5

Consuming 3rd party components

This chapter covers

* Learning the basic MvcContrib Grid
* Advanced MvcContrib Grid techniques

The ASP.NET MVC Framework provides a lot of control out of the box but that comes at a cost. 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 the Grid component available from the open source MvcContrib project that can be used to render an HTML table.. The second component is a portable area which builds on top of the areas functionality in MVC 2 with some additional features that ease the deployment of components within 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 refactoring-friendly 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<Person> to a view for rendering.

Listing 5.1 The Action to render a list of Person objects

public ActionResult AutoColumns() { |#A

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

}

#A Passes IEnumerable<Person> 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 ignores more advanced features like paging. It would simply send every Person 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 formatted view of our Person objects.

Listing 5.2 Rendering a table using the Grid AutoGenerateColumns

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

#A Renders a table from the model.

The AutoGenerateColumns method will automatically generate columns in the table based on the public properties of the Person object, as shown in Figure 5.1. However, this is only useful in certain situations. You will see from Figure 5.1 that there are some columns for which the grid does not know how to render a value. The default behavior is to call ToString on each property value. However, this is not particularly useful for complex types as this just displays the type name, as illustrated in Figure 5.1. Because of this, AutoGenerateColumns is most useful if you're using a dedicated Presentation Model rather than a nested object hierarchy.

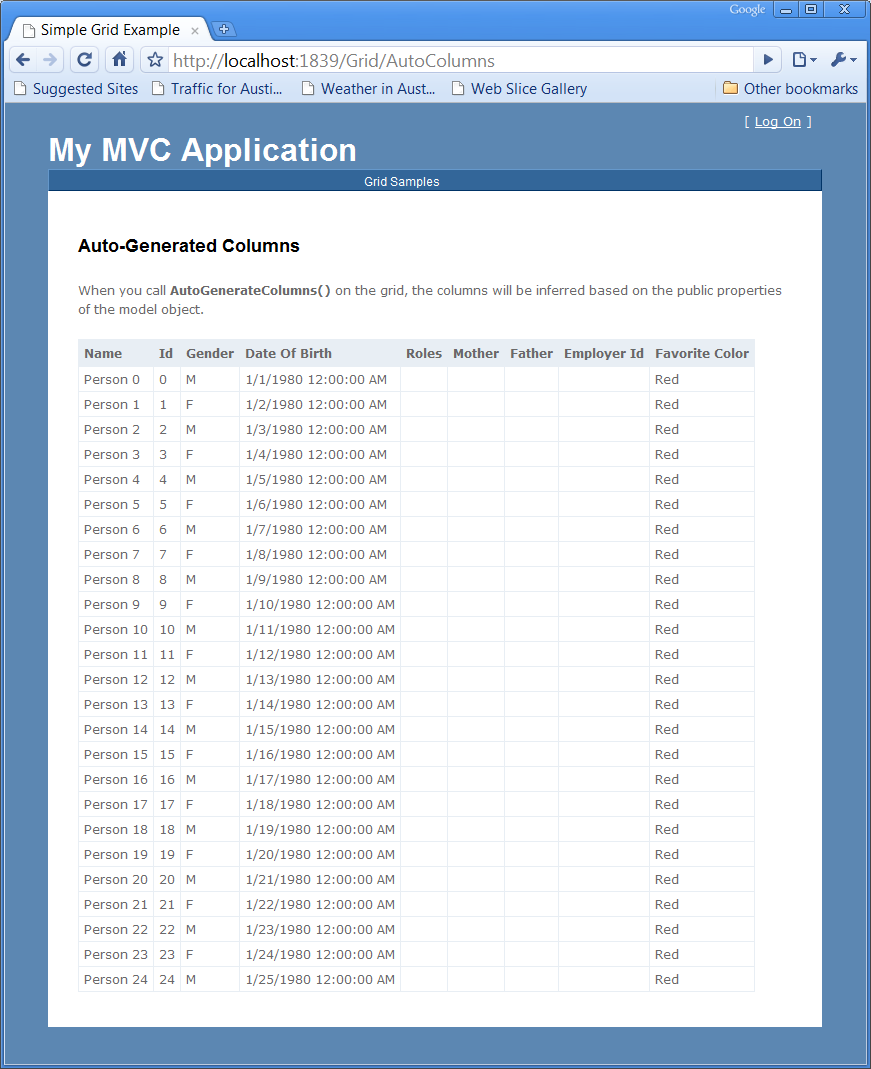


Figure 5.1 - The view of a Grid.AutoGenerateColumns

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 opinions about how it is going to render a model. For example, it assumes that all public properties should be rendered as columns (unless they are decorated with the ScaffoldColumn attribute). If you do not like this behavior then you do have some more options—this is where the power of the Grid really comes into play. Listing 5.3 shows how you can use the grid to customize the output for individual columns.

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 Specifies individual columns

#B Overrides column name

#C Rendered using default conventions

#D Specifies custom output format

#E Defines a custom column

The MvcContrib Grid created using the view code from Listing 5.3 will render nicely in a table as shown in Figure 5.2.

The main reason to explicitly specify the columns for the grid is to be able to customize the output of various columns (for example, by using a custom string format or to add additional columns to the table). The syntax for defining the grid may look odd at first—it uses some of the newer features of the C# language. For example, lambda expressions are used to specify which properties should be rendered as columns in the table. 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 on the MvcContrib project. For more information about the Grid, go to the MvcContrib project at http://www.MvcContrib.org and see more information and blog posts from the creator of the grid at http://www.jeremyskinner.co.uk. There are a large number of additional features built in to the grid that we cannot cover in this chapter. The MvcContrib project has a number of samples which walk through the extensive options for using the Grid.

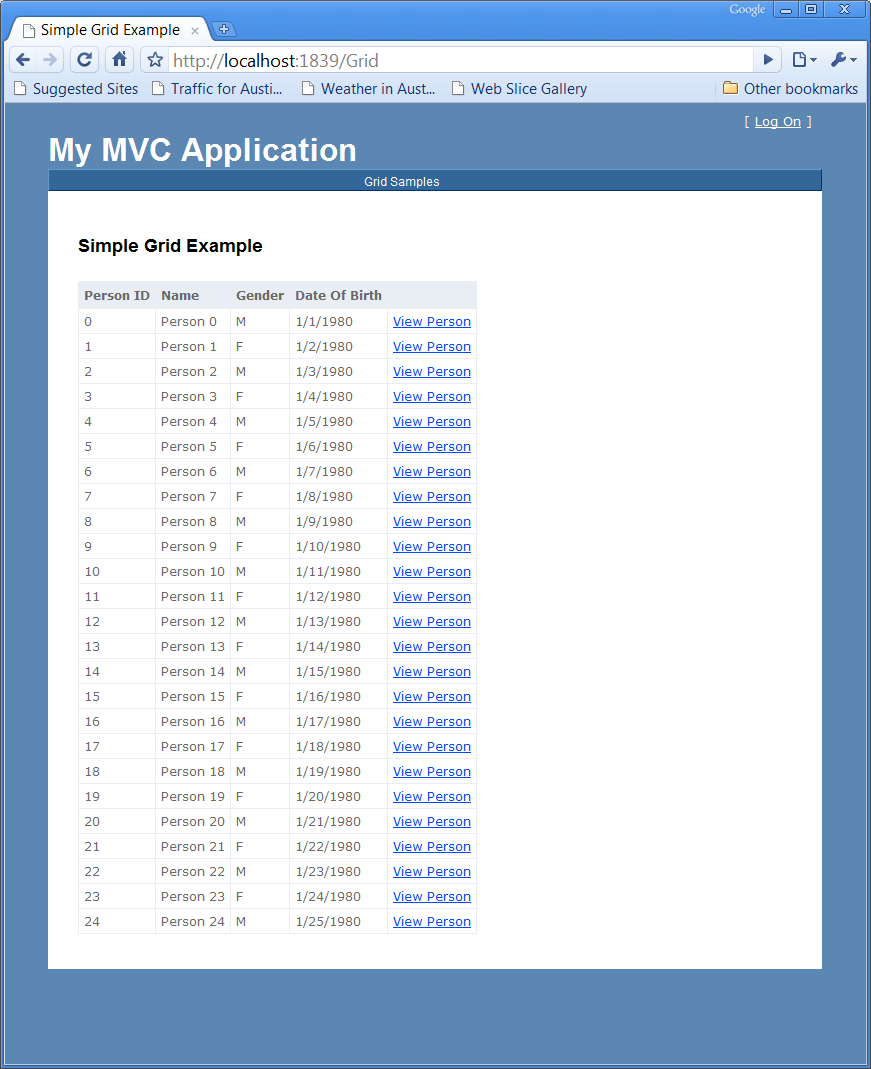


Figure 5.2 The MvcContrib Grid rendered using Column configuration

5.5 Summary

This chapter covered using 3rd party components in an MVC application. We covered using a page level component, the MvcContrib Grid. We walked through using the Grid's AutoGenerateColumns feature for simple cases. We also demonstrated a more advanced usage of the Grid, using its powerful strongly typed API. We defined a portable area as a component which takes the MVC 2 concept of an area and packages that area into a single DLL for easy portability. We then walked through a small demonstration of how to add and use a portable area from within a MVC 2 web application project. These two different types of components show that there are different ways to look at how much functionality a component could provide. The Grid provides a single control like experience, while the portable area can provide a multi-page component. By using these components you can provide more functionality quickly. The next chapter will cover hosting a MVC application in Internet Information Server.