5

Consuming 3rd party components

This chapter covers

* Learning the basic MvcContrib Grid
* Advanced MvcContrib Grid
* Portable Area Summary
* Consuming a Portable Area

Summary

The ASP.Net MVC Framework provides a lot of control but that comes at a cost out of the box. The html helpers are basic and provide very simple user interface elements. This leaves you with the choice to hand craft nice user interfaces using html and css. While this is a great option for an experienced web designer, most developers find relying on a 3rd party component much more productive. This allows you to develop your application rather than spend lots of time on user interface infrastructure. This chapter will demonstrate two 3rd party components that will show you different styles of integrating with the MVC framework. The first is a html helper based integration that can render a html table, this is the Grid component available from the open source project MvcContrib. The second style of component uses the portable area style component. The portable area style component builds on top of the built in Area functionality of MVC 2, with some functionality to ease the deployment of the component into your application. First let's look at the MvcContrib Grid.

5.1 The MvcContrib Grid

The MvcContrib Grid is a user interface component which creates a well formed html table. It uses a fluent interface, which allows you to define the configuration of the grid with a strongly typed and refactorable syntax. The refactoring support makes this style of component work very nicely with refactoring tools like JetBrains Resharper and DevExpress Refactor Pro. This type of component generally requires a strongly typed view, which is used to drive the api of the grid. The scenario where you would want to use a grid like this would be to display a list of model objects. First let's look at the Action method which would send a list or IEnumerable<People> to a view for rendering.

Listing 5.1 The Action to render a list of People objects

public ActionResult AutoColumns() { |A

return View(\_peopleFactory.CreatePeople()); |A

}

A - This action returns a IEnumerable<People> to the view.

Listing 5.1 shows the Action that will send a IEnumerable model to the view for rendering. This is a fairly common scenario. This example ignore more advanced features like grid paging. It would simply send every People object in the application to the view for rendering. The next step is to use the MvcContrib Grid to see how easily we can get a table formated view of our People objects.

Listing 5.2 Rendering a table using the Grid AutoGenerateColumns

<%= Html.Grid(Model).AutoGenerateColumns() %> | A

A - This view code renders a well formed html table based on the model.

The AutoGenerateColumns can give you a lot of productivity, if you follow some conventions. First, as shown in Figure 5.1, when you send a general Model to the view and AutoGenerateColumns, you will see that there are some columns which the Grid does not know how to render values for properly. When the Grid finds complex objects, it does not make and assumption about which property is should use to represent that property, instead it does not render a value to the table cell. The best way to use the Grid with the AutoGenerateColumns is to use a dedicated Presentation Model for this type of listing page. In this case you only send a model with the exact properties that you want to render as cells in your table. Using the AutoGeneratedColumns method with a dedicated Presentation Model will provide the most develop productivity.

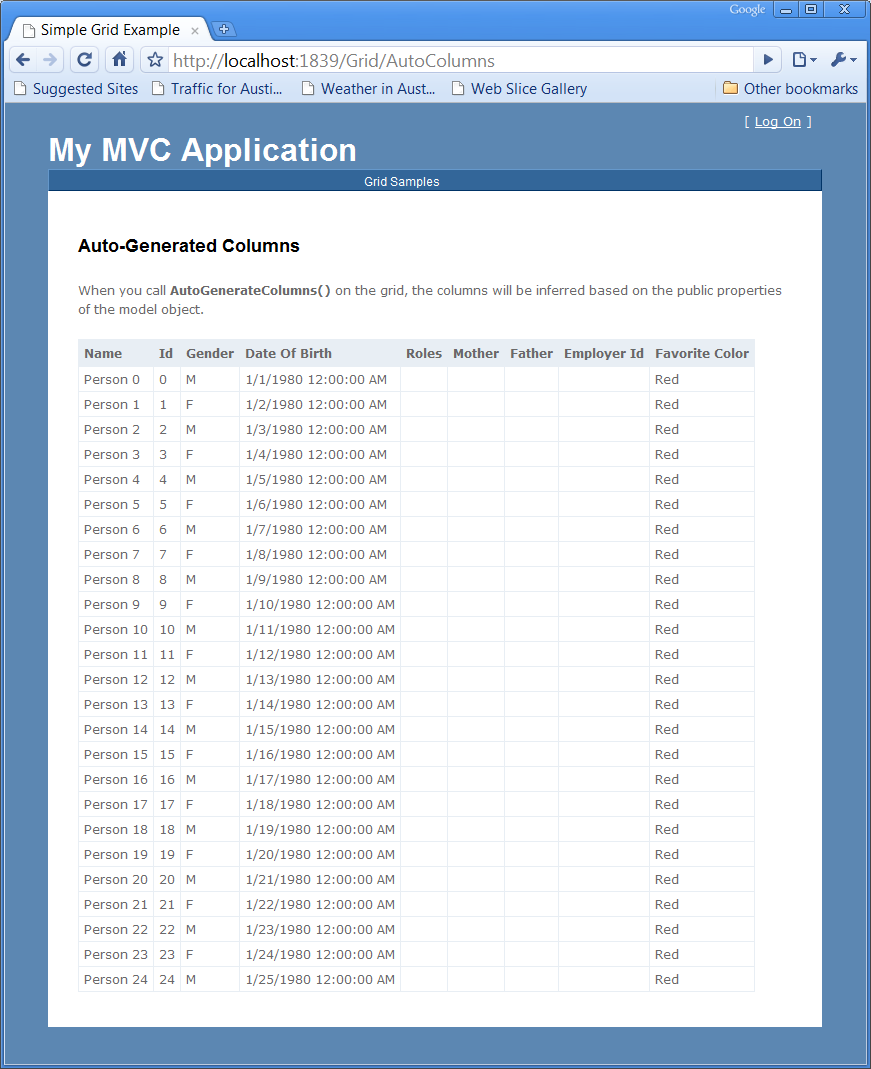


Figure 5.1 - The view of a Grid.AutoGenerateColumns

5.2 The MvcContrib Grid advanced usage.

While the previous example of the MvcContrib Grid seemed to just work magically with a single line of view code, it has some pretty strong oppinions about how it is going to render a model. If you do not like the defaults of how the Grid works you do have some more options, this is where the power of the Grid really comes into play. Using the same model as the previous example we will not show how you can configure the Grid on the view to finely control more aspects of how it is rendered.

Listing 5.3 The MvcContrib Grid with more control

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

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

column.For(x => x.Name); |C

column.For(x => x.Gender); |C

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

column.For(x => Html.ActionLink("View Person", "Show", |E

new { id = x.Id})).DoNotEncode(); |E

}) %>

A - The Grid is created and the Columns method is used to customize the columns.

B - The Id property is rendered with the title Person ID

C - These properties are rendered with the default conventions

D - The DateOfBirth column is formatted with a format string

E - A custom column is created and renders a hyperlink to display the Person object using the current controllers Show Action method.

The MvcContrib Grid created using the view code from Listing 5.3 will render nicely in a table and does not require a dedicated Presentation Model. The html of the Grid is displayed in Figure 5.2.

Eliminating the need for a Presentation Model is not the main reason to specify the columns for the Grid. The main reason to specify the columns for the Grid is to modify the values using formatting string or to add additional columns to the table. The syntax for defining the Grid may look very odd at first, it uses some of the newer features of the C# language, which were put in to support the LINQ technology. The parameters that use Lambda Expressions, the syntax x => x.Id may seem like a strange way to specify how the columns to render in the table but this is the way that developers who are using strongly typed views are going. By using this syntax, when you change the name of a property using a refactoring tool, the property gets changed in your view code. This eliminates the runtime errors that you would see when using magic strings and late binding to configure how to pull property values out of your model and render them into a table. While the MvcContrib was one of the first components to use this method of configuration this style has caught on. The Grid was created and is currently maintained by Jeremy Skinner, a committer to the MvcContrib project. His work on the Grid has influenced many of the newer APIs that are new to the MVC 2 view helpers. We see the Lambda Expression style configuration of html helpers being released now and the MvcContrib Grid was one of the first components to really prove that there are some advantages to using this in a strongly typed view. For more information about the Grid, go to the MvcContrib project at http://www.MvcContrib.org and see more information and blogs from the creator of the Grid at http://www.jeremyskinner.co.uk . There are a large number of features to the Grid that we cannot cover in this chapter. The MvcContrib project has a number of samples which walk through the various ways to get some increadibly power out of this component.

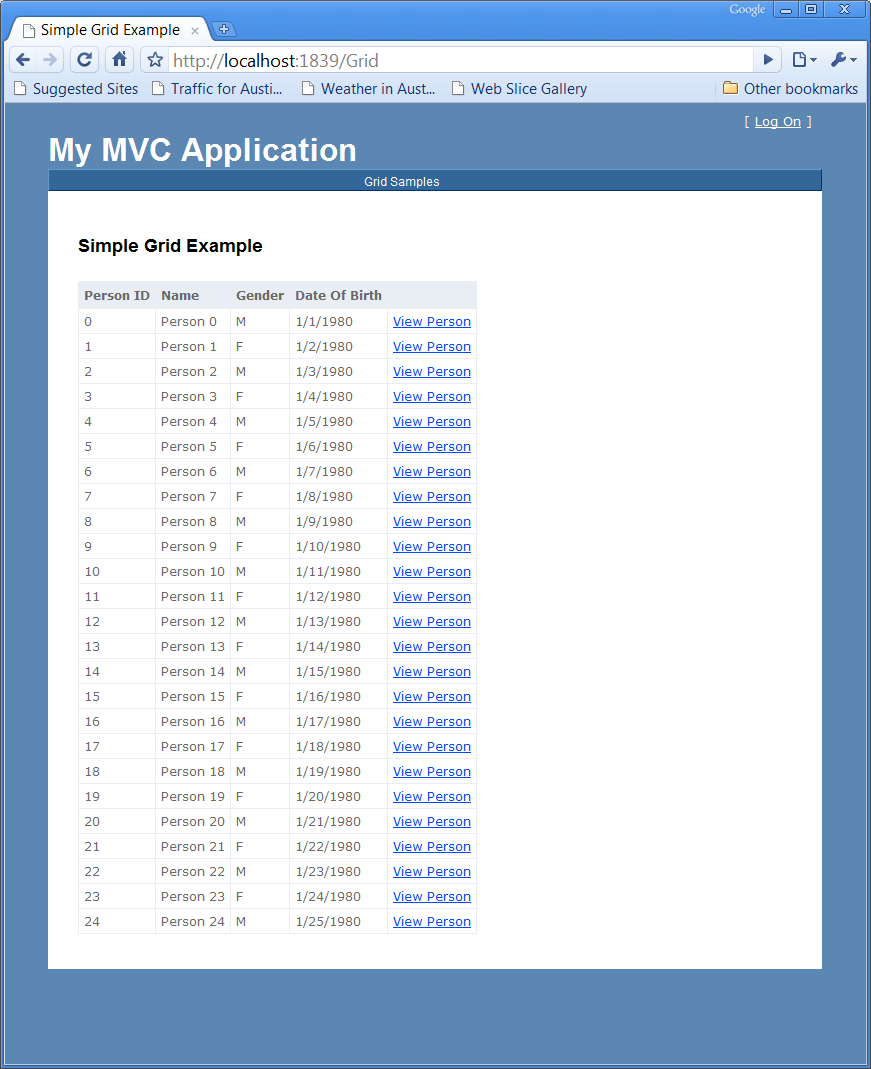


Figure 5.2 The MvcContrib Grid rendered using Column configuration

5.3 Understanding the Portable Area

The Portable Area is a concept that comes from the MvcContrib project. As the name describes it, it is a native MVC 2 Area packaged up in a way that is easier to distribute and consume than an Area built with the out of the box MVC 2 support. That is a pretty broad statement so let's first look at what is in an Area and then cover which pieces may need to be made portable. Areas are simply put a subset of an MVC application, that is separated in a way that gives it some physical distance from other groups of functionality in an MVC application. This means that an Area will have one or more Routes, Controllers, Actions, Views, Partial Views, Master Pages and content files, such as CSS, Javascript , and image files. These are all the pieces which may be used in an Area. Of those individual elements many of them are not part of the binary distribution of a MVC application. Only the Routes, Controllers, and Actions get compiled into an assembly. The rest of the elements are individual files which need to be copied and managed with the other assets which are part of your application. This is reasonably trivial to manage if you build an Area for your application and just use it as a way of managing smaller modules of your application. But if you want to use an Area as a way for packaging up and sharing/distributing a piece of multi-page user interface functionality, managing all of the individual files make this option a bad choice when integrating someone else's component with your application. This is where the MvcContrib project developed the idea of a Portable Area. By building on top of the existing Area functionality, it only takes some minor changes to your Area project to make it portable. The process of making an Area portable, is pretty trivial. As an Area developer, instead of leaving the file assets as content items in your project, you make them embedded resources. An embedded resource is a content file which is compiled into the assembly of a project. The file still exists and it can be programmatically extracted from the assembly at runtime. This means that a Portable Area only contains a single file, the assembly of the project, rather than all the individual content files.