[MSDN Blogs](http://blogs.msdn.com/) > [Vincent Sibal's Blog](http://blogs.msdn.com/b/vinsibal/)> [WPF DataGrid: Dissecting the Visual Layout](http://blogs.msdn.com/b/vinsibal/archive/2008/08/14/wpf-datagrid-dissecting-the-visual-layout.aspx)

WPF DataGrid: Dissecting the Visual Layout

[Vincent Sibal - MSFT](http://blogs.msdn.com/136277/ProfileUrlRedirect.ashx)

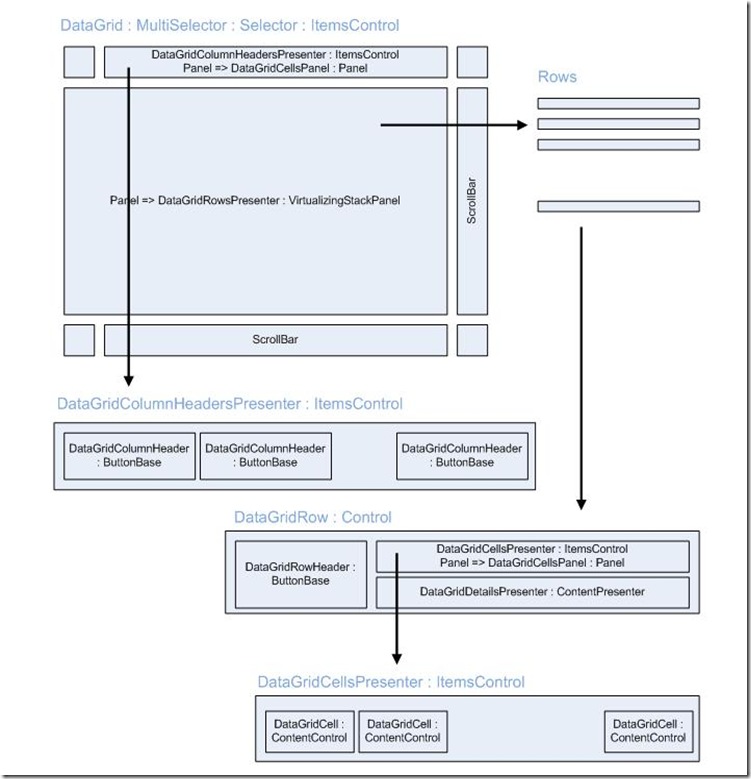
**14 Aug 2008 10:39 AM**

I’m going to be dissecting and discussing the DataGrid visuals and how they are all assembled together to form the overall DataGrid. Note: This is really more of a post for people who want to understand how the DataGrid works internally. I’m not going to be going over how to use some of the APIs in this post.

If you haven’t already, get the binaries and source for it [here](http://www.codeplex.com/wpf/Release/ProjectReleases.aspx?ReleaseId=15598).  For more related material take a look at these posts:

* [DataGrid Intro](http://blogs.msdn.com/vinsibal/archive/2008/08/11/net-3-5-sp1-and-wpf-datagrid-ctp-is-out-now.aspx)
* [Stock and Template Columns](http://blogs.msdn.com/vinsibal/archive/2008/08/19/wpf-datagrid-stock-and-template-columns.aspx)
* [Working with DataGridComboBoxColumn (Part1)](http://blogs.msdn.com/vinsibal/archive/2008/08/26/wpf-datagrid-working-with-datagridcomboboxcolumn.aspx)
* [Working with DataGridComboBoxColumn (Part2)](http://blogs.msdn.com/vinsibal/archive/2008/08/28/wpf-datagrid-working-with-datagridcomboboxcolumns-part-2.aspx)

Understanding the visual tree isn’t the most important thing to know about basic DataGrid usage, but it is very helpful for such things like customizing its appearance through the different styling templates available, replacing the visual tree altogether, accessing visual properties, understanding routing events, or when you want to learn how the data maps to the visual tree. Here is the big picture view of the DataGrid:

[](http://blogs.msdn.com/blogfiles/vinsibal/WindowsLiveWriter/WPFDataGridDissectingtheVisualLayout_6B39/DataGridVisual_2.jpg)

You can view the equivalent in Generic.xaml in the source for DataGrid. The DataGrid, like a generic spreadsheet, is made of rows and cells. The rows are generated from the DataGrid’s GetContainerForItemOverride. That should sound familiar as DataGrid ultimately derives from ItemsControl. If you need a brush up on that, please check out Dr. WPF’s awesome series on [ItemsControl](http://www.drwpf.com/blog/Home/tabid/36/EntryID/37/Default.aspx). The cells are generated from DataGridCellsPresenter’s GetContainerForItemOverride. The important containers are DataGridRowsPresenter and DataGridCellsPanel. DataGridRowsPresenter derives from VSP and is a really small class that does some clean up on the ItemsHost and some scrolling work. If you plan to replace this container, be sure to handle scrolling. The DataGridCellsPanel on the other hand is a very specific implementation to the default DataGrid which handles sizing of each DataGridCell. Replacing this container will need a good understanding of the DataGrid internals which I will not go into here.

In my previous [post](http://blogs.msdn.com/vinsibal/archive/2008/08/11/net-3-5-sp1-and-wpf-datagrid-ctp-is-out-now.aspx) I did a really brief intro of the DataGrid and just got it up and running. Well, now I want to break down how it actually works. Whether columns are auto-generated or built manually, the DataGrid will use those columns to map the data source to each cell. Notice that the list of DataGridColumns is not in the diagram as it is not really a visual. DataGridColumn is used as the glue between the data and the DataGridCells. So given this column implementation as an example,

<dg:DataGrid AutoGenerateColumns="False">

  <dg:DataGrid.Columns>

    <dg:DataGridTextColumn Binding="{Binding Path=FirstName}" />

    <dg:DataGridTextColumn Binding="{Binding Path=LastName}" />

    <dg:DataGridCheckBoxColumn Binding="{Binding Path=LikesCake}"/>

    <dg:DataGridHyperlinkColumn Binding="{Binding Path=Homepage}" />

  </dg:DataGrid.Columns>

</dg:DataGrid>

here is the sequence that follows in creating the DataGrid when the ItemsSource is set (assuming all the Path values are properties on the data source),

1. When ItemsSource is set on the DataGrid, PrepareContainerForItemOverride is called on each item of the data source collection which prepares a DataGridRow for each.

2. When a DataGridRow is prepared, it passes the item to its DataGridCellsPresenter which internally creates a copied collection of that item and sets its ItemsSource to that collection.

3. When DataGridCellsPresenter.ItemsSource is set, PrepareContainerForItemOverride is called on each copied item which prepares a DataGridCell for each.

4. When a DataGridCell is prepared (which is a ContentControl), it gets its corresponding column from the DataGrid and asks that column to generate the visual tree for DataGridCell’s Content property.

It is in the generation of the visual tree where the binding is hooked up from the data source to the UI. Recall in the column implementation, the Binding property was set to a property on the data source. In the visual tree generation it takes that Binding and applies it to the generated UIElement (which in step 4 is set to DataGridCell’s Content property).  That is basically it. You should also know that different column types will generate different UIElements. DataGridTextColumn will generate a textbox; DataGridCheckBoxColumn will generate a checkbox, etc. So hopefully some of the magic has been revealed on how the DataGrid populated itself.

In addition to understanding the mapping of the data to the UI, you should have a good grasp of the visuals involved from looking at the diagram. This helps when you want to style the DataGrid. Convenience properties for styling have also been added the DataGrid class itself so you can look there for the available styles as well, but let’s stop here for now as I will get more into styling and templating in an upcoming post.

[MSDN Blogs](http://blogs.msdn.com/) > [Vincent Sibal's Blog](http://blogs.msdn.com/b/vinsibal/)> [WPF DataGrid: Working with DataGridComboBoxColumn CTP](http://blogs.msdn.com/b/vinsibal/archive/2008/08/26/wpf-datagrid-working-with-datagridcomboboxcolumn.aspx)

WPF DataGrid: Working with DataGridComboBoxColumn CTP (Part 1)

[Vincent Sibal - MSFT](http://blogs.msdn.com/136277/ProfileUrlRedirect.ashx)

**26 Aug 2008 4:45 PM**

**UPDATE: DataGridComboBoxColumn has been updated from CTP to V1.  See the post** [here](http://blogs.msdn.com/vinsibal/archive/2008/10/31/wpf-datagrid-datagridcomboboxcolumn-v1-intro.aspx)**for the updates to the DataGridComboBoxColumn as well as an updated sample.**

If you haven’t already, you can download the binaries and source for the DataGrid v1 [here](http://www.codeplex.com/wpf/Release/ProjectReleases.aspx?ReleaseId=15598).

DataGridComboBoxColumn is a column that deserves some special attention.  What makes it a little unique is how it hooks up to a source list of items and how the current SelectedItem maps back to the data source.  Recall from my previous [post](http://blogs.msdn.com/vinsibal/archive/2008/08/19/wpf-datagrid-stock-and-template-columns.aspx) on DataGridColumns, you will generally set the DataFieldBinding to map a property of the data source to the cells of a column,

<dg:DataGridTextColumn DataFieldBinding="{Binding Path=FirstName}" />

Also recall that each column type has a separate UIElement when in an editing and non-editing state.  For a DataGridTextColumn, it generates a TextBlock for non-editing and TextBox for editing.  When each is generated the DataFieldBinding is mapped to the UIElement.  No problems.  For a DataGridComboBoxColumn, the default template for a non-editing state is a TextBlock and the default template for an editing state is a ComboBox. Mapping the DataFieldBinding to the TextBlock works but will not work so easily with the ComboBox as it’s an ItemsControl that is defined by a list and not a single object.  With that in mind, some special APIs were introduced to DataGridComboBoxColumn.

**DataGridComboBoxColumn APIs**

These are the two public APIs specific to DataGridComboBoxColumn:

public class DataGridComboBoxColumn : DataGridBoundColumn

{  
  public static readonly DependencyProperty DataFieldTargetProperty;

  public static readonly DependencyProperty ItemsSourceProperty;

  public ComboBoxDataFieldTarget DataFieldTarget { get; set; }

  public IEnumerable ItemsSource { get; set; }  
}

ItemsSource, which is pretty straightforward, is the ItemsSource set on the generated ComboBox for an editing state.  DataFieldTarget does two things.  First, it represents the selection value from the ComboBox.  This value can be the ComboBox.SelectedItem, ComboBox.SelectedValue, or ComboBox.Text.  Second, it is the mapping between the DataFieldBinding and the ComboBox.  It is in this second part that allows you to update the ComboBox’s SelectedItem and changes will reflect back to the DataGrid’s data source.

**Example**

Using the [Northwind](http://www.microsoft.com/downloads/details.aspx?FamilyID=06616212-0356-46A0-8DA2-EEBC53A68034&displaylang=en) Database as an example (I’ve also included instructions in the sample), let’s say I want to display an editable Orders table and for the CustomerID foreign key column, I want to display a drop down list of possible choices.  The choices will be all the CustomerID values from the Customer table.  For the implementation, I’m going to auto-generate the columns so I can also show an example of how to customize columns when they are auto-generated.

Here is how I am populating the DataGrid with the Orders table:

private void btn\_GetOrders\_Click(object sender, RoutedEventArgs e)

{  
  \_orderDataSet = DBAccess.GetOrders();

  if (\_orderDataSet != null)

  {  
    \_orderDateTable = \_orderDataSet.Tables["Orders"];

    DataGrid\_Standard.ItemsSource = \_orderDateTable.DefaultView;  
  }  
}

I have a separate utility class to retrieve the DataSet which I call DBAccess.  Then I get the DataTable and set the default view to the DataGrid.  This will trigger the DataGridColumns to be auto-generated.  For auto-generated columns you have access to two events, AutoGeneratingColumn and AutoGeneratedColumns, where you can customize the behavior of the columns.  In my case I want to create a DataGridComboBoxColumn for the CustomerID field of the Orders table.  Here is a possible implementation:

private void DataGrid\_Standard\_AutoGeneratingColumn(object sender,DataGridAutoGeneratingColumnEventArgs e)

{  
  if (e.PropertyName == "CustomerID")

  {  
    DataGridComboBoxColumn column = new DataGridComboBoxColumn();

    column.DataFieldBinding = new Binding("CustomerID");

    column.DataFieldTarget = ComboBoxDataFieldTarget.SelectedValue;

    column.ItemsSource =DBAccess.GetCustomers().Tables["Customers"].DefaultView;

    column.EditingElementStyle =  (Style)this.RootGrid.FindResource("CustomerFKStyle");

    e.Column = column;  
  }  
}

<Style x:Key="CustomerFKStyle" TargetType="ComboBox">

  <Setter Property="SelectedValuePath" Value="CustomerID" />

  <Setter Property="ItemTemplate">

    <Setter.Value>

      <DataTemplate>

        <TextBlock Text="{Binding Path=CustomerID}" />

      </DataTemplate>

    </Setter.Value>

  </Setter>

</Style>

I set the DataFieldBinding to the CustomerID as that is the value that I want to represent in each cell of that column.  I set the column.ItemsSource to the Customers table.   Each item in the ComboBox.ItemsSource represents a record in the Customers table but I really just want to show is its CustomerID.  I can do this by setting its SelectedValuePath and ItemTemplate to CustomerID which I do in the CustomerFKStyle.  Remember that a DataGridComboBoxColumn generates a ComboBox only in the editing state so I have to set this style on column.EditingElementStyle.  Lastly, I set column.DataFieldTarget to SelectedValue as this is value that will update the CustomerID property of my Orders table.

You can check out the full sample [here](http://blogs.msdn.com/vinsibal/attachment/8898333.ashx).  Note: I didn’t write the code to persist changes back to the database.

Also check out, [WPF DataGrid: Working with DataGridComboBoxColumn (Part 2)](http://blogs.msdn.com/vinsibal/archive/2008/08/28/wpf-datagrid-working-with-datagridcomboboxcolumns-part-2.aspx) for more info on DataGridComboBoxColumns.

**More Related Material:**

* [DataGrid Intro](http://blogs.msdn.com/vinsibal/archive/2008/08/11/net-3-5-sp1-and-wpf-datagrid-ctp-is-out-now.aspx)
* [Dissecting the Visual Layout](http://blogs.msdn.com/vinsibal/archive/2008/08/14/wpf-datagrid-dissecting-the-visual-layout.aspx)

**Other Samples:**

* [ScrollViewer with ToolTip](http://blogs.msdn.com/vinsibal/archive/2008/08/25/wpf-datagrid-sample-add-a-preview-tooltip-to-a-scrollviewer.aspx)
* [Custom sorting, column selection, single-click editing](http://blogs.msdn.com/vinsibal/archive/2008/08/27/more-datagrid-samples-custom-sorting-drag-and-drop-of-rows-column-selection-and-single-click-editing.aspx)
* [Tri-state sorting](http://blogs.msdn.com/vinsibal/archive/2008/08/29/wpf-datagrid-tri-state-sorting-sample.aspx)

[MSDN Blogs](http://blogs.msdn.com/) > [Vincent Sibal's Blog](http://blogs.msdn.com/b/vinsibal/)> [WPF DataGrid: Working with DataGridComboBoxColumns CTP (Part 2)](http://blogs.msdn.com/b/vinsibal/archive/2008/08/28/wpf-datagrid-working-with-datagridcomboboxcolumns-part-2.aspx)

WPF DataGrid: Working with DataGridComboBoxColumns CTP (Part 2)

[Vincent Sibal - MSFT](http://blogs.msdn.com/136277/ProfileUrlRedirect.ashx)

**28 Aug 2008 9:15 AM**

**UPDATE: DataGridComboBoxColumn has been updated from CTP to V1.  See the post**[here](http://blogs.msdn.com/vinsibal/archive/2008/10/31/wpf-datagrid-datagridcomboboxcolumn-v1-intro.aspx)**for the updates to the DataGridComboBoxColumn as well as an updated sample.**

Previously I started a [post](http://blogs.msdn.com/vinsibal/archive/2008/08/26/wpf-datagrid-working-with-datagridcomboboxcolumn.aspx) on DataGridComboBoxColumns where I introduced the APIs specific to it and showed an example of how to use it.  Well, one thing I didn't show that seems to be a common ask is how to mask what is displayed versus what is actually being updated.  Let’s say I have a CustomerID that I want to display as before with a DataGridComboBoxColumn but I want the display values to be a little more user friendly.  For the ComboBox (editing state) I can use DisplayMemberBinding for the display value and SelectedValuePath for the actual value but for the TextBlock (non-editing state) I don't have anything that can map the CustomerID to something else.  Let’s take a look at a couple solutions.

One solution could be using a converter on the DataFieldBinding to transform the CustomerID to some other alias.  Remember though in our example, the overall table (Orders) that the DataGrid is displaying is different that the table that the ComboBox is using (Customers).  So in your converter you may need to optimize the calls to get the Customer table.  I have excluded that from the example as that is a separate topic of discussion.

<dg:DataGridComboBoxColumn

        DataFieldBinding="{Binding CustomerID,

                                   Converter={StaticResourceCustomerConverter},

                                  ConverterParameter=ContactName}"

        ItemsSource="{Binding Source={StaticResourcecustomerDataProvider}}"

        Header="CustomerID (ContactName alias)"

        DataFieldTarget="Text">

  <dg:DataGridComboBoxColumn.EditingElementStyle>

    <Style TargetType="ComboBox">

      <Setter Property="SelectedValuePath" Value="CustomerID" />

      <Setter Property="DisplayMemberPath" Value="ContactName" />

    </Style>

  </dg:DataGridComboBoxColumn.EditingElementStyle>

</dg:DataGridComboBoxColumn>

public object Convert(object value, Type targetType, object parameter,CultureInfo culture)

{

  string customerID = (string)value;

  DataTable dataTable = DBAccess.GetCustomers().Tables["Customers"];

  foreach (DataRow row in dataTable.Rows)

  {

    if (row["CustomerID"].ToString() == customerID)

      return row[parameter.ToString()];

  }

  return null;

}

The converter is returning a string value and remember that the DataFieldTarget is what maps the ComboBox value back to the DataFieldBinding, so I have set the DataFieldTarget to “Text”.  Also, as the DisplayMemberPath and ConverterParameter show, I am using the ContactName as the alias.

Another solution which is very similar to the first one is to use the converter as I did above but with a DataGridTemplateColumn.  There really isn’t a big advantage going with a DataGridTemplateColumn over a DataGridComboBoxColumn in this particular example, but it does give you more flexibility on the type of element to display in the non-editing state.  Anyway, here is an example:

<dg:DataGridTemplateColumn Header="CustomerID (ContactName alias)">

  <dg:DataGridTemplateColumn.CellTemplate>

    <DataTemplate>

      <TextBlock Text="{Binding Path=CustomerID, Mode=OneWay, Converter={StaticResource CustomerConverter}, ConverterParameter=ContactName}" />

    </DataTemplate>

  </dg:DataGridTemplateColumn.CellTemplate>

  <dg:DataGridTemplateColumn.CellEditingTemplate>

    <DataTemplate>

      <ComboBox ItemsSource="{Binding Source={StaticResourcecustomerDataProvider}}"

        SelectedValue="{Binding CustomerID}"SelectedValuePath="CustomerID" DisplayMemberPath="ContactName" />

    </DataTemplate>

  </dg:DataGridTemplateColumn.CellEditingTemplate>

</dg:DataGridTemplateColumn>

One thing to note about this example is what is bound to what.  In CellTemplate, the TextBlock is using DataGrid’s DataContext and is bound to the CustoemrID field of the Orders table.  In CellEditingTemplate, the ComboBox is using the Customer table as the Source for the ItemsSource but the DataContext is still the DataGrid for everything elese.  That means that SelectedValue’s binding to CustomerID is based on the CustomerID of the Orders table and that binding is the one that will update the source.

A third and a little more time consuming approach would be to subclass DataGridComboBoxColumn and add a property for the display.  If you take a look at the DataGridHyperlinkColumn it actually has a special API for something like this which is ContentBinding.  ContentBinding maps the content it is bound to what gets displayed as the link in the cells of that column.  However, there is still a DataFieldBinding which maps to the actual hyperlink that it represents.

Here is a [sample](http://blogs.msdn.com/vinsibal/attachment/8903487.ashx) for the first two solutions.

[MSDN Blogs](http://blogs.msdn.com/) > [Vincent Sibal's Blog](http://blogs.msdn.com/b/vinsibal/)> [WPF DataGrid: Stock and Template Columns](http://blogs.msdn.com/b/vinsibal/archive/2008/08/19/wpf-datagrid-stock-and-template-columns.aspx)

WPF DataGrid: Stock and Template Columns

[Vincent Sibal - MSFT](http://blogs.msdn.com/136277/ProfileUrlRedirect.ashx)

**19 Aug 2008 8:58 AM**

**Overview**

The *DataGrid* uses a set of *DataGridColumns* to describe how to display its data just like a*GridView* has a set of *GridViewColumns* to describe its data.  In my first [post](http://blogs.msdn.com/vinsibal/archive/2008/08/11/net-3-5-sp1-and-wpf-datagrid-ctp-is-out-now.aspx), the sample I used auto-generated the columns for you but this time I want to go over how to create the columns manually.

**Column Types**

Currently in our WPF DataGrid we have four available stock columns and a template column:

         *DataGridTextColumn*

         *DataGridCheckBoxColumn*

         *DataGridComboBoxColumn*

         *DataGridHyperlinkColumn*

         *DataGridTemplateColumn*

Each column type corresponds to the *UIElement* that it will show in each of the cells in that column.  The *DataGridTemplateColumn* allows you to customize the template with a custom*UIElement* or tree of Elements for each of the cells in the column.  Template columns derive from *DataGridColumn*, whereas stock columns derive from *DataGridBoundColumn* which derives from *DataGridColumn*.

Why the difference in inherited classes?  The main difference is that a*DataGridBoundColumn* includes the property *Binding*, which is a convenience property for mapping a data item property in the *ItemsSource* to its corresponding set of cells in the column.  Here is an example:

<dg:DataGridTextColumn Binding="{Binding Path=FirstName}" />

The concept is basically the same as setting the *DisplayMemberBinding* property on a*GridViewColumn* of a *GridView*.  I also talked about this in my post, [Dissecting the Visual Layout](http://blogs.msdn.com/vinsibal/archive/2008/08/14/wpf-datagrid-dissecting-the-visual-layout.aspx).  For a *DataGridTemplateColumn*, since you are free to set the template to whatever you like, we cannot automatically figure out where you would like to place the binding. Because of that, you are required to setup the binding yourself and therefore the class does not derive from *DataGridBoundColumn*.

**Stock Columns and styling**

For stock columns or *DataGridBoundColumns*, the main properties you will likely use are the*Binding* property, which I describe above, and the *Header* property.  Header is just like*GridViewColumn*.*Header*, which displays the text on the *DataGridColumnHeader* of that column.

<dg:DataGridTextColumn Header="First Name" Binding="{BindingPath=FirstName}" />

Styling a stock column can be accomplished with these particular properties:

         *CellStyle*

         *EditingElementStyle*

         *ElementStyle*

Recall in my post, [Dissecting the Visual Layout](http://blogs.msdn.com/vinsibal/archive/2008/08/14/wpf-datagrid-dissecting-the-visual-layout.aspx), when the *DataGridCell* is created its *Content*property is set after asking *DataGridColumn* to generate the visual tree.  Well, depending if you are in an editing state or not, the column will generate a different *UIElement* for each state.  For a *DataGridTextColumn* in an editing state, a *TextBox* is created.  In a non-editing state, a *TextBlock* is created.  For a *DataGridComboBoxColumn* in an editing state, a*ComboBox* is created, while in a non-editing state, a *TextBlock* is created.

With that in mind, *CellStyle* is the style for the overall *DataGridCell* (which is a*ContentControl*and items container), *EditingElementStyle* is the style for the *DataGridCell’s*Content generated during the editing state and *ElementStyle* is the style for the*DataGridCell’s* *Content* generated during the non-editing state.  Here is an example:

<dg:DataGridTextColumn Header="First Name" Binding="{BindingPath=FirstName}">

  <dg:DataGridTextColumn.CellStyle>

    <Style TargetType="{x:Type dg:DataGridCell}">

      <Setter Property="Foreground" Value="Blue" />

      <Setter Property="FontWeight" Value="Bold" />

    </Style>

  </dg:DataGridTextColumn.CellStyle>

  <dg:DataGridTextColumn.EditingElementStyle>

    <Style TargetType="TextBox">

      <Setter Property="Foreground" Value="Red" />

    </Style>

  </dg:DataGridTextColumn.EditingElementStyle>

  <dg:DataGridTextColumn.ElementStyle>

    <Style TargetType="TextBlock">

      <Setter Property="Foreground" Value="LightBlue" />

    </Style>

  </dg:DataGridTextColumn.ElementStyle>

</dg:DataGridTextColumn>

You should note that the *Bold* *FontWeight* will be applied to both the editing and non-editing element while the Foreground property set in *CellStyle* will be overwritten by the *Foreground*property set by each of the element styles.

**Template Columns**

For *DataGridTemplateColumns*, the *Header* and *CellStyle* property still apply but there isn’t a*Binding*, *EditingElementStyle*, or *ElementStyle* property.  Instead, you have the properties,*CellTemplate* and *CellEditingTemplate*.  *DataGridCell’s* *Content* generation is still the same; it’s just that it uses these two cell templates to generate the content instead.  Here is an example:

<dg:DataGridTemplateColumn Header="First Name">

  <dg:DataGridTemplateColumn.CellTemplate>

    <DataTemplate>

      <Button Content="{Binding Path=FirstName}" />

    </DataTemplate>

  </dg:DataGridTemplateColumn.CellTemplate>

  <dg:DataGridTemplateColumn.CellEditingTemplate>

    <DataTemplate>

      <TextBox Text="{Binding Path=FirstName, Mode=TwoWay,UpdateSourceTrigger=PropertyChanged}" />

    </DataTemplate>

  </dg:DataGridTemplateColumn.CellEditingTemplate>

</dg:DataGridTemplateColumn>

In a non-editing state a Button is created as the *UIElement* for the cell and is bound to the FirstName property of my data source.  In an editing state a *TextBox* is created as the*UIElement*.  Having an editable Button in a *DataGrid* may not be a common scenario but I did want to show both *CellTemplate* and *CellEditingTemplate*.  If you do not want to make it editable, one thing you can do is just not declare the *CellEditingTemplate*.  Notice in both templates I setup the binding myself since I do not have a *Binding* like*DataGridBoundColumns*.

**Important**: Also notice that in *CellEditingTemplate* I had to declare the binding with Mode set to TwoWay and UpdateSourceTrigger set to PropertyChanged.  This is another consequence of not getting the binding utilities that *DataGridBoundColumns* provide.  In a*DataGridBoundColumn*, when cells are edited and committed, the *DataGrid* will take care of updating the sources for you.  In the case of the template column, you will have to update the source yourself.  This issue is only for a commit scenario.  The *DataGrid* will still be able to rollback on a cancel command.

**Sizing Columns**

Column sizing uses a special class called *DataGridLength* that has sizing properties specifically for a row and column scenario.  The types of widths are as follows:

         Pixel

         SizeToCells

         SizeToHeader

         Auto

         Star

 Pixel uses absolute sizing on the column width, SizeToCells sizes the column width to the largest cell, SizeToHeader sizes the column width to the header width, Auto sizes the column width to either the largest cell or the header width whichever is larger, and Star sizing follow the proportional sizing like a *Grid* panel.  See the final example below for usage of the different widths.

**Putting it all together**

So with all that, I have updated the sample from my first [post](http://blogs.msdn.com/vinsibal/archive/2008/08/11/net-3-5-sp1-and-wpf-datagrid-ctp-is-out-now.aspx) on *DataGrid* with what I discuss here.  Instead of auto-generating the columns, I’ve explicitly declared each column and I’ve setup different widths for you to get an idea of how that works.  You can also double-click on the column header grippers to change the width to Auto.  This functionality is similar to Windows Explorer.  I’ve also created a template column that displays an *ImageUIElement* when in non-edit mode and a *CombBox* *UIElement* when in editing mode.  Notice the differences in how I setup the data bindings between the template column versus the bound columns.  I’ve also added a *DataGridCell* style and *DataGridRow* style that updates the*BorderBrush* and *BorderThickness* when in edit mode.  You can download the sample [here](http://blogs.msdn.com/vinsibal/attachment/8878680.ashx).

<Style x:Key="defaultCellStyle" TargetType="{x:Type dg:DataGridCell}">

  <Style.Triggers>

      <Trigger Property="IsEditing" Value="True">

          <Setter Property="BorderBrush" Value="Red" />

          <Setter Property="BorderThickness" Value="2" />

      </Trigger>

  </Style.Triggers>

</Style>

<dg:DataGrid AutoGenerateColumns="False" CellStyle="{StaticResourcedefaultCellStyle}" …>

  <dg:DataGrid.Columns>

      <dg:DataGridTextColumn Width="130" Header="First Name" Binding="{Binding Path=FirstName}" />

      <dg:DataGridTextColumn Width="Auto" Header="Last Name" Binding="{Binding Path=LastName}" />

      <dg:DataGridCheckBoxColumn Width="SizeToCells" Header="Likes Cake"Binding="{Binding Path=LikesCake}" />

      <dg:DataGridComboBoxColumn Width="200" Header="Cake"SelectedItemBinding="{Binding Path=Cake}">

          <dg:DataGridComboBoxColumn.ItemsSource>

                 <col:ArrayList>

                       <sys:String>Chocolate</sys:String>

                       <sys:String>Vanilla</sys:String>

                 </col:ArrayList>

          </dg:DataGridComboBoxColumn.ItemsSource>

      </dg:DataGridComboBoxColumn>

      <dg:DataGridHyperlinkColumn Width="SizeToHeader" Header="Homepage"Binding="{Binding Path=Homepage}" />

      <dg:DataGridTemplateColumn MaxWidth="250" Header="Picture">

          <dg:DataGridTemplateColumn.CellTemplate>

                 <DataTemplate>

                       <Image Source="{Binding Path=Picture}" />

                 </DataTemplate>

          </dg:DataGridTemplateColumn.CellTemplate>

          <dg:DataGridTemplateColumn.CellEditingTemplate>

                 <DataTemplate>

                       <ComboBox SelectedItem="{Binding Path=Picture,Mode=TwoWay, UpdateSourceTrigger=PropertyChanged}">

                             <ComboBox.ItemsSource>

                                   <col:ArrayList>

                                     <sys:String>Assets\Autumn Leaves.jpg</sys:String>

                                     <sys:String>Assets\Butterfly.JPG</sys:String>

                                     <sys:String>Assets\Green Sea Turtle.jpg</sys:String>

                                   </col:ArrayList>

                             </ComboBox.ItemsSource>

                       </ComboBox>

                 </DataTemplate>

          </dg:DataGridTemplateColumn.CellEditingTemplate>

      </dg:DataGridTemplateColumn>

  </dg:DataGrid.Columns>

</dg:DataGrid>

**More Related Material:**

* [Dissecting the Visual Layout](http://blogs.msdn.com/vinsibal/archive/2008/08/14/wpf-datagrid-dissecting-the-visual-layout.aspx)
* [Working with DataGridComboBoxColumn (Part1)](http://blogs.msdn.com/vinsibal/archive/2008/08/26/wpf-datagrid-working-with-datagridcomboboxcolumn.aspx)
* [Working with DataGridComboBoxColumn (Part2)](http://blogs.msdn.com/vinsibal/archive/2008/08/28/wpf-datagrid-working-with-datagridcomboboxcolumns-part-2.aspx)
* [Overview of the editing features in the DataGrid](http://blogs.msdn.com/vinsibal/archive/2008/10/01/overview-of-the-editing-features-in-the-wpf-datagrid.aspx)

**Other Samples:**

* [ScrollViewer with ToolTip](http://blogs.msdn.com/vinsibal/archive/2008/08/25/wpf-datagrid-sample-add-a-preview-tooltip-to-a-scrollviewer.aspx)
* [Custom sorting, column selection, single-click editing](http://blogs.msdn.com/vinsibal/archive/2008/08/27/more-datagrid-samples-custom-sorting-drag-and-drop-of-rows-column-selection-and-single-click-editing.aspx)
* [Tri-state sorting](http://blogs.msdn.com/vinsibal/archive/2008/08/29/wpf-datagrid-tri-state-sorting-sample.aspx)

[MSDN Blogs](http://blogs.msdn.com/) > [Vincent Sibal's Blog](http://blogs.msdn.com/b/vinsibal/)> [Overview of the editing features in the WPF DataGrid](http://blogs.msdn.com/b/vinsibal/archive/2008/10/01/overview-of-the-editing-features-in-the-wpf-datagrid.aspx)

### Overview of the editing features in the WPF DataGrid

[Vincent Sibal - MSFT](http://blogs.msdn.com/136277/ProfileUrlRedirect.ashx)

**1 Oct 2008 10:50 AM**

**Introduction**

I’m going to talk a little on the editing features of the DataGrid.  I will dive deep into a couple things but overall I just want to give you a good idea of the APIs you can work with and how to tweak a couple things.  So basically I will be introducing a bit of the internal implementation that may be beneficial for you to understand, the editing commands and how to customize, and the editing events that are available.

**Background on the data source and DataGrid working together**

Some major updates were done in 3.5 SP1 to enable this editing scenario for the *DataGrid*.  In particular, the [*IEditableCollectionView*](http://msdn.microsoft.com/en-us/library/system.componentmodel.ieditablecollectionview.aspx) interface was added and [*ListCollectionView*](http://msdn.microsoft.com/en-us/library/system.windows.data.listcollectionview.aspx) and[*BindingListCollectionView*](http://msdn.microsoft.com/en-us/library/system.windows.data.bindinglistcollectionview.aspx) were updated to support this interface.  Read more on*IEditableCollectionView* [here](http://blogs.msdn.com/vinsibal/archive/2008/05/20/wpf-3-5-sp1-feature-ieditablecollectionview.aspx).  As a refresher, *ListCollectionView* is the view created for an*ItemsControl* when your data source implements [*IList*](http://msdn.microsoft.com/en-us/library/system.collections.ilist.aspx) such as *ObservableCollection<T>*.  *BindingListCollectionView* is the view created for an *ItemsControl* when your data source implements [*IBindingList*](http://msdn.microsoft.com/en-us/library/system.componentmodel.ibindinglist.aspx)such as an ADO.net DataView.

The DataGrid uses *IEditableCollectionView* underneath the covers to support transactional editing as well as adding and removing data items.  In the implementations of*IEditableCollectionView* in *ListCollectionView* and *BindingListCollectionView*, they both follow the [*IEditableObject*](http://msdn.microsoft.com/en-us/library/system.componentmodel.ieditableobject.aspx) pattern where the calls to *IEditableCollectionView.EditItem*,*IEditableCollectionView.CancelEdit*, and *IEditableCollectionView.CommitEdit* end up calling*IEditableObject.BeginEdit*, *IEditableObject.CancelEdit*, and *IEditableObject.CommitEdit*respectively.  It is in the implementation of *IEditableObject* where you can provide the functionality to commit or rollback changes to the data source.  For an ADO.net DataTable, you get this functionality for free as DataRowView already implements *IEditableObject*. When using an *ObservableCollection<T>* you will have to provide your own implementation for *IEditableObject* for T.  See the MSDN documentation of [*IEditableObject*](http://msdn.microsoft.com/en-us/library/system.componentmodel.ieditableobject.aspx) for an example of how to implement it for a data item.

Things to keep in mind:

         DataGrid checks for *IEditableCollectionView’s* *CanAddNew*, *CanCancelEdit*, and *CanRemove* properties before executing the *EditItem*, *CancelEdit*, or *CommitEdit*methods.  So if editing appears to not work for some reason, be sure to check that it is able to edit.

|  |  |  |
| --- | --- | --- |
|  | ListCollectionView | BindingListCollectionView |
| CanAddNew | True if there is not an edit transaction occurring, if the collection is not a fixed size, and if it can create an object of the correct type. | True if there is not an edit transaction occurring, if the collection is not a fixed size, and if the collection is not read-only. |
| CanCancelEdit | True if the edited item implements the*IEditableObject* interface. | True if the edited item implements the*IEditableObject* interface. |
| CanRemove | True if the collection is not a fixed size and if an add or edit transaction is not occurring. | True if the collection is not a fixed size and if an add or edit transaction is not occurring and if the collection is not read-only. |

For information on how the data binding is hooked to the UI, see this post on [Stock and Template Columns](http://blogs.msdn.com/vinsibal/archive/2008/08/19/wpf-datagrid-stock-and-template-columns.aspx) and [Dissecting the Visual Layout](http://blogs.msdn.com/vinsibal/archive/2008/08/14/wpf-datagrid-dissecting-the-visual-layout.aspx).

**Note about DataGrid properties related to editing**

There are three properties on DataGrid to control editing/adding/deleting.  These properties are:

         *CanUserAddRows*

         *CanUserDeleteRows*

         *IsReadOnly* (not in CTP)

They are basically self-documenting but beware of *CanUserAddRows* and*CanUserDeleteRows* as they can appear a little magical.  Their values are coerced based on other properties such as *DataGrid.IsReadOnly*, *DataGrid.IsEnabled*,*IEditableCollectionView.CanAddNew*, and *IEditableCollectionView.CanRemove*.  So this is another thing to watch out for when editing.  If you run into a situation where you set CanUserAddRows or CanUserDeleteRows to true but it is changed to false automatically, check that the conditions below are met.

|  |  |
| --- | --- |
| CanUserAddRows | True if the DataGrid is not ReadOnly and IsEnabled, CanUserAddRows is set to true and IEditableCollectionView.CanAddNew is true |
| CanUserDeleteRows | True if the DataGid is not ReadOnly and IsEnabled, CanUserDeleteRows is set to true and IEditableCollectionView.CanRemove is true |

**Working with editing commands**

Default commands have been added to the *DataGrid* to support editing.  These commands and their default input bindings are:

         *BeginEditCommand* (F2)

         *CancelEditCommand* (Esc)

         *CommitEditCommand* (Enter)

         *DeleteCommand* (Delete)

When each command is executed it will do some internal housekeeping and at some point it will call into its *IEditableCollectionView* counterpart.  For example, *BeginEditCommand* calls into *IEditableCollectionView.EditItem*and *CancelEditCommand* calls into*IEditableCollectionView.CancelItem*.

DataGrid also has APIs where you can call editing commands programmatically.  Not surprisingly, the APIs are *BeginEdit*, *CancelEdit*, and *CommitEdit*.

**Adding new input gestures**

Adding new input gestures is similar to any other control in WPF.  The DataGrid commands are added through the *CommandManager* so one possible solution would be to register a new *InputBinding*with the*CommandManager*:

CommandManager.RegisterClassInputBinding(

                      typeof(DataGrid),

                new InputBinding(DataGrid.BeginEditCommand, newKeyGesture(Key.<new key>)));

**Disabling commands**

You can disable any of the editing commands by attaching to the*CommandManager.CanExecuteEvent*, looking for the command to disable, then setting e.CanExecute accordingly.  Here is an example:

\_handler = newCanExecuteRoutedEventHandler(OnCanExecuteRoutedEventHandler);

EventManager.RegisterClassHandler(typeof(DataGrid),CommandManager.CanExecuteEvent, \_handler);

void OnCanExecuteRoutedEventHandler(object sender,CanExecuteRoutedEventArgs e)

{

  RoutedCommand routedCommand = (e.Command as RoutedCommand);

  if (routedCommand != null)

  {

    if (routedCommand.Name == "<command name>")

    {

      e.CanExecute = <some condition>;

      if(!e.CanExecute)

        e.Handled = true;  
    }  
  }

}

This is a relatively cumbersome way of disabling an editing command from being executed. Fortunately events were added to the DataGrid so that they can be canceled in a more direct fashion (although no direct event exists for the *DeleteCommand*).

**Editing events on the DataGrid**

These are the editing events that you can listen to and cancel the operation or modify data:

         *RowEditEnding*

         *CellEditEnding*

         *BeginningEdit*

         *PreparingCellForEdit*

         *InitializingNewItem*

In the first three events; Row*EditEnding*, *CellEditEnding*, and *BeginningEdit*, you have access to the *DataGridRow* and *DataGridColumn* that is being committed, cancelled, or edited. These events are called right before the actual operation will occur.  You have the ability to cancel the operation completely by setting e.Cancel to true.  RowEditEnding and CellEditEnding both have a parameter EditAction which lets you know if it is a commit or cancel action.  From *CellEditEnding*, you also have access to the editing *FrameworkElement*. This gives you the ability to set or get properties on the visual itself before a cell commit or cancel.

*PreparingCellForEdit* is fired right after the cell has changed from a non-editing state to an editing state.  In this event you have the ability to modify the contents of the cell. *InitializingNewItem* is called when a new item is added and in this event you have the option to set any properties on the newly created item.  This event is good when you want to set initial default values on a new item.

**Summary**

So hopefully this will give you an idea on what APIs you have available for editing scenarios as well as some gotchas on how particular editing features work.  If there is other editing questions/topics that you would like to read about please let me know.  Another item that I plan to discuss in the future is how row validation will be tied into the DataGrid, so stay tuned!

[MSDN Blogs](http://blogs.msdn.com/) > [Vincent Sibal's Blog](http://blogs.msdn.com/b/vinsibal/)> [WPF DataGrid Design-time Walkthrough](http://blogs.msdn.com/b/vinsibal/archive/2008/11/07/wpf-datagrid-design-time-walkthrough.aspx)

### WPF DataGrid Design-time Walkthrough

[Vincent Sibal - MSFT](http://blogs.msdn.com/136277/ProfileUrlRedirect.ashx)

**7 Nov 2008 10:29 AM**

The DataGrid [walkthrough](http://windowsclient.net/wpf/wpf35/wpf-35sp1-toolkit-datagrid-feature-walkthrough.aspx#designtime) on windowsclient.net/wpf and the [Tips & Tricks section](http://www.codeplex.com/wpf/Wiki/View.aspx?title=Design%20Time&referringTitle=Tips%20%26%20Tricks) on codeplex talk briefly about the Design-time support for DataGrid.  I thought I’d expand on that just a little more so you get a good idea of what you have available.

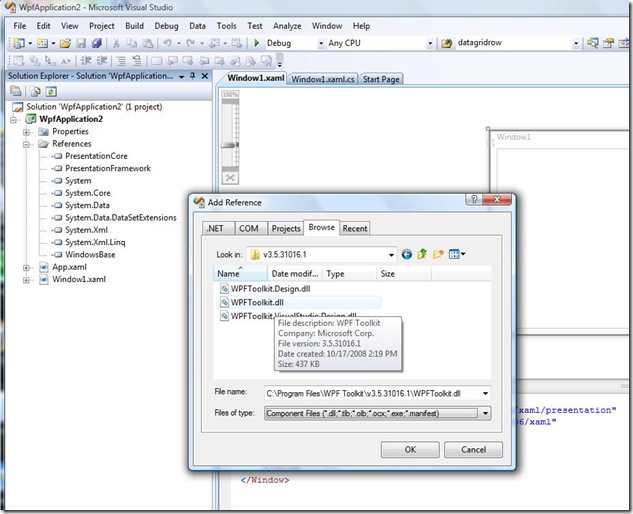
**Installation**

So to get it to work you need to install the toolkit using the WPFToolkit.msi (download from [here](http://www.codeplex.com/wpf/Release/ProjectReleases.aspx?ReleaseId=15598)).  This will put the dlls under your “C:\Program Files\WPF Toolkit\<version number>\” directory as well as set a reg key in Visual Studio to recognize the controls in the toolbox.  In your application, you will need to reference the wpftoolkit.dll from this install directory location to get the design-time support to work.

**Walkthrough**

Ok, I’ll start from the very beginning.

0.       Install and set a reference to wpftoolkit.dll

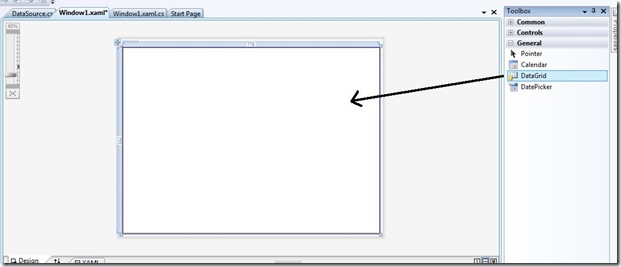
[](http://blogs.msdn.com/blogfiles/vinsibal/WindowsLiveWriter/WPFDataGridDesigntimeWalkthrough_6BB9/designtime_step1_2.jpg)

1.       Open the Toolbox.  You should see a group with DataGrid, DatePicker, and Calendar.

a.       If you don’t, you can add them by right-clicking in the Toolbox, selecting “Choose Toolbox Items”, tab to the WPF Components tab, then selecting DataGrid, DatePicker, and Calendar.

2.       **Drag & drop support:**  Drag the DataGrid item from the Toolbox to the window on the design surface.

a.       Note: You may need to resize the window as well as work with the layout so the DataGrid fits as you would like.

[](http://blogs.msdn.com/blogfiles/vinsibal/WindowsLiveWriter/WPFDataGridDesigntimeWalkthrough_6BB9/designtime_part2_2.jpg)

3.       Setup the ItemsSource for the DataGrid

a.       In my example I’m going to use a DataTable that retrieves the Customers Table from the Northwind database.   The code for that is in the sample on this [post](http://blogs.msdn.com/vinsibal/archive/2008/10/22/wpf-datagrid-and-the-wpftoolkit-have-released.aspx).

<Window.Resources>

  <ObjectDataProvider x:Key="CustomerDataProvider"

                 ObjectType="{x:Type local:Customers}"

                 MethodName="GetCustomers" />

</Window.Resources>

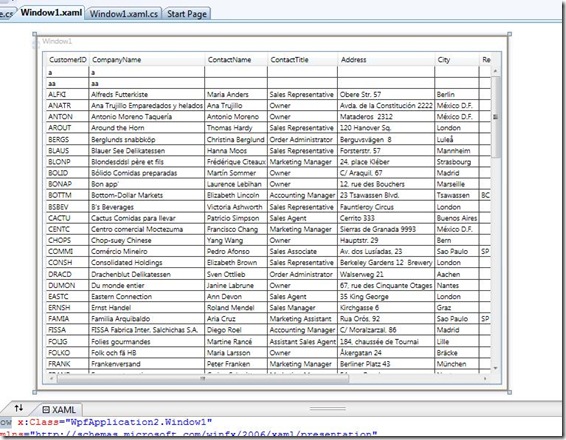
<Grid>

  <my:DataGrid Name="dataGrid1"

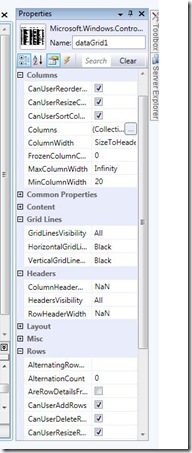
          ItemsSource="{Binding Source={StaticResourceCustomerDataProvider}}" />

</Grid>

b.      As soon as the ItemsSource is set, the design surface should update to show you the data on the DataGrid.  This is because DataGrid.AutoGenerateColumns is set to true by default.

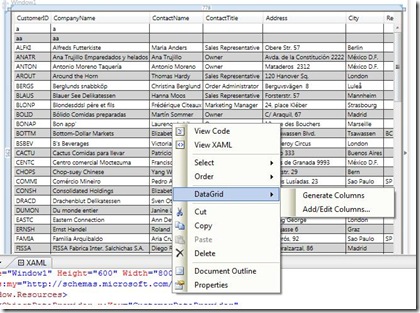
[](http://blogs.msdn.com/blogfiles/vinsibal/WindowsLiveWriter/WPFDataGridDesigntimeWalkthrough_6BB9/designtime_part3_2.jpg)

4.       **Properties View grouping support:**  Now, click on the DataGrid in the design window and open the Properties View.  Set the grouping to “Categorized”.  You will find some groups that are specific to DataGrid such as the Columns group, GridLines group, Headers group, and Rows group.

[](http://blogs.msdn.com/blogfiles/vinsibal/WindowsLiveWriter/WPFDataGridDesigntimeWalkthrough_6BB9/designtime_part4_2.jpg)

a.       In my example, from the Properties View I set EnableColumnVirtualization to true and AlternationRowBackground to LightGray

5.       Now, right-click on the DataGrid in design view.  You should see a DataGrid menu item with additional sub-items.  The sub-items will be a little different depending if you have an ItemsSource set or not.  Here is what it looks like with an ItemsSource set but no columns explicitly added:

[](http://blogs.msdn.com/blogfiles/vinsibal/WindowsLiveWriter/WPFDataGridDesigntimeWalkthrough_6BB9/designtime_part5_1_2.jpg)

When no ItemsSource is set, some sub-items will not appear.  When explicit columns are set on the DataGrid, additional sub-items will be added.  Now, let’s try out each of the options.

6.       **Generating columns all at once:**  Right-click on the DataGrid in design view.  Select the DataGrid menu item, then select the “Generate Columns” sub-menu item.  What just happened?

a.       That may be a little confusing at first since the design view didn’t change. The design-time sets DataGrid.AutoGenerateColumns to false, then generates a list of columns with a one-to-one mapping of the data source’s properties and adds them to the columns collection in the xaml.  Here is what my DataGrid looks like now in the xaml:

<my:DataGrid Name="dataGrid1" ItemsSource="{Binding Source={StaticResource CustomerDataProvider}}"EnableColumnVirtualization="True" AlternatingRowBackground="LightGray"AutoGenerateColumns="False">

  <my:DataGrid.Columns>

    <my:DataGridTextColumn Binding="{Binding Path=CustomerID}"Header="CustomerID" />

    <my:DataGridTextColumn Binding="{Binding Path=CompanyName}"Header="CompanyName" />

    <my:DataGridTextColumn Binding="{Binding Path=ContactName}"Header="ContactName" />

    <my:DataGridTextColumn Binding="{Binding Path=ContactTitle}"Header="ContactTitle" />

    <my:DataGridTextColumn Binding="{Binding Path=Address}"Header="Address" />

    <my:DataGridTextColumn Binding="{Binding Path=City}" Header="City" />

    <my:DataGridTextColumn Binding="{Binding Path=Region}"Header="Region" />

    <my:DataGridTextColumn Binding="{Binding Path=PostalCode}"Header="PostalCode" />

    <my:DataGridTextColumn Binding="{Binding Path=Country}"Header="Country" />

    <my:DataGridTextColumn Binding="{Binding Path=Phone}" Header="Phone" />

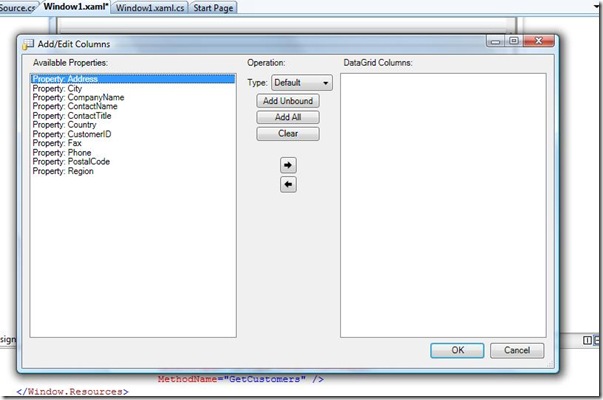
    <my:DataGridTextColumn Binding="{Binding Path=Fax}" Header="Fax" />

  </my:DataGrid.Columns>

</my:DataGrid>

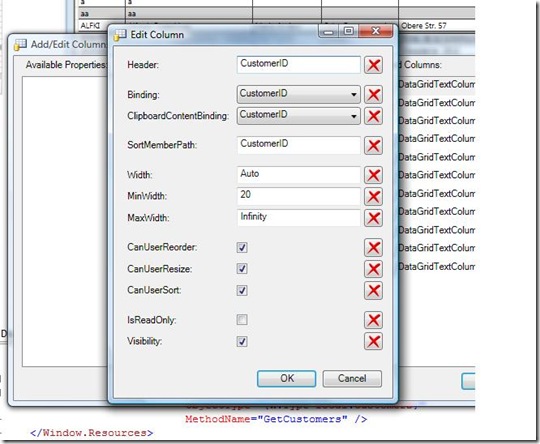
7.       **Removing all columns:**   Now that I have explicit columns set, if I wanted to clear them I can right-click on the DataGrid in design view, select DataGrid -> Remove Columns.  This will remove all the columns from xaml but it will not change the AutoGenerateColumns property back to true.

8.       **Creating a column with the Add/Edit Columns editor:**With all columns removed now, select DataGrid -> Add/Edit Columns…  You should be presented with a column editor where the data sources properties are mapped on the left panel.The middle area is where you can choose the type of DataGridColumn to create and the right panel shows the columns that will be created.

[](http://blogs.msdn.com/blogfiles/vinsibal/WindowsLiveWriter/WPFDataGridDesigntimeWalkthrough_6BB9/designtime_part7_2.jpg)

a.       Leave Type as Default and press the Add All button.  You should see all the properties on the left panel moved to the right panel and mapped to columns. The default column is DataGridTextColumn.

b.      Now, press the “Edit…” button on the first column in the right panel.  An “Edit Column” editor should pop up with a list of common properties to set on the column.

[](http://blogs.msdn.com/blogfiles/vinsibal/WindowsLiveWriter/WPFDataGridDesigntimeWalkthrough_6BB9/designtime_part8_b_2.jpg)

c.       Edit any of the fields of the columns you created and press ok on the “Add/Edit Columns” editor.  The xaml should be updated with the set of columns you’ve created.

**Summary**

To recap, the cider design-time supports toolbox drag and drop, generating all columns, generating specific columns, editing columns, and removing columns.  Remember that you have to set the ItemsSource on the DataGrid declaratively in order to get that functionality. The Properties View has also been updated for the DataGrid to support grouping of specific properties on the DataGrid.  Please try it out and any feedback would be greatly appreciated.

**Other Design-time Resources**

[Karl Shifflett](http://karlshifflett.wordpress.com/) has created this awesome tool called [XAML Power Toys](http://karlshifflett.wordpress.com/xaml-power-toys/).  Included in the tool is a utility for creating and customizing a WPF DataGrid.  Check out the video walkthrough of it[here](http://silverlight.services.live.com/invoke/48184/XAMLPowerToysv3-4WPFDataGrid/iframe.html).

[MSDN Blogs](http://blogs.msdn.com/) > [Vincent Sibal's Blog](http://blogs.msdn.com/b/vinsibal/)> [5 Random Gotchas with the WPF DataGrid](http://blogs.msdn.com/b/vinsibal/archive/2009/04/07/5-random-gotchas-with-the-wpf-datagrid.aspx)

### 5 Random Gotchas with the WPF DataGrid

[Vincent Sibal - MSFT](http://blogs.msdn.com/136277/ProfileUrlRedirect.ashx)

**7 Apr 2009 9:54 AM**

Since the release of the WPF DataGrid there have been several common patterns of questions that developers were asking on the [discussion list](http://wpf.codeplex.com/Thread/List.aspx).  I thought that I would capture some of that here so it would be easy to find.  I'll also try to keep it as short as possible and refer to other links for more information.  These gotchas are in no particular order.

**1.**       **DataGridColumns are not part of the visual tree.**

Since a DataGridColumn has a Binding DP it is easy to think that bindings to other DataGridColumn DPs will work just the same but they don't.  Below is an example showing a DataGridCheckBoxColumn setting the Binding DP and CanUserSort through data binding. While Binding will bind correctly, CanUserSort will not.

<dg:DataGridCheckBoxColumn Header="Selection"

                           Binding="{Binding IsSelected}"

                           CanUserSort="{Binding CanSort}"/>

Note that the reason Binding does work is because the internal code dynamically sets the binding of DataGridColumn.Binding to the DataGridCell.Content.  It does not do so for the other DataGridColumn DPs.

Workaround => Check out [Jaime’s post](http://blogs.msdn.com/jaimer/archive/2008/11/22/forwarding-the-datagrid-s-datacontext-to-its-columns.aspx) on how to get the DataGridColumns to use the same DataContext as the DataGrid.

**2.**       **DataGridColumn.Binding will automatically coerce the binding to BindingMode.TwoWay and UpdateSourceTrigger.Explicit**

No matter how you declare the binding for DataGridColumn.Binding, internally it will coerce the BindingMode to TwoWay and UpdateSourceTrigger to Explicit (even if you set them explicitly yourself).  This is because the DataGrid manages the binding operations itself for editing functionality.  Now, you might run into an issue when you have a read-only property with a TwoWay binding mode.  If you set DataGridColumn.IsReadOnly to true, then the BindingMode will not be coerced to TwoWay.

**3.**       **You manage the bindings when using a DataGridTemplateColumn.**

The DataGrid internal code does not coerce the bindings set on DataGridTemplateColumn because DataGridTemplateColumn doesn’t even have a Binding DP.  Since you are setting the bindings up yourself, you have to tell it when to commit.  One common way to do this is to setup a OneWay binding on the UIElement in the CellTemplate and a TwoWay binding on the CellEditingTemplate.  See this previous [post](http://blogs.msdn.com/vinsibal/archive/2008/08/19/wpf-datagrid-stock-and-template-columns.aspx) for example code (under “Template Columns”).  The example code uses an UpdateSourceTrigger of PropertyChanged.  If that does not suite your needs for committing the data, you can mimic what the DataGrid internal code does and update the source yourself.  Set the UpdateSourceTrigger to Explicit and listen to the DataGrid.CellEditEnding event.  You can then update the source explicitly in that event.

**4.**       **Use DataGridColumn.ElementStyle and DataGridColumn.EditingElementStyle to set properties on a cell’s Content.**

Each stock DataGridColumn type produces a specific type of Control as the content for each DataGridCell in the column.  See this previous [post](http://blogs.msdn.com/vinsibal/archive/2008/08/19/wpf-datagrid-stock-and-template-columns.aspx) for more info on that.  It seems that the first instinct to set properties on the cell’s content is to set them through the DataGridCell Style when they should instead be set on DataGridColumn.ElementStyle and/or DataGridColumn.EditingElementStyle.  ElementStyle is the Style applied to the non-editing element.  EditingElementStyle is for the editing element.

In a previous gotcha, I mention that DataGridColumns are not part of the visual tree.  With that said, you can still create bindings that map to the same DataContext as DataGridColumn.Binding on DPs from a Style set on DataGridColumn.ElementStyle and DataGridColumn.EditingElementStyle.  That is because these styles are applied dynamically to the cell’s content which has the correct DataContext.  Below I am setting a DataGridTextBoxColumn’s ElementStyle.  Notice that I have a binding setup on Background with a “MyBrush” property path.  I am assuming that property is coming from the DataContext of the DataGridCell (which is the same DataContext of the DataGridRow).

<dg:DataGridTextColumn Header="First Name"

            Binding="{Binding Path=FirstName}">

  <dg:DataGridTextColumn.ElementStyle>

      <Style TargetType="{x:Type TextBlock}">

        <Setter Property="ToolTip" Value="{StaticResource TextToolTip}" />

        <Setter Property="Background" Value="{Binding MyBrush}" />

        <Setter Property="TextWrapping" Value="Wrap" />

      </Style>

  </dg:DataGridTextColumn.ElementStyle>

</dg:DataGridTextColumn>

**5.**       **Data source items should implement IEditableObject to get transactional editing functionality.**

If you are not familiar with IEditableObject, see this MSDN [article](http://msdn.microsoft.com/en-us/library/system.componentmodel.ieditableobject.aspx) which has a good explanation and code sample.  The DataGrid has baked in functionality for transactional editing via the IEditableObject interface.  When you begin editing a cell, the DataGrid gets into cell editing mode as well as row editing mode.  What this means is that you can cancel/commit cells as well as cancel/commit rows.  For example, I edit cell 0 and press tab to the next cell.  Cell 0 is committed when pressing tab.  I start typing in cell 1 and realize I want to cancel the operation.  I press ‘Esc’ which reverts cell 1.  I now realize I want to cancel the whole operation so I press ‘Esc’ again and now cell 0 is reverted back to its original value.

See more [WPF DataGrid gotchas](http://blogs.msdn.com/vinsibal/archive/2009/04/14/5-more-random-gotchas-with-the-wpf-datagrid.aspx).