ComponentOne

FlexGrid for UWP

GrapeCity US

GrapeCity
201 South Highland Avenue, Suite 301
Pittsburgh, PA 15206
Tel: 1.800.858.2739 | 412.681.4343
Fax: 412.681.4384
Website: https://www.grapecity.com/en/
E-mail: us.sales@grapecity.com

Trademarks

Warranty

ComponentOne warrants that the media on which the software is delivered is free from defects in material and workmanship, assuming normal use, for a period of 90 days from the date of purchase. If a defect occurs during this time, you may return the defective media to ComponentOne, along with a dated proof of purchase, and ComponentOne will replace it at no charge. After 90 days, you can obtain a replacement for the defective media by sending it and a check for $2 5 (to cover postage and handling) to ComponentOne.

Except for the express warranty of the original media on which the software is delivered is set forth here, ComponentOne makes no other warranties, express or implied. Every attempt has been made to ensure that the information contained in this manual is correct as of the time it was written. ComponentOne is not responsible for any errors or omissions. ComponentOne's liability is limited to the amount you paid for the product. ComponentOne is not liable for any special, consequential, or other damages for any reason.

Copying and Distribution

Table of Contents

FlexGrid for UWP

Display tabular data across columns and down rows with this simple, yet powerful grid control. **FlexGrid for UWP** delivers a familiar, traditional datagrid re-designed for modern, touch-based enterprise applications.

Help with UWP Edition

For information on installing **ComponentOne Studio UWP Edition**, licensing, technical support, namespaces and creating a project with the control, please visit Getting Started with ComponentOne Studio UWP Edition.

FlexGrid for UWP Key Features

FlexGrid for UWP includes the following key features:

- **Touch Interaction**

  In the future there are no scroll bars. Designed to provide a UX more fitting for the mobile world, **FlexGrid for UWP** allows users to slide the grid to scroll across columns and traverse rows.

- **Flexible Data Binding**

  Easily bind the **C1FlexGrid** control to anything that implements **IEnumerable** or bind directly to a list of objects.

- **Unbound**

  In addition to data binding support, **C1FlexGrid** also works great in unbound mode. Simply add rows and columns to the grid using familiar syntax from the WinForms version of the control. When unbound, the **C1FlexGrid** control stores data internally.

- **Grouping**

  C1FlexGrid supports grouping that is performed on the underlying CollectionView when the control is data bound. Grouped rows are created and can be expanded/collapsed by tapping.

- **Cell Editing**

  C1FlexGrid supports cell editing through an easy-to-use interface for the Windows Store. The editing is not inline as it would be in a Web or desktop application, but rather in a separate pop-up perfect for the touch interface.

- **Custom Cells**

  C1FlexGrid makes it easier to define custom cells by providing a simpler interface, ICellFactory. Custom cells are useful if you want to display anything other than text in a grid cell, or if you need to apply conditional formatting.

- **Cell Merging**

  C1FlexGrid enables cell merging at the grid level. Just set the AllowMerging property on specific rows and columns to enable this feature where you need it. Cell merging will merge adjacent cells that have the same content, while positioning the text so that it is always most readable to the user.

- **Column and Row Headers**

  FlexGrid supports multiple fixed rows for column headers and multiple fixed columns for row headers. This concept allows you to create "bands" or multi-cell headers useful for organizing columns into groups. Cell merging is used to achieve this functionality.

- **Autosize Rows and Columns**

  The **C1FlexGrid** control has AutoSizeRow and AutoSizeColumn methods that take into account the data currently visible. This can be done quickly and accurately.

- **Save to Stream**

  Output and persist data with ease using the Save method. Formats include plain text, html and CSV.

- **Drag and Drop Support**

  Use the AllowDragging and AllowDrop properties to enable the ability to drag and drop rows and columns to a new location within the grid.

- **Row Details**

  FlexGrid provides the flexibility to use row details as a data template to show additional information with the help of text, images as well as data bound controls.

- **Print Grid Content**

  FlexGrid allows you to print the grid content using the Print method and preview it using the PrintPreview method.

FlexGrid for UWP Quick Start

The following quick start guide is intended to get you up and running with **FlexGrid for UWP**. In this quick start you'll start in Visual Studio and create a new project, add a **FlexGrid for UWP** control to your application, and add content to the control.

Step 1 of 3: Adding C1FlexGrid to the Application

In this step you'll begin in Visual Studio to create a UWP-style application using **FlexGrid for UWP** . To set up your project and add a C1FlexGrid control to your application, complete the following steps:

1. In Visual Studio, Select **File | New | Project**.
2. Select Templates | Visual C# | Windows | Universal. From the templates list, select Blank App (Universal Windows). Enter a **Name** and click **OK** to create your project.
3. Open **MainPage.xaml** if it isn't already open, place the cursor between the <Grid> and </Grid> tags, and click once.
   Navigate to the Toolbox and double-check the **C1FlexGrid** icon to add the control to your application.
4. Edit the **C1FlexGrid**'s markup so it appears similar to the following:

   Markup

   ```
   <FlexGrid:C1FlexGrid x:Name="flexgrid1" AllowResizing="Both"
   AllowDragging="Both" AllowDrop="True" ColumnHeaderForeground="Black"/>
   ```

   This markup sets the control's name, makes the rows and columns resizable, sets the ability to drag and drop rows and columns and sets the color of the header text.

What You've Accomplished

In this topic you successfully created a UWP-style application. In the next step you'll add code to the application to add data to be displayed in the grid.

Step 2 of 3: Adding Data to the C1FlexGrid Application

In the previous step you created a new UWP-style project and added a C1FlexGrid control to the application, but the grid currently contains no data – if you run the application now you'll see a blank grid. In this step you'll continue by adding an XML data source to your project and binding the grid to the data source.

To add a data source and bind the grid in Visual Studio, complete the following steps:

1. In the Solution Explorer window, right-click the project and select **Add | New Item**.
2. In the **Add New Item** dialog box, select **XML File** from the list of installed templates, name the file "Products.xml", and click **Add** to close the dialog box.

   The **Products.xml** file should now be included in your project, and should have opened automatically.

3. Replace the existing text in the **Products.xml** file with the following XML markup and save the file:

   Markup

   ```
   <?xml version="1.0" encoding="utf-8" ?>
   <Products>
     <Product Name="Chai" Category="Beverages" Unit="10 boxes x 20 bags" Price="18"
   />
   ```

```
  <Product Name="Chang" Category="Beverages" Unit="24 - 12 oz bottles"
Price="19" />
  <Product Name="Aniseed Syrup" Category="Condiments" Unit="12 - 550 ml bottles"
Price="10" />
  <Product Name="Chef Anton's Cajun Seasoning" Category="Condiments" Unit="48 -
6 oz jars" Price="22" />
  <Product Name="Chef Anton's Gumbo Mix" Category="Condiments" Unit="36 boxes"
Price="21.35" />
  <Product Name="Grandma's Boysenberry Spread" Category="Condiments" Unit="12 -
8 oz jars" Price="25" />
  <Product Name="Uncle Bob's Organic Dried Pears" Category="Produce" Unit="12 -
1 lb pkgs." Price="30" />
  <Product Name="Northwoods Cranberry Sauce" Category="Condiments" Unit="12 - 12
oz jars" Price="40" />
  <Product Name="Mishi Kobe Niku" Category="Meat/Poultry" Unit="18 - 500 g
pkgs." Price="97" />
  <Product Name="Ikura" Category="Seafood" Unit="12 - 200 ml jars" Price="31" />
  <Product Name="Queso Cabrales" Category="Dairy Products" Unit="1 kg pkg."
Price="21" />
  <Product Name="Queso Manchego La Pastora" Category="Dairy Products" Unit="10 -
500 g pkgs." Price="38" />
  <Product Name="Konbu" Category="Seafood" Unit="2 kg box" Price="6" />
  <Product Name="Tofu" Category="Produce" Unit="40 - 100 g pkgs." Price="23.25"
/>
  <Product Name="Genen Shouyu" Category="Condiments" Unit="24 - 250 ml bottles"
Price="15.5" />
  <Product Name="Pavlova" Category="Condiments" Unit="32 - 500 g boxes"
Price="17.45" />
  <Product Name="Alice Mutton" Category="Meat/Poultry" Unit="20 - 1 kg tins"
Price="39" />
  <Product Name="Carnarvon Tigers" Category="Seafood" Unit="16 kg pkg."
Price="62.5" />
  <Product Name="Teatime Chocolate Biscuits" Category="Confections" Unit="10
boxes x 12 pieces" Price="9.2" />
  <Product Name="Sir Rodney's Marmalade" Category="Confections" Unit="30 gift
boxes" Price="81" />
  <Product Name="Sir Rodney's Scones" Category="Confections" Unit="24 pkgs. x 4
pieces" Price="10" />
  <Product Name="Gustaf's Knäckebröd" Category="Grains/Cereals" Unit="24 - 500 g
pkgs." Price="21" />
  <Product Name="Tunnbröd" Category="Grains/Cereals" Unit="12 - 250 g pkgs."
Price="9" />
  <Product Name="Guaraná Fantástica" Category="Beverages" Unit="12 - 355 ml
cans" Price="4.5" />
  <Product Name="NuNuCa Nuß-Nougat-Creme" Category="Confections" Unit="20 - 450
g glasses" Price="14" />
</Products>
```

This will add data taken from the *Products* table of the standard Microsoft Northwind database.

4. Select **View | Code** to switch to Code view.

5. Add code to the page's constructor so that it appears like the following:

Visual Basic

```
Public Sub New()
    Me.InitializeComponent()
    LoadData()
End Sub
```

C#

```
public MainPage()
{
    this.InitializeComponent();
    LoadData();
}
```

6. Add the following code just below the page's constructor and within the main class:

Visual Basic

```
' Create the Product class.
    Public Class Product
        Private _Name As String
        Public Property Name() As String
            Get
                Return _Name
            End Get
            Set(ByVal value As String)
                _Name = value
            End Set
        End Property
        Private _Category As String
        Public Property Category() As String
            Get
                Return _Category
            End Get
            Set(ByVal value As String)
                _Category = value
            End Set
        End Property
        Private _Unit As String
        Public Property Unit() As String
            Get
                Return _Unit
            End Get
            Set(ByVal value As String)
                _Unit = value
            End Set
        End Property
        Private _Price As String
        Public Property Price() As String
            Get
```

```
                Return _Price
            End Get
            Set(ByVal value As String)
                _Price = value
            End Set
        End Property
    End Class
    Private Sub LoadData()
        ' Initialize the XML data source.
        Dim ProductsDoc As XDocument = XDocument.Load("Products.xml")
        Dim data As IEnumerable(Of Product) = (From Product In
ProductsDoc.Descendants("Product") Select New Product With {.Name =
Product.Attribute("Name").Value, .Category =
Product.Attribute("Category").Value, .Unit = Product.Attribute("Unit").Value,
.Price = Product.Attribute("Price").Value}).ToList
        ' Bind the C1FlexGrid control to the data source.
        flexgrid1.ItemsSource = data
    End Sub
```

C#

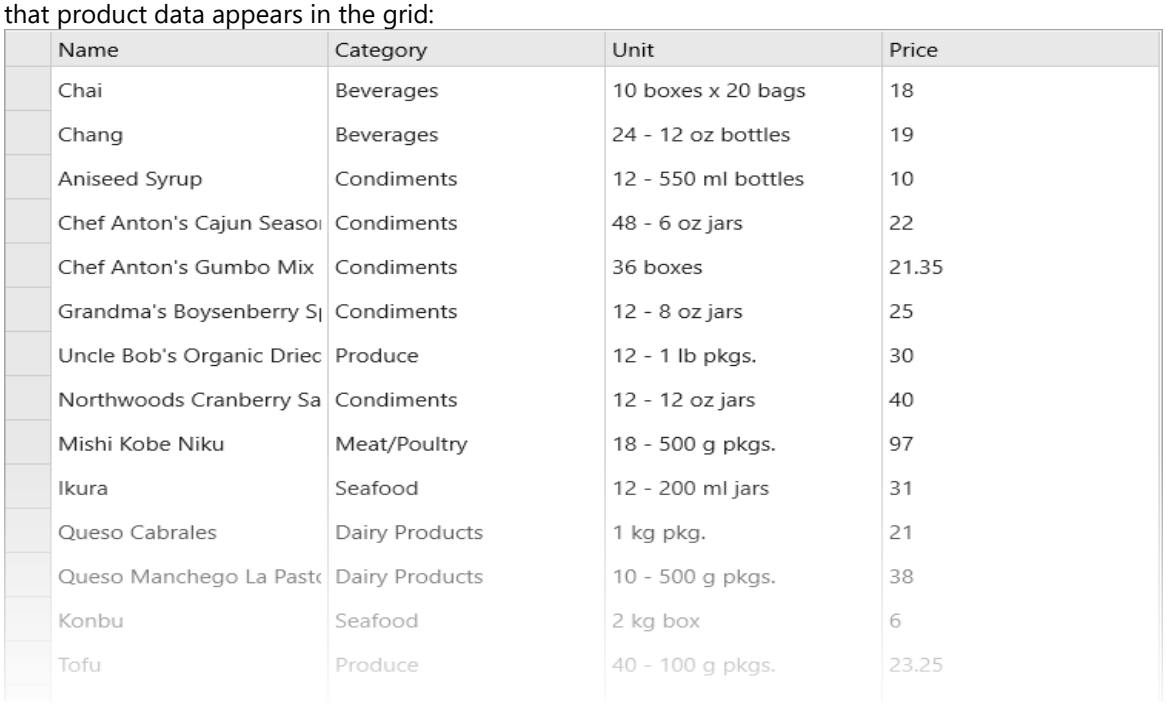
```
// Create the Product class.
public class Product
{
    public string Name { get; set; }
    public string Category { get; set; }
    public string Unit { get; set; }
    public string Price { get; set; }
}
private void LoadData()
{
    // Initialize the XML data source.
    XDocument ProductsDoc = XDocument.Load("Products.xml");
    List<Product> data = ( from Product in ProductsDoc.Descendants( "Product" )
        select new Product
        {
            Name = Product.Attribute("Name").Value,
            Category = Product.Attribute("Category").Value,
            Unit = Product.Attribute("Unit").Value,
            Price = Product.Attribute("Price").Value
         }
    ).ToList();
    // Bind the C1FlexGrid control to the data source.
    flexgrid1.ItemsSource = data;
}
```

In this step you completed adding data to your application. In the next step you'll run the application and observe run-time interactions.

Step 3 of 3: Running the C1FlexGrid Application

Now that you've created a UWP-style application and customized the application's appearance and behavior, the only thing left to do is run your application. To run your application and observe **FlexGrid for UWP**'s run-time behavior, complete the following steps:

1. From the **Debug** menu, select **Start Debugging** to view how your application will appear at run time. Notice that product data appears in the grid:



	Name	Category	Unit	Price
	Chai	Beverages	10 boxes x 20 bags	18
	Chang	Beverages	24 - 12 oz bottles	19
	Aniseed Syrup	Condiments	12 - 550 ml bottles	10
	Chef Anton's Cajun Seaso	Condiments	48 - 6 oz jars	22
	Chef Anton's Gumbo Mix	Condiments	36 boxes	21.35
	Grandma's Boysenberry Sp	Condiments	12 - 8 oz jars	25
	Uncle Bob's Organic Driec	Produce	12 - 1 lb pkgs.	30
	Northwoods Cranberry Sa	Condiments	12 - 12 oz jars	40
	Mishi Kobe Niku	Meat/Poultry	18 - 500 g pkgs.	97
	Ikura	Seafood	12 - 200 ml jars	31
	Queso Cabrales	Dairy Products	1 kg pkg.	21
	Queso Manchego La Pastc	Dairy Products	10 - 500 g pkgs.	38
	Konbu	Seafood	2 kg box	6
	Tofu	Produce	40 - 100 g pkgs.	23.25

2. Tap a column header, for example *Category*, and drag it before or after another column header, for example *Price*, to change the location of the column.
3. Similarly, tap a row header and drag it before or after another row header to change the location of the row.
4. Tap the left or right edge of a column header, for example *Name*, and drag the edge of the column to resize it.
5. Similarly, tap the upper or lower edge of a row header and drag the edge of the row to resize the row.

Congratulations!

You've completed the **FlexGrid for UWP** quick start and created a **FlexGrid for UWP** application, customized the **C1FlexGrid** control, and viewed some of the run-time capabilities of your application.

XAML Quick Reference

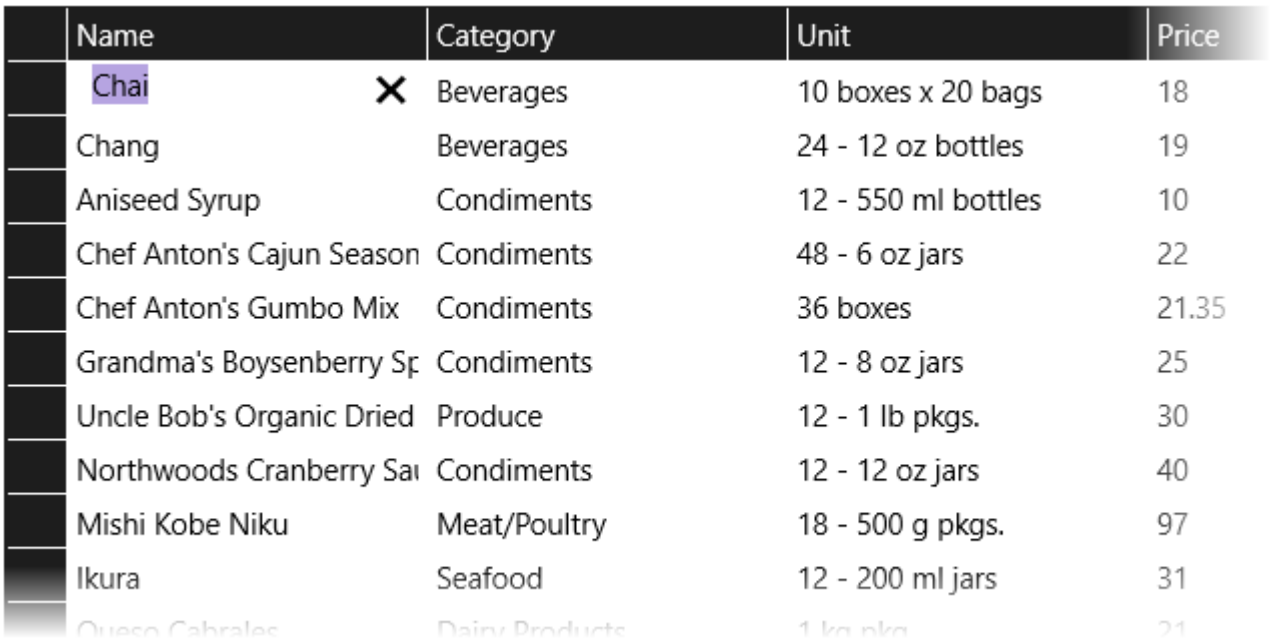
To get started developing, add a **ComponentOne** namespace declaration in the root element tag:

XAML

```
xmlns:FlexGrid="using:C1.Xaml.FlexGrid"
```

You can also add both the control and the namespace by locating the control in your Visual Studio ToolBox and double-clicking or tapping.

FlexGrid for UWP will resemble the following image:



The preceding image is taken from the Quick Start topic.

There are two ways to create Columns in a C1FlexGrid control: allowing the control to automatically generate the columns, or creating the columns yourself. One property controls this option: AutoGenerateColumns.

AutoGenerateColumns="True"

If you don't need or want to generate the columns yourself in XAML, you can set the **AutoGenerateColumns** property to True to allow the **C1FlexGrid** control to create the columns for your grid. The sample below is similar to what you will do in the **Quick Start**:

XAML

```
<Page xmlns:FlexGrid="using:C1.Xaml.FlexGrid"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:FlexGridQSTest"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:Xaml="using:C1.Xaml"
```

```
    x:Class="FlexGridQSTest.MainPage"
    mc:Ignorable="d">
    <Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        <RowDefinition/>
    </Grid.RowDefinitions>
   <FlexGrid:C1FlexGrid x:Name="flexgrid1" AutoGenerateColumns="True"
AllowResizing="Both" AllowDragging="Columns" AllowDrop="True"
ColumnHeaderForeground="White"  />
    </Grid>
</Page>
```

AutoGenerateColumns="False"

If you want to define your own columns, you can set the **AutoGenerateColumns** property to False. Your markup will resemble the following sample. In this sample, the Columns are bound:

XAML

```
<Page xmlns:FlexGrid="using:C1.Xaml.FlexGrid"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:FlexGridGroupTest"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:Xaml="using:C1.Xaml"
    x:Class="FlexGridGroupTest.MainPage"
    mc:Ignorable="d">
    <Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition/>
        </Grid.RowDefinitions>

        <FlexGrid:C1FlexGrid x:Name="c1FlexGrid1" AutoGenerateColumns="False"
ShowOutlineBar="True" BorderThickness="1" >
            <FlexGrid:C1FlexGrid.Columns>
                <FlexGrid:Column Binding="{Binding ID}" Format="g0"/>
                <FlexGrid:Column Binding="{Binding Name}" />
                <FlexGrid:Column Binding="{Binding Country}" />
                <FlexGrid:Column Binding="{Binding First}" />
                <FlexGrid:Column Binding="{Binding Last}" />
                <FlexGrid:Column Binding="{Binding Active}"/>
                <FlexGrid:Column Binding="{Binding Weight}" Format="n2"/>
                <FlexGrid:Column Binding="{Binding Hired}" Format="d"/>
                <FlexGrid:Column Binding="{Binding Father}" />
                <FlexGrid:Column Binding="{Binding Brother}" />
            </FlexGrid:C1FlexGrid.Columns>
        </FlexGrid:C1FlexGrid>
    </Grid>
```

```
</Page>
```

Features

Aggregating Data

When your grid is populated with certain types of data, such as sales data, you may want certain categories to be aggregated, such as sales data by country or product.

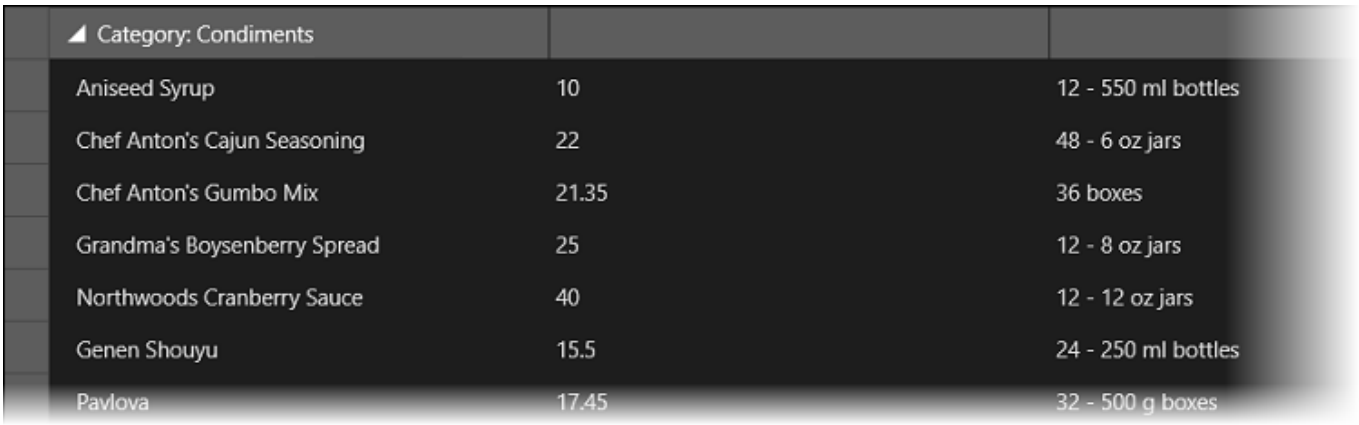
You can accomplish this easily with the **C1FlexGrid** control. You can simply set the GroupAggregate property on the columns you want to aggregate, either in XAML markup or in code. Once you set the property, **C1FlexGrid** will automatically calculate and display the aggregates in the group header rows.

This is apparent in the following grid definition:

XAML

```
<c1:C1FlexGrid x:Name="c1FlexGrid1" AutoGenerateColumns="False" ShowOutlineBar="True"
BorderThickness="1">
    <c1:C1FlexGrid.Columns>
        <c1:Column Binding="{Binding Name}" Width="*"/>
        <c1:Column Binding="{Binding Line}" />
        <c1:Column Binding="{Binding Color}" />
        <c1:Column Binding="{Binding Price}" Format="c2" Width="*"/>
        <c1:Column Binding="{Binding Weight}" Format="n2" Width="*" />
        <c1:Column Binding="{Binding Cost}" Format="c2" Width="*"/>
        <c1:Column Binding="{Binding Volume}" Width="*"/>
        <c1:Column Binding="{Binding Rating}" Format="n2" Width="*"/>
        <c1:Column Binding="{Binding Discontinued}" Width="*"/>
    </c1:C1FlexGrid.Columns>
</c1:C1FlexGrid>
```

You can group items in your **C1FlexGrid** if you set the ItemsSource property to a **C1CollectionView** object configured to group the data by product line. Follow this link to see how you can perform grouping before you aggregate your data:

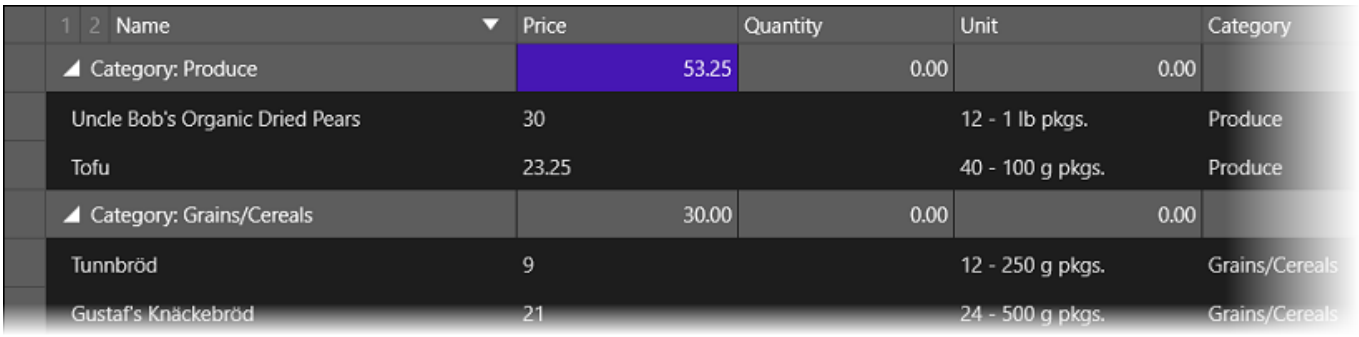


FlexGrid allows you to go further than just grouping items. If you wanted to display totals for the Price, Weight, Cost, Volume, and Rating columns, your XAML markup would resemble the following:

XAML

```
<c1:C1FlexGrid x:Name="c1FlexGrid1" AutoGenerateColumns="False" ShowOutlineBar="True"
BorderThickness="1">
    <c1:C1FlexGrid.Columns>
        <c1:Column Binding="{Binding Name}" Width="*"/>
        <c1:Column Binding="{Binding Line}" />
        <c1:Column Binding="{Binding Color}" />
        <c1:Column Binding="{Binding Price}" Format="c2" GroupAggregate="Sum"
Width="*"/>
        <c1:Column Binding="{Binding Weight}" Format="n2" GroupAggregate="Sum"
Width="*" />
        <c1:Column Binding="{Binding Cost}" Format="c2" GroupAggregate="Sum"
Width="*"/>
        <c1:Column Binding="{Binding Volume}" GroupAggregate="Sum" Width="*"/>
        <c1:Column Binding="{Binding Rating}" Format="n2" GroupAggregate="Average"
Width="*"/>
        <c1:Column Binding="{Binding Discontinued}" Width="*"/>
    </c1:C1FlexGrid.Columns>
</c1:C1FlexGrid>
```

Note that the Rating column is set to aggregate as an average. After you make this change, your **C1FlexGrid** control will resemble the following image:



Your column headers should display the aggregate values for each column.

Cell Merging

The AllowMerging property on the grid enables cell merging at the grid level. Once you have enabled merging at the grid level, use the Rows.AllowMerging and the Columns.AllowMerging properties to select the specific rows and columns that should be merged.

For example, the code below causes the grid to merge cells that contain the same names:

Visual Basic

```
' enable merging in the scrollable area
c1FlexGrid1.AllowMerging = AllowMerging.Cells

' on column "Name"
c1FlexGrid1.Columns("Name").AllowMerging = True
```

C#

```
// enable merging in the scrollable area
c1FlexGrid1.AllowMerging = AllowMerging.Cells;

// on column "Name"
c1FlexGrid1.Columns["Name"].AllowMerging = true;
```

Custom Icon

FlexGrid displays various icons during its operations such as sorting, filtering etc. These icons can be changed using various icon templates provided in the FlexGrid control. These icon templates can be accessed through following properties.

Properties	Description
SortAscendingIconTemplate	Allows you to set the template of sort icon for sorting values in ascending order.
SortDescendingIconTemplate	Allows you to set the template of sort icon for sorting values in descending order.
ExpandedAboveIconTemplate	Allows you to set the template for icon when the item group/detail is expanded above.
ExpandedBelowIconTemplate	Allows you to set the template for icon when the item group/detail is expanded below.
CollapsedIconTemplate	Allows you to set the template for group icon when the item group/detail is collapsed.
NewRowIconTemplate	Allows you to set the template of new row icon displayed in the header of a new row.

You can change the icons set by these templates either to the built-in icons provided by the FlexGrid or to your own custom image, geometric figures, font etc as an icon.

The following image displays a custom image which is set as a sort icon for sorting values in descending order.

ID	Name	CountryID	Active	First	Last
56	Zeb Myers	Italy	☐	Zeb	Myers
64	Zeb Myers	Ethiopia	☑	Zeb	Myers
87	Xavier Griswold	Congo	☑	Xavier	Griswold
11	Vic Ulam	Russia	☑	Vic	Ulam
24	Vic Stevens	Vietnam	☑	Vic	Stevens
84	Vic Richards	Egypt	☐	Vic	Richards
65	Vic Quaid	Brazil	☑	Vic	Quaid
58	Vic Krause	United Kingdo	☐	Vic	Krause
93	Vic Jammers	Russia	☐	Vic	Jammers
71	Vic Griswold	Nigeria	☐	Vic	Griswold
32	Vic Griswold	Pakistan	☐	Vic	Griswold
0	Vic Evers	Thailand	☑	Vic	Evers
23	Ulrich Krause	China	☐	Ulrich	Krause
74	Ulrich Bishop	Mexico	☑	Ulrich	Bishop
90	Ted Trask	Myanmar	☑	Ted	Trask

FlexGrid also allows you to change the appearance of the different icons used in the control using the **C1Icon** class. The C1Icon class is an abstract class that provides a series of different objects that can be used for displaying monochromatic icons which can easily be tinted and resized.

Using built-in Icons

To set the built-in icons for the abovementioned templates, you can set the following properties of the C1IconTemplate class.

Icon	Image
Edit	
Asterisk	
ArrowUp	
ArrowDown	
ChevronUp	
ChevronDown	
ChevronLeft	

ChevronRight	
TriangleNorth	
TriangleSouth	
TriangleEast	
TriangleWest	
TriangleSouthEast	
Star5	

For instance, to change the default sort ascending icon to a built-in icon, for example, TriangleNorth, use the following code:

C#

```
grid.SortAscendingIconTemplate = C1IconTemplate.TriangleNorth;
```

Using Custom Icons

FlexGrid also allows you to set your own custom image, font, or path as an icon through the respective classes.

Icon Type	Icon Class Name
Bitmap/Image	C1BitmapIcon class
Font character	C1FontIcon class
SVG	C1PathIcon class (child class of C1VectorIcon class)

For instance, to change the default sort descending icon to a custom image, use the following code:

C#

```
_flexGrid.SortDescendingIconTemplate = new C1IconTemplate(() => new C1BitmapIcon()
{
     Source = new BitmapImage() { UriSource = new Uri("ms-
appx:///Assets/arrow_down.png") },
     Width = 20,
     Height = 20
});
```

Custom Cells

If you have used any of the Microsoft data grid controls (WinForms, Silverlight, or WPF), you probably know that in order to do any significant customization you have to create custom Column objects, override several methods, then

add the custom columns to the grid using code. The C1FlexGrid control uses a very different approach. In code you can use simple Cell Factories to generate custom cells, and in XAML you can use standard Data Templates.

Custom Cells in Code will walk you through creating an ICell Factory interface.

Custom Cells in XAML gives an example of using the **Column** object's properties to define custom visual elements.

Custom Cells in Code

> If you understand the ICellFactory class, **you can click here to skip to the C1FlexGrid CellFactory implementation**.

The grid has a CellFactory class that is responsible for creating every cell shown on the grid. To create custom cells, you have to create a class that implements the ICellFactory interface and assign this class to the grid's CellFactory property. Like custom columns, custom **ICellFactory** classes can be highly specialized and application-specific, or they can be general, reusable, configurable classes. In general, custom **ICellFactory** classes are a lot simpler than custom columns since they deal directly with cells (columns, by contrast, need to deal with the columns themselves and also with the cells and other objects contained in the column).

The **ICellFactory** interface is very simple:

C#

```
public interface ICellFactory
 {
   FrameworkElement CreateCell(
     C1FlexGrid grid,
     CellType cellType,
     CellRange range);
   FrameworkElement CreateCellEditor(
     C1FlexGrid grid,
     CellType cellType,
     CellRange range)
   void DisposeCell(
     C1FlexGrid grid,
     CellType cellType,
     FrameworkElement cell);
 }
```

The **ICellFactory** interface calls three methods:

- **CreateCell**

  This method is responsible for creating FrameworkElement objects used to represent cells. The parameters include the grid that owns the cells, the type of cell to create, and the CellRange to be represented by the cells.

- **CreateCellEditor**

  This method is nearly identical to the first, but it creates the cell in edit mode.

- **DisposeCell**

  This method is called after the cell has been removed from the grid. It gives the caller a chance to dispose of

any resources associated with the cell object.

When using custom cells, it is important to understand that grid cells are transient. Cells are constantly created and destroyed as the user scrolls, sorts, or selects ranges on the grid. This process is known as virtualization and is quite common in Windows Store applications. Without virtualization, a grid would typically have to create several thousand visual elements at the same time, which would ruin performance.

Implementation

Implementing custom **ICellFactory** classes is fairly easy because you can inherit from the default **CellFactory** class included with the **C1FlexGrid**.

The default **CellFactory** class was designed to be extensible, so you can let it handle all the details of cell creation and customize only what you need. In the following code, conditional formatting is applied to turn low values red and high values green:

C#

```
c1FlexGrid1.CellFactory = new ConditionalFormattingFactory();

class ConditionalFormattingFactory : CellFactory
{
    // create brushes used to indicate low and high values
    static Brush _brLowValue = new SolidColorBrush(Colors.Red);
    static Brush _brHighValue = new SolidColorBrush(Colors.Green);

    // overridden to apply the custom brushes based on the cell value
    public override void ApplyCellStyles(C1FlexGrid grid, CellType cellType,
CellRange range, Border bdr)
    {
        // we are interested only in data cells (no headers)
        if (cellType == CellType.Cell)
        {
            // we are interested in double values only
            var col = grid.Columns[range.Column];
            if (col.DataType == typeof(double))
            {
                // get the cell value
                var value = (double)grid[range.Row, col];

                // apply formatting if value is out of range
                if (value < 100 || value > 1000)
                {
                    var tb = bdr.Child as TextBlock;
                    if (tb != null)
                    {
                        bdr.Background = value < 100 ? _brLowValue : _brHighValue;
                    }
                }
            }
        }
    }
}
```

Custom Cells in XAML

If you prefer to create custom cells in XAML instead of writing code, you can do that as well. The **C1FlexGrid** Column object has CellTemplate and CellEditingTemplate properties that you can use to specify the visual elements responsible for showing and editing cells in the column.

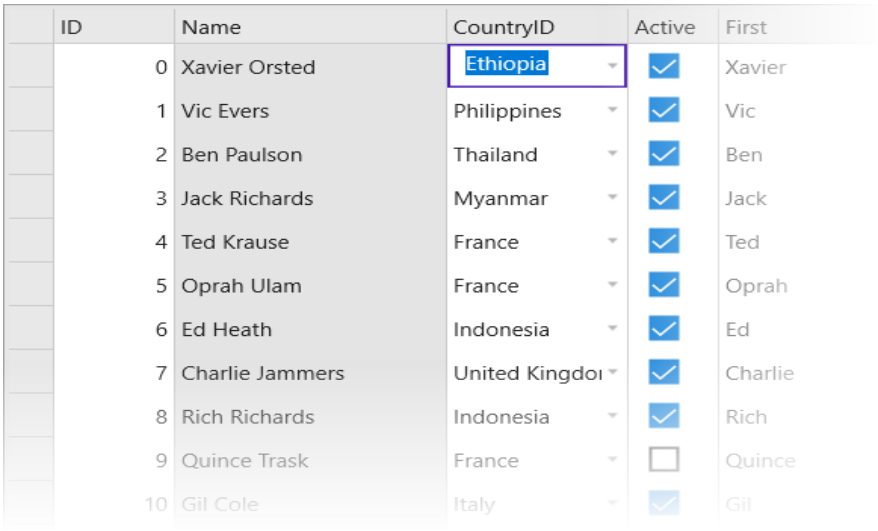
For example, the XAML code below defines custom visual elements used to show and edit values in a column. Cells in that column are shown as green, bold, center-aligned text, and edited using a textbox that has an edit icon next to it:

Markup

```
<c1:C1FlexGrid x:Name="_fgTemplated">
   <c1:C1FlexGrid.Columns>
     <!-- add a templated column -->
     <c1:Column ColumnName="_colTemplated" Header="Template" Width="200">
       <!-- template for cells in display mode -->
       <c1:Column.CellTemplate>
         <DataTemplate>
           <TextBlock Text="{Binding Name}"
            Foreground="Green" FontWeight="Bold"
            VerticalAlignment="Center"/>
         </DataTemplate>
       </c1:Column.CellTemplate>
       <!-- template for cells in edit mode -->
       <c1:Column.CellEditingTemplate>
         <DataTemplate>
           <Grid>
             <Grid.ColumnDefinitions>
               <ColumnDefinition Width="Auto" />
               <ColumnDefinition Width="*" />
             </Grid.ColumnDefinitions>
             <Image Source="edit_icon.png" Grid.Column="0" />
             <TextBox Text="{Binding Name, Mode=TwoWay}" Grid.Column="1" />
           </Grid>
         </DataTemplate>
       </c1:Column.CellEditingTemplate>
     </c1:Column>
   </c1:C1FlexGrid.Columns>
 </c1:C1FlexGrid>
```

Editing Features

C1FlexGrid supports cell editing through an easy-to-use interface for the Windows Store. The editing is not inline as it would be in a Web or desktop application, but rather in a separate pop-up perfect for the touch interface. Double-click or tap on the cell you wish to edit to begin editing. The cell will appear as in the following image:



Custom Editors

The **C1FlexGrid** provides two single built-in editors: a checkbox for Boolean values and the C1FlexComboBox that extends a regular TextBox with auto-complete capability and list selection as described above.

You can create and use your own editors using the same mechanisms used to create custom cells that were described earlier in this document:

- Implement a custom CellFactory class and override the CreateCellEditor method to create and bind your editor to the underlying data value.
- Use XAML to specify a CellEditingTemplate for the columns that need the custom editors.

Auto-complete and Mapped Columns

Auto-complete and mapped columns are implemented with a built-in class called **ColumnValueConverter**. This class deals with two common binding scenarios:

Auto-complete exclusive mode (ListBox-style editing)
Columns that can only take on a few specific values. For example, you have a "Country" column of type string and a list of country names. Users should select a country from the list, and not be allowed to enter any countries not on the list.

You can handle this scenario with two lines of code:

C#

```
var c = flexgrid1.Columns["Country"];
c.ValueConverter = new ColumnValueConverter(GetCountryNames(), true);
```

The first parameter in the **ColumnValueConverter** constructor provides the list of valid values. The second parameter determines whether users should be allowed to enter values that are not on the list (in this example they are not).

Auto-complete non-exclusive mode (ComboBox-style editing)
Columns that have a few common values, but may take on other values as well. For example, you have a "Country" column of type string and want to provide a list of common country names that users can select easily. But in this case users should also be allowed to type values that are not on the list.

You can also handle this scenario with two lines of code:

C#

```
var c = flexgrid1.Columns["Country"];
c.ValueConverter = new ColumnValueConverter(GetCountryNames(), false);
```

As before, the first parameter in the **ColumnValueConverter** constructor provides the list of valid values. The second parameter in this case determines that the list is not exclusive, so users are allowed to enter values that are not on the list.

Data-Mapped Columns

Data-mapped columns contain keys instead of actual values. For example, the column may contain an integer that represents a country ID, but users should see and edit the corresponding country name instead.

This scenario requires a few lines of code:

C#

```
// map countryID column so it shows country names instead of their IDs
Dictionary<int, string> dct = new Dictionary<int, string>();
foreach (var country in Customer.GetCountries())
{
    dct[dct.Count] = country;
}
col = _flexEdit.Columns["CountryID"];
col.ValueConverter = new ColumnValueConverter(dct);
col.HorizontalAlignment = HorizontalAlignment.Left;
col.Width = new GridLength(120)
```

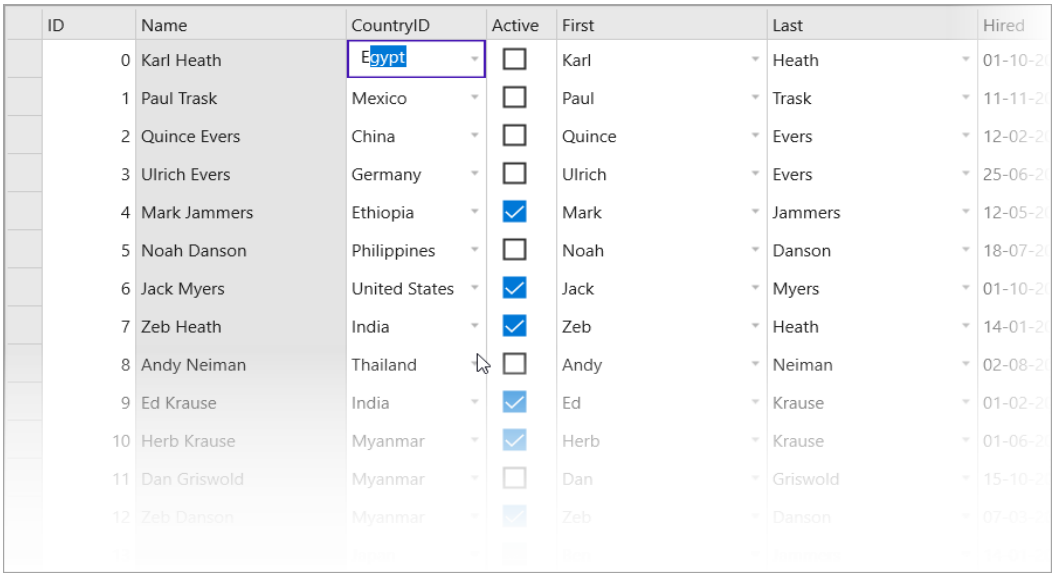
The code starts by building a dictionary that maps country ID values (integers) to country names (strings).

It then uses the dictionary to build a **ColumnValueConverter** and assigns the converter to the column's **ValueConverter** property.

The user will be able to select any countries present in the dictionary, and will not be able to enter any unmapped values.

Finally, the code sets the column alignment to left. Since the column actually contains integer values, it is aligned to the right by default. But since we are now displaying names, left alignment is a better choice here.

The image below shows the appearance of the editor while selecting a value from a list. Notice how the editor supports smart auto-completion, so as the user types “F”, the cell automatically replaces the text with "France".



Configuring Editors

Whether you are using built-in or custom editors, you can take advantage of the **PrepareCellForEdit** event to configure the editor before it is activated. For example, the code below changes the editor to show selections as yellow on blue:

C#

```
// hook up event handler
_grid.PrepareCellForEdit += _grid_PrepareCellForEdit;
// customize editor by changing the appearance of the selection
void _grid_PrepareCellForEdit(object sender, CellEditEventArgs e)
{
  var b = e.Editor as Border;
  var tb = b.Child as TextBox;
  tb.SelectionBackground = new SolidColorBrush(Colors.Blue);
  tb.SelectionForeground = new SolidColorBrush(Colors.Yellow);
}
```

Filtering Data

The **C1CollectionView** interface also includes support for filtering data through its **Filter** property. The **Filter** property specifies a method that is called for each item in the collection. If the method returns true, then the item is included in the view. If the method returns false, then the item is filtered out of view. This type of method is a predicate method.

The **FlexGridSamples** solution included with this Studio's samples includes a **SearchBox** control that consists of a **TextBox** control where the user types a value to search for and a timer. The timer provides a small delay to allow users to type the values to search for without re-applying the filter after each character.

When the user stops typing, the timer elapses and applies the filter using this code:

C#

```
bool Filter(object item)
{
```

```
    // get search text
    var srch = _txtSearch.Text;
    // no text? show all items
    if (string.IsNullOrEmpty(srch))
    {
        return true;
    }
    // show items that contain the text in any of the specified properties
    foreach (PropertyInfo pi in _propertyInfo)
    {
        var value = pi.GetValue(item, null) as string;
        if (value != null && value.IndexOf(srch, StringComparison.OrdinalIgnoreCase)
> -1)
        {
            return true;
        }
    }
    // exclude this item...
    return false;
}
```

Note how the code sets the value of the **Filter** property using a boolean function.

This function takes an item as a parameter, gets the value of the specified properties for the object, and returns true if any of the object's properties contain the string being searched for.

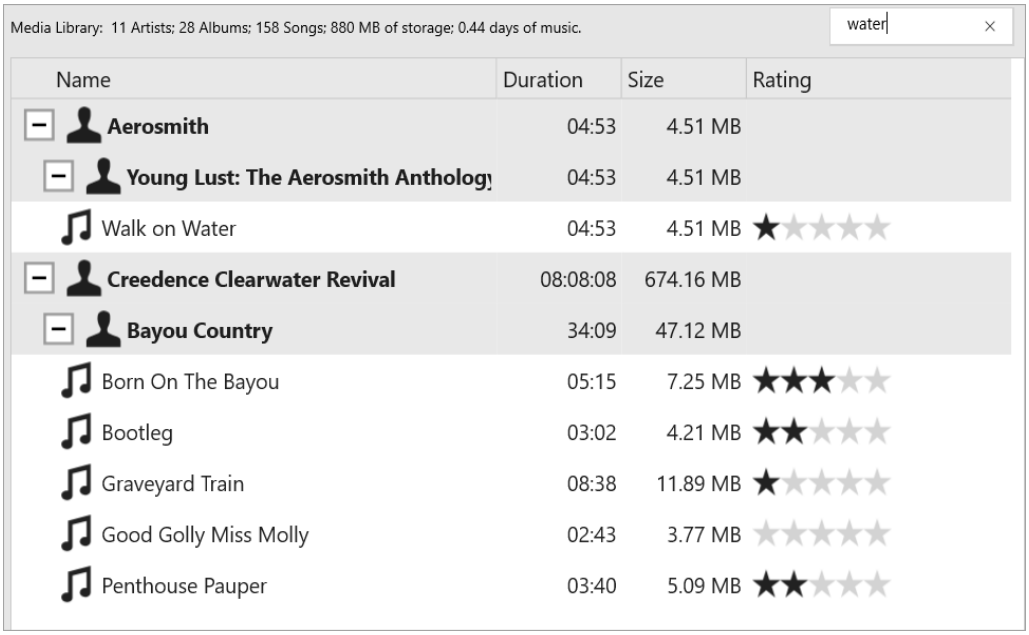
For example, if the objects were of type "Song" and the properties specified were "Title", "Album", and "Artist", then the function would return true if the string being searched for were found in the song's title, album, or artist. This is a powerful and easy-to-use search mechanism similar to the one used in Apple's iTunes application.

As soon as the filter is applied, the grid (and any other controls bound to the **IC1CollectionView** object) will reflect the result of the filter by showing only the items selected by the filter. The following code is used to filter songs using Filter property of IC1CollectionView:

C#

```
var view = View as C1.Xaml.IC1CollectionView;
if (view != null)
{
  view.Filter = Filter;
  view.Refresh();
}
```

Note that filtering and grouping work perfectly well together. The image below shows a very large song list with a filter applied to it:



The image shown was taken when the filter was set to the word "Water". The filter looks for matches in all fields (song, album, artist), so all "Creedence Clearwater Revival" songs are automatically included.

Search Data using Full Text Filter

Searching a grid with huge data source can be a tedious task even in a grid that supports column filtering and sorting. To make this task easier, FlexGrid supports searching the entire data source connected to the grid through full text filtering. To apply full text filtering, you need to use BindingPaths property of the FullTextFilterCondition class, which specifies the data column(s) that will participate in the full text search filtering. For searching, you can use an editor control to act as a search box where user provides the input for obtaining the search results. You need to handle an event depending on whether you want to search as the user types in or once the input has completed. In that event, you can invoke ApplyFullTextFilter method of the **C1CollectionView** class which accepts input text and instance of the FullTextFilterCondition class as its parameters.

In addition, the **FullTextFilterCondition** class provides conditions to perform full text search filtering in the control. These conditions can be applied to the user input for customizing the filtering behavior using the MatchCase, MatchWholeWord and TreatSpacesAsAndOperator properties.

In the following example, we used a TextBox as a search box and invoked the ApplyFullTextFilter method in TextChanged event of the TextBox to implement full text search filtering in FlexGrid.

1. Add C1FlexGrid and TextBox control to the XAML designer and add five columns to the grid to display data using the following code:

XAML

```
<Grid>
    <TextBox x:Name="filterTextBox" HorizontalAlignment="Left" VerticalAlignment="Top"
             TextChanged="filterTextBox_TextChanged" Margin="0,20,0,0" Width="250"/>
    <Custom:C1FlexGrid x:Name="c1FlexGrid1" HorizontalAlignment="Left"
                       AutoGenerateColumns="false" ColumnHeaderForeground="Black"
                       Margin="0,60,0,0">
        <Custom:C1FlexGrid.Columns>
            <Custom:Column Binding="{Binding ID, Mode=TwoWay}"/>
            <Custom:Column Binding="{Binding Name, Mode=TwoWay}" Width="*"/>
            <Custom:Column Binding="{Binding Email, Mode=TwoWay}" Width="*" />
            <Custom:Column Binding="{Binding OrderDate, Mode=TwoWay}" Width="*"/>
            <Custom:Column Binding="{Binding OrderTotal, Mode=TwoWay}" Width="*"/>
        </Custom:C1FlexGrid.Columns>
    </Custom:C1FlexGrid>
```

```
</Grid>
```

2. Switch to the Code view and create a class, Customers, to add data in the FlexGrid.

Visual Basic

```
Public Class Customer
    Private _id As Integer
    Private _name, _email As String
    Private _OrderDate As DateTime
    Private _orderTotal As Double

    Shared _rnd As Random = New Random()
    Shared _firstNames As String() =
"Andy|Ben|Charlie|Dan|Ed|Fred|Herb|Elena|Stefan|Gina".Split("|"c)
    Shared _lastNames As String() =
"Ambers|Bishop|Cole|Danson|Evers|Frommer|Heath|Salvatore|Spencer|Rodriguez".Split("|"c)

    Shared _emailServers As String() = "gmail|yahoo|outlook|aol".Split("|"c)

    Public Sub New()
        Me.New(_rnd.[Next]())
    End Sub

    Public Sub New(ByVal id As Integer)
        ID = id
        Name = GetName()
        Email = String.Format("{0}@{1}.com", (Name.Substring(0, 1)).ToLower(),
GetString(_emailServers))
        OrderDate = DateTime.Today.AddDays(-_rnd.[Next](1, 365)).AddHours(_rnd.[Next]
(0, 24)).AddMinutes(
                                        _rnd.[Next](0, 60))
        OrderTotal = Math.Round(_rnd.NextDouble() * 10000.00, 2)
    End Sub

    Public Property ID As Integer
        Get
            Return _id
        End Get
        Set(ByVal value As Integer)

            If value <> _id Then
                _id = value
            End If
        End Set
    End Property

    Public Property Name As String
        Get
            Return _name
        End Get
        Set(ByVal value As String)

            If value <> _name Then
                _name = value
            End If
        End Set
    End Property

    Public Property Email As String
        Get
            Return _email
        End Get
        Set(ByVal value As String)

            If value <> _email Then
                _email = value
```

```
            End If
        End Set
    End Property

    Public Property OrderDate As DateTime
        Get
            Return _OrderDate
        End Get
        Set(ByVal value As DateTime)

            If value <> _OrderDate Then
                _OrderDate = value
            End If
        End Set
    End Property

    Public Property OrderTotal As Double
        Get
            Return _orderTotal
        End Get
        Set(ByVal value As Double)

            If value <> _orderTotal Then
                _orderTotal = value
            End If
        End Set
    End Property

    Private Shared Function GetString(ByVal arr As String()) As String
        Return arr(_rnd.[Next](arr.Length))
    End Function

    Private Shared Function GetName() As String
        Return String.Format("{0} {1}", GetString(_firstNames), GetString(_lastNames))
    End Function

    Public Shared Function GetCustomerList(ByVal count As Integer) As
ObservableCollection(Of Customer)
        Dim list = New ObservableCollection(Of Customer)()

        For i As Integer = 0 To count - 1
            list.Add(New Customer(i))
        Next

        Return list
    End Function
End Class
```

C#

```
public class Customer
{
    //fields
    int _id;
    string _name, _email;
    DateTime _OrderDate;
    double _orderTotal;

    //data generators
    static Random _rnd = new Random();
    static string[] _firstNames =
"Andy|Ben|Charlie|Dan|Eddy|Fred|Herb|Elena|Stefan|Gina".Split('|');
    static string[] _lastNames =
"Ambers|Bishop|Cole|Danson|Evers|Frommer|Heath|Salvatore|Spencer|Rodriguez".Split('|');

    static string[] _emailServers = "gmail|yahoo|outlook|aol".Split('|');
```

```
public Customer()
    : this(_rnd.Next())
{
}
public Customer(int id)
{
    ID = id;
    Name = GetName();
    Email = string.Format("{0}@{1}.com", (Name.Substring(0, 3)).ToLower(),
            GetString(_emailServers));
    OrderDate = DateTime.Today.AddDays(-_rnd.Next(1, 365)).AddHours(
                _rnd.Next(0, 24)).AddMinutes(_rnd.Next(0, 60));
    OrderTotal = Math.Round(_rnd.NextDouble() * 10000.00, 2);
}
//Object model
public int ID
{
    get { return _id; }
    set
    {
        if (value != _id)
        {
            _id = value;
        }
    }
}
public string Name
{
    get { return _name; }
    set
    {
        if (value != _name)
        {
            _name = value;
        }
    }
}
public string Email
{
    get { return _email; }
    set
    {
        if (value != _email)
        {
            _email = value;
        }
    }
}
public DateTime OrderDate
{
    get { return _OrderDate; }
    set
    {
        if (value != _OrderDate)
        {
            _OrderDate = value;
        }
    }
}

public double OrderTotal
{
    get { return _orderTotal; }
    set
    {
```

```
            if (value != _orderTotal)
            {
                _orderTotal = value;
            }
        }
    }

    // ** utilities
    static string GetString(string[] arr)
    {
        return arr[_rnd.Next(arr.Length)];
    }
    static string GetName()
    {
        return string.Format("{0} {1}", GetString(_firstNames),
GetString(_lastNames));
    }
    // ** static list provider
    public static ObservableCollection<Customer> GetCustomerList(int count)
    {
        var list = new ObservableCollection<Customer>();
        for (int i = 0; i < count; i++)
        {
            list.Add(new Customer(i));
        }
        return list;
    }
}
```

3. Add the following code to filter data using full text filtering in the FlexGrid control.

Visual Basic

```
Public NotInheritable Partial Class MainPage
    Inherits Page

    Private cv As C1CollectionView
    Private FilterCondition As FullTextFilterCondition

    Public Sub New()
        Me.InitializeComponent()
        Dim list = New ObservableCollection(Of Customer)()

        For i As Integer = 0 To 100 - 1
            list.Add(New Customer())
        Next

        cv = New C1CollectionView(list)
        c1FlexGrid1.ItemsSource = cv
        FilterCondition = New FullTextFilterCondition()
        FilterCondition.BindingPaths = New List(Of String)() From {
            "ID",
            "Name",
            "OrderDate",
            "OrderTotal"
        }
        FilterCondition.MatchCase = True
    End Sub

    Private Sub filterTextBox_TextChanged(ByVal sender As Object, ByVal e As
TextChangedEventArgs)
        cv.ApplyFullTextFilter(filterTextBox.Text, FilterCondition)
    End Sub
End Class
```

C#

```
public sealed partial class MainPage : Page
{
    C1CollectionView cv;
    FullTextFilterCondition FilterCondition;
    public MainPage()
    {
        this.InitializeComponent();

        var list = new ObservableCollection<Customer>();
        for (int i = 0; i < 100; i++)
            list.Add(new Customer());

        // create a C1CollectionView from the list
        cv = new C1CollectionView(list);

        c1FlexGrid1.ItemsSource = cv;

        FilterCondition = new FullTextFilterCondition();
        FilterCondition.BindingPaths = new List<string>() { "ID", "Name",
                    "OrderDate", "OrderTotal" };

        // Update FullTextFilterCondition
        FilterCondition.MatchCase = true;
    }
    void filterTextBox_TextChanged(object sender, TextChangedEventArgs e)
    {
        // Apply FullTextFilter
        cv.ApplyFullTextFilter(filterTextBox.Text, FilterCondition);
    }
}
```

Grouping Data

This topic uses code that can be found in the **FlexGridDemo** page in the **FlexGridSamples** project. The project also contains the **Customer.cs** code file. You can find this project installed on your machine here: **Documents\ComponentOne Samples\UWP\C1.UWP.FlexGrid\CS\FlexGridSamples**

Data grouping in the C1FlexGrid control is accomplished through the **C1CollectionView** class. For example, in a grid containing a list of customers, you can add a little code to your **C1CollectionView** to group customers by country:

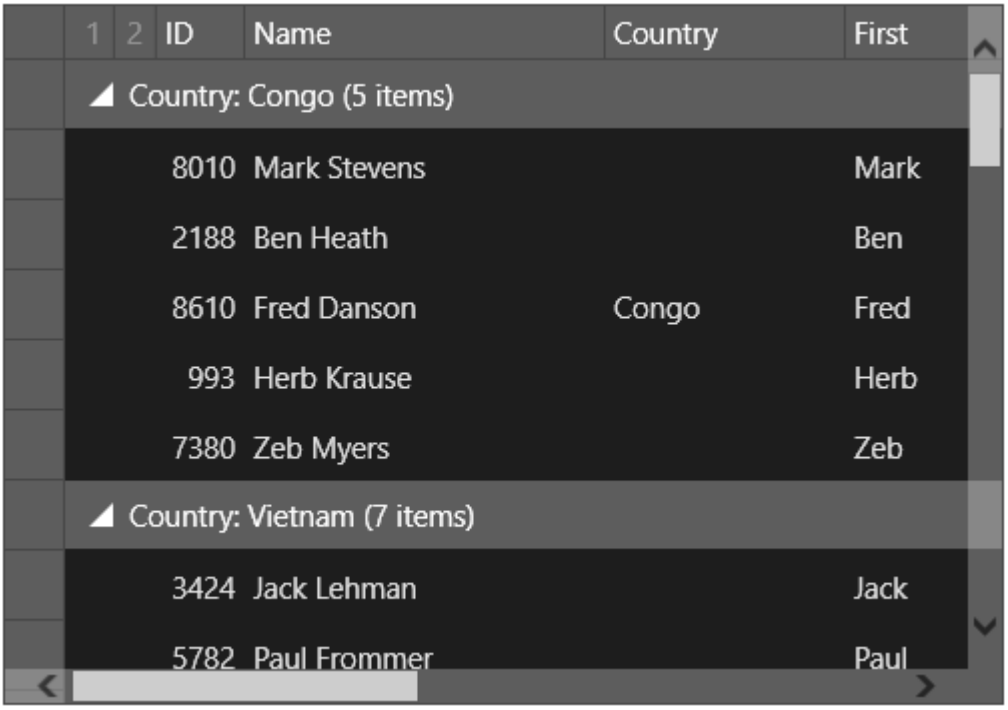
C#

```
C1.Xaml.C1CollectionView view;

            // create an observable list of customers
            var list = new System.Collections.ObjectModel.ObservableCollection();
            for (int i = 0; i < 100; i++)
                list.Add(new Customer());
            // create a C1CollectionView from the list
            view = new C1.Xaml.C1CollectionView(list);
            // group customers by country
            view.GroupDescriptions.Add(new
C1.Xaml.PropertyGroupDescription("Country"));

            c1FlexGrid1.ItemsSource = view;
```

Running your project will give you a FlexGrid that resembles the following image:



The data items are grouped by country. Users can click or tap the icon next to the country name to collapse or expand the country category. Users can also control how the data is sorted by clicking or tapping the column name.

If you wanted to change the grouping, all you have to do is change the column name in the following line of code:

C#

```
view.GroupDescriptions.Add(new
    C1.Xaml.PropertyGroupDescription("Country"));
```

If you want to sort by the customer's last name, all you have to do is change "Country" to "Last":

C#

```
// create an observable list of customers
var list = new System.Collections.ObjectModel.ObservableCollection<Customer>();
for (int i = 0; i < 100; i++)
    list.Add(new Customer());
// create a C1CollectionView from the list
// (it supports sorting, filtering, and grouping)
view = new C1.Xaml.C1CollectionView(list);
// group customers by country
view.GroupDescriptions.Add(new C1.Xaml.PropertyGroupDescription("Last"));
```

When you change the grouping, your FlexGrid control will resemble the following image:

1 2	ID	Name	Country	First
Last: Ambers (4 items)				
	3348	Ben Ambers	Russia	Ben
	9407	Jack Ambers	Thailand	Jack
	8547	Steve Ambers	Ethiopia	Steve
	7186	Mark Ambers	United States	Mark
Last: Bishop (6 items)				
	3860	Zeb Bishop	Nigeria	Zeb
	3889	Ben Bishop	Thailand	Ben
	8151	Noah Bishop	Ethiopia	Noah

Multi-Cell Row and Column Headers

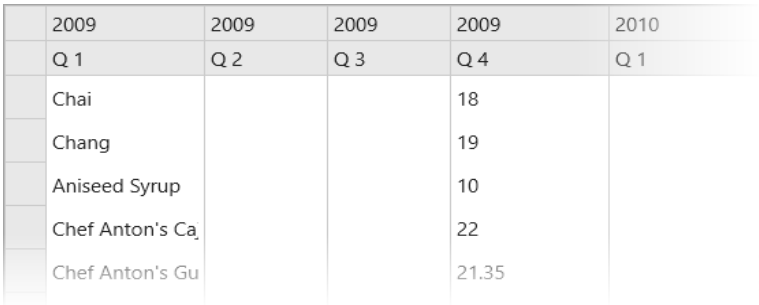
Most grids support row and column header cells, used to display a header over columns and to indicate the status of the row that contains the selection.

The C1FlexGrid takes this concept a little further and supports multi-cell headers. You could for example have two rows of column headers, one showing the year and one showing the quarter. Here's some code that shows how you could set that up:

C#

```
// add an extra column header row
var ch = flexgrid1.ColumnHeaders;
 ch.Rows.Add(new Row());
 // populate the header rows
 for (int c = 0; c < ch.Columns.Count; c++)
 {
   ch[0, c] = 2009 + c / 4; // row 0: year
   ch[1, c] = string.Format("Q {0}", c % 4 + 1); // row 1: quarter
 }
```

This code would produce a grid that looks like this:



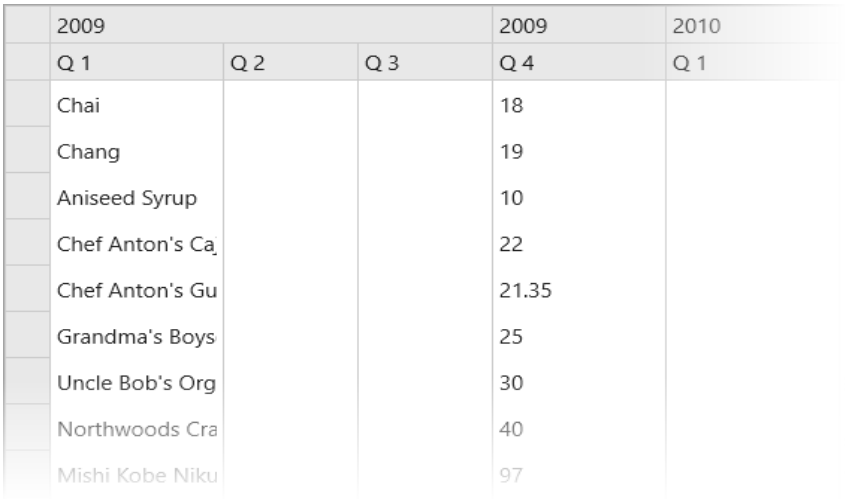
	2009	2009	2009	2009	2010
	Q 1	Q 2	Q 3	Q 4	Q 1
	Chai			18	
	Chang			19	
	Aniseed Syrup			10	
	Chef Anton's Ca			22	
	Chef Anton's Gu			21.35	

Notice the two rows used to display the column headers. You could achieve a similar effect in a traditional grid using column headers with line breaks. The difference becomes apparent when we add cell merging to the top fixed row, so columns that refer to the same year are automatically merged. It only takes two lines of code:

C#

```
// add an extra column header row
var ch = flexgrid1.ColumnHeaders;
ch.Rows.Add(new Row());
 // populate the header rows
 for (int c = 0; c < ch.Columns.Count; c++)
 {
   ch[0, c] = 2009 + c / 4; // row 0: year
   ch[1, c] = string.Format("Q {0}", c % 4 + 1); // row 1: quarter
 }
 // merge the top fixed row
 flexgrid1.AllowMerging = AllowMerging.All;
 ch.Rows[0].AllowMerging = true;
```

The result is displayed in the image below:



Selection Modes

In addition to showing data in a tabular format, most grid controls allow users to select parts of the data using the mouse and the keyboard.

The C1FlexGrid has a rich selection model controlled by the SelectionMode property. The following selection modes are available in the SelectionMode enumeration:

- **Cell:**
  Selection corresponds to a single cell.
- **CellRange:**
  Selection corresponds to a cell range (block of adjacent cells).
- **Row:**
  Selection corresponds to a single whole row.
- **RowRange:**
  Selection corresponds to a set of contiguous rows.
- **ListBox:**
  Selection corresponds to an arbitrary set of rows (not necessarily contiguous).

The default **SelectionMode** is **CellRange**, which provides a familiar Excel-like selection behavior. The row-based options are also useful in scenarios where it makes sense to select whole data items instead of individual grid cells.

Regardless of the selection mode, the grid exposes the current selection with the Selection property. This property gets or sets the current selection as a **CellRange** object.

Monitoring the Selection

Whenever the selection changes, either as a result of user actions or code, the grid fires the SelectionChanged event, which allows you to react to the new selection.

For example, the code below monitors the selection and outputs information to the console when the selection changes:

C#

```
void _flex_SelectionChanged(object sender, CellRangeEventArgs e)
{
  CellRange sel = _flex.Selection;
  Console.WriteLine("selection: {0},{1} - {2},{3}",
      sel.Row, sel.Column, sel.Row2, sel.Column2);
  Console.WriteLine("selection content: {0}",
      GetClipString(_flex, sel));
}
static string GetClipString(C1FlexGrid fg, CellRange sel)
{
  var sb = new System.Text.StringBuilder();
  for (int r = sel.TopRow; r <= sel.BottomRow; r++)
  {
    for (int c = sel.LeftColumn; c <= sel.RightColumn; c++)
    {
      sb.AppendFormat("{0}\t", fg[r, c].ToString());
    }
     sb.AppendLine();
  }
  return sb.ToString();
}
```

Whenever the selection changes, the code lists the coordinates of the CellRange that represents the current selection. It also outputs the content of the selected range using a GetClipString method that loops through the selected items and retrieves the content of each cell in the selection using the grid's indexer described earlier in this document.

Notice that the for loops in the **GetClipString** method use the **CellRange's** TopRow, BottomRow, LeftColumn, and RightColumn properties instead of the Row, Row2, Column, and Column2 properties. This is necessary because **Row** may be greater or smaller than **Row2**, depending on how the user performed the selection (dragging the mouse up or down while selecting).

You can easily extract a lot of useful information from the **Selection** using the RowCollection.GetDataItems method. This method returns a collection of data items associated with a **CellRange**. Once you have this collection, you can use LINQ to extract and summarize information about the selected items.

For example, consider this alternate implementation of the SelectionChanged event for a grid bound to a collection of Customer objects:

C#

```
void _flex_SelectionChanged(object sender, CellRangeEventArgs e)
{
  // get customers in the selected range
  var customers =
    _flex.Rows.GetDataItems(_flex.Selection).OfType<Customer>();
  // use LINQ to extract information from the selected customers
  _lblSelState.Text = string.Format(
    "{0} items selected, {1} active, total weight: {2:n2}",
    customers.Count(),
    (from c in customers where c.Active select c).Count(),
    (from c in customers select c.Weight).Sum());
}
```

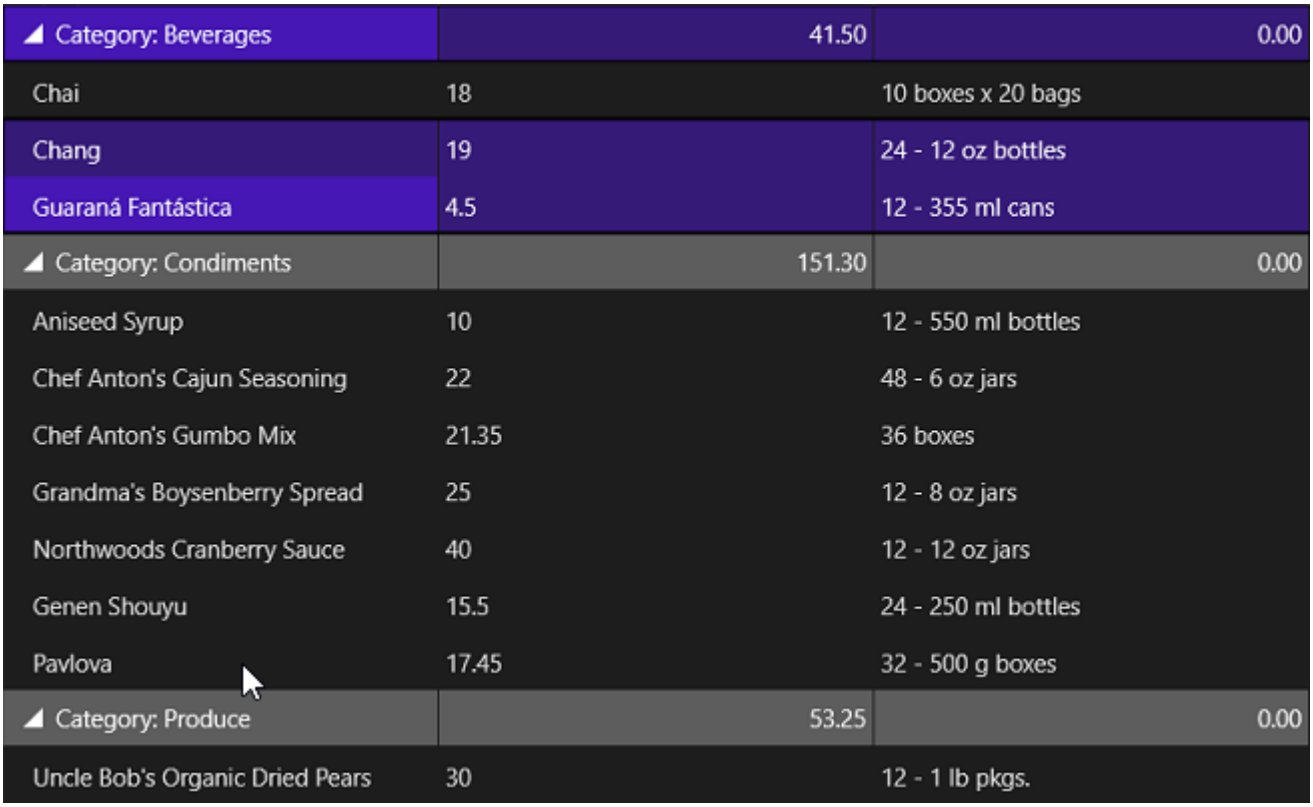
Notice how the code uses the **OfType** operator to cast the selected data items to type **Customer**. Once that is done, the code uses LINQ to get a total count, a count of "active" customers, and the total weight of the customers in the selection. LINQ is the perfect tool for this type of job. It is flexible, expressive, compact, and efficient.

Customizing the Selection Display

The C1FlexGrid includes two features that allow you to customize the way in which the selection is highlighted for the user:

- **Excel-Style Marquee**: If you set the ShowMarquee property to true, the grid will automatically draw a rectangle around the selection, making it extremely easy to see. By default, the marquee is a two-pixel thick black rectangle, but you can customize it using the Marquee property.
- **Selected Cell Headers**: If you assign custom brush objects to the grid's ColumnHeaderSelectedBackground and RowHeaderSelectedBackground properties, the grid will highlight the headers that correspond to selected cells, making it easy for users to see which rows and columns contain the selection.

Together, these properties make it easy to implement grids that have the familiar Excel look and feel. The image below shows an example:



Category: Beverages	41.50	0.00
Chai	18	10 boxes x 20 bags
Chang	19	24 - 12 oz bottles
Guaraná Fantástica	4.5	12 - 355 ml cans
Category: Condiments	151.30	0.00
Aniseed Syrup	10	12 - 550 ml bottles
Chef Anton's Cajun Seasoning	22	48 - 6 oz jars
Chef Anton's Gumbo Mix	21.35	36 boxes
Grandma's Boysenberry Spread	25	12 - 8 oz jars
Northwoods Cranberry Sauce	40	12 - 12 oz jars
Genen Shouyu	15.5	24 - 250 ml bottles
Pavlova	17.45	32 - 500 g boxes
Category: Produce	53.25	0.00
Uncle Bob's Organic Dried Pears	30	12 - 1 lb pkgs.

Unbound Mode

The C1FlexGrid was designed to work with **C1CollectionView** data sources, and to take full advantage of the features it provides.

But it can also be used in unbound mode. If you simply add rows and columns to the grid, you can get or set values in the cells using the familiar indexing notation shown below:

C#

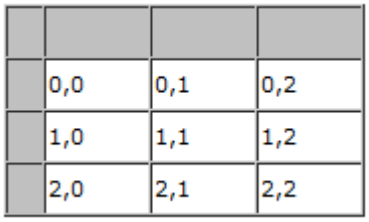
```
// add rows/columns to the unbound grid
for (int i = 0; i < 12; i++) // three years, four quarters per year
{
    fg.Columns.Add(new Column());
}
for (int i = 0; i < 500; i++)
{
    fg.Rows.Add(new Row());
}
// populate the unbound grid with some stuff
for (int r = 0; r < fg.Rows.Count; r++)
{
    for (int c = 0; c < fg.Columns.Count; c++)
    {
        fg[r, c] = string.Format("cell [{0},{1}]", r, c);
    }
}
```

The indexing notation should also be familiar to **C1FlexGrid** users. It is the same notation implemented by the

WinForms version of the control. You can specify cells by the row and column indices, by row index and column name, or by row index and Column object.

The indexing notation works in bound and unbound modes. In bound mode, the data is retrieved or applied to the items in the data source. In unbound mode, the data is stored internally by the grid.

> In the UWP Edition version of the control, row and column indexing does not include fixed rows and columns in the count. You can see this in the image below:



This type of notation makes indexing easier because the indices match the index of the data items (row zero contains item zero) and the column count matches the number of properties being displayed.

With this type of notation, though, you'll need the RowHeaders and ColumnHeaders properties to access the content of the fixed cells.

For example, you could use this code to customize the row headers:

C#

```
// set unbound column headers
var ch = fg.ColumnHeaders;
ch.Rows.Add(new Row()); // one header row for years, one for quarters
for (int c = 0; c < ch.Columns.Count; c++)
{
    ch[0, c] = 2009 + c / 4; // year
    ch[1, c] = string.Format("Q {0}", c % 4 + 1); // quarter
}
// allow merging the first fixed row
ch.Rows[0].AllowMerging = true;
// set unbound row headers
var rh = fg.RowHeaders;
rh.Columns.Add(new Column());
for (int c = 0; c < rh.Columns.Count; c++)
{
    rh.Columns[c].Width = new GridLength(60);
    for (int r = 0; r < rh.Rows.Count; r++)
    {
        rh[r, c] = string.Format("hdr {0},{1}", c == 0 ? r / 2 : r, c);
    }
}
```

You can customize and populate the row and column headers using the same object model and techniques you use when working with the content area of the grid.

Performance Optimization

It's a common requirement to display large data sets in the C1FlexGrid control. The most common technique to handle large data sets in UWP is called UI virtualization. With UI virtualization, items are not generated until they are brought into view. This means that only the number of cells visible in a viewport (size of window or client area) will be generated. By default, the **C1FlexGrid** control supports UI virtualization for both vertical and horizontal scrolling.

C1FlexGrid also takes advantage of container recycling for cell templates, which optimizes scrolling performance by reusing cell elements that are scrolled out of view, rather than destroying them. It's important to note that custom cell templates support container recycling but cell factories do not. To improve performance with custom cells, consider using cell templates with simplified bindings and converters rather than cell factories. You can find more information in the following topics:

- Creating custom cells in XAML using cell templates
- Creating custom cells in code using cell factories

You get UI virtualization and cell template recycling for free, out of the box, without having to do any extra work with **C1FlexGrid**. If that's not enough to improve performance you can also consider Incremental Template Uploading:

- Incremental Template Updating is useful for rendering complex custom cells in phases as the user scrolls.

Incremental Template Updating

With incremental template updating through the CellContentChanging event, you can update the cell template in phases. This improves load time and scrolling performance because cells can be rendered in parts, or phases, rather than all at once.

For example, as a user scrolls through the grid very quickly, the cell content might be displayed as plain text. When the grid stops scrolling, the cell content can be updated to display heavier content that requires expensive calculation; content you wouldn't need to have rendered for every cell during a scroll.

Successive phases can be used to perform heavier actions, and will be executed provided the UI-thread is free. This is modeled after a similar feature from the Microsoft GridView control.

> **Note**: You can see the **CellContentChanging** event in action in the **Financial** sample. In the Control Explorer, it's the **See it in action!** sample. You can find the **FlexGrid samples** installed here on your machine: **Documents\ComponentOne Samples\UWP\C1.UWP.FlexGrid\CS**

The **CellContentChanging** event resembles the following:

C#

```
private void _flexFinancial_CellContentChanging(C1FlexGrid sender,
CellContentChangingEventArgs args)
{
    if (args.Phase == 0)
    {
        args.RegisterUpdateCallback(_flexFinancial_CellContentChanging);
    }
    else if (args.Phase == 1)
    {
        var factory = _flexFinancial.CellFactory as FinancialCellFactory;
        if (factory != null)
            factory.ShowLiveData(_flexFinancial, args.Range, args.Cell);
    }
}
```

The preceding **CellContentChanging** sample calls the **ShowLiveData** event:

C#

```
public void ShowLiveData(C1FlexGrid grid, CellRange range, FrameworkElement cell)
{
    var stockTicker = (cell as Border).Child as StockTicker;
    if (stockTicker != null)
    {
        var c = grid.Columns[range.Column];
        var r = grid.Rows[range.Row];
        var pi = c.PropertyInfo;

        // to show sparklines
        stockTicker.Tag = r.DataItem;
        stockTicker.BindingSource = pi.Name;

        var binding = new Binding { Path = new PropertyPath(pi.Name) };
        binding.Converter = new MyConverter();
        binding.Source = r.DataItem;
        binding.Mode = BindingMode.OneWay;
        stockTicker.SetBinding(StockTicker.ValueProperty, binding);
    }
}
```

The StockTicker class used in the event represents a sparkline, which is the heavy part of the implementation. By calling this from the **CellContentChanging** event, the **ShowLiveData** event isn't executed until the UI-thread is free.

Printing

The FlexGrid control provides the Print method to print the grid. This method has three overloads which takes parameters that allow you to specify the document name, page margins, scaling, maximum number of pages to print, and print parameters. The output is a faithful rendering of the grid, where all style elements, fonts, gradients, images, row and column headers are included on every page.

The following code uses Print method that opens a Print dialog directly for printing the FlexGrid:

Visual Basic

```
c1FlexGrid1.Print("FlexGrid_Print")
```

C#

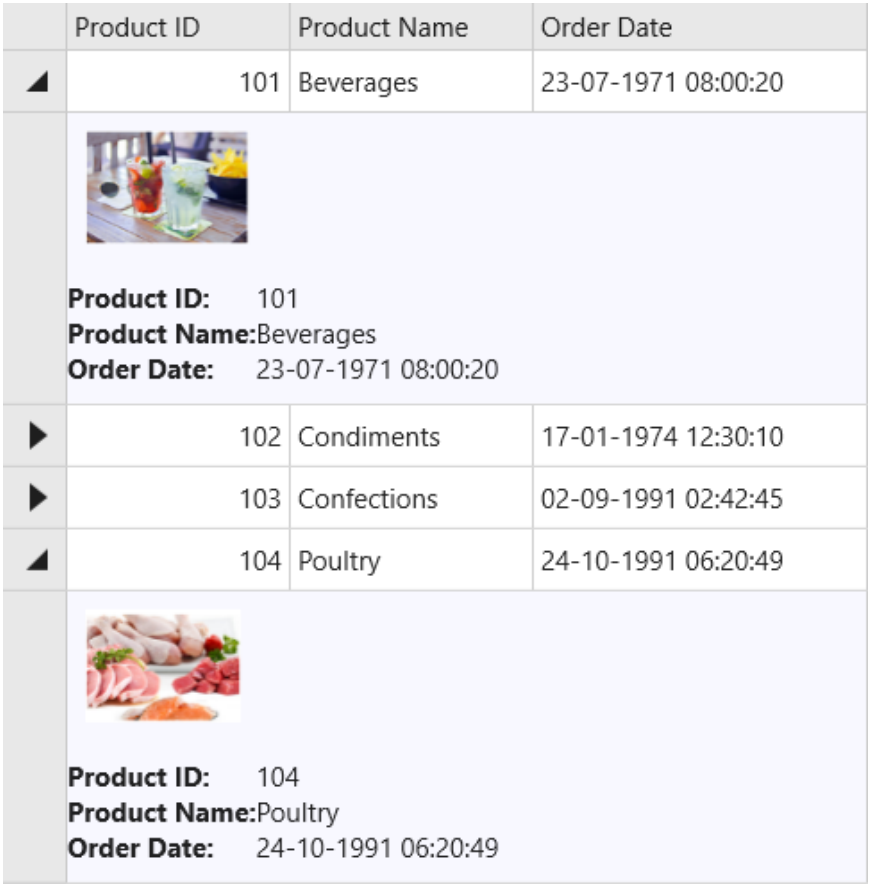
```
c1FlexGrid1.Print("FlexGrid_Print");
```

Row Details Template

FlexGrid provides the flexibility to show additional information about each row through a row details template. A row details template is an empty data template that can be added to a row for displaying details. You can embed text, UI elements, and data-bound controls such as InputPanel in the row details template. For each row in a grid, you can

insert a data template to present its summary, or enlist details in other controls such as text box, without affecting the dimensions of the grid. You can also use this template to create hierarchical grids displaying grouped data.

The following image shows details of each row displayed in a row details template.



To add row details template in FlexGrid

1. Add C1FlexGrid control to the XAML designer and set the name of the control as grid.
2. In XAML code, create three columns inside the <Grid> tag to display some data fields such as Product ID, Product Name and Order Date.

XAML

```
<FlexGrid:C1FlexGrid.Columns>
    <FlexGrid:Column Header="Product ID" Binding="{Binding ProductId}"
Width="75" />
    <FlexGrid:Column Header="Product Name" Binding="{Binding ProductName}"
Width="150" />
    <FlexGrid:Column Header="Order Date" Binding="{Binding OrderDate}"
Width="300" />
</FlexGrid:C1FlexGrid.Columns>
```

3. Create a row details template to include an Image control and six TextBlock controls within a StackPanel.

XAML

```
<FlexGrid:C1FlexGrid.RowDetailsTemplate>
    <DataTemplate>
        <StackPanel Background="GhostWhite">
            <Image HorizontalAlignment="Left" Name="img"
                   Source="{Binding ImgSource}"  Height="64" Margin="10" />
```

```
            <Grid Margin="0, 10">
                <Grid.ColumnDefinitions>
                    <ColumnDefinition Width="Auto" />
                    <ColumnDefinition Width="*" />
                </Grid.ColumnDefinitions>
                <Grid.RowDefinitions>
                    <RowDefinition Height="Auto" />
                    <RowDefinition Height="Auto" />
                    <RowDefinition Height="Auto" />
                </Grid.RowDefinitions>
                <TextBlock Text="Product ID: " FontWeight="Bold" />
                <TextBlock Text="{Binding ProductId}" Grid.Column="1" />
                <TextBlock Text="Product Name: " FontWeight="Bold" Grid.Row="1"
/>
                <TextBlock Text="{Binding ProductName}" Grid.Column="1"
Grid.Row="1" />
                <TextBlock Text="Order Date: " FontWeight="Bold" Grid.Row="2" />
                <TextBlock Text="{Binding OrderDate}" Grid.Column="1"
Grid.Row="2" />
            </Grid>
        </StackPanel>
    </DataTemplate>
</FlexGrid:C1FlexGrid.RowDetailsTemplate>
```

4. Right-click on the project name and select **Add|New Folder** from the menu to add a folder named **Resources**.
5. Add images to the **Resources** folder.
6. Switch to the **Code** view and create a class, ProductDetails, to add data in the FlexGrid.

Visual Basic

```
Public Class ProductDetails
    Public Property ProductId() As Integer
        Get
            Return m_ProductId
        End Get
        Set
            m_ProductId = Value
        End Set
    End Property
    Private m_ProductId As Integer
    Public Property ProductName() As String
        Get
            Return m_ProductName
        End Get
        Set
            m_ProductName = Value
        End Set
    End Property
    Private m_ProductName As String
    Public Property OrderDate() As DateTime
        Get
            Return m_OrderDate
        End Get
        Set
            m_OrderDate = Value
        End Set
    End Property
    Private m_OrderDate As DateTime
```

```
    Public Property ImgSource() As ImageSource
        Get
            Return m_ImgSource
        End Get
        Set
            m_ImgSource = Value
        End Set
    End Property
    Private m_ImgSource As ImageSource
End Class
```

C#

```
public class ProductDetails
{
    public int ProductId { get; set; }
    public string ProductName { get; set; }
    public DateTime OrderDate { get; set; }
    public ImageSource ImgSource { get; set; }
}
```

7. Create a list of products, add data to it and bind the FlexGrid control with the list using ItemsSource property.

Visual Basic

```
'Create a list
Dim users As New List(Of ProductDetails)()
'Add items to the list
users.Add(New ProductDetails() With {
  .ProductId = 101,
  .ProductName = "Beverages",
  .OrderDate = New DateTime(1971, 7, 23, 8, 0, 20),
  .ImgSource = New BitmapImage(New Uri("ms-appx:///Resources/Beverage.png"))
})
users.Add(New ProductDetails() With {
  .ProductId = 102,
  .ProductName = "Condiments",
  .OrderDate = New DateTime(1974, 1, 17, 12, 30, 10),
  .ImgSource = New BitmapImage(New Uri("ms-appx:///Resources/Condiments.png"))
})
users.Add(New ProductDetails() With {
  .ProductId = 103,
  .ProductName = "Confections",
  .OrderDate = New DateTime(1991, 9, 2, 2, 42, 45),
  .ImgSource = New BitmapImage(New Uri("ms-appx:///Resources/Confections.png"))
})
users.Add(New ProductDetails() With {
  .ProductId = 104,
  .ProductName = "Poultry",
  .OrderDate = New DateTime(1991, 10, 24, 6, 20, 49),
  .ImgSource = New BitmapImage(New Uri("ms-appx:///Resources/Poultry.png"))
})
'Populate the grid
grid.ItemsSource = users
```

C#

```
//Create a list
List<ProductDetails> users = new List<ProductDetails>();

//Add items to the list
users.Add(new ProductDetails() { ProductId = 101, ProductName = "Beverages",
```

```
  OrderDate = new DateTime(1971, 7, 23, 8, 0,20),
  ImgSource = new BitmapImage(new Uri("ms-appx:///Resources/Beverage.png"))});
users.Add(new ProductDetails() { ProductId = 102, ProductName = "Condiments",
  OrderDate = new DateTime(1974, 1, 17, 12,30,10),
  ImgSource = new BitmapImage(new Uri("ms-
appx:///Resources/Condiments.png"))});
users.Add(new ProductDetails() { ProductId = 103, ProductName = "Confections",
  OrderDate = new DateTime(1991, 9, 2, 2,42,45),
  ImgSource = new BitmapImage(new Uri("ms-
appx:///Resources/Confections.png"))});
users.Add(new ProductDetails() { ProductId = 104, ProductName = "Poultry",
  OrderDate = new DateTime(1991, 10, 24, 6,20,49),
  ImgSource = new BitmapImage(new Uri("ms-appx:///Resources/Poultry.png"))});

//Populate the grid
grid.ItemsSource = users;
```

Tutorials

Populating the Grid with Data

> For a complete walk-through on populating the C1FlexGrid control by implementing **C1CollectionView**, please see Grid Population.

Like most other grids, once you've added the grid to your application, you'll populate it using the ItemsSource property that expects an object that implements an IEnumerable interface. In most cases, though, you'll work at a higher level and use an object that implements the **C1CollectionView** interface.

The **C1CollectionView** class delivers functionality that's missing in the UWP ICollectionView, like sorting, filtering, grouping, and editing. **C1CollectionView** provides a familiar object model, but fills the ICollectionView gaps with a rich interface.

For example, to display a list of customer objects in a **C1FlexGrid for UWP** application, you would use code such as this:

Example Title

```
List<Customer> customers = await GetCustomerData();
var view = new C1.Xaml.C1CollectionView(customers);
```

And then bind it to your **C1FlexGrid** control:

XAML

```
c1FlexGrid1.ItemsSource = view;
```

You could also bind the grid directly to the customer list, of course. But binding to a **C1CollectionView** is usually a better idea because it retains a lot of the data configuration for the application, and that can be shared across controls.

If you bind many different controls to the same **C1CollectionView** object, they will all show the same view. Selecting an item in one control will automatically update the selection on all other controls. You can also share filtering, grouping, or sorting across all controls bound to the same view.

Using the above code, the grid will scan the data source and automatically generate columns for each public property of the data source items. Automatically created columns can be customized using code, or you may disable the automatic column generation altogether and create the columns yourself, in code or in XAML.

For example, the XAML below disables the automatic column generation and specifies the columns in XAML instead:

XAML

```
<FlexGrid:C1FlexGrid x:Name="flexgrid1" AllowResizing="Both" AllowDragging="Both"
AllowDrop="True" AutoGenerateColumns="false" ColumnHeaderForeground="White" >
<FlexGrid:C1FlexGrid.Columns>
   <FlexGrid:Column Binding="{Binding Active, Mode=TwoWay}" />
   <FlexGrid:Column Binding="{Binding ID, Mode=TwoWay}" />
   <FlexGrid:Column Binding="{Binding Name, Mode=TwoWay}" Width="*"/>
   <FlexGrid:Column Binding="{Binding Country, Mode=TwoWay}" Width="*"/>
   <FlexGrid:Column Binding="{Binding Hired, Mode=TwoWay}" Format="d" Width="*" />
   <FlexGrid:Column Binding="{Binding Father, Mode=TwoWay}" Width="*"/>
   <FlexGrid:Column Binding="{Binding Weight, Mode=TwoWay}" Width="*"/>
</FlexGrid:C1FlexGrid.Columns>
```

```
</FlexGrid:C1FlexGrid>
```

This is similar to the XAML you would use to accomplish the same task using the Microsoft DataGrid or the original ComponentOne DataGrid controls:

XAML

```
<!-- create columns on an MSDataGrid (or C1DataGrid) -->
 <ms:DataGrid Name="_msSongs"
   AutoGenerateColumns="False" >
   <ms:DataGrid.Columns>
     <ms:DataGridTextColumn Binding="{Binding Name}" Header="Title"
         CanUserReorder="False" Width="300" />
     <ms:DataGridTextColumn Binding="{Binding Duration}" />
     <ms:DataGridTextColumn Binding="{Binding Size}" />
     <ms:DataGridTextColumn Binding="{Binding Rating}" Width="200" />
   </ms:DataGrid.Columns>
 </ms:DataGrid>
```

As you can see, the markup is virtually identical.

You can use the binding as an indexer into the grid's Columns collection. For example, if you wanted to make the "Rating" column 300 pixels wide using code, you could write this:

C#

```
c1FlexGrid1.Columns["Rating"].Width = new GridLength(300);
```

This syntax should be familiar to **C1FlexGrid** users. It is exactly the same command you would use when working with the WPF version of the control.

Grid Population

This help will walk you through populating the grid by binding data from a separate code file. It will also demonstrate the **C1FlexGrid** control's filtering and grouping capabilities.
Follow these steps:

1. Open the MainPage.xaml page if it isn't open already. Click within the <Grid> </Grid> tags and insert the following XAML markup to set up the general ComboBox and TextBox controls that will control both filtering and grouping:

   XAML

   ```
   <Grid>
          <Grid.RowDefinitions>
              <RowDefinition Height="Auto"/>
              <RowDefinition Height="Auto"/>
              <RowDefinition />
          </Grid.RowDefinitions>
          <Border BorderThickness="0,1,0,1" BorderBrush="White" Margin="5">
              <StackPanel Orientation="Horizontal">
                  <StackPanel.Resources>
                      <Style TargetType="TextBlock">
                          <Setter Property="FontSize" Value="14"/>
                          <Setter Property="Margin" Value="10,0,6,0"/>
                          <Setter Property="VerticalAlignment" Value="Center"/>
                      </Style>
                  </StackPanel.Resources>
   ```

```
                <TextBlock Text="Group on:" HorizontalAlignment="Right"/>
                <ComboBox Name="groupComboBox" Grid.Column="1"
                                          SelectionChanged="groupComboBox_SelectionChanged_1"
Height="30" Width="110" Margin="5"/>
                <TextBlock Text="Filter on:" Grid.Row="1" HorizontalAlignment="Right"/>
                <ComboBox Name="filterComboBox" Grid.Row="1" Grid.Column="1"

SelectionChanged="filterComboBox_SelectionChanged_1" Height="30" Width="110" Margin="5"/>
                <TextBox Name="filterTextBox" Grid.Row="1" Grid.Column="2"
TextChanged="filterTextBox_TextChanged_1" Height="25"
                            Width="110" Margin="5"/>
            </StackPanel>
        </Border>
```

2. Add a **C1FlexGrid** control to your application by double-clicking the control in your Visual Studio Toolbox or by adding the following XAML markup to your application:

XAML

```
<FlexGrid:C1FlexGrid x:Name="flexgrid1" AllowResizing="Both" AllowDragging="Both"
AllowDrop="True" ColumnHeaderForeground="White" />
```

3. Add the following XAML markup to your <FlexGrid:C1FlexGrid> </FlexGrid: C1FlexGrid> tags to set up the FlexGrid.Columns and to create the bindings in markup:

XAML

```
<FlexGrid:C1FlexGrid.Columns>
    <FlexGrid:Column Binding="{Binding Active, Mode=TwoWay}" />
    <FlexGrid:Column Binding="{Binding ID, Mode=TwoWay}" />
    <FlexGrid:Column Binding="{Binding Name, Mode=TwoWay}" Width="*"/>
    <FlexGrid:Column Binding="{Binding Country, Mode=TwoWay}" Width="*"/>
    <FlexGrid:Column Binding="{Binding Hired, Mode=TwoWay}" Format="d" Width="*" />
    <FlexGrid:Column Binding="{Binding Father, Mode=TwoWay}" Width="*"/>
    <FlexGrid:Column Binding="{Binding Weight, Mode=TwoWay}" Width="*"/>
 </FlexGrid:C1FlexGrid.Columns>
```

4. Right-click on your application name and select **Add | New Item** from the list. Select **Code File** from the **Add New File** dialog box and name it **Customer.cs**. The new code file should open immediately.
5. In the **Customer.cs** file, add the following namespaces and code to create the data with which we will populate the grid:

C#

```
using System;
 using System.Collections;
 using System.Collections.Generic;
 using System.Collections.ObjectModel;
 using System.ComponentModel;
 using System.Reflection;
namespace FlexGridSamples
 {
        public class Customer :
        INotifyPropertyChanged,
        IEditableObject
    {
        // ** fields
        int _id, _countryID;
        string _first, _last;
        string _father, _brother, _cousin;
        bool _active;
```

```
        DateTime _hired;
        double _weight;
        // ** data generators
        static Random _rnd = new Random();
        static string[] _firstNames =
"Andy|Ben|Charlie|Dan|Ed|Fred|Gil|Herb|Jack|Karl|Larry|Mark|
Noah|Oprah|Paul|Quince|Rich|Steve|Ted|Ulrich|Vic|Xavier|Zeb".Split('|');
        static string[] _lastNames =
"Ambers|Bishop|Cole|Danson|Evers|Frommer|Griswold|Heath|Jammers|Krause|Lehman|Myers|Neiman|
Orsted|Paulson|Quaid|Richards|Stevens|Trask|Ulam".Split('|');
        static string[] _countries = "China|India|United
States|Indonesia|Brazil|Pakistan|Bangladesh|Nigeria|Russia|Japan|Mexico|Philippines
|Vietnam|Germany|Ethiopia|Egypt|Iran|Turkey|Congo|France|Thailand|UnitedKingdom|
Italy|Myanmar".Split('|');
        // ** ctors
        public Customer()
            : this(_rnd.Next(10000))
        {
        }
        public Customer(int id)
        {
            ID = id;
           First = GetString(_firstNames);
            Last = GetString(_lastNames);
            CountryID = _rnd.Next() % _countries.Length;
            Active = _rnd.NextDouble() >= .5;
            Hired = DateTime.Today.AddDays(-_rnd.Next(1, 365));
            Weight = 50 + _rnd.NextDouble() * 50;
            _father = string.Format("{0} {1}", GetString(_firstNames), Last);
            _brother = string.Format("{0} {1}", GetString(_firstNames), Last);
            _cousin = GetName();
        }
        // ** object model
        public int ID
        {
            get { return _id; }
           set
            {
                if (value != _id)
               {
                    _id = value;
                    RaisePropertyChanged("ID");
                }
            }
        }
        public string Name
        {
            get { return string.Format("{0} {1}", First, Last); }
        }
        public string Country
        {
            get { return _countries[_countryID]; }
        }
        public int CountryID
       {
            get { return _countryID; }
            set
            {
```

```
                if (value != _countryID && value > -1 && value < _countries.Length)
                {
                    _countryID = value;
                   // call OnPropertyChanged with null parameter since setting this
property
                    // modifies the value of "CountryID" and also the value of "Country".
                    RaisePropertyChanged("");
                }
            }
        }
        public bool Active
        {
            get { return _active; }
            set
            {
                if (value != _active)
                {
                    _active = value;
                    RaisePropertyChanged("Active");
                }
            }
        }
        public string First
        {
            get { return _first; }
            set
            {
                if (value != _first)
                {
                    _first = value;
                   // call OnPropertyChanged with null parameter since setting this
property
                    // modifies the value of "First" and also the value of "Name".
                    RaisePropertyChanged("");
                }
            }
        }
        public string Last
        {
            get { return _last; }
            set
            {
                if (value != _last)
                {
                    _last = value;
                   // call OnPropertyChanged with null parameter since setting this
property
                    // modifies the value of "First" and also the value of "Name".
                    RaisePropertyChanged("");
                }
            }
        }
        public DateTime Hired
        {
            get { return _hired; }
            set
            {
                if (value != _hired)
```

```
                {
                    _hired = value;
                    RaisePropertyChanged("Hired");
                }
            }
        }
        public double Weight
        {
            get { return _weight; }
            set
            {
                if (value != _weight)
                {
                    _weight = value;
                    RaisePropertyChanged("Weight");
                }
            }
        }
        // some read-only stuff
        public string Father
        {
            get { return _father; }
        }
        public string Brother
        {
            get { return _brother; }
        }
        public string Cousin
        {
            get { return _cousin; }
        }
        // ** utilities
        static string GetString(string[] arr)
        {
            return arr[_rnd.Next(arr.Length)];
        }
        static string GetName()
        {
            return string.Format("{0} {1}", GetString(_firstNames),
GetString(_lastNames));
        }
        // ** static list provider
        public static ObservableCollection<Customer> GetCustomerList(int count)
        {
            var list = new ObservableCollection<Customer>();
            for (int i = 0; i < count; i++)
            {
                list.Add(new Customer(i));
            }
            return list;
        }
        // ** static value providers
        public static string[] GetCountries() { return _countries; }
        public static string[] GetFirstNames() { return _firstNames; }
        public static string[] GetLastNames() { return _lastNames; }
        #region ** INotifyPropertyChanged Members
        // this interface allows bounds controls to react to changes in the data objects.
        void RaisePropertyChanged(string propertyName)
```

```
        {
            OnPropertyChanged(new PropertyChangedEventArgs(propertyName));
        }
        public event PropertyChangedEventHandler PropertyChanged;
        protected void OnPropertyChanged(PropertyChangedEventArgs e)
        {
            if (PropertyChanged != null)
                PropertyChanged(this, e);
        }
       #endregion
               #region IEditableObject Members
       // this interface allows transacted edits (user can press escape to restore
previous values).
       Customer _clone;
        public void BeginEdit()
        {
            _clone = (Customer)this.MemberwiseClone();
        }
        public void EndEdit()
        {
            _clone = null;
        }
        public void CancelEdit()
        {
            if (_clone != null)
            {
                foreach (var p in this.GetType().GetRuntimeProperties())
                {
                    if (p.CanRead && p.CanWrite)
                    {
                        p.SetValue(this, p.GetValue(_clone, null), null);
                    }
                }
            }
        }
       #endregion
    }
 }
```

Visual Basic

```
Visual Basic
 Imports System
 Imports System.Collections
 Imports System.Collections.Generic
 Imports System.Collections.ObjectModel
 Imports System.ComponentModel
 Imports System.Reflection
Namespace FlexGridSamples
     Public Class Customer
        Implements INotifyPropertyChanged
         Implements IEditableObject
         ' ** fields
         Private _id As Integer, _countryID As Integer
         Private _first As String, _last As String
         Private _father As String, _brother As String, _cousin As String
         Private _active As Boolean
         Private _hired As DateTime
         Private _weight As Double
```

```
        ' ** data generators
         Shared _rnd As New Random()
         Shared _firstNames As String() =
"Andy|Ben|Charlie|Dan|Ed|Fred|Gil|Herb|Jack|Karl|Larry|
Mark|Noah|Oprah|Paul|Quince|Rich|Steve|Ted|Ulrich|Vic|Xavier|Zeb".Split("|"c)
         Shared _lastNames As String() =
"Ambers|Bishop|Cole|Danson|Evers|Frommer|Griswold|Heath|Jammers|Krause|Lehman|Myers
|Neiman|Orsted|Paulson|Quaid|Richards|Stevens|Trask|Ulam".Split("|"c)
         Shared _countries As String() = "China|India|United
States|Indonesia|Brazil|Pakistan|Bangladesh|Nigeria|Russia|Japan|Mexico|
Philippines|Vietnam|Germany|Ethiopia|Egypt|Iran|Turkey|Congo|France|Thailand|United
Kingdom|Italy|Myanmar".Split("|"c)
        ' ** ctors
         Public Sub New()
             Me.New(_rnd.[Next](10000))
         End Sub
         Public Sub New(id__1 As Integer)
             ID = id__1
            First = GetString(_firstNames)
             Last = GetString(_lastNames)
             CountryID = _rnd.[Next]() Mod _countries.Length
             Active = _rnd.NextDouble() >= 0.5
             Hired = DateTime.Today.AddDays(-_rnd.[Next](1, 365))
             Weight = 50 + _rnd.NextDouble() * 50
             _father = String.Format("{0} {1}", GetString(_firstNames), Last)
             _brother = String.Format("{0} {1}", GetString(_firstNames), Last)
             _cousin = GetName()
         End Sub
        ' ** object model
         Public Property ID() As Integer
             Get
                 Return _id
             End Get
             Set(value As Integer)
                 If value <> _id Then
                     _id = value
                     RaisePropertyChanged("ID")
                 End If
             End Set
         End Property
         Public ReadOnly Property Name() As String
             Get
                 Return String.Format("{0} {1}", First, Last)
             End Get
         End Property
         Public ReadOnly Property Country() As String
             Get
                 Return _countries(_countryID)
             End Get
         End Property
         Public Property CountryID() As Integer
             Get
                 Return _countryID
             End Get
             Set(value As Integer)
                 If value <> _countryID AndAlso value > -1 AndAlso value <
_countries.Length Then
                     _countryID = value
```

```
                    ' call OnPropertyChanged with null parameter since setting this
property
                    ' modifies the value of "CountryID" and also the value of "Country".
                    RaisePropertyChanged("")
                End If
            End Set
        End Property
        Public Property Active() As Boolean
            Get
                Return _active
            End Get
            Set(value As Boolean)
                If value <> _active Then
                    _active = value
                    RaisePropertyChanged("Active")
                End If
            End Set
        End Property
        Public Property First() As String
            Get
                Return _first
            End Get
            Set(value As String)
                If value <> _first Then
                    _first = value
                   ' call OnPropertyChanged with null parameter since setting this
property
                    ' modifies the value of "First" and also the value of "Name".
                    RaisePropertyChanged("")
                End If
            End Set
        End Property
        Public Property Last() As String
            Get
                Return _last
            End Get
            Set(value As String)
                If value <> _last Then
                    _last = value
                   ' call OnPropertyChanged with null parameter since setting this
property
                    ' modifies the value of "First" and also the value of "Name".
                    RaisePropertyChanged("")
                End If
            End Set
       End Property
        Public Property Hired() As DateTime
            Get
                Return _hired
            End Get
            Set(value As DateTime)
                If value <> _hired Then
                    _hired = value
                    RaisePropertyChanged("Hired")
                End If
            End Set
        End Property
        Public Property Weight() As Double
```

```
            Get
                Return _weight
            End Get
            Set(value As Double)
                If value <> _weight Then
                    _weight = value
                    RaisePropertyChanged("Weight")
                End If
            End Set
        End Property
        ' some read-only stuff
        Public ReadOnly Property Father() As String
            Get
                Return _father
            End Get
        End Property
        Public ReadOnly Property Brother() As String
            Get
                Return _brother
            End Get
        End Property
        Public ReadOnly Property Cousin() As String
            Get
                Return _cousin
            End Get
        End Property
        ' ** utilities
        Private Shared Function GetString(arr As String()) As String
            Return arr(_rnd.[Next](arr.Length))
        End Function
        Private Shared Function GetName() As String
            Return String.Format("{0} {1}", GetString(_firstNames), GetString(_lastNames))
        End Function
        ' ** static list provider
        Public Shared Function GetCustomerList(count As Integer) As
ObservableCollection(Of Customer)
            Dim list = New ObservableCollection(Of Customer)()
            For i As Integer = 0 To count - 1
                list.Add(New Customer(i))
            Next
            Return list
        End Function
        ' ** static value providers
        Public Shared Function GetCountries() As String()
            Return _countries
        End Function
        Public Shared Function GetFirstNames() As String()
            Return _firstNames
        End Function
        Public Shared Function GetLastNames() As String()
            Return _lastNames
        End Function
 #Region "** INotifyPropertyChanged Members"
        ' this interface allows bounds controls to react to changes in the data objects.
        Private Sub RaisePropertyChanged(propertyName As String)
            OnPropertyChanged(New PropertyChangedEventArgs(propertyName))
        End Sub
        Public Event PropertyChanged As PropertyChangedEventHandler
```

```
        Protected Sub OnPropertyChanged(e As PropertyChangedEventArgs)
            RaiseEvent PropertyChanged(Me, e)
        End Sub
#End Region
 #Region "IEditableObject Members"
       ' this interface allows transacted edits (user can press escape to restore previous
values).
       Private _clone As Customer
        Public Sub BeginEdit()
            _clone = DirectCast(Me.MemberwiseClone(), Customer)
        End Sub
        Public Sub EndEdit()
            _clone = Nothing
        End Sub
        Public Sub CancelEdit()
            If _clone IsNot Nothing Then
                For Each p In Me.[GetType]().GetRuntimeProperties()
                    If p.CanRead AndAlso p.CanWrite Then
                        p.SetValue(Me, p.GetValue(_clone, Nothing), Nothing)
                    End If
                Next
            End If
        End Sub
#End Region
        Public Sub BeginEdit1() Implements IEditableObject.BeginEdit
       End Sub
       Public Sub CancelEdit1() Implements IEditableObject.CancelEdit
       End Sub
       Public Sub EndEdit1() Implements IEditableObject.EndEdit
       End Sub
       Public Event PropertyChanged1(sender As Object, e As PropertyChangedEventArgs)
Implements INotifyPropertyChanged.PropertyChanged
    End Class
 End Namespace
```

6. Switch to Code View and add the following namespace:

C#

```
using C1.Xaml.FlexGrid;
using C1.Xaml;
using System.Collections.ObjectModel;
```

Visual Basic

```
Imports C1.Xaml.FlexGrid
Imports C1.Xaml
Imports System.Collections.ObjectModel
```

7. Add the following code above the MainPage() constructor to represent the ICollectionView interface implementation:

C#

```
private readonly C1CollectionView _c1CollectionView;
private const string NoneItem = "(None)";
```

Visual Basic

```
Dim _c1CollectionView As C1CollectionView
Const NoneItem As String = "(None)"
```

8. Insert the following code directly below the InitializeComponent() method to create a list and an observable collection. This

code also sets the C1FlexGrid.ItemsSource property:

C#

```
IList<string> fieldNames = new string[] { "ID", "Name", "Country", "Active", "Hired",
"Father", "Weight" };
            List<string> groupFields = new List<string>(fieldNames);
             groupFields.Sort();
             //groupFields.Remove("Active");
             List<string> filterFields = new List<string>(groupFields);
             groupFields.Insert(0, NoneItem);
             groupComboBox.ItemsSource = groupFields;
             groupComboBox.SelectedItem = NoneItem;
             filterComboBox.ItemsSource = filterFields;
             filterComboBox.SelectedIndex = 0;

             ObservableCollection<Customer> customers = Customer.GetCustomerList(50);
             _c1CollectionView = new C1CollectionView();
             _c1CollectionView.SourceCollection = customers;
            c1FlexGrid1.ItemsSource = _c1CollectionView;
         }
```

Visual Basic

```
Dim fieldNames As IList(Of String) = New String() {"ID", "Name", "Country", "Active",
"Hired", "Father", "Weight"}
        Dim groupFields As New List(Of String)(fieldNames)
         groupFields.Sort()
         groupFields.Remove("Active")
         Dim filterFields As New List(Of String)(groupFields)
         groupFields.Insert(0, NoneItem)
         groupComboBox.ItemsSource = groupFields
         groupComboBox.SelectedItem = NoneItem
         filterComboBox.ItemsSource = filterFields
         filterComboBox.SelectedIndex = 0
        Dim customers As ObservableCollection(Of Customer) = Customer.GetCustomerList(50)
         _c1CollectionView = New C1CollectionView()
         _c1CollectionView.SourceCollection = customers
        flexgrid1.ItemsSource = _c1CollectionView
```

9. To add grouping capabilities, add the following code:

C#

```
void UpdateGrouping()
         {
             if (_c1CollectionView == null)
                 return;
             using (_c1CollectionView.DeferRefresh())
             {
                 _c1CollectionView.GroupDescriptions.Clear();
                 if (groupComboBox.SelectedItem != NoneItem)
                 {
                     _c1CollectionView.GroupDescriptions.Add(new
PropertyGroupDescription((string)groupComboBox.SelectedItem));
                 }
             }
         }
```

Visual Basic

```
Private Sub UpdateGrouping()
```

```
        If (_c1CollectionView Is Nothing) Then
            Return
        End If
        _c1CollectionView.DeferRefresh()
        _c1CollectionView.GroupDescriptions.Clear()
        If (groupComboBox.SelectedItem <> NoneItem) Then
            _c1CollectionView.GroupDescriptions.Add(New
PropertyGroupDescription(CType(groupComboBox.SelectedItem, String)))
        End If
    End Sub
```

10. Set and control the C1FlexGrid's filtering capabilities with the following code:

C#

```
void UpdateFiltering()
        {
            if (filterTextBox.Text.Length == 0)
                _c1CollectionView.Filter = null;
            else
            {
                if (_c1CollectionView.Filter == null)
                    _c1CollectionView.Filter = FilterFunction;
                else
                    _c1CollectionView.Refresh();
            }
        }
       bool FilterFunction(object customer)
        {
            Customer cust = customer as Customer;
            if (cust == null)
                return false;
            object propValue = null;
            switch ((string)filterComboBox.SelectedItem)
            {
                case "ID":
                    propValue = cust.ID;
                    break;
                case "Name":
                    propValue = cust.Name;
                    break;
                case "Country":
                    propValue = cust.Country;
                    break;
                case "Hired":
                    propValue = cust.Hired;
                    break;
                case "Father":
                    propValue = cust.Father;
                    break;
                case "Weight":
                    propValue = cust.Weight;
                    break;
                default:
                    return true;
            }
            if (propValue == null)
                return false;
            return propValue.ToString().StartsWith(filterTextBox.Text,
```

```
StringComparison.CurrentCultureIgnoreCase);
         }
```

Visual Basic

```
Private Sub UpdateFiltering()
         If filterTextBox.Text.Length = 0 Then
             _c1CollectionView.Filter = Nothing
         Else
             If _c1CollectionView.Filter Is Nothing Then
                 _c1CollectionView.Filter = AddressOf FilterFunction
             Else
                 _c1CollectionView.Refresh()
             End If
         End If
     End Sub
    Private Function FilterFunction(ByVal customer As Object) As Boolean '
         Dim cust As Customer = CType(customer, Customer)
         If (cust Is Nothing) Then
             Return False
         End If
        Dim propValue As Object = Nothing
         Select Case (CType(filterComboBox.SelectedItem, String))
             Case "ID"
                 propValue = cust.ID
             Case "Name"
                 propValue = cust.Name
             Case "Country"
                 propValue = cust.Country
             Case "Hired"
                 propValue = cust.Hired
             Case "Father"
                 propValue = cust.Father
             Case "Weight"
                 propValue = cust.Weight
             Case Else
                 Return True
         End Select
```

11. Finally, add the following code to handle the ComboBox and TextBox SelectionChanged and TextChanged events:

C#

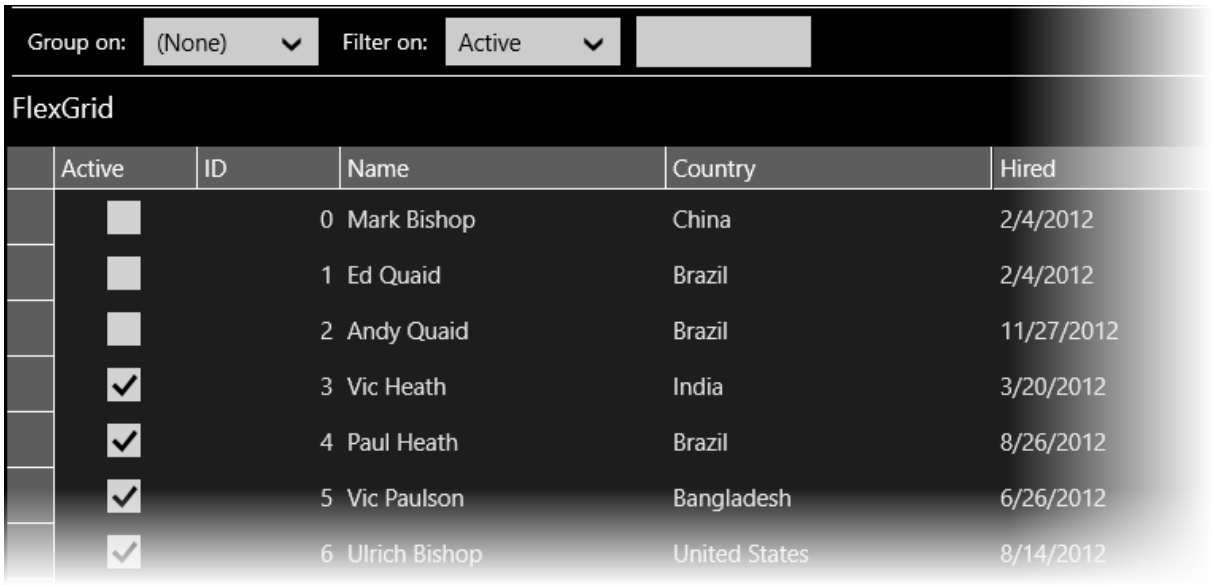
```
private void groupComboBox_SelectionChanged_1(object sender, SelectionChangedEventArgs e)
         {
             UpdateGrouping();
         }
        private void filterTextBox_TextChanged_1(object sender, TextChangedEventArgs e)
         {
             UpdateFiltering();
         }
        private void filterComboBox_SelectionChanged_1(object sender,
SelectionChangedEventArgs e)
         {
             filterTextBox.Text = "";
         }
```

Visual Basic

```
If (propValue Is Nothing) Then
             Return False
```

```
        End If
        Return propValue.ToString.StartsWith(filterTextBox.Text,
StringComparison.CurrentCultureIgnoreCase)
    End Function
    Private Sub groupComboBox_SelectionChanged_1(ByVal sender As Object, ByVal e As
SelectionChangedEventArgs)
        UpdateGrouping()
    End Sub
    Private Sub filterTextBox_TextChanged_1(ByVal sender As Object, ByVal e As
TextChangedEventArgs)
        UpdateFiltering()
    End Sub
    Private Sub filterComboBox_SelectionChanged_1(ByVal sender As Object, ByVal e As
SelectionChangedEventArgs)
        filterTextBox.Text = ""
    End Sub
```

12. When you run your application, you should be able to filter and group data in several different ways. Your **C1FlexGrid** control should resemble the following image:



Task-Based Help

Creating a C1Flex Grid in Code

To create a C1FlexGrid control, insert the following code below the application's constructor:

Visual Basic

```
Dim flexgrid1 As C1FlexGrid
LayoutRoot.Children.Add(flexgrid1)
```

C#

```
var flexgrid1 = new C1FlexGrid();
LayoutRoot.Children.Add(flexgrid1);
```