Dotnet France
Technologies Sharepoint, SQL Server & .NET

Association Dotnet France

La Mise en Forme en WPF

Julien DOLLON

Microsoft Student Partners

Sommaire

1 Introduction

Comme vous avez surement pu le voir dans les autres cours de Dotnet-France nottament celui sur les WinForms, on se rend vite compte qu'en matière de modification d'apparence, avec les WinForms par exemple, on reste très limités. En effet avec les WinForms il n'était pas possible de modifier un contrôle au-delà des propriétés définit par le créateur de ce dernier.

En WPF tout cela est différent, les contrôles n'ont en fait pas d'apparence définit. Cela implique que seulement leur parti fonctionnel est défini et que par conséquent l'apparence de ces derniers sont entièrement modifiable à tout moment.

Comme on a pu le voir dans les chapitres précédent du cours sur WPF, en WPF le code métier de notre application est séparé du code designer, ce qui permet beaucoup plus facilement de modifier l'apparence de notre application et plus particulièrement de la déléguer à quelqu'un.

Pour ce faire, tout au long de ce chapitre, on verra deux manières différentes les modifications qu'on pourra apporter à nos contrôles en WPF. On commencera par voir qu'on peut utiliser les styles pour ce qui est de la partie dites « visuelle » de nos contrôles pour ensuite se pencher plus que les templates qui consistera plus à remplacer l'apparence d'un contrôle plutôt que de la modifier.

2 Les Styles

2.1 Pourquoi les Styles ?

Les Styles sont une nouveauté de WPF. Ils vont vous permettre d'appliquer un ensemble d'attributs prédéfinis à un ou plusieurs contrôles WPF. Si vous avez déjà utilisé des feuilles de Style CSS en ASP.NET ou en HTML simple, vous devriez comprendre très vite leur utilité.

L'avantage des Styles est de pouvoir modifier l'apparence d'un ensemble de contrôles sans à avoir à paramétrer chaque contrôle un par un : il suffit de modifier, d'ajouter ou de supprimer un attribut du Style. Enfin, généralement, nous plaçons nos styles dans un fichier Ressource ce qui nous permet de l'utiliser dans tous les contrôles de l'application.

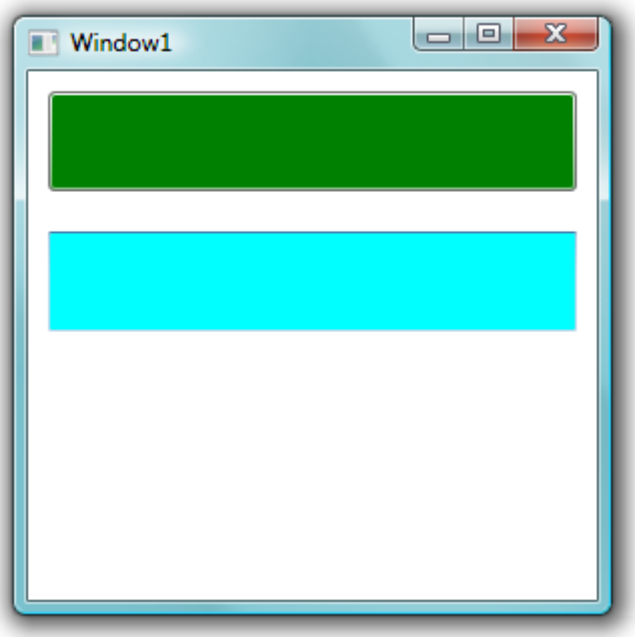
2.2 Réutilisation du code

L'intérêt majeur des Styles est dans la réutilisation du code. Cela va vous permettre de gagner beaucoup de temps dans le développement de vos IHM.

Pour illustrer cette notion, nous allons voir un exemple sur quelques utilisations des Styles :

```
<!--XAML-->
<Window
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      x:Class="WPF.Window1" x:Name="Window" Title="Window1"
      Width="300" Height="300">
    <Window.Resources>
        <Style x:Key="MonStyleBleu">
            <Setter Property="Button.Background" Value="Aqua" />
            <Setter Property="Control.Margin" Value="10" />
            <Setter Property="TextBox.Height" Value="50" />
        </Style>
        <Style x:Key="MonStyleVert" BasedOn="{StaticResource
MonStyleBleu}">
            <Setter Property="Button.Background" Value="Green" />
        </Style>
    </Window.Resources>
    <StackPanel>
        <Button Style="{StaticResource MonStyleVert}" />
        <TextBox Style="{StaticResource MonStyleBleu}" ></TextBox>
    </StackPanel>
</Window>
```

Nous avons créé dans cet exemple deux Style en tant que ressources de la fenêtre, ainsi que deux contrôles dans un StackPanel.



Regardons en détail le premier Style :

```
<!--XAML-->
<Style x:Key="MonStyleBleu">
   <Setter Property="Button.Background" Value="Aqua" />
   <Setter Property="Control.Margin" Value="10" />
   <Setter Property="TextBox.Height" Value="50" />
</Style>
```

Comme vous pouvez le voir, nous avons tout d'abord donné une clé à notre Style, cela va nous permettre de le réutiliser plus tard en donnant son nom. A l'intérieur des balises <Style> nous retrouvons des balises Setter. Ces balises vont nous permettre d'appliquer une valeur à une propriété d'un contrôle. Par exemple le premier Setter va mettre à Aqua le Background des Button. La seconde ligne va mettre à 10 la propriété Margin de tous les contrôles héritant de Control.

Note : Si une des propriétés ne peut s'appliquer à un contrôle, deux cas de figures se présentent, soit le contrôle a une propriété possédant le même nom, et dans ce cas là Setter propage la valeur correctement, soit la propriété n'existe pas, et dans ce cas là il ne se passe rien et aucune exception n'est levée.

Le second Style ressemble beaucoup au premier, à la différence que l'on y rajoute l'attribut BasedOn. Cet attribut permet tout simplement d'hériter d'un Style déjà définit et de (re)définir certaines de ses valeurs. Par exemple ici nous remplaçons la couleur Aqua de Button.Background par Green :

```
<!--XAML-->
<Style x:Key="MonStyleVert" BasedOn="{StaticResource MonStyleBleu}">
    <Setter Property="Button.Background" Value="Green" />
</Stvle>
```

Remarquez également la syntaxe de la valeur de BasedOn qui est celle que nous avions vue au chapitre 1 dans la partie consacrée aux ressources ! En effet nos Styles sont des ressources de notre page dans l'exemple présent.

Maintenant pour appliquer nos styles à nos contrôles, il suffit simplement d'utiliser l'attribut Style :

```
<!--XAML-->
<StackPanel>
    <Button Style="{StaticResource MonStyleVert}" />
    <TextBox Style="{StaticResource MonStyleBleu}" ></TextBox>
</StackPanel>
```

Le bouton utilise MonStyleVert, il devra donc avoir un fond vert, une marge de 10, et une hauteur de 50. Notre TextBox elle, aura un fond Aqua, une marge de 10 et une hauteur de 50. Comme nous appliquons des Setters sur des contrôles qu'ils ne sont pas censés toucher, il va y avoir popagation des valeurs, c'est pourquoi malgré que seuls les boutons soient censés changer de couleur, notre Textbox en change quand même.

Nous l'avons vu dans cette partie, malgré que nous indiquions vouloir appliquer un Setter à un seul type de contrôle, la propagation de la valeur se faisait tout de même. Nous allons donc voir comment forcer un style à ne s'appliquer qu'à un seul type de contrôle. En rajoutant cette protection, votre programme lèvera une exception si vous l'appliquez à un style non géré.

Pour cela, il suffit de rajouter l'attribut TargetType à vos balise Style, avec pour valeur le type du contrôle :

```
<!--XAML-->
<Style x:Key="MonStyleVert" BasedOn="{StaticResource MonStyleBleu}"
TargetType="{x:Type Button}">
    <Setter Property="Button.Background" Value="Green" />
</Style>
```

Ce Style ne s'appliquera donc plus qu'aux contrôles de type Button. Faites attention à la syntaxe lorsque vous indiquez la valeur de TargetType.

A ce stade là, vous devriez pouvoir utiliser les styles dans beaucoup de situation.

3 Les Templates

Après avoir vu comment on se servait des styles, à quoi ils servaient etc. On va maintenant voir comment on se sert des templates en WPF.

3.1 Pourquoi les Templates ?

Mais pourquoi les templates ? C'est vrai qu'après avoir étudier les styles en WPF on pourrait se demander ce qu'il nous faut de plus en ce qui concerne la personnalisation…

Et bien comme on pourra le voir dans cette partie, les styles sont loin de nous apporter autant que les templates. En effet les styles comme on l'a vu vont nous permettent de modifier l'apparence d'un ensemble de contrôles de par leurs propriétés. Alors que les templates vont totalement nous permettre de changer le modèle d'un contrôle comme le verra.

C'est d'ailleurs avec les templates que l'on va concrétement voir qu'en WPF les contrôles ont bien deux parties distinctes, la partie fonctionnelle et la partie apparence.

3.2 Modification manuel du modèle

La syntaxe pour créer les templates se rapproche de celle des styles, en effet on garde cette notion de key, de TargetType pour, je le rappel, limiter l'utilisation de notre template à un seul et unique type de contrôle.

Voici comment nous allons créer notre premier template :

```
<!--XAML-->
<Window x:Class="DFWPF.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Window1" Height="300" Width="300">

    <Window.Resources>

        <SolidColorBrush x:Key="myBorder" Color="Brown" />

        <ControlTemplate x:Key="myTemplate" TargetType="{x:Type Button}">
            <Border x:Name="border"
                    CornerRadius="100" BorderThickness="1" Padding="2"
                    Background="Aqua"
                    BorderBrush="{DynamicResource myBorder}">
                <ContentPresenter VerticalAlignment="Center"
HorizontalAlignment="Center" Content="{TemplateBinding Button.Content}"
/>
            </Border>
        </ControlTemplate>
    </Window.Resources>


    <Grid>
        <Button Name="myButton" Template="{StaticResource
myTemplate}">Vive Dotnet-France</Button>
    </Grid>
</Window>
```

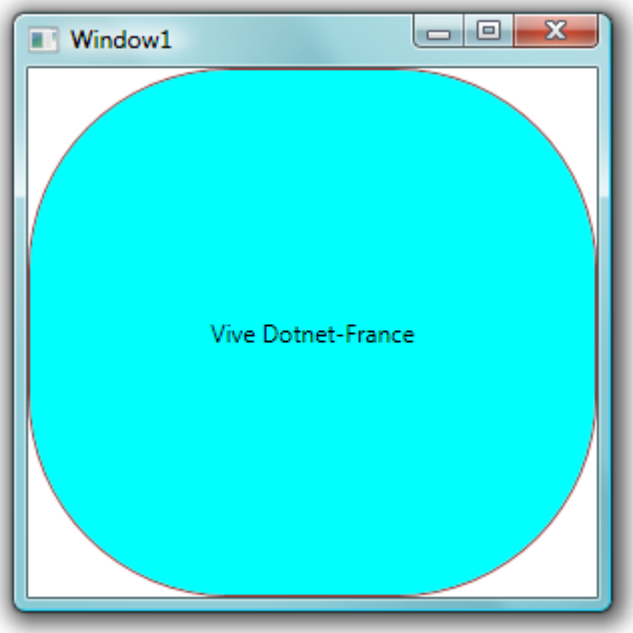
Comme vous pouvez le voir on a pris l'exemple simple c'est-à-dire celui du bouton rond avec un border et un fond.

Tout cela est rendu possible grâce à l'attribut ControlTemplate, ensuite je vous ai montré qu'on pouvait dans un template définir nos modification à la main comme pour le Background ou alors avec des ressources comme vu dans le chapitre sur le layout WPF que je vous invites à consulter si vous ne vous en souvenez plus.

Ensuite nous avons utiliser ce qu'on appel un ContentPresenter, concrètement qu'est-ce qu'un ContentPresenter ? Tout simplement si vous l'enlevez vous allez vous rendre compte que notre texte

de bouton ne sera pas affiché… En effet pour que cela fonctionne il faut Binder les propriétés de notre Bouton de base pour que cela fonctionne. On fait ça grâce à TemplateBinding comme vous pouvez l'observer.

Au final on se retrouve avec ce résultat :



3.3 Modification du modèle grâce à Expression Blend

Comme vous avez pu l'observer précédemment les templates sont très facile à mettre en place mais pas forcément très rapide. En effet vous vous doutez bien que lorsque vous devez modifier entièrement l'apparence d'un contrôle cela devient très vite fastidieux…

C'est pourquoi Microsoft met à votre disposition un logiciel nommé Expression Blend qui va vous permettre de faire tout ce travail avec un véritable outil de designer.

Pour vous présenter comment faire on va revenir à un boutton de base :

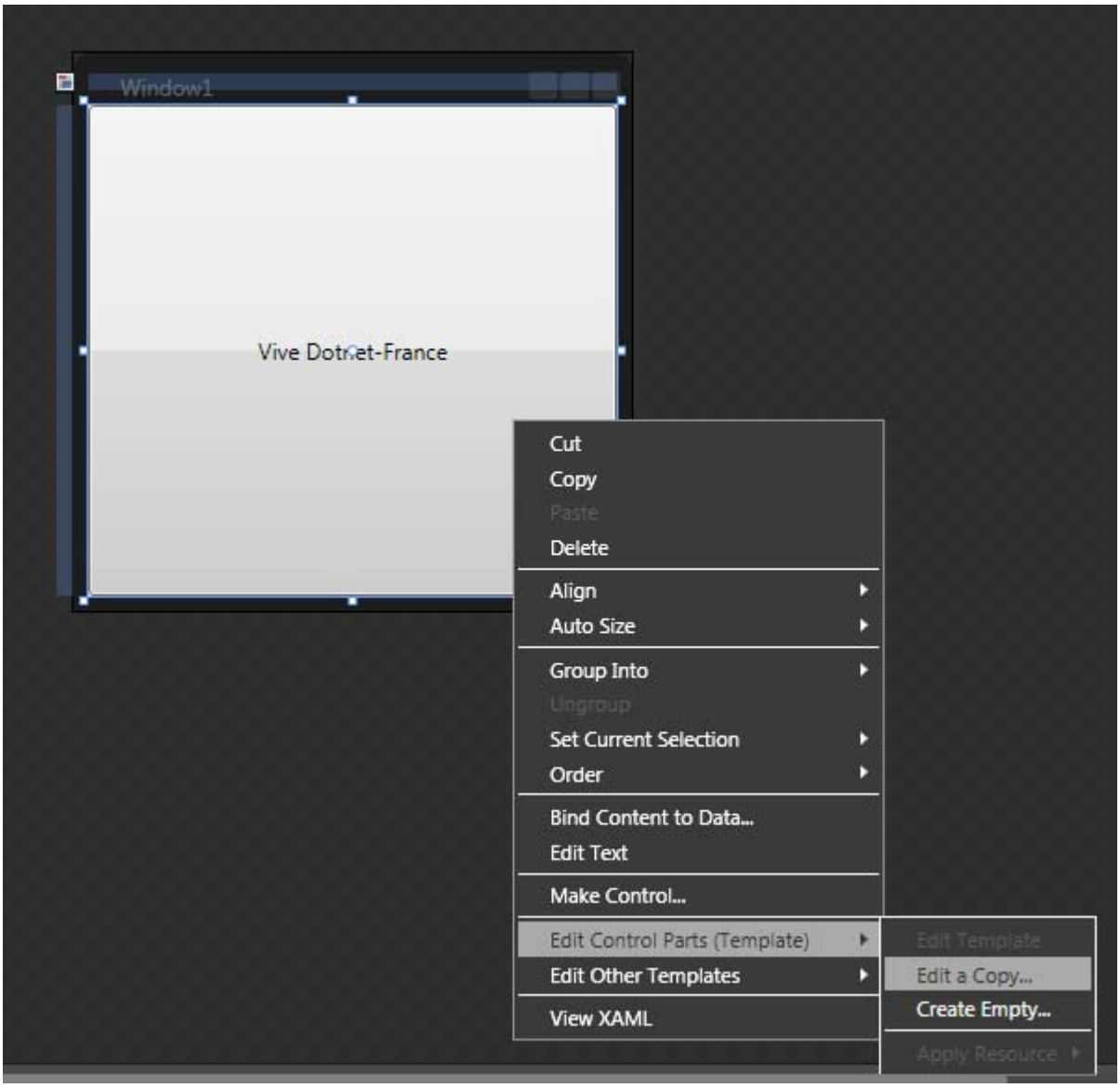
```
<!--XAML-->
<Window x:Class="DFWPF.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Window1" Height="300" Width="300">

    <Grid>
        <Button Name="myButton">Vive Dotnet-France</Button>
    </Grid>
</Window>
```

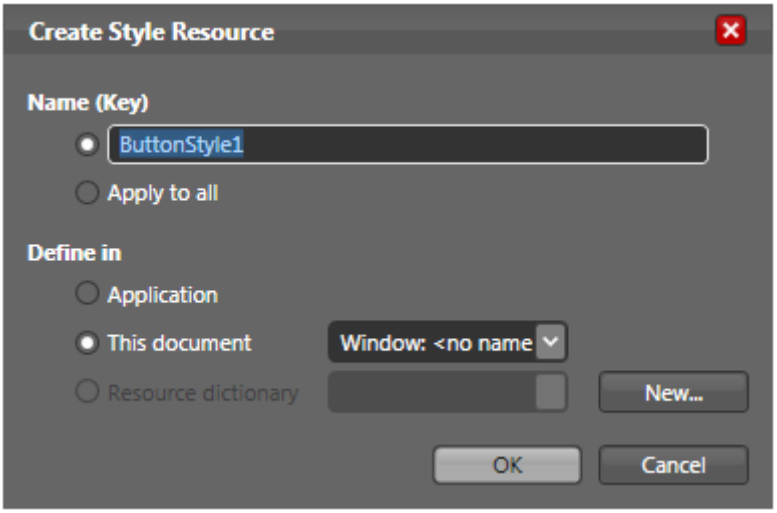
Ensuite je vous invites à ouvrir votre solution (fichier .sln) avec Expression Blend, vous vous rendez donc compte qu'a ce niveau là expression blend gère les projet visual studio, et encore mieux, le même projet peut être utilisé par les deux softwares en même temps, cela n'est bien sûr pas conseillé.

Une fois dans l'interface de Expression Blend, nous allons changer le curseur de la souris lorsque nous allons passer sur le boutton en mettant un curseur dit « d'attente ».

Pour ce faire, nous allons déjà créer le template du Boutton sous Blend :

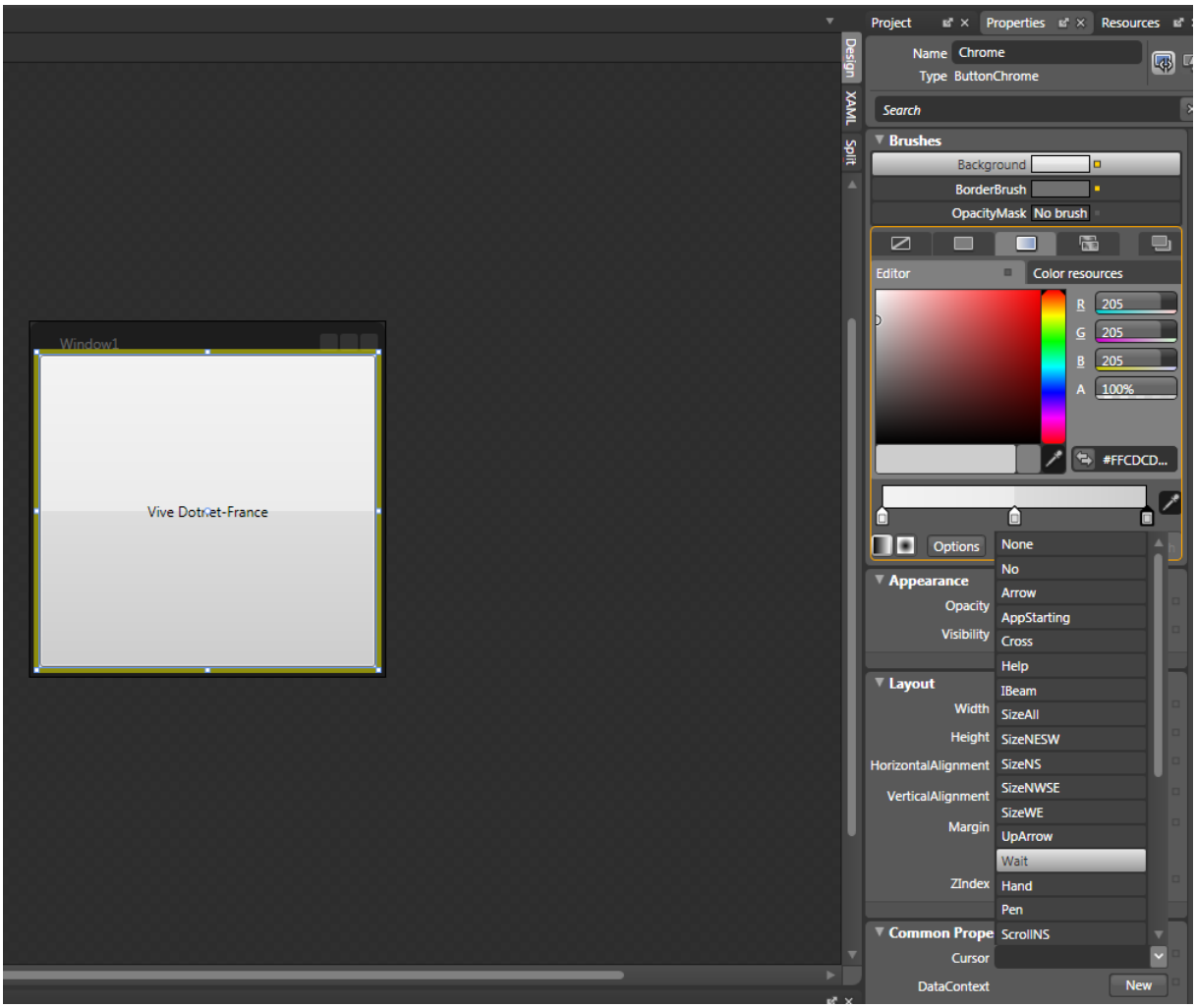


Vous arrivez ensuite sur la fenêtre suivante :



Validez après avoir choisis le nom de votre Key.

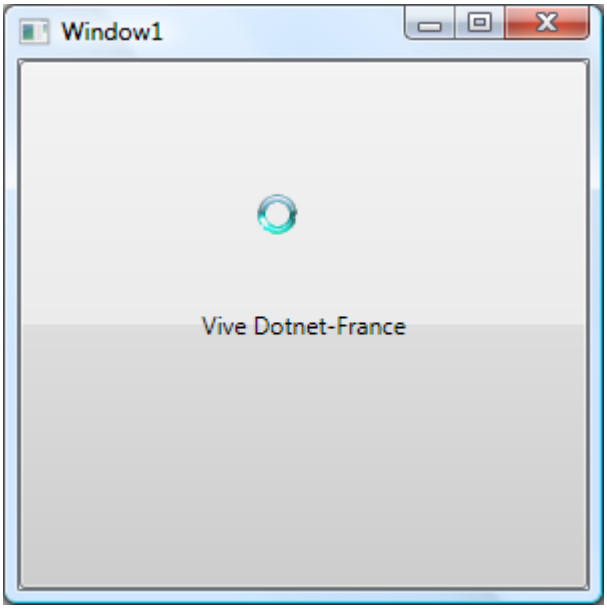
Une fois que c'est fait nous allons aller dans les propriétés de ce template pour changer le type de curseur :



Une fois que cela est fait, sauvegardez votre projet sous Blend et retournez dans Visual Studio pour actualiser votre designer.

Et comme vous pouvez l'observez dans votre éditeur XAML il vous a généré tout le code XAML pour permettre de faire ça… Bien évidemment ici il y a plus de structure qu'autre chose, vous vous doutez bien que si on avait voulu faire changer le curseur, ça aurait été beaucoup plus simple à la main.

C'est pourquoi il faut se souvenir qu'il vaut mieux utiliser Blend quand on travaille en colaboration avec une équipe de graphistes, ou pour refondre complêtement un template. Quoi qu'il en soit voici le résultat :



4 Les CustomControl

4.1 Qu'est ce qu'un CustomControl ?

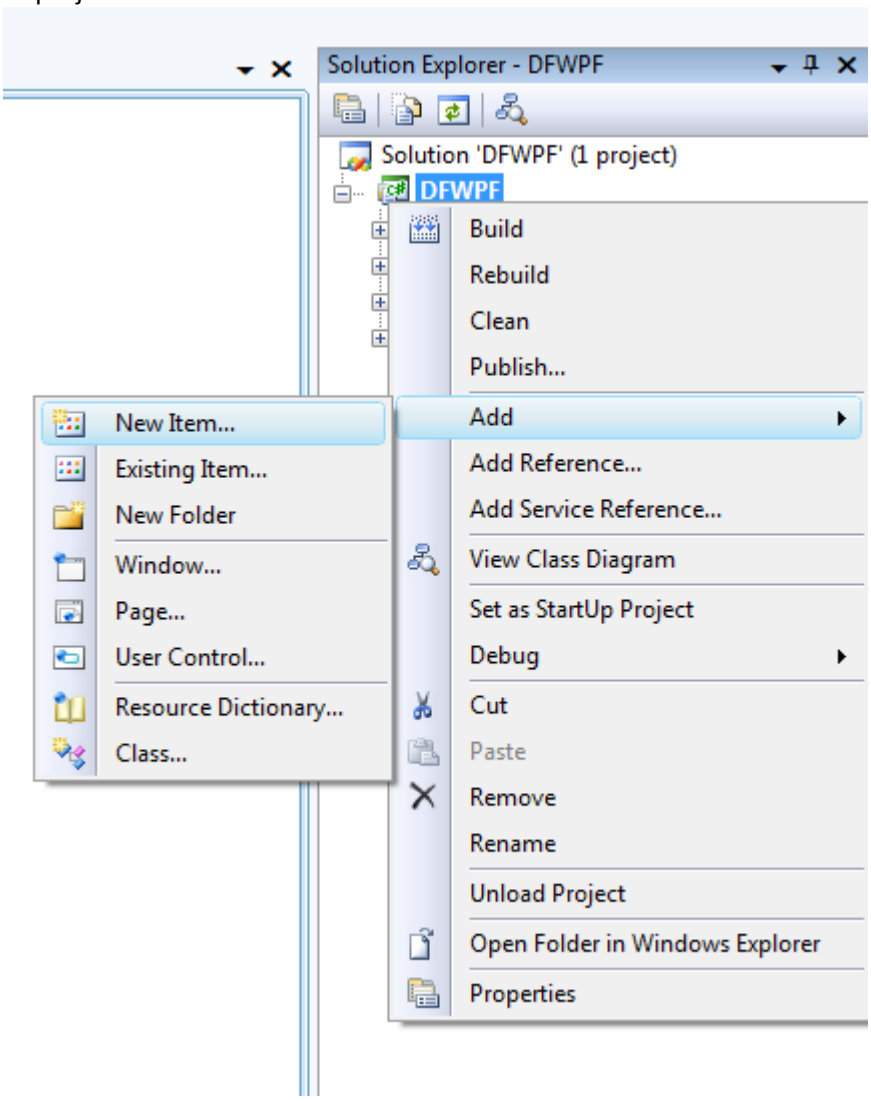
Souvenez vous dans notre chapitre précédent, on a parlé des UserControl. On a vu que les UserControl nous permettez de créer un unique contrôle à partir de plusieurs autres.

Ici nous allons pouvoir aborder la notion de CustomControl. Alors concrètement quelle est la différence ?

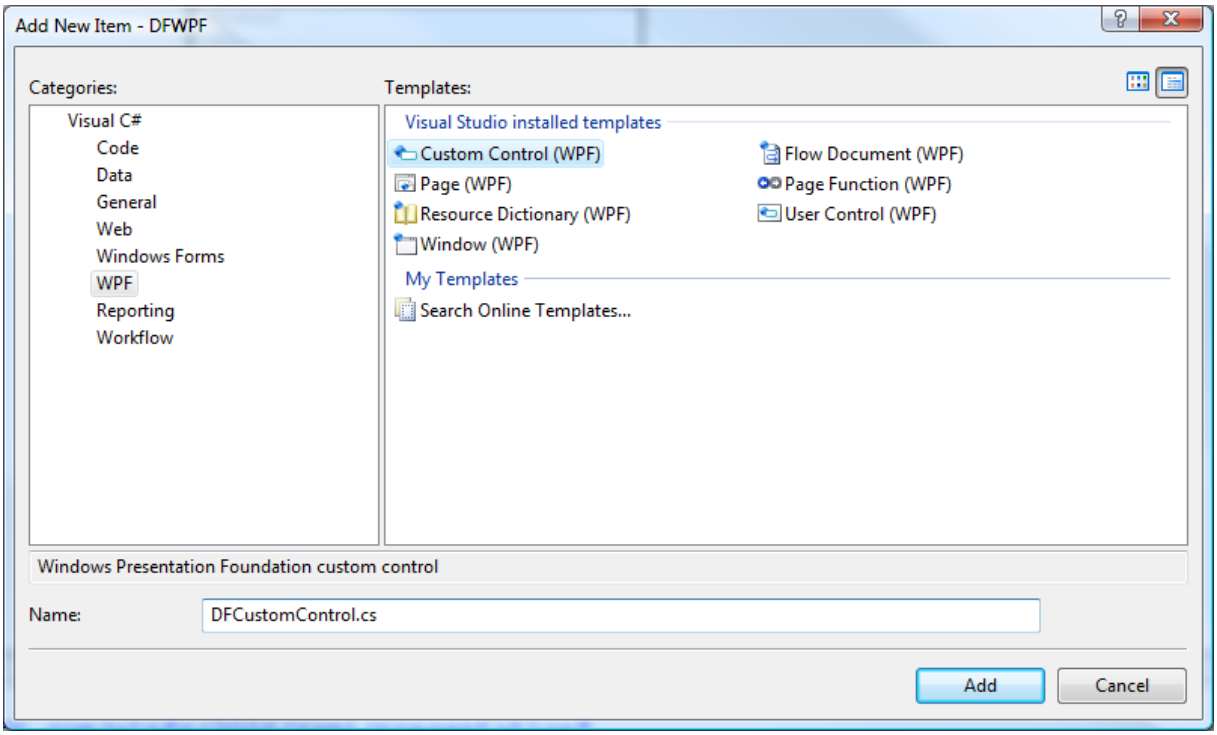
Tout simplement avec un CustomControl on va hériter d'un contrôle existant, justement pour utiliser les templates pour modifier son apparence tout en conservant ses fonctionnalités. Ce qui nous montre encore uen fois qu'en WPF l'apparence d'un contrôle et sa partie fonctionnelle sont vraiment deux choses bien distinctes.

4.2 Exemple pratique

Nous allons maintenant passer à un exemple pratique pour la création d'un CustomControl, pour ce faire nous allons créer un nouveau projet WPF. Et maintenant nous ajoutons un nouvel item au projet.



Même démarche donc que pour les UserControl, sauf qu'au lieu d'ajouter un item UserCOntrol on ajoute un item CustomControl.



Une fois que nous avons ajouter notre CustomControl, DFCustomControl, Visual Studio va nous générer deux fichiers différent, si vous avez bien suivi les cours WPF jusqu'à maintenant vous êtes surement déjà en train de vous dire qu'un fichier va contenir la logique métier et l'autre la partie apparence. Et bien tout à fait.

On va avoir un premier fichier dans le répertoire **Themes** de notre application nommé Generic.xaml qui contiendra donc tout ce qui concerne l'apparence de notre CustomControl.

Et également on va avoir un fichier DFCustomControl.cs qui contiendra la logique métier de notre contrôle.

```
// C#
public class DFCustomControl : Control
{
    static DFCustomControl()
    {
        DefaultStyleKeyProperty.OverrideMetadata(typeof(DFCustomControl),
new FrameworkPropertyMetadata(typeof(DFCustomControl)));
    }
}
```

```
' VB
Public Class DFCustomControl
    Inherits Control

    Shared Sub New()

DefaultStyleKeyProperty.OverrideMetadata(GetType(DFCustomControl), New
FrameworkPropertyMetadata(GetType(DFCustomControl)))
    End Sub

End Class
```

Comme vous pouvez le voir notre contrôle hérite pour l'instant de Control et donc n'a aucune spécificité fonctionnellement parlant.

Ici nous ne nous occuperons pas du côté fonctionnel car le but de ce chapitre est principalement de parler des styles et des templates donc nous allons dire au niveau fonctionnel que DFCustomControl hérité du contrôle simple Button.

```
// C#
public class DFCustomControl : Button
{
    static DFCustomControl()
    {
        DefaultStyleKeyProperty.OverrideMetadata(typeof(DFCustomControl),
new FrameworkPropertyMetadata(typeof(DFCustomControl)));
    }
}
```

```
' VB
Public Class DFCustomControl
    Inherits Button

    Private Function DFCustomControl() As [Static]

DefaultStyleKeyProperty.OverrideMetadata(GetType(DFCustomControl), New
FrameworkPropertyMetadata(GetType(DFCustomControl)))
    End Function

End Class
```

Maintenant pour ce qui est du style, nous allons ouvrir notre fichier Generic.xaml et changer le style de notre Button de tel façon :

```
<!--XAML-->
<ResourceDictionary
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="clr-namespace:DFWPF">

    <Style x:Key="{x:Type local:DFCustomControl}" TargetType="{x:Type
local:DFCustomControl}" BasedOn="{StaticResource {x:Type Button}}">
        <Setter Property="Button.Background" Value="Aqua" />
        <Setter Property="Button.Margin" Value="100" />
    </Style>

</ResourceDictionary>
```

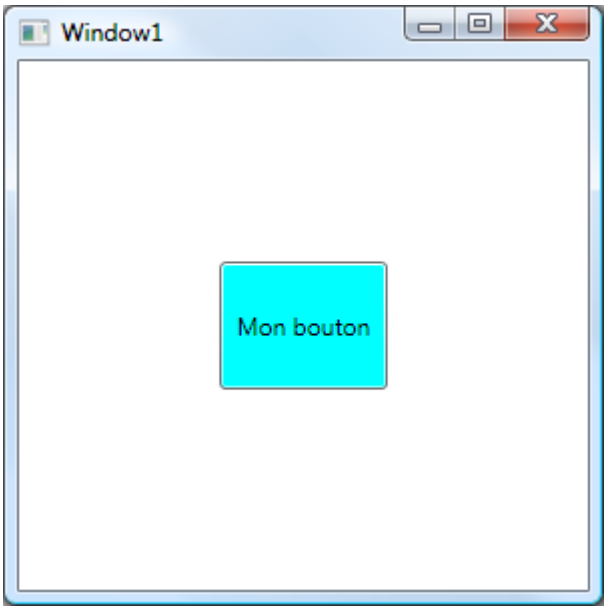
Et maintenant rajoutons à notre fenêtre notre CustomControl :

```
<!--XAML-->
<Window x:Class="DFWPF.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="clr-namespace:DFWPF"
    Title="Window1" Height="300" Width="300">
    <Grid>
        <local:DFCustomControl>Mon bouton</local:DFCustomControl>
    </Grid>
</Window>
```

Comme vous le voyez nous n'oublions pas de spécifier le namespace vers notre CustomControl grâce à cette ligne :

```
xmlns:local="clr-namespace:DFWPF"
```

Le reste est un jeux d'enfant. On obtient donc le résultat suivant :


Window1
Mon bouton

5 Les Triggers

5.1 Qu'est ce que sont les Triggers ?

Depuis le début de ce chapitre, nous avons vu comment utiliser les Styles ou les Templates, mais vous avez peut être remarqué que dans plusieurs cas nous avons été limité sur certains aspects, par exemple, nous avons définit un background à un bouton, mais celui si reprenait sa couleur normale quand nous passions la souris, ou que nous cliquions dessus.

Les Triggers vont nous permettre de surveiller les valeurs des propriétés de vos contrôles et ainsi rajouter des conditions dans vos Styles ou Templates. Généralement, les Triggers sont utilisés avec les Styles, sachez qu'il est possible de les appliquer aux Templates également.

Pour utiliser les Triggers, il faut remplir la propriété Triggers de votre Style ou de votre template avec les objets Trigger.

Note : Dans ce chapitre nous ne verrons que les propriétés Triggers et les MultiTrigger. Cependant il existe également les DataTriggers et MultiDataTrigger qui fonctionnent grâce au Binding et permettent de surveiller les changements de valeurs des propriétés du Framework .NET, d'un UserControl, ou d'une de vos classes. Nous traiterons de ces deux types de Triggers dans le chapitre consacré au Bidding.

5.2 Property Triggers

Les Property Triggers s'appliquent aux propriétés des contrôles fournies par le Framework. Ils vont nous permettre de surveiller les états du contrôle et lui appliquer le bon style en fonction.

Prenons l'exemple d'un Button dont nous voulons changer le style. Nous déterminons trois états que nous souhaitons changer, l'état standard, l'état focus et l'état survolé.

Nous allons utiliser dans notre cas deux Triggers, appliqués respectivement sur les propriétés IsMouseOver qui passe à true quand la souris survole notre button, et IsFocused qui passe à true quand le button est focus.

Voici le code que nous allons utiliser :

```
<!--XAML-->
<Window
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      x:Class="WPF.Window1" x:Name="Window" Title="Window1"
      Width="300" Height="300">
    <Window.Resources>
        <Style x:Key="MonStyleVert" TargetType="{x:Type Button}">
            <Style.Triggers>
                <Trigger Property="Button.IsMouseOver" Value="True">
                    <Setter Property="Button.Background" Value="Red" />
                </Trigger>
                <Trigger Property="Button.IsFocused" Value="True">
                    <Setter Property="Button.Background" Value="Blue" />
                </Trigger>
            </Style.Triggers>
            <Setter Property="Button.Background" Value="Green" />
            <Setter Property="Control.Margin" Value="10" />
            <Setter Property="TextBox.Height" Value="50" />
        </Style>
    </Window.Resources>
    <StackPanel>
        <Button Style="{StaticResource MonStyleVert}" />
    </StackPanel>
</Window>
```

Notre Style est inspiré des exemples de la partie Style de ce chapitre. Comme vous pouvez le voir nous avons encapsulé des balises <Trigger> dans la balise <Style.Triggers>. A l'intérieur de chacune des balises <Trigger> vous devrez rajouter tous les Setter correspondant à la condition que gère le Trigger.

Pour surveiller les changements de valeurs des propriétés du contrôle, il suffit d'indiquer au Trigger, grâce aux attributs Property et Value, la propriété à surveiller, et la valeur qui déclenchera le Trigger.

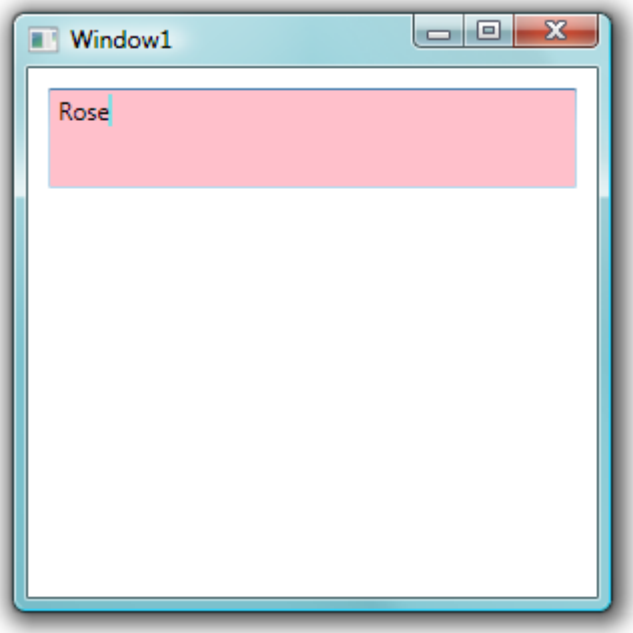
Si nous exécutons cet exemple nous voyons un bouton vert, qui devient rouge quand la souris passe dessus, et qui devient bleu si nous le prenons en focus (par exemple en cliquant dessus)

5.3 MultiTrigger

MultiTrigger permet de surveiller plusieurs propriétés d'un contrôle à la fois avant d'effectuer un changement, son utilisation ressemble fortement à celle des Property Triggers. Voici un exemple :

```
<!--XAML-->
<Window
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      x:Class="WPF.Window1" x:Name="Window" Title="Window1"
      Width="300" Height="300">
    <Window.Resources>
        <Style x:Key="MonStyleVert" TargetType="{x:Type TextBox}">
            <Style.Triggers>
                <MultiTrigger>
                    <MultiTrigger.Conditions>
                        <Condition Property="TextBox.IsFocused"
Value="True" />
                        <Condition Property="Text" Value="Rose" />
                    </MultiTrigger.Conditions>
                    <Setter Property="TextBox.Background" Value="Pink" />
                </MultiTrigger>
                <Trigger Property="TextBox.IsMouseOver" Value="True">
                    <Setter Property="TextBox.Background" Value="Red" />
                </Trigger>
            </Style.Triggers>
            <Setter Property="TextBox.Background" Value="Green" />
            <Setter Property="TextBox.Margin" Value="10" />
            <Setter Property="TextBox.Height" Value="50" />
        </Style>
    </Window.Resources>
    <StackPanel>
        <TextBox  Style="{StaticResource MonStyleVert}" />
    </StackPanel>
</Window>
```

Nous avons reprit l'exemple précédent que nous avons un petit peu modifié. Nous appliquons notre style à une TextBox, et nous voulons que si elle a le focus et que « Rose » est inscrit à l'intérieur, son background devienne de couleur rose.



Nous avons donc définit notre MultiTrigger dans la collection de Triggers, et à l'intérieur de celui-ci, nous encapsulons les conditions de déclenchement et le ou les setter associés. Pour ajouter des conditions, nous devons les encapsuler dans la balise <MultiTrigger.Conditions>. Chaque condition va surveiller une propriété comme les Triggers Traditionnels, ici nous surveillons si la textbox a le focus et si la propriété Text vaut Rose.

Si nous compilons ce projet, et que nous écrivons rose dans notre textbox, le background va devenir rose.

6 Conclusion

On a vu que les styles nous ont permis de changer complétement l'apparence visuel de nos contrôles de façon beaucoup plus radical que ce qu'il nous était proposé avec les WinForms et plus particulièrement que c'était une aide précieuse au niveau de la réutilisation de notre code.

Mais aussi on a également vu que grâce aux templates, nos contrôles pouvait présenter un comportement et un affichage totalement indépendant.

A l'heure actuelle on se rend compte que les développeurs pensent de plus en plus à la couche métier et de moins en moins à l'utilisateur final. C'est pourquoi il est peut-être temps de penser à séparer nos compétences en matière de développement et de design.

Pour cela les styles et les templates que nous propose WPF sont une des possibilités offertes pour palier à ce problème.

On pourra prolonger notre étude sur WPF en voyant que les triggers vu ici peuvent également servir au DataBinding.