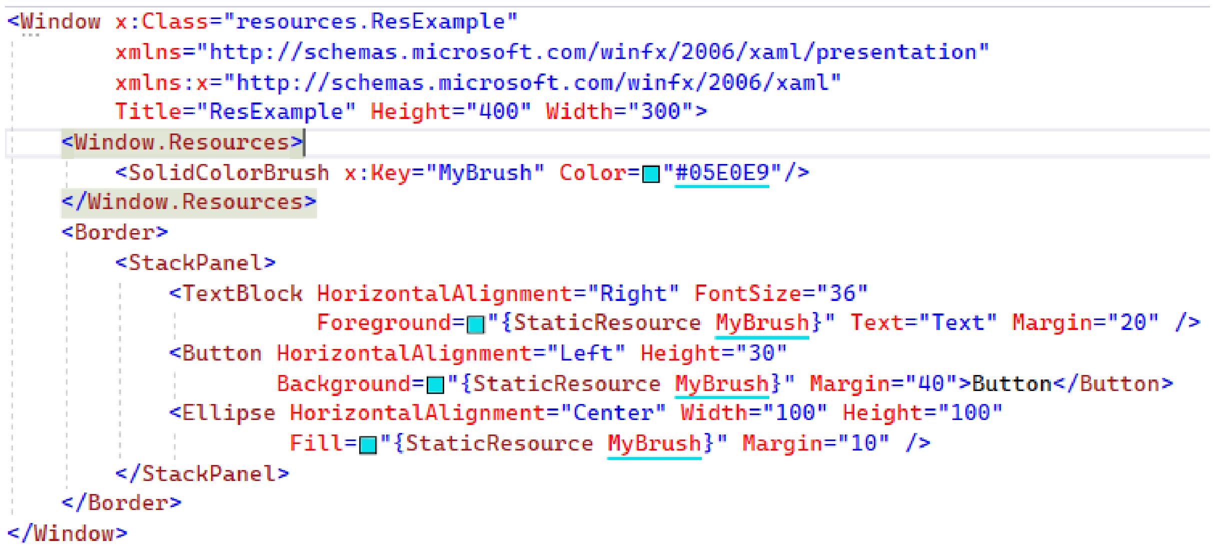
5. WPF - Resources and Data templates

5.1 Resources

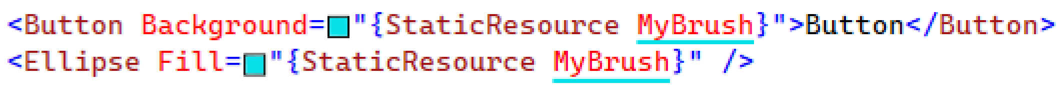
A resource is an object that can be reused in different places in your app. This overview describes how to use resources in Extensible Application Markup Language (XAML).

The following example defines a SolidColorBrush as a resource on the root element of a page. The example then references the resource and uses it to set properties of several child elements, including an Ellipse, a TextBlock, and a Button.



```
<Window x:Class="resources.ResExample"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="ResExample" Height="400" Width="300">
    <Window.Resources>
        <SolidColorBrush x:Key="MyBrush" Color="#05E0E9"/>
    </Window.Resources>
    <Border>
        <StackPanel>
            <TextBlock HorizontalAlignment="Right" FontSize="36"
                       Foreground="{StaticResource MyBrush}" Text="Text" Margin="20" />
            <Button HorizontalAlignment="Left" Height="30"
                    Background="{StaticResource MyBrush}" Margin="40">Button</Button>
            <Ellipse HorizontalAlignment="Center" Width="100" Height="100"
                     Fill="{StaticResource MyBrush}" Margin="10" />
        </StackPanel>
    </Border>
</Window>
```

Every framework-level element (FrameworkElement or FrameworkContentElement) has a Resources property, which is a ResourceDictionary type that contains defined resources. You can define resources on any element, such as a Button. However, resources are most often defined on the root element, which is Window in the example.
Each resource in a resource dictionary must have a unique key. When you define resources in markup, you assign the unique key through the x:Key Directive. Typically, the key is a string.
You can use a defined resource with the resource markup extension syntax that specifies the key name of the resource. For example, use the resource as the value of a property on another element.

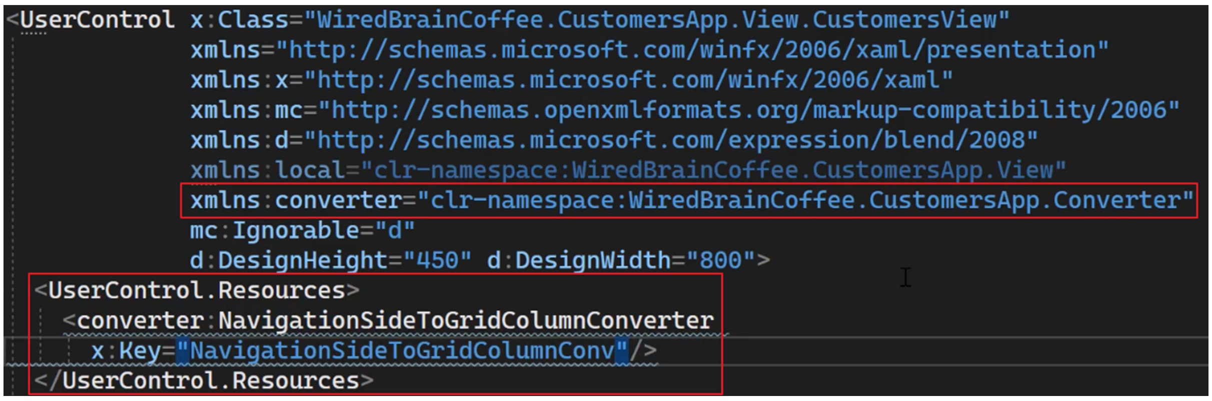


```
<Button Background="{StaticResource MyBrush}">Button</Button>
<Ellipse Fill="{StaticResource MyBrush}" />
```

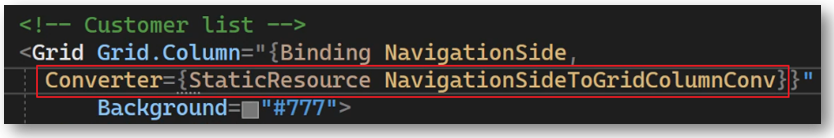
In the preceding example, when the XAML loader processes the value *{StaticResource MyBrush}* for the Background property on Button, the resource lookup logic first checks the resource dictionary for the Button element. If Button doesn't have a definition of the resource

key *MyBrush* (in that example it doesn't; its resource collection is empty), the lookup next checks the parent element of Button. If the resource isn't defined on the parent, it continues to check the object's logical tree upward until it's found.
If you define resources on the root element, all the elements in the logical tree, such as the Window or Page, can access it. And you can reuse the same resource for setting the value of any property that accepts the same type that the resource represents. In the previous example, the same MyBrush resource sets two different properties: Button.Background and Rectangle.Fill.

Below is another example that defines an instance of a converter as a resource:

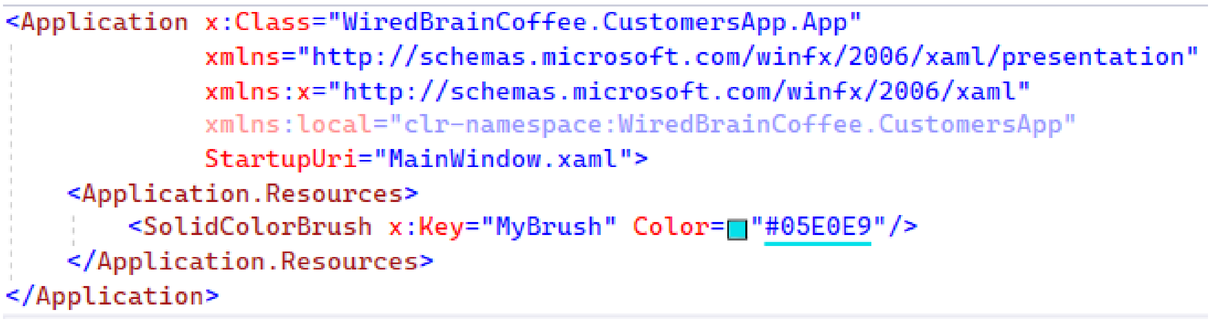


```
<UserControl x:Class="WiredBrainCoffee.CustomersApp.View.CustomersView"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
             xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
             xmlns:local="clr-namespace:WiredBrainCoffee.CustomersApp.View"
             xmlns:converter="clr-namespace:WiredBrainCoffee.CustomersApp.Converter"
             mc:Ignorable="d"
             d:DesignHeight="450" d:DesignWidth="800">
  <UserControl.Resources>
    <converter:NavigationSideToGridColumnConverter
      x:Key="NavigationSideToGridColumnConv"/>
  </UserControl.Resources>
```



```
<!-- Customer list -->
<Grid Grid.Column="{Binding NavigationSide,
  Converter={StaticResource NavigationSideToGridColumnConv}}"
      Background="#777">
```

Resources can also be defined at the application level, through the *App.xaml* file. Resources that are defined by the application are globally scoped and accessible by all parts of the application.



```
<Application x:Class="WiredBrainCoffee.CustomersApp.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:local="clr-namespace:WiredBrainCoffee.CustomersApp"
             StartupUri="MainWindow.xaml">
    <Application.Resources>
        <SolidColorBrush x:Key="MyBrush" Color="#05E0E9"/>
    </Application.Resources>
</Application>
```

5.1.1 Merge resource dictionaries

Windows Presentation Foundation (WPF) resources support a merged resource dictionary feature. This feature provides a way to define the resources portion of a WPF application outside of the compiled XAML application. Resources can then be shared across applications and are also more conveniently isolated for localization.
In markup, you use the following syntax to introduce a merged resource dictionary into a page:

```
<Page.Resources>
  <ResourceDictionary>
    <ResourceDictionary.MergedDictionaries>
      <ResourceDictionary Source="myresourcedictionary.xaml"/>
      <ResourceDictionary Source="myresourcedictionary2.xaml"/>
    </ResourceDictionary.MergedDictionaries>
  </ResourceDictionary>
</Page.Resources>
```

The ResourceDictionary element doesn't have an x:Key Directive, which is generally required for all items in a resource collection. But another ResourceDictionary reference within the MergedDictionaries collection is a special case, reserved for this merged resource dictionary scenario. Further, the ResourceDictionary that introduces a merged resource dictionary can't have an x:Key Directive.
Typically, each ResourceDictionary within the MergedDictionaries collection specifies a Source attribute. The value of *Source* should be a uniform resource identifier (URI) that resolves to the location of the resources file to be merged. The destination of that URI must be another XAML file, with *<ResourceDictionary>* as its root element.

Resources in a merged dictionary occupy a location in the resource lookup scope that's just after the scope of the main resource dictionary they are merged into. Although a resource key must be unique within any individual dictionary, a key can exist multiple times in a set of merged dictionaries. In this case, the resource that's returned will come from the last dictionary found sequentially in the MergedDictionaries collection. If the MergedDictionaries collection was defined in XAML, then the order of the merged dictionaries in the collection is the order of the elements as provided in the markup. If a key is defined in the primary dictionary and also in a dictionary that was merged, then the resource that's returned will come from the primary dictionary.

5.2 Flexible content model

The Content property of a ContentControl (e.g. a Button) can be set to a simple string.

```
<Button Content="Add customer"/>
```

But instead of a simple string, you can set the Content property to a complex object by using the property element syntax.
For example, you can assign a StackPanel that contains an image and a text block.

```
<Button>
  <Button.Content>
    <StackPanel Orientation="Horizontal">
      <Image Source="/Images/add.png"
        Height="18" Margin="0 0 5 0"/>
      <TextBlock Text="Add"/>
    </StackPanel>
  </Button.Content>
</Button>
```

Now, the Button has an Image and Text as a Content.

The property element Button.Content is optional. You can omit it, which makes your XAML code more compact.
So the Button supports a simple string as content, but also a complex UIElement, like a StackPanel. This is WPF's Flexible Content Model.
The Button is a subclass of ContentControl. This class defines the Content property.
Beside the Button class, also other classes inherit from ContentControl like, for example, CheckBox, Label, and Window.
The Content property is of type object. This means that you can assign an arbitrary object to the Content property. How the content is rendered depends on the type of the object that is assigned to the Content property:

- UIElement
    - UIElement is a WPF-based class for all panels and all controls. If you assign a UIElement to the Content property of a ContentControl, that UIElement is just rendered. You can assign an image element or a StackPanel to the Content property, and these elements are just rendered.
- Object that does not inherit from UIElement
    - By default, the ToString method on the object is called, and the result of the ToString method is rendered. Behind the scenes, WPF creates a text block UIElement to display the string.
    - The default behavior can be overridden by using a DataTemplate that defines a user interface to render the (complex) object. In the DataTemplate, you can use UIElements like panels and controls to create the user interface. You can bind properties of these elements to the object that was assigned to the Content property of the ContentControl. That object is automatically in the data context of the DataTemplate. You can assign a DataTemplate to a ContentControl by setting the ContentTemplate property of your ContentControl.

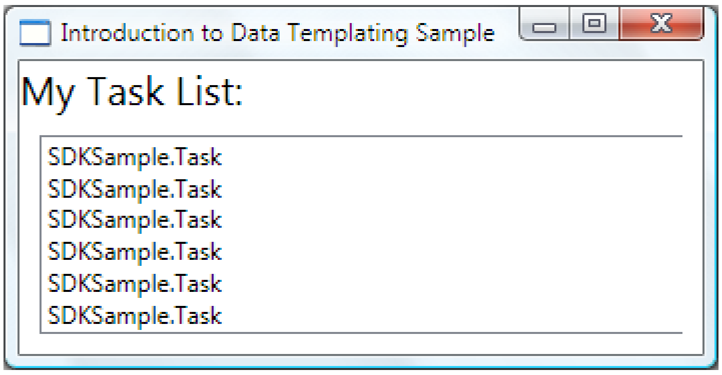
Besides the ContentControl class, WPF contains another class that supports the flexible content model, and that is the ItemsControl class. ContentControl is for a single object, and ItemsControl is for a collection of objects. Some typical classes that inherit from ItemsControl are ListView, ComboBox, and TabControl.
The ItemsControl class has an ItemsSource property to which you can assign a collection of objects. For these objects, the same rules apply as within the ContentControl. UIElements are rendered, and for any other object, the result of the ToString method is shown in a text block. If the ToString result is not what you want, you can assign a DataTemplate to the ItemTemplate property.
With that DataTemplate, you define the user interface for the objects in your ItemsControl.

5.3 Using a DataTemplate in an ItemsControl

To demonstrate why DataTemplate is important, let's walk through a data binding example. In this example, we have a ListBox that is bound to a list of Task objects. Each Task object has a TaskName (string), a Description (string), a Priority (int), and a property of type TaskType, which is an Enum with values Home and Work.

```
<Window x:Class="SDKSample.Window1"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local="clr-namespace:SDKSample"
        Title="Introduction to Data Templating Sample">
    <Window.Resources>
        <local:Tasks x:Key="myTodoList"/>
    </Window.Resources>
    <StackPanel>
        <TextBlock Name="blah" FontSize="20" Text="My Task List:"/>
        <ListBox Width="400" Margin="10"
                 ItemsSource="{Binding Source={StaticResource myTodoList}}"/>
    </StackPanel>
</Window>
```

Without a DataTemplate, our ListBox currently looks like this:



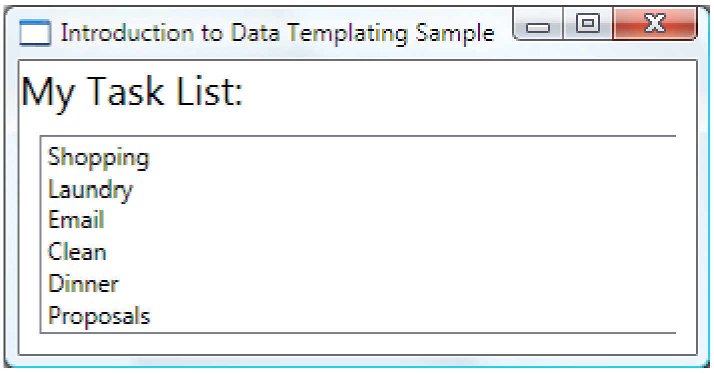
My Task List:

```
SDKSample.Task
SDKSample.Task
SDKSample.Task
SDKSample.Task
SDKSample.Task
SDKSample.Task
```

What's happening is that without any specific instructions, the ListBox by default calls ToString when trying to display the objects in the collection. Therefore, if the Task object overrides the ToString method, then the ListBox displays the string representation of each source object in the underlying collection.
For example, if the Task class overrides the ToString method this way, where name is the field for the TaskName property:

```
public override string ToString()
{
    return name.ToString();
}
```

Then the ListBox looks like the following:



My Task List:

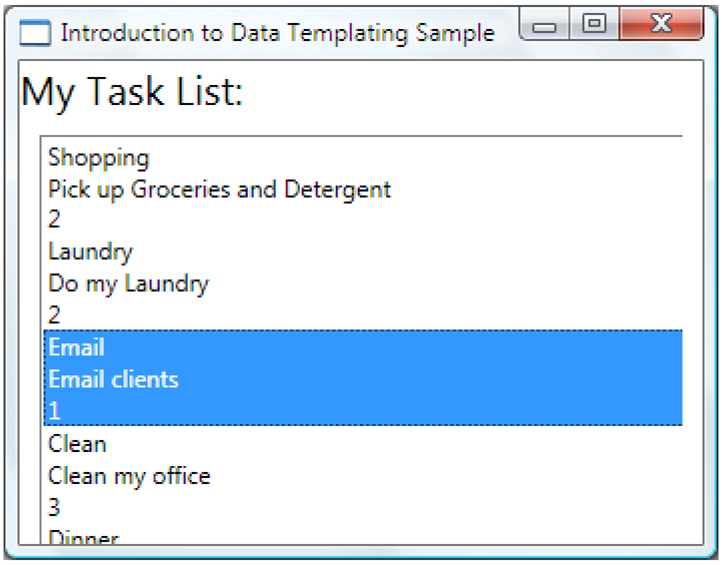
```
Shopping
Laundry
Email
Clean
Dinner
Proposals
```

However, that is limiting and inflexible.

The solution is to define a DataTemplate. One way to do that is to set the ItemTemplate property of the ListBox to a DataTemplate. What you specify in your DataTemplate becomes the visual structure of your data object. The following DataTemplate is fairly simple. We are giving instructions that each item appears as three TextBlock elements within a StackPanel. Each TextBlock element is bound to a property of the Task class.

```
<ListBox Width="400" Margin="10"
         ItemsSource="{Binding Source={StaticResource myTodoList}}">
   <ListBox.ItemTemplate>
     <DataTemplate>
       <StackPanel>
         <TextBlock Text="{Binding Path=TaskName}" />
         <TextBlock Text="{Binding Path=Description}"/>
         <TextBlock Text="{Binding Path=Priority}"/>
       </StackPanel>
     </DataTemplate>
   </ListBox.ItemTemplate>
 </ListBox>
```

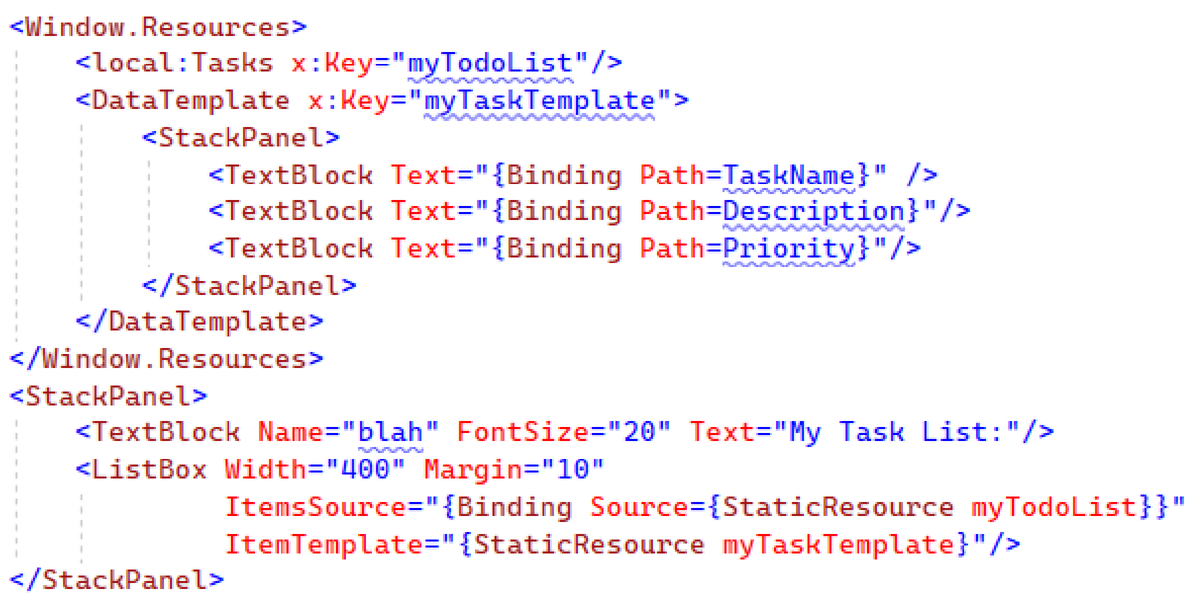
Now our ListBox looks like the following:



My Task List:

Shopping
Pick up Groceries and Detergent
2
Laundry
Do my Laundry
2
Email
Email clients
1
Clean
Clean my office
3
Dinner

In the above example, we defined the DataTemplate inline. It is more common to define it in the resources section so it can be a reusable object, as in the following example:



```
<Window.Resources>
    <local:Tasks x:Key="myTodoList"/>
    <DataTemplate x:Key="myTaskTemplate">
        <StackPanel>
            <TextBlock Text="{Binding Path=TaskName}" />
            <TextBlock Text="{Binding Path=Description}"/>
            <TextBlock Text="{Binding Path=Priority}"/>
        </StackPanel>
    </DataTemplate>
</Window.Resources>
<StackPanel>
    <TextBlock Name="blah" FontSize="20" Text="My Task List:"/>
    <ListBox Width="400" Margin="10"
             ItemsSource="{Binding Source={StaticResource myTodoList}}"
             ItemTemplate="{StaticResource myTaskTemplate}"/>
</StackPanel>
```

Because myTaskTemplate is a resource, you can now use it on other controls that have a property that takes a DataTemplate type. As shown above, for ItemsControl objects, such as the ListBox, it is the ItemTemplate property. For ContentControl objects, it is the ContentTemplate property.

5.3.1 The DataType Property

The DataTemplate class has a DataType property. When you set this property to the data type without specifying an x:Key, the DataTemplate gets applied automatically to data objects of that type. Note that when you do that the x:Key is set implicitly. Therefore, if you assign this DataTemplate an x:Key value, you are overriding the implicit x:Key and the DataTemplate would not be applied automatically.

Therefore, instead of specifying an x:Key for the DataTemplate in the above example, you can do the following:

```
<DataTemplate DataType="{x:Type local:Task}">
  <StackPanel>
    <TextBlock Text="{Binding Path=TaskName}" />
    <TextBlock Text="{Binding Path=Description}"/>
    <TextBlock Text="{Binding Path=Priority}"/>
  </StackPanel>
</DataTemplate>
```

Now it is no longer needed to specify an ItemTemplate on the ListBox.

5.4 Using a DataTemplate in a ContentControl

Using a DataTemplate for a ContentControl does not seem very interesting at first sight. Why should you use a DataTemplate, when you can already do so much with Content property and data binding.

It gets interesting when you want to make the way the data is displayed more dynamic. A good use case is setting up an MVVM pattern in your application. The main Window typically has to show different views as the user is navigating the application.
Let’s assume that the main Window displays a CustomersView UserControl like this:

```
<view:CustomersView Grid.Row="2" DataContext="{Binding SelectedViewModel}"/>
```

The same functionality can be achieved by using a ContentControl tag and setting the ContentTemplate:

```
<ContentControl Grid.Row="2" Content="{Binding SelectedViewModel}">
  <ContentControl.ContentTemplate>
    <DataTemplate>
      <view:CustomersView/>
    </DataTemplate>
  </ContentControl.ContentTemplate>
</ContentControl>
```

To make the view that is being rendered dynamic. A data template can be defined as a resource for each view.

```
<Window.Resources>
  <DataTemplate DataType="{x:Type viewModel:CustomersViewModel}">
    <view:CustomersView/>
  </DataTemplate>
  <DataTemplate DataType="{x:Type viewModel:ProductsViewModel}">
    <view:ProductsView/>
  </DataTemplate>
</Window.Resources>
```

```
<ContentControl Grid.Row="2" Content="{Binding SelectedViewModel}"/>
```

Notice that the DataTemplate resources don’t have an explicit key set. Instead the DataType is set. Now the template being used depends on the type of the Content object which is a certain ViewModel type.