Agenda – Digging into HTMLHelper Methods

- HTMLHelper Methods
- Render HTML Form
- Using DropDownList
- Binding HtmlHelper to Model
- Binding HtmlHelper to ViewData dictionary
- Using “For” Methods with Typed Model
- Overriding Display Templates
- Overriding Editor Templates
- Reusing using custom @helper Methods
- Writing Custom Helper Extension methods

Understanding Html Helpers

An HTML Helper is just a method that returns a string. The string can represent any type of content that you want. HTML Helpers can be used to render standard HTML tags like HTML <input> and <img> tags or to render more complex content such as a <table> of database data.

The ASP.NET MVC framework includes the following set of standard HTML Helpers:

- Html.ActionLink()
- Html.BeginForm()
- Html.CheckBox()
- Html.DropDownList()
- Html.ListBox()
- Html.Password()
- Html.RadioButton()
- Html.TextArea()
- Html.TextBox()
- Html.Hidden()

Rendering Links

```
@Html.ActionLink("Home", "Index", "Home", new { Id1 = 1, name = "Sandeep Soni"  }, new { target = "_blank" })
```

Note: The text of link and route values are automatically encoded

To render Image as link:

```
<a href="@Url.Action("Delete", "Employee", new { Id = 1 })"><img src="delete.gif" alt="delete" /></a>
```

Rendering Form and its Elements

Option1:

```
@{ Html.BeginForm("ActionName", "ControllerName"); }
        Name: @Html.TextBox("txtName")
            <input type="submit" value="Submit" />
@{ Html.EndForm(); }
```

Option2:

```
@{
    Html.BeginForm("ActionName", "ControllerName");
        @:Name:
        @Html.TextBox("txtName")
            <input type="submit" value="Submit" />
    Html.EndForm();
}
```

Option3:

```
@using (Html.BeginForm("ActionName", "ControllerName"))
    {
        @:Name:
        @Html.TextBox("txtName")
        <input type="submit" value="Submit" />
    }
```

Note: Html.BeginForm return MvcForm and this implements IDisposable and Dispose method of this is calling Html.EndForm() which renders “</form>”

Working with TextBox and other simple controls:

```
@Html.Label("name","Name:")
@Html.TextBox("name")
```

Working with RadioButton:

```
@Html.RadioButton("color", "red", new { id = "rbnRed", @checked="checked" })@Html.Label("rbnRed", "Red")
@Html.RadioButton("color", "green", new { id = "rbnGreen" })@Html.Label("rbnGreen", "Green")
@Html.RadioButton("color", "blue", new { id = "rbnBlue" })@Html.Label("rbnBlue", "Blue")
```

Working with DropDownList and ListBox

```
@Html.DropDownList("DDL1", new List<SelectListItem> {
                                            new SelectListItem() {Text="Item1",Value="1"},
                                            new SelectListItem() {Text="Item2",Value="2",Selected=true},
                                            new SelectListItem() {Text="Item3",Value="3"},
                                            new SelectListItem() {Text="Item4",Value="4"} });
 @Html.ListBox("LST1", new List<SelectListItem> {
                                            new SelectListItem() {Text="Item1",Value="1"},
                                            new SelectListItem() {Text="Item2",Value="2",Selected=true},
                                            new SelectListItem() {Text="Item3",Value="3"},
                                            new SelectListItem() {Text="Item4",Value="4"} });
 <br />
 @Html.DropDownList("DDL2", new List<SelectListItem> {
                                            new SelectListItem() {Text="Item1",Value="1"},
                                            new SelectListItem() {Text="Item2",Value="2",Selected=true},
                                            new SelectListItem() {Text="Item3",Value="3"},
                                            new SelectListItem() {Text="Item4",Value="4"} },
                                    new { size = "4" }); //For single selection listbox
}
```

Binding Html Helpers to ViewData dictionary

In Controller:

```
public ActionResult Index()
{
  ViewData["Id"] = 1;
  ViewData["Name"] = "E1";
  ViewData["Salary"] = 10000;
  ViewData["IsActive"] = true;
  ViewData["DeptId"] = 2;
  ViewData["DateOfJoin"] = DateTime.Now;
  ViewData["EmailAddress"] = "test@test.com";

  List<Department> depts = new List<Department>();
  depts.Add(new Department() { DeptName = "D1", DeptId = 1 });
  depts.Add(new Department() { DeptName = "D2", DeptId = 2 });
  SelectList sl = new SelectList(depts, "DeptId", "DeptName", 2);
  ViewData["DeptId"] = sl;
  return View();
}
```

In View:

```
@using (Html.BeginForm("Index", "Home"))
{
  @Html.Label("Id", "ID: ")
  @Html.TextBox("Id")
  <br />
  @Html.Label("DeptId", "Department: " );
  @Html.DropDownList("DeptId")
  <br />
  @Html.Label("Name", "Name: ")
  @Html.TextBox("Name")
  <br />
  @Html.Label("Salary", "Salary: ")
  @Html.TextBox("Salary")
  <br />
  @Html.CheckBox("IsActive")
  @Html.Label("IsActive", "Is Active:")
  <br />
  @Html.Label("Email", "Email: ")
  @Html.Display("EmailAddress")
  <br />
  @Html.Label("DateOfJoin", "Date of Join: ")
  @Html.TextBox("DateOfJoin")
  <br />
  <input type="submit" name="btnSumit" value="Submit" />
}
```

Binding Html Helpers to Model

1. **Add the following to the Model folder**

```
public class Employee
{
  public int Id { get; set; }
  public string Name{ get; set; }
  public decimal Salary { get; set; }
  public bool IsActive { get; set; }
  public string EmailAddress { get; set; }
```

```
    public DateTime DateOfJoin { get; set; }
}
```

2. Add the following to the controller

```
public ActionResult Index()
{
    Employee emp = new Employee() { Id = 1, Name = "E1", Salary = 10000, IsActive = true, DateOfJoin =
DateTime.Now,
EmailAddress = "test@test.com" };
    return View(emp);
}

[HttpPost()]
public ActionResult Index(Employee e)
{
    ModelState.Clear(); //So that the updated Model is used by Html Helper Methods.
    e.Name = "New Name";
    return View(e);
}
```

3. Add the following to the View

```
@using (Html.BeginForm("Index","Home"))
{
    @Html.Label("Id","ID: ")
    @Html.TextBox("Id")
    <br />
    @Html.Label("Name","Name: ")
    @Html.TextBox("Name")
    <br />
    @Html.Label("Salary","Salary: ")
    @Html.TextBox("Salary")
    <br />
    @Html.CheckBox("IsActive")
    @Html.Label("IsActive","Is Active:")
    <br />
    @Html.Label("Email","Email: ")
    @Html.Display("EmailAddress")
    <br />
```

```
        @Html.Label("DateOfJoin","Date of Join: ")
        @Html.TextBox("DateOfJoin")
        <br />
        <input type="submit" name="btnSumit" value="Submit" />
    }
```

Note: Html.TextBox first looks in the posted request values and then in the model that you update in your controller. In the posted request values it finds is the old value:

Example Demonstrating "For" Methods with Typed Model:

All these methods also have their corresponding "For" methods. For example we have two methods **Html.CheckBox** and **Html.CheckBoxFor**

Difference between both the methods is, later is used only if view is tightly coupled with model and lambda expression can be used in these methods.

Eg: DropDownListFor used when view is tightly coupled to model otherwise we should use DropDownList method.

4. Add the following on top of View and make it as Typed View

   ```
   @model MvcApplication1.Models.Employee
   ```

5. Add the following to the View

   ```
   @Html.LabelFor(e=>e.Id, "ID: ")
   @Html.TextBoxFor(e => e.Id)
   ```

Binding DropDownList to Model Property of type Enum using Html.EnumDropDownListFor:

```
public enum EmployeeType
{
  Trainee=1,
  Junior=2,
  Senior=3
}
public class Employee
{
  ...
  public EmployeeType Type { get; set; };
}
@Html.EnumDropDownListFor(m=>m.Type,"Select Type")
```

Binding DropDownList to Model Property which is a object of another type.

6. Add the following to Model Folder

```
public class Department
{
    public int DeptId { get; set; }
    public string DeptName { get; set; }
}
```

7. Add the property DeptId to Employee class

```
public class Employee
{
    …
    public int DeptId { get; set; }
}
```

8. Initialize TempDate in Index method as shown below

```
public ActionResult Index()
{
    List<Department> depts = new List<Department>();
    depts.Add(new Department() { DeptName = "D1", DeptId = 1 });
    depts.Add(new Department() { DeptName = "D2", DeptId = 2 });
    SelectList sl = new SelectList(depts, "DeptId", "DeptName");
    TempData["DepartmentList"] = sl;
    TempData.Keep();

    Employee emp = . . .
    return View(e);
}
[HttpPost()]
 public ActionResult Index(Employee e)
 {
    //. . .
    TempData.Keep();
    e.Name = "New Name";
    return View(e);
}
```

9. Add the following to the View

```
@Html.Label("DeptId", "Department: " );
```

```
@Html.DropDownList("DeptId", TempData["DepartmentList"] as SelectList)
OR
@Html.LabelFor(e=>e.DeptId, "Department: " );
@Html.DropDownListFor(e=>e.DeptId, TempData["DeptId"] as SelectList)
. . .
```

Note: Lambda Expression is to mention the name of the HTML field and same is two way binded to property of Model

Display and Editor Method

These methods generates different HTML markup depending on the **data type of the property** that is being rendered, and according to whether the property is marked with certain attributes.

This method's renders markup according to the following rules:

1. If the property is typed as a primitive type (integer, string, and so on), the method renders a string that represents the property value.
2. If the property type is Boolean, the method renders an HTML input element for a check box. For example, a Boolean property named Enabled might render markup such as the following:

   <input class="check-box" **disabled="disabled"** type="checkbox" checked="checked" />
3. If a property is annotated with a data type attribute, the attribute specifies the markup that is generated for the property. For example, if the property is marked with the [DataType(DataType.EmailAddress)] attribute, the method generates markup that contains an HTML anchor that is configured with the mailto protocol, as in the following example:

   <a href='**mailto**:test@test.com'>test@test.com</a>
4. If the object contains multiple properties, for each property the method generates a string that consists of markup for the property name and markup for the property value.

   For Example

```
@Html.DisplayFor(e=>e)  will render:
<div class="display-field">1</div>
<div class="display-label">DeptId</div>
<div class="display-field">2</div>
<div class="display-label">Name</div>
<div class="display-field">E1</div>
<div class="display-label">Salary</div>
<div class="display-field">10000.00</div>
<div class="display-label">IsActive</div>
```

```
<div class="display-field"><input checked="checked" class="check-box" disabled="disabled" type="checkbox"
/></div>
<div class="display-label">EmailAddress</div>
<div class="display-field"><a href='mailto:test@test.com'>test@test.com</a></div>
<div class="display-label">DateOfJoin</div>
<div class="display-field">11/20/2012 6:52:36 PM</div>
```

Custom Templates for Display and Editor Methods

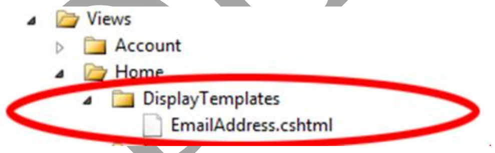
If a template whose name matches the templateName parameter is found in the controller's **DisplayTemplates** folder, that template is used to render the expression. If a template is not found in the controller specific DisplayTemplates folder, the **Views\Shared\DisplayTemplates** is searched for a template that matches the name of the templateName parameter. If no template is found, the default template is used.

Overriding Display Template

1. Create a Model as below:

```
using System.ComponentModel.DataAnnotations;
public class Employee
{
  //. . .
  //[DataType(DataType.EmailAddress)]
  [UIHint("EmailAddress")]
  public string EmailAddress { get; set; }
}
```

Notes: Display templates are .cshtml partial views that have the same name as the **type** they're going to override the default templates provided by MVC. To create a Display Template you just create a Folder named "DisplayTemplates" within one of your controller views (or the shared folder)



2. Create an **EmailAddress.cshtml** template that will format an email address with a mailto: link

```
@inherits System.Web.Mvc.WebViewPage<string>
<a href='mailto:@Model'>@Html.DisplayTextFor(m => m)</a>
```

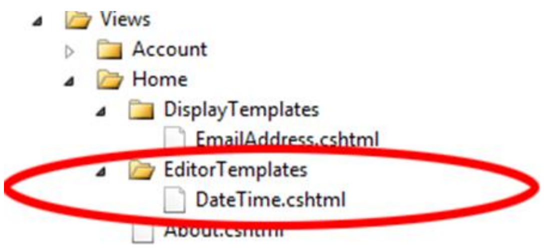
Note: Both methods, **@Model** and an **Html helper** are used to display the actual value of the model.

3. In View use as below:

```
@Html.Display("EmailAddress")
```

Overriding Editor Templates:

To create an Editor Template you just create a folder named "EditorTemplates" within one of your controller views (or the shared folder) like so:



4. In DateTime.cshtml

```
@inherits System.Web.Mvc.WebViewPage<System.DateTime>
@Html.TextBox("", (Model.ToShortDateString()), new { @class = "datePicker" })
```

Note: the @class is the html attribute we're using to assign the datePicker to textbox. Below is the jquery to add the datepicker to our new Textbox with the added class attribute.

5. Add the following to View

```
@section scripts
{
    @Styles.Render("~/Content/themes/base/css")
    @Scripts.Render("~/bundles/jqueryui")
    <script type='text/javascript'>
            $(document).ready(function () {
                    $(".datePicker").datepicker();
            });
    </script>
}
```

Reusing using custom @helper Methods

The @helper syntax within Razor enables you to easily create re-usable helper methods that can encapsulate output functionality within your view templates. They enable better code reuse, and can also facilitate more readable code.

Add following code to View (Specific to the view)

```
@helper RenderTable(int tableOf)
{
  <table>
  @for (int i = 1; i < 10; i++)
   {
      <tr>
        <td>@tableOf</td>
        <td>*</td>
        <td>@i</td>
        <td>=</td>
        <td>@(tableOf*i)</td>
      </tr>
    }
  </table>
}
```

Now when ever required we can use this method as shown below

```
@RenderTable(15)
@RenderTable(25)
@RenderTable(35)
```

To reuse Razor Helper in multiple views:

1. Add **App_Code** folder to the project
2. Add new razor view called MyHtmlHelper.cshtml
3. Add following code to MyHtmlHelper.cshtml

```
@using System.Web.Mvc;
@helper Script(string scriptName, UrlHelper url)
{
  <script src="@url.Content("~/Scripts/"+scriptName)" type="text/javascript" ></script>
}
```

Note: Runtime will compile any razor views which it finds inside this folder and make their helpers available as static methods in a class whose name is same as .cshtml (MyHtmlHelper.cshtml)

4. Now whenever required we can use this method as shown below

```
@MyHtmlHelper.Script ("jquery-1.4.1.min.js", Url)
@MyHtmlHelper.Script ("jquery.unobtrusive-ajax.min.js", Url)
@MyHtmlHelper.Script ("MicrosoftMvcAjax.js", Url)
```

Building Custom Helpers using Extension Methods

HTML Helpers provide a clean way to encapsulate view code so you can keep your views simple and markup focused. There are lots of built in HTML Helpers in the System.Web.Mvc.HtmlHelper class, but one of the best features is that you can easily create your own helpers

```
using System.Web.Mvc;
namespace MvcApplication1
{
  public static class HtmlHelperExtention
  {
    public static string FormatToCurrency(this HtmlHelper helper, decimal amount)
    {
      return string.Format("{0:c}", amount);
    }
    public static MvcHtmlString Image(this HtmlHelper helper, string src, string alt)
    {
      TagBuilder bulder = new TagBuilder("img");
      bulder.MergeAttribute("src", src);
      bulder.MergeAttribute("alt", alt);
      return MvcHtmlString.Create(bulder.ToString(TagRenderMode.SelfClosing));
    }
  }
}
```

Note: we are extending the HtmlHelper defined in System.Web.Mvc and not System.Web.WebPages. The using statement is important.

Our views can make use of this by either importing the namespace with the @using keyword to the views.

```
@using MvcApplication1
@Html.FormatToCurrency(1020003.00M);
```

Alternatively, instead of placing **@using** in all views we can place it in web.config

Adding ref to web.config:

```
 <system.web.webPages.razor>
   <host factoryType="System.Web.Mvc.MvcWebRazorHostFactory, System.Web.Mvc, Version=3.0.0.0,
Culture=neutral, PublicKeyToken=31BF3856AD364E35" />
   <pages pageBaseType="System.Web.Mvc.WebViewPage">
     <namespaces>
   . . .
         <add namespace="MvcApplication1 "/>
```

```
    </namespaces>
  </pages>
</system.web.webPages.razor>
```