WPF User Interface Customization Workshop – Design your own controls

# Summary

This workshop shows you how to customize the style of your user interface (UI).

You will learn how to use basic shapes to change the look and feel of a standard control like a button. This will enable you to create any type of control you want to while using existing functionality, like click events.

# Content

* Create a Visual Studio Solution and Project
* Add controls to your UI
* Change the control style by changing basic attributes, like colors and borders
* Create control templates from scratch
* Add interaction functionality
* Use styles to apply your design to all controls
* Create nifty Animations

# Language

XAML

# Prerequisites

* Visual Studio Community

<https://www.visualstudio.com/products/visual-studio-community-vs>

* .NET Framework 4.0/4.5

Note: The .NET Framework is installed with Visual Studio

# Author

Mertsch - <https://github.com/Mertsch>

# Get ready for some coding

## Goal

Set up a Visual Studio project to work with.

## Instructions

Start Visual Studio and click "New project", select "WPF Application" and give it a name like "CustomUI".

Press F6 to compile (build) your code. Press F5 to compile (build) and run your code in debug mode.

Run and close your (empty) application.

## Explanation

In Visual Studio a project usually represents a collection of code files which will then be compiled into executable code. A number of different project form a solution. In this case we will need only one project, which will automatically be put under a solution. The WPF Application represents a template which will pre-set a few things for you. For example the output is set to "Windows application" which will let you execute the result, as you know for any other Windows application. Also a few code files are already put into the project to get you started with the basics.

# Setting up a test environment

## Goal

Change the look of your main window and add a very basic layout with two buttons.

## Instructions

Change the MainWindow.xaml code as follows

<Window x:Class="CustomUI.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

Title="MainWindow"

**Height="150" Width="350" Background="Black"**>

<Grid>

**<Grid.ColumnDefinitions>**

**<ColumnDefinition/>**

**<ColumnDefinition/>**

**</Grid.ColumnDefinitions>**

**<Button Grid.Column="0" HorizontalAlignment="Center" VerticalAlignment="Center"**

**Content="Hello World!" Click="Button\_Click">**

**</Button>**

**<Button Grid.Column="1" HorizontalAlignment="Center" VerticalAlignment="Center"**

**Content="Hello World!" Click="Button\_Click">**

**</Button>**

</Grid>

</Window>

Add the click handler in the code behind MainWindow.xaml.cs

private void Button\_Click(object sender, RoutedEventArgs e)

{

MessageBox.Show("Click");

}

## Explanation

The simple two button layout helps us understand the effect of certain changes we will be making to the XAML code now. The click handler lets us verify that the button is still working.

# Let's customize a button

## Goal

Change the look of the first button.

## Instructions

Set a few basic properties of the first button and examine the results. Use the zoom feature of the designer, to see the effects in detail.

<Button Grid.Column="0" HorizontalAlignment="Center" VerticalAlignment="Center"

Content="Hello World!" Click="Button\_Click"

**Background="Black" BorderBrush="CornflowerBlue" BorderThickness="2"**

**Foreground="White" FontSize="20">**

</Button>

## Explanation

Changing properties of a control like a button can have various effects. It is possible that a property has no effect at all and it's possible that it will change the visual completely. What happens is up to the control designer, as a control template is used to describe how a control looks like.

When zooming in on the control, you will notice that there are certain things unaffected by the properties we set. For example the control still has round corners and an inner white border.

# Let's create our own button

## Goal

Completely change the design of a button by setting a control template.

## Instructions

Set the Template property of your first Button with a new ControlTemplate by adding the following code into your button element.

<Button.Template>

<ControlTemplate TargetType="Button">

<Border Background="{TemplateBinding Background}"

BorderThickness="{TemplateBinding BorderThickness}"

BorderBrush="{TemplateBinding BorderBrush}"

SnapsToDevicePixels="True">

<ContentPresenter Margin="5" Content="{TemplateBinding Content}"

HorizontalAlignment="{TemplateBinding HorizontalContentAlignment}"

VerticalAlignment="{TemplateBinding VerticalContentAlignment}"/>

</Border>

</ControlTemplate>

</Button.Template>

## Explanation

Every complex control in WPF has a default control template which describes the visual representation of the control. By setting the Template property of any WPF control you can override the default look and completely change it to something else. There are almost no restrictions in what you can do.

When creating a control template it is important to specify the type (TargetType) of control the template is for, as properties from the specified type can be reused within the template.

Bindings are a means of connecting properties in WPF, automatically exchanging the value of those. A TemplateBinding is a special form of Binding which binds the targeted property to another property of the control the template is for. In this case we bind a lot of properties, like the Background property, to another control (Border) used to represent our button.

Our button is now visually represented by a simple Border control. Since we still want to be able to have content, like text, within our button, we use a ContentPresenter to display any form of content set on the buttons Content property.

# Apply your cool new look to any button

## Goal

Create a style which applies the new look to any button in your application.

## Instructions

Create a new Style element as resource of the Grid containing the buttons. Move all the values set on the first button into the style.

<Grid.Resources>

<Style TargetType="Button">

<Setter Property="Background" Value="Black"/>

<Setter Property="Foreground" Value="White"/>

<Setter Property="BorderThickness" Value="2"/>

<Setter Property="BorderBrush" Value="CornflowerBlue"/>

<Setter Property="FontSize" Value="20"/>

<Setter Property="Template">

<Setter.Value>

<ControlTemplate TargetType="Button">

<Border Background="{TemplateBinding Background}"

BorderThickness="{TemplateBinding BorderThickness}"

BorderBrush="{TemplateBinding BorderBrush}"

SnapsToDevicePixels="True">

<ContentPresenter Margin="5" Content="{TemplateBinding Content}"

HorizontalAlignment="{TemplateBinding HorizontalContentAlignment}"

VerticalAlignment="{TemplateBinding VerticalContentAlignment}"/>

</Border>

</ControlTemplate>

</Setter.Value>

</Setter>

</Style>

</Grid.Resources>

## Explanation

Styles are frequently used in WPF to set properties for a number of controls. Almost every control has a default style which is setting the default look of the control. By defining a Style for the TargetType Button, we can override the default button style and apply any properties we want. Note that the second button in our application also changed its looks.

Another cool thing about styling controls is that you can override certain values whenever needed. For example when you need an orange button in one place, you can simply set the Background property to orange and override the default black background color from our style.

# OK, but my button does not behave like a button

## Goal

Give visual feedback when the user is interacting with the button.

## Instructions

Add some Trigger to the ControlTemplate which enable you to react to IsMouseOver and IsPressed properties.

<ControlTemplate.Triggers>

<Trigger Property="IsMouseOver" Value="true">

<Setter Property="Background" Value="CornflowerBlue"/>

</Trigger>

<Trigger Property="IsPressed" Value="true">

<Setter Property="Foreground" Value="Black"/>

</Trigger>

</ControlTemplate.Triggers>

## Explanation

Trigger elements allow you to specify actions which should be executed when the trigger condition is met. The trigger actions are very similar to the style setters we used before.

In this case we use the two properties of Button, IsMouseOver and IsPressed which are set to true when the mouse is over the button and when the button is pressed. We then set other properties to whatever value we see fit. This will give some visual feedback to the user, letting him or her know that there is something they can do.

# Smooth animations make it look even more awesome

## Goal

Smooth out instant changes to the visual, like colors, by using animations.

## Instructions

Give the Border inside the ControlTemplate a name and replace the trigger to IsMouseOver with a more advanced trigger containing an animation.

...

<Border **x:Name="border"** Background="{TemplateBinding Background}"

...

<ControlTemplate.Triggers>

**<EventTrigger RoutedEvent="MouseEnter">**

**<EventTrigger.Actions>**

**<BeginStoryboard Name="beginStoryboard">**

**<Storyboard Storyboard.TargetName="border"**

**Storyboard.TargetProperty="Background.Color">**

**<ColorAnimation To="CornflowerBlue" Duration="0:0:0.7"/>**

**</Storyboard>**

**</BeginStoryboard>**

**</EventTrigger.Actions>**

**</EventTrigger>**

**<EventTrigger RoutedEvent="MouseLeave">**

**<EventTrigger.Actions>**

**<RemoveStoryboard BeginStoryboardName="beginStoryboard"/>**

**</EventTrigger.Actions>**

**</EventTrigger>**

<Trigger Property="IsPressed" Value="True">

<Setter Property="Foreground" Value="Black"/>

</Trigger>

</ControlTemplate.Triggers>

## Explanation

By giving the Border control a name, we are able to address the control directly from within the trigger actions. Instead of a trigger reacting to the value of a property, we can also use an EventTrigger, which reacts to the given event. Here we react to the MouseEnter and MouseLeave events of the button.

Animations within triggers are wrapped with story boards. A Storyboard can contain many different animations and also defines important parameters, like the target of the animation. In this example we are using two specialized story boards. The BeginStoryboard element creates a new Storyboard when the trigger condition is met and the RemoveStoryboard element removes the animation, by removing the Storyboard.

The target of our animation is the Border named border in our control template. The value of the targets Background property has a property named Color, which we want to change during the animation. The color will be animated from the current value (Black by default) to CornflowerBlue over a time span of 700 milliseconds.

# Congratulations

You have done it! You have customized your (first) WPF control by setting various properties, replacing the control template and re-defining the user interaction. You applied animations, to have a great user experience.

# More customization

## Goal

Create even better controls.

## Instructions

Here are a few examples of what you can do:

* Set Background="Coral" on the second button and see what happens
* Use other shapes and controls in your template
* Add even more animations, to the BorderBrush for example
* Customize a different control, like a CheckBox
* …