Hands-On Lab

Building Rich Web Applications Using ASP.NET MVC 2

Lab version: 1.0.0

Last updated: 12/10/2010
## CONTENTS

### OVERVIEW

---

### EXERCISE 1: UI RENDERING METADATA IN ASP.NET MVC 2

- Task 0 – Exploring the Solution
- Task 1 – Modifying Markup to Make Use New LabelFor and DisplayFor Methods
- Task 2 – Adding Metadata for UI Rendering
- Exercise 1: Verification

---

### EXERCISE 2: TEMPLATE HELPERS IN ASP.NET MVC 2

- Task 0 – Exploring the Solution
- Task 1 – Creating a Custom Display Template for Rendering an Activity
- Task 2 – Creating a Custom Editor Template for Rendering the ActivityCreateModel
- Task 3 – Using the UIHint Attribute for Rendering the Phone Number
- Exercise 2: Verification

---

### EXERCISE 3: CLIENT AND SERVER SIDE VALIDATION IN ASP.NET MVC 2

- Task 0 – Exploring the Solution
- Task 1 – Removing jQuery Client Side Validation
- Task 2 – Removing Imperative Server Side Model Validation
- Task 3 – Adding Model Validation Metadata
- Task 4 – Enabling Client Side Validation
- Exercise 3: Verification

---

### EXERCISE 4: ASP.NET MVC AREAS

- Task 1 – Creating the New Account Area
- Task 2 – Moving the Account Controller and Views from the Main Project to the new Area
- Task 3 – Registering Account Routes
- Task 4 – Modifying the Code to Navigate Between the Main and Account Area
- Exercise 4: Verification

---

### SUMMARY
Overview

This Hands-On Lab is based on the Plan My Night demo and uses the following technologies: ASP.NET MVC 2, Visual Studio 2010, .Net Framework 4.0 and ASP.NET Ajax Library Beta. In order to simplify this Lab the original demo solution was reduced in functionality, nevertheless, if you wish to explore the whole demo you can find it in http://msdn.microsoft.com.

ASP.NET MVC is a framework built on top of ASP.NET for creating web applications that use the model-view-controller pattern. Among other features, the ASP.NET MVC framework provides complete control over your HTML Markup, AJAX Integration, Separation of concerns, and Testability. For introductory labs on the MVC Framework, you can refer to http://r.ch9.ms/vs2010b2f.

In this Hands-On Lab, you will go through four new features that have been included in the present release of the MVC Framework.

- **UI Metadata**: It is a common problem when creating applications to separate the concerns when rendering views that are tied to an entity. UI Metadata is a set of attributes that can be used to describe how an entity must be rendered. Altogether with the Templated Helpers, the UI Metadata can be used to remove the model-related information from the views, providing an new level of abstraction.

- **Validation**: First introduced in ASP.NET Dynamic Data, and now added in ASP.NET MVC 2 Framework, data annotations validation support allows to have the validation logic added through Metadata attributes. Similarly as with UI Metadata, when validating an entity, you have certain data and validation rules that are highly tied to the entity that is being validated. Throughout this lab you will see how to make use of the Metadata Attributes for adding both Server and Client-Side validation.

- **Templated Helpers** allow you to automatically associate edit and display elements with the entity being rendered. By doing this, you avoid having to define how data will be rendered in every view. Instead, you define a template, that will be rendered everytime certain data-type is displayed.

- **MVC Areas**: It is a common scenario on large web applications having different functionality which have little or no relation at all with the rest of the components and can be grouped or partitioned. MVC Areas provide a way of encapsulating these areas in the form of folders in your project which will then converge into a single web application. Because areas are self contained, this helps managing the complexity when building large web applications since it provides a simplified environment for working with multiple teams.

In this lab, you will begin with an MVC1-like application, and enhance it with these new features while learning the benefits that each one of them provide.

This Hands-On Lab assumes that the developer has basic experience with HTML, JavaScript, the ASP.NET MVC Framework, and the Entity Framework.
Objectives
In this Hands-On Lab, you will learn how to:

- Make use of the UI Metadata Attributes for reducing the model-related data needed in the views, when rendering entities present in your data model.
- Add Server and Client-Side Data Annotations Validation to an existing MVC Application.
- Use Template Helpers for defining how to render specific data-types.
- Configure and make use of MVC Areas for structuring an MVC Application

System Requirements
You must have the following items to complete this lab:

- Microsoft Visual Studio 2010
- Microsoft SQL 2005 or Microsoft SQL 2008 (Express edition or above)
- ASP.NET MVC 2

Setup
All the requisites for this lab are verified using the Configuration Wizard. To make sure that everything is correctly configured, follow these steps.

**Note:** To perform the setup steps you need to run the scripts in a command window with administrator privileges.

1. Run the Configuration Wizard for the Training Kit if you have not done it previously. To do this, run the CheckDependencies.cmd script located under the Setup folder. Install any pre-requisites that are missing (rescanning if necessary) and complete the wizard.

**Note:** For convenience, much of the code you will be managing along this lab is available as Visual Studio code snippets. The CheckDependencies.cmd file launches the Visual Studio installer file that installs the code snippets.

Exercises
This Hands-On Lab is comprised by the following exercises:
1. Exercise 1: UI Rendering Metadata in Asp.Net MVC2
2. Exercise 2: Client and Server Side Validation in Asp.Net MVC2
3. Exercise 3: Templated Helpers in Asp.Net MVC2
4. Exercise 4: Asp.Net MVC Areas

Estimated time to complete this lab: **120 minutes**.

**Note:** Each exercise is accompanied by an **End** folder containing the resulting solution you should obtain after completing the exercises. You can use this solution as a guide if you need additional help working through the exercises.

**Note:** Each exercise contains a Visual Basic and a C# version; Inside the **End/Begin** solution folder you will find two folders: **VB**, containing the Visual Basic version of the exercise, and **C#**, containing the C# version of it.

---

**Exercise 1: UI Rendering Metadata in ASP.NET MVC 2**

In this Exercise, you will learn how to add Metadata Attributes to your model entities, and then take advantage of them to help rendering entity instances in an MVC View. These attributes are used to construct the Model Metadata objects that will be used in the views.

The following attributes can be added to your entity to describe how it should be rendered:

- **[DisplayName]** defines the name of the property that will be displayed when rendering the entity.
- **[HiddenInput]** generates a hidden input field for the property. If the DisplayValue flag is set to true, it also generates a visible field containing its value; though the input field will still be hidden.
- **[UIHint]** sets the TemplateHint property with the given value. This property is used to suggest which template to use when rendering this model.
- **[DataType]** sets the DataTypeName property with the specified value. This property is used upon rendering, to format the data.
- **[ScaffoldColumn]** sets both the ShowForEdit and ShowForDisplayed properties which defines whether the model should be displayed in editable/list or detail views respectively.
• **[ReadOnly]** sets the IsReadOnly property with its value.

• **[DisplayFormat]** allows defining the value of different properties which describe how to display the model. The available properties are: NullDisplayText, DataFormatString, ApplyFormatInEditMode, EditFormatString, and ConvertEmptyStringToNull

**Note:** Most of these attributes are included in System.ComponentModel and System.ComponentModel.DataAnnotations namespaces; the exception is HiddenInput which is included in System.Web.Mvc

Throughout this lab you will decorate your model entities with attributes to render the Activities in the Create view. Later on, you will extract this data from the view to create a Templated Helper for the Activity Class.

**Task 0 – Exploring the Solution**

In this task, you will explore the assets provided with this Lab in order to get a basic idea of what has to be done to implement the functionality it shows using MVC 1.0

1. Open **Microsoft Visual Studio 2010**. Click **Start | All Programs | Microsoft Visual Studio 2010 | Microsoft Visual Studio 2010**.

2. Open the solution file **Begin.sln** located under **\Ex1-UIMetadata\begin\** (Choosing the folder that matches the language of your preference)

3. The **PlanMyNight** solution consist of an MVC based application that demonstrate different scenarios created using the version 1.0 of the ASP.NET MVC framework.

![Solution Explorer](image)

**Figure 1**

*Exploring the PlanMyNight solution (C#)*
4. The Details view of the Activities controller show all the relevant information about the Activity entity. To review this file, expand the Views\Activities folder and open the Details.aspx file.

5. Review the Details.aspx file and notice the use of `Html.Encode` and `<label>` tags to show data. Also, you should notice the specific use of the rating CSS class to show the activity current rating. To do this, expand the `<div>` element with the CSS `items` class applied.

6. You can verify how this view is rendered by navigating directly to an activity. To do this, start a new instance of the application by pressing `CTRL + F5` within Visual Studio. When the browser’s window appears, edit the address bar to navigate to `http://localhost:50000/`.

7. You should search for activities and click on activity link in the search results. To do this, select `OH` in the `State` dropdown list, enter `43082` in the `Zip` textbox and click the `Search` button. When the result list appears, click on the link of the first item. The Details view will appear.
Figure 3
Exploring the Activity Details view

8. The Details view also let you rate an activity which will turn the rating rendering in a 5-stars control. To do this, select a value in the Rating control at the bottom of the page and lick the Rate link.
9. Close the Browser.

Task 1 – Modifying Markup to Make Use New LabelFor and DisplayFor Methods

ASP.NET MVC 2 introduced several strongly typed HTML helpers in addition to the MVC 1 helpers. These methods use a lambda expression when referencing models or viewmodels which enables compile-time checking of views and better code intellisense inside views.

In this task you will modify the activity details view used to display information for a selected activity and to rate the activity to make use of the new strongly-typed helper methods.

1. Open the Details.aspx file used to render the Activity information. To do this, expand the Views\Activities folder in the PlanMyNight project and double click the Details.aspx file to open it.

2. You will make use of the new strongly-typed helpers to replace the labels and encoded values displayed in the view. Each call to the LabelFor method will replace the <label> tags, and a call to the DisplayFor method will replace the Encode method. To do this, replace the code inside the items <div> element and replace it with the bolded code shown below.

(Code Snippet – Building MVC2 App – ActivityDetailItems CSharp)
<div class="items">
  <p><%= Html.LabelFor(activity=>activity.Rating) %></p><br/>
  <% if(rating > 0) { %>
    <span class="rating rating_"><%= rating.ToString("0.0", System.Globalization.CultureInfo.InvariantCulture).Replace( ".", ",") %></span>
  <% } else { %>
    Awaiting at least one more vote.
  <% } %>
  <p><%= Html.LabelFor(activity => activity.Street) %></p>
  <p><%= Html.DisplayFor(activity => activity.Street) | Html.DisplayFor(activity => activity.City) %>,<br />
    <%= Html.DisplayFor(activity => activity.State) %><br />
    <%= Html.DisplayFor(activity => activity.Zip) %></p>
  <p><%= Html.LabelFor(activity => activity.PhoneNumber) %></p>
  <p><%= Html.DisplayFor(activity => activity.PhoneNumber) %></p>
</div>

(Code Snippet – Building MVC2 App – ActivityDetailItems VB)
**Note:** One of the new features in ASP.NET MVC 2 is the support strongly-typed HTML helpers that use lambda expressions. This enables better compile-time checking of views (at build-time as opposed to runtime), and also enables better code intellisense support within view templates.

These methods use an `Html.HelperNameFor()` naming convention. For example: `Html.TextBoxFor()`, `Html.CheckBoxFor()`, `Html.TextAreaFor()`.

In MVC 1 helper methods you should specify both the control name and the model value to render on it. New helper methods support using a lambda expression to specify both the name/id of the element, as well as the value to render for it.

The resulting HTML rendered is the same as the MVC 1 late-bound version of the helper.

Below is a list of strongly-typed HTML helper methods available:

- `Html.TextBoxFor()`
- `Html.TextAreaFor()`
- `Html.DropDownListFor()`
- `Html.CheckboxFor()`
- `Html.RadioButtonFor()`
- `Html.ListBoxFor()`
- `Html.PasswordFor()`
- `Html.HiddenFor()`
- `Html.LabelFor()`
- `Html.EditorFor()`
- `Html.DisplayFor()`
- `Html.DisplayTextFor()`
- `Html.ValidationMessageFor()`

3. Now, you will verify that the new approach is displaying the data as you expected. To do this, press **CTRL + F5** to execute the solution without debugging.

4. You will open the detailed information for a specific Activity to review the results of the work done. You will see that the activity information is being displayed but the labels for each field are not friendly. To do this, edit your browser’s address bar to navigate to [http://localhost:50000/Activities/Details/49](http://localhost:50000/Activities/Details/49).
Task 2 – Adding Metadata for UI Rendering

One of the classes introduced in MVC 2 is ModelMetadata. This class is intended to provide convenient information about your model entities with the objective of enhancing the displaying and edition of them.

In this task you will create a new metadata class for including attributes for rendering the activity entity.

1. You will create a new class to store the Activity entity metadata. This class will provide simple information of how the entity should be displayed. To do this, expand and right-click on the Model\Entities folder in the PlanMyNight project and select Add | Class... When the Add New Item dialog appears enter ActivityMetadata.cs (C#) or ActivityMetadata.vb (Visual Basic) in the Name field.
Figure 6
*ActivityMetadata class (C#)*

Figure 7
*ActivityMetadata class (Visual Basic)*
2. Add the following namespaces in the `ActivityMetadata.cs (C#)` or `ActivityMetadata.vb (Visual Basic)` class.

   **C#**
   ```csharp
   using System.ComponentModel;
   using System.ComponentModel.DataAnnotations;
   ```

   **Visual Basic**
   ```vb
   Imports System.ComponentModel
   Imports System.ComponentModel.DataAnnotations
   ```

3. The PlanMyNight project uses ADO.NET Entity Data Model to automatically generate classes that represent entities. Manual changes applied to this classes will be lost if the associated class file is regenerated. Conveniently, entity classes are defined as partial classes which allow you to still decorate this entity classes in a file of your control by adding a new partial class definition. To do this, in the `ActivityMetadata.cs (C#)` or `ActivityMetadata.vb (Visual Basic)` class file add a partial class definition named `Activity`.

   **C#**
   ```csharp
   partial class Activity
   {
   }
   ```

   **Visual Basic**
   ```vb
   Partial Class Activity
   End Class
   ```

4. Next, you will associate the Activity entity class with another class containing its metadata by decorating it with the `MetadataType` attribute. This allows you to apply metadata to an unmanaged partial class with no risk of losing changes. To do this, decorate the `Activity` partial class with the `MetadataType` attribute and passing the `ActivityMetadata` type as the parameter.

   *(Code Snippet – Building MVC2 App – ActivityPartialClass CSharp)*

   **C#**
   ```csharp
   [MetadataType(typeof(ActivityMetadata))]
   partial class Activity
   {
   }
   ```
5. The ActivityMetadata class is now associated with the Activity class and you can use the metadata attributes over its properties as if you were decorating the Activity class properties itself. Property names of the metadata class must match the property names of the entity class, but the data type is irrelevant. You will create metadata properties for several entity properties. To do this, create three `object` properties named `Rating`, `PhoneNumber` and `Street` in the ActivityMetadata class.

```csharp
public class ActivityMetadata
{
    public object Rating { get; set; }
    public object PhoneNumber { get; set; }
    public object Street { get; set; }
}
```

```visualbasic
Public Class ActivityMetadata
    Public Property Rating As Object
    Public Property PhoneNumber As Object
    Public Property Street As Object
End Class
```

6. Now that you have the properties available, they can be decorated with any metadata attribute. In this case, you will use the DisplayName attribute to control the label text generated in the Activity details view. To do this, decorate the three properties created with the `DisplayName` attribute as shown in the bolded code below.

```
public class ActivityMetadata
{
    [DisplayName("Rating:")]}
```

(Code Snippet – Building MVC2 App – ActivityPartialClass VB)
public object Rating { get; set; }

[DisplayName("Phone:" )]
public object PhoneNumber { get; set; }

[DisplayName("Address:" )]
public object Street { get; set; }
}

(Code Snippet – Building MVC2 App – ActivityMetadataProperties VB)

Visual Basic

Public Class ActivityMetadata
    <DisplayName("Rating:" )>
    Public Property Rating As Object

    <DisplayName("Phone:" )>
    Public Property PhoneNumber As Object

    <DisplayName("Address:" )>
    Public Property Street As Object
End Class

7. Build the solution by pressing CTRL + SHIFT + B.

Exercise 1: Verification

1. Press CTRL+F5 to run the solution without debugging.

2. You will open the detailed information for a specific Activity to review the results of the work done. To do this, edit your browser’s address bar to navigate to http://localhost:50000/Activities/Details/49.

3. You should notice that the label for each field is managed by the DisplayName attribute parameter specified in the metadata class.
Exercise 2: Template Helpers in ASP.NET MVC 2

In this Exercise, you will learn how to use Templates in your ASP.NET MVC 2 application. You will begin by using the templates provided out of the box by the framework and then create and use custom templates for displaying and creating an Activity.

The Framework has nine pre-defined display templates: Boolean, Decimal, EmailAddress, HiddenInput, Html, Object, String, Text, and Url.

Note: There is no actual difference between the Text and String template.

And seven pre-defined editor templates: Boolean, Decimal, HiddenInput, MultilineText, Object, Password, and String.
Finally, you will see how to use the UIHint attribute to define the template that should when rendering a property. The UIHint attribute sets the value of the TemplateHint property in ModelMetadata. This property has the highest priority when choosing the template to render data.

Task 0 – Exploring the Solution

In this task, you will explore the solution you obtained after completing exercise 1, to see how Views were used in the original Plan My Night application, so as to understand the benefits that MVC 2 templates provide.


2. Open the solution file Begin.sln located under \Ex2-Templates\begin\ (Choosing the folder that matches the language of your preference). Alternatively, you can continue working with the solution obtained after completing exercise 1.

Up to now, the definition of how to render the entities is spread in each view. Create and Details view have the information on how to render an Activity within them.
Figure 9

Exploring the PlanMyNight solution (C#)
3. In the Solution Explorer, double-click on Details.aspx, located under Views\Activities, to open it.
```
var rating = Model.Rating ?? 0;
rating = Math.Round(rating * 2) / 2;

<div class="innerPanel">
  <div class="item">
    <span class="rating">rating: @rating.ToString("0.0", System.Globalization.CultureInfo.InvariantCulture).Replace(",", ",")</span>
    <span class="voting">Voting at least one more vote.</span>
  </div>
</div>

<p>
  @Html.LabelFor(model => model.Street)<br>
  @Html.DisplayFor(model => model.Street)<br>
  @Html.LabelFor(model => model.PhoneNumber)<br>
  @Html.DisplayFor(model => model.PhoneNumber)
</p>
```

Using (Html.BeginForm("Rate", "Activities", new { id = Model.Id }, FormMethod.Post, null)) {
    Rating: <br>
    <br>
    <input type="submit" value="Rate" />
    </form>
```

Figure 11
Details.aspx (C#)
As you can see, the Details view has the information on how to display an Activity. This approach has two big downsides; on one hand, if you wanted to display an activity in another page, you would have to duplicate the code there, while on the other hand, the View is not easy to read, is hard to understand that an activity is being rendered there.

**Note:** DisplayFor is a Helper Method that renders the template for the given property. This method was added on the previous exercise.

Notice the following code: 

```html
<h2><span>%=Html.DisplayNameFor(activity=&gt;activity.Name)%</h2>
```

DisplayFor searches for the default string display template, which defines how to render a String (because the Name property of the Activity class is a string). This template is provided out-of-the-box with MVC 2.

4. In the Solution Explorer, double-click on Create.aspx, located under Views\Activities, to open it.
Figure 13

Create.aspx (C#)
Similar as before, the Create view has the information on how to render the CreateActivityModel view model.

**Task 1 – Creating a Custom Display Template for Rendering an Activity**

In this task, you will create a new Activity display template, which will contain the information on how to render an activity.

1. In the Solution Explorer, right-click on the **Activities** folder, located under **Views**, point to Add and choose New Folder.

2. Type **DisplayTemplates** as the folder name.

**Note:** When searching for the display template that should be used to render a property, there are two things that must be taken into account.

First, you need to know where to locate the template. MVC 2 searches for the template to render as follows:

1. DisplayTemplates folder inside the corresponding Views sub-folder (in this case, Activities)
2. DisplayTemplates folder inside the Views\Shared folder.
4- Uses the pre-defined templates (listed in the overview of this exercise.)

*Note: if you are using Areas, the framework first searches within the Area folder.*

**The difference with the EditorTemplates,** is that the framework searches for the templates inside the EditorTemplates folder instead.

The second thing you need to take into account is the criteria that the framework uses for choosing the template to use.

1- TemplateHint property, from the ModelMetadata (set through the **UIHint** attribute)
2- DataTypeName property, from the ModelMetadata (set through the **DataType** attribute)
3- The name of the Type.

---

3. The first template you will create is the **Rating** template, to do this, right-click on the **DisplayTemplates** folder you have just created, point to **Add** and select **View**...

4. Make sure **Create a partial view (.ascx)** is checked, and type **Rating** as **View name**

---

**Figure 15**

*DisplayTemplates Folder*
The Rating display template will contain the information on how to display the rating of an activity. This will clean up the View of unnecessary code that is strongly related to how the Rating is displayed.

5. Copy the following code inside Rating.ascx:

(Code Snippet – Building MVC2 App – Rating Display Template CSharp)

```csharp
<% double rating; double.TryParse(Model as string, out rating); rating = Math.Round(rating * 2) / 2; %>
<p>
<%=Html.LabelFor(m=>m) %>
<% if(rating > 0) { %>
  <span class="rating rating_<%=rating.ToString("0.0", System.Globalization.CultureInfo.InvariantCulture).Replace(".", ",")%>">
    <%=rating.ToString("0.0")%>
  </span>
<% } %>
```
<% } else { %>
   Awaiting at least one more vote.
<% } %>
</p>

(Code Snippet – Building MVC2 App – Rating Display Template VB)

ASP.NET (Visual Basic)

<% Dim rating As Double
    Double.TryParse(Model, rating)
    rating = Math.Round(rating * 2) / 2
%>

<p>
<%= Html.LabelFor(Function(m) m)%>
<% If rating > 0 Then%>
   <span class="rating rating_<%= rating.ToString("0.0", System.Globalization.CultureInfo.InvariantCulture).Replace(".", "_")%>">
<%= rating.ToString("0.0")%></span>
<% Else%>
   Awaiting at least one more vote.
<% End If%>
</p>

Note: This code is similar to the code in the Details.aspx View. Later on this exercise you will remove the unnecessary code from this view.

6. Next, you will create the Activity display template, to do this, right-click on the DisplayTemplates folder you have just created, point to Add and select View...

7. Make sure Create a partial view (.ascx) is checked, and type Activity as View name

8. Check the Create a strongly-typed view checkbox, and type PlanMyNight.Models.Entities.Activity (C#) PlanMyNight.Activity (VB) as View data class
Figure 17

Add View Activity.ascx dialog (C#)
The Activity display template will contain the information on how to display the activity. For simplicity you will simply add the same code that is used in the Details.aspx View. Later on this exercise you will create a custom template to display the phone number.

9. Copy the following code inside **Activity.ascx**:

(Code Snippet – Building MVC2 App – Activity Display Template CSharp)

```
<%-- Building MVC2 App -->

<asp:Content ID="Content0" ContentPlaceHolderID="MainContent" runat="server">
    <div class="innerPanel">
        <h2><span>%=Html.DisplayNameFor(activity=>activity.Name) %</span></h2>
        <div class="items">
            <%= Html.DisplayFor(activity => activity.Rating, "Rating") %>
            <p>%=Html.LabelFor(activity => activity.Street)%>
            <%=Html.DisplayFor(activity => activity.Street)%>
            | %=Html.DisplayNameFor(activity => activity.City)%>
            %=Html.DisplayFor(activity => activity.State)%>
            %=Html.DisplayNameFor(activity => activity.Zip)%>
            <p>%=Html.LabelFor(activity => activity.PhoneNumber)%>
            <%=Html.DisplayFor(activity => activity.PhoneNumber)%>
        </div>
    </div>
</asp:Content>

```

**Figure 18**
Add View Activity.ascx dialog (VB)
ASP.NET (Visual Basic)

```vbnet
<div class="innerPanel">
  <h2><span>%= Html.DisplayFor(Function(activity) activity.Name) %</span></h2>
  <div class="items">
    <p>%= Html.DisplayFor(Function(activity) activity.Rating, "Rating") %</p>
    <p>%= Html.LabelFor(Function(activity) activity.Street) %<%= Html.DisplayFor(Function(activity) activity.City) %<%= Html.DisplayFor(Function(activity) activity.State) %<%= Html.DisplayFor(Function(activity) activity.Zip) %</p>
    <p>%= Html.LabelFor(Function(activity) activity.PhoneNumber) %<%= Html.DisplayFor(Function(activity) activity.PhoneNumber) %</p>
  </div>
</div>
```

Notice that you have replaced the rendering of the rating from:

ASP.NET (C#)

```csharp
@Html.LabelFor(activity=>activity.Rating) %
@if(rating > 0) {
  <span class="rating rating_@rating.ToString("0.0", System.Globalization.CultureInfo.InvariantCulture).Replace(".","_")">@rating.ToString("0.0")</span>
} else {
  Awaiting at least one more vote.
}
```

to:

ASP.NET (Visual Basic)

```vbnet
<p>%= Html.LabelFor(Function(activity) activity.Rating) %</p>
@if (rating > 0) Then
  <span class="rating rating_@rating.ToString("0.0", System.Globalization.CultureInfo.InvariantCulture).Replace(".","_")">@rating.ToString("0.0")</span>
Else
  Awaiting at least one more vote.
End If
</p>
```
ASP.NET (C#)

```csharp
<p>%
    <%= Html.DisplayFor(activity => activity.Rating, "Rating") %>
</p>
```

ASP.NET (Visual Basic)

```vbnet
<p>%
    <%= Html.DisplayFor(Function(activity) activity.Rating, "Rating") %>
</p>
```

**Note:** The `DisplayFor` Helper Method, with the lambda expression `activity => activity.Rating`, indicates to the framework that the Rating property of the current Model (Activity) must be displayed. The second parameter, “Rating”, indicates to the framework which template to use. The framework will search for the Rating display template inside the DisplayTemplate folder and use the Rating template you have just created to display that property.

Finally, you need to remove the code used for rendering the Activity from the `Details.aspx` View, and replace it with the `DisplayForModel` Helper method that will indicate the framework that it should render the Model (Activity) using the corresponding template (Activity.ascx).

10. In the Solution Explorer, double-click on the `Details.aspx` file, located under `Views\Activities`, to open it.

    **Note:** `DisplayForModel` is equivalent to `DisplayFor(model => model)` which displays the HTML markup required for rendering each property in the model.

11. Replace the `MainContent <asp:Content>` element with the following:

    ```csharp
    (Code Snippet – Building MVC2 App – Details.aspx MainContent CSharp)
    
    ASP.NET (C#)
    
    <asp:Content ID="Content1" ContentPlaceHolderID="MainContent" runat="server">
        <div class="panel">
            <%= Html.DisplayForModel() %>
            <div class="toolbox">
                <% using (Html.BeginForm("Rate", "Activities", new {
                    Id=Model.Id,
                    FormMethod.Post,null)) {
                        %>
                | Rating:
                <%=Html.DropDownList("Rating", new[] { new SelectListItem
                    { Text = "1" }, new SelectListItem
                    { Text = "2" }, new SelectListItem
                    { Text = "3" }, new SelectListItem
                    { Text = "4" }, new SelectListItem
                    { Text = "5" } })%>
                
                <input type="submit" value="Rate »" />
                <% } %>
            </div>
        </div>
    </asp:Content>
    ```
(Code Snippet – Building MVC2 App – Details.aspx MainContent VB)

ASP.NET (Visual Basic)

```vbnet
<asp:Content ContentPlaceHolderID="MainContent" runat="server">
    <div class="panel">
        <%= Html.DisplayForModel()%>
        <div class="toolbox">
            <% Using (Html.BeginForm("Rate", "Activities", New With {.Id = Model.Id}, FormMethod.Post, Nothing)) %>
                | Rating:
                <%= Html DropDownList("Rating", New SelectListItem With {.Text = "1"}, New SelectListItem With {.Text = "2"}, New SelectListItem With {.Text = "3"}, New SelectListItem With {.Text = "4"}, New SelectListItem With {.Text = "5"})%>
            <input type="submit" value="Rate »" />
            <% End Using%>
        </div>
    </div>
</asp:Content>
```

Up to this point, you have completed creating a custom template for the Activity class, including a second template for the Rating property.

Notice that no change was done to the Activity class; this is possible because you are using the same name of the Property or Class when creating the Templates. By default, MVC searches for a template matching the property name when the DisplayFor Helper Method, or the model type when the DisplayForModel Helper Method are used.

---

**Task 2 – Creating a Custom Editor Template for Rendering the ActivityCreateModel**

In this task, you will create a custom editor template, which will contain the information required to render the form for creating an activity.

1. In the Solution Explorer, right-click on the Activities folder, located under Views, point to Add and choose New Folder.

2. Type EditorTemplates as the folder name.

**Note:** As with the Display Template, templates are first searched in this folder.
You will create a **String** template, which will override the pre-defined template, to include the validation and label information altogether with the edit text box, when rendering a string.

**Note:** This template will only override the pre-defined string template, only when rendering the edit form of a string.

3. In the Solution Explorer, Right-click on the **EditorTemplates** folder you have just created, point to **Add** and select **View**...

4. Make sure **Create a partial view (.ascx)** is checked, and type **String** as **View name**
5. Copy the following code inside the String.ascx template:

(Code Snippet – Building MVC2 App – String Template C#)

C#

```csharp
<p>
    @Html.LabelFor(m => m)
    @Html.TextBoxFor(m=>m)
    @Html.ValidationMessageFor(m => m)
</p>
```

(Code Snippet – Building MVC2 App – String Template VB)

Visual Basic

```vbnet
<p>
    @Html.LabelFor(Function(m) m)
    @Html.TextBoxFor(Function(m) m)
    @Html.ValidationMessageFor(Function(m) m)
</p>
```

**Note:** The code above uses three different Html Helpers.

The first, **LabelFor**, is used to display a label of the given model; in this example, the helper renders the value of the **DisplayName** attribute set on the **ActivityCreateModel** class. By default, the helper reflects the model and prints the name of the property.

Next, **TextBoxFor**, is used to display a textbox for the corresponding property. This field will allow the user to input the value for the property.

Finally, **ValidationMessageFor**, will be used to render a validation error message if the user enters an invalid value. This error message is taken from the ModelState. Further details on how validation works in MVC2 can be found in the next exercise.

6. At this point, you can go back to Create.aspx and replace the code required for rendering all the string properties to use the EditorFor Template Helper Method. Instead, you will create a Custom template for rendering the Edit form, and use this helper there.

**Note:** The EditorFor Helper Method works as the DisplayFor Helper. It searches for the template that should be used to render the field, and uses it. If it finds none, it will use the
pre-defined templates instead. In the example, using the EditorFor Helper Method on the String properties will use the template you have just created.

7. In the Solution Explorer, right-click on the EditorTemplates folder you created before, point to Add and select View...

8. Make sure Create a partial view (.ascx) is checked, and type CreateActivity as View name

9. Check the Create a strongly-typed view checkbox, and type PlanMyNight.ViewModels.ActivityCreateModel as View data class

![Add View CreateActivity.ascx dialog](image)

Figure 21
Add View CreateActivity.ascx dialog

The CreateActivity editor template will contain the information on how to render the form for creating an activity.

10. Copy the following code inside the CreateActivity.ascx Editor Template:

(Code Snippet – Building MVC2 App –CreateActivity Template CSharp)

ASP.NET (C#)
<div class="fieldsA">
    <%= Html.EditorFor(m => m.Name) %>
    <%= Html.EditorFor(m => m.PhoneNumber) %>
    <p>
        <%= Html.LabelFor(m => m.State) %>
        <%= Html.DropDownListFor(m => m.State, Model.States) %>
        <%= Html.ValidationMessageFor(m => m.State) %>
    </p>
    <%= Html.EditorFor(m => m.Zip) %>
</div>

<div class="fieldsB">
    <%= Html.EditorFor(m => m.Street) %>
    <%= Html.EditorFor(m => m.City) %>
    <p>
        <%= Html.LabelFor(m => m.ActivityTypeId) %>
        <%= Html.DropDownListFor(m => m.ActivityTypeId, Model.ActivityTypes) %>
        <%= Html.ValidationMessageFor(m => m.ActivityTypeId) %>
    </p>
</div>

(Code Snippet – Building MVC2 App – CreateActivity Template VB)

**ASP.NET (Visual Basic)**

<div class="fieldsA">
    <%= Html.EditorFor(Function(m) m.Name)%>
    <%= Html.EditorFor(Function(m) m.PhoneNumber)%>
    <p>
        <%= Html.LabelFor(Function(m) m.State)%>
        <%= Html.DropDownListFor(Function(m) m.State, Model.States)%>
        <%= Html.ValidationMessageFor(Function(m) m.State)%>
    </p>
    <%= Html.EditorFor(Function(m) m.Zip)%>
</div>

<div class="fieldsB">
    <%= Html.EditorFor(Function(m) m.Street)%>
    <%= Html.EditorFor(Function(m) m.City)%>
    <p>
        <%= Html.LabelFor(Function(m) m.ActivityTypeId)%>
        <%= Html.DropDownListFor(Function(m) m.ActivityTypeId, Model.ActivityTypes)%>
        <%= Html.ValidationMessageFor(Function(m) m.ActivityTypeId)%>
    </p>
</div>
Note: Notice that you have added the **EditorFor** Helper. As said before, this Helper will find the String EditorTemplate you previously created and render the three fields: **label**, **textbox** and **validationmessage**.

At this point, you have completed creating the EditorTemplates; in order for your web application to use them, you will replace the **Create.aspx** code used to render all fields, with the **EditorForModel** Helper. In this case, you will use the overload that receives the Template name as parameter.

**Note:** By using the overload of the EditorForModel Helper that receives the TemplateName as parameter, you will be able to force the framework to use a certain Template.

11. In the Solution Explorer, double-click on **Create.aspx**, located under **Views\Activities**, to open it.

12. Replace the `<fieldset>` definition with the following code:

   (Code Snippet – Building MVC2 App –Create.aspx Fieldset CSharp)

   ```aspnet
   <fieldset>
   <%= Html.EditorForModel("CreateActivity") %>
   <p>
   <input type="submit" value="Save" />
   </p>
</fieldset>
```

Once more, you have created the editor templates required for creating a new Activity, without modifying the code.

**Task 3 – Using the UIHint Attribute for Rendering the Phone Number**

In this task, you will learn how to use the UIHint Attribute to let the MVC framework know which template to use when rendering a property.

1. In the Solution Explorer, right-click on the **DisplayTemplates** folder, located under **Views\Activities**, point to Add and select View...

2. Make sure **Create a partial view (.ascx)** is checked, and type **PhoneNumber** as **View name**. Since PhoneNumber render strings, leave Create strongly-typed view unchecked.
The PhoneNumber display template will contain the information on how to format a string, to render it as if it were a phone number.

3. Add the following code, to render the string with the pattern (###) ###-####, only when the digit count is ten.

   Note: You could provide different formatting option, this is just an example to show the possibilities that MVC 2 Templates provide.

(Code Snippet – Building MVC2 App –PhoneNumber Template CSharp)

```csharp
<%
    long numericPhone;
    var phone = (Model as string) ?? string.Empty;
    long.TryParse(phone, out numericPhone);
%>
```
4. Next, you will decorate the Activity’s PhoneNumber property with the UIHint attribute in order to let the framework know that it must use the PhoneNumber template when rendering this property. To do this, double click on the ActivityMetadata class, located under Models\Entities, to open it.

5. Replace the attributes decorating the PhoneNumber property with the following:

```csharp
[DisplayName("Phone:")]
[UIHint("PhoneNumber")]
public object PhoneNumber { get; set; }
```

```visual-basic
<DisplayName("Phone:")>
<UIHint("PhoneNumber")>
Public Property PhoneNumber As Object
```

**Exercise 2: Verification**

You have completed creating the templates, and configuring your application to use them. In this verification, you will go through the solution to check that it’s still rendering the Display and Create pages correctly. And to see how PhoneNumber is now rendered when displaying an activity.
1. Press **CTRL+F5** to run the solution without debugging.

2. You will open the detailed information for a specific Activity to review the results of the work done. To do this, edit your browser’s address bar to navigate to 

3. You should notice that the usage of templates is transparent for the user, unless you intentionally change something, as done with the PhoneNumber. Notice that it is rendered following the pattern (###) ###-####

![Figure 23](image)

*PlanMyNight Activity details using Display Templates*

4. Next, you will open the create activity view to verify the correct rendering of the page. Browse to: [http://localhost:50000/Activities/Create](http://localhost:50000/Activities/Create).
PlanMyNight Create Activity using Display Templates

Note: Since the template you have created has no UI changes, you will see no difference between rendering the page with Templates than without them. Nevertheless, by adding the EditorTemplates you have cleared your View’s code.

Exercise 3: Client and Server Side Validation in ASP.NET MVC 2

In this Exercise, you will learn how to add validation attributes to your model, and have them reflected in the UI, adding both client, and server side validation.

Out of the box, MVC provides the following set of validation rules which can be added to your model as attributes:

- **Required:** defines if the property is required or not.
- **RegularExpression**: matches the value of the property to the regular expression. Returns the value of the match.
- **Range**: defines the range of valid values for the given property; (minValu, maxValue).
- **StringLength**: defines the maximum length that the string property can have.

Adding simple Server-Side validation is extremely easy with the validation attributes support added in MVC 2, simply verifying if the ModelState is valid in your controller, and rendering the view if it is not should be enough.

```csharp
if (!ModelState.IsValid)
{
    return this.View();
}
```

```visualbasic
If Not ModelState.IsValid Then
    Return Me.View()
End If
```

Simple Client Side validation is also quite simple to add and uses the validation attributes added in the model, avoiding the needs of replicating the rules throught the application. Simply adding the following markup before the form definition should be enough:

**ASP.NET (C#)**
```csharp
<% Html.EnableClientValidation(); %>
```

**ASP.NET (Visual Basic)**
```csharp
<% Html.EnableClientValidation() %>
```

**Note**: The framework also allows you to extend the validation functionality to provide custom rules which apply to your business model. To do this, you must inherit the ValidationAttribute class, overriding the IsValid method.

**Task 0 – Exploring the Solution**
In this task, you will explore the solution you obtained after completing exercise 2 to understand how client and server side validation is performed in a non-MVC2 application. This will help understanding the benefits provided by the MVC framework.

1. Open **Microsoft Visual Studio 2010**. Click **Start | All Programs | Microsoft Visual Studio 2010 | Microsoft Visual Studio 2010**.

2. Open the solution file **Begin.sln** located under \Ex3-Validation\begin\ (Choosing the folder that matches the language of your preference). Alternatively, you can continue working with the solution obtained after completing exercise 2.

3. The Activity Create view performs validation of the data entered on several fields. These validations take place both on the client side (i.e. the browser) and on the server side (i.e. the view’s controller).

4. Client side validation can be implemented in several ways; there also exists several frameworks to achieve the goal of validating data. One of those frameworks, very popular amongst MVC 1 users, is the **jQuery Validation** plugin. This plugin is included out-of-the-box in any version of the MVC project template for Visual Studio. To review these script files, expand the Scripts folder in the PlanMyNight project.

![Solution Explorer](image-url)
5. Validation is executed on the client by the means of JavaScript code attached to the view. This code uses the jQuery Validation plugin to setup the rules to be applied on the fields included in the HTML form. In this case you can examine the **required** rules configured for the **Name**, **PhoneNumber** and **Street** input fields. Expand the Views\Activities folder in the PlanMyNight project and open the Create.aspx file to review the validation implementation.
6. Also, it is important to notice that because the validation occurs on the client, knowing the HTML id attribute is very important in order to properly setup the validation rules. Scroll up in the view markup and notice the call to the `Html.BeginForm` method for an example.

```csharp
using (Html.BeginForm("Create", "Activities", FormMethod.Post, new { @id = "validationForm" })
```

**Figure 28**

`BeginForm method call (C#)`

```vbnet
Using (Html.BeginForm("Create", "Activities", FormMethod.Post, New With {.id = "validationForm"})
```

**Figure 29**

`BeginForm method call (Visual Basic)`

7. On the other hand, server side validation is performed on the view’s controller. The `ModelState` object is used to send information about issues back to the view. When a model binder encounter input errors it automatically populates the ModelState collection, but for business rules validation an imperative approach is required. To review the view’s controller, expand the Controllers folder in the PlanMyNight and open the `ActivitiesController.cs (C#)` or `ActivitiesController.vb (Visual Basic)` class file.

8. **DataAnnotations** provides a way to encapsulate validation rules in the model itself by providing a set of attributes that can be used as metadata to determine if the specified model is valid. In our example we use the `TryValidateObject` method of the `Validator` class to
imperatively validate the model object. By using this approach we get a collection of errors that can be used to manually populate the ModelState collection to render the error messages on the view. To review this approach, scroll down to the Create method and examine the code below the commented line.

```csharp
// Model Validation
if (ModelState.IsValid)
{
    var validationContext = new ValidationContext(model, null, null);
    var validationResults = new Collection<ValidationResult>();
    Validator.TryValidateObject(model, validationContext, validationResults);
    foreach (var validationResult in validationResults)
    {
        ModelState.AddModelError(validationResult.MemberNames.First(), validationResult.ErrorMessage);
    }
}
```

**Figure 30**

*Server side imperative model validation (C#)*

```vbnet
' Model Validation
If ModelState.IsValid Then
    Dim validationContext = New ValidationContext(model, Nothing, Nothing)
    Dim validationResults = New Collection(Of ValidationResult)()
    Validator.TryValidateObject(model, validationContext, validationResults)
    For Each validationResult In validationResults
        ModelState.AddModelError(validationResult.MemberNames.First(), validationResult.ErrorMessage)
    Next validationResult
End If
```

**Figure 31**

*Server side imperative model validation (Visual Basic)*

9. This sample uses the RequiredAttribute to decorate properties that will be mapped to fields to get validated in the controller’s action. As you can notice, in the absence of a RequiredAttribute decorating the PhoneNumber property, the ActivityCreateModel validation attributes does not match the client side validation rules implemented using jQuery Validation plugin. This is a major disadvantage of this approach because you must maintain validation rules in sync manually. To verify this, expand the ViewModels folder in the PlanMyNight project and open the ActivityCreateModel.cs (C#) or ActivityCreateModel.vb (Visual Basic) class file.
public class ActivityCreateModel
{
    [Display(Name="Phone:")]
    public string PhoneNumber { get; set; }

    [Display(Name="Name:")]
    [Required(ErrorMessage = "Please enter the Activity Name.")]
    public string Name { get; set; }

    [Display(Name="State:")]
    [Required(ErrorMessage = "Please enter the Activity State.")]
    public string State { get; set; }

    [Display(Name="Zip:")]
    [Required(ErrorMessage = "Please enter the Activity Zip code.")]
    public string Zip { get; set; }

    [Display(Name="Address:")]
    [Required(ErrorMessage = "Please enter the Activity Address.")]
    public string Street { get; set; }

    [Display(Name="City:")]
    [Required(ErrorMessage = "Please enter the Activity City.")]
    public string City { get; set; }

    [Display(Name="Activity:")]
    public int ActivityTypeId { get; set; }

    public IEnumerable<SelectListItem> ActivityTypes { get; set; }

    public IEnumerable<SelectListItem> States { get; set; }
}

Figure 32
ActivityCreateModel model validation attributes(C#)
Public Class ActivityCreateModel
    <DisplayName>("Phone:")>
    Public Property PhoneNumber() As String

    <DisplayName>("Name:")>
    <Required(ErrorMessage:="Please enter the Activity Name.")>
    Public Property Name() As String

    <DisplayName>("State")>
    <Required(ErrorMessage:="Please enter the Activity State.")>
    Public Property State() As String

    <DisplayName>("Zip:")>
    <Required(ErrorMessage:="Please enter the Activity Zip code.")>
    Public Property Zip() As String

    <DisplayName>("Address:")>
    <Required(ErrorMessage:="Please enter the Activity Address.")>
    Public Property Street() As String

    <DisplayName>("City:")>
    <Required(ErrorMessage:="Please enter the Activity City.")>
    Public Property City() As String

    <DisplayName>("Activity:")>
    Public Property ActivityTypeId() As Integer

    Public Property ActivityTypes() As IEnumerable(Of SelectListItem)

    Public Property States() As IEnumerable(Of SelectListItem)
End Class

---

Figure 33
ActivityCreateModel model validation attributes (Visual Basic)

Task 1 – Removing jQuery Client Side Validation

1. In the ActivityCreateModel class you verified the lack of synchronization between client side and server side validation. You will make this difference more evident by removing the jQuery validation rules in the Create view. To do this, expand the Views\Activities folder and open the Create.aspx markup by double-clicking on the file.

2. Luckily, the jQuery validation implementation is a little portion of code isolated at the bottom of the view markup. Since you want replace this portion of code with the MVC client validation, you can get rid of it. To do this, scroll down to the <script> section containing the jQuery validation and remove it.
3. You will browse the Create page to verify that client side validation is not being performed. To do this, press **CTRL + F5** to run the solution without debugging and when the browser is launched navigate to: [http://localhost:50000/Activities/Create](http://localhost:50000/Activities/Create). When the page is fully loaded press **CTRL + F5** to reload the page from the server in case it is stored in the browser’s cache.

4. In the Create page fill the Name, Zip, Address and City fields with random data and press the Create button. No client validation will be performed and you will be redirected to the Details page when the new item is created in the database.

Note: Remember that because validation is not fully enabled, you should enter valid data in order to avoid errors. For example, the Zip database field permits only five characters in its content.

```javascript
$(document).ready(function () {
    $('form').validate({
        errorContainer: '#clientValidationMessage',
        errorLabelContainer: $('ul', $('#clientValidationMessage')),
        wrapper: 'li',
        rules: {
            Name: { required: true },
            PhoneNumber: { required: true },
            Street: { required: true }
        },
        messages: {
            Name: 'Please enter the Activity Name.',
            PhoneNumber: 'Please enter the Activity Phone Number.',
            Street: 'Please enter the Activity Address.'
        }
    });
});
</script>

Figure 34
Portion of Create.aspx file that should be removed.
5. Now, clear all the fields in the page and press the Create. The server will perform validation and require the Name, Zip, Address and City fields to be entered.
6. Close the browser.

Task 2 – Removing Imperative Server Side Model Validation

1. In a previous task you verified how the Activities controller performed imperative validation using the DataAnnotations Validation class and populating the ModelState object to send back errors to the view. This task is now performed automatically in ASP.NET MVC 2.0; consequently the code used to perform validation is no longer required. To do this, expand the Controllers folder and open the ActivitiesController.cs (C#) or ActivitiesController.vb (Visual Basic) class file by double-clicking on it. Scroll down to the Create method and remove the code below the commented line as in the following code.

```csharp
[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Create(ActivityCreateModel model) {
    // Removed Validation code
    if (!ModelState.IsValid)
    {
        return this.Create();
    }
}
```
...  
}

**Visual Basic**

```visualbasic
<AcceptVerbs(HttpVerbs.Post)>
Public Function Create(ByVal model As ActivityCreateModel) As ActionResult
    ' Removed Validation code
    If (Not ModelState.IsValid) Then
        Return Me.Create()
    End If
...  
End Function
```

2. As you can notice, the code to test the ModelState.IsValid property is still required since it contains the result of all validations.

3. You will browse the Create page to verify that server side validation is being performed automatically. To do this, press **CTRL + F5** to run the solution without debugging and when the browser is launched navigate to: [http://localhost:50000/Activities/Create](http://localhost:50000/Activities/Create).

4. Now, clear all the fields in the page and press the Create. The server will automatically perform validation and require the **Name**, **Address**, **City** and **Zip** fields to be entered.

5. Close the browser.

### Task 3 – Adding Model Validation Metadata

1. MVC 2.0 validation engine uses validation metadata in the form of DataAnnotations attributes to describe the validation rules an object must conform. To add more strict validation rules you want to decorate the Phone property with the RequiredAttribute. To do this, open the `ActivityCreateModel.cs (C#)` or `ActivityCreateModel.vb (Visual Basic)` file located in the `ViewModels` folder and then replace the Phone property declaration with the bolded code below.

   **C#**
   ```csharp
   [DisplayName("Phone:")]  
   [Required(ErrorMessage = "Please enter the Activity Phone Number."ırım)]
   public string Phone { get; set; }
   ```

   **Visual Basic**
   ```vbnet
   <DisplayName("Phone:")>
   <Required(ErrorMessage:="Please enter the Activity Phone Number.")>
   Public Property Phone() As String
   ```
2. You will decorate the view model with more complex rules as well. In addition to the RequiredAttribute, decorate the Zip property with the StringLengthAttribute. To do this, replace the Zip property declaration with the bolded code below.

```csharp
[DisplayName("Zip: ")]
[Required(ErrorMessage = "Please enter the Activity Zip.")]  
[StringLength(5, ErrorMessage = "Enter a valid Zip code for the Activity")]
public string Zip { get; set; }
```

```vbnet
<DisplayName("Zip: ")>
<Required(ErrorMessage="Please enter the Activity Zip." )>
<StringLength(5, ErrorMessage="Enter a valid Zip code for the Activity")>
Public Property Zip() As String
```

**Task 4 – Enabling Client Side Validation**

1. ASP.NET MVC 2.0 includes support for client side validation based on jQuery Validation plugin out-of-the-box. Enabling client side validation simply involves calling Html.EnableClientValidation() before you call Html.BeginForm(). Under the hood, this sets a flag in the ViewContext which lets the BeginForm method know that client validation is enabled. To do this, expand the Views\Activities folder in the PlanMyNight project and open the Create.aspx file by double-clicking on it. Then, scroll down to the Html.BeginForm() method call and paste the following bolded code in the line above.

```html
<% using (Html.BeginForm()) %>
<% Html.EnableClientValidation(); %>
<% using (Html.BeginForm()) %>
```

2. Also, you confirmed that jQuery validation implementation require to specify the id attribute of the resulting HTML form. If you set an id for the form, the MVC runtime will know which ID to use when hooking up client validation. If you don’t, the form will render one for you. Update the call to the Html.BeginForm() method to not pass parameters. To do this, replace the Html.BeginForm() method call using four parameters by the following bolded code.
3. Since out-of-the-box validation is based on JQuery you want to confirm you include the required script files. Also, the MVC implementation of client validation includes the **MicrosoftMvcJQueryValidation.js** script file which adapts the jQuery Validation plugin to an MVC application. To verify this, expand the **Views\Shared** folder in the PlanMyNight project and open the **Site.Master** file. You will notice the references to the three required script files.

```html
<head runat="server">
<title>asp:ContentPlaceHolder ID="TitleContent" runat="server" /></title>
<meta http-equiv="content-type" content="text/html; charset=UTF-8" />
<meta http-equiv="X-UA-Compatible" content="IE=8" />
<link href="../../Content/Site.css" rel="stylesheet" type="text/css" />
<asp:ContentPlaceHolder ID="HtmlHeadContent" runat="server" />
<script type="text/javascript" src="/Scripts/jquery-1.3.2.js"></script>
<script type="text/javascript" src="/Scripts/jquery.validate.js"></script>
<script type="text/javascript" src="/Scripts/MicrosoftMvcJQueryValidation.js"></script>
</head>

**Figure 37**

*Master page referencing the script files required for client validation.*

---

**Exercise 3: Verification**

1. Press **CTRL+F5** to run the solution without debugging.

2. You will open the Create Activity view to review the results of the work done. To do this, edit your browser’s address bar to navigate to [http://localhost:50000/Activities/Create](http://localhost:50000/Activities/Create).

3. First, you will verify that added model validation metadata is replicated on the client side validation rules. To do this, clear all the fields in the view and click on the Create button. Validation will occur on the client side without a roundtrip to the server.
4. Now, you will verify specifically one of the new included model validation metadata attributes. In the second task you decorated the Zip property with a StringLength attribute to ensure the zip value has a maximum length of 5 characters.

5. You will verify the string length rule by entering a value on each of the required fields: Name, Address and Phone. Additionally, enter a long string value in the Zip field with more than 5 characters, for example enter “123456” and then click on the Create button. A client validation message should appear.

Figure 38

*Client side validation in sync with model validation metadata.*
Exercise 4: ASP.NET MVC Areas

In this exercise, you will learn how to use ASP.NET MVC Areas, taking advantage of the new tooling in ASP.NET MVC 2 for VS2010, which includes templates and context menu items (for example, Add Area item template). Moreover you will see how to register the routes of the area you have created.

MVC Areas were designed to provide a way for partitioning your application into areas of functionality. This helps building and managing large applications.

The exercise starts with an all-in-one solution, which simulates a large web application, and guides you through the process of extracting the logon functionality to a new area.

Task 1 – Creating the New Account Area

2. Open the solution file \Begin.sln located under \Ex4-MVCAreas\begin (Choosing the folder that matches the language of your preference). Alternatively, you can continue working with the solution obtained after completing exercise 3.

3. You will take advantage of the item template provided by MVC 2 to create an area which will contain assets required for managing an account; to do this, right-click the PlanMyNight project and select Add | Area…

4. Set the Area Name to Account in the Add Area dialog, and click OK

A new Areas folder will be created, containing the Account subfolder inside.

![Solution Explorer](Figure 40)

*Solution structure after create the Account Area (C#)*
A folder for each area is created inside the Areas folder of the MVC project when you use the New Area item template. Each area folder has a similar structure than your main project, Controllers, Models and Views folder can be found within them.

The **AccountAreaRegistration.cs (C#)** or **AccountAreaRegistration.vb** (Visual Basic) file is also created inside the new area folder. This file contains the area-specific route definition.

---

**Task 2 – Moving the Account Controller and Views from the Main Project to the new Area**

In this task, you will move the assets needed for managing account to the new **Account** area.

1. Drag the **AccountController.cs (C#)** or **AccountController.vb** (Visual Basic) file, located inside the root’s **Controllers** folder, and drop it inside **Areas/Account/Controllers**.
2. Proceed as before, but now drag the **Account** folder instead, located inside the root’s **Views** folder, and drop it inside **Areas/Account/Views**.

3. After completing the previous steps, your solution structure should look like the following:

![Solution Explorer](image)

*Figure 42*

*Solution structure after move security assets to the new Account area (C#)*
Task 3 – Registering Account Routes

Up to this point, you have moved the required assets to the corresponding area folder. In order for the main project to be aware of the Account Area, you need to register the corresponding routes in the root’s RouteCollection.

To do this, you will invoke the AreaRegistration.RegisterAllAreas() method from within the project’s Global.asax file.

**Note:** The new MVC 2 Project Templates already include the invocation to this method within the global.asax file.
1. In the Solution Explorer double-click on the **Global.asax** file, located in PlanMyNight’s root directory, to open it.

2. Add an invocation to **RegisterAllAreas** inside **RegisterRoutes** as it is shown below:

   **C#**
   ```csharp
   public static void RegisterRoutes(RouteCollection routes)
   {
       routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

       AreaRegistration.RegisterAllAreas();

       routes.MapRoute(
           "Default", // Route name
           "{controller}/{action}/{id}", // URL with parameters
           new { controller = "Home", action = "Index", id = "" } // Parameter defaults
           );
   }
   ```

   **Visual Basic**
   ```vbnet
   Shared Sub RegisterRoutes(ByVal routes As RouteCollection)
       routes.IgnoreRoute("{resource}.axd/{*pathInfo}")

       AreaRegistration.RegisterAllAreas()

       ' MapRoute takes the following parameters, in order:
       ' (1) Route name
       ' (2) URL with parameters
       ' (3) Parameter defaults
       routes.MapRoute(_.
           "Default", _.
           "{controller}/{action}/{id}", _
           New With {.controller = "Home", .action = "Index", .id = ""} _
           )
   End Sub
   ```

   **Note:** **RegisterAllAreas** method searches for the different areas on your project, and executes the **RegisterArea** method of each of them (you can find this method inside the **AreaRegistration** class created automatically by the Areas Template.)

   Within the **AreaRegistration** class, you should add the code to register all the routes required by your application.
Task 4 – Modifying the Code to Navigate Between the Main and Account Area

Finally, you will update the navigation links to target the correct area.

1. In the solution explorer, Double-click on the **Site.Master** file, located inside **Views\Shared**, to open it.

2. Update the Html.ActionLink declaration inside the `<div>` with “navigation” id, to define if the action is located in a specific area (“account”) or on the root project (“”).

   **ASP.NET (C#)**

   ```csharp
   <div id="navigation">
       <ul>
           <li><%= Html.ActionLink("Search", "Index", "Home", new { area = "" }, null) %></li>
           <li><%= Html.ActionLink("Create", "Create", "Activities", new { area = "" }, null) %></li>
           <li><%= Html.ActionLink("About", "#", null) %></li>
           <% if (Request.IsAuthenticated) { %>
           <% } else { %>
           <% } %>
       </ul>
   </div>
   
   **ASP.NET (Visual Basic)**

   ```vbnet
   <div id="navigation">
       <ul>
           <li><%= Html.ActionLink("Search", "Index", "Home", New With {.area = ""}, Nothing) %></li>
           <li><%= Html.ActionLink("Create", "Create", "Activities", New With {.area = ""}, Nothing) %></li>
           <li><%= Html.ActionLink("About", "#", Nothing) %></li>
           <% If (Request.IsAuthenticated) Then %>
               <li><%= Html.ActionLink("Log Off", "LogOff", "Account", New With {.area = "account"}, Nothing) %></li>
           <% Else %>
           <% End If %>
       </ul>
   </div>
   ```
Else
  \%
  \<li\>\%= Html.ActionLink("Log On", "LogOn", "Account", New With
  \{\.area = "account"\}, Nothing)\%/</li>\%
  End If
\%</ul>
</div>

**Note:** The fourth parameter defines the `routeValues` for the action link; while the fifth is the HtmlAttributes parameter and is not required.

**Note:** Since the Master Page will be used throughout the application, you need to explicitly tell in which area the target page should be found.

That is why you add the `routeValue area = ""` on those links that point to the pages located in the main project.

3. Additionally, you need to change the `RedirectToAction` declaration inside the `AccountController` to point to the main area, since they are invoking the actions on the `HomeController`. To do this, open the `AccountController.cs` (C#) or `AccountController.vb` (VB) file, located inside the `Areas\Account\Controllers` folder.

4. Update the 3 `RedirectToAction` invocations, to include the `routeValue` specifying the main area.

   The updated code should look as follows:

   **C#**
   ```csharp
   return RedirectToAction("Index", "Home", new { area = "" });
   ```

   **Visual Basic**
   ```vbnet
   Return RedirectToAction("Index", "Home", New With {.area = ""})
   ```

   **Note:** `RedirectToAction` invocations can be found on the following 3 methods of the `AccountController`:
   - `LogOn(string, string, bool, string)`
   - `LogOff()`
   - `Register(string, string, string, string)`
5. If you run the solution now, the Account controller inside the new area will not be found because MVC will be search for it inside the `PlanMyNight.Areas.Account.Controllers` namespace. Since you have moved the controller from the root project, Update the Account Controller’s namespace to match the following:

**C#**

```csharp
{
    [HandleError]
    public class AccountController : Controller
    {
        ...
    }
}
```

**Visual Basic**

```vbnet
    <HandleError()> _
    Public Class AccountController
        ...
    End Class
End Namespace
```

---

**Exercise 4: Verification**

1. Press `CTRL+F5` to run the solution without debugging.

2. Hover the mouse pointer over the links in the navigation bar. Notice that the first two links point to actions inside controllers of the MVC project, while LOG ON point to the LogOn action available on the Account controller of the Account area.
Figure 44
PlanMyNight’s Search page pointing to the LOG ON link

| Note: About is a placeholder only; that is why it links to self (#.) |

3. Click the **LOG ON** link at the top of the page.
4. Click the **Register** link to create a new user.
5. Fill the Account Information form with some valid data and click **Register**.
Figure 45
Registering a user in PlanMyNight

6. You should be redirected to the PlanMyNight’s Search page where the LOG ON link changes to LOG OFF. It points to the LogOff action available in the Account controller of the Account Area.

Figure 46
PlanMyNight’s Search page pointing to the LOG OFF link

Summary

By completing this Hands-On Lab you have learnt how to make use of the new features provided as part of the second version of the MVC Framework, such as the UI and Validation Metadata, Template Helpers, and MVC Areas.