Introduction to Control Templates

We first introduced the concept of templates in *'D' is for DataTemplate*. At that time, we learned that a template is just a tree of visual elements (along with some resources and triggers) used to define the look and feel of a member of the logical tree. As it builds the logical tree, the framework watches for controls and data items that have corresponding templates. When such an element is encountered, the appropriate template is "inflated" into the actual visuals that represent the logical item and those visuals are inserted into the visual tree.

We've already learned that a DataTemplate is used to declare the visual representation of a data item that appears within an application's logical tree. In *'P' is for Panel*, we learned that an ItemsPanelTemplate is used to declare the items host used within an ItemsControl. (Note that an ItemsPanelTemplate is a special template whose visual tree must consist of only a single Panel element.) In this section, we will look at a third type of template... a*ControlTemplate*.

As the name implies, a ControlTemplate is used to declare the visual representation for a control. All native WPF controls have default control templates for each Windows theme. These templates give controls their default visual appearance.

It should come as no surprise that the visual representation of a control like a Button is composed of several other WPF elements, including a ContentPresenter and an element to render the outer and inner lines that give the Button a 3-dimensional appearance. These subelements which make up the control's appearance are part of its ControlTemplate.

The nice thing about the lookless control model is that we have full control over the visual appearance of all WPF controls. We can either accept a control's default appearance or we can change the control's appearance by defining our own ControlTemplate. Below is an example of a ControlTemplate for a ListBox:

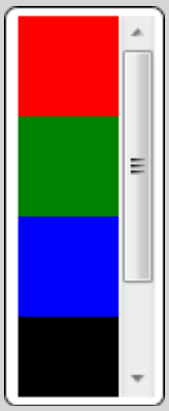
```
<ControlTemplate x:Key="MyListBoxTemplate" TargetType="{x:Type ListBox}">
  <Border Background="White" BorderBrush="Black"
      BorderThickness="1" CornerRadius="6">
    <ScrollViewer Margin="4">
      <ItemsPresenter />
    </ScrollViewer>
  </Border>
</ControlTemplate>
```

This template defines a visual tree for the ListBox that consists of a Border, a ScrollViewer, and an ItemsPresenter. The logic behind the ListBox is still defined within the ListBox class, but the appearance is defined by us.

Note that similar to a style, the ControlTemplate class exposes a TargetType property. This property *must* always be set. If you ever notice your template is not being applied as expected, be sure to verify that the TargetType matches the control type.

Now to apply our custom template to a ListBox, we simply set the Template property on the ListBox, as shown here:

```
<ListBox Width="80" Height="200"
    Template="{StaticResource MyListBoxTemplate}">
  <Rectangle Width="50" Height="50" Fill="Red" />
  <Rectangle Width="50" Height="50" Fill="Green" />
  <Rectangle Width="50" Height="50" Fill="Blue" />
  <Rectangle Width="50" Height="50" Fill="Black" />
</ListBox>
```



Setting the Template in a Style

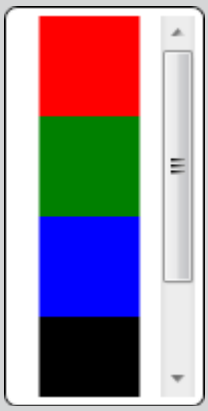
In the previous example, we assigned a ControlTemplate to a ListBox by setting the Template property directly on the ListBox. This is not the most common scenario, though. Styles and templates were designed to go hand-in-hand. As such, templates are very often included as part of a style.

Remember that a style consists of a collection of Setter objects. For a control style, one of these Setter objects usually sets the Template property on the control, as shown in the following ListBox style:

```
<Style TargetType="{x:Type ListBox}">
  <Setter Property="Background" Value="LightGray" />
  <Setter Property="BorderThickness" Value="2" />
  <Setter Property="HorizontalContentAlignment" Value="Center" />
  <Setter Property="Padding" Value="5" />
  <Setter Property="Template">
    <Setter.Value>
      <ControlTemplate TargetType="{x:Type ListBox}">
        <Border Background="White" BorderBrush="Black"
            BorderThickness="1" CornerRadius="6">
          <ScrollViewer Margin="4">
            <ItemsPresenter />
          </ScrollViewer>
        </Border>
      </ControlTemplate>
    </Setter.Value>
  </Setter>
</Style>
```

Suppose we add the above style to our application's resources. Note that this style is using the exact same control template as we presented earlier. And since we've only set the TargetType on this style, all ListBox elements in our application will implicitly use our control template. So now we can add the following ListBox to our UI:

```
<ListBox Width="100" Height="200">
  <Rectangle Width="50" Height="50" Fill="Red" />
  <Rectangle Width="50" Height="50" Fill="Green" />
  <Rectangle Width="50" Height="50" Fill="Blue" />
  <Rectangle Width="50" Height="50" Fill="Black" />
</ListBox>
```

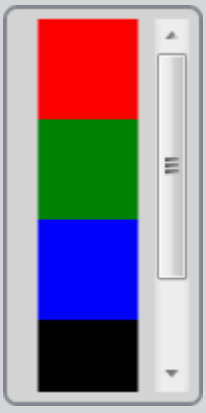


That looks pretty good, don't you think? Well, upon closer inspection of our style, you might notice that a few things are not quite right. (Hint: Look at the first few Setter objects in the style.)

Clearly, we want our template and style to play nicely together. If we use a Setter in the style to set a property, we want the template to leverage that property. The problem in our example is that several property values (Background, BorderBrush, BorderThickness, Margin, etc) are hard coded on elements in our template. Other properties of the control are simply not bound to anything in the template. This clearly isn't what we want. To fix these issues, we need to use something called a *TemplateBinding* within the template.

A TemplateBinding is a lightweight binding object used to link the value of a property on an element in the control template to the value of a property on the actual control that is being templated. In our ListBox example, we can add a few such bindings, as shown below, to achieve the desired result:

```
<Style TargetType="{x:Type ListBox}">
  <Setter Property="Background" Value="LightGray" />
  <Setter Property="BorderThickness" Value="2" />
  <Setter Property="HorizontalContentAlignment" Value="Center"
/>
  <Setter Property="Padding" Value="5" />
  <Setter Property="Template">
    <Setter.Value>
      <ControlTemplate TargetType="{x:Type ListBox}">
        <Border Background="{TemplateBinding Background}"
            BorderBrush="{TemplateBinding BorderBrush}"
            BorderThickness="{TemplateBinding
BorderThickness}"
            CornerRadius="6">
          <ScrollViewer Margin="{TemplateBinding Padding}">
            <ItemsPresenter />
          </ScrollViewer>
        </Border>
      </ControlTemplate>
    </Setter.Value>
  </Setter>
</Style>
```



Now all of the properties being set within the style are connected to elements within the template. Thus, those properties are truly reflected in the control's visual representation.

Introduction to Triggers

So far we've seen how to provide the *look* for a lookless control. Now let's peek at the other half of the user experience… the *feel*. A control gets its *look* from the visuals. It gets its *feel* from how those visuals respond to state changes or user interactions. This is where triggers come into play.

A trigger is a collection of Setter objects (or animation actions) that get applied only when a given condition is true. WPF contains three main types of triggers: Trigger (also referred to as a *property trigger*), DataTrigger, and EventTrigger. It's also possible to respond to multiple simultaneous conditions by using the related MultiTrigger and MultiDataTrigger objects.

There is a Triggers collection within each of the Style, ControlTemplate, and DataTemplate classes. This means that triggers can be used in both styles and templates, by adding them to the appropriate Triggers collection. (There is also a Triggers collection on FrameworkElement, but it can only contain event triggers… not property or data triggers.)

Let's take a look at a very simple example. Here is a style for a Button:

```
<Style TargetType="{x:Type Button}">
  <Style.Triggers>
    <Trigger Property="IsMouseOver" Value="True">
      <Setter Property="Opacity" Value="0.7" />
      <Setter Property="TextBlock.FontWeight" Value="Bold" />
    </Trigger>
  </Style.Triggers>
</Style>
```

As you might expect, moving the mouse over a Button that is using this style causes the opacity of the Button to change to 0.7. It also changes the inherited FontWeight value for any contained TextBlock elements to Bold.

As the example above illustrates, triggers are fairly intuitive. You can pretty much understand how they work just by reading the markup. As such, we won't spend too much time explaining their usage. Instead, we will just look at some tips and tricks for leveraging triggers in different scenarios.

A data trigger allows you to trigger off of a property on your data item by leveraging a Binding. This is most often used in the context of a DataTemplate, but don't overlook the power of data triggers in other scenarios. For example, a data trigger is very handy for triggering off of a property on another object, as shown in this example:

```
<DataTemplate x:Key="MyItemTemplate">
  <Border x:Name="root" BorderThickness="2" CornerRadius="6">
    <TextBlock Margin="4" Text="{Binding XPath=@First}" />
  </Border>
  <DataTemplate.Triggers>
    <DataTrigger Value="True" Binding="{Binding Path=IsSelected,
        RelativeSource={RelativeSource AncestorType={x:Type ListBoxItem}}}">
      <Setter TargetName="root" Property="BorderBrush" Value="Pink" />
    </DataTrigger>
```

```
  </DataTemplate.Triggers>
</DataTemplate>
```

In this case, we are creating a DataTrigger that acts more like a property trigger by using a FindAncestor binding to get to a source object. Namely, we are triggering off of the IsSelected property on an ancestor ListBoxItem. This will allow us to draw a pink border around our template whenever it is selected. Now suppose our data item is a person and we only want a pink border when decorating a female. If the person is male, we instead want a blue border. We could achieve this by adding a second trigger, as shown in the following template:

```
<DataTemplate x:Key="MyItemTemplate">
  <Border x:Name="root" BorderThickness="2" CornerRadius="6">
    <TextBlock Margin="4" Text="{Binding XPath=@First}" />
  </Border>
  <DataTemplate.Triggers>
    <DataTrigger Value="True" Binding="{Binding Path=IsSelected,
        RelativeSource={RelativeSource AncestorType={x:Type ListBoxItem}}}">
      <Setter TargetName="root" Property="BorderBrush" Value="Pink" />
    </DataTrigger>
    <MultiDataTrigger>
      <MultiDataTrigger.Conditions>
        <Condition Binding="{Binding XPath=@Gender}" Value="Male" />
        <Condition Value="True" Binding="{Binding Path=IsSelected,
            RelativeSource={RelativeSource AncestorType={x:Type
ListBoxItem}}}" />
      </MultiDataTrigger.Conditions>
      <Setter TargetName="root" Property="BorderBrush" Value="Blue" />
    </MultiDataTrigger>
  </DataTemplate.Triggers>
</DataTemplate>
```

Now we have added a MultiDataTrigger with two conditions. The first contains a Binding to a property on the data item that identifies the gender of the person. The second is the same binding we used earlier to get at the IsSelected property on the ancestor ListBoxItem. Now if the person is male and they are selected, the border will be blue.

The above example illustrates another important aspect of triggers and setters… namely that the *order* of triggers and setters is important. When multiple triggers evaluate to true, all of their setters are applied. More specifically, they are applied sequentially in the order that the triggers appear within the Triggers collection and in the order that the setters appear within each trigger. Here's a simple way to think about this… if multiple setters target the same property, the last setter wins!

The final thing I will point out about this example is that it demonstrates how to combine both property triggers and data triggers by leveraging a MultiDataTrigger. At times, this can come in very handy. People tend to forget that a DataTrigger can always be used as a property trigger simply by setting the source of the binding appropriately.

There is one more type of trigger called an EventTrigger. An event trigger, as the name implies, can be used to start an action in response to an event. More specifically, an event trigger executes in response to a *routed event*. Here's a kaxample of a Button that uses an event trigger to begin an opacity animation when the Button is first loaded: