**[MvcFluentHtml - Fluent HTML Interface For MS MVC](http://lunaverse.wordpress.com/2008/11/24/mvcfluenthtml-fluent-html-interface-for-ms-mvc/)**

Filed under: [C#](http://en.wordpress.com/tag/c/) — Tags: [HTML](http://en.wordpress.com/tag/html/), [MS MVC](http://en.wordpress.com/tag/ms-mvc/) — Tim Scott @ 6:45 pm

**UPDATE: MvcFluentHtml has been moved to** [**MvcContrib**](http://www.codeplex.com/MVCContrib) **where it lives in its own assembly, MvcContrib.FluentHtml.**

A few weeks ago I attended a presentation at the [KaizenConf](http://www.kaizenconf.com/) by [Jeremy Miller](http://codebetter.com/blogs/jeremy.miller) and [Chad Myers](http://www.lostechies.com/blogs/chad_myers) in which they showed their “opinionated” approach to MS MVC.  One of the things that caught my attention was the fluent interface they had created for HTML generation.   So I went home and started working on this for my own application.

I have put my work in progress here:

Incidentally, since I started working on MvcFluentHtml, Chad and Jeremy have put out some bits of their code as [opinionatedmvc](http://code.google.com/p/opinionatedmvc/).  And Karl Seguin, who also saw their presentation, has created some fluent HTML stuff which he shows in [this blog post](http://codebetter.com/blogs/karlseguin/archive/2008/11/13/textboxfor-u-gt-u-name-unleash-the-power.aspx).

So what problem does MvcFluentHtml solve?  Can’t I be happy with the HtmlHelpers that come with the framework?

I have come to truly hate the overloading approach taken by the out-of-the-box Html helpers.  Methods with long signatures are hard to read, and it takes investigation to see what’s happening.  What’s worse, you must worry about problems with overload resolution, especially when some parameters are typed as object.  As a result, HtmlHelper is not easily extensible.  It’s hard to bend it to do new things without breaking existing functionality.  We saw an example of this when Beta1 was released with [breaking changes](http://stackoverflow.com/questions/208468/issues-during-aspnet-mvc-upgrade-from-preview-5-to-beta#219181).  With a fluent interface, it’s much easier to extend with new behaviors while leaving existing behaviors closed.

So how does it work?  Let’s start with some examples of how you might use it in your views:

<%=this.TextBox(x => x.FirstName).Class("required").Label("First Name:")%>

<%=this.Select(x => x.ClientId).Options((SelectList)ViewData["clients"]).Label("Client:")%>

<%=this.MultiSelect(x => x.UserId).Options(ViewModel.Users)%>

<%=this.CheckBox("enabled").LabelAfter("Enabled").Title("Click to enable.").Styles(vertical\_align => "middle")%>

As you can see in this example, the helpers are methods of the view itself, not of some helper class.  You will also notice two different kinds of methods.  One kind takes the name as a string which is meant to evaluate to an object in the ViewData dictionary.  These methods are extensions of IViewDataContainer, and as such they can be used in any type of view (ViewPage, ViewPage<T>, ViewUserControl, ViewUserControl<T>, ViewMasterPage, ViewMasterPage<T>).   The other kind takes a lambda expression that refers to a member of ViewData.Model.  This kind of method extends a new interface that is defined in MvcContrib.FluentHtml:

public interface IViewModelContainer<T> where T : class

{

T ViewModel { get; }

IEnumerable<IMemberBehavior> MemberBehaviors { get; }

}

Both kinds of methods set the value (value attribute, inner text, etc.) from ViewData.  Both kinds set the name of form elements so that most binders will pick up the value on post.

So what must you do before you can use these methods in your views?  To use the methods that extend IViewDataContainer, nothing.  To use the methods that extend IViewModelContainer<T>, you have some options:

1. Derive your strongly-typed views from ModelViewPage<T>, ModelViewUserControl<T> or ModelViewMasterPage<T> which are bits of MvcFluentHtml.
2. Implement IViewModelContainer<T> directly on your views.
3. Define your own base view classes that implement IViewModelContainer<T>.

There is a fourth option.  MvcFluentHtml contains a set of classes, ConventionModelViewPage<T>, ConventionModelViewUserControl<T> and ConventionModelViewMasterPage<T>.  These classes apply some default behaviors.  These behaviors operate on certain model attributes, which are also part of MvcContrib.

So given this model:

public class Person

{

[Required]

[MaxLength(200)]

public string Name { get; set; }

}

And this in the view:

<%this.TextBox(x => x.Name)%>

The resulting HTML would be something like:

<input type="text" maxlength="200" id="Name" name="Name" class="required" value="Pete"/>

These attributes are found in a separate assembly, MvcContrib.ModelAttributes.  So in case your models are in a separate assembly from your MVC web UI, you don’t have to reference the UI specific bits in your model assembly.

But what if you want different behaviors and different attributes?  You can create custom behaviors by implementing IMemberBehavior.

public interface IMemberBehavior

{

void Execute(IMemberElement element);

}

For example:

public class MyMemberBehavior1 : IMemberBehavior

{

public void Execute(IMemberElement element)

{

var helper = new MemberBehaviorHelper<RequiredAttribute>();

var attribute = helper.GetAttribute(element);

if (attribute != null)

{

element.SetAttr("class", "req");

}

}

}

Then create your own “opinionated” views something like this:

public class MyOpinionatedModelViewPage<T> : ModelViewPage<T> where T : class

{

public MyOpinionatedModelViewPage() : base(new MyMemberBehavior1(), new MyMemberBehavior2()) { }

}

Or using your favorite IoC container:

public class MyOpinionatedModelViewPage<T> : ModelViewPage<T> where T : class

{

public MyOpinionatedModelViewPage(IMyMemberBehavior1Marker myBehavior1, IMyMemberBehavior2Marker myBehavior1)

: base(myMemberBehavior1, myMemberBehavior1) { }

}

One final point.  The goal of MvcFluentHtml was to leave the opinions to you.  We did this by allowing you to define your own behaviors.  However, it is not without opinions regarding practices. For example the Select object does not have any “first option” functionality. That’s because in my opinion adding options to selects is not a view concern. Also, while we allow you to set inline styles we do not expose any methods that abstract styling (e.g., Width, TextAlign).  That’s because we want to encourage semantic HTML and leave the styling to CSS.  For the same reason we do not render any layout tags (e.g. <br/>).  Also we generally don’t try to abstract the language of HTML (e.g., we use “Select” instead of “DropDownList”).  In my opinion, MvcFluentHtml is more of a helper library than an abstraction over HTML, and moving developers away from understanding what’s going on with the HTML is not a good thing.

I heartily welcome critique, especially if you want to contribute! Areas that I know need some work are: radio button, radio button list, checkbox list, performance tweaks, XML comments, and CI.

### [**Editing a Variable Length List Of Items With MvcContrib.FluentHtml**](http://lunaverse.wordpress.com/2009/01/02/editing-a-variable-length-list-of-items-with-mvccontribfluenthtml/)

Steve Sanderson recently showed  [a nifty way to edit variable length lists with MS MVC](http://blog.codeville.net/2008/12/22/editing-a-variable-length-list-of-items-in-aspnet-mvc/) .  I commented that I liked his approach.  I also mentioned that MvcContrib.FluentHtml might be used to improve a bit of awkward code used to build HTML element names.  To illustrate my point I offered some code examples.   Steve responded by pointing out the shortcomings of  my examples.  He was correct.   I had not really thought it through.  While my examples illustrated the use of FluentHtml, they missed the mark in terms of showing a real solution.

Still I knew that the basic idea of was sound.  This article shows in detail how to use MvcContrib.FluentHtml to edit a variable length list of items.

[Get The Code](http://www.lunaversesoftware.com/codebits/MvcEditableListDemo.zip)

I will use the same basic application as Steve did — a simple gift request form.

## Setup

We start with the Gift entity:

public class Gift

{

public int Id { get; set; }

public string Name { get; set; }

public decimal Price { get; set; }

}

## Render The View

Our view derives from MvcContrib.FluentHtml.ModelViewPage<IList<Gift>> and is rendered via a controller action:

[AcceptGet]

public ViewResult Index()

{

ViewData.Model = new List<Gift>

{

new Gift {Id = 1, Name = "Tar Tinker", Price = 23.44m},

new Gift {Id = 2, Name = "Flu Flooper", Price = 11.42m}

};

return View();

}

The view presents a list of the gifts:

<form action="<%=Html.BuildUrlFromExpression<HomeController>(x => x.Index(null))%>" method="post">

<div id="addGiftItem">

<a href="" id="addGift">Add</a>

</div>

<fieldset id="giftLineItems">

<legend>Gifts</legend>

<%for (var i = 0; i < ViewData.Model.Count; i++) {%>

<%Html.RenderPartial("GiftLineItem", ViewData.Model, new ViewDataDictionary {{"index", i}});%>

<%}%>

</fieldset>

<%=this.SubmitButton("Save")%>

</form>

GiftLineItem is a user control that derives from MvcContrib.FluentHtml.ViewModelUserControl<IList<Gift>>:

<%var i = (int)ViewData["index"];%>

<div class="giftLineItem">

<%=this.Hidden(x => x[i].Id)%>

<%=this.TextBox(x => x[i].Name).Label("Name of gift:")%>

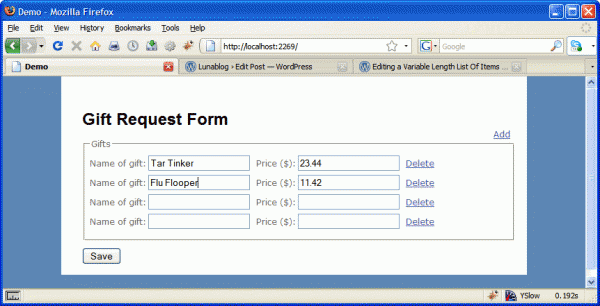
<%=this.TextBox(x => x[i].Price).Label("Price ($):")%>

<a href="" class="removeGift">Delete</a>

</div>

You might observe that this user control uses IList<Gift> as its model whereas Steve’s uses Gift.   But the user control represents one gift, right?  Have I sacrificed cohesion? Yes, however a closer look shows that neither Steve’s user control nor mine is cohesive.  That is, neither stands alone outside of the context of a list.  We simply use different techniques to name the elements within the context of an owning list.

Here’s the rendered view (after clicking “Add” twice):



## Add Behavior

Now we need to add behavior to the “Add” and “Delete” links.  For that we’ll use jQuery:

$(document).ready(function() {

var nextGiftIndex = <%=ViewData.Model.Count%>;

setRemoveLinks();

$('#addGift').click(function() {

$.get('/Home/AddGift?index=' + nextGiftIndex, 'html', function(lineItemHtml) {

$('#giftLineItems').append(lineItemHtml);

setRemoveLinks();

});

nextGiftIndex++;

return false;

});

});

function setRemoveLinks() {

$('a.removeGift').click(function() {

$(this).parent('div').remove();

return false;

});

}

Clicking the “Add” link calls the AddGift action asyncronously and passes the index of the item to be added.  The index value is initialized when the page is rendered and incremented on the client side with each addition.  The “Delete” links are wired up to a simple function that removes the parent div.

The AddGift action simply renders the user control using the specified index:

public ViewResult AddGift(int index)

{

ViewData["index"] = index;

return View("~/Views/Home/GiftLineItem.ascx");

}

Notice we have not specified a model for the user control, which means it will be null.  As a result, the name and price textboxes will be blank, which is what we want.  And by specifying the desired index, these textboxes will be named such that they will bind correctly when form is saved.

## Save Changes

Clicking the “Save” button posts the form to the following action:

[AcceptPost]

public ViewResult Index([Deserialize]IList<Gift> gifts)

{

//code to save the changes

ViewData.Model = gifts;

return View();

}

Notice that the parameter “gifts” is decorated with the Deserialize attribute. This attribute uses MvcContrib’s NameValueDeserializer to obtain our collection of gifts from the form post.  It does not require that collections be indexed purely in sequence. Thus it handles any holes left by deleted line items.

(NOTE: I expected that using [Bind(Prefix = "")] would work the same as [Deserialize], but it does not.  Seems to be a limitation.)

## Conclusion

The solution presented here differs from Steve’s in that it uses FluentHtml. There are a few other differences, such as jQuery versus MS Ajax.  My point is not present a solution that is superior, rather just to show how it might be done using  FluentHtml and expressions to avoid some fiddly construction of HTML element names in views.

### [**Editing a Variable Length List Of Items With MvcContrib.FluentHtml - Take 2**](http://lunaverse.wordpress.com/2009/01/13/editing-a-variable-length-list-of-items-with-mvccontribfluenthtml-take-2/)

Filed under: [MS MVC](http://en.wordpress.com/tag/ms-mvc/), [Uncategorized](http://lunaverse.wordpress.com/category/uncategorized/) — Tim Scott @ 11:48 pm

In a [previous post](http://lunaverse.wordpress.com/2009/01/02/editing-a-variable-length-list-of-items-with-mvccontribfluenthtml/), I showed how to edit a variable length list using MvcContrib.FluentHtml.  The post was inspired by a [post on the same topic by Steve Sanderson](http://blog.codeville.net/2008/12/22/editing-a-variable-length-list-of-items-in-aspnet-mvc/).  Steve commented on my post pointing out some limitations.  To address these limitations required some enhancements to MvcContrib.FluentHtml and some changes to my example.  So here we go with take 2.

# MvcContrib.FluentHtml Enhancements

The limitations were mostly related to handling validation.  To support validation using Model State we have added to MvcContrib.FluentHtml:

* Html Prefix Support
* A Validation Helper
* List Indexing Support

## Html Prefix Support

HTML elements generated for strongly typed views have no prefix by default.  So elements for ModelViewPage<Person> might look something like this:

<input name="FirstName" id="FirstName" value="Jim" type="text"/>

<input name="LastName" id="LastName" value="Smith" type="text"/>

The problem is, neither MvcContrib’s NameValueSerializer nor MS MVC’s default model binder fully handle this scenario.   While NameValueDeserializer does the binding just fine, it does not place bind errors into ModelState.  The default model binder does not handle quite handle the binding — it does not deserialize enumerable properties when no prefix is used.

To bind everything correctly and get any errors into ModelState we need a prefix.  One way to handle this is to wrap our primary model in a view model. This is a valid pattern. It’s even a standard for some applications. However, we don’t want to have to do this just to get a prefix. So we provide a way to specify a prefix for a strongly typed view.

Either in the code-behind:

public class Index : ModelViewPage<IList<Gift>>

{

public Index() : base("person") { }

}

Or in the view:

<%this.HtmlNamePrefix = "person"%>

Thus our HTML elements will be prefixed:

<input name="person.FirstName" id="person\_FirstName" value="Jim" type="text"/>

<input name="person.LastName" id="person\_LastName" value="Smith" type="text"/>

Then in our action method, we can use the default binder without any attribute.  Any bind errors will be captured in ModelState.

[AcceptPost]

public ViewResult Index(Person person)

## New Validation Helper

We have added validation support to FluentHtml for strongly typed views. The following works basically the same as HtmlHelper’s ValidationMessage.

<%this.ValidationMessage(x => x.Price, "Price must be numeric.")%>

We have also added a behavior to ModelViewPage<T>, ModelViewMasterPage<T> and ModelViewUserControl<T> which basically mimics validation used by HtmlHelper.  That is, it adds a CSS class “input-validation-error” to any HTML element with an error in ModelState.  If you wish, you can remove this behavior or change the CSS class name in the derived class.

## List Indexing Support

MS MVC uses [a special technique](http://haacked.com/archive/2008/10/23/model-binding-to-a-list.aspx) to deserialize enumerable properties.  With this technique you set an arbitrary value in a specially named hidden element to signify that a group of elements belongs to a particular instance of the enumerable property. This has an advantage over using positional indexes in that it preserves the identity of an instance across posts, which is necessary for ModelState based validation to work properly.

Therefore, we have added support for this to MvcContrib.FluentHtml.  Let’s assume ModelViewPage<IList<Person>> with a prefix of “persons”.  Then this markup:

<%var id = Model[i].Id;%>

<%=this.Index(x => x).Value(id)%>

<%=this.TextBox(x => x[id].FirstName).Value(Model[i].FirstName)%>

Will generate HTML like this:

<input id="persons\_Index\_123" name="persons.Index" value="123" type="hidden"/>

<input id="persons[123]\_FirstName" name="persons[123].FirstName" value="Jim" type="text"/>

# Changes To Our Example

So based on these changes to FluentHtml we made a few changes to the “Gift Request Form” demo from the previous post. In the main view:

<%for (var i = 0; i < ViewData.Model.Count; i++) {%>

<%ViewData["index"] = i;

Html.RenderPartial("GiftLineItem");%>

<%}%>

And in the user control:

<%var i = (int)ViewData["index"];

var gift = ViewModel == null ? null : ViewModel[i];

var id = gift == null ? -1 \* new Random().Next() : gift.Id;

var name = gift == null ? null : gift.Name;

var price = gift == null ? (decimal?)null : gift.Price;%>

<div class="giftLineItem">

<%=this.Index(x => x, x => x[i].Id)%>

<%=this.Hidden(x => x[id].Id).Value(id)%>

<%=this.TextBox(x => x[id].Name).Value(name).Label("Name of gift:")%>

<%=this.TextBox(x => x[id].Price).Value(price).Label("Price ($):")%>

<%=this.ValidationMessage(x => x[id].Price, "Must be a number")%>

<a href="" class="removeGift">Delete</a>

</div>

Viola, validation works:



Note that we stole Steve’s technique of using random negative Ids for unsaved instances, which we will handle in the save method on the server.

[**Eliminate Magic Strings In Javascript With MvcContrib.FluentHtml**](http://lunaverse.wordpress.com/2009/02/05/eliminate-magic-strings-in-javascript-with-mvccontribfluenthtml/)

Filed under: [Uncategorized](http://lunaverse.wordpress.com/category/uncategorized/) — Tim Scott @ 3:27 am

[MvcContrib](http://www.codeplex.com/MVCContrib).[FluentHtml](http://lunaverse.wordpress.com/2008/11/24/mvcfluenthtml-fluent-html-interface-for-ms-mvc/) provides a library of HTML helper methods that use expressions with strongly typed views to generate HTML elements. Not too long after that MS MVC beta introduced HtmlHelper<T> which allows users to create extension methods that use expressions in the same way.   The benefits are obvious.  By trading magic strings for expressions we gain compile time checking, better refactoring, etc.

So this:

<%=this.TextBox(x => x.User.FirstName)%>

Generates this:

<input type="text" name="User.FirstName" id="User\_FirstName" value="Joe"/>

This is very nice, but what about all the other magic strings in our view?  I’m talking of course about Javascript. Does this look familiar?

$('#User\_FirstName').change(doSomething);

Doh!  We just can’t escape those magic strings. Or can we?  How about this?

$('#<%=this.IdFor(x => x.User.FirstName)%>').change(doSomething);

MsMvc.FluentHtml has introduced IdFor and NameFor. Okay, so we won’t eliminate *all* magic string in Javascript but perhaps quite a few. Perhaps we give up a tiny bit of readability, but I think it’s worth it to stop fretting whether some refactor is going to break a bunch of Javascript.

I owe credit to [Jimmy Bogard](http://www.lostechies.com/blogs/jimmy_bogard/default.aspx) for planting this idea in my mind in one of his tweets.