Filter Data in WPF DataGrid

In most business applications, users wish to filter data that has been displayed to them. This blog post is not about how you filter, per se, but how to display the filtering options to the user. In the first scenario, an expander control where the user selects values to filter is used. In the second scenario, the filters are displayed within the column header on the data grid control. The third scenario ensures that column headers are aligned consistently across each column.

Customer Classes

The sample for this blog post uses the AdventureWorksLT database included with SQL Server. I am going to perform filtering on the Customer table from that database. As such, I have created a Customer entity class, an AdventureWorksDbContext class and a CustomerViewModel class as shown in Figure 1.

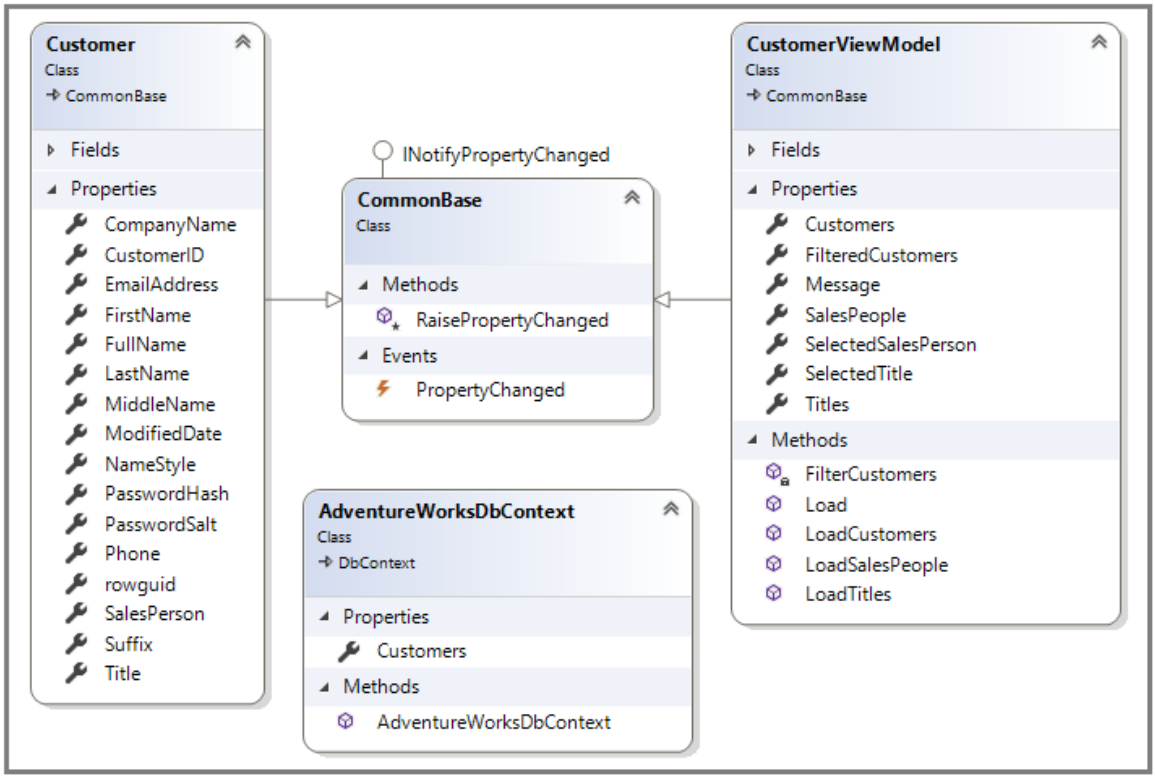

Figure 1: Overview of Customer classes

Customer Class

This class is a typical Entity Framework entity class. It contains a property for each column in the Customer table in the AdventureWorksLT database. The appropriate data annotations are added to the class and properties. Download the sample project at the end of this blog to view this class.

Customer View Model Class

You should always be using a Model-View-View-Model (MVVM) approach in your WPF applications. In this application, the CustomerViewModel class is used to bind to your WPF user control where the user interface (UI) is created.

Properties in the Customer View Model Class

There are several properties and methods you need to create in this class as outlined in the table below.

Property Name	Data Type
Customers	ObservableCollection<Customer>
FilteredCustomers	ObservableCollection<Customer>

Message	string
SalesPeople	ObservableCollection<string>
SelectedSalesPerson	string
SelectedTitle	string
Titles	ObservableCollection<string>

Methods in the Customer View Model Class

The methods in this view model class are used to load the titles, sales persons and customers from the Customer table. There is also a method to filter the customer data after it has been loaded into the *Customers* property.

LoadCustomers Method

The LoadCustomers() method uses the Entity Framework to read the list of customer records from the Customer table and place them into an ObservableCollection of Customer objects. Once all customer records are loaded, the *FilteredCustomers* property is assigned to the *Customers* property.

All the DataGrid controls you use in this blog post are bound to the *FilteredCustomers* property. The *Customers* property in this view model should always contain the complete list of customer records read from the table. The *FilteredCustomers* property is always going to contain the result of the user filtering the data. Using two properties like this lessens the amount of times you need to read data from your SQL Server. This saves time and computing resources.

```
public void LoadCustomers()
{
  AdventureWorksDbContext db = null;

  Customers = new ObservableCollection<Customer>();
  try {
    db = new AdventureWorksDbContext();
    Customers = new ObservableCollection<Customer>(db.Customers);
  }
  catch (Exception ex) {
    Message = ex.ToString();
  }

  FilteredCustomers = Customers;
}
```

LoadTitles Method

In the Customer table, there is a Title field that contains the title of each customer. In this blog post, you are going to be filtering the customer data by selecting a single title on which to filter. The *Titles* property is a collection of string values to which you can bind to a ComboBox so all titles within the customer table can be displayed.

After the LoadCustomers() method is called, the *Customers* property is loaded with all current customers. Call The LoadTitles() method after this method and it will select all distinct *Title* properties from the *Customers* property.

After all titles are loaded, a value of "All" is inserted into the first position in this collection. This value is displayed as the first entry in a ComboBox. Next, the *SelectedTitle* property is set to the value of "All" to force the data binding on the ComboBox to select that row in the collection and display it in the ComboBox.

```
public void LoadTitles()
{
  Titles = new ObservableCollection<string>(
    this.Customers.Select(c => c.Title).Distinct());

  Titles.Insert(0, "All");

  SelectedTitle = "All";
}
```

LoadSalesPeople Method

In the Customer table, there is a SalesPerson field that contains the name of a sales person of each customer. In this blog post, you are going to be filtering the customer data by selecting a single sales person on which to filter. The *SalesPeople* property is a collection of string values to which you can bind to a ComboBox so all sales people within the customer table can be displayed. After the LoadCustomers() method is called, the *Customers* property is loaded with all current customers. Call The LoadSalesPeople() method after this method and it will select all distinct *SalesPerson* properties from the *Customers* property.

After all sales people are loaded, a value of "All" is inserted into the first position in this collection. This value is displayed as the first entry in a ComboBox. Next, the *SelectedSalesPerson* property is set to the value of "All" to force the data binding on the ComboBox to select that row in the collection and display it in the ComboBox.

```
public void LoadSalesPeople()
{
  SalesPeople = new ObservableCollection<string>(
    this.Customers.Select(c => c.SalesPerson).Distinct());

  SalesPeople.Insert(0, "All");

  SelectedSalesPerson = "All";
}
```

Load Method

Instead of having to call all three of the above methods from the code behind in your WPF user control, create a method named Load() to call each of these three

methods. Make sure the LoadCustomers() method is called first so the customer data is loaded before you attempt to load distinct titles and sales people.

```
public void Load()
{
  // Load customers before loading titles and sales people
  LoadCustomers();
  LoadTitles();
  LoadSalesPeople();
}
```

FilterCustomers() Method

When the user selects either a customer title or sales person from a ComboBox, you should call the FilterCustomers() method. This method uses LINQ to apply a Where clause to the *Customers* collection to set the *FilteredCustomers* collection to the result of the data found.

```
private void FilterCustomers()
{
  FilteredCustomers = new ObservableCollection<Customer>(
    Customers.Where(c =>
      (SelectedTitle == "All" ? true : c.Title == SelectedTitle)
   && (SelectedSalesPerson == "All" ? true :
         c.SalesPerson == SelectedSalesPerson)));
}
```

Encapsulate Filter Criteria in an Expander

On the screen shown in Figure 2, an expander control is used to encompass two combo box controls. The first combo box is filled with the distinct list of titles created in the LoadTitles() method of your customer view model. The second combo box is filled with the distinct list of sales people created in the LoadSalesPeople() method. This UI approach is good if you want the ability to hide the filter criteria when you first come into the screen. It also works well if users don't frequently filter the data.

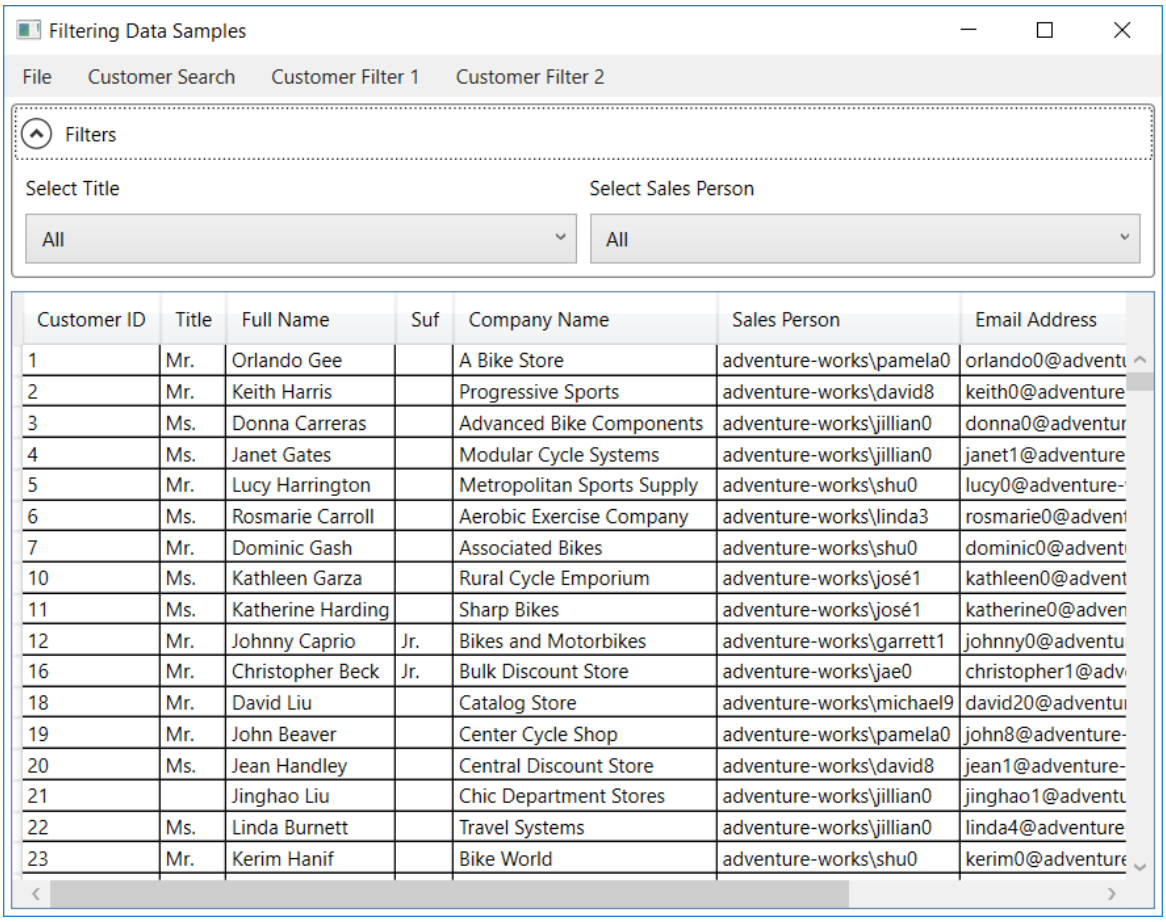

Figure 2: Use an Expander control to encompass filters so the area may be collapsed

To create the screen displayed in Figure 2, Add an XML namespace to a user control to reference the CustomerViewModel class and a Loaded event procedure. I have named my user control **CustomerSearchControl**, but feel free to call it whatever you wish.

```
<UserControl
  ...
  xmlns:vm="clr-namespace:FilterDataSample.ViewModels"
  Loaded="UserControl_Loaded">
```

Add a UserControl.Resources element in your user control and create an instance of the CustomerViewModel. Assign a key name of "viewModel" to this resource.

```
<UserControl.Resources>
  <vm:CustomerViewModel x:Key="viewModel" />
</UserControl.Resources>
```

Add the DataContext attribute to the <Grid> element and bind it to the view model created in the resources. Add two row definitions; one for the search expander and one for the DataGrid.

```
<Grid DataContext="{Binding Source={StaticResource viewModel}}">
  <Grid.RowDefinitions>
    <RowDefinition Height="Auto" />
    <RowDefinition Height="*" />
  </Grid.RowDefinitions>

</Grid>
```

Add Search Expander

In the first row of the Grid control, add an Expander control to enclose the search fields. You do not have to use an Expander, but I would recommend you use a GroupBox or some other container control to separate the search area from the DataGrid control. Add the following XAML just below the </Grid.RowDefinition> element.

```
<Expander Grid.Row="0"
          Header="Filters">
  <Grid>
    <Grid.RowDefinitions>
      <RowDefinition Height="Auto" />
      <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="*" />
      <ColumnDefinition Width="*" />
    </Grid.ColumnDefinitions>
    <TextBlock Grid.Row="0"
               Grid.Column="0"
               Text="Select Title" />
    <ComboBox Grid.Row="1"
              Grid.Column="0"
              SelectedValue="{Binding Path=SelectedTitle}"
              ItemsSource="{Binding Path=Titles}" />
    <TextBlock Grid.Row="0"
               Grid.Column="1"
               Text="Select Sales Person" />
    <ComboBox Grid.Row="1"
              Grid.Column="1"
              SelectedValue="{Binding Path=SelectedSalesPerson}"
              ItemsSource="{Binding Path=SalesPeople}" />
  </Grid>
</Expander>
```

Set the first ComboBox control's *ItemsSource* property to the *Titles* property in the customer view model class. Set the *SelectedValue* property to the *SelectedTitle* property in the view model. The second ComboBox bind the ItemsSource property

to the *SalesPeople* property and the *SelectedValue* property to the *SelectedSalesPerson* property. As you remember, these values are set when you call the LoadTitles() and LoadSalesPeople() methods in the view model.

Add a DataGrid

Below the Expander, add a DataGrid control. Set the *AutoGenerateColumns* property to false because you are going to be adding your own columns to this grid. I set the *IsReadOnly* property to true so no editing can be done within the grid, but feel free to change this if you want. Finally, set the *ItemsSource* property to the *FilteredCustomers* property from the view model. Initially, this property is set to all customers, but this collection will change as you select different filtering criteria from the two ComboBox controls.

```
<DataGrid Grid.Row="1"
          AutoGenerateColumns="False"
          IsReadOnly="True"
          ItemsSource="{Binding Path=FilteredCustomers}">
  <DataGrid.Columns>
    <DataGridTextColumn Binding="{Binding Path=CustomerID}"
                        Header="Customer ID" />
    <DataGridTextColumn Binding="{Binding Path=Title}"
                        Header="Title" />
    <DataGridTextColumn Binding="{Binding Path=FullName}"
                        Header="Full Name" />
    <DataGridTextColumn Binding="{Binding Path=Suffix}"
                        Header="Suffix" />
    <DataGridTextColumn Binding="{Binding Path=CompanyName}"
                        Header="Company Name" />
    <DataGridTextColumn Binding="{Binding Path=SalesPerson}"
                        Header="Sales Person" />
    <DataGridTextColumn Binding="{Binding Path=EmailAddress}"
                        Header="Email Address" />
    <DataGridTextColumn Binding="{Binding Path=Phone}"
                        Header="Phone" />
  </DataGrid.Columns>
</DataGrid>
```

Add Code to Load Customers

In order to retrieve the list of customers to display within the DataGrid, add some code in the code-behind for this user control. Add a private field to hold an instance of a customer view model.

```
CustomerViewModel _viewModel = null;
```

Set this field by retrieving the view model instance created by the XAML. In the constructor of this user control, get the instance by calling the FindResource() method and passing in the key set in the UserControl.Resources element.

```
public CustomerSearchControl()
{
  InitializeComponent();

  _viewModel = (CustomerViewModel)this.FindResource("viewModel");
}
```

Call the Load() method in the view model from the UserControl_Loaded event procedure. This method loads the customers into the *FilteredCustomers* property that is bound to the DataGrid. It also loads the titles and the sales people bound to the two ComboBox controls.

```
private void UserControl_Loaded(object sender, RoutedEventArgs e) {
  _viewModel.Load();
}
```

Filtering Data

Now that you have the two ComboBox controls and the DataGrid bound to the view model, and you have loaded the customer data, you can run the application and see the data displayed within each of the controls. It's now time to filter the customer data when you change either of the ComboBox controls to a different title or sales person. You do this by calling the FilterCustomers() method in the customer view model when either the *SelectedTitle* or *SelectedSalesPerson* properties are changed.

SelectedTitle Property

In the customer view model class, locate the *SelectedTitle* property and call the FilterCustomers() method from within the **set** procedure.

```
public string SelectedTitle
{
  get { return _SelectedTitle; }
  set {
    if (_SelectedTitle != value) {
      _SelectedTitle = value;
      RaisePropertyChanged("SelectedTitle");

      // Filter Customers based on selected title
      FilterCustomers();
    }
  }
}
```

SelectedSalesPerson Property

In the customer view model class, locate the *SelectedSalesPerson* property and call the FilterCustomers() method from within the **set** procedure.

```
public string SelectedSalesPerson
{
  get { return _SelectedSalesPerson; }
  set {
    if (_SelectedSalesPerson != value) {
      _SelectedSalesPerson = value;
      RaisePropertyChanged("SelectedSalesPerson");

      // Filter Customers based on selected Sales Person
      FilterCustomers();
    }
  }
}
```

Try it Out

These two changes are all you need to allow filtering of the customer data when the user selects a new title or sales person from the two ComboBox controls within the Expander.

Filter in Column Header

Now that you have the code written to filter customers and a DataGrid to display those filtered customers, let's change our UI a little. In Figure 3, you see the ComboBox controls have been removed from the Expander control and are instead right in the header of the appropriate columns in the DataGrid. Integrating these controls directly in the header makes the UI a little more compact.

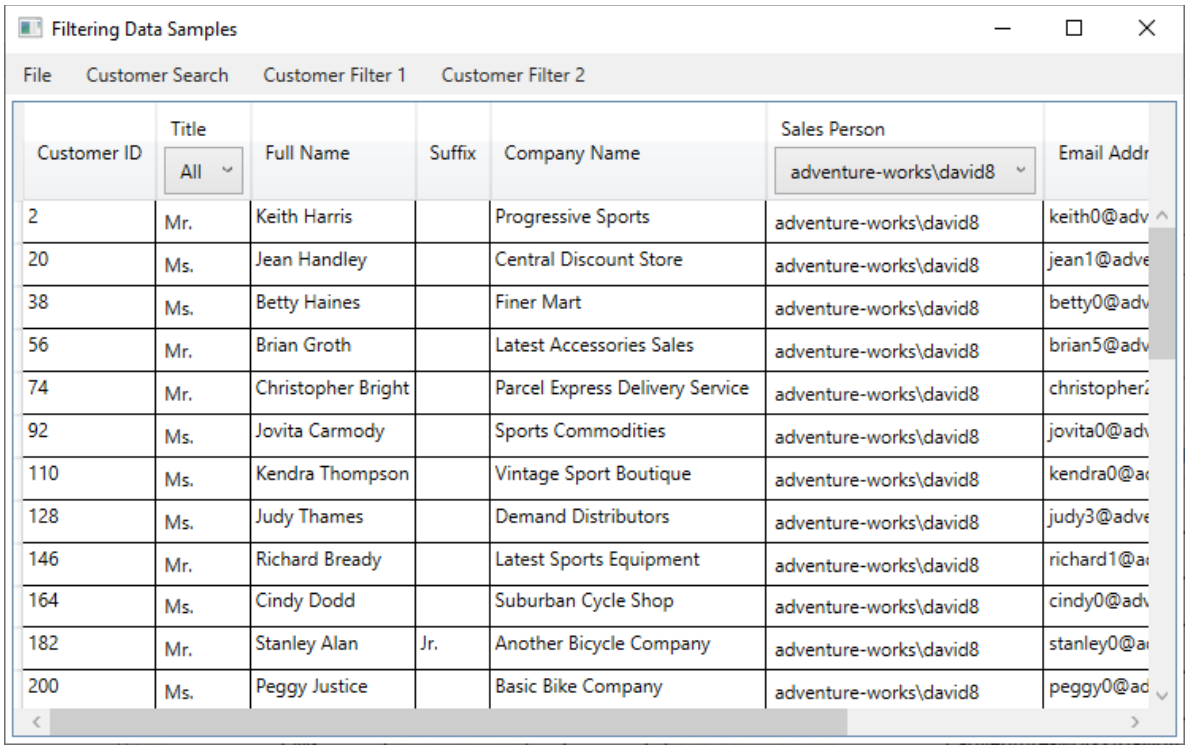

Figure 3: Add filtering controls to the headers in the DataGrid

Add a Style

To create this new DataGrid, first add a style in the <UserControl.Resources> element to reduce the margin of the ComboBox controls to zero. If you don't add this control, the header area in the Title and SalesPerson columns will look too large.

```
<Style TargetType="ComboBox">
  <Setter Property="Margin"
          Value="0" />
</Style>
```

Replace the Title DataGridTextColumn

In the previous DataGrid, you used DataGridTextColumn objects to build each column of the grid. However, you now need to use a DataGridTemplateColumn so you can define the header area of the Title and SalesPerson columns. Locate the DataGridTextColumn for the Title column; it looks like the following:

```
<DataGridTextColumn Binding="{Binding Path=Title}"
                    Header="Title" />
```

Replace the above line of XAML with the following lines:

```
<DataGridTemplateColumn>
  <DataGridTemplateColumn.CellTemplate>
    <DataTemplate>
      <TextBlock Text="{Binding Path=Title}" />
    </DataTemplate>
  </DataGridTemplateColumn.CellTemplate>
  <DataGridTemplateColumn.HeaderTemplate>
    <DataTemplate>
      <Grid>
        <Grid.RowDefinitions>
          <RowDefinition Height="Auto" />
          <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>
        <TextBlock Grid.Row="0"
                   Text="Title" />
        <ComboBox Grid.Row="1"
                  SelectedValue="{Binding Path=SelectedTitle,
Source={StaticResource viewModel}}"
                  ItemsSource="{Binding Path=Titles,
Source={StaticResource viewModel}}" />
      </Grid>
    </DataTemplate>
  </DataGridTemplateColumn.HeaderTemplate>
</DataGridTemplateColumn>
```

The CellTemplate uses a TextBlock control to bind to the *Title* property of the customer view model. The HeaderTemplate is where you add a significant amount of new XAML code. The header for the Title, as shown in Figure 3, needs one row for the label, and the next row for the ComboBox.

Define a Grid and add two RowDefinitions. In the first row, place a TextBlock to display the text "Title". In the next row, you can cut out the ComboBox that displays the *Titles* from the Expander control, and place that ComboBox into this row.

Replace the Sales Person DataGridTextColumn

Locate the DataGridTextColumn for the sales person column. It looks like the following:

```
<DataGridTextColumn Binding="{Binding Path=SalesPerson}"
                    Header="Sales Person" />
```

Replace this XAML with another DataGridTemplateColumn and create the same type of XAML you used for the Sales Person column.

```
<DataGridTemplateColumn>
  <DataGridTemplateColumn.CellTemplate>
    <DataTemplate>
      <TextBlock Text="{Binding Path=SalesPerson}" />
    </DataTemplate>
  </DataGridTemplateColumn.CellTemplate>
  <DataGridTemplateColumn.HeaderTemplate>
    <DataTemplate>
      <Grid>
        <Grid.RowDefinitions>
          <RowDefinition Height="Auto" />
          <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>
        <TextBlock Grid.Row="0"
                   Text="Sales Person" />
        <ComboBox Grid.Row="1"
                  SelectedValue="{Binding Path=SelectedSalesPerson,
Source={StaticResource viewModel}}"
                  ItemsSource="{Binding Path=SalesPeople,
Source={StaticResource viewModel}}" />
      </Grid>
    </DataTemplate>
  </DataGridTemplateColumn.HeaderTemplate>
</DataGridTemplateColumn>
```

Remove the Expander Control

After you have moved all the XAML out of the Expander control, remove this expander as it is no longer needed. Run the application and ensure the new columns look correct and that they still filter the customer data.

Align Headers

The problem with the UI shown in Figure 3 is the headers for the Title and Sales Person columns do not line up horizontally with the other columns in the grid. You can fix this by defining a new ContentTemplate for the DataGridColumnHeader control (see Figure 4). This new style will apply to any DataGridTextColumn controls added to the DataGrid. However, when you use a DataGridTemplateColumn, you are defining a new header template, so this style will not apply.

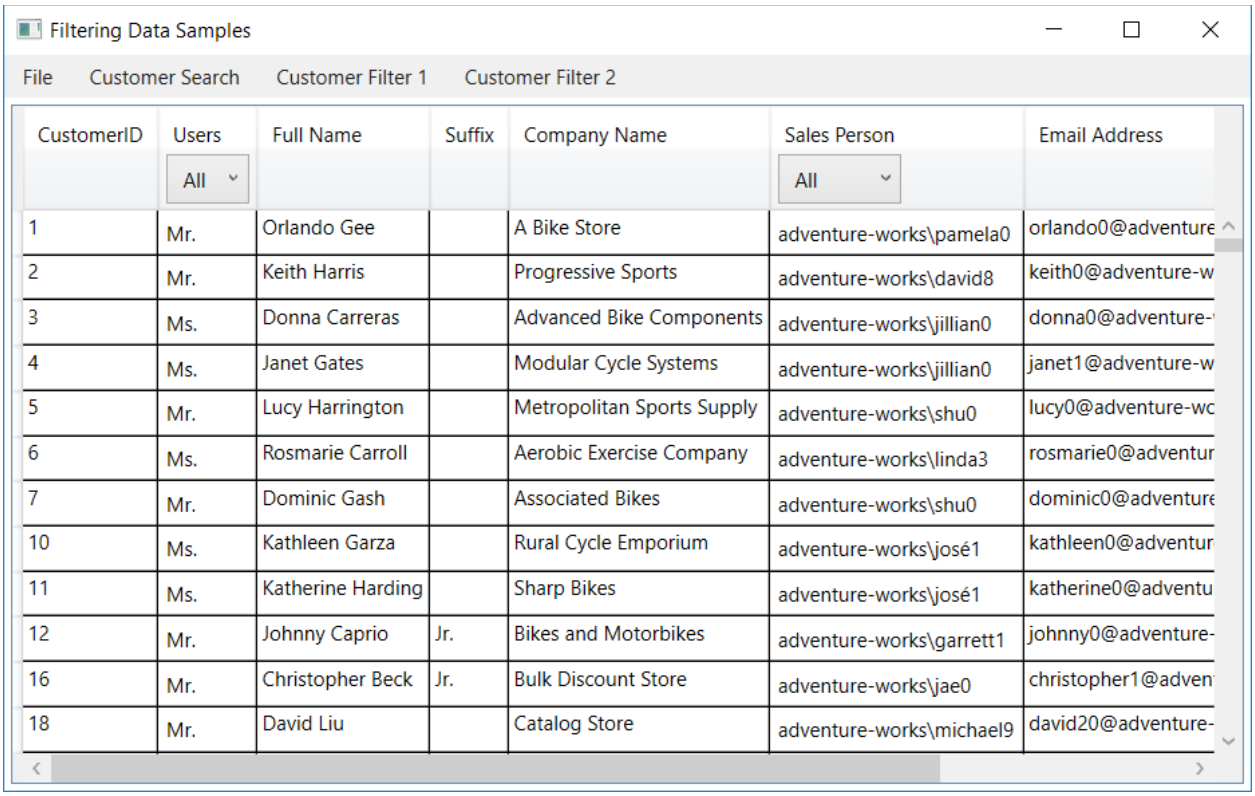

Figure 4: Align headers in the DataGrid by creating your own ContentTemplate

Add a Resource to DataGrid

Define a <DataGrid.Resources> element within the DataGrid control. In this resources section, you may define a style that has a TargetType of "DataGridColumnHeader".

```
<DataGrid AutoGenerateColumns="False"
          IsReadOnly="True"
          ItemsSource="{Binding Path=FilteredCustomers}">
  <DataGrid.Resources>
    <Style TargetType="DataGridColumnHeader">
      <Setter Property="ContentTemplate">
        <Setter.Value>
          <DataTemplate>
            <Grid>
              <Grid.RowDefinitions>
                <RowDefinition Height="Auto" />
                <RowDefinition Height="Auto" />
              </Grid.RowDefinitions>
              <TextBlock Text="{Binding}"
                         TextWrapping="Wrap" />
              <ComboBox Grid.Row="1"
                        Visibility="Hidden" />
            </Grid>
          </DataTemplate>
        </Setter.Value>
      </Setter>
    </Style>
  </DataGrid.Resources>
  <DataGrid.Columns>

  ... // The rest of the XAML HERE
```

In the style defined for the DataGridColumnHeader, you set the *ContentTemplate* property to a new DataTemplate. In the DataTemplate, create a Grid that has two rows. The first row contains a TextBlock in which you use the Binding keyword. This binds whatever you set in the *Header* property to the text of this TextBlock control. The second row contains a ComboBox with its *Visibility* property set to Hidden. Setting this to hidden allows the space for the ComboBox to be there, but the ComboBox will not be shown. As you are using a ComboBox in the Title and Sales Person columns, you want all the other columns to have a ComboBox control that takes up the same exact space. This way, you get all your columns to align as shown in Figure 4.

Summary

In this blog post, you learned how to filter data using Entity Framework, LINQ, and WPF. You were shown three different UI's on how you might present the data to your user for filtering. By moving filtering controls within the DataGrid, you can save some real estate on your UI. Using an Expander control for your search criteria controls allows you to have other kinds of filtering that might not fit well within a DataGrid header.

NOTE: You can download the sample code for this article by visiting my website at http://www.pdsa.com/downloads. Select “Fairway/PDSA Blog,” then select “Filter Data in WPF DataGrid” from the dropdown list.