

12/10/2008



MICROSOFT
SWITZERLAND

WINDOWS PRESENTATION FOUNDATION - LINE OF BUSINESS HANDS-ON LAB

Based on WPF 3.5 and the WPF Toolkit, October CTP | Sascha P. Corti

Hands-On Lab created and written by Sascha P. Corti
(sascha.corti@microsoft.com)

Sample code and lots of ideas contributed by Ronnie Saurenmann
(ronnie.saurenmann@microsoft.com)

Special thanks to the beta testers:
Ken Casada
Olaf Feldkamp

Document Version 1.4 - English

Special Annotations

Throughout the hands-on lab you will find the following annotations:



Goal: This explains the goal of the current chapter.



Note: Notes inform you of an important aspect of the programming steps you were just taking.



Hint: Hints may point out other possibilities to reach a goal.

Source code Snippets:

```
<Grid Height="120" HorizontalAlignment="Stretch" VerticalAlignment="Top">
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="50"/>
    <ColumnDefinition Width="*"/>
  </Grid.ColumnDefinitions>
</Grid>
```

Source Code Snippet 2a

- Each snippet has an associated number and can be found on the Hands-on lab CD-Rom in the file "Source Code Snippets.txt" under this number.

Index

Special Annotations	3
Introduction.....	5
Goal.....	5
Prerequisites	6
Getting Started.....	7
Step 1: Creating the Solution and adding the Data Source.....	7
Step 2: Creating the basic User Interface.....	12
Step 3: Wiring the User Interface to the Data Source.....	20
Step 3.1: Loading Data from the Database	21
Step 3.2: Saving Data back to the Database.	22
Step 4: Implementing an Inline Master/Details View	23
Step 5: Changing the Columns shown in the DataGrid.....	25
Step 6: Adding custom Controls to the DataGrid.....	27
Step 6.1: Showing the Employee in a ComboBox.	27
Step 6.2: Showing a DatePicker when editing OrderDate.....	30
Step 6.3: Color-Coding the Freight Column.	31
Step 7: Adding a Details Pane containing Customer Data.....	34
Step 8: Creating a Custom Control to visualize Data.....	37
Step 9: Printing.....	48
Step 10: Styling the Application	52
Step 11: Utilizing the Ribbon Control.....	56
Step 11.1: Adding the basic Ribbon Control.....	56
Step 11.2: Customizing the Ribbon Control.....	63
Step 11.3: Adding a Quick Access Toolbar to the Ribbon.....	71
Step 11.4: Providing Vista Glass Effects.	73

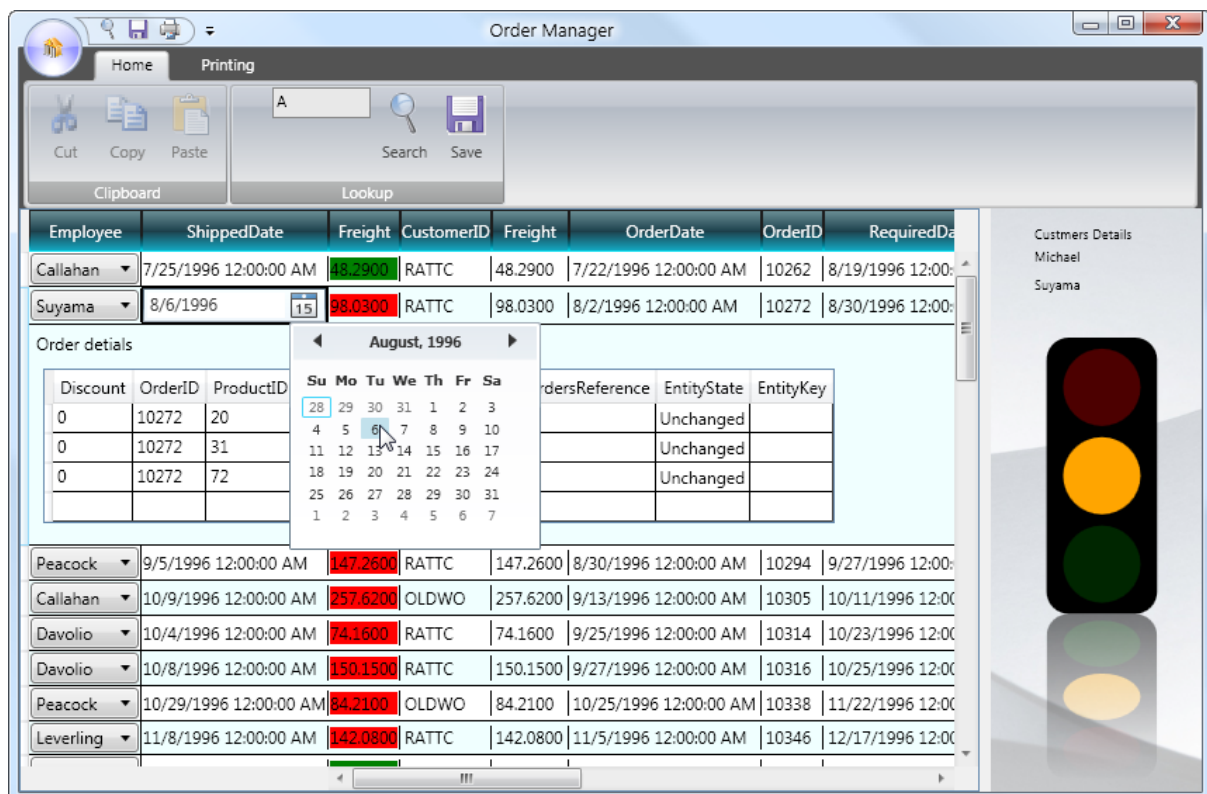
Introduction

Windows Presentation Foundation together with the new WPF control toolkit offers a whole new set of controls to build state of the art business desktop applications where classically Windows Forms would have been used.

Goal

In this half-day hands-on lab you will learn how to use WPF and the Control Toolkit to create such an end-to-end, data bound line of business application. You will leverage the Entity Framework to connect to the database, implement the new Data Grid to visualize data and the Ribbon control and control styling to create a state a of the art user experience. Finally you will make your application capable of printing.

The application itself is an order management system based on the Northwind sample database from Microsoft. The final user interface will look like this:



Prerequisites

Visual Studio 2008 Professional

- Trial download: <http://msdn.microsoft.com/en-us/vstudio/products/aa700831.aspx>

Microsoft SQL Server 2008 Express

- Trial download: <http://www.microsoft.com/express/sql/download/>



Hint: The basic version, “SQL Server 2008 Express” is sufficient for this project but optionally installing “SQL Server 2008 Express with Tools” gives you a SQL Server Management Studio that comes in very handy when configuring SQL Server for your own projects.

Visual Studio 2008 & .NET Framework 3.5 Service Pack 1

- Download: <http://msdn.microsoft.com/en-us/vstudio/cc533448.aspx>

Expression Blend 2

- Trial download:
<http://www.microsoft.com/downloads/details.aspx?FamilyId=5FF08106-B9F4-43CD-ABAD-4CC9D9C208D7&displaylang=en>

Expression Blend 2 SP 1

- Download: <http://www.microsoft.com/downloads/details.aspx?FamilyId=EB9B5C48-BA2B-4C39-A1C3-135C60BBBE66&displaylang=en>

WPF Toolkit Binaries

- Download, extract and install the “WPF Toolkit Binaries”. The installer will place the DLLs in the “C:\Program Files\WPF Toolkit” folder. They contain the WPF DataGrid and DatePicker controls that we will be using in our project.
- Download:
<http://www.codeplex.com/wpf/Release/ProjectReleases.aspx?ReleaseId=15598>



Note: The **WPF ribbon** is a **last, optional step** in this hands-on lab. As everyone who wants the bits needs to sign an online Office UI license, we cannot provide these bits on the hands-on lab CD-Rom. So if you don’t plan on implementing this last, final step, you may skip the following download.

WPF Ribbon Preview

- You can download a preview version of the WPF Ribbon from the Office UI Licensing Site:
<http://www.codeplex.com/wpf/Wiki/View.aspx?title=WPF%20Ribbon%20Preview>
 - Follow the link above and click on “License the Office UI”
 - Login using your Windows Live ID. If you don’t have a Windows Live ID already, you can create one for free.
 - Read and accept the Office UI License
 - Licensing the Office UI (including the Ribbon) is free. You must accept the license in order to access the WPF Ribbon download.

- Click on the WPF Ribbon download button
- Accept the WPF Ribbon CTP License and follow the instructions to save the Ribbon binaries to your computer
- Click on the buttons for "2007 Microsoft Office Fluent UI Design Guidelines License" and "Microsoft Fluent Third Party Usage Guidelines" to download the Office UI Licensing Guidelines
 - By signing the Office UI License you have agreed to abide by the Office UI Licensing Guidelines when developing your WPF Ribbon application. Please make sure that you are familiar with the guidelines and that your Ribbon is compliant with the requirements outlined in the guideline documents. This will ensure that your Ribbon UI delivers a high quality end-user experience and is within the terms of the license.

Getting Started

Create a new folder "C:\WPF HOL".



Note: You can create this folder wherever you like but the lab manual will be pointing to this path.

Copy the content from the Hands-On Lab CD-ROM to this folder.

Step 1: Creating the Solution and adding the Data Source.



Goal: First we want to set up an empty solution, add the existing Northwind database and create an ADO.NET entity data model to access the data.

- Start Visual Studio and select "**New → Project**" from the "**File**"-Menu.
- Choose "**Visual C#**" as language and "**WPF Application**" as template.
- Select "C:\WPF HOL\Project" as location and name your project "**WpfLobApp**".

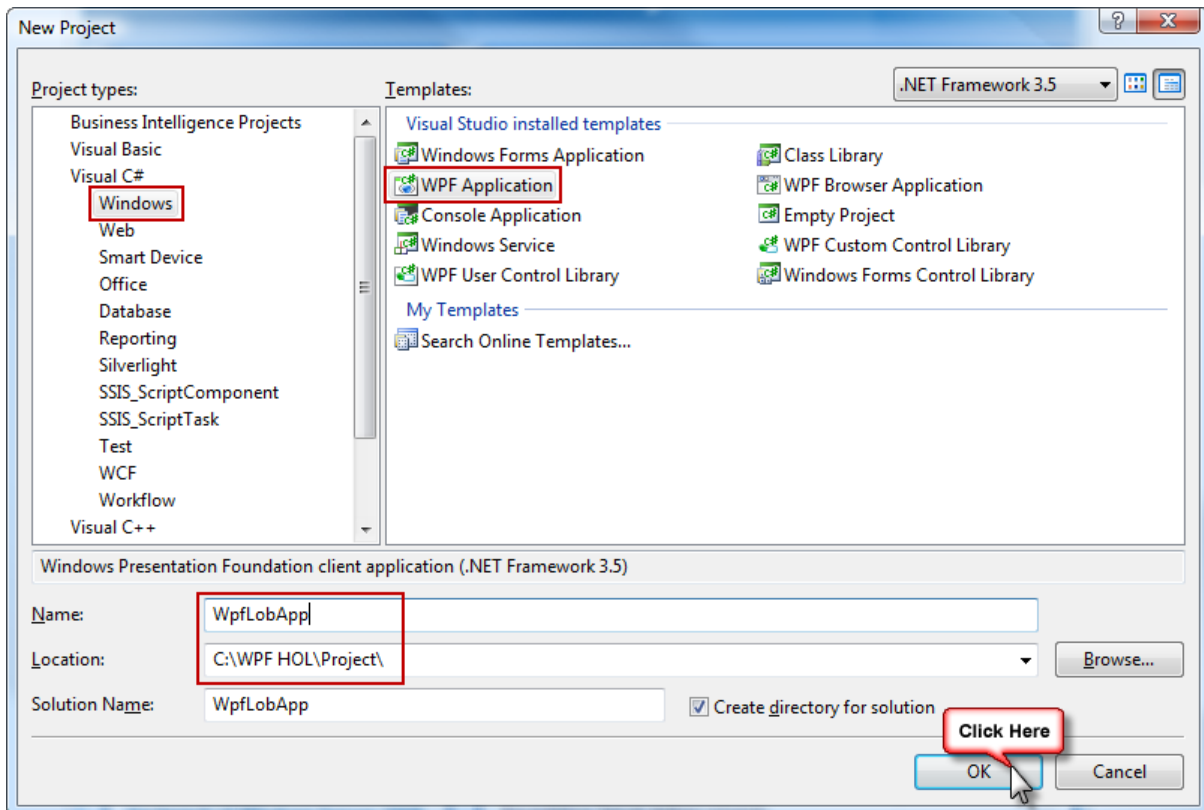


Figure 1.1

- To add the data source, right click on the project name “**WpfLobApp**” in the solution explorer and click “**Add → New Item**”.

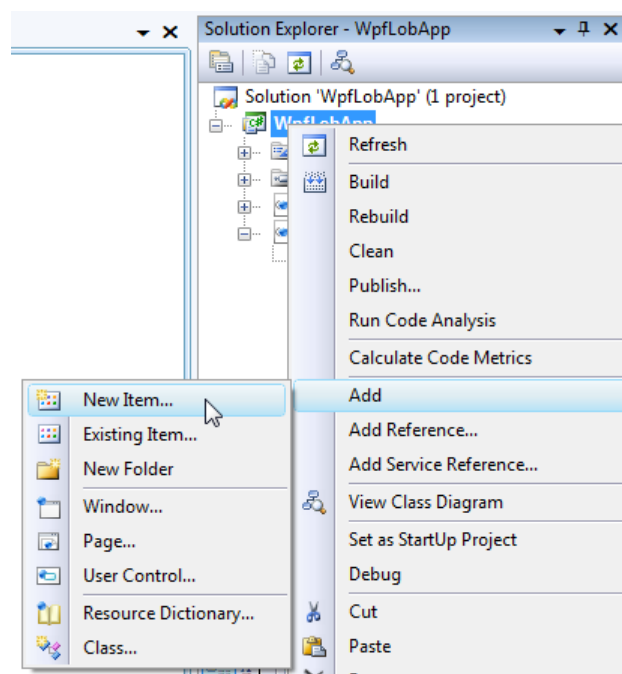


Figure 1.2

- In the “**Add New Item**” dialog, select “**ADO.NET Entity Data Model**”. Name it “**NorthwindModel.edmx**”.

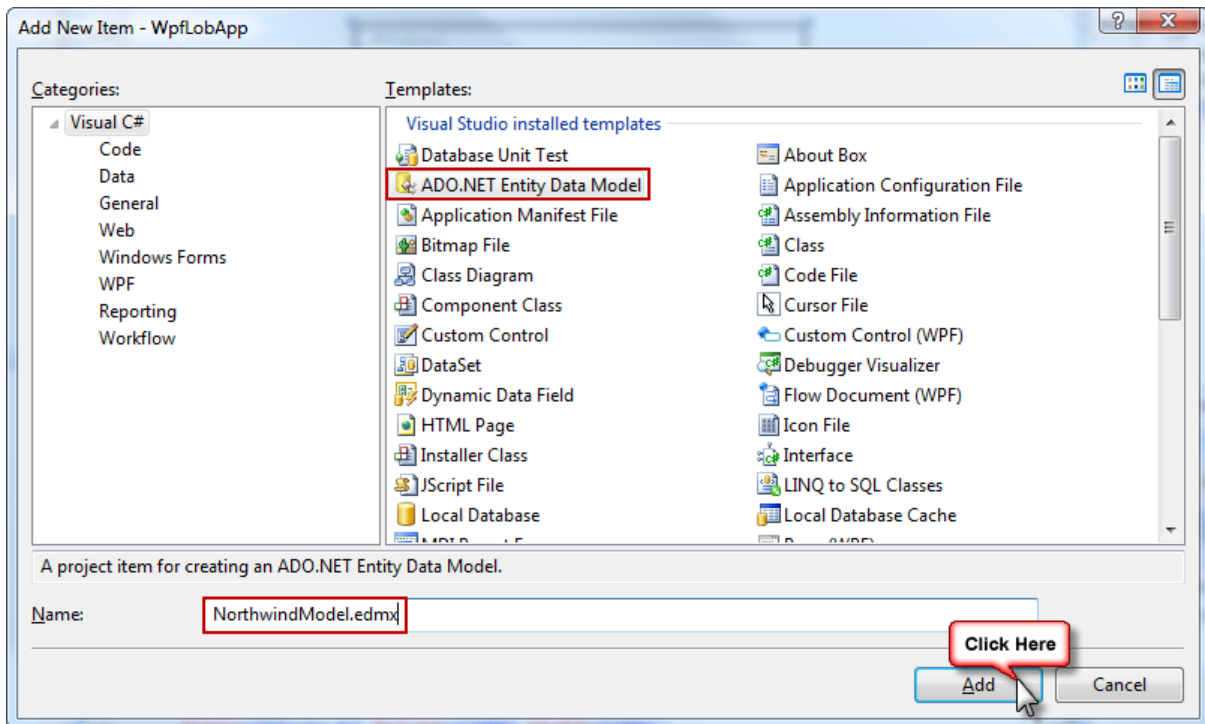


Figure 1.3

This will launch the “**Entity Data Model Wizard**”. Complete the wizard as follows:

- For “**Choose Model Contents**” select “**Generate from Database**”.

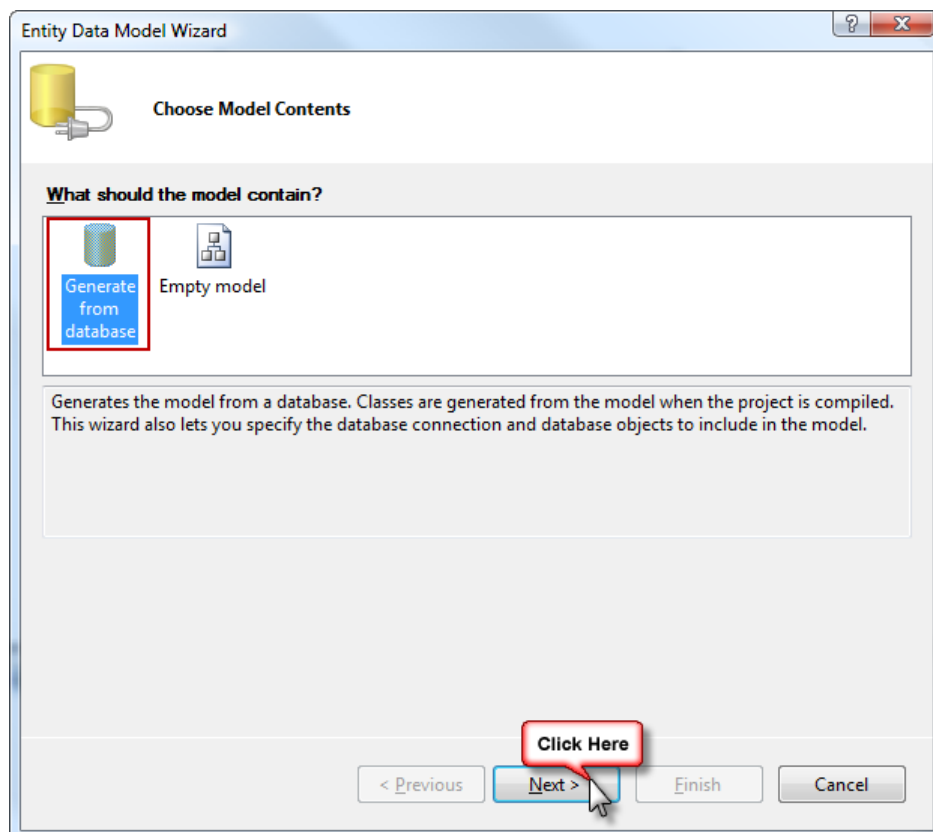


Figure 1.4

- In the step “**Choose your Data Connection**” click the “**New Connection**” button, select “**Microsoft SQL Server Database File (SqlClient)**” as “**Data Source**” and browse to the database provided in “C:\WPF HOL\Database\northwnd.mdf”.



Note: The “Microsoft SQL Server Database File” data source belongs to the “.NET Framework Data Provider for SQL Server” and allows attaching to a database only when the application is running. This can be very handy if a database is to be deployed together with the solution.

- Click “**Test Connection**” and if this succeeds, hit “**OK**”.

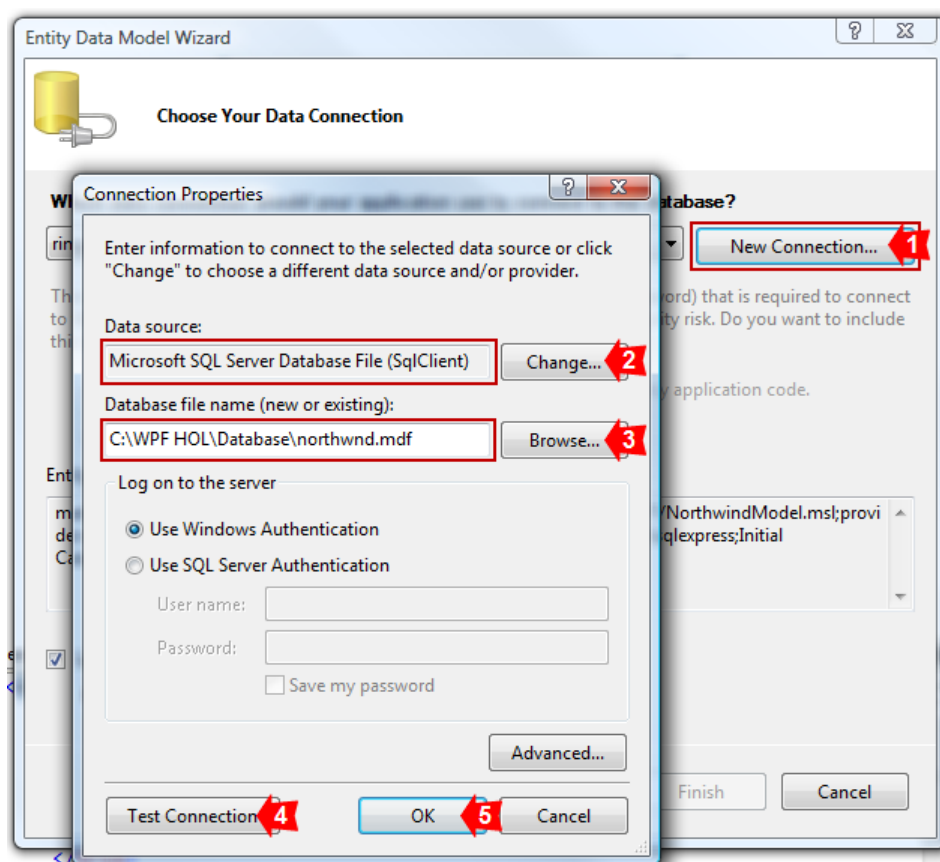


Figure 1.5

- The entity connection settings will be stored in the **App.config** file as “**northwindEntities**”. Confirm by clicking on “**Next**”.
- In the “**Choose Your Database Objects**” step select “**Employees**”, “**Order Details**” and “**Orders**” from the “**Tables**” group. We will be working with these three tables in our application. Accept the default Model Namespace of “**northwindModel**”

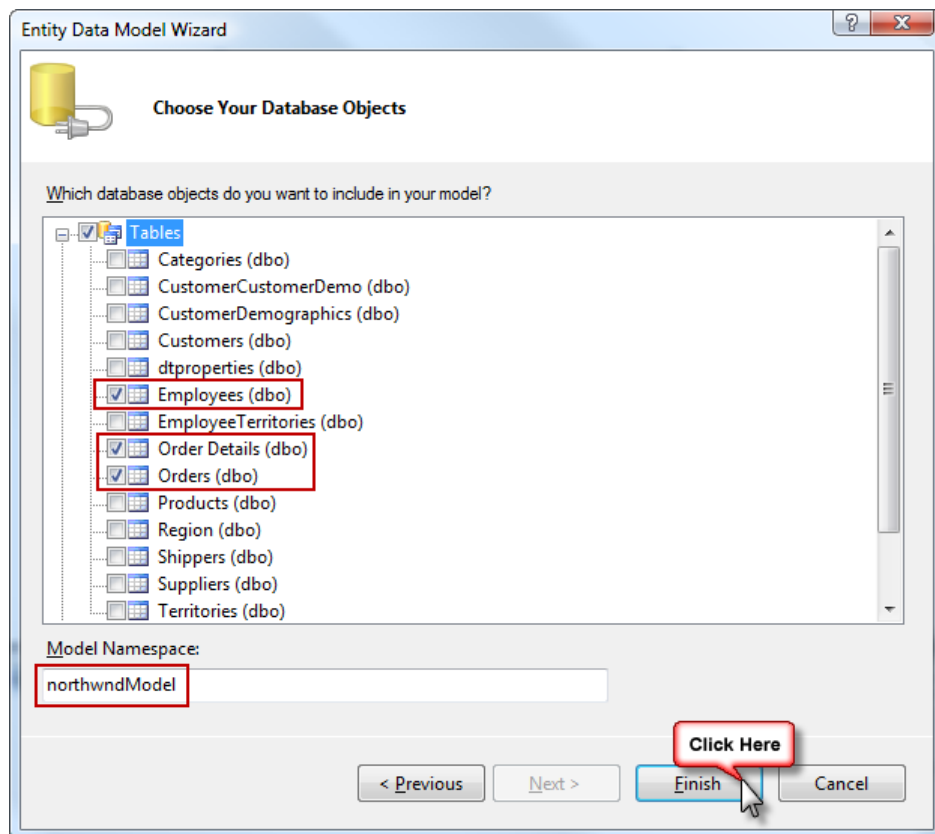


Figure 1.6

- At this point, all the required libraries will be referenced in the project and you will be presented with a view of the data model.

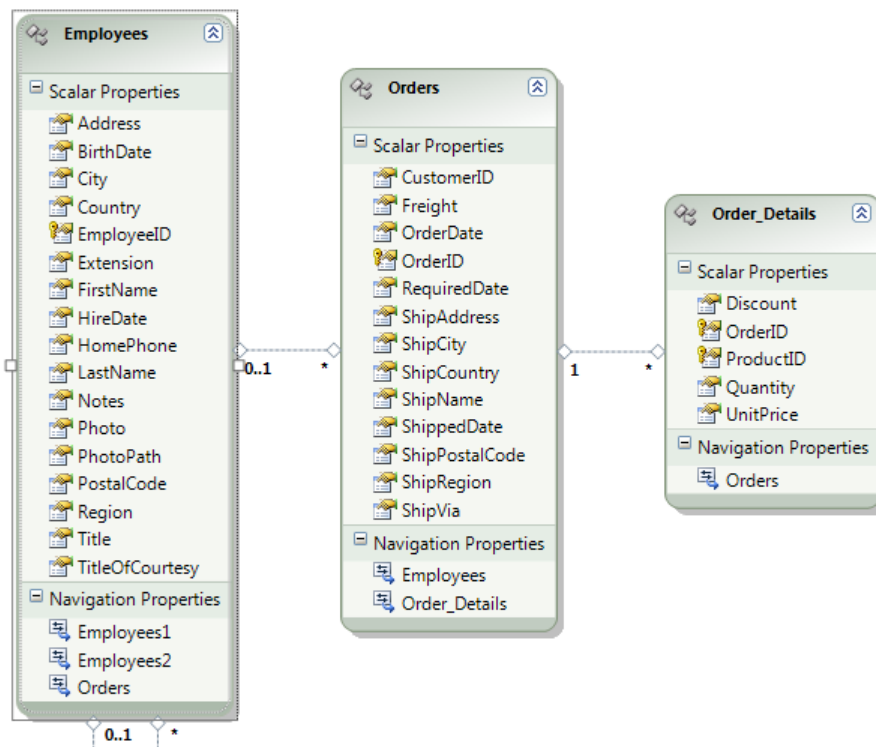


Figure 1.7



Note: Normally, this kind of application would be designed using a **3-tier architecture** using a layer of business objects and/or services that expose the data. To keep this user-interface centric lab simple however, we connect directly to the data source from within the application.



Hint: To learn in detail how to create a 3-tier database application using the Entities Framework, have a look at our following Hands-On Lab:

http://blogs.msdn.com/swiss_dpe_team/archive/2008/10/27/ado-net-entity-framework-hands-on-lab.aspx

Step 2: Creating the basic User Interface



Goal: We will now create the basic user interface and use a DataGrid control to visualize our data source. We will use the WPF designer built into Visual Studio for the basic application layout. We will later use Expression Blend to create a custom user control and to style our DataGrid.

- Still in Visual Studio, close the “**NorthwindModel.edmx**” data model and select the “**Window1.xaml**”, our main application window.
- Inspect the **Visual Studio Toolbox** and make sure that the “**WPF Toolkit**” section was added by installing this toolkit in the prerequisites.

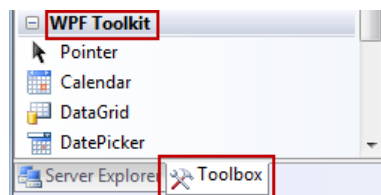


Figure 2.1



Hint: Should the toolbox entries be missing, they can be added as follows

- Right-click in the toolbox and select “**Add Tab**”.
- Name the tab “**WPF Toolkit**” and select “**Choose Items...**” from the new tab’s context menu.
- In the “**Choose Toolbox Items**” Dialog, switch to the “**WPF Components**” tab and click the “**Browse**” button.
- Navigate to “C:\Program Files\WPF Toolkit\v3.5.31016.1” and select “WPFToolkit.dll”.
- Sort the list by “**Assembly Name**” and make sure all three elements from the “**WPFToolkit**” assembly are checked. Hit “**OK**”.

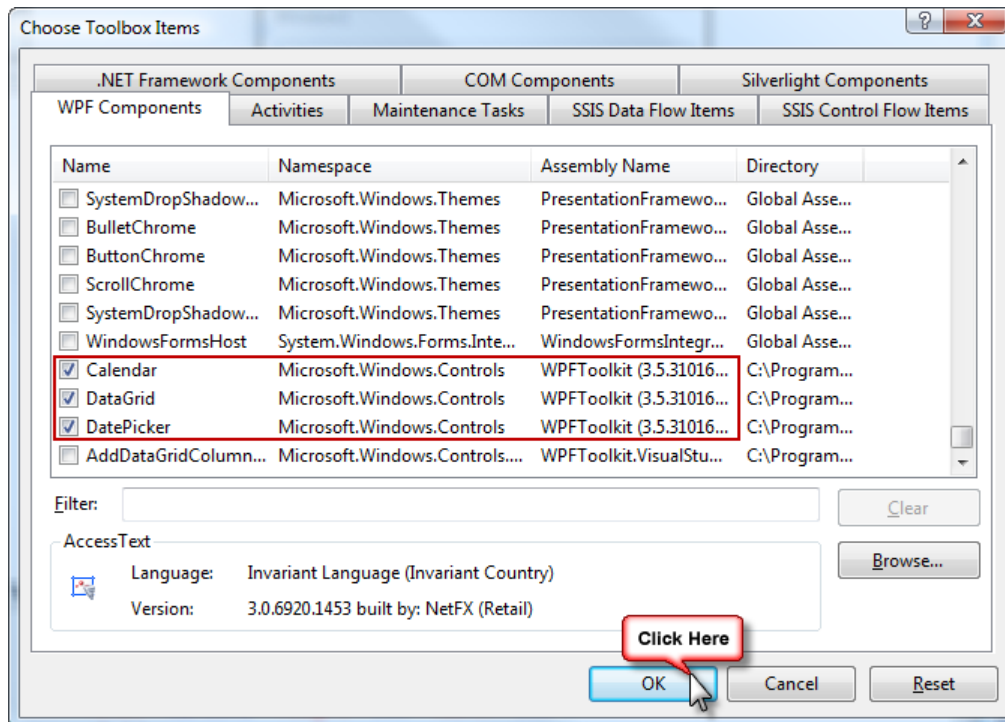


Figure 2.2

- In **Window1.xaml**, click the **outer window border** on the **design surface** (1). This will mark the **<Window>** element in the **XAML source view** (2) and show its **properties** in the **properties pane** (3).

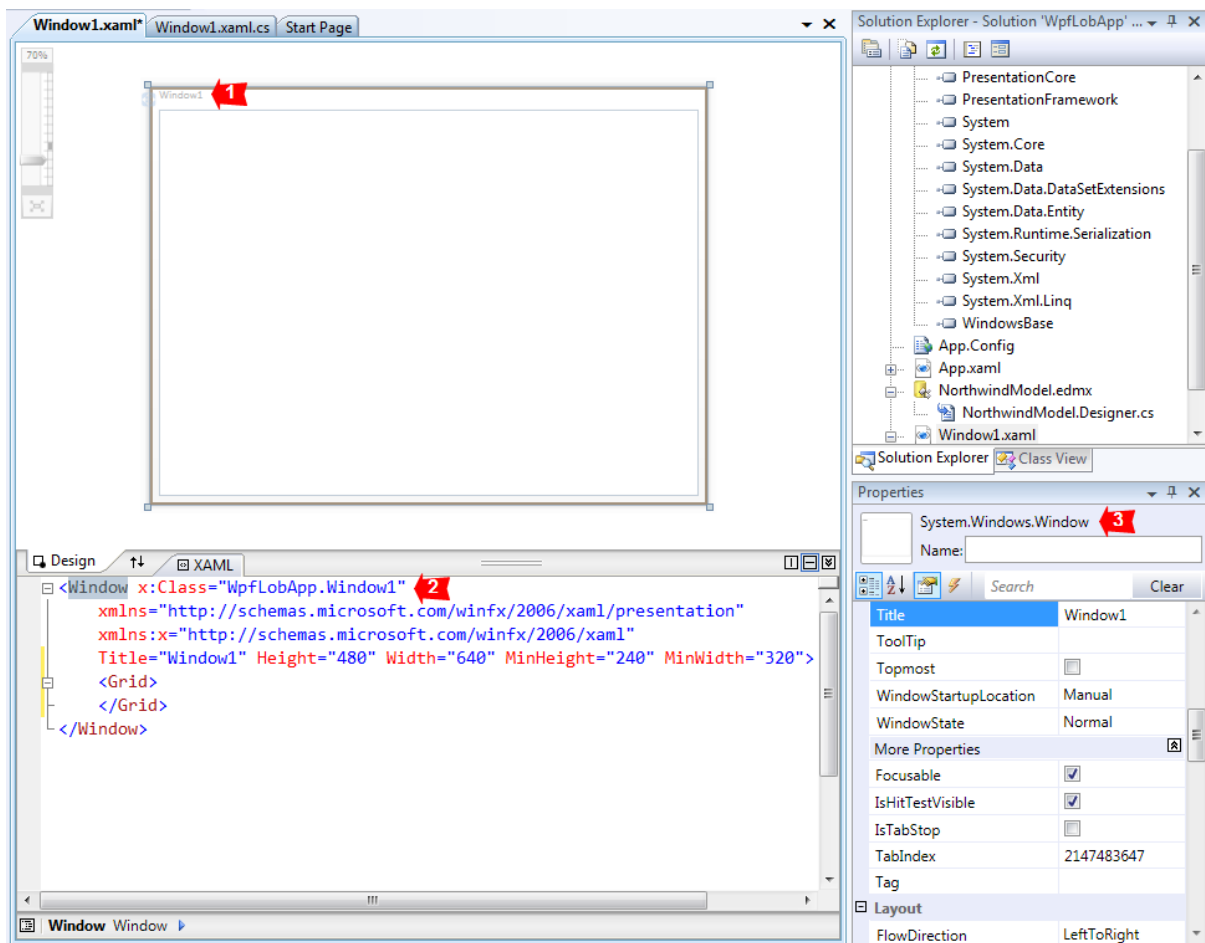


Figure 2.3



Hint: Use the “**Zoom to Fit**” button in the designer’s top left corner if you cannot see the entire window on screen. Alternatively, use the **zoom slider** to enlarge or shrink the preview in the designer.

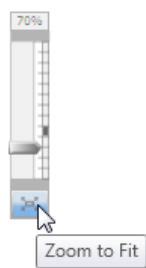


Figure 2.4

- Use the properties pane to set the following properties on **Window1**:

Property	Value
Width	640
Height	480
MinWidth	320
MinHeight	240



Note: You can also edit the values manually in the XAML source. Visual Studio 2008 offers intellisense for editing XAML which makes this a viable option.

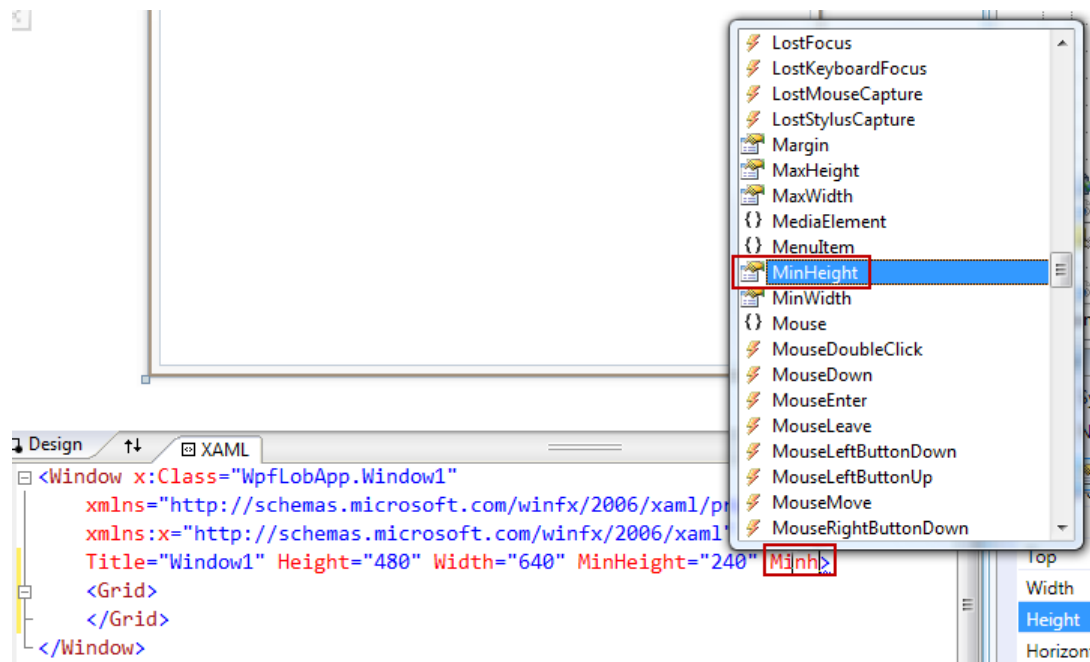


Figure 2.5

- The **MinWidth** and **MinHeight** properties specify the minimal size to which our application can be resized.
- Remove the **<Grid></Grid>** control from **Window1**.



Note: The project template for WPF added the **Grid** control automatically for you but in our sample, we want a **DockPanel** as the top control inside **Window1**.

- Add a **DockPanel** to **Window1** with the following properties:

Property	Value
Name	dockPanelMain
Margin	0
Width	{remove}
Height	{remove}
HorizontalAlignment	Stretch
VerticalAlignment	Stretch



Note: To add the control, select it in Visual Studio's Toolbox and place it on the currently selected control on the design surface. Then set the desired attributes in the properties window. If Visual Studio added unwanted properties like for example width or height in this case, you can just remove them using the properties window or in the XAML source code.



Note: Margin is usually set as four values split by comma (**0,10,20,30**) and set the margin as follows: (**left, top, right, bottom**). If all four values are the same however (0,0,0,0), a single value can be entered as shown above.

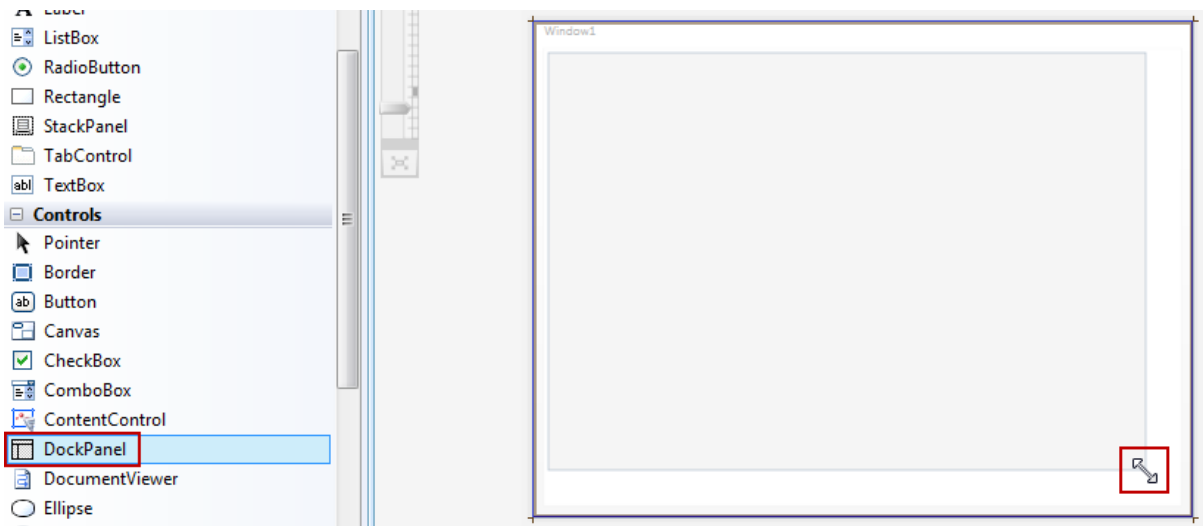


Figure 2.6

- We won't be using the DockPanel until later in the lab when we add the Ribbon control which we will dock to the top of the screen.
- Add a **Grid** control to dockPanelMain with the following properties:

Property	Value
Name	gridMain
Margin	0
Width	{remove}
Height	{remove}
HorizontalAlignment	Stretch
VerticalAlignment	Stretch

Now split the Grid control in **two rows** and **three columns** by clicking in the light-blue border around the Grid control in the designer:

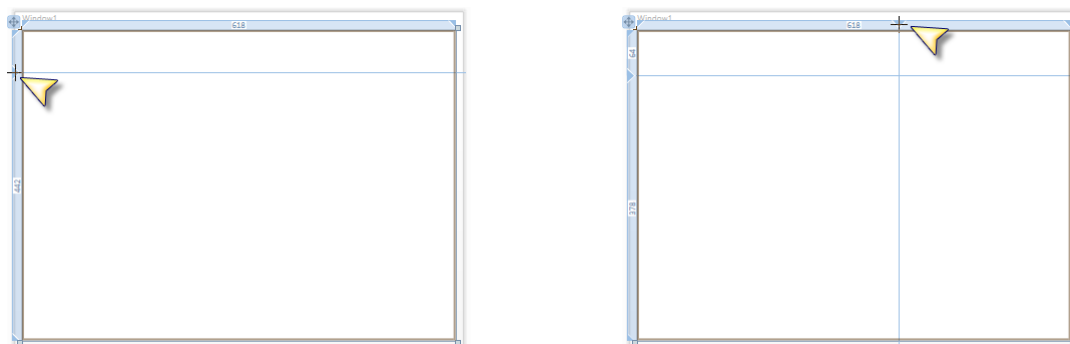


Figure 2.7

- Change the resulting **Grid.ColumnDefinitions** and **Grid.RowDefinitions** as follows:

```
<Grid Name="gridMain" Margin="0">
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="*" />
```



```

<ColumnDefinition Width="8" />
<ColumnDefinition Width="150" />
</Grid.ColumnDefinitions>
<Grid.RowDefinitions>
  <RowDefinition Height="50" />
  <RowDefinition Height="*" />
</Grid.RowDefinitions>
</Grid>

```

Source Code Snippet 2a



Note: By setting one row or column to a fixed value and the other(s) to “*”, you tell WPF to automatically size the row/column to fill the remaining space.

- In the designer, draw a **GridSplitter** control that fills the middle column, second row. Don’t worry about being too precise, we will edit it manually in the next step.
- Set its properties as follows:

Property	Value
Name	gridSplitterV
Margin	0
Width	{remove}
Height	{remove}
HorizontalAlignment	Stretch
VerticalAlignment	Stretch
Grid.Column	1
Grid.Row	1



Note: Note the “**Grid.Column**” and “**Grid.Row**” **Attached Properties**. They are properties inherited by the containing **gridMain** control and are used to set the row and column for all controls that live inside the Grid. This is a very noteworthy concept of XAML.



Note: Note as well that **numbering in XAML starts with 0**. That is why the second column and the second row are numbered “1” above.

First we need a few controls to query the data.

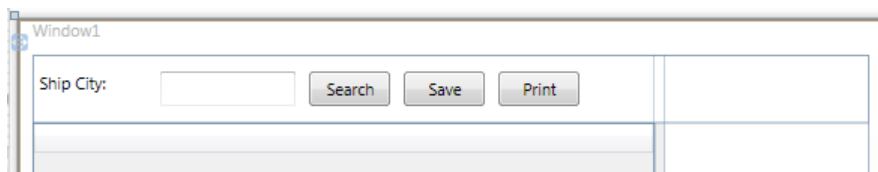


Figure 2.8

We will replace these controls with a ribbon bar in the last exercise of this hands-on lab.

- Draw a **StackPanel** control inside **gridMain**’s the top row across all three columns. Set the properties as follows:

Property	Value
Name	stackPanelTop

Margin	0
Width	{remove}
Height	{remove}
HorizontalAlignment	Stretch
VerticalAlignment	Stretch
Grid.Column	0
Grid.Row	0
Grid.ColumnSpan	3
Orientation	Horizontal



Note: The designer in Visual Studio shows you how a control is placed and anchored in its parents as follows:

1. This is the drag handle to move the control in the designer.
2. Three resize handles in all other corners.
3. The small arrow on each side shows that this side of the control is anchored to the parent control.
4. The long arrow (that complements the small arrow) shows the distance the control will keep to its anchored side.
5. The small dot shows that the control is not anchored on this side.

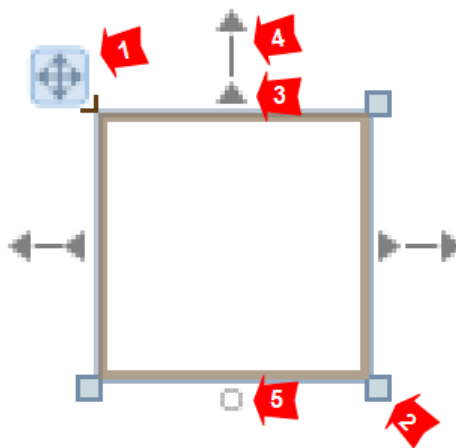


Figure 2.9

- Inside **stackPanelTop** add a **TextBlock**, a **TextBox** and three **Button** controls.
- Set the controls properties as follows:

TextBlock control:

Property	Value
Name	textBlock1
Margin	5
Width	80
Height	25
Text	Ship City:

TextBox control:

Property	Value
----------	-------

Name	textBoxShipCity
Margin	5
Width	100
Height	25

Button control Nr. 1:

Property	Value
Name	buttonSearch
Margin	5
Width	60
Height	25
Content	Search

Button control Nr. 2:

Property	Value
Name	buttonSave
Margin	5
Width	60
Height	25
Content	Save

Button control Nr. 3:

Property	Value
Name	buttonPrint
Margin	5
Width	60
Height	25
Content	Print

The resulting user interface should look like Figure 2.8 above.

- Drag a **DataGrid control** from the Toolbox to the first column, second row of our gridMain. Look at the source code generated.
- Set the properties as follows:

Property	Value
Name	dataGridMain
Margin	0
Width	{remove}
Height	{remove}
HorizontalAlignment	Stretch
VerticalAlignment	Stretch
Grid.Column	0
Grid.Row	1

- You will see that it used a new **namespace** “my:” for the DataGrid and defined it inside the control.

```
<my:DataGrid Grid.Column="0" Grid.Row="1" Margin="0" Name="dataGridMain"
xmlns:my="http://schemas.microsoft.com/wpf/2008/toolkit" />
```

Source Code Snippet 2b

- We don't want the namespace set on the control itself but for the entire window. Move the "**xmlns:my**" property to the **Window1** control and rename it to "**data**".
- Don't forget to also rename the namespace of the DataGrid control.

```
<Window x:Class="WpfLobApp.Window1"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:data="http://schemas.microsoft.com/wpf/2008/toolkit"
Title="Window1" Height="480" Width="640" MinHeight="240" MinWidth="320">
...
    <data:DataGrid Grid.Column="0" Grid.Row="1" Margin="0" Name="dataGridMain" />
...
</Window>
```

• Source Code Snippet 2c



Note: Blend may have already added the namespace to **Window1** automatically, in this case it is enough to simply rename it to "**data**".

- Run your application by pressing F5. The user interface should render and the GridSplitter should let you resize the columns.

Step 3: Wiring the User Interface to the Data Source



Goal: Now that we have a basic, working user interface we want to connect it to the data source that we added to our project in the beginning.



Note: Editing the project's source code or XAML and compiling it may render the designer disabled. To re-enable it click on the yellow bar in the top portion of the designer.

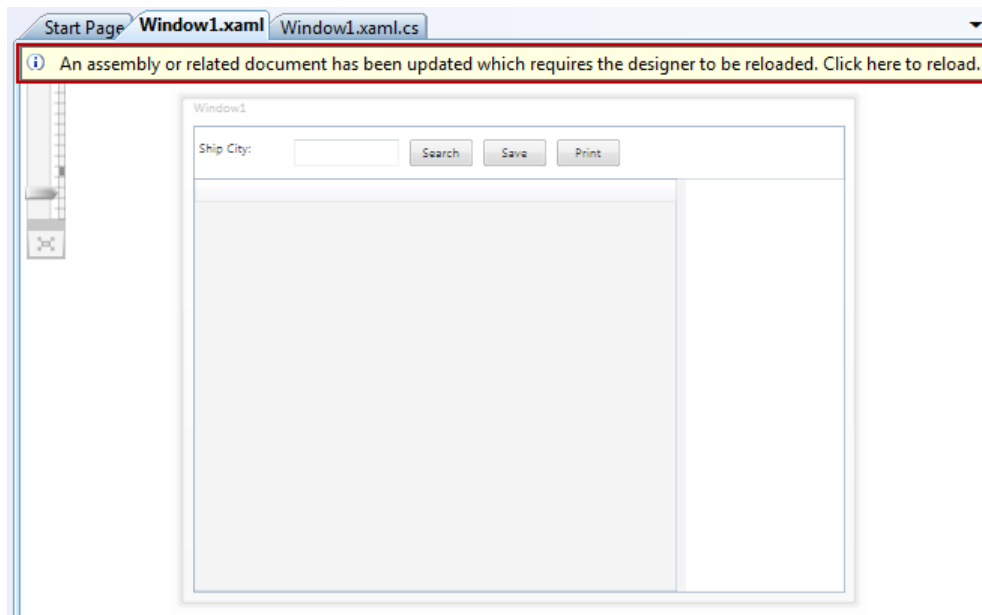


Figure 3.1

Step 3.1: Loading Data from the Database



Goal: First we will wire up the **buttonSearch** button to load data using a LINQ query against the Entity model we crated of the Northwind database.

- In the designer, double-click on **buttonSearch**. Visual Studio will generate an event handler for the “Click” event and switch to source code view of the class behind **Window1** (`Window1.xaml.cs`).
- In code, define a class-wide variable “**db**” of type **northwindEntities**.

```
...
public partial class Window1 : Window
{
    northwndEntities db = new northwndEntities();
}
...
```

Source Code Snippet 3a

- Add the following **LINQ to Entities** query to get all **orders** including linked **employees** where the **ShipCity** field starts with the letters entered in the **textBoxShipCity** control.
- Then set the **DataContext** of our Window1 to the returned list of orders.

```
...

// Event Handler: buttonSearch Click event
private void buttonSearch_Click(object sender, RoutedEventArgs e)
{
    var query = from o in db.Orders.Include("Employees")
                where o.ShipCity.StartsWith(textBoxShipCity.Text)
                orderby o.OrderID ascending
                select o;

    this.DataContext = query.ToList();
}
```

...

Source Code Snippet 3b

- Now all we need to do is tell **dataGridMain** what data to bind to. This can be done by simply setting the **DataContext** and **ItemSource** attributes to **{Binding }**:

...

```
<data:DataGrid Grid.Column="0" Grid.Row="1" Margin="0"
    Name="dataGridMain"
    DataContext="{Binding }"
    ItemSource="{Binding }"/>
```

...

Source Code Snippet 3c

- Run the application by pressing F5. If the application starts, type "a" in **textBoxShipCity** and press **buttonSearch**.

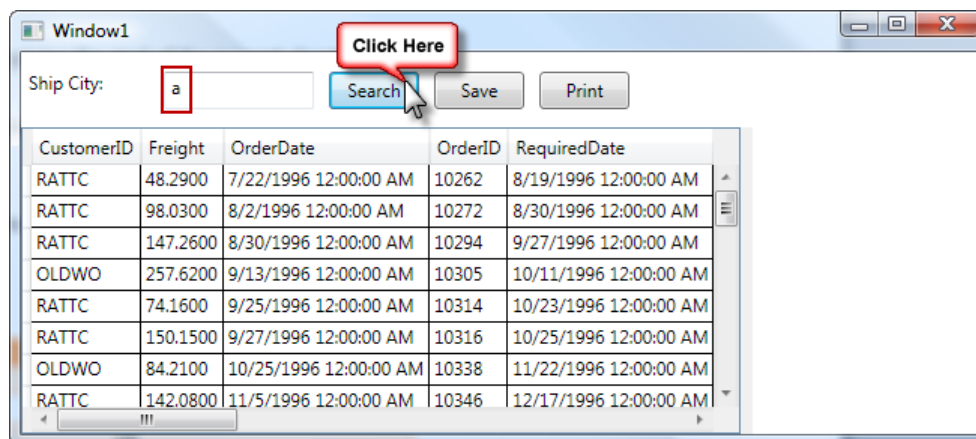


Figure 3.2

Step 3.2: Saving Data back to the Database.



Goal: Next we will wire up the **buttonSave** button. So far we have been able to edit data in **dataGridMain**, but the changes were not saved back to the database. Let's fix this.

- In the designer, double click the **buttonSave** control to get an event handler generated for the **Click** event.
- Simply call the **"SaveChanges"** method on our **northwindEntities** object **"db"**.

...

```
// Event Handler: buttonSave Click event
private void buttonSave_Click(object sender, RoutedEventArgs e)
{
    db.SaveChanges();
}
```

...

Source Code Snippet 3d

- Save the project, run it, modify some data, click the save button.

- Now reload the data and verify that your change was persisted properly.

Step 4: Implementing an Inline Master/Details View.



Goal: We want to show the order items in a new **DataGrid** that is displayed inside **dataGridMain** when one of its row is selected.

We can accomplish this inline by using a **DataGrid.RowDetailsTemplate**. Inside this template we add a **DataTemplate** containing a **TextBlock** and a **DataGrid** to which we will bind the order details data.

- Add the following **RowDetailsTemplate** to the **dataGridMain** control:

```
...
<data:DataGrid Name="dataGridMain" ...
...
    <!-- DataGrid Rows -->
    <data:DataGrid.RowDetailsTemplate>
        <DataTemplate>
            <StackPanel Margin="5,5,5,5" Orientation="Vertical">
                <TextBlock Text="Order details"/>
                <data:DataGrid Margin="5,10,10,10" HorizontalAlignment="Left"
                    x:Name="dataGridDetails" >
                </data:DataGrid>
            </StackPanel>
        </DataTemplate>
    </data:DataGrid.RowDetailsTemplate>
</data:DataGrid>
...
```

Source Code Snippet 4a

- Next create an event handler for the **dataGridMain**'s "**LoadingRowDetails**" event. In XAML, add the definition:

```
...
<data:DataGrid Grid.Column="0" Grid.Row="1" Margin="0"
    Name="dataGridMain"
    DataContext="{Binding }"
    ItemsSource="{Binding }"
    LoadingRowDetails="dataGridMain_LoadingRowDetails"
>
...
```

Source Code Snippet 4b



Note: Remember that you can either create the event handler in XAML code as depicted above or use the **designer** in **Visual Studio** to generate the event handler. To do this select the control (**DataGridMain**), click the "**Events**"-Button in the **Properties** pane and then **double-click** the desired **event name** to automatically have an event handler generated for you.

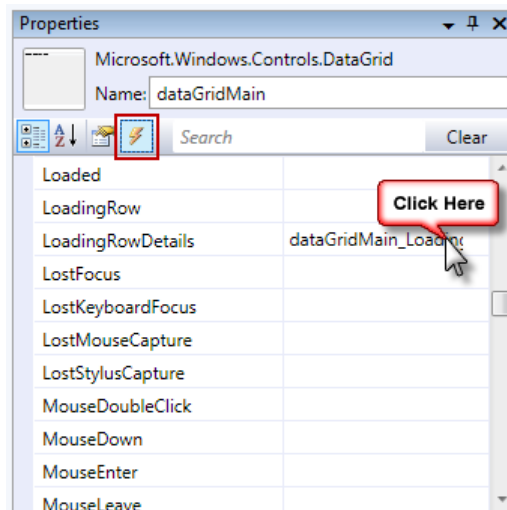


Figure 4.1

- In the code behind, add a reference to the **Microsoft.Windows.Controls** where the DataGrid control lives and implement the event handler for the **LoadingRowDetails** event. The handler's task is to get the **OrderID** from the selected order (row), find the **dataGridDetails** and populate it with the order details that match the **OrderID**.

```
...
using Microsoft.Windows.Controls;
...
// Event Handler: dataGrid LoadingRowDetails event
private void dataGridMain>LoadingRowDetails(object sender,
    Microsoft.Windows.Controls.DataGridRowDetailsEventArgs e)
{
    int orderid = ((Orders)e.DetailsElement.DataContext).OrderID;
    var dataGridDetails = (DataGrid)e.DetailsElement.FindName("dataGridDetails");
    dataGridDetails.ItemsSource = db.Order_Details.Where
        (od => od.OrderID == orderid).ToList();
}
...
```

Source Code Snippet 4c

- Run the application, search for orders and then click on a row of data in the **dataGridMain**. This should expand to show **dataGridDetails** with the order details. Click on a few **rows** and see if the displayed order details change based on the selected order.

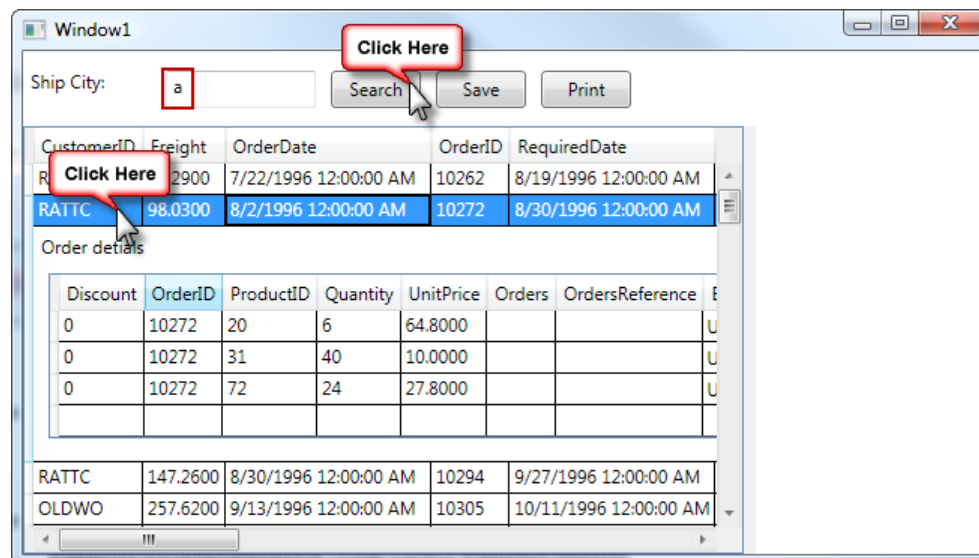


Figure 4.2

Step 5: Changing the Columns shown in the DataGrid.



Goal: We want to change the columns that are shown in **dataGridMain** and adapt the basic looks of our **dataGridMain**.

We don't want to show the last two columns **EntityState** and **EntityKey** to our users, but the default behavior of the DataGrid is to show all columns.



Note: If in your own project you don't want a DataGrid to automatically generate columns, you can override the default behavior by setting the DataGrid's **"AutoGenerateColumns"** property to false. We won't set this in our example however.

- Add an event handler to **dataGridMain**'s **AutoGeneratingColumn** event.

```
...
<data:DataGrid Grid.Column="0" Grid.Row="1" Margin="0"
    Name="dataGridMain"
    DataContext="{Binding }"
    ItemsSource="{Binding }"
    AutoGeneratingColumn="dataGridMain_AutoGeneratingColumn"
    LoadingRowDetails="dataGridMain>LoadingRowDetails"
>
...
```

Source Code Snippet 5a

- Implement the event handler in the code behind. It simply cancels the generation of the columns if their names match **"EntityState"** or **"EntityKey"**:

```
...
private void dataGridMain_AutoGeneratingColumn(object sender,
    Microsoft.Windows.Controls.DataGridAutoGeneratingColumnEventArgs e)
{
    ...
}
```

```

    if (e.PropertyName == "EntityState" || e.PropertyName == "EntityKey")
    {
        e.Cancel = true;
    }
}
...

```

Source Code Snippet 5b

- Now let's freeze the first two columns so they never scroll off screen and introduce alternating background colors for the rows in **dataGridMain**. This can be done very easily using the properties of the **DataGrid** control's **FrozenColumnCount** and **AlternatingRowBackground** properties.

```

...
<data:DataGrid Grid.Column="0" Grid.Row="1" Margin="0"
    Name="dataGridMain"
    DataContext="{Binding }"
    ItemsSource="{Binding }"
    FrozenColumnCount="2"
    AlternatingRowBackground="Azure"
    AutoGeneratingColumn="dataGridMain_AutoGeneratingColumn"
    LoadingRowDetails="dataGridMain_LoadingRowDetails"
>
...

```

Source Code Snippet 5c



Note: Instead of manually adding these properties, all properties can be set in the Properties pane in Visual Studio.

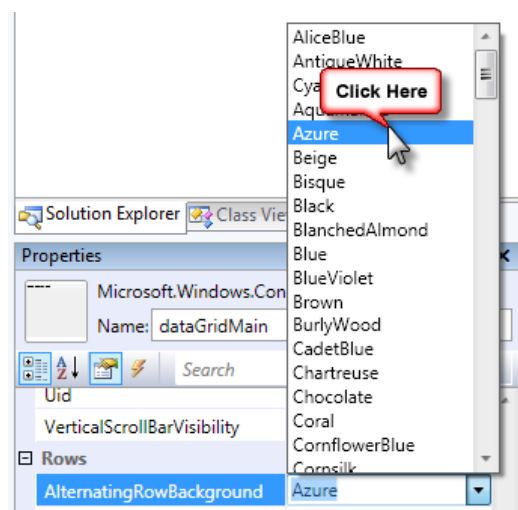


Figure 5.1

- Run the application and verify the changes work, including the two last columns **"EntityState"** or **"EntityKey"** being removed.

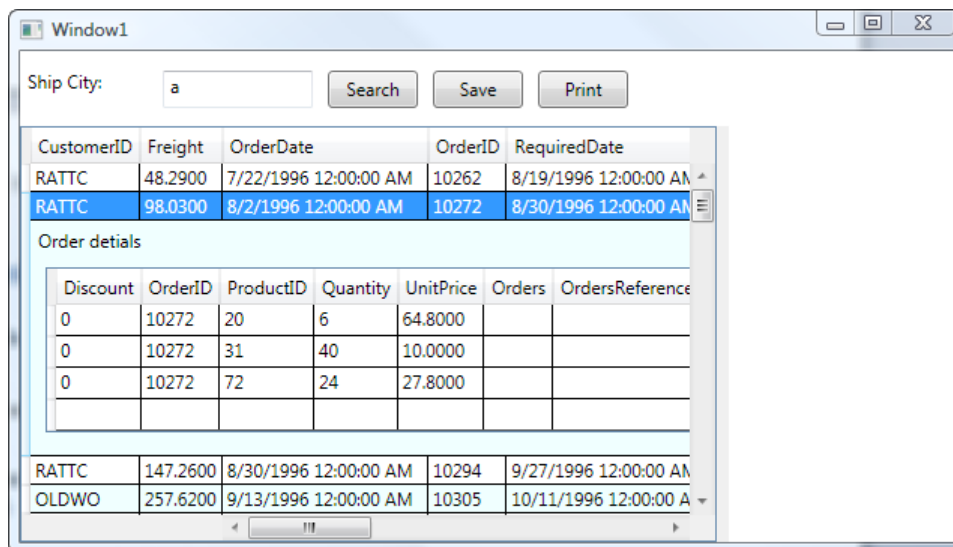


Figure 5.2

Step 6: Adding custom Controls to the DataGrid.

Step 6.1: Showing the Employee in a ComboBox.



Goal: In **dataGridMain** we want to display a column showing a **ComboBox** containing the **Employee** associated with the order and pre-populated with all Employees from the linked table.

First let's create the ComboBox containing the names of employees associated with the order.

- In **Window1.xaml.cs** in the **WpfLobApp** namespace create a second class called **EmployeesDataSource** of type **List<Employees>**.

```
...
public class EmployeesDataSource : List<Employees> { }
...
```

Source Code Snippet 6a

- Switch to **Window1.xaml** and add the **xmlns:local** to the Window1 control, pointing to the **WpfLobApp** namespace. We use this to point to ourselves, namely to the **EmployeesDataSource** class from within XAML.

- Now add a **Window.Resources** element as the first child of the **Window** element, creating an instance of the **EmployeesDataSource** class called **EmployeesDS**.

```
<Window x:Class="WpfLobApp.Window1"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local="clr-namespace:WpfLobApp"
        xmlns:data="http://schemas.microsoft.com/wpf/2008/toolkit"
        Title="Window1" Height="480" Width="640" MinHeight="240" MinWidth="320">

    <!-- Resources -->
    <Window.Resources>
        <local:EmployeesDataSource x:Key="EmployeesDS" />
    </Window.Resources>

    ...
```

Source Code Snippet 6b

- Switch back to **Window1.xaml.cs**.
- In the constructor of **Window1**, add the following initialization code for **EmployeesDS**

```
...
public Window1()
{
    InitializeComponent();
    EmployeesDataSource eds =
        (EmployeesDataSource)this.FindResource("EmployeesDS");
    eds.AddRange(db.Employees.ToList());
}
...
```

Source Code Snippet 6c

- Inside the **dataGridMain** control, add the **DataGrid.Columns** element and define the **DataGridTemplateColumn** for **Employee**. This consists of a **CellTemplate** that contains a **DataTemplate** in which we add the **ComboBox** that we described above. The **ComboBox**'s **ItemSource** can now be set to the **EmployeesDS** and the **SelectedItem** should display the "LastName" from "Employees". With the property "TwoWay" we specify that the data should not just be read but also written back to the database if changed.

```
...
<!-- DataGrid Columns -->
<data:DataGrid.Columns>

    <data:DataGridTemplateColumn Header="Employee">
        <data:DataGridTemplateColumn.CellTemplate>
            <DataTemplate>
                <ComboBox
                    ItemsSource="{Binding Source={StaticResource EmployeesDS}}"
                    SelectedItem="{Binding Employees, Mode=TwoWay}"
                    SelectedValuePath="Employees"
                    DisplayMemberPath="LastName" />
            </DataTemplate>
        </data:DataGridTemplateColumn.CellTemplate>
    </data:DataGridTemplateColumn>

</data:DataGrid.Columns>

...
```

Source Code Snippet 6d

- It's time to test our ComboBox. Run the application. It should look as follows:

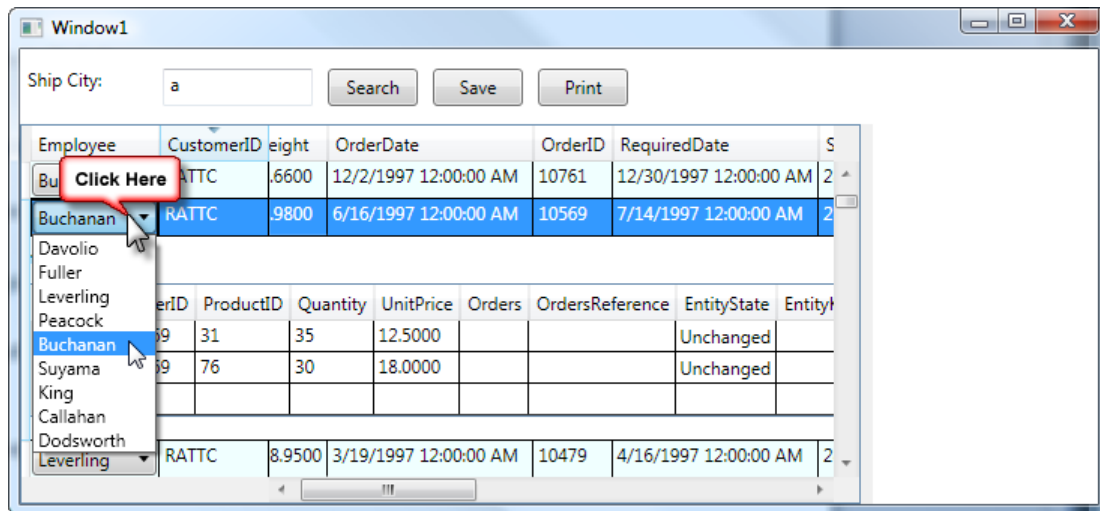


Figure 6.1

Step 6.2: Showing a DatePicker when editing OrderDate.



Goal: In **dataGridMain** we want to display a **DatePicker** control when the user edits the **OrderDate** column. For this, we will make a difference between the control's **normal state** in which we will show a **TextBlock** and the **editing state** in which we will swap in the **DatePicker**.

- In XAML, introduce a new “**DataGridTemplateColumn**” for **ShippedDate**. It contains an element for **CellTemplate** containing a **TextBlock** and one for **CellEditingTemplate** containing the **DatePicker** control, each bound to **ShippedDate**.

```
...
<!-- DataGrid Columns -->
<data:DataGrid.Columns>
...
    <data:DataGridTemplateColumn Header="ShippedDate">
        <data:DataGridTemplateColumn.CellTemplate>
            <DataTemplate>
                <TextBlock Text="{Binding Path=ShippedDate}"/>
            </DataTemplate>
        </data:DataGridTemplateColumn.CellTemplate>
        <data:DataGridTemplateColumn.CellEditingTemplate>
            <DataTemplate>
                <data:DatePicker
                    SelectedDate="{Binding Mode=TwoWay, Path=ShippedDate}" />
            </DataTemplate>
        </data:DataGridTemplateColumn.CellEditingTemplate>
    </data:DataGridTemplateColumn>
</data:DataGrid.Columns>
...
```

Source Code Snippet 6e

- Run the application and try editing a record's **ShippedDate**. The control should switch to a **DatePicker**:

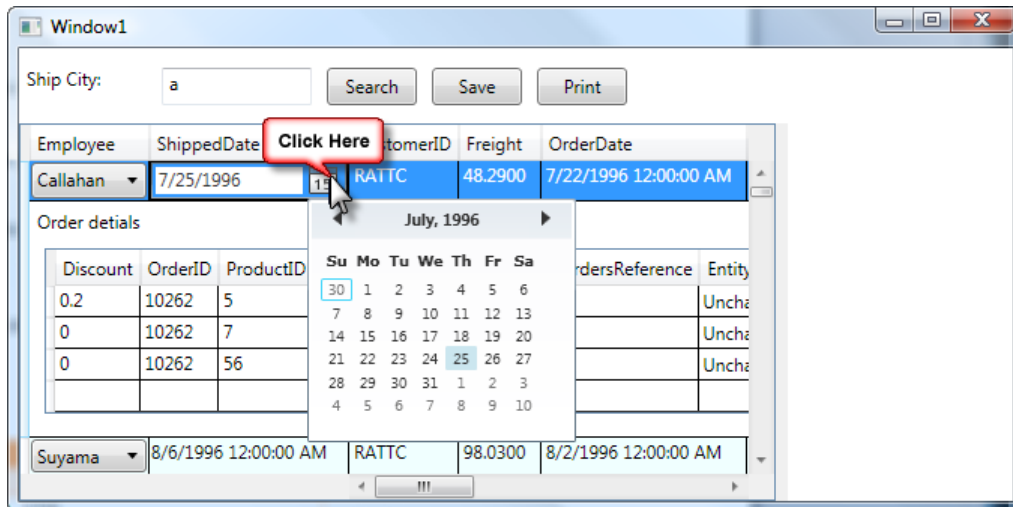


Figure 6.2

Step 6.3: Color-Coding the Freight Column.



Goal: In **dataGridMain** we want to display the **Freight** field with a red background if its value is greater than 50 and with a green background if not.

- We first need a helper class that we call "**BackgroundColorConverter**". This class implements the interface **IValueConverter** and will do the decision in which color the cell's background should be painted.
- Add a new **Class** to the project and call it "**BackgroundColorConverter.cs**".

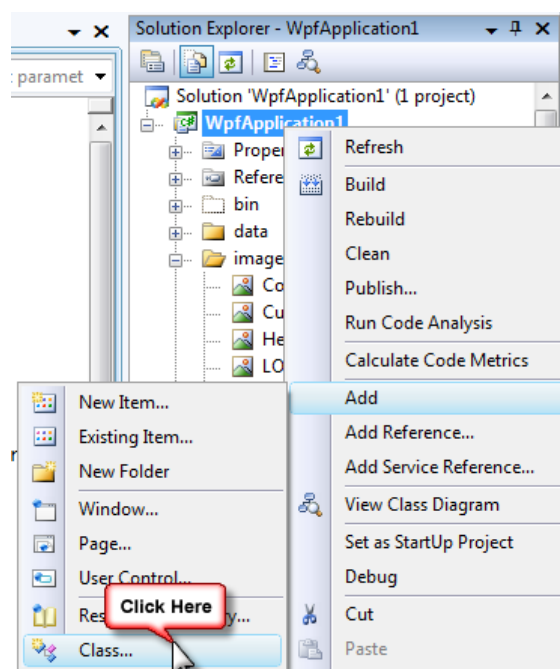


Figure 6.3

- Implement the methods **Convert** and **ConvertBack** as follows. Don't forget to set the **IValueConverter** interface:

```
...
public class BackgroundColorConverter : IValueConverter
{
    public object Convert(object value, Type targetType, object parameter,
        CultureInfo culture)
    {
        if (value != null)
        {
            if ((decimal)value > 50)
                return new SolidColorBrush(Colors.Red);
            else
                return new SolidColorBrush(Colors.Green);
        }

        return new SolidColorBrush(Colors.White);
    }

    public object ConvertBack(object value, Type targetType, object parameter,
        CultureInfo culture)
    {
        return System.Convert.ToDecimal(value);
    }
}
...
```

Source Code Snippet 6f

- Add the missing using statements for the **System.Windows.Media**, **System.Windows.Data** and **System.Globalization** namespaces.



Note: Our class is missing a few references. The classes that are not recognized are displayed underlined in red. You can use the **tooltip** on these classes to let Visual Studio 2008 help you add the missing “using” statements.

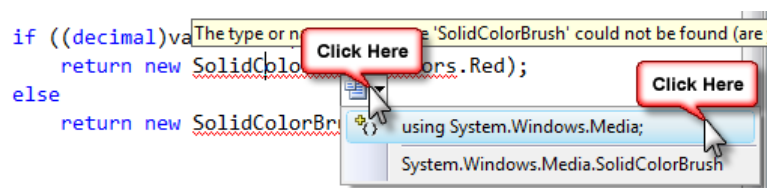


Figure 6.4

- Now add the **BackgroundColorConverter** as a resource to **Window1**.

```
...
<Window x:Class="WpfLobApp.Window1"
    ...
>

<!-- Resources -->
<Window.Resources>
    <local:EmployeesDataSource x:Key="EmployeesDS" />
    <local:BackgroundColorConverter x:Key="BackgroundColorConverter" />
</Window.Resources>
```


...

Source Code Snippet 6g

- Add the **DataGridTemplateColumn** for “**Freight**”. It will again contain a **CellTemplate** and a **CellEditingTemplate** like the previous example.
- The **CellTemplate** will contain a **Border** containing a **TextBlock**. The Border’s background is bound to the data field “**Freight**” and uses the “**Converter**” property set to **BackgroundColorConverter**. As we remember, BackgroundColorConverter returns a **SolidColorBrush** that will set the background color.
- In the **CellEditingTemplate** we specify a **TextBox** with a **FontSize** set to **30**, so when editing the Freight field, the user will be presented with a large font for easy editing.

...

```

<!-- DataGrid Columns -->
<data:DataGrid.Columns>

...

    <data:DataGridTemplateColumn Header="Freight">

        <data:DataGridTemplateColumn.CellTemplate>
            <DataTemplate>
                <Border Background="{Binding Freight, Converter={StaticResource
                    BackgroundColorConverter}}">
                    <TextBlock Text="{Binding Freight}" />
                </Border>
            </DataTemplate>
        </data:DataGridTemplateColumn.CellTemplate>

        <data:DataGridTemplateColumn.CellEditingTemplate>
            <DataTemplate>
                <TextBox FontSize="30" Text="{Binding Freight}" />
            </DataTemplate>
        </data:DataGridTemplateColumn.CellEditingTemplate>

    </data:DataGridTemplateColumn>

</data:DataGrid.Columns>

```

...

Source Code Snippet 6h

- Run the application and inspect the freight column’s background color, also trying to edit the value contained.

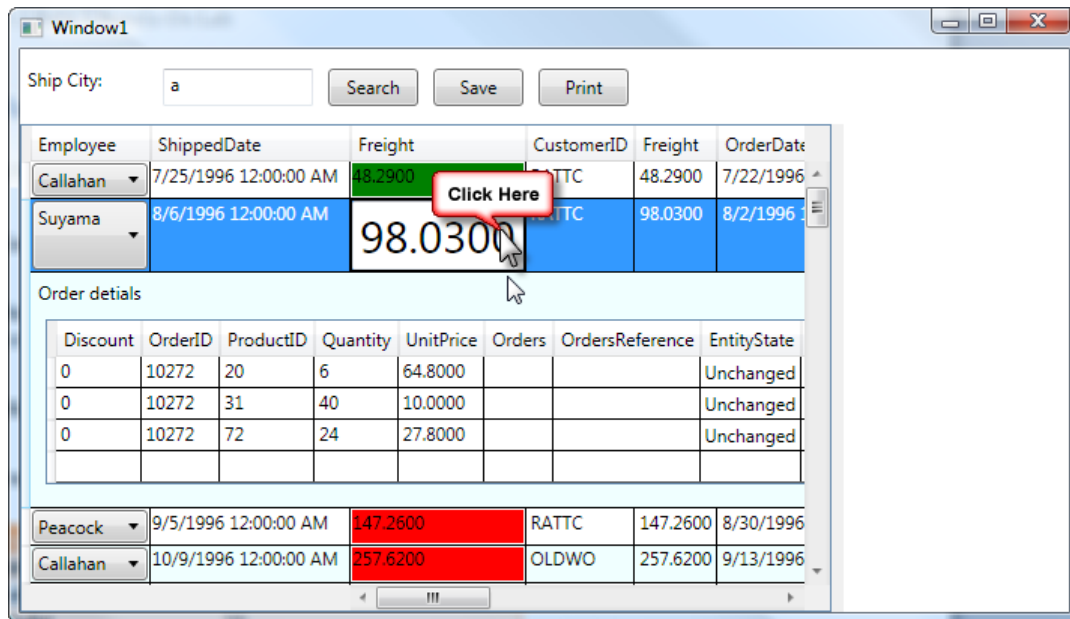


Figure 6.5

Step 7: Adding a Details Pane containing Customer Data.



Goal: We want to add a pane to the right of the **dataGridMain** where Customer data is displayed. The contents should auto-resize when **gridSplitterV** is moved.

- Add a **Viewbox** to **gridMain**'s Column 2, Row 1 containing a **StackPanel** that contains four **TextBlock** controls.



Note: When using the designer to add the controls, make sure that the **Viewbox** encloses the **StackPanel** and that it encloses the **TextBlocks**. You may easily correct this in the XAML source if the designer got it wrong.

Viewbox control:

Property	Value
Name	viewboxDetail
Grid.Column	2
Grid.Row	1
VerticalAlignment	Top

StackPanel control:

Property	Value
Name	stackPanelDetail
Margin	20,15,10,0

TextBlock control Nr. 1:

Property	Value
Text	Customers Details

TextBlock control Nr. 2:

Property	Value
Height	20
Margin	0,5,0,0

TextBlock control Nr. 3:

Property	Value
Height	20
Margin	0,5,0,0

TextBlock control Nr. 4:

Property	Value
Height	20
Margin	0,5,0,0

- Now data bind the TextBlocks 2 through 4 to the Employee's **FirstName**, **LastName** and **Region** fields.

Try it before turning to the solution on the next page. The result should look like this:

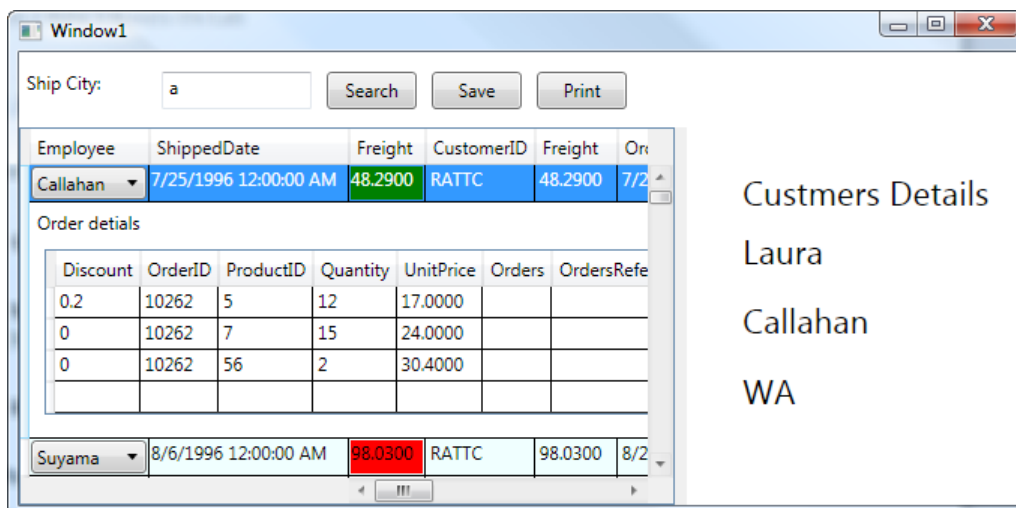


Figure 7.1

```

...
<Viewbox Grid.Column="2" Grid.Row="1" VerticalAlignment="Top" Name="viewboxDetail" >
  <StackPanel Margin="20,15,10,0" Name="stackPanelDetail" >
    <TextBlock>Customers Details</TextBlock>
    <TextBlock Height="20" Margin="0,5,0,0"
      DataContext="{Binding }" Text="{Binding Employees.FirstName}" />
    <TextBlock Height="20" Margin="0,5,0,0"
      DataContext="{Binding }" Text="{Binding Employees.LastName}" />
    <TextBlock Height="20" Margin="0,5,0,0"
      DataContext="{Binding }" Text="{Binding Employees.Region}" />
  </StackPanel>
</Viewbox>
</Grid>
...

```

Source Code Snippet 7a

- Run the application and verify the detail pane shows.

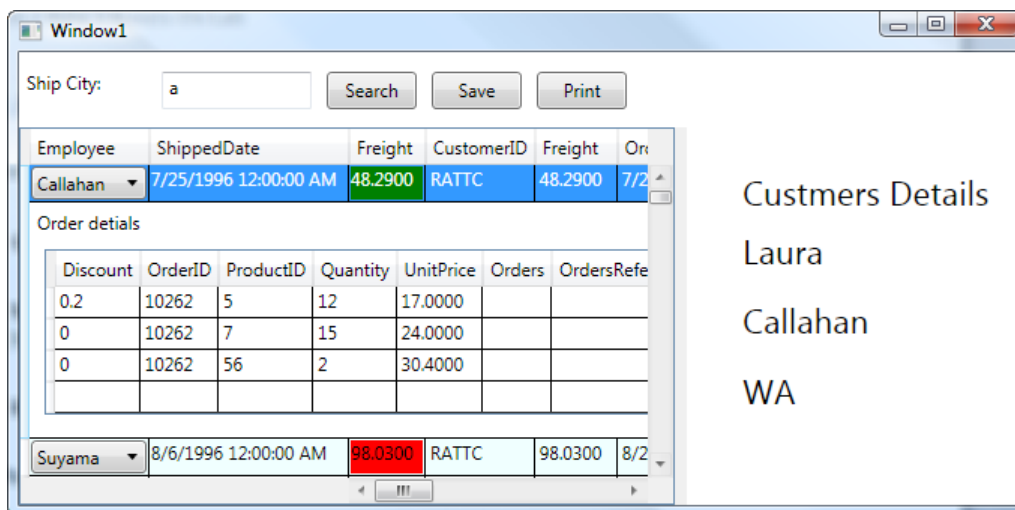


Figure 7.2



Note: Even with the first customer's details showing, the details pane doesn't update when changing records.

- To fix this, set the **IsSynchronizedWithCurrentItem** property of **dataGridMain** to **true**.

```

...
<data:DataGrid Grid.Column="0" Grid.Row="1" Margin="0"
  Name="dataGridMain"
  DataContext="{Binding }"
  ItemsSource="{Binding }"
  FrozenColumnCount="2"
  AlternatingRowBackground="Azure"
  IsSynchronizedWithCurrentItem="True"
  AutoGeneratingColumn="dataGridMain_AutoGeneratingColumn"
  LoadingRowDetails="dataGridMain_LoadingRowDetails"
>
...

```

Source Code Snippet 7b

- Run the application again and select a few records. Make sure the details pane updates when switching records.

Step 8: Creating a Custom Control to visualize Data.



Goal: The “ShipVia” field contains an integer from 1 to 3. We want to prominently display this value using a traffic light that shows a red light for the value 1, yellow for 2 and green for 3 – including an animation when switching lights. We will use Expression Blend to design this control.



Figure 8.1

- Make sure the solution is saved in Visual Studio.
- Start **Expression Blend 2** and open the solution (C:\WPF_HOL\Project\WpfLobApp\WpfLobApp.sln).
- Close “**Window1.xaml**” if Blend opened it and switch to the **Project** pane (1).
- Select “**New Item...**” from the **File** menu and in the dialog select **UserControl**, name it “**TrafficLightUserControl.xaml**” and make sure that “**Include code file**” is enabled.

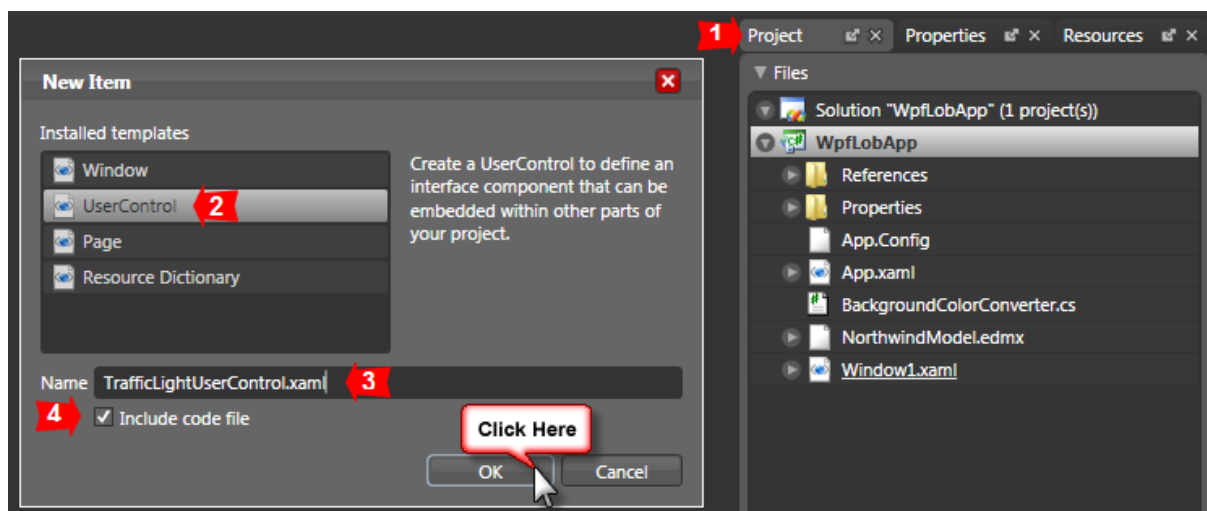


Figure 8.2



Hint: You can use **F6** to toggle between the Design- and Animation-Workspaces, the two operation modes of Expression Blend. We start in the Design Workspace for our example.

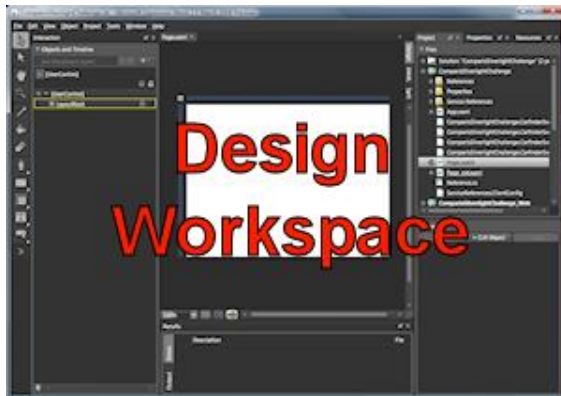


Figure 8.3

- Select the **Split** view (1) to show the source-code pane (2) and the design view (3).
- Click inside the design-view and select "**Fit to screen**" from the **View**-pulldown-menu.



Hint: You can do this whenever you want to automatically resize the design view to show all content by hitting CTRL+0.

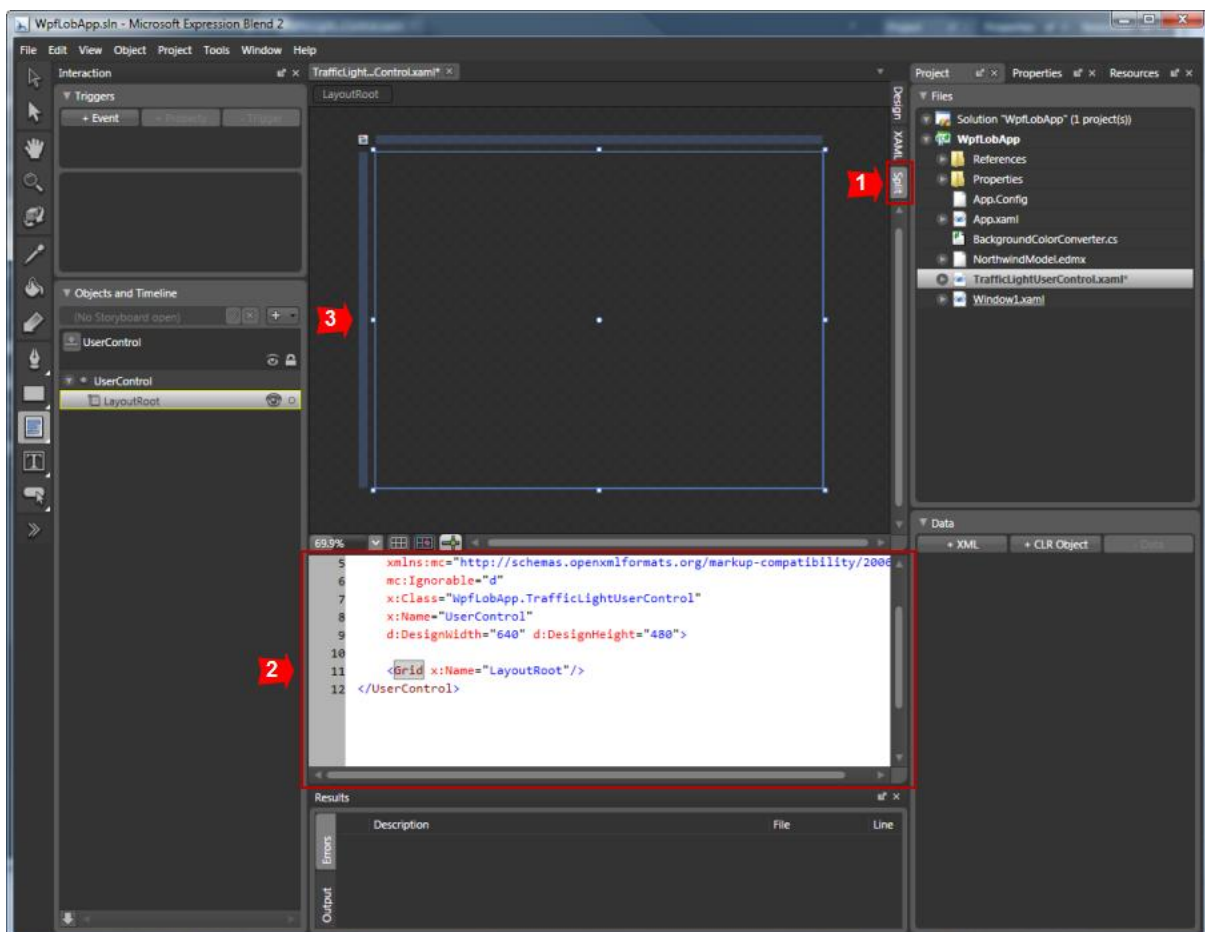


Figure 8.4

- On the UserControl, set the **d:DesignWidth** to **150** and **d:DesignHeight** to **500**. This is only for the designer.
- Double click the **LayoutRoot** in the “**Objects and Timeline**” pane to make it active.

Note: Remember, that Expression Blend distinguishes between:



- **Selected control ②:** The control that we are manipulating and that are represented in the properties pane. Selected controls appear with a light grey background.
- **Active control ③:** The control that is the parent for the subsequent controls which will be added as child nodes. Active controls appear with a yellow border.

- Use the selection tool ① and single-click on a control in the “**Objects and Timeline**” window to select it, double-click to activate it.

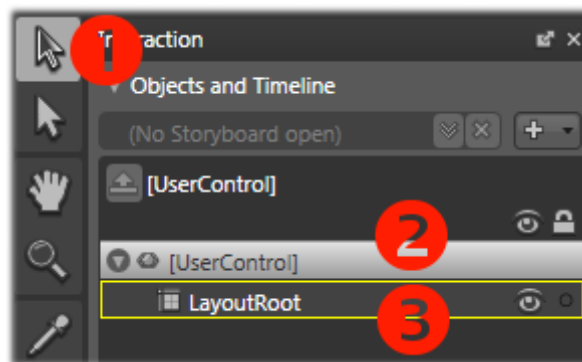


Figure 8.5

- Add a **StackPanel** control to **LayoutRoot**. To select the StackPanel control, click and hold the encapsulation controls button in the Expression Blend toolbox and select **StackPanel** from the choice of controls.

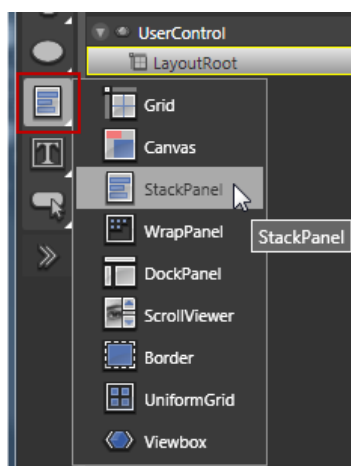


Figure 8.6

- Double click the **StackPanel** to automatically add it to the **active control**.
- Switch from the **Project** to the **Properties** pane.

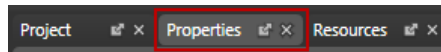


Figure 8.7

- Locate the “**Layout**” section in the properties pane. You will see that Blend automatically set a few properties of this control for us. You can spot them by the **little, white dots** to the right of the property boxes. We want to reset them to the default values. To do this, click on the white boxes and select “**Reset**”.

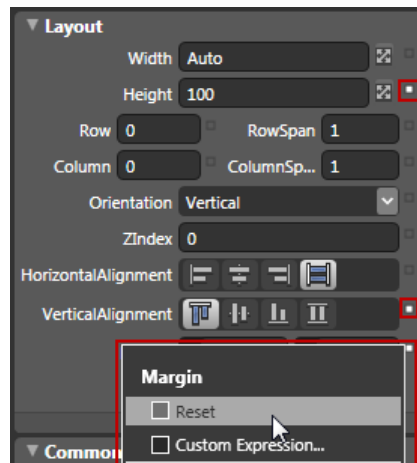



Figure 8.8



Note: Also remember that you can click the  -Button to set the width or height to “Auto”.

- Set the StackPanel’s name to “**stackPanelMain**”

Property	Value
Name	stackPanelMain
Height	{reset to Auto}
Width	{reset to Auto}

- stackPanelMain** should now have all of its properties removed and fill the enclosing **LayoutRoot**. Make **stackPanelMain** active (double click it in the “**Objects and Timeline**” pane) and add a **Border** inside it with the following properties:

Property	Value
Name	borderMain
Margin	11,11,11,0
Width	100
CornerRadius	20,20,20,20
Background	#FF000000



Note: You can always set the properties in the XAML source or use the properties pane. As some of the controls have quite a few properties to set, there is a very useful **Search** box in the top section of this pane.

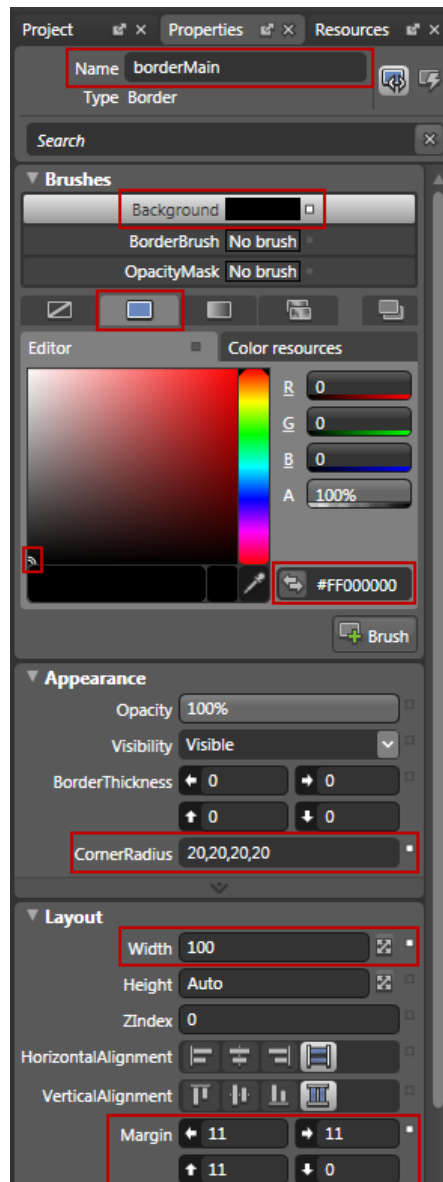


Figure 8.9

- Inside **borderMain** add a **StackPanel** with all properties reset to default.
- Inside the StackPanel add three **Ellipse** controls as follows:

Ellipse Nr. 1:

Property	Value
Name	ellipseRed
Margin	0,10,0,0
Width	70
Height	70
Opacity	0.3 (30% in the properties panel)
Fill	#FFFF0000

Ellipse Nr. 2:

Property	Value
Name	ellipseOrange
Margin	0,10,0,0
Width	70
Height	70
Opacity	0.3 (30% in the properties panel)
Fill	#FFFA500

Ellipse Nr. 3:

Property	Value
Name	ellipseGreen
Margin	0,10,0,10
Width	70
Height	70
Opacity	0.3 (30% in the properties panel)
Fill	#FF008000



Note: If you don't like the hex values to define the colors (#FF008000), you can also use the source code view and replace the hex values with color names, for example "Red", "Orange" and "Green".

You should now see the traffic light in the designer with all lights "off" (their opacity set to 0.3). Now let's add the reflection.

- Make **stackPanelMain** active and add a new **Border** control with these properties:

Property	Value
Name	borderReflection
Margin	11,0,11,11
Height	150
CornerRadius	20,20,20,20

- Select the **borderMain** control and from the **Tools** menu select "Make Brush Resource" → "Make Visual Brush Resource..."
- Name the new VisualBrush "**visualBrush_borderMain**" and hit **OK**.

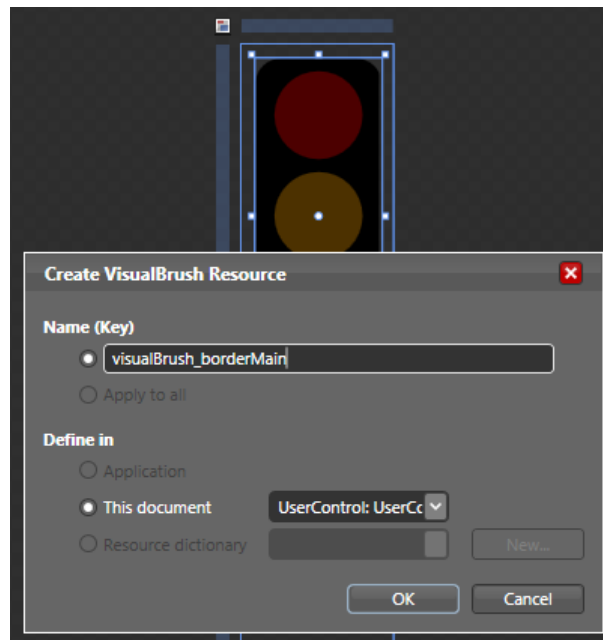


Figure 8.10

- Select **borderReflection** and in the properties pane, select the newly created **visualBrush_borderMain** as the control's background.

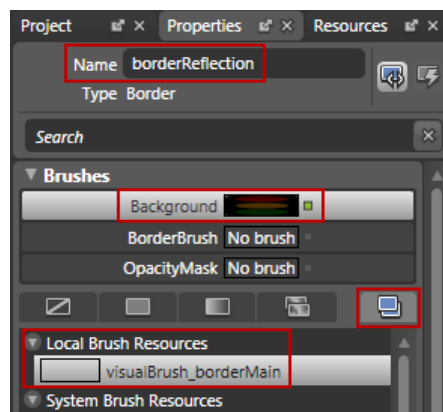


Figure 8.11

- In **borderReflection**'s **RenderTransform** properties, select "**Flip Y axis**" to mirror the control.

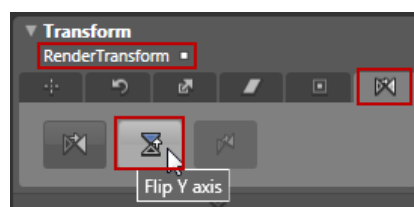


Figure 8.12

- Set **borderReflection**'s **OpacityMask** to a **Gradient Brush** with an **alpha** value (transparency) from **0%** to **50%**.

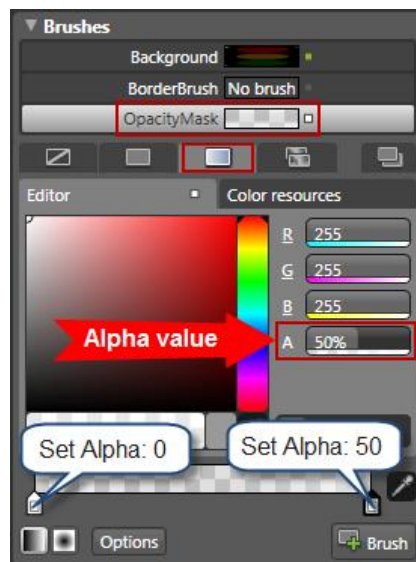
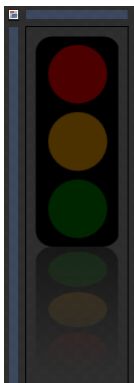


Figure 8.13



Your control should look like this in the designer.
Don't forget to save the project!

We will now add two animations, one to “turn on” the red light and one to “turn it back off”. We will then use code to apply these two animations to all the lights at run-time.

- In the **Objects and Timeline** pane click the “Create Storyboard Resource” button (+) and name it **lightAnimation**.

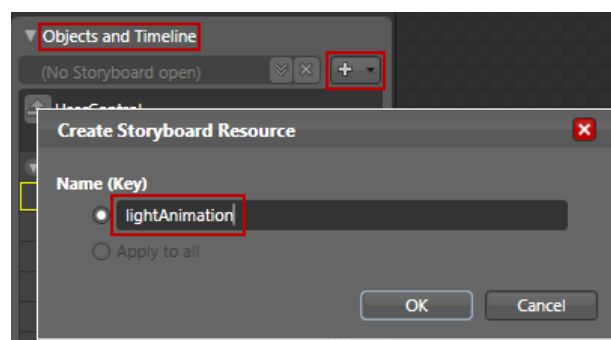


Figure 8.14



Note: Blend switches to the timeline view. If you don't have enough room for the larger Objects and Timeline window, you can hit F6 to switch to the animation view.

- Select **ellipseRed** control (1), move the **time slider** to **0.5 seconds** (2) and set the **Fill Brush's** (3) **opacity** value to **100%** (4). This will create a **Key Frame** (5) in the timeline.

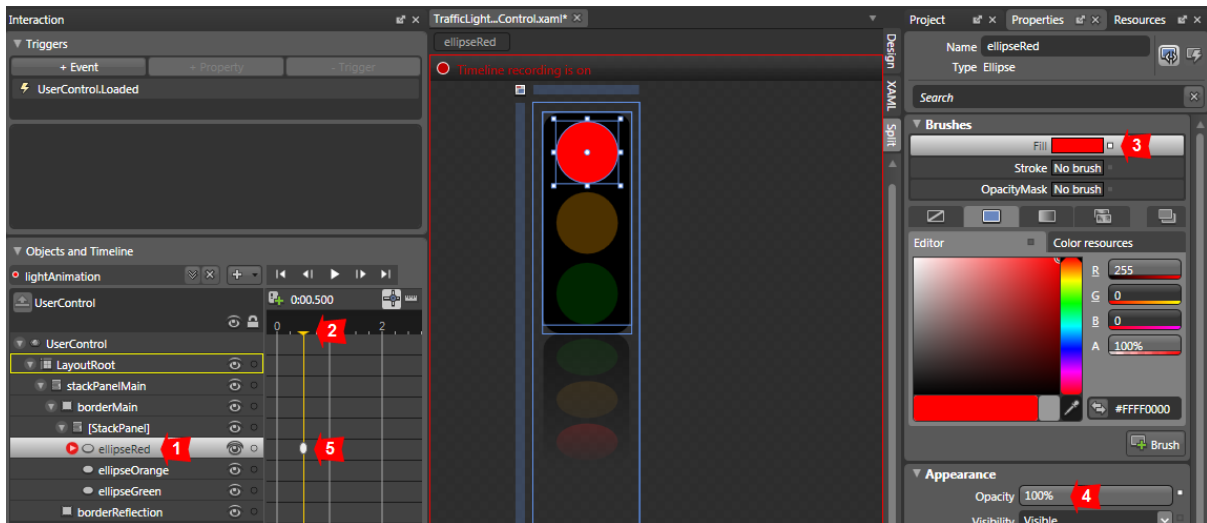


Figure 8.15

- Now create **two more key frames**, one at **1.0 seconds** setting the **opacity to 30%** and one at **1.5 seconds** setting the **opacity to 100%** again. This will create a flashing effect.



Note: You can test your storyboards by clicking the **Play** button above the timeline.

- Add another Storyboard Resource named **"lightAnimationReset"** and in this one select **ellipseRed** control again, move the **time slider** to **0.5 seconds** and set the **Fill Brush's opacity** value to **30%**. We will use this animation to turn off the active traffic light smoothly.



Hint: You may not be able to create the key frame with opacity set to 30% as the starting opacity is already set to 30%. To get around this, set the value to 35% and then in the XAML source, locate the **Storyboard** element and replace the value of the **SplineDoubleKeyFrame** at 0.5 seconds with 0.3.

```

...
<Storyboard x:Key="lightAnimationReset">
  <DoubleAnimationUsingKeyFrames BeginTime="00:00:00"
    Storyboard.TargetName="ellipseRed"
    Storyboard.TargetProperty="(UIElement.Opacity)">
    <SplineDoubleKeyFrame KeyTime="00:00:00.5000000" Value="0.3"/>
  </DoubleAnimationUsingKeyFrames>
</Storyboard>
...

```

Source Code Snippet 8a

- Close the storyboard in the "Objects and Timeline" pane by clicking the "X"-button

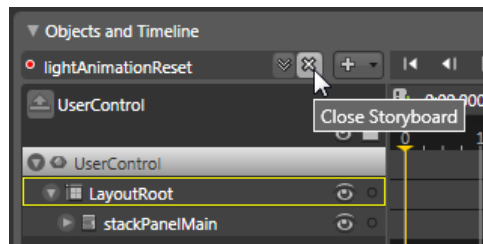


Figure 8.16

There is one last thing to do in Blend. It created triggers that play the storyboards when the user control has loaded. We don't want this behavior, so we remove these triggers.

- In the Triggers pane, select the “**UserControl.Loaded**” trigger and remove it by clicking on “- Trigger”.

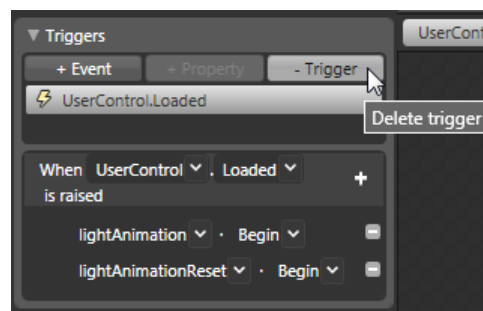


Figure 8.17

- Save the project and return to Visual Studio.



Note: Visual Studio notices that the project has been changed in another tool and prompts you to reload it.



Hint: Always save the project in the old tool and reload it in the new tool when switching between tools to make sure that your work isn't replaced with old data ever.

- Back in Visual Studio, edit `TrafficLightUserControl.xaml.cs`
- Add a public `DependencyProperty` **TrafficLightValueProperty** and a public property **TrafficLightValue** to the **TrafficLightUserControl** class.

```
...
public partial class TrafficLightUserControl
{
    public static DependencyProperty TrafficLightValueProperty
        = DependencyProperty.Register(
            "TrafficLightValue", typeof(decimal),
            typeof(TrafficLightUserControl));

    public decimal TrafficLightValue
    {
        get { return (decimal)GetValue(TrafficLightValueProperty); }
        set { SetValue(TrafficLightValueProperty, value); }
    }
}
```

...

Source Code Snippet 8b

- In the constructor, instantiate the **DependencyPropertyDescriptor** and the **AddValueChanged** event handler with its code inline (using a **delegate**).
- The event handler first retargets the **lightAnimationReset** storyboard (that we created for the red light only) to all three ellipses and plays it each time.
- Then it reads the current **TrafficLightValue** and retargets the **lightAnimation** to the proper ellipse (ellipseRed for 1, ellipseOrange for 2 and ellipseGreen for 3) and plays it.
- Also we need to set the Background of **borderReflection** again.

```
...
public TrafficLightUserControl()
{
    this.InitializeComponent();

    borderReflection.Background = new VisualBrush(borderMain);

    DependencyPropertyDescriptor dpd = DependencyPropertyDescriptor.FromProperty(
        TrafficLightValueProperty, typeof(TrafficLightUserControl));
    if (dpd != null)
    {
        dpd.AddValueChanged(this, delegate
        {
            Storyboard sb = (Storyboard)this.Resources["lightAnimation"];
            Storyboard sbr = (Storyboard)this.Resources["lightAnimationReset"];

            sbr.Children[0].SetValue(Storyboard.TargetNameProperty, "ellipseRed");
            sbr.Begin();
            sbr.Children[0].SetValue(Storyboard.TargetNameProperty, "ellipseOrange");
            sbr.Begin();
            sbr.Children[0].SetValue(Storyboard.TargetNameProperty, "ellipseGreen");
            sbr.Begin();

            if (this.TrafficLightValue == 1)
                sb.Children[0].SetValue(Storyboard.TargetNameProperty, "ellipseRed");
            else if (this.TrafficLightValue == 2)
                sb.Children[0].SetValue(Storyboard.TargetNameProperty, "ellipseOrange");
            else
                sb.Children[0].SetValue(Storyboard.TargetNameProperty, "ellipseGreen");

            sb.Begin();

        });
    }
}
...
```

Source Code Snippet 8c

- Don't forget to add using statements for the **System.ComponentModel** and **System.Windows.Media.Animation** namespaces.
- The last thing to do for our traffic light is to add it to the **viewboxDetail** control and bind its **DataContext** and **TrafficLightValue** properties to the **data source**.

```
...
<Viewbox Grid.Column="2" Grid.Row="1" VerticalAlignment="Top" Name="viewboxDetail" >
    <StackPanel Margin="20,15,10,0" Name="stackPanelDetail" >
```

```

<TextBlock>Customers Details</TextBlock>
<TextBlock Height="20" Margin="0,5,0,0"
    DataContext="{Binding }" Text="{Binding Employees.FirstName}" />
<TextBlock Height="20" Margin="0,5,0,0"
    DataContext="{Binding }" Text="{Binding Employees.LastName}" />
<TextBlock Height="20" Margin="0,5,0,0"
    DataContext="{Binding }" Text="{Binding Employees.Region}" />
<local:TrafficLightUserControl x:Name="trafficLightControl1"
    DataContext="{Binding }" TrafficLightValue="{Binding ShipVia}" />
</StackPanel>
</Viewbox>
...

```

Source Code Snippet 8d

- Run the application and see our traffic light control in action. Select different rows and make sure the storyboards play to show a smooth transition between changing lights. Also resize the application and move the grid splitter to see our detail grid including the traffic light properly resize based on the available space.

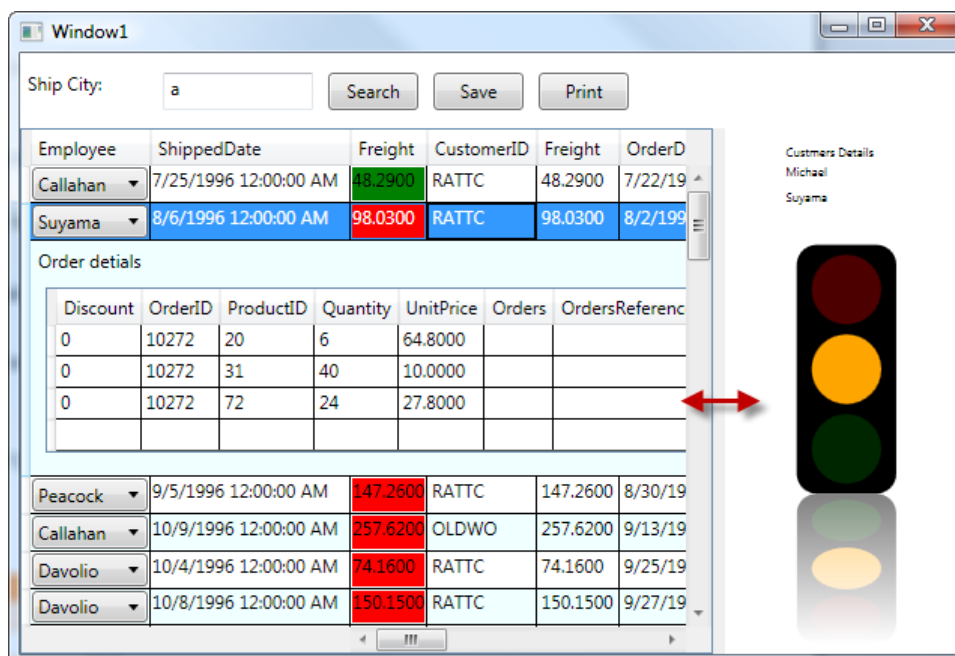


Figure 8.18

Step 9: Printing



Goal: We want to be able to print the data grid. To do this, we will resize the **dataGridMain** on the fly to fit the default printer's paper size, shrink its font size and send it to the printer and then resize it back to screen size.

- Add references to the libraries "**System.Printing.dll**" and "**ReachFramework.dll**" using the "**Add Reference**" dialog.



Note: **System.Printing** contains basic printing capabilities. We want to query the selected printer's capabilities which are implemented by **ReachFramework.dll**.

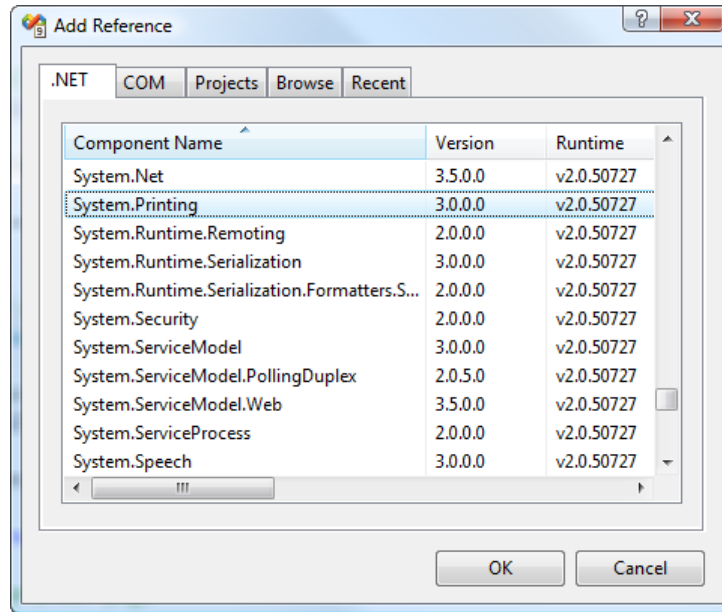


Figure 9.1

- In Visual Studio, double click the **buttonPrint** control in the designer to have an event handler for the **Click** event generated.
- In the event handler, invoke a **System.Windows.Controls.PrintDialog** and check if the user confirmed the dialog.
- If **true**, call the **PrintVisual** method on the **PrintDialog** and send it the **dataGridMain** control to print.

```

...
// Event Handler: buttonPrint Click event
private void buttonPrint_Click(object sender, RoutedEventArgs e)
{
    // Show the print dialog and let the user pick a printer
    PrintDialog printDlg = new System.Windows.Controls.PrintDialog();

    // Print, if the user clicked "Print"
    if (printDlg.ShowDialog() == true)
    {
        // Print dataGridMain "as is"
        printDlg.PrintVisual(dataGridMain, "Order Manager Print");
    }
}
...

```

Source Code Snippet 9a

- Compile and run the code. Query some data and hit the **Print** button. In the print dialog, select a printer driver that prints to a file like for example "Microsoft XPS Document Writer" or "Microsoft Office Document Image Writer" if you have Office installed.
- Open and inspect the resulting, printed document.



Note: You will see that dataGridMain printed in its current size which is not optimal for printing. We will optimize this in the next step.

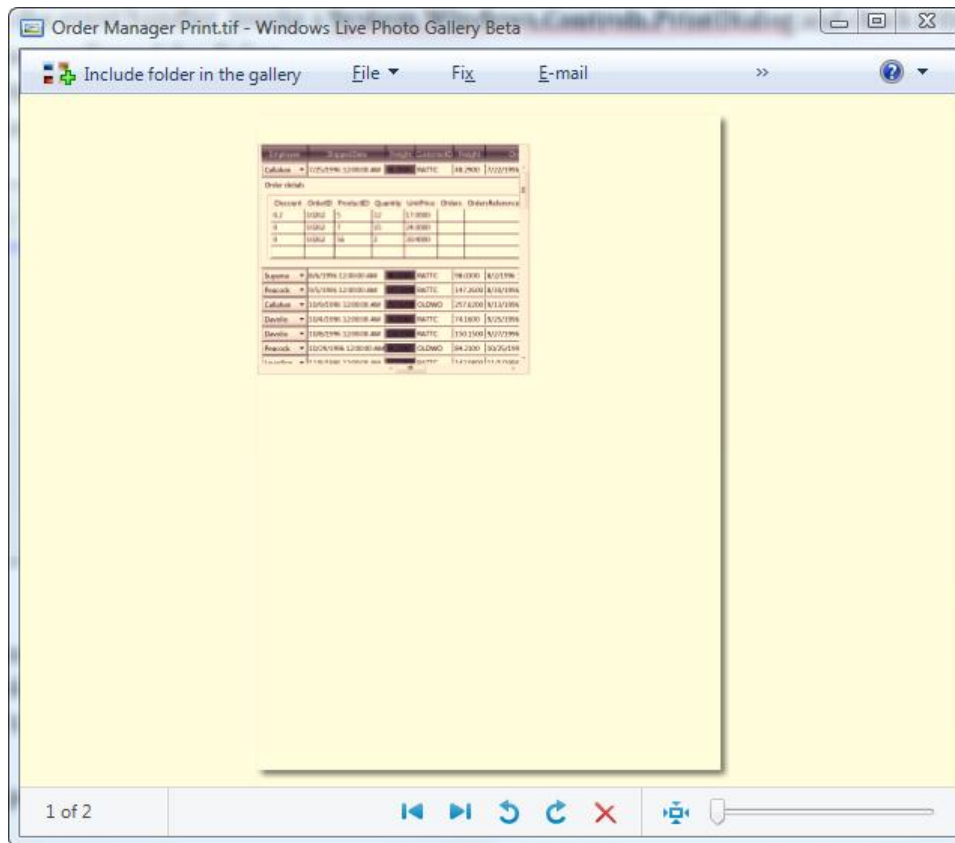


Figure 9.2

- Now swap in the following code that scales **dataGridMain** down to 50% due to the higher DPI of printers and then resizes it to fit the paper size of the selected printer before printing.

```
...
// Event Handler: buttonPrint Click event
private void buttonPrint_Click(object sender, RoutedEventArgs e)
{
    // Show the print dialog and let the user pick a printer
    PrintDialog printDlg = new System.Windows.Controls.PrintDialog();

    // Print, if the user clicked "Print"
    if (printDlg.ShowDialog() == true)
    {
        // Get selected printer capabilities
        System.Printing.PrintCapabilities capabilities =
            printDlg.PrintQueue.GetPrintCapabilities(printDlg.PrintTicket);

        // Hide dataGridMain's scroll bars for printing
        dataGridMain.VerticalScrollBarVisibility = ScrollBarVisibility.Disabled;
        dataGridMain.HorizontalScrollBarVisibility = ScrollBarVisibility.Disabled;

        // Scale dataGridMain to 50% due to printer's higher DPI over the screen
        dataGridMain.LayoutTransform = new ScaleTransform(0.5, 0.5);
        dataGridMain.Margin = new Thickness(20);

        // Get the printer's default page size
        Size sz = new Size(capabilities.PageImageableArea.ExtentWidth,
            capabilities.PageImageableArea.ExtentHeight);
    }
}
```

```

        // Update the layout of dataGridMain to the printer page size
        dataGridMain.Measure(sz);
        dataGridMain.Arrange(new Rect(new Point(
            capabilities.PageImageableArea.OriginWidth,
            capabilities.PageImageableArea.OriginHeight), sz));

        // Now print dataGridMain fitting on one page
        printDlg.PrintVisual(dataGridMain, "Order Manager Print");

        // Resize dataGridMain back to the original size
        dataGridMain.VerticalScrollBarVisibility = ScrollBarVisibility.Auto;
        dataGridMain.HorizontalScrollBarVisibility = ScrollBarVisibility.Auto;
        dataGridMain.LayoutTransform = new ScaleTransform(1, 1);
        dataGridMain.Margin = new Thickness(0);
    }
}
...

```

Source Code Snippet 9b

- Re-run the application and print some data. You will see **dataGridMain** now represented much more printer-friendly.

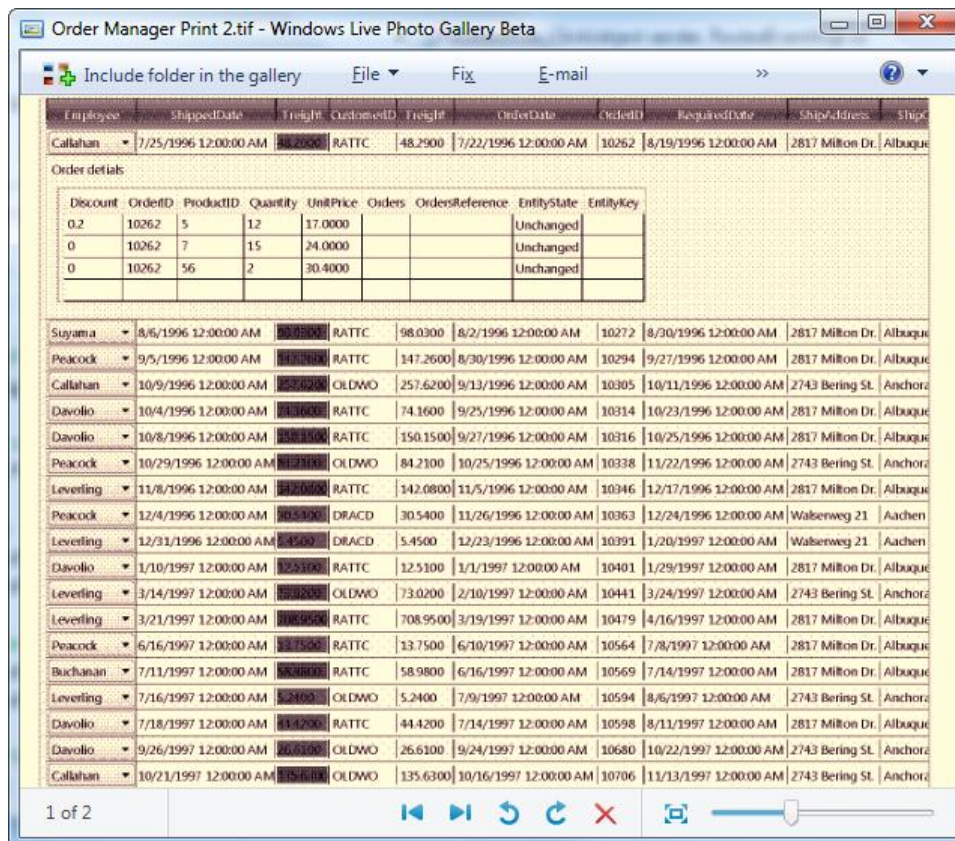


Figure 9.3



Note: This is just a very simple example to give you an idea on how to implement printing. In real life, you might think about implementing a special user control that displays the data completely different just for printing.

When printing on paper, this example works fine. With the Office Document Image Writer printer driver however, the control may not always be stretched to fill the entire page.

Step 10: Styling the Application



Goal: In this step we will enhance the look of our application using styles.

First, we will add a few ready-made items to our project, namely some images and a XAML resource containing predefined colors and styles.

- Copy the contents of the folder “C:\WPF HOL\Resources\Styling” into our project folder “C:\WPF HOL\Project\WpfLobApp\WpfLobApp”.
- Include these new items in the project in Visual Studio by clicking on the **project name** in the **Solution Explorer** and then activating “**Show all files**” from the menu.

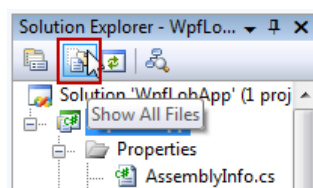


Figure 10.1

- Now find the grayed out “**images**” folder as well as the **ColorsStyles.xaml** file. Right click them and select “**Include in Project**”.

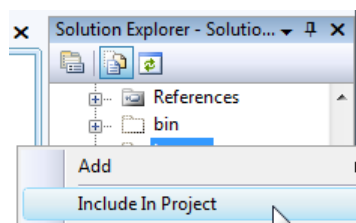


Figure 10.2

- We have to tell our project about the new **ColorsStyles.xaml** resource file. To do this, open **App.xaml** and add the following “**ResourceDictionary**” element:

```
<Application x:Class="WpfLobApp.App"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  StartupUri="Window1.xaml">
  <Application.Resources>

    <ResourceDictionary>
      <ResourceDictionary.MergedDictionaries>
        <ResourceDictionary Source="ColorsStyles.xaml"/>
      </ResourceDictionary.MergedDictionaries>
    </ResourceDictionary>

  </Application.Resources>
</Application>
```

Source Code Snippet 10a

- Save and rebuild your project.

- Switch to **Expression Blend**.
- Activate the resources pane. You will see the new “**ColorsStyles.xaml**” in the list of resources. Expand it and inspect the added resources. You can right-click each resource and select “**View XAML**” to see the XAML associated with the resource.

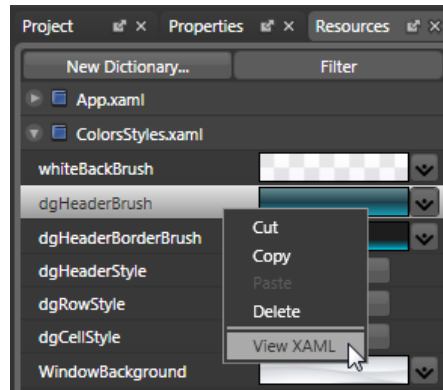


Figure 10.3

- Activate “**Window1.xaml**” and the **Properties** pane again.
- Select the “**Window**” element from the “**Objects and Timeline**” pane and set its **Brushes** → **Background** setting to the Brush resource “**WindowBackground**”.

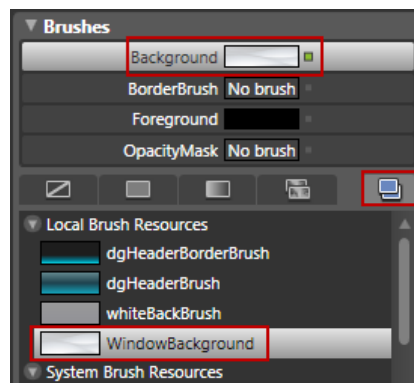


Figure 10.4

- Let’s also use this opportunity to give our window a proper **Title** “**Order Manager**” and **Icon** “**LOB.png**”.



Note: The image will show up in the drop-down list for the icon property as we have imported images as resources into our project earlier.

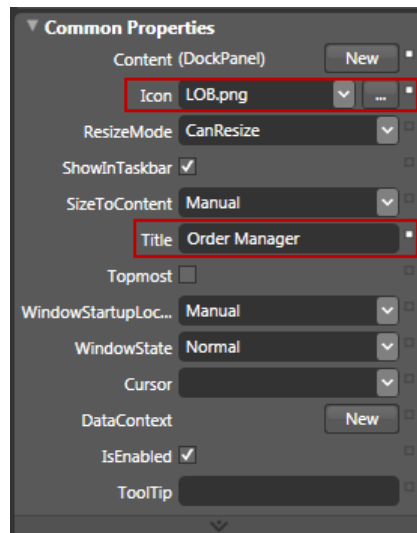


Figure 10.5

- Now find the dataGridMain control and apply the following styles:

Property	Value
Columns → CellStyle	dgCellStyle
Headers → ColumnHeaderStyle	dgHeaderStyle
Rows → RowStyle	dgRowStyle



Hint: You can easily do this by clicking on the **small square** right of each property and then selecting “Local Resource → {resource name}”. You will see the entire property circled in a green border when a local resource is applied.

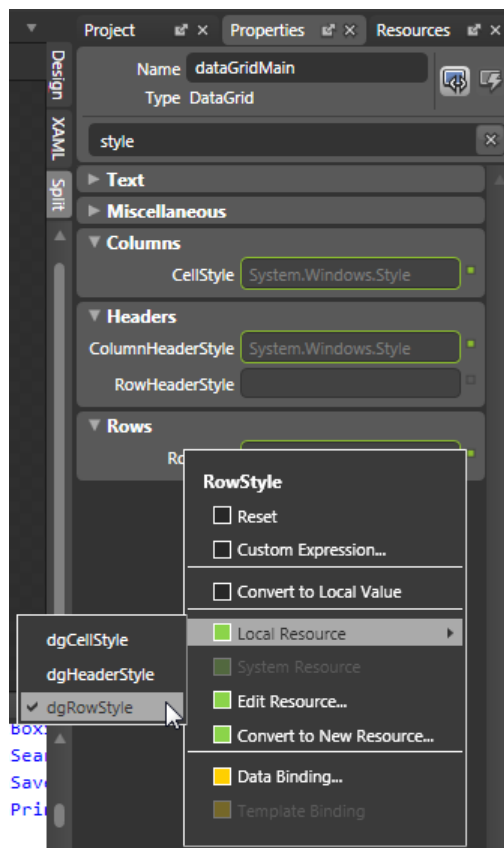


Figure 10.6

- Save the project in Expression Blend, switch to Visual Studio and reload it.
- Run the application and note the new styles in action on the **Window1** and **DataGridMain** controls.

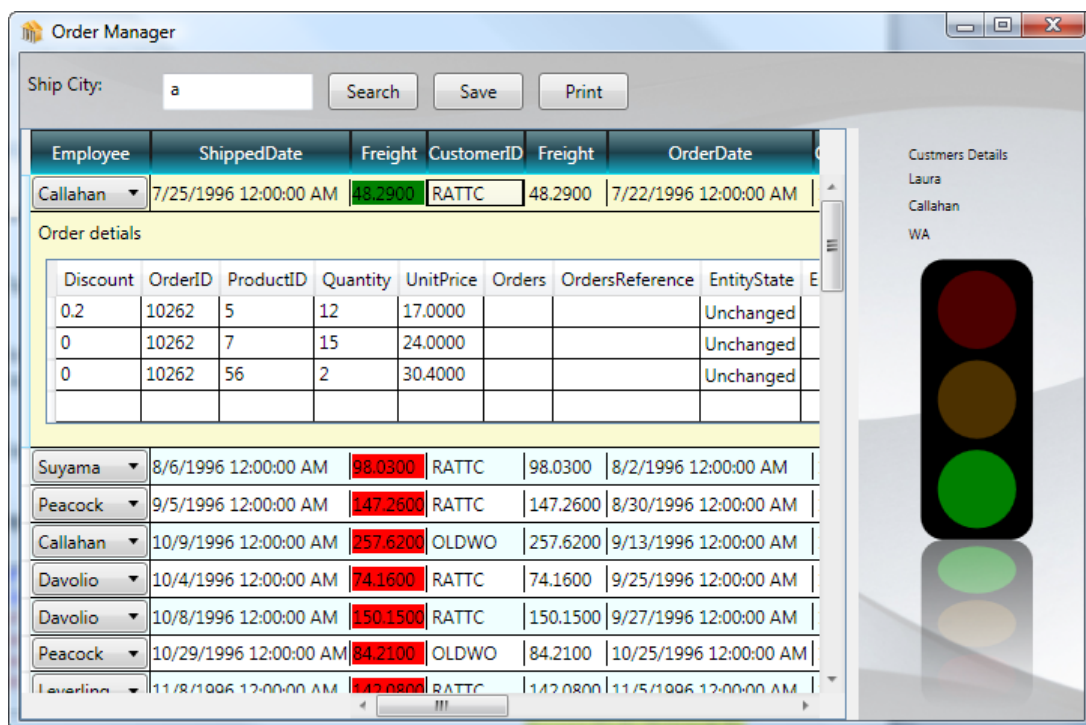


Figure 10.7



Note: Optionally, if you decided to download the **WPF ribbon control** and sign the online **Office UI license**, you may follow this last step in our hands-on lab, adding an Microsoft Office 2007-style Ribbon to the Order Manager application.

Step 11: Utilizing the Ribbon Control.

Since the release of Office 2007, people have started adopting the Office Ribbon in their applications. It provides a modern and user friendly look for an application:

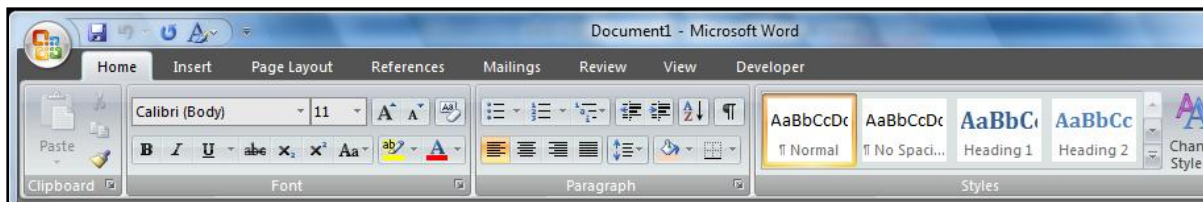


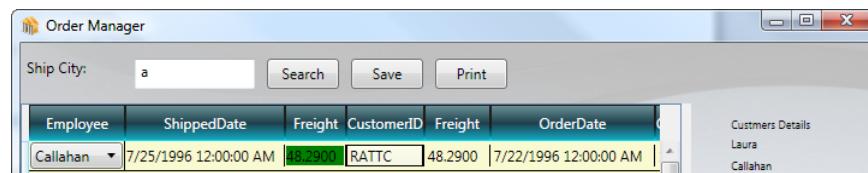
Figure 11.1

Now, with the WPF Toolkit, you can take advantage of the built-in Ribbon control. In this exercise, we will examine the Ribbon and see how to utilize it in our application. It supports all the features you see in the Office variety along with WPF's styling and theme capabilities.



Goal: This exercise is intended to demonstrate how to utilize the Ribbon – we won't cover all scenarios here, but at the end you will have used the Ribbon in the most common fashion and seen what it takes to integrate it into an application. We will also apply the Vista Glass effect to our window to make it look more modern.

Before:



After:

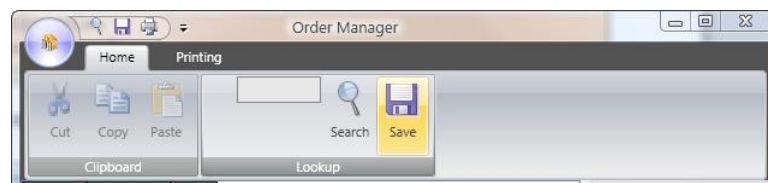


Figure 11.2

Step 11.1: Adding the basic Ribbon Control.



Goal: We will replace our simple input menu with a ribbon bar as we know it from the Microsoft Office 2007 applications.

- The first step is to add a reference to the Ribbon assembly. Since the Ribbon is licensed (for free) under the Office Fluent UI License, you must agree to this license before you can use the Ribbon control. To get the Ribbon binary:
 - Go to <http://msdn.microsoft.com/officeui>.
 - Click on “License the Office UI.”
 - Sign in using your Windows Live ID. If you don’t have a Windows Live ID already, you can create one for free using the links on that page.
 - Fill out the Office Fluent UI License form and agree to the license.
 - On the following page, click on the button that says “WPF Ribbon Control” and download the Ribbon.
- Next, you’ll need to add the **RibbonControlsLibrary.dll** to this project. Extract the Ribbon download, find the **RibbonControlsLibrary.dll** in “WPFRibbonCTP\WPFRibbonCTP\RibbonBinaries”.



Note: The Ribbon control is shipped with its full source code for you to study or enhance it to your liking.

- Add a reference to the Ribbon assembly **RibbonControlsLibrary.dll** by right-clicking on the **Reference Folder** in the **Solution Explorer** and selecting “**Add Reference...**”.
- In the “**Add Reference**” dialog, select the “**Browse**” tab and select **RibbonControlsLibrary.dll**.

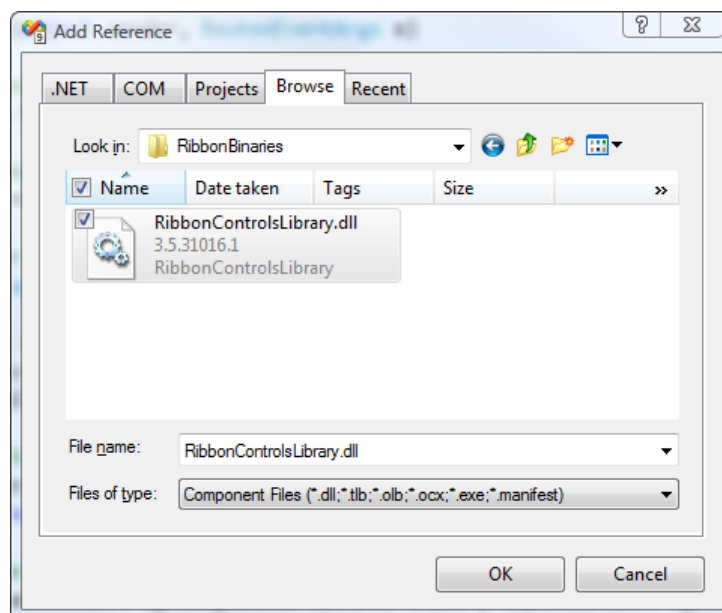


Figure 11.3

- Next, add the **RibbonControlsLibrary.dll** to Visual Studio’s Toolbox
- Right-click in the toolbox and select “**Add Tab**”.
- Name the tab “**WPF Ribbon**” and select “**Choose Items...**” from the new tab’s context menu.
- In the “**Choose Toolbox Items**” Dialog, switch to the “**WPF Components**” tab and click the “**Browse**” button.
- Navigate to the **RibbonControlsLibrary.dll** again and select it.
- Sort the list by “**Assembly Name**” and make sure all elements from the “**RibbonControlsLibrary**” assembly are checked. Hit “**OK**”.

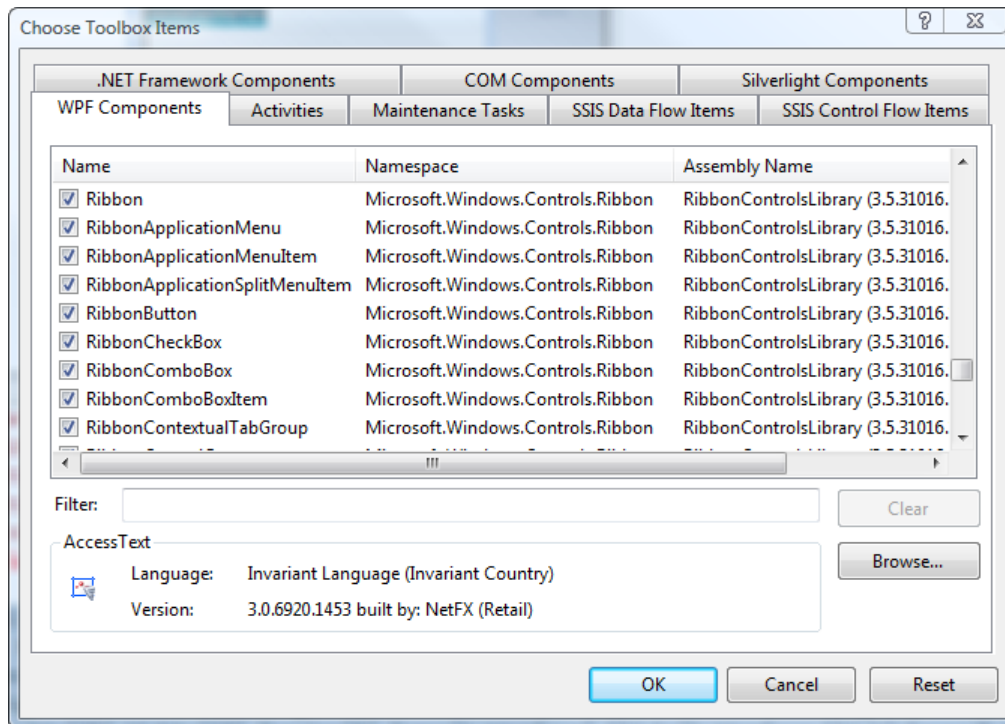


Figure 11.4

- Verify the “WPF Ribbon” tab is populated.
- Now open **Window1.xaml** and reduce the **height** of the **first row** in **gridMain** to **0** (we want to place the ribbon outside the grid and dock it to the top of the window).

```

...
<Grid Name="gridMain" Margin="0">
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="*" />
    <ColumnDefinition Width="8" />
    <ColumnDefinition Width="150" />
  </Grid.ColumnDefinitions>
  <Grid.RowDefinitions>
    <RowDefinition Height="0" />
    <RowDefinition Height="*" />
  </Grid.RowDefinitions>
...

```

Source Code Snippet 11a



Note: Doing this will hide **stackPanelTop**. This is the part that we will replace by a ribbon control.

- Next, add a namespace declaration to be able to access the Ribbon controls to the top of the file.
- Give it a prefix – the lab will use “rib”.
- The CLR namespace is “**Microsoft.Windows.Controls.Ribbon**” and the assembly is **RibbonControlsLibrary**.

```

...
<Window x:Class="WpfLabApp.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:rib="http://schemas.microsoft.com/winfx/2006/xaml"

```

```

xmlns:local="clr-namespace:WpfLobApp"
xmlns:data="http://schemas.microsoft.com/wpf/2008/toolkit"
xmlns:rib="clr-namespace:Microsoft.Windows.Controls.Ribbon;assembly=RibbonControlsLibrary"
Title="Order Manager"
Background="{DynamicResource WindowBackground}"
Height="480" Width="640" MinHeight="240" MinWidth="320" Icon="images\LOB.png">
...

```

Source Code Snippet 11b

- Now, just after **dockPanelMain**, add an **<rib:Ribbon>** control and dock it to the top of the window. You should see the ribbon appear in the design view.

```

...
<DockPanel Name="dockPanelMain" Margin="0">

    <rib:Ribbon DockPanel.Dock="Top">
    </rib:Ribbon>

...

```

Source Code Snippet 11c

The Ribbon is driven through **RoutedCommands** – just like menus and toolbars (normally). The Ribbon requires more than just the textual name however – it needs images, tooltip information and descriptions to display the command appropriately. To that end, the Ribbon defines a new subclass of **RoutedCommand** called **RibbonCommand** and everything you do with the Ribbon will generally involve a **RibbonCommand**.

Each **RibbonCommand** has several pieces of information which it adds to the **RoutedCommand**:

- **LabelText** – used to display a label on the item in the Ribbon
- **LabelDescription** – used to display descriptive information.
- **ToolTipTitle** – used to display a title for the tooltip created for the command
- **ToolTipDescription** – used to display a second line of text on the tooltip
- **ToolTipImageSource** – used to display an image on the tooltip
- **SmallImageSource** – the image used when the item is rendered in small size
- **LargeImageSource** – the image used when the item is rendered in large size
- **ToolTipFooterTitle** – used to display a footer title for the tooltip created for the command
- **ToolTipFooterDescription** – used to display a second line of text on the footer tooltip
- **ToolTipFooterImageSource** – used to display an image on the footer tooltip

The **ToolTip** properties are used by the Ribbon to automatically create tooltips for controls that are associated to the command. The **Small/Large** images are used to render the content for the control along with the **LabelText** in some cases.

The first step in using the Ribbon is to define the Application Menu. This is used to present the menu from the application menu which is the most evident portion of the Ribbon we see so far.

- Assign the **<r:Ribbon.ApplicationMenu>** property to a new **<r:RibbonApplicationMenu>** object.
 - The primary thing we need to assign is a **RibbonCommand** to the **Command** property of the **ApplicationMenu** – this is where the image and default action will come from.

- Create a new **<rib:RibbonApplicationMenu.Command>** element in the ribbon and add a **RibbonCommand** element with the following properties

Property	Value
Executed	OnCloseApplication
LabelText	Application Button
LabelDescription	Close the Application
SmallImageSource	images/LOB.png [NOTE THE CASING!]
LargeImageSource	images/LOB.png
ToolTipTitle	Order Manager
ToolTipDescription	{ some text to display under the title }



Note: The **Executed** property is a handler to the code behind method “OnCloseApplication” that executes “Close()”.

- The XAML should look like this.

```

...
<rib:Ribbon DockPanel.Dock="Top">

    <rib:Ribbon.ApplicationMenu>
        <rib:RibbonApplicationMenu>
            <rib:RibbonApplicationMenu.Command>
                <rib:RibbonCommand
                    Executed="OnCloseApplication"
                    LabelText="Application Button"
                    LabelDescription="Close the application."
                    SmallImageSource="images/LOB.png"
                    LargeImageSource="images/LOB.png"
                    ToolTipTitle="Order Manager"
                    ToolTipDescription="Click here to control the application." />
            </rib:RibbonApplicationMenu.Command>
        </rib:RibbonApplicationMenu>
    </rib:Ribbon.ApplicationMenu>

</rib:Ribbon>
...

```

Source Code Snippet 11d



Note: In XAML, you can **right-click** on the event handler name (**OnCloseApplication**) and select “**Navigate to Event Handler**” to be taken to the proper place in the code behind.

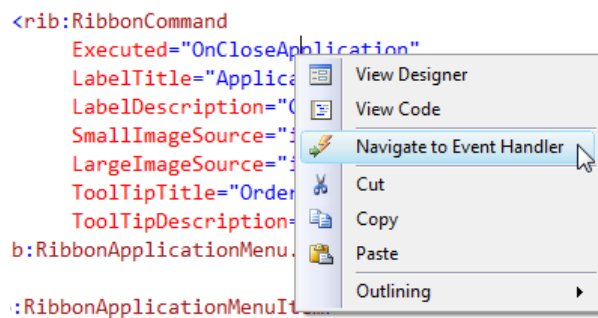


Figure 11.5

- In the code behind, add the OnCloseApplication event handler.

```
...
// Event Handler: Close
private void OnCloseApplication(object sender, ExecutedRoutedEventArgs e)
{
    Close();
}
...
```

Source Code Snippet 11e

- Run the application and hover over the button

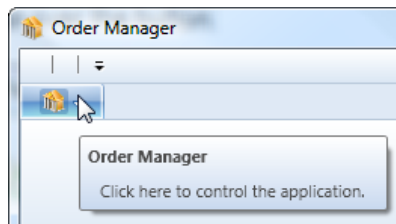


Figure 11.6

This is the effect of the **RibbonCommand** you created. It supplies the action and visual details to this element in the Ribbon and it's exactly how everything is setup in the Ribbon.



Note: If you click on the button, you will see that it pulls down an empty menu – this is because we didn't put any items into it. The application menu expects to have **RibbonApplicationMenuItem** children added to it.

- Let's add a simple menu item just as an example. Go back to the XAML and create a new **<rib:RibbonApplicationMenuItem>** inside the Ribbon application menu tag:

```
...
    ToolTipTitle="Order Manager"
    ToolTipDescription="Click here to control the application." />
</rib:RibbonApplicationMenu.Command>

<rib:RibbonApplicationMenuItem>
</rib:RibbonApplicationMenuItem>

</rib:RibbonApplicationMenu>
...
```

Source Code Snippet 11f

- Assign the **RibbonApplicationMenuItem.Command** property to a new **RibbonCommand** – use the same basic definition you assigned to the application menu itself, except change the **LabelText** to “Close”:

```
...
</rib:RibbonApplicationMenu.Command>

<rib:RibbonApplicationMenuItem>
    <rib:RibbonApplicationMenuItem.Command>
        <rib:RibbonCommand
```

```

        LabelTitle="_Close"
        LabelDescription="Close the Application"
        Executed="OnCloseApplication" />
    </rib:RibbonApplicationMenuItem.Command>
</rib:RibbonApplicationMenuItem>

</rib:RibbonApplicationMenu>
</rib:Ribbon.ApplicationMenu>
...

```

Source Code Snippet 11g

- Run the application again and note that you now have a menu item exposed in the ribbon application menu.

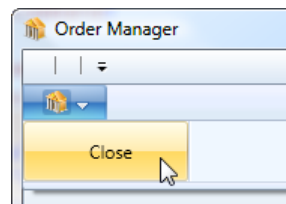


Figure 11.7

As you can see, **RibbonCommand** is very central to the behavior and appearance of the ribbon control itself. The next step would normally be to convert anything you want to expose through the Ribbon into a **RibbonCommand** – however that would take a significant amount of time so the we have prepared the code necessary for you.

- Copy the contents of the folder "C:\WPF HOL\Resources\Ribbon" into our project folder "C:\WPF HOL\Project\WpfLobApp\WpfLobApp".
- Include these new items in the project in Visual Studio by clicking on the **project name** in the **Solution Explorer** and then activating "**Show all files**" from the menu.

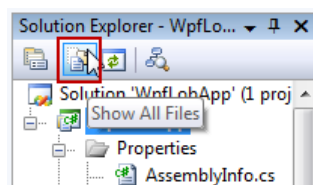


Figure 11.8

- Now find the grayed out "**AppCommands.cs**" and **AppCommands.xaml** files. Right click them and select "**Include in Project**".

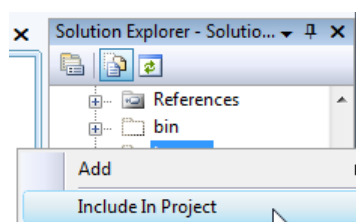


Figure 11.9

- Add the new **AppCommands.xaml** to the **Application.Resources** in **App.xaml**

```

...
<ResourceDictionary>
  <ResourceDictionary.MergedDictionaries>
    <ResourceDictionary Source="ColorsStyles.xaml"/>
    <ResourceDictionary Source="AppCommands.xaml" />
  </ResourceDictionary.MergedDictionaries>
</ResourceDictionary>
...

```

Source Code Snippet 11h

- Inspect “**AppCommands.cs**” and **AppCommand.xaml** and familiarize yourself with their structure.



Note: There are a couple of ways to create **RibbonCommands** – we’ve chosen to use XAML here because most of it is text and this makes it easier to internationalize the content, however you could create them completely in code-behind as well.

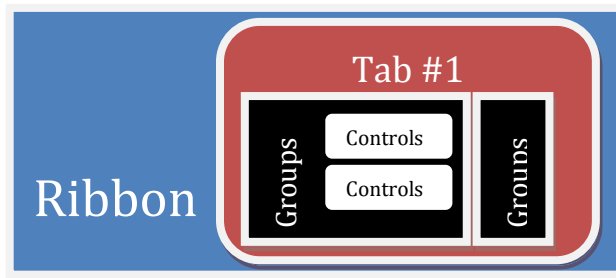
- Compile the code and make sure it runs.

Step 11.2: Customizing the Ribbon Control.



Goal: In this step we will add tabs, groups and controls to the Ribbon.

Next, let’s begin creating the ribbon itself. The Ribbon is composed of **Tabs**, **Groups** and **Controls**. Groups and Controls are bound to commands – that’s where they get their visual information as well as any Command to raise when they are invoked.



It’s helpful to sit down and decide how you want your commands to be organized and what the appropriate representation would be. Typically:

- **Tabs** are used to group common functionality – things that are used often, or features that naturally work together
- **Groups** are used within tabs to separate similar, but distinct commands. Office, for example, has a single Tab “Home” but has a group for Clipboard, Font, Paragraph, Styles and Editing.

That brings us to Controls. Controls are the elements the user interacts with. The most common control you will place on the Ribbon will be the button, but the Ribbon has several base

controls to choose from for the most common UI scenarios. For ultimate flexibility you can always create your own controls to be hosted in the ribbon. The Ribbon comes with:

- RibbonButton
- RibbonCheckBox
- RibbonToggleButton
- RibbonDropDownButton
- RibbonSplitButton
- RibbonComboBox
- RibbonTextBox
- RibbonLabel
- RibbonSeparator
- RibbonToolTip

All of these correspond directly to the normal WPF counterpart and in most cases are actually implemented by deriving from the underlying WPF control and adding the implementation for a new interface the Ribbon uses to manage the control called **IRibbonControl**. This is the interface you would use yourself to create new controls.

Our goal is to expose our complete set of features (i.e. the things we've create RibbonCommands for). We want to group them appropriately so for this exercise we will create the layout to look like:



Figure 11.10

Notice we have two Tabs ("Home" and "Printing") and two Groups in the Home tab ("Clipboard" and "Lookup"). Let's see how to implement that.

- Open **Window1.xaml** and locate your Ribbon control. Create two `<rib:RibbonTab>` elements and place them into the Ribbon
- Set the **Label** property on each to be the text you want displayed under the tab.

```
...
</rib:Ribbon.ApplicationMenu>

<rib:RibbonTab Label="Home">
</rib:RibbonTab>

<rib:RibbonTab Label="Printing">
</rib:RibbonTab>

</rib:Ribbon>
...
```

Source Code Snippet 11i

Now, let's define the groupings. Each Group has several things it needs – a Command (where it gets the image when it is sized too small to display the group as well as the Command it will raise for dialogs created from the group, and a set of controls to hold in the group.

- Create a **<rib:RibbonTab.Group>** element in the **Home** tab.
- Inside that, create a **<rib:RibbonGroup />**

```
...
<rib:RibbonTab Label="Home">
  <!-- Define the groups in this tab -->
  <rib:RibbonTab.Groups>
    <!-- Clipboard commands -->
    <rib:RibbonGroup>
    </rib:RibbonGroup>
  </rib:RibbonTab.Groups>
</rib:RibbonTab>
...
```

Source Code Snippet 11j

Each group can have a collection of **GroupSizeDefinitions** associated with it that tells the group exactly how to place any children. If it's not defined the Ribbon will use a default layout. Using these definitions, the Ribbon resizes the group based on the available size – this allows the ribbon to resize the group dynamically and the controls will shift around as necessary based on the definition.

If you need even more control, you can create your own custom layout and apply it to the Ribbon so it uses whatever criteria you desire to resize. For now, we will create a simple **GroupSizeDefinition** to place three controls next to each other using large icons:

- In the group, define a **<rib:RibbonGroup.GroupSizeDefinitions>** property
- Create a **<rib:RibbonGroupSizeDefinitionCollection>** and assign it to the property
- Add a **<rib:RibbonGroupSizeDefinition>** to the collection
- Add three **<rib:RibbonControlSizeDefinition>** objects to the group size definition created in the prior step.
 - Each should set the **ImageSize** to "Large" and **IsLabelVisible** to "True".
 - **Look at this next code but don't add it to your project just yet:**

```
...
<!-- Clipboard commands -->
<rib:RibbonGroup>
  <rib:RibbonGroup.GroupSizeDefinitions>
    <rib:RibbonGroupSizeDefinitionCollection>
      <rib:RibbonGroupSizeDefinition>
        <rib:RibbonControlSizeDefinition
          ImageSize="Large" IsLabelVisible="True"/>
        <rib:RibbonControlSizeDefinition
          ImageSize="Large" IsLabelVisible="True"/>
        <rib:RibbonControlSizeDefinition
          ImageSize="Large" IsLabelVisible="True"/>
      </rib:RibbonGroupSizeDefinition>
    </rib:RibbonGroupSizeDefinitionCollection>
  </rib:RibbonGroup.GroupSizeDefinitions>
</rib:RibbonGroup>
...
```

Source Code Snippet 11k

- To share common ordering and layout, you will commonly place your group size definitions into resources and then just assign the property to a single set of sizes. So instead of the code above, we will implement the following:

```
...
<rib:Ribbon DockPanel.Dock="Top">
  <rib:Ribbon.Resources>
    <rib:RibbonGroupSizeDefinitionCollection x:Key="RibbonLayout">
      <rib:RibbonGroupSizeDefinition>
        <!-- Control sizes: L,L,L -->
        <rib:RibbonControlSizeDefinition ImageSize="Large" IsLabelVisible="True"/>
        <rib:RibbonControlSizeDefinition ImageSize="Large" IsLabelVisible="True"/>
        <rib:RibbonControlSizeDefinition ImageSize="Large" IsLabelVisible="True"/>
      </rib:RibbonGroupSizeDefinition>
      <rib:RibbonGroupSizeDefinition>
        <!-- Control sizes: L,M,M -->
        <rib:RibbonControlSizeDefinition ImageSize="Large" IsLabelVisible="True"/>
        <rib:RibbonControlSizeDefinition ImageSize="Small" IsLabelVisible="True"/>
        <rib:RibbonControlSizeDefinition ImageSize="Small" IsLabelVisible="True"/>
      </rib:RibbonGroupSizeDefinition>
      <rib:RibbonGroupSizeDefinition>
        <!-- Control sizes: L,S,S -->
        <rib:RibbonControlSizeDefinition ImageSize="Large" IsLabelVisible="True"/>
        <rib:RibbonControlSizeDefinition ImageSize="Small" IsLabelVisible="False"/>
        <rib:RibbonControlSizeDefinition ImageSize="Small" IsLabelVisible="False"/>
      </rib:RibbonGroupSizeDefinition>
      <!-- Collapsed -->
      <rib:RibbonGroupSizeDefinition IsCollapsed="True" />
    </rib:RibbonGroupSizeDefinitionCollection>
  </rib:Ribbon.Resources>
  ...

  <rib:RibbonTab Label="Home">
    <!-- Define the groups in this tab -->
    <rib:RibbonTab.Groups>
      <!-- Clipboard commands -->
      <rib:RibbonGroup GroupSizeDefinitions="{StaticResource RibbonLayout}">
        </rib:RibbonGroup>
      </rib:RibbonTab.Groups>
    </rib:RibbonTab>
  ...
```

Source Code Snippet 111

Here we define several group size definitions – one for each layout we expect. Note that we assume here we will not have more than 3 controls in a group, clearly that would be a design decision and dictated by how you are organizing controls. This definition will now allow us to reuse this single group definition across all our defined groups. Now let's add some controls!

- We'll use **RibbonButton** controls for the clipboard elements – inside the **<rib:RibbonGroup>** add three **<rib:RibbonButton>** elements and tie the Command properties to
 - AppCommands.Cut**
 - AppCommands.Copy**
 - AppCommands.Paste**
- Finally, add a Command onto the group – just create a RibbonCommand directly and assign it to the **rib:RibbonGroup.Command** property.

- Set the **LabelText** to "Clipboard" and the **SmallImageSource** to "images/Paste.png". This will be used to render the group as a whole when it is collapsed.



Note: Currently, the XAML designer in Visual Studio has issues rendering controls as soon as the **Command** property is attached. The designer will keep displaying the following message even though the code compiles and runs.

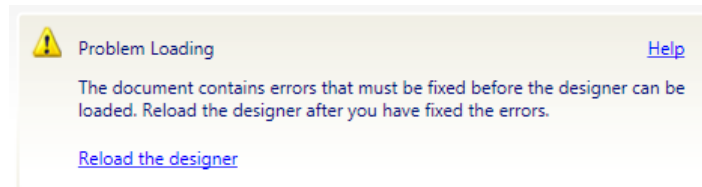


Figure 11.11

- Your code should look like:

```
...
<rib:RibbonTab.Groups>
  <!-- Clipboard commands -->
  <rib:RibbonGroup GroupSizeDefinitions="{StaticResource RibbonLayout}">
    <rib:RibbonGroup.Command>
      <rib:RibbonCommand LabelTitle="Clipboard" SmallImageSource="images/Paste.png"/>
    </rib:RibbonGroup.Command>
    <rib:RibbonButton Command="local:AppCommands.Cut"/>
    <rib:RibbonButton Command="local:AppCommands.Copy"/>
    <rib:RibbonButton Command="local:AppCommands.Paste"/>
  </rib:RibbonGroup>
</rib:RibbonTab.Groups>
...
```

Source Code Snippet 11m

- Run the application to see the effects of the group – you should now have valid Cut/Copy/Paste elements!



Note: Note that the Cut/Copy/Paste elements are disabled because we have not implemented event handlers for the three.



Figure 11.12

- Finish this tab by creating the additional groups in the screen shot – bind to the appropriate commands (you can see the list in **AppCommands.xaml**) and use the same technique you did in the previous steps to define the groups.

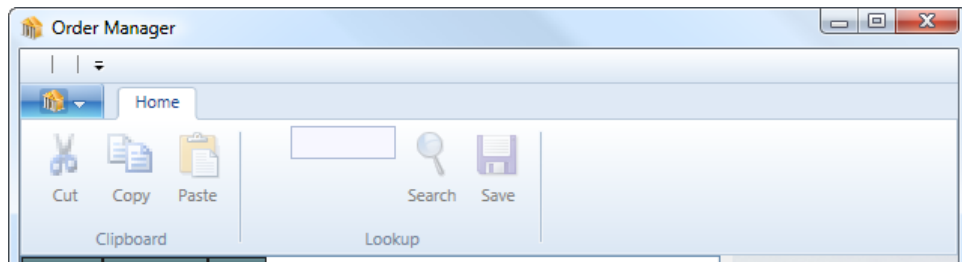


Figure 11.13



Note: Note that the first element is a `<rib:RibbonTextBox>` that we will name **"tbRibbon"**.

- The final code will look like this.

```
...
<rib:RibbonTab Label="Home">
  <rib:RibbonTab.Groups>
    <!-- Clipboard commands -->
    <rib:RibbonGroup GroupSizeDefinitions="{StaticResource RibbonLayout}">
      <rib:RibbonGroup.Command>
        <rib:RibbonCommand LabelTitle="Clipboard"
          SmallImageSource="images/Paste.png"/>
      </rib:RibbonGroup.Command>
      <rib:RibbonButton Command="local:AppCommands.Cut"/>
      <rib:RibbonButton Command="local:AppCommands.Copy"/>
      <rib:RibbonButton Command="local:AppCommands.Paste"/>
    </rib:RibbonGroup>
    <!-- Lookup commands -->
    <rib:RibbonGroup GroupSizeDefinitions="{StaticResource RibbonLayout}">
      <rib:RibbonGroup.Command>
        <rib:RibbonCommand LabelTitle="Lookup"
          SmallImageSource="images/Search.png" />
      </rib:RibbonGroup.Command>
      <rib:RibbonTextBox x:Name="tbRibbon" />
      <rib:RibbonButton Command="local:AppCommands.Search"/>
      <rib:RibbonButton Command="local:AppCommands.Save"/>
    </rib:RibbonGroup>
  </rib:RibbonTab.Groups>
</rib:RibbonTab>
...
```

Source Code Snippet 11n

- Run the application and make sure it works. Here as well, the new RibbonButtons are not enabled yet, we will bind them later.

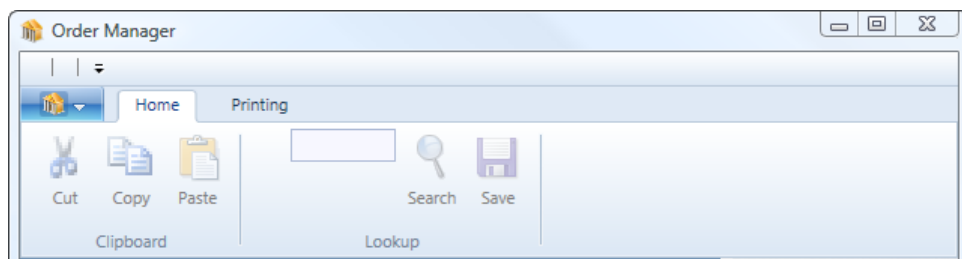


Figure 11.14

- The second tab looks much like the first with one exception – it uses a **RibbonDropDownButton** with a set of **MenuItem** objects to display a list of reports.

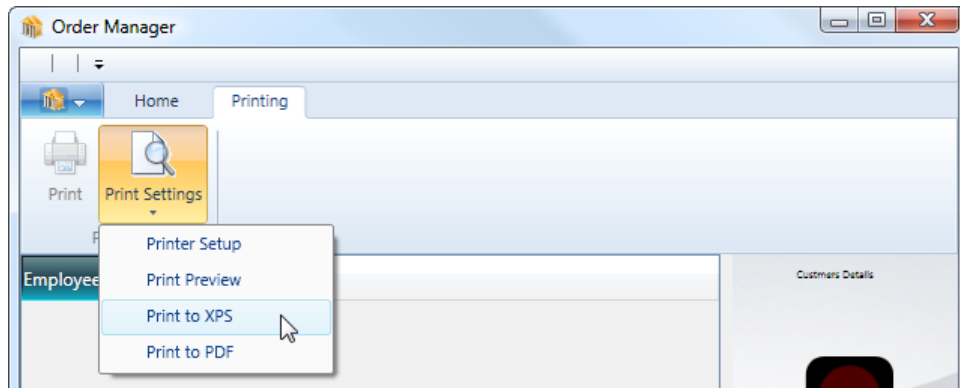


Figure 11.15

- Try creating the second tab yourself. The final code will look like this:

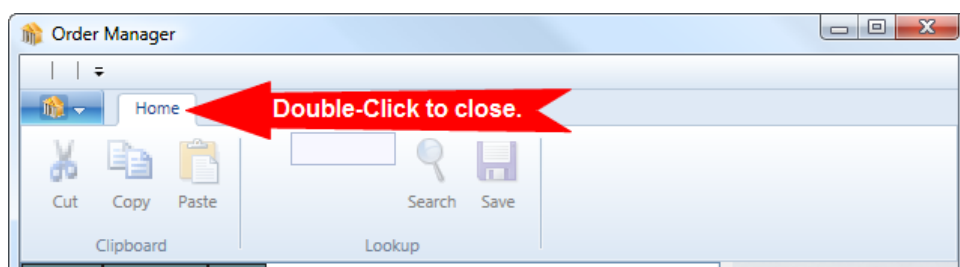
```

...
<rib:RibbonTab Label="Printing">
  <rib:RibbonTab.Groups>
    <rib:RibbonGroup GroupSizeDefinitions="{StaticResource RibbonLayout}">
      <rib:RibbonGroup.Command>
        <rib:RibbonCommand LabelTitle="Printing"
          SmallImageSource="images/Print.png" />
      </rib:RibbonGroup.Command>
      <rib:RibbonButton Command="local:AppCommands.Print" />
      <rib:RibbonDropDownButton Command="local:AppCommands.PrintPreview">
        <MenuItem Header="Printer Setup" />
        <MenuItem Header="Print Preview" />
        <MenuItem Header="Print to XPS" />
        <MenuItem Header="Print to PDF" />
      </rib:RibbonDropDownButton>
    </rib:RibbonGroup>
  </rib:RibbonTab.Groups>
</rib:RibbonTab>
...

```

Source Code Snippet 110

- Run the application and have a look at the enhanced ribbon. Try double-clicking on a **RibbonTab** to open and close the ribbon. In closed state, single-click a **RibbonTab** to pop-out the ribbon. The rest of the window's content should expand when the Ribbon is closed.



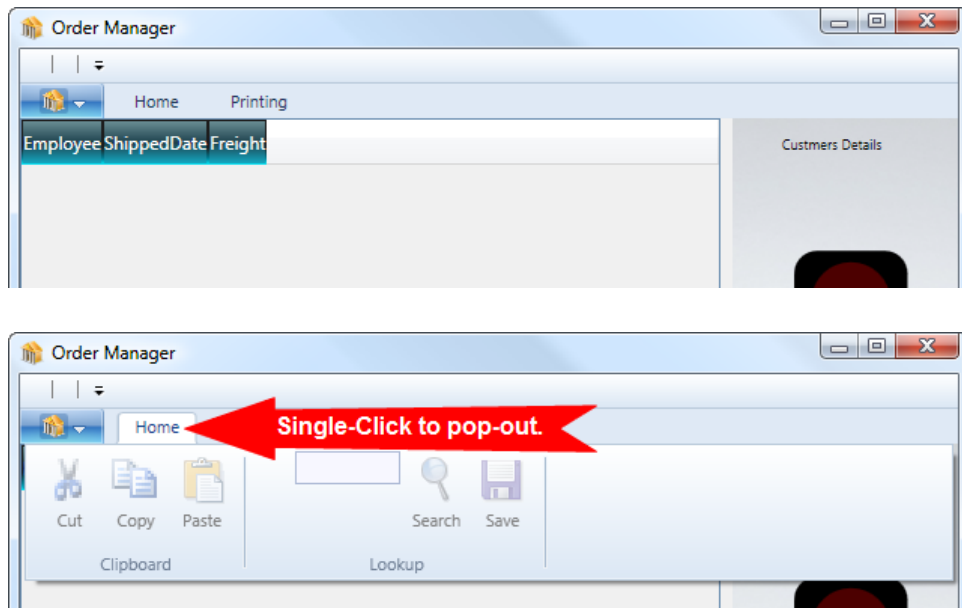


Figure 11.16

Still no button in our Ribbon is enabled, let's change that!

- Add the following **CommandBindings** to the **Window** object:

```
...
<Window x:Class="WpfLobApp.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="clr-namespace:WpfLobApp"
  xmlns:data="http://schemas.microsoft.com/wpf/2008/toolkit"
  xmlns:rib="clr-namespace:Microsoft.Windows.Controls.Ribbon;assembly=RibbonControlsLibrary"
  Title="Order Manager"
  Background="{DynamicResource WindowBackground}"
  Height="480" Width="640" MinHeight="240" MinWidth="320" Icon="images\LOB.png">

  <!-- Command bindings -->
  <Window.CommandBindings>
    <CommandBinding Command="local:AppCommands.Search" Executed="OnSearch" />
    <CommandBinding Command="local:AppCommands.Save" Executed="OnSave" />
    <CommandBinding Command="local:AppCommands.Print" Executed="OnPrint" />
  </Window.CommandBindings>

  <!-- Resources -->
  <Window.Resources>
    ...
```

Source Code Snippet 11p

- We will now repurpose the event handlers we implemented earlier to be called by the **Executed** properties of our **CommandBinding** elements. Change the three event handlers **buttonSearch_Click**, **buttonSave_Click** and **buttonPrint_Click** as follows.
- Also change **textBoxShipCity** to **tbRibbon**, our LINQ query will be getting the text from here now.

```
...
private void OnSearch(object sender, ExecutedRoutedEventArgs e)
{
```

```

        var query = from o in db.Orders.Include("Employees")
                     where o.ShipCity.StartsWith(tbRibbon.Text)
                     orderby o.OrderID ascending
                     select o;

        this.DataContext = query.ToList();
    }
...
    private void OnSave(object sender, ExecutedRoutedEventArgs e)
...
    private void OnPrint(object sender, ExecutedRoutedEventArgs e)
...

```

Source Code Snippet 11q

- Now **comment or delete** the **stackPanelTop** control, containing our text block, text box and buttons. This is the part that we have now replaced by a ribbon control.

```

...
<!--
<StackPanel Grid.ColumnSpan="3" Name="stackPanelTop" Orientation="Horizontal">
    <TextBlock Height="25" Width="80" Margin="5" Name="textBlock1" Text="Ship City:" />
    <TextBox Height="25" Width="100" Margin="5" Name="textBoxShipCity" />
    <Button Height="25" Width="60" Margin="5" Name="buttonSearch"
        Click="buttonSearch_Click">Search</Button>
    <Button Height="25" Width="60" Margin="5" Name="buttonSave"
        Click="buttonSave_Click">Save</Button>
    <Button Height="25" Width="60" Margin="5" Name="buttonPrint"
        Click="buttonPrint_Click">Print</Button>
</StackPanel>
-->
...

```

Source Code Snippet 11r



Hint: A quick and easy way to uncomment a code block in VS.NET is to highlight the block and press CTRL+K+U, use the Edit | Advanced | Uncomment Selection option from the menu bar or simply use the button in the toolbar.



- Run the application. The Search- Save- and Print-buttons should have come to life and the application should be working again now.

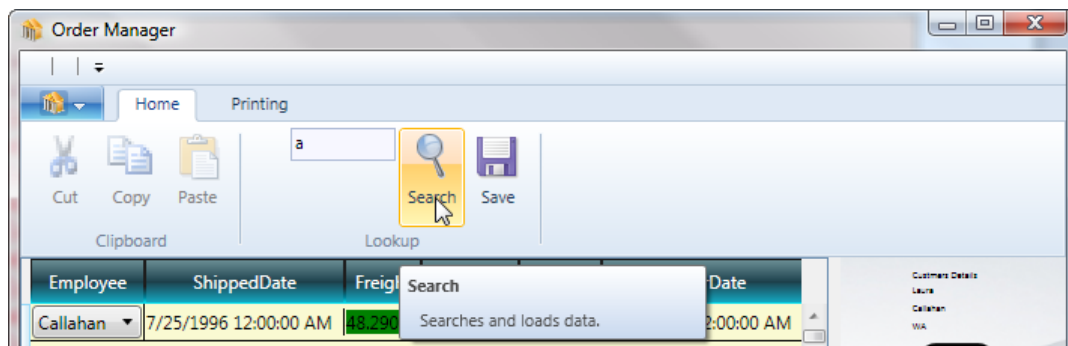


Figure 11.17

Step 11.3: Adding a Quick Access Toolbar to the Ribbon.



Goal: The next thing we want to do in our application is take the most actively used commands in the ribbon and add them to the “Quick Access Toolbar” list which is located next to the Application Menu.

- This is done by assigning a **QuickAccessToolBar** to the **Ribbon.QuickAccessToolBar** property.
- Create a `<r:Ribbon.QuickAccessToolBar>` tag in the ribbon (just after the resources is a good place).
- Add a `<r:RibbonQuickAccessToolBar>` into the tag and set the **CanUserCustomize** property to “True”
- Add your favorite commands using **RibbonButton** objects into the toolbar. The lab will use the commands **Search**, **Save** and **Print**.
- You can use the **RibbonQuickAccessToolBar.Placement** attached property on each button to decide whether the user can remove the command from the toolbar. The valid settings are “**InCustomizeMenu**”, “**InToolBar**” and “**InCustomizeMenuAndToolBar**”.
- The completed code looks something like:

```
...
</rib:Ribbon.Resources>

<!-- Quick access menu -->
<rib:Ribbon.QuickAccessToolBar>
  <rib:RibbonQuickAccessToolBar CanUserCustomize="True">
    <rib:RibbonButton Command="local:AppCommands.Search"
      rib:RibbonQuickAccessToolBar.Placement="InCustomizeMenuAndToolBar" />
    <rib:RibbonButton Command="local:AppCommands.Save"
      rib:RibbonQuickAccessToolBar.Placement="InCustomizeMenuAndToolBar" />
    <rib:RibbonButton Command="local:AppCommands.Print"
      rib:RibbonQuickAccessToolBar.Placement="InCustomizeMenuAndToolBar" />
  </rib:RibbonQuickAccessToolBar>
</rib:Ribbon.QuickAccessToolBar>

<rib:Ribbon.ApplicationMenu>
...

```

Source Code Snippet 11s

- Run the application and test the Quick Access Toolbar

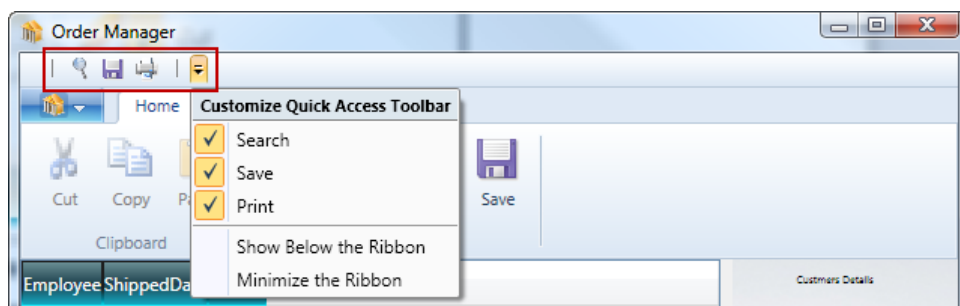


Figure 11.18



Note: Notice how our popular picks are now always available no matter which tab we are on and the user can click the down-arrow and customize the list (within the limits we provided through the attached properties).

Step 11.4: Providing Vista Glass Effects.



Goal: In this task, we will look at another common request – how to apply “glass” effects in our application when running under Windows Vista. The Ribbon control adds this capability through a new Window-derived class called **RibbonWindow**. This new class simply replaces the Window and automatically removes the title bar and makes the application more modern looking..



Note: In order to run this successfully and see the effect shown in the picture below, you will need to be running on Windows Vista with the “**Aero**” theme enabled.

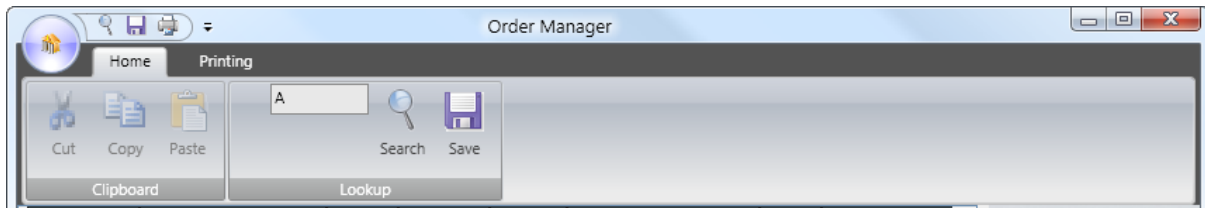


Figure 11.19

- Open the **Window1.xaml** file and replace the root **<Window>** tag with **<rib:RibbonWindow>**. Make sure to also replace the ending tag.
- In the code behind class file, replace the base “**Window**” class with “**RibbonWindow**”, or just remove it altogether as the partial class generated by the XAML compiler will have the proper base.
- Don’t forget to add a using statement for the **Microsoft.Windows.Controls.Ribbon** namespace.
- Run the application – notice how our quick access menu has moved into the title bar – this is what the **GlassWindow** will do – reduce the space necessary and give you the Office 2007 effect.

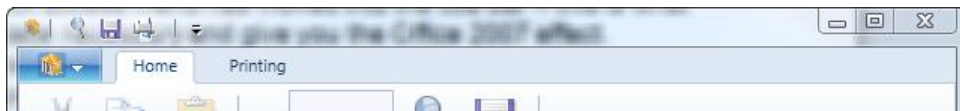


Figure 11.20

It looks decent but there are two issues with it

1. There is no title on the title bar now – it was removed by the Glass Window.
2. The icon is being displayed under the application menu, not replacing it.

We can fix both of these issues in the XAML.

- First, locate the **<Ribbon>** tag – we need to set a **Title** property on this element to get our title display with the Ribbon Window. We could move the Window version, or use data binding to bind to the existing window title.

```
...
<rib:Ribbon DockPanel.Dock="Top"
    Title="{Binding RelativeSource={RelativeSource FindAncestor,
        AncestorType={x:Type Window}},Path=Title}">
```

...

Source Code Snippet 11t

- This will walk the visual tree to locate the **Window** ancestor and then bind the ribbon's **Title** property to the Title property assigned to the window.
- Next, remove the **Icon** property off the **GlassWindow**.
- Run the application – it should now look more Office 2007 style.

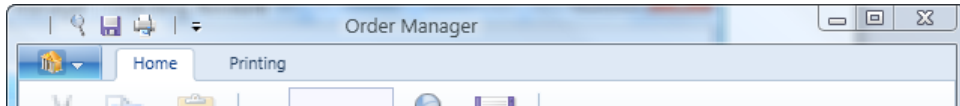


Figure 11.21

The final thing you might consider is to change the default style of the ribbon control – it, by default, utilizes a Windows 7 theme, but you can change it to look more like Office 2007 by simply merging in different styles into your application or window resources.

- Locate the window constructor in the code behind Window1.
- As the first line, add the following code to merge in the Office 2007 “black” theme

```
...
public Window1()
{
    this.Resources.MergedDictionaries
        .Add(Microsoft.Windows.Controls
            .Ribbon.PopularApplicationSkins.Office2007Black);

    InitializeComponent();
}
...
```

Source Code Snippet 11u

- Run the application and note the difference in appearance.

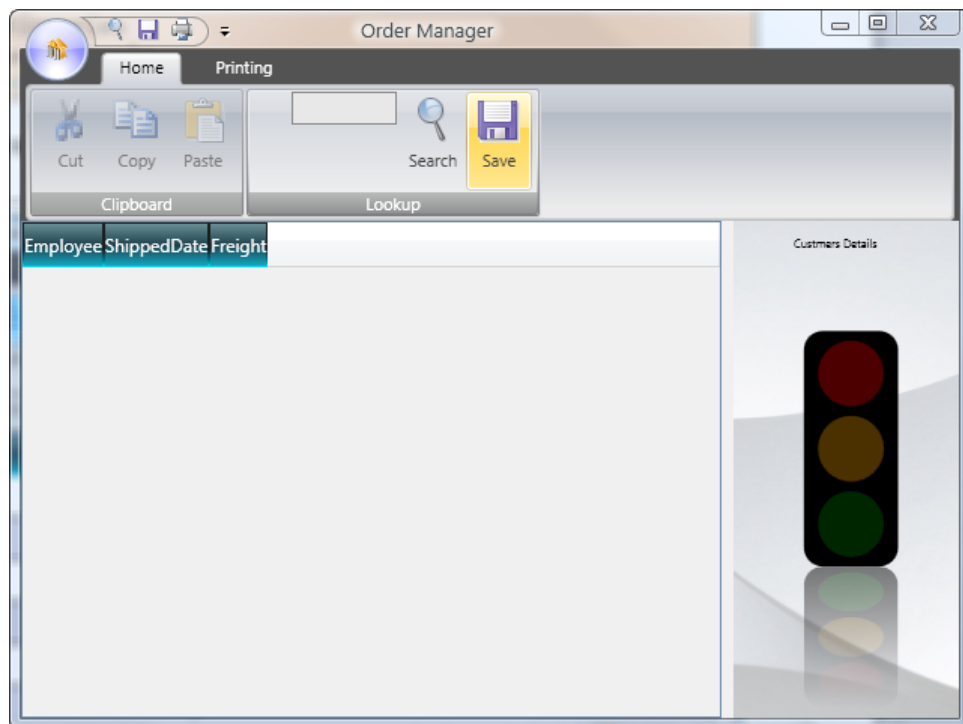


Figure 11.22

This concludes our hands-on lab. Hopefully you enjoyed learning about the new features of Windows Presentation Foundation. The next step is taking these ideas and applying them to your own projects.

The End.