Attendance

Meeting Admin Email

Confirm Email Address
 Email addresses must match.

© 2014 @RobZeit @JMDuffy

Implementing Client-Side Validation

The validation we've done so far has all been performed on the server side. Today's modern web applications require more and more operations be performed in the browser and validation is one of them. Fortunately it's pretty easy to do. You need to make sure the configuration settings are correct in web.config and you need to make sure that the jquery validation scripts are referenced on the form.

Step 1: Updating Web.Config

Open the web.config file and make sure the ClientValidationEnabled and UnobtrusiveJavaScriptEnabled app settings are set to true.

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

Step 2: Adding jQuery Validation script tags

Make sure the jquery validation scripts are referenced by the form.

```
<script src="@Url.Content("~/Scripts/jquery-1.10.2.min.js")"></script>
<script src="@Url.Content("~/Scripts/jquery.validate.min.js")"></script>
<script src="@Url.Content("~/Scripts/jquery.validate.unobtrusive.min.js")"></script>
```

We've added all three lines to the top of the Create view.

```
Create.cshtml  MeetingsController.cs  Meeting.cs

@model MvcDemo.Models.Meeting

@{
    ViewBag.Title = "Create";
}

<h2>Create</h2>
<script src="@Url.Content("~/Scripts/jquery-1.10.2.min.js")"></script>
<script src="@Url.Content("~/Scripts/jquery.validate.min.js")"></script>
<script src="@Url.Content("~/Scripts/jquery.validate.unobtrusive.min.js")"></script>

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

    <div class="form-horizontal">
        <h4>Meeting</h4>
        <hr />
        @Html.ValidationSummary(true, "", new { @class = "text-danger" })
        <div class="form-group">
            @Html.LabelFor(model => model.Title, htmlAttributes: new { @class = "control-label col-sm-2" })
            <div class="col-sm-10">
                @Html.EditorFor(model => model.Title, new { htmlAttributes = new { @class = "form-control" } })
            </div>
        </div>
    </div>
}
```



Save and run the project. Create a new meeting record and immediately try to enter 500 in the estimated count. You should see the validation message appear right away.

The End!

That should give you enough to chew on for now. Good luck learning more about ASP.NET MVC!