Title: ASP.NET MVC Partial Views with Partial Models

Author: Dave W Hunt

Email: dave.hunt@cambseng.com

Language: C#

Platform: Windows

Technology: ASP.NET MVC 5

Level: Intermediate

Description: Wouldn't it be nice to use a partial view with its own model on multiple pages?

Section: Web Development

SubSection: ASP.NET

License: CPOL

# Introduction

This article solves the problem of posting back Partial Views where the view content contains form elements.

Code reuse is a very useful time saving feature and any good engineer will build up lots of useful functions during the course of their work. It’s common sense for Web applications to make use of the same code over and over again. However in this instance we also have HTML mark-up code. ASP.NET MVC has Partial Views, Child Actions and Editor/Display templates to solve this problem. Partial Views can use the Page Model for their data whereas Child Actions use independent data from the Controller. Editor/Display templates pass items from the model to the system but can be overridden by user partial views. This article looks at problems when posting back data from Partial Views by taking the example of a company address where part of its content is the postal address. The postal address, when implemented as a partial view, can then be re-used in other pages such as a sales address. Once this problem is solved large pages can become more manageable by breaking them up into several parts. A good example of this would be the content under tabs.

# Background

Before we dive into the code I would like to provide a short background. I am a professional software engineer working for a small company. I have been on the Microsoft bandwagon since Visual Basic 6. What gives me the buzz is when I solve a problem that I believe will be useful for future projects and benefit my colleagues. This doesn’t happen very often, but when it does, I add it to a class library. I have included a part of this in the sample project. I also use One-Note to quickly record any of these ideas with links to other engineers’ findings.

The first port of call when you hit a problem is to see if anybody else has had it. All engineers need to have good searching capabilities. There are many forums out there, including Code Project, and lots of ideas can be obtained before resorting to doing your own thing.

When working with Microsoft MVC for Web design most of the features are included, but occasionally you need something else. The area of forms is one of these. Microsoft use HTML helpers to generate form elements. These go a long way, but there are a lot of omissions. Fortunately they are easy to extend and many examples can be found on the internet.

This article will present a problem and solve it by first looking at what others have done and then producing a new HTML helper that solves the problem.

# The Problem

First let’s look at the standard MVC Controller pattern used for forms:

[HttpGet]

public ActionResult Index()

{

// load the test data

TestViewModel model = new TestViewModel();

return View(model);

}

[HttpPost]

public ActionResult Index(TestViewModel model)

{

if (ModelState.IsValid)

{

// save the test data

}

return View(model);

}

The HttpGet action loads data into a model and calls the view to display the data. The view displays the data on a form which the user can edit. Clicking a submit button posts the form back to the controller’s HttpPost action binding the data on the form to the model. The controller validates the model and if all is well then saves the data. The controller then calls the same view to display the edited data. Note the same model is used in both actions and sticking to this pattern guarantees the data binding will work even with complex models that contain other classes and/or lists. In fact this is one of the very clever things MVC does for us if we stick to this pattern.

Here is the very simple model:

public class TestViewModel

{

public TestModel Test { get; set;}

}

public class TestModel

{

[Display(Name = "Name:")]

[Required(ErrorMessage = "Please provide a name")]

public string Name { get; set; }

public TestPartialModel Partial { get; set; }

}

public class TestPartialModel

{

[Display(Name="Partial Name:")]

[Required(ErrorMessage="Please provide a name")]

public string Name { get; set; }

}

And here is the View:

@model CSE.Partial.WebApp.Models.TestViewModel

@{

ViewBag.Title = "Test";

}

<h2>@ViewBag.Title</h2>

@using (Html.BeginForm())

{

@Html.AntiForgeryToken()

<div>

<hr />

<dl class="dl-horizontal">

<dt>@Html.LabelFor(m => m.Test.Name)</dt>

<dd>

@Html.EditorFor(m => m.Test.Name)

@Html.ValidationMessageFor(m => m.Test.Name)

</dd>

<dt>@Html.LabelFor(m => m.Test.Partial.Name)</dt>

<dd>

@Html.EditorFor(m => m.Test.Partial.Name)

@Html.ValidationMessageFor(m => m.Test.Partial.Name)

</dd>

<dt></dt>

<dd><input class="btn btn-primary" type="submit" value="Save" /></dd>

</dl>

</div>

}

This works as expected and when pressing the save button any inputted values are posted back to the controller and the new view is returned based on those values. So the test is simply the values being persisted.

Next step is to move the Partial bit into a partial view.

\_TestPartial which can be in the same view folder or in Shared

@model CSE.Partial.WebApp.Models.TestViewModel

<dt>@Html.LabelFor(m => m.Test.Partial.Name)</dt>

<dd>

@Html.EditorFor(m => m.Test.Partial.Name)

@Html.ValidationMessageFor(m => m.Test.Partial.Name)

</dd>

and the view is now:

@model CSE.Partial.WebApp.Models.TestViewModel

@{

ViewBag.Title = "Test";

}

<h2>@ViewBag.Title</h2>

@using (Html.BeginForm())

{

@Html.AntiForgeryToken()

<div>

<hr />

<dl class="dl-horizontal">

<dt>@Html.LabelFor(m => m.Test.Name)</dt>

<dd>

@Html.EditorFor(m => m.Test.Name)

@Html.ValidationMessageFor(m => m.Test.Name)

</dd>

@Html.Partial("\_TestPartial")

<dt></dt>

<dd><input class="btn btn-primary" type="submit" value="Save" /></dd>

</dl>

</div>

}

This works fine as expected as the whole model is passed by default to the partial view.

But I want the partial view to be re-usable and in the above example it is tied to the page’s model.

So let’s try and pass the partial model to the partial view.  
Modify the partial view to use its own model:

@model CSE.Partial.Service.Models.TestPartialModel

<dt>@Html.LabelFor(m => m.Name)</dt>

<dd>

@Html.EditorFor(m => m.Name)

@Html.ValidationMessageFor(m => m.Name)

</dd>

Pass the correct model from the page view:

@Html.Partial("\_TestPartial", Model.Test.Partial)

Now when you post back there is an exception. On examining the posted back data the partial model is missing. This is the problem!

# Solving the problem

So what’s going wrong? It’s pretty obvious when you look at the HTML produced using the F12 tool or view source. The element naming that was present on the working tests is different on the final test.

This works:

<input name="Test.Partial.Name" id="Test\_Partial\_Name" type="text" value="">

This fails:

<input name="Name" id="Name" type="text" value="">

The name prefix is missing in the partial HTML. Pretty obvious really; how on earth could it possibly know it was part of a bigger model!

A search of the forums reveals other people trying to do the same thing and we discover that built into MVC is a class called TemplateInfo. This has a property named HtmlFieldPrefix. If we set this before calling the partial view we can force all element names to have a prefix.

@{ Html.ViewData.TemplateInfo.HtmlFieldPrefix = "Test.Partial"; }

@Html.Partial("\_TestPartial", Model.Test.Partial)

That works in this scenario but any element after the partial would also get the prefix.

Then there is the 3rd parameter in @Html.Partial( , , ViewDataDictionary). We can pass a brand new ViewData to out partial view with the HtmlFieldPrefix set as above.

@Html.Partial("\_TestPartial", Model.Test.Partial, new ViewDataDictionary

{

TemplateInfo = new System.Web.Mvc.TemplateInfo

{

HtmlFieldPrefix = "Test.Partial"

}

})

That isolates the prefix to the partial view. But now the ModelState and ViewData are missing! The error messages now fail to appear. There is a lot of stuff missing in that 3rd parameter that got passed by default. If we first take a copy of the view data and then modify the TemplateInfo that should fix this problem. The ViewDataDictionary constructor will copy Html.ViewData for us.

@Html.Partial("\_TestPartial",

Model.Test.Partial,

new ViewDataDictionary(Html.ViewData)

{

TemplateInfo = new System.Web.Mvc.TemplateInfo

{

HtmlFieldPrefix = "Test.Partial"

}

})

There is still one last problem. If we use this technique with nested partials the prefix will have to be appended rather than set. I found a method in MVC to do this. So here is a working solution.

@{

string name = Html.NameFor(m => m.Test.Partial).ToString();

string prefix = Html.ViewContext.ViewData.TemplateInfo.GetFullHtmlFieldName(name);

ViewDataDictionary viewData = new ViewDataDictionary(Html.ViewData)

{

TemplateInfo = new TemplateInfo { HtmlFieldPrefix = prefix }

};

}

@Html.Partial("\_TestPartial", Model.Test.Partial, viewData)

This does the trick but it’s a bit messy and it’s a lot of code to put in front of each Html.Partial. So now it’s working let’s turn this into a HTML helper. This is what we would like:

@Html.PartialFor(m => m.Test.Partial)

Here is the extension method. Notice the lambda expression. This is a great type safe way of passing the helper the name and value of the partial model. The partial view name can be passed as a parameter or it will be set to the class name of TProperty or a UIHint("template name") if present.

/// <summary>

/// Return Partial View.

/// The element naming convention is maintained in the partial view by setting the prefix name from the expression.

/// The name of the view (by default) is the class name of the Property or a UIHint("partial name").

/// @Html.PartialFor(m => m.Address) - partial view name is the class name of the Address property.

/// </summary>

/// <param name="expression">Model expression for the prefix name (m => m.Address)</param>

/// <returns>Partial View as Mvc string</returns>

public static MvcHtmlString PartialFor<TModel, TProperty>(this HtmlHelper<TModel> html,

Expression<Func<TModel, TProperty>> expression)

{

return html.PartialFor(expression, null);

}

/// <summary>

/// Return Partial View.

/// The element naming convention is maintained in the partial view by setting the prefix name from the expression.

/// </summary>

/// <param name="partialName">Partial View Name</param>

/// <param name="expression">Model expression for the prefix name (m => m.Group[2])</param>

/// <returns>Partial View as Mvc string</returns>

public static MvcHtmlString PartialFor<TModel, TProperty>(this HtmlHelper<TModel> html,

Expression<Func<TModel, TProperty>> expression,

string partialName

)

{

string name = ExpressionHelper.GetExpressionText(expression);

string modelName = html.ViewContext.ViewData.TemplateInfo.GetFullHtmlFieldName(name);

ModelMetadata metaData = ModelMetadata.FromLambdaExpression(expression, html.ViewData);

object model = metaData.Model;

if (partialName == null)

{

partialName = metaData.TemplateHint == null

? typeof(TProperty).Name // Class name

: metaData.TemplateHint; // UIHint("template name")

}

// Use a ViewData copy with a new TemplateInfo with the prefix set

ViewDataDictionary viewData = new ViewDataDictionary(html.ViewData)

{

TemplateInfo = new TemplateInfo { HtmlFieldPrefix = modelName }

};

// Call standard MVC Partial

return html.Partial(partialName, model, viewData);

}

The extra bits in here are the 2 methods from MVC:

ExpressionHelper.GetExpressionText gets the name of the expression m => m.Test.Partial would return “Test.Partial”.

ModelMetadata.FromLambdaExpression gets the value of the expression. e.g. the model Partial.

I am not claiming to be the total inventor of this extension but more of a distiller of several people’s ideas to form a working method.

# Editor/Display templates

After solving the problem above, I thought I would investigate Editor Templates. As per Microsoft MSDN:

*The EditorFor method is used to generate MVCHtmlString mark-up depending on the data type of the expression passed into it.*

There are lots of examples of this on the web and you come away thinking that it is a single method that will adapt its rendered output to suit the type of the property and use the metadata of the property (property attributes) to add extra detail. E.g. it will show a text box for a string and a checkbox for a bool. It will set the type of input to the appropriate Html5 attribute. When you use an object it spits out <div>s with labels and input elements for each property. This is where it gets interesting as you can override the default behaviour of all the system templates by adding a folder called EditorTemplates and/or DisplayTemplates to the view’s folder or the Shared folder. In this you place a partial view with the name of the type you are overriding. This applies to simple types and complex types. I tried this out with the project above and as if by magic it worked in exactly the same way as the PartialFor extension method. It also passed the prefixes down through the layers.

In the project (available by the zip link) I have named all the partials with their model class name and placed them in the ViewName/DisplayTemplate or the Shared/DisplayTemplate folders. You can then use them like this:

@Html.EditorFor(m => m.Partial.First)

The Html.Partial expects the partial view to be in the current view folder or in the shared folder and not the sub-folder. For testing I overrode this by using the partial name parameter to use the same partial view used by EditorFor.

@Html.PartialFor(m => m.Partial.First, "EditorTemplates/FirstPartialModel")

## PartialFor vs EditorFor

So what is the difference? Not much as far as I can make out. The main difference is the location of the partial view.

* PartialFor – Same as Partial; in the current view folder or in the Shared folder
* EditorFor – In EditorTemplates sub-folder of view or shared folder

I ran the debugger and traced into the MVC EditorFor code (see appendix on how to do this). It worked out the location and then tested the type to decide it was a complex class, copied the ViewData, set the appended prefix in TemplateInfo and called the same partial view code as @Html.Partial does. If you search the internet for “html.editorfor vs partial view” you find a lot of discussions. I wish I had found this a long time ago. One advantage of PartialFor would be when you wanted to control the absolute location of the partial view. PartialFor also works with inheritance, interfaces and collections.

* @Html.PartialFor(expression, partialName); partialName can be a name, full path or a relative path.
* @HtmlEditorFor(expression, templateName); templateName can be a name or a relative path. “EditorTemplates” is hard coded in the source code.

## Interfaces and inheritance

Armed with this new knowledge I started using templates to define the views for my data objects to be used on pages using a form. All went very well until I tried to split out the inherited part of a class into its own partial view. I thought this was a very good candidate for code re-use. EditorFor rendered absolutely nothing whereas PartrialFor worked as expected. Into the debugger and tracing MVC code revealed that the EditorFor code was testing for previously visited objects and rejected the inherited part of the object. It is basically trying to protect us from recursion but in our case it is not going to recurse, it is just going to render the derived and inherited parts of the object as separate partial views. So I am glad I did not delete my version of PartialFor.

## Partial View vs ChildAction vs EditorFor

|  |  |  |  |
| --- | --- | --- | --- |
|  | Centricity | Description | Uses |
| Partial view | View centric | Works with no model or the page model. Can work with a sub-model if it’s read-only. | Nice for splitting mark-up on a complex page. e.g. Tabs. Dynamic view selection when used on layout pages. |
| ChildAction | View centric | Creating partial views as a controller method [ChildActionOnly]. Call using @Html.Action("action"). Can use controller logic before rendering the view. Does not use the page model. | Good for re-use of independent views on many pages. |
| EditorFor | Model centric | Works with simple types and complex models and maintains the element naming convention for correct post back model binding. Element naming adheres to model hierarchies. | Excellent for re-usable data model views used in forms. |
| DisplayFor | Model centric | By convention a read-only version of EditorFor. |  |
| PartialFor | Model centric | Similar to EditorFor to be used with complex objects. No test for recursive objects. | With inherited or interface parts of objects. Works with collections. |

Which? It’s up to you to encourage maximum code reuse avoiding repetition.

# Using the code

You will need Microsoft Visual Studio 2013 to run this solution. The test project can be downloaded in a zip file. Unzip the file into a test folder and double click the solution file (CSE.Partial.WebApp.sln). Before trying to run the project you will need to restore the NuGet packages. This may work automatically depending on your Visual Studio setup. If it doesn’t try:

* Right click solution and click Enable Nuget Package Restore
* Make sure Tools/Options/NuGet Package Manager/General Package Restore and check Allow NuGet to download missing packages.

The menu items contain a drop down named “Partial”. Clicking this will show 3 test views.

* “Postal Address” shows a form containing a company details and a postal address. The postal address is implemented in a partial view. It is a simple demo but shows the separation of the partial view. In the future the postal address could be extended to use a post code lookup service to provide a very nice “Component” for re-use.
* “Nested Partial” is a page that tests partials within partials and shows the databinding workings at more than one depth. It also proves ModelState error processing still works and that the ViewData is being passed into the partial. Several alternative solutions I found on the forums failed in this respect.
* “Test” is the source for “The Problem” and “Solving the problem” above.

# Points of Interest

I believe the controller pattern used above is great and the data binding works magically. I found the ability to trace though MVC code very helpful. In fact I would not have been able to do the above without it. The CSE namespace uses the initials of my company Cambridge Software Engineering.

# Appendix

## Stepping into .NET framework code

Instructions from MSDN for a Symbol/source server.

To configure Visual Studio for symbol/server use, follow these instructions:

1. Go to Tools -> Options -> Debugger -> General.
2. Uncheck “Enable Just My Code (Managed only)”.
3. Uncheck “Enable .NET Framework source stepping”. Yes, it is misleading, but if you don't, then Visual Studio will ignore your custom server order (see further on).
4. Check “Enable source server support”.
5. Uncheck “Require source files to exactly match the original version”
6. Go to Tools -> Options -> Debugger -> Symbols.
7. Select a folder for the local symbol/source cache.
8. <http://srv.symbolsource.org/pdb/Public> adds MVC
9. Setup as below

Machine generated alternative text: Options
? X
Search Options (Ctrl+E)
L> Projects and Solutions
L> Source Control
L> Text Editor
A Debugging
General
Edit and Continue
Just-In-Time
Output Window
Symbols
L> IntelliTrace
L> Performance Tools
L> Analysis Services Tabular Designers
L> Business Intelligence Designers
L> Database Tools
L> F#Tools
L> Graphics Diagnostics
L> HTML Designer
Load all symbols
Automatically load symbols for
®All modules, unless excluded
Specify excluded mcdules
Q Qnly specified modules
Specify modules
p Symbols
Symbol file (.pdb) locations:
Microsoft Symbol Servers
http://srvsymbolsource.org/pdb/Public
<
Cache symbols in this directory:
H
C:\SymCache
Browse... J
empty Symbol Cache
OK [ Cancel

# History

Version 1 – 10 Feb 2016