**Rewriting the MvcContrib Grid**

February 8, 2009 – 2:47 pm

This is part 1 of a series on the new MvcContrib grid component.

* [Part 1 - Introduction](http://www.jeremyskinner.co.uk/2009/02/08/rewriting-the-mvccontrib-grid/)
* [Part 2 - New Syntax](http://www.jeremyskinner.co.uk/2009/02/22/rewriting-the-mvccontrib-grid-part-2-new-syntax/)
* [Part 3 - GridModels and GridRenderes](http://www.jeremyskinner.co.uk/2009/02/23/rewriting-the-mvccontrib-grid-part-3-gridmodels-and-gridrenderers)
* [Part 4 - Limitations of the WebFormsViewEngine](http://www.jeremyskinner.co.uk/2009/02/28/mvccontrib-grid-part-4-limitations-of-the-webformsviewengine)
* [Part 5 - The Action Syntax](http://www.jeremyskinner.co.uk/2009/03/01/mvccontrib-grid-part-5-the-action-syntax)

Over the next couple of weeks I’m hoping to write a few posts on the upgrades I’m doing to the [MvcContrib grid component.](http://www.codeplex.com/MVCContrib/Wiki/View.aspx?title=Grid&referringTitle=Documentation) These are the features that I’m planning on adding:

**A Fluent Interface**

Like the HTML-helpers built into the ASP.NET MVC framework, the mvccontrib grid suffers from ‘overload hell’. Switching to a fluent interface should provide a cleaner and more discoverable API.

**Discouraging logic in Views**

One side effect of the fluent interface is that it will no longer be possible to use the grid’s section overrides. However, I consider this to be a good thing. Embedding custom sections makes it all to easy to get stuck in ‘tag soup’. To make up for this, the new grid will support rendering partials for custom sections.

**Reusable Grid Models**

The new grid will support reusable grid definitions. That is, you will be able to define a grid in a external class and then reference this from your views. One benefit of this approach is that it means the grid will finally be accessible to VB users. The current grid is C#-only due to its use of lambda-statements for defining columns.

**Customisable Grid Rendering**

The existing grid only supports rendering in an HTML table. This means that if you wanted to render a grid in a different format (for example, as an OpenXML Spreadsheet) it meant you needed to inherit from the Grid class, override several methods and then create a new helper. The new grid will allow a custom GridRenderer class to be provided in the grid definition. Rendering as an HTML table will still be the default behaviour.

**Separate Pagination Component**

The current grid has built in support for paging. With the new grid, this will be moved into a separate component.

**Sorting**

Time permitting, the new grid will have support for sorting.

In my next post I’ll focus on the grid’s Fluent Interface.

**Rewriting the MvcContrib Grid part 2 - New Syntax**

February 22, 2009 – 2:40 pm

This is part two of a series on the new MvcContrib grid component.

* [Part 1 - Introduction](http://www.jeremyskinner.co.uk/2009/02/08/rewriting-the-mvccontrib-grid/)
* [Part 2 - New Syntax](http://www.jeremyskinner.co.uk/2009/02/22/rewriting-the-mvccontrib-grid-part-2-new-syntax/)
* [Part 3 - GridModels and GridRenderes](http://www.jeremyskinner.co.uk/2009/02/23/rewriting-the-mvccontrib-grid-part-3-gridmodels-and-gridrenderers)
* [Part 4 - Limitations of the WebFormsViewEngine](http://www.jeremyskinner.co.uk/2009/02/28/mvccontrib-grid-part-4-limitations-of-the-webformsviewengine)
* [Part 5 - The Action Syntax](http://www.jeremyskinner.co.uk/2009/03/01/mvccontrib-grid-part-5-the-action-syntax)

**Edit: 28 Feb 09** I have removed the use of Partial views in the latest MvcContrib trunk for certain areas where it didn’t really make sense. The examples below have been updated to reflect this.

The updated version of the MvcContrib grid has a new syntax that uses method chaining and lambda expressions to build a fluent interface. Here is a grid defined with the new syntax:

**<%=** Html.Grid**(**Model.People**)**.Columns**(**column **=>** **{**

column.**For(**x **=>** x.Id**)**.Named**(**"Person ID"**);**

column.**For(**x **=>** x.Name**);**

column.**For(**x **=>** x.DateOfBirth**)**.Format**(**"{0:d}"**);**

**})**

.Attributes**(**style **=>** "width:100%"**)**

.Empty**(**"There are no people."**)**

.RowStart**(**row **=>** "<tr foo='bar'>"**)** **%>**

Compare this to the same grid defined using the old syntax:

**<%**

Html.Grid**(**Model.People,

**new** Hash**(**empty **=>** "There are no people", style **=>** "width:100%"**)**,

column **=>** **{**

column.**For(**x **=>** x.Id, "Person Id"**);**

column.**For(**x **=>** x.Name**);**

column.**For(**x **=>** x.DateOfBirth**)**.Format**(**"{0:d}"**);**

**}**, sections **=>** **{**

sections.RowStart**(**item **=>** **{** **%>**

<tr> <!-- some custom html for the row start here -->

**<%** **});**

**%>**

Note how grid options (eg Empty) are now specified as part of the fluent interface rather than part of the HTML Attributes dictionary (which I always felt was messy). The use of method chaining also means that it is no longer possible to embed html blocks inside the grid definition (eg RowStart). However, you can still achieve the same functionality using the **RowStart** method and specifying specifying a lambda that will return the HTML string to output.

The column definitions are almost identical identical.

**Paging support**

Unlike its predecessor, the new grid does not have any logic for dealing with paging. This has all been moved to a separate **Pager** component. The old grid rendered the pager in its footer, while with the new grid you’d need to write your page like this:

**<%=** Html.Grid**(**Model.People**)**.Columns**(**column **=>** **{**

column.**For(**x **=>** x.Id**)**.Named**(**"Person ID"**);**

column.**For(**x **=>** x.Name**);**

column.**For(**x **=>** x.DateOfBirth**)**.Format**(**"{0:d}"**);**

**})** **%>**

**<%=** Html.Pager**(**Model.People**)** **%>**

**Where’s the code?**

The code for the new grid is not in the mvccontrib release, so if you want to play with it you’ll need to download and build [the source](http://mvccontrib.googlecode.com/svn/trunk). Alternatively, I’ve put together a custom build that can be[downloaded here](http://www.jeremyskinner.co.uk/files/mvccontrib-r829.zip) for the subversion-impaired.

In my next post, I’ll be focussing on extending the grid by using GridModels and GridRenderers.

**Rewriting the MvcContrib Grid part 3 - GridModels and GridRenderers**

February 23, 2009 – 7:47 pm

This is part 3 of a series about the new MvcContrib grid.

* [Part 1 - Introduction](http://www.jeremyskinner.co.uk/2009/02/08/rewriting-the-mvccontrib-grid/)
* [Part 2 - New Syntax](http://www.jeremyskinner.co.uk/2009/02/22/rewriting-the-mvccontrib-grid-part-2-new-syntax/)
* [Part 3 - GridModels and GridRenderes](http://www.jeremyskinner.co.uk/2009/02/23/rewriting-the-mvccontrib-grid-part-3-gridmodels-and-gridrenderers)
* [Part 4 - Limitations of the WebFormsViewEngine](http://www.jeremyskinner.co.uk/2009/02/28/mvccontrib-grid-part-4-limitations-of-the-webformsviewengine)
* [Part 5 - The Action Syntax](http://www.jeremyskinner.co.uk/2009/03/01/mvccontrib-grid-part-5-the-action-syntax)

Internally, the new MvcContrib grid makes use of the **GridModel** and **GridRenderer** classes to create the grid definition and then render it. It is possible to make use of these classes yourself to customise the grid.

**Grid Models**

The GridModel stores the definition for the grid. That is, it contains column definitions, custom attributes and the renderer to use (see below). Each time you call a method on the Grid class it is delegated to the underlying GridModel. For example, calling the **Columns** defines a collection of GridColumn objects which are instantiated and stored in the GridModel:

**<%=** Html.Grid**(**Model.People**)**.Columns**(**column **=>** **{**

column **=>** column.**For(**x **=>** x.Name**);**

column **=>** column.**For(**x **=>** x.DateOfBirth**);**

**})** **%>**

Instead of defining columns in your view, you can create your own GridModel and then pass this object to a call to Html.Grid:

public class PersonGridModel : GridModel<Person> {

public PersonGridModel() {

Column.For(x => x.Name);

Column.For(x => x.DateOfBirth);

}

}

**<%=** Html.Grid**(**Model.People**)**.WithModel**(new** PersonGridModel**())** **%>**

Note that GridModels aren’t limited just to column definitions. Everything that you could invoke by calling Html.Grid has an equivalent method on the GridModel (eg setting Empty Text, custom section overrides, header attributes etc).

There are several advantages of this approach:

* If you have large grid definitions using a GridModel can help to keep the view uncluttered.
* It is easy to re-use the same grid in multiple locations.
* You can re-use common grid code through inheritance.

There is one other benefit to using a GridModel. Visual Basic 2008 doesn’t support Lambda Statements meaning that it is not possible to use the Columns() method on the Grid directly. By using a grid model then you can still make use of the grid from within VB (if you’re that way inclined).

**Grid Renderers**

Internally, it is the job of the GridRenderer to take a GridModel and render it to the output stream. The default GridRenderer is the **HtmlTableGridRenderer**. Which, surprisingly enough, renders the grid as an HTML table. By specifying your own GridRenderer, you can completely alter how the grid is rendered without resorting to completely rewriting the grid component.

For example, in one of my larger ASP.NET MVC applications I frequently have the need to output data as an Excel spreadsheet and I used the grid to do so. In order to make this work I had to inherit from GridBase and override the methods related to generating output and then define a new set of helper methods to cater for the various Grid overloads. In the end I ended up with several overloads like this:

public static void ExcelGrid<T>(this HtmlHelper helper, string viewDataKey, Action<IRootGridColumnBuilder<T>> columns) where T : class

public static void ExcelGrid<T>(this HtmlHelper helper, string viewDataKey, Action<IRootGridColumnBuilder<T>> columns, Action<IGridSections<T>> sections) where T : class

public static void ExcelGrid<T>(this HtmlHelper helper, string viewDataKey, IDictionary htmlAttributes, Action<IRootGridColumnBuilder<T>> columns) where T : class

public static void ExcelGrid<T>(this HtmlHelper helper, string viewDataKey, IDictionary htmlAttributes, Action<IRootGridColumnBuilder<T>> columns, Action<IGridSections<T>> sections) where T : class

public static void ExcelGrid<T>(this HtmlHelper helper, IEnumerable<T> dataSource, Action<IRootGridColumnBuilder<T>> columns) where T : class

public static void ExcelGrid<T>(this HtmlHelper helper, IEnumerable<T> dataSource, Action<IRootGridColumnBuilder<T>> columns, Action<IGridSections<T>> sections) where T : class

public static void ExcelGrid<T>(this HtmlHelper helper, IEnumerable<T> dataSource, IDictionary htmlAttributes, Action<IRootGridColumnBuilder<T>> columns) where T : class

public static void ExcelGrid<T>(this HtmlHelper helper, IEnumerable<T> dataSource, IDictionary htmlAttributes, Action<IRootGridColumnBuilder<T>> columns, Action<IGridSections<T>> sections) where T : class

…which was largely a duplication of the Grid overloads in mvccontrib. They could then be used like this:

**<%** Html.ExcelGrid**(**Model.People, column **=>** **{**

column.**For(**x **=>** x.Name**);**

column.**For(**x **=>** x.DateOfBirth**);**

**});** **%>**

…which would produce the grid as Excel-XML.

With the new grid I can create a new GridRenderer implementation and then use it like this, with no need to create additional HtmlHelper extensions:

**<%=** Html.Grid**(**Model.People**)**.Column**(**column **=>** **{**

column **=>** column.**For(**x **=>** x.Name**);**

column **=>** column.**For(**x **=>** x.DateOfBirth**);**

**})**.RenderUsing**(new** ExcelGridRenderer**<**Person**>());**

The ExcelGridRenderer could inherit from GridRenderer and override the methods needed for constructing the grid header, footer, rows etc. You may notice that the methods on GridRenderer are very similar to those on the GridBase class. This is not a coincidence - most of the code in the GridRenderer is re-used from the old GridBase.

Note it is also possible to specify the renderer from within a GridModel:

public class PersonGridModel : GridModel<Person> {

public PersonGridModel() {

Column.For(x => x.Name);

Column.For(x => x.DateOfBirth);

RenderUsing(new ExcelGridRenderer<Person>());

}

}

Hopefully this brief introduction to GridRenderers and GridModels shows how it is possible to make your grids extendable and reusable. My next post covers working around some of the issues of the WebForms View Engine.

**MvcContrib Grid Part 4 - Limitations of the WebFormsViewEngine**

February 28, 2009 – 4:58 pm

This is part 4 of a series about the new MvcContrib grid.

* [Part 1 - Introduction](http://www.jeremyskinner.co.uk/2009/02/08/rewriting-the-mvccontrib-grid/)
* [Part 2 - New Syntax](http://www.jeremyskinner.co.uk/2009/02/22/rewriting-the-mvccontrib-grid-part-2-new-syntax/)
* [Part 3 - GridModels and GridRenderes](http://www.jeremyskinner.co.uk/2009/02/23/rewriting-the-mvccontrib-grid-part-3-gridmodels-and-gridrenderers)
* [Part 4 - Limitations of the WebFormsViewEngine](http://www.jeremyskinner.co.uk/2009/02/28/mvccontrib-grid-part-4-limitations-of-the-webformsviewengine)
* [Part 5 - The Action Syntax](http://www.jeremyskinner.co.uk/2009/03/01/mvccontrib-grid-part-5-the-action-syntax)

I was recently asked a question by one of the other MvcContrib comitters, Will Shaver, about why the **ToString** method of the Grid does not return the HTML for the grid, but instead writes the html its internal textwriter and then returns null.

This is the offending piece of code:

public override string ToString()

{

\_gridModel.Renderer.Render(\_gridModel, DataSource, \_writer, context);

return null;

}

This means that when ToString is called on the grid the GridRenderer will render the appropriate HTML to a textwriter (which is actually the HTTP Response output stream). From the end user’s perspective, this shouldn’t make a difference because it ‘appears’ that the grid is being rendered at the correct location:

**<%=** Html.Grid**(**Model.People**)**.Columns**(**...**)** **%>**

ToString is implicitly called at the end of the <%= %> block which causes the rendering to occur.

**What a silly way to create some html!**

Yes, it may seem like a strange way of doing things but there is actually a good reason for this. In order to support rendering Partials for parts of the grid it is necessary that the grid is **written directly to the response stream** due to limitations of the WebForms View Engine.

**What about passing a custom TextWriter to IView.Render?**

The ASP.NET MVC IView represents the view that will be rendered. Take a look at the signature for its Render method:

void Render(ViewContext viewContext, TextWriter writer);

You might think that it would be possible to pass a StringWriter to the Render method, capture the output of the partial view and then store it internally in the grid. Unfortunately, this isn’t possible. I’m guessing the TextWriter parameter here is a hook for a future extensibility point - the current implementation of WebFormView **completely ignores the parameter.** In fact, internally WebFormView has to make use of HttpContext.Current in order to work around the limitation of .aspx/web forms.

**What about the BlockRenderer?**

MvcContrib contains a class called the [BlockRenderer](http://mvccontrib.googlecode.com/svn/trunk/src/MVCContrib/UI/BlockRenderer.cs) that can be used to capture the outputs of a view by making use of a Response Filter. While I could have used this approach, it suffers from several major flaws. In order for this approach to work, it is necessary to Flush the response stream before inserting the CapturingResponseFilter. This causes problems if you’re trying to do anything funky with HTTP Headers, or even something as simple as trying to set the Content Type. Check out [this post on the MVC Forums](http://forums.asp.net/p/1376598/2896402.aspx) for more details. There are other hacks that can be used, but none of them are ideal

In the end I decided the least evil approach was to render to Response.Output from within ToString as it causes the fewest problems for the end user. Hopefully with the release of ASP.NET 4.0 the MVC Team will be allowed to make some changes to the underlying WebForms engine in order to make it possible to capture view output.

**MvcContrib Grid Part 5 - The Action Syntax**

March 1, 2009 – 2:33 pm

This is part 5 of a series about the new MvcContrib grid.

* [Part 1 - Introduction](http://www.jeremyskinner.co.uk/2009/02/08/rewriting-the-mvccontrib-grid/)
* [Part 2 - New Syntax](http://www.jeremyskinner.co.uk/2009/02/22/rewriting-the-mvccontrib-grid-part-2-new-syntax/)
* [Part 3 - GridModels and GridRenderes](http://www.jeremyskinner.co.uk/2009/02/23/rewriting-the-mvccontrib-grid-part-3-gridmodels-and-gridrenderers)
* [Part 4 - Limitations of the WebFormsViewEngine](http://www.jeremyskinner.co.uk/2009/02/28/mvccontrib-grid-part-4-limitations-of-the-webformsviewengine)
* [Part 5 - The Action Syntax](http://www.jeremyskinner.co.uk/2009/03/01/mvccontrib-grid-part-5-the-action-syntax)

The rewritten MvcContrib grid now has an alternative syntax thanks to the work of [Will Shaver](http://www.primedigit.com/). This syntax is called the ‘Action’ syntax as it makes heavy use of the System.Action delegate.

**Enabling the ActionSyntax**

By default, the methods associated with the Action Syntax will not show up as they are extension methods that live in a different namespace to the rest of the grid. Out of the box only the default syntax will work. To enable the Action Syntax you’ll need to either import the **MvcContrib.UI.Grid.ActionSyntax** namespace either in your .aspx view…

**<%@** Import Namespace**=**"MvcContrib.UI.Grid.ActionSyntax" **%>**

…or in your web.config’s namespace import section:

**<system.web>**

**<pages>**

**<namespaces>**

**<add** namespace="MvcContrib.UI.Grid.ActionSyntax" **/>**

**</namespaces>**

**</pages>**

**</system.web>**

**Comparing the two syntaxes**

There are several differences between the two syntaxes. Here is a grid definition using the default syntax:

**<%=** Html.Grid**(**Model**)**

.Columns**(**column **=>** **{**

column.**For(**x **=>** x.Id**)**.Named**(**"Person ID"**);**

column.**For(**x **=>** x.Name**);**

column.**For(**x **=>** x.Gender**);**

column.**For(**x **=>** x.DateOfBirth**)**.Format**(**"{0:d}"**);**

column.**For(**"View Person"**)**.Named**(**""**)**.Partial**(**"ViewPersonPartial"**);** //Example of using a Partial view for complex cells

**})**.RowStart**(**row **=>** **string**.Format**(**"<tr{0}>", row.IsAlternate **?** "style=\"background-color**:**#CCDDCC\"" **:** ""**))**

**%>**

Here we’re specifying that the **View Person** column is a custom column that should be rendered using a Partial View, and the start of every alternate row has a background colour\*\*.

Here is the same grid defined using the ActionSyntax:

**<%** Html.Grid**(**Model**)**.Columns**(**column **=>** **{**

column.**For(**x **=>** x.Id**)**.Named**(**"Person ID"**);**

column.**For(**x **=>** x.Name**);**

column.**For(**x **=>** x.Gender**);**

column.**For(**x **=>** x.DateOfBirth**);**

column.**For(**"View Person"**)**.Named**(**""**)**.Action**(**p **=>** **{** **%>**

<td style="font-weight:bold">

**<%=** Html.ActionLink**(**"View Person", "Show", **new** **{** id **=** p.Id **})%>**

</td>

**<%** **});**

**})**.RowStart**((**p,row**)** **=>** **{**

**if** **(**row.IsAlternate**)** **{** **%>**

<tr style="background-color:#CCDDCC">

**<%** **}** **else** **{** **%>**

<tr>

**<%** **}**

**})**.Render**();** **%>**

Note the following differences:

* The ActionSyntax requires the use of <% %> tags rather than <%= %> tags.
* The ActionSyntax requires that you end your grid definition with a call to the Render method or **the grid will not show up if you do not call Render.**
* You must terminate the call to Render with a semicolon
* You can directly embed HTML in the grid definition rather than using strings or partial views

Both syntaxes have their advantages and disadvantages but the end result is the same. You’re free to use whichever one you prefer.

\*\*Both the partial view/actionsyntax approach are designed only for complex column definitions. This example was quite contrived and you could achieve the same result with a far simpler approach:

<style type="text/css">

.gridrow\_alternate { background-color:#CCDDCC; }

</style>

**<%=** Html.Grid**(**Model**)**

.Columns**(**column **=>** **{**

column.**For(**x **=>** x.Id**)**.Named**(**"Person ID"**);**

column.**For(**x **=>** x.Name**);**

column.**For(**x **=>** x.Gender**);**

column.**For(**x **=>** x.DateOfBirth**)**.Format**(**"{0:d}"**);**

column.**For(**x **=>** Html.ActionLink**(**"View Person", "Show", **new** **{** id **=** p.Id **}))**

.Attributes**(**x **=>** **new** Hash**(**style **=>** "font-weight:bold"**));**

**})** **%>**

Credits:

[Jeremy Skinner](http://www.jeremyskinner.co.uk/author/jskinner/) http://www.jeremyskinner.co.uk