eBook Gratuit

APPRENEZ xaml

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#xaml

Table des matières

À propos

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: xaml

It is an unofficial and free xaml ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official xaml.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapitre 1: Démarrer avec xaml

Remarques

E **X** tensible **A** PPLICATION **M** arkup **L** angue (XAML) est un langage de balisage basé sur XML développé par Microsoft. Il est utilisé dans plusieurs technologies Microsoft telles que Windows Presentation Foundation (WPF), Silverlight, WinRT, Universal Windows Platform, etc. pour définir l'interface utilisateur des applications.

Versions

Version	Date de sortie
WPF XAML	2006-11-21
Silverlight 3	2009-07-09
Silverlight 4	2010-04-15
Windows 8 XAML	2011-09-01

Examples

Installation ou configuration

Le moyen le plus simple pour écrire votre premier fichier XAML consiste à installer Microsoft Visual Studio. Ceci est disponible gratuitement auprès de Microsoft.

Une fois installé, vous pouvez créer un nouveau projet, de type WPF Application, avec un code VB.NET ou C #.

Ceci est similaire aux formulaires Windows dans le sens où vous avez une série de fenêtres, la principale différence étant que ces fenêtres sont écrites en XAML et sont beaucoup plus sensibles à différents périphériques.

Encore besoin d'amélioration.

Bonjour le monde

Voici un exemple simple d'une page XAML dans WPF. Il se compose d'une `Grid` , d'un `TextBlock` et d'un `Button` - les éléments les plus courants de XAML.

```
<Window x:Class="FirstWpfApplication.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

```
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        mc:Ignorable="d"
        Title="MainWindow"
        Height="350"
        Width="525">
    <Grid>
        <TextBlock Text="Welcome to XAML!"
                   FontSize="30"
                   Foreground="Black"
                   HorizontalAlignment="Center"
                   VerticalAlignment="Center"/>

        <Button Content="Hello World!"
                Background="LightGray"
                Foreground="Black"
                FontSize="25"
                Margin="0,100,0,0"
                VerticalAlignment="Center"
                HorizontalAlignment="Center"/>
    </Grid>
</Window>
```

Syntaxe	La description
`<Window>`	Le conteneur racine qui héberge le contenu qui visualise les données et permet aux utilisateurs d'interagir avec celui-ci. Une fenêtre WPF est une combinaison d'un fichier XAML (.xaml), où l'élément est la racine et d'un fichier CodeBehind (.cs).
`<Grid>`	Un panneau de disposition qui organise ses éléments enfants dans une structure tabulaire de lignes et de colonnes.
`<TextBlock>`	Fournit un contrôle léger pour afficher du texte de chaîne dans sa propriété Text ou des éléments de contenu de flux Inline, tels que Bold, Hyperlink et InlineUIContainer, dans sa propriété Inlines.
`<Button>`	Représente un contrôle de bouton qui réagit avec l'utilisateur cliquez dessus.

Propriété	La description
`Title`	Obtient ou définit le titre d'une fenêtre.
`Height`	Obtient ou définit la hauteur d'un élément.
`Width`	Obtient ou définit la largeur d'un élément.
`Text`	Obtient ou définit le contenu textuel d'un élément de texte.
`FontSize`	Obtient ou définit la taille de police de niveau supérieur pour le texte.
`Background`	Obtient ou définit la couleur du pinceau qui peint l'arrière-plan d'un élément.

Propriété	La description
`Foreground`	Obtient ou définit la couleur du pinceau qui peint la police d'un texte dans un élément.
`Margin`	Obtient ou définit la valeur qui décrit l'espace extérieur entre un élément et les autres.
`HorizontalAlignment`	Obtient ou définit les caractéristiques d'alignement horizontal appliquées à l'élément lorsqu'il est composé dans un élément parent, tel qu'un panneau ou un contrôle d'éléments.
`VerticalAlignment`	Obtient ou définit les caractéristiques d'alignement vertical appliquées à l'élément lorsqu'il est composé dans un élément parent tel qu'un panneau ou un contrôle d'éléments.

Lire Démarrer avec xaml en ligne: https://riptutorial.com/fr/xaml/topic/903/demarrer-avec-xaml

Chapitre 2: Contrôles de disposition

Examples

Toile

`Canvas` est le plus simple des panneaux. Il place les éléments aux coordonnées `Top/Left` spécifiées.

```
<Canvas>
  <TextBlock
    Canvas.Top="50"
    Canvas.Left="50"
    Text="This is located at 50, 50"/>
  <TextBlock
    Canvas.Top="100"
    Canvas.Left="50"
    Width="150"
    Height="23"
    Text="This is located at 50, 100 with height 23 and width 150"/>
</Canvas>
```

DockPanel

`DockPanel` aligne le contrôle en fonction de la propriété dock, dans l'ordre dans `DockPanel` il est placé dans le contrôle.

REMARQUE: `DockPanel` fait partie du framework WPF, mais n'est pas `DockPanel` avec Silverlight / WinRT / UWP. Les implémentations open-source sont faciles à trouver.

```
<DockPanel LastChildFill="False">
  <!-- This will strech along the top of the panel -->
  <Button DockPanel.Dock="Top" Content="Top"/>
  <!-- This will strech along the bottom of the panel -->
  <Button DockPanel.Dock="Bottom" Content="Bottom"/>
  <!-- This will strech along the remaining space in the left of the panel -->
  <Button DockPanel.Dock="Left" Content="Left"/>
  <!-- This will strech along the remaining space in the right of the panel -->
  <Button DockPanel.Dock="Right" Content="Right"/>
  <!-- Since LastChildFill is false, this will be placed at the panel's right, to the left of
the last button-->
  <Button DockPanel.Dock="Right" Content="Right"/>
</DockPanel>
```

```
<!-- When lastChildFill is true, the last control in the panel will fill the remaining space,
no matter what Dock was set to it -->
<DockPanel LastChildFill="True">
  <!-- This will strech along the top of the panel -->
  <Button DockPanel.Dock="Top" Content="Top"/>
  <!-- This will strech along the bottom of the panel -->
  <Button DockPanel.Dock="Bottom" Content="Bottom"/>
```

```
  <!-- This will strech along the remaining space in the left of the panel -->
  <Button DockPanel.Dock="Left" Content="Left"/>
  <!-- This will strech along the remaining space in the right of the panel -->
  <Button DockPanel.Dock="Right" Content="Right"/>
  <!-- Since LastChildFill is true, this will fill the remaining space-->
  <Button DockPanel.Dock="Right" Content="Fill"/>
</DockPanel>
```

StackPanel

StackPanel place ses commandes les unes après les autres. Il agit comme un panneau de quai avec toutes les stations de contrôle réglées sur la même valeur.

```
<!-- The default StackPanel is oriented vertically, so the controls will be presented in order
from top to bottom -->
<StackPanel>
  <Button Content="First"/>
  <Button Content="Second"/>
  <Button Content="Third"/>
  <Button Content="Fourth"/>
</StackPanel>
```

```
<!-- Setting the Orientation property to Horizontal will display the control in order from
left to right (or right to left, according to the FlowDirection property) -->
<StackPanel Orientation="Horizontal">
  <Button Content="First"/>
  <Button Content="Second"/>
  <Button Content="Third"/>
  <Button Content="Fourth"/>
</StackPanel>
```

Pour empiler des éléments de bas en haut, utilisez un panneau d'ancrage.

la grille

`Grid` est utilisée pour créer des dispositions de table.

Définitions de base des lignes et des colonnes

```
<Grid>
  <!-- Define 3 columns with width of 100 -->
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="100"/>
    <ColumnDefinition Width="100"/>
    <ColumnDefinition Width="100"/>
  </Grid.ColumnDefinitions>
  <!-- Define 3 rows with height of 50 -->
  <Grid.RowDefinitions>
    <RowDefinition Height="50"/>
    <RowDefinition Height="50"/>
    <RowDefinition Height="50"/>
  </Grid.RowDefinitions>
  <!-- This is placed at the top left (first row, first column) -->
```

```
  <Button
      Grid.Column="0"
      Grid.Row="0"
      Content="Top Left"/>
  <!-- This is placed at the top left (first row, second column) -->
  <Button
      Grid.Column="1"
      Grid.Row="0"
      Content="Top Center"/>
  <!-- This is placed at the center (second row, second column) -->
  <Button
      Grid.Column="1"
      Grid.Row="1"
      Content="Center"/>
  <!-- This is placed at the bottom right (third row, third column) -->
  <Button
      Grid.Column="2"
      Grid.Row="2"
      Content="Bottom Right"/>
</Grid>
```

REMARQUE: Tous les exemples suivants utilisent uniquement des colonnes, mais s'appliquent également aux lignes .

Définitions de taille automatique

Les colonnes et les lignes peuvent être définies avec "Auto" comme largeur / hauteur. La taille automatique prendra *autant d'espace que nécessaire* pour afficher son contenu, et pas plus.
Les définitions de taille automatique peuvent être utilisées avec des définitions de taille fixe.

```
<Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="Auto"/>
    <ColumnDefinition Width="Auto"/>
    <ColumnDefinition Width="50"/>
  </Grid.ColumnDefinitions>
  <!-- This column won't take much space -->
  <Button Grid.Column="0" Content="Small"/>
  <!-- This column will take much more space -->
  <Button Grid.Column="1" Content="This text will be very long."/>
  <!-- This column will take exactly 50 px -->
  <Button Grid.Column="2" Content="This text will be cut"/>
</Grid>
```

Définitions simples en taille d'étoile

Les colonnes et les lignes peuvent être définies avec `*` comme largeur / hauteur. Les lignes / colonnes de la taille d'une étoile prendront *autant d'espace que possible* , quel que soit son contenu.
Les définitions de taille d'étoile peuvent être utilisées avec des définitions de taille fixe et automatique. La taille de l'étoile est la valeur par défaut et la largeur de la colonne ou la hauteur

de la ligne peuvent être omises.

```
<Grid>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*"/>
        <ColumnDefinition Width="50"/>
    </Grid.ColumnDefinitions>
    <!-- This column will be as wide as it can -->
    <Button Grid.Column="0" Content="Small"/>
    <!-- This column will take exactly 50 px -->
    <Button Grid.Column="2" Content="This text will be cut"/>
</Grid>
```

Définitions proportionnelles en étoile

Outre le fait que l'étoile occupe autant d'espace que possible, les définitions des étoiles sont également proportionnelles. Si rien d'autre n'est mentionné, chaque définition d'étoile prendra autant d'espace que les autres dans la grille actuelle.

Cependant, il est possible de définir un ratio entre les tailles des différentes définitions en y ajoutant simplement un multiplicateur. Ainsi, une colonne définie comme `2*` sera deux fois plus large qu'une colonne définie comme `*` . La largeur d'une seule unité est calculée en divisant l'espace disponible par la somme des multiplicateurs (s'il n'y en a pas, il est compté comme 1). Ainsi, une grille avec 3 colonnes définies comme `*` , `2*` , `*` sera présentée comme 1/4, 1/2, 1/4. Et une avec 2 colonnes définies comme `2*` , `3*` sera présentée 2/5, 3/5.

S'il y a des définitions automatiques ou fixes dans le jeu, celles-ci seront calculées en premier et les définitions d'étoiles prendront l'espace restant après cela.

```
<Grid>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*"/>
        <ColumnDefinition Width="2*"/>
        <ColumnDefinition Width="*"/>
    </Grid.ColumnDefinitions>
    <!-- This column will be as wide as the third column -->
    <Button Grid.Column="0" Content="Small"/>
    <!-- This column will be twice as wide as the rest -->
    <Button Grid.Column="1" Content="This text will be very long."/>
    <!-- This column will be as wide as the first column -->
    <Button Grid.Column="2" Content="This text will may be cut"/>
</Grid>
```

Colonne / Rangée

Il est possible d'étendre un contrôle au-delà de sa cellule en définissant Row / ColumnSpan. La valeur définie est le nombre de lignes / colonnes

```
<Grid>
    <Grid.ColumnDefinitions>
```

```
        <ColumnDefinition Width="*"/>
        <ColumnDefinition Width="2*"/>
        <ColumnDefinition Width="*"/>
    </Grid.ColumnDefinitions>
    <!-- This control will streach across most of the grid -->
    <Button Grid.Column="0" Grid.ColumnSpan="2" Content="Small"/>
    <Button Grid.Column="2" Content="This text will may be cut"/>
</Grid>
```

WrapPanel

Le panneau de recouvrement agit de la même manière que le panneau de pile. Sauf quand il reconnaît que les articles dépasseront sa taille, il les enroulerait dans une nouvelle ligne / colonne, en fonction de son orientation.

Orientation horizontale

```
<WrapPanel Width="100">
    <Button Content="Button"/>
    <Button Content="Button"/>
    <Button Content="Button"/>
    <Button Content="Button"/>
    <Button Content="Button"/>
    <Button Content="Button"/>
    <Button Content="Button"/>
</WrapPanel>
```

Panneau vertical

```
<WrapPanel Height="70" Orientation="Vertical">
    <Button Content="Button"/>
    <Button Content="Button"/>
    <Button Content="Button"/>
    <Button Content="Button"/>
    <Button Content="Button"/>
    <Button Content="Button"/>
    <Button Content="Button"/>
</WrapPanel>
```

UniformGrid

Une grille uniforme placera tous ses enfants dans une grille, chaque enfant dans sa propre cellule. Toutes les cellules auront la même taille. On peut penser qu'il s'agit d'un raccourci vers une grille où toutes les définitions de lignes et de colonnes sont définies sur `*`

Lignes et colonnes par défaut

Par défaut, UniformGrid essaiera de créer un nombre égal de lignes et de colonnes. Quand une ligne deviendra longue, une nouvelle colonne sera ajoutée.

Ce code produira une grille de 3x3 avec les 2 premières lignes remplies et la dernière avec un bouton:

```
<UniformGrid>
    <Button Content="Button"/>
    <Button Content="Button"/>
    <Button Content="Button"/>
    <Button Content="Button"/>
    <Button Content="Button"/>
    <Button Content="Button"/>
    <Button Content="Button"/>
</UniformGrid>
```

Lignes / colonnes spécifiées

Vous pouvez indiquer à UniformGrid exactement le nombre de lignes et / ou de colonnes que vous souhaitez avoir.

```
<UniformGrid Columns="2" >
    <Button Content="Button"/>
    <Button Content="Button"/>
    <Button Content="Button"/>
    <Button Content="Button"/>
    <Button Content="Button"/>
    <Button Content="Button"/>
    <Button Content="Button"/>
</UniformGrid>
```

REMARQUE: dans le cas où les lignes et les colonnes sont définies et qu'il y a plus d'enfants que de cellules, les derniers enfants de la grille ne seront pas affichés.

Propriété FirstColumn

Une fois la propriété Columns définie, vous pouvez définir la propriété FirstColumn. Cette propriété entrera x cellules vides dans la première ligne avant que le premier enfant ne soit affiché. FirstColumn doit être défini sur un nombre inférieur à la propriété Columns.

Dans cet exemple, le premier bouton sera affiché dans la deuxième colonne de la première ligne:

```
<UniformGrid Columns="2" FirstColumn="1">
    <Button Content="Button"/>
    <Button Content="Button"/>
    <Button Content="Button"/>
    <Button Content="Button"/>
    <Button Content="Button"/>
    <Button Content="Button"/>
    <Button Content="Button"/>
```

```
</UniformGrid>
```

RelativePanel

`RelativePanel` a été introduit dans Windows 10 et sert principalement à prendre en charge les dispositions adaptatives, où les éléments enfants du panneau sont disposés différemment en fonction de l'espace disponible. `RelativePanel` est généralement utilisé avec les états visuels , qui permettent de changer la configuration de la disposition, de s'adapter à la taille de l'écran ou à la fenêtre, à l'orientation ou au cas d'utilisation. Les éléments enfants utilisent des propriétés attachées qui définissent où ils se trouvent par rapport au panneau et entre eux.

```
<RelativePanel
    VerticalAlignment="Stretch"
    HorizontalAlignment="Stretch">
    <Rectangle
        x:Name="rectangle1"
        RelativePanel.AlignLeftWithPanel="True"
        Width="360"
        Height="50"
        Fill="Red"/>
    <Rectangle
        x:Name="rectangle2"
        RelativePanel.Below="rectangle1"
        Width="360"
        Height="50"
        Fill="Green" />
</RelativePanel>
<VisualStateManager.VisualStateGroups>
    <VisualStateGroup>
        <VisualState>
            <VisualState.StateTriggers>
                <!--VisualState to be triggered when window width is >=720 effective pixels.--
>
                <AdaptiveTrigger
                    MinWindowWidth="720" />
            </VisualState.StateTriggers>
            <VisualState.Setters>
                <Setter
                    Target="rectangle2.(RelativePanel.Below)"
                    Value="{x:Null}" />
                <Setter
                    Target="rectangle2.(RelativePanel.RightOf)"
                    Value="rectangle1" />
                <Setter
                    Target="rectangle1.(RelativePanel.AlignLeftWithPanel)"
                    Value="False" />
                <Setter
                    Target="rectangle1.(RelativePanel.AlignVerticalCenterWithPanel)"
                    Value="True" />
                <Setter
                    Target="rectangle2.(RelativePanel.AlignVerticalCenterWithPanel)"
                    Value="True" />
            </VisualState.Setters>
        </VisualState>
    </VisualStateGroup>
</VisualStateManager.VisualStateGroups>
```

Lire Contrôles de disposition en ligne: https://riptutorial.com/fr/xaml/topic/3634/controles-de-disposition

Chapitre 3: Convertisseurs

Paramètres

Paramètre	Détails
valeur	La valeur à convertir de
targetType	Le type en cours de conversion en
paramètre	Valeur facultative pour contrôler le fonctionnement de la conversion
Culture	Objet CultureInfo - requis si la localisation est nécessaire

Remarques

La méthode `Convert` convertit la valeur de la source (généralement le modèle de vue) en cible (généralement une propriété d'un contrôle).

La méthode `ConvertBack` convertit la valeur de la cible vers la source. Il n'est nécessaire que si la liaison est `TwoWay` ou `OneWayToSource` .

Lorsqu'un `ConvertBack` n'est pas pris en charge, c'est-à-dire qu'il n'y a pas de correspondance `ConvertBack` entre la valeur de pré-conversion et la valeur de post-conversion, il est courant que la méthode `ConvertBack` renvoie `DependencyProperty.UnsetValue` . C'est une meilleure option que de lancer une exception (par exemple, `NotImplementedException`) car elle évite les erreurs d'exécution inattendues. En outre, les liaisons peuvent bénéficier de leur `FallbackValue` lorsque `DependencyProperty.UnsetValue` est renvoyé par un convertisseur.

Examples

Chaîne au convertisseur IsChecked

Dans XAML:

```
<RadioButton IsChecked="{Binding EntityValue, Mode=TwoWay,
                         Converter={StaticResource StringToIsCheckedConverter},
                         ConverterParameter=Male}"
             Content="Male"/>

<RadioButton IsChecked="{Binding EntityValue, Mode=TwoWay,
                         Converter={StaticResource StringToIsCheckedConverter},
                         ConverterParameter=Female}"
             Content="Female"/>
```

La classe C #:

```
public class StringToIsCheckedConverter : IValueConverter
{
    public object Convert(object value, Type targetType, object parameter,
System.Globalization.CultureInfo culture)
    {
        string input = (string)value;
        string test = (string)parameter;
        return input == test;
    }

    public object ConvertBack(object value, Type targetType, object parameter,
System.Globalization.CultureInfo culture)
    {
        if (value == null || !(value is bool))
        {
            return string.Empty;
        }
        if (parameter == null || !(parameter is string))
        {
            return string.Empty;
        }
        if ((bool)value)
        {
            return parameter.ToString();
        }
        else
        {
            return string.Empty;
        }
    }
}
```

Convertisseurs 101

Les contrôles XAML peuvent avoir des propriétés de dépendance pouvant être liées à des objets à partir de `DataContext` ou d'autres contrôles. Lorsque le type de l'objet lié est différent du type de l'objet `DependencyProperty` cible, un **convertisseur** peut être utilisé pour adapter un type à un autre.

Les convertisseurs sont des classes implémentant `System.Windows.Data.IValueConverter` ou `System.Windows.Data.IMultiValueConverter` ; WPF implémente certains convertisseurs prêts à l'emploi, mais les développeurs peuvent les utiliser dans des implémentations personnalisées, comme c'est souvent le cas.

Pour utiliser un convertisseur dans XAML, une instance peut être instanciée dans la section `Resources` . Pour l'exemple ci-dessous, `System.Windows.Controls.BooleanToVisibilityConverter` sera utilisé:

```
<UserControl.Resources>
    <BooleanToVisibilityConverter x:Key="BooleanToVisibilityConverter"/>
</UserControl.Resources>
```

Notez l'élément `x:Key` défini, qui est ensuite utilisé pour référencer l'instance de `BooleanToVisibilityConverter` dans la liaison:

```
<TextBlock Text="This will be hidden if property 'IsVisible' is true"
           Visibility="{Binding IsVisible,
                                Converter={StaticResource BooleanToVisibilityConverter}}"/>
```

Dans l'exemple ci-dessus, une propriété booléenne `IsVisible` est convertie en une valeur de l'énumération `System.Windows.Visibility` ; `Visibility.Visible` si vrai ou `Visibility.Collapsed` sinon.

Création et utilisation d'un convertisseur: BooleanToVisibilityConverter et InvertibleBooleanToVisibilityConverter

Pour étendre et développer l'expérience de liaison, nous avons des convertisseurs pour convertir une valeur d'un type en une autre valeur d'un autre type. Pour tirer parti des convertisseurs dans une liaison de données, vous devez d'abord créer une classe DataConverter

- `IValueConverter` **(WPF & UWP)**

ou

- `IMultiValueConverter` **(WPF)**

si vous voulez convertir plusieurs types en un seul type
Dans ce cas, nous nous concentrons sur la conversion d'une valeur `boolean True/False` en une visibilité `Visibility.Visible` et `Visibility.Collapsed` :

```
public class BooleanToVisibilityConverter : IValueConverter
{
    public object Convert(object value, Type targetType, object parameter, string language)
    {
        return (value is bool && (bool) value) ? Visibility.Visible : Visibility.Collapsed;
    }

    public object ConvertBack(object value, Type targetType, object parameter, string
language)
    {
        return (value is Visibility && (Visibility) value == Visibility.Visible);
    }
}
```

La méthode **`Convert`** est appelée à chaque fois que vous **`GET`** données à **`FROM ViewModel`** .
Le **`ConvertBack`** est appelé **`SET`** ing données **`TO`** la **`ViewModel`** pour **`BindingMode.TwoWay`** fixations.

Bien entendu, vous pouvez également utiliser les propriétés de votre convertisseur. Jetez un oeil à celui-ci:

```
public class InvertibleBooleanToVisibilityConverter : IValueConverter
{
    public bool Invert { get; set; } = false;

    public object Convert(object value, Type targetType, object parameter, string language)
    {
        return (value is bool && (bool) value != Invert) ? Visibility.Visible :
Visibility.Collapsed;
    }
```

```
    public object ConvertBack(object value, Type targetType, object parameter, string
language)
    {
        return (value is Visibility && ((Visibility) value == Visibility.Visible) != Invert);
    }
}
```

Si vous souhaitez utiliser un convertisseur dans une `Binding` , déclarez-le simplement comme une ressource dans votre page, fenêtre ou autre élément, donnez-lui une clé et fournissez les propriétés potentiellement nécessaires:

```
<Page ...
    xmlns:converters="using:MyNamespce.Converters">
<Page.Resources>
    <converters:InvertibleBooleanToVisibilityConverter
        x:Key="BooleanToVisibilityConverter"
        Invert="False" />
</Page.Resources>
```

et l'utiliser comme un `StaticResource` dans une liaison:

```
<ProgressRing
        Visibility="{Binding ShowBusyIndicator,
            Converter={StaticResource BooleanToVisibilityConverter},
            UpdateSourceTrigger=PropertyChanged,
            Mode=OneWay}" />
```

Lire Convertisseurs en ligne: https://riptutorial.com/fr/xaml/topic/2996/convertisseurs

Chapitre 4: Différences dans les différents dialectes XAML

Remarques

XAML est utilisé dans les applications Silverlight, Windows Phone, Windows RT et UWP. Partager du code ou convertir du code entre ceux-ci est parfois plus difficile que souhaitable en raison des différences subtiles entre les différents dialectes XAML. Ce sujet vise à donner un aperçu de ces différences avec une courte explication.

Examples

Liaisons de données compilées: l'extension de balisage {x: Bind}

Les databings sont essentiels pour travailler avec XAML. Le dialecte XAML pour les applications UWP fournit un type de liaison: l'extension de balisage {x: Bind}.

Travailler avec {Binding XXX} et {x: Bind XXX} est essentiellement équivalent, à la différence près que l'extension x: Bind fonctionne au moment de la compilation, ce qui permet de meilleures capacités de débogage (par exemple, des points d'arrêt) et de meilleures performances.

```
<object property="{x:Bind bindingPath}" />
```

L'extension de balisage x: Bind est uniquement disponible pour les applications UWP. En savoir plus à ce sujet dans cet article MSDN: https://msdn.microsoft.com/en-us/windows/uwp/data-binding/data-binding-in-depth .

Alternatives pour Silverlight, WPF, Windows RT: utilisez la syntaxe standard {Binding XXX}:

```
<object property="{Binding bindingPath}" />
```

Importer des espaces de noms dans XAML

La plupart du temps, vous devez importer des espaces de noms dans votre fichier XAML. La façon dont cela est fait est différente pour les différentes variantes de XAML.

Pour Windows Phone, Silverlight, WPF utilisent la syntaxe clr-namespace:

```
<Window ... xmlns:internal="clr-namespace:rootnamespace.namespace"
            xmlns:external="clr-namespace:rootnamespace.namespace;assembly=externalAssembly"
>
```

Windows RT, UWP utilise la syntaxe d'utilisation:

```
<Page ... xmlns:internal="using:rootnamespace.namespace"
            xmlns:external="using:rootnamespace.namespace;assembly=externalAssembly"
>
```

Reliure multiple

Multi Binding est une fonctionnalité exclusive pour le développement de WPF. Il permet une liaison à plusieurs valeurs à la fois (généralement utilisée avec un MultiValueConverter).

```
<TextBox>
    <TextBox.Text>
        <MultiBinding Converter="{StaticResource MyConverter}">
            <Binding Path="PropertyOne"/>
            <Binding Path="PropertyTwo"/>
        </MultiBinding>
    </TextBox.Text>
</TextBox>
```

Les plates-formes autres que WPF ne prennent pas en charge la liaison multiple. Vous devez trouver des solutions alternatives (comme déplacer le code de l'affichage et des convertisseurs vers le modèle de vue) ou utiliser des comportements tiers comme dans cet article: http://www.damirscorner.com/blog/posts/20160221-MultibindingInUniversalWindowsApps.html)

Lire Différences dans les différents dialectes XAML en ligne: https://riptutorial.com/fr/xaml/topic/4498/differences-dans-les-differents-dialectes-xaml

Chapitre 5: Liaison de données

Syntaxe

- `<TextBlock Text="{Binding Title}"/>`
- `<TextBlock Text="{Binding Path=Title}"/>`
- `<TextBlock> <TextBlock.Text> <Binding Path="Title"/> </TextBlock.Text> </TextBlock>`

Remarques

Toutes ces balises produisent le même résultat.

Examples

Chaîne de liaison à la propriété Text

Pour modifier le contenu de l'interface utilisateur en cours d'exécution, vous pouvez utiliser la `Binding` . Lorsque la propriété liée est modifiée à partir du code, elle sera affichée dans l'interface utilisateur.

```
<TextBlock Text="{Binding Title}"/>
```

Pour notifier les modifications à l'interface utilisateur, la propriété doit `INotifyPropertyChanged` événement `PropertyChanged` partir de l'interface `INotifyPropertyChanged` ou vous pouvez utiliser la `Dependency Property` .

La liaison fonctionne si la propriété "Title" se trouve dans le fichier xaml.cs ou dans la classe Datacontext du fichier `XAML` .

Le Datacontext peut être configuré directement dans le XAML

```
<Window x:Class="Application.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="clr-namespace:Application">
<Window.DataContext>
   <local:DataContextClass/>
</Window.DataContext>
```

Mise en forme des liaisons de chaînes

Lorsque vous effectuez une liaison de quelque chose, par exemple une date, vous voudrez peut-être l'afficher dans un format spécifique sans le modifier avec le code.

Pour ce faire, nous pouvons utiliser la propriété StringFormat.

Voici quelques exemples:

```
Text="{Binding Path=ReleaseDate, StringFormat=dddd dd MMMM yyyy}"
```

Cela formate ma date comme suit:

Mardi 16 août 2016

Voici un autre exemple de température.

```
Text="{Binding Path=Temp, StringFormat={}{0}°C}"
```

Ce formats à:

25 ° C

Les bases de INotifyPropertyChanged

Si vous souhaitez non seulement afficher des objets statiques, mais que votre interface utilisateur répond aux modifications apportées aux objets en corrélation, vous devez comprendre les principes de base de l'interface `INotifyPropertyChanged` .

En supposant que nous avons notre `MainWindow` définie comme

```
<Window x:Class="Application.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:vm="clr-namespace:Application.ViewModels>
    <Window.DataContext>
       <vm:MainWindowViewModel/>
    </Window.DataContext>
    <Grid>
        <TextBlock Text={Binding Path=ApplicationStateText}" />
    </Grid>
</Window>
```

Avec notre classe `MainWindowViewModel` défini comme

```
namespace Application.ViewModels
{
    public class MainWindowViewModel
    {
        private string _applicationStateText;

        public string ApplicationStateText
        {
            get { return _applicationStateText; }
            set { _applicationStateText = value; }
        }
        public MainWindowViewModel()
        {
```

```
            ApplicationStateText = "Hello World!";
        }

    }
}
```

Le TextBlock de notre application affichera le texte *Hello World en* raison de sa liaison. Si notre ApplicationStateText change pendant l'exécution, notre interface utilisateur ne sera pas informée de ce changement.
Pour implémenter cela, notre DataSource, dans ce cas notre `MainWindowViewModel` , doit implémenter l'interface `INotifyPropertyChanged` . Cela va permettre à nos `Bindings` de s'abonner à `PropertyChangedEvent` .
Tout ce que nous devons faire est d'appeler le `PropertyChangedEventHandler` chaque fois que nous changeons notre propriété `ApplicationStateText` comme ceci:

```
using System.ComponentModel;
using System.Runtime.CompilerServices;

namespace Application.ViewModels
{
    public class MainWindowViewModel : INotifyPropertyChanged
    {
        public event PropertyChangedEventHandler PropertyChanged;
        public void NotifyPropertyChanged( [CallerMemberName] string propertyName = null)
        {
            PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
        }

        private string _applicationStateText;

        public string ApplicationStateText
        {
            get { return _applicationStateText; }
            set
            {
                if (_applicationStateText != value)
                {
                    _applicationStateText = value;
                    NotifyPropertyChanged();
                }
            }
        }
        public MainWindowViewModel()
        {
            ApplicationStateText = "Hello World!";
        }
    }
}
```

et assurez-vous que notre `Binding` de `TextBlock.Text` écoute réellement un `PropertyChangedEvent` :

```
<Window x:Class="Application.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:vm="clr-namespace:Application.ViewModels">
```

```
    <Window.DataContext>
       <vm:MainWindowViewModel/>
    </Window.DataContext>
    <Grid>
        <TextBlock Text={Binding Path=ApplicationStateText,
UpdateSourceTrigger=PropertyChanged }" />
    </Grid>
</Window>
```

Liaison à une collection d'objets avec INotifyPropertyChanged et INotifyCollectionChanged

Supposons que vous ayez un `ListView` qui est censé afficher tous `User` objets `User` listés sous la propriété `Users` du `ViewModel` où les propriétés de l'objet User peuvent être mises à jour par programme.

```
<ListView ItemsSource="{Binding Path=Users}" >
    <ListView.ItemTemplate>
        <DataTemplate DataType="{x:Type models:User}">
            <StackPanel Orientation="Horizontal">
                <TextBlock Margin="5,3,15,3"
                           Text="{Binding Id, Mode=OneWay}" />
                <TextBox Width="200"
                           Text="{Binding Name, Mode=TwoWay,
UpdateSourceTrigger=PropertyChanged, Delay=450}"/>
            </StackPanel>
        </DataTemplate>
    </ListView.ItemTemplate>
</ListView>
```

Malgré pour `INotifyPropertyChanged` beeing implémenté correctement pour l'objet `User`

```
public class User : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    private int _id;
    private string _name;

    public int Id
    {
        get { return _id; }
        private set
        {
            if (_id == value) return;
            _id = value;
            NotifyPropertyChanged();
        }
    }
    public string Name
    {
        get { return _name; }
        set
        {
            if (_name == value) return;
            _name = value;
```

```
            NotifyPropertyChanged();
        }
    }

    public User(int id, string name)
    {
        Id = id;
        Name = name;
    }

    private void NotifyPropertyChanged([CallerMemberName] string propertyName = null)
    {
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
    }
}
```

et pour votre objet `ViewModel`

```
public sealed class MainWindowViewModel : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    private List<User> _users;
    public List<User> Users
    {
        get { return _users; }
        set
        {
            if (_users == value) return;
            _users = value;
            NotifyPropertyChanged();
        }
    }
    public MainWindowViewModel()
    {
        Users = new List<User> {new User(1, "John Doe"), new User(2, "Jane Doe"), new User(3,
"Foo Bar")};
    }

    private void NotifyPropertyChanged([CallerMemberName] string propertyName = null)
    {
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
    }
}
```

Votre interface utilisateur ne sera pas mise à jour si une modification est apportée à un utilisateur par programmation.

C'est simplement parce que vous avez défini INotifyPropertyChanged uniquement sur l'instance de la liste elle-même. Ce n'est que si vous réinstanciez complètement la liste si une propriété d'un élément change que votre interface utilisateur sera mise à jour.

```
// DO NOT DO THIS
User[] userCache = Users.ToArray();
Users = new List<User>(userCache);
```

Ceci est cependant très fatigant et incroyablement mauvais pour la performance.

Si vous avez une liste de 100 000 éléments affichant à la fois l'ID et le nom de l'utilisateur, il y aura 200 000 DataBindings en place, chacun devant être recréé. Cela se traduit par un décalage notable pour l'utilisateur chaque fois qu'un changement est apporté à quelque chose.

Pour résoudre en partie ce problème, vous pouvez utiliser `System.ComponentModel.ObservableCollection<T>` au lieu de `List<T>` :

```
private ObservableCollection<User> _users;
public ObservableCollection<User> Users
{
    get { return _users; }
    set
    {
        if (_users == value) return;
        _users = value;
        NotifyPropertyChanged();
    }
}
```

`ObservableCollection` nous fournit l'événement `CollectionChanged` et implémente `INotifyPropertyChanged` lui-même. Selon MSDN, l'événement se produira, "[..] lorsqu'un élément est *ajouté* , *supprimé* , *modifié* , *déplacé* ou que la *liste entière est actualisée* ".
Vous réaliserez toutefois rapidement qu'avec .NET 4.5.2 et versions antérieures, `ObservableCollection` ne `ObservableCollection` pas d'événement CollectionChanged si une propriété d'un élément de la collection change, comme indiqué ici .

À la suite de cette solution , nous pouvons simplement mettre en œuvre notre propre `TrulyObservableCollection<T>` sans `INotifyPropertyChanged` contrainte pour `T` ayant tout ce que nous devons et exposer wether `T` implémente `INotifyPropertyChanged` ou non:

```
/*
 * Original Class by Simon @StackOverflow http://stackoverflow.com/a/5256827/3766034
 * Removal of the INPC-Constraint by Jirajha @StackOverflow
 * according to to suggestion of nikeee @StackOverflow
http://stackoverflow.com/a/10718451/3766034
 */
public sealed class TrulyObservableCollection<T> : ObservableCollection<T>
{
    private readonly bool _inpcHookup;
    public bool NotifyPropertyChangedHookup => _inpcHookup;

    public TrulyObservableCollection()
    {
        CollectionChanged += TrulyObservableCollectionChanged;
        _inpcHookup =
typeof(INotifyPropertyChanged).GetTypeInfo().IsAssignableFrom(typeof(T));
    }
    public TrulyObservableCollection(IEnumerable<T> items) : this()
    {
        foreach (var item in items)
        {
            this.Add(item);
        }
    }
```

```
    private void TrulyObservableCollectionChanged(object sender,
NotifyCollectionChangedEventArgs e)
    {
        if (NotifyPropertyChangedHookup && e.NewItems != null && e.NewItems.Count > 0)
        {
            foreach (INotifyPropertyChanged item in e.NewItems)
            {
                item.PropertyChanged += ItemPropertyChanged;
            }
        }
        if (NotifyPropertyChangedHookup && e.OldItems != null && e.OldItems.Count > 0)
        {
            foreach (INotifyPropertyChanged item in e.OldItems)
            {
                item.PropertyChanged -= ItemPropertyChanged;
            }
        }
    }
    private void ItemPropertyChanged(object sender, PropertyChangedEventArgs e)
    {
        var args = new NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction.Replace,
sender, sender, IndexOf((T)sender));
        OnCollectionChanged(args);
    }
}
```

et définir nos `Users` propriété comme `TrulyObservableCollection<User>` dans notre ViewModel

```
private TrulyObservableCollection<string> _users;
public TrulyObservableCollection<string> Users
{
    get { return _users; }
    set
    {
        if (_users == value) return;
        _users = value;
        NotifyPropertyChanged();
    }
}
```

Notre interface utilisateur sera désormais informée d'une fois une propriété INPC d'un élément dans les modifications de la collection sans qu'il soit nécessaire de recréer chaque `Binding` .

Lire Liaison de données en ligne: https://riptutorial.com/fr/xaml/topic/3329/liaison-de-donnees

Chapitre 6: Modèles de contrôle

Examples

Modèles de contrôle

Les interfaces utilisateur par défaut pour les contrôles WPF sont généralement construites à partir d'autres contrôles et formes. Par exemple, un bouton est composé à la fois des contrôles ButtonChrome et ContentPresenter. ButtonChrome fournit l'apparence du bouton standard, tandis que ContentPresenter affiche le contenu du bouton, comme spécifié par la propriété Content. Parfois, l'apparence par défaut d'un contrôle peut ne pas correspondre à l'apparence générale d'une application. Dans ce cas, vous pouvez utiliser un ControlTemplate pour modifier l'apparence de l'interface utilisateur du contrôle sans modifier son contenu et son comportement.

XAML

```
<Window
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   x:Class="SDKSample.ControlTemplateButtonWindow"
   Title="Button with Control Template" Height="158" Width="290">

   <!-- Button using an ellipse -->
   <Button Content="Click Me!" Click="button_Click">
     <Button.Template>
       <ControlTemplate TargetType="{x:Type Button}">
         <Grid Margin="5">
           <Ellipse Stroke="DarkBlue" StrokeThickness="2">
             <Ellipse.Fill>
               <RadialGradientBrush Center="0.3,0.2" RadiusX="0.5" RadiusY="0.5">
                 <GradientStop Color="Azure" Offset="0.1" />
                 <GradientStop Color="CornflowerBlue" Offset="1.1" />
               </RadialGradientBrush>
             </Ellipse.Fill>
           </Ellipse>
           <ContentPresenter Name="content" HorizontalAlignment="Center"
             VerticalAlignment="Center"/>
         </Grid>
       </ControlTemplate>
     </Button.Template>

   </Button>

  </Window>
```

Code C

```
using System.Windows;

   namespace SDKSample
   {
```

```
        public partial class ControlTemplateButtonWindow : Window
        {
            public ControlTemplateButtonWindow()
            {
                InitializeComponent();
            }

            void button_Click(object sender, RoutedEventArgs e)
            {
                MessageBox.Show("Hello, Windows Presentation Foundation!");
            }
        }
    }
```

Lire Modèles de contrôle en ligne: https://riptutorial.com/fr/xaml/topic/6782/modeles-de-controle

Chapitre 7: Modèles de données

Examples

Utilisation de DataTemplate dans un ListBox

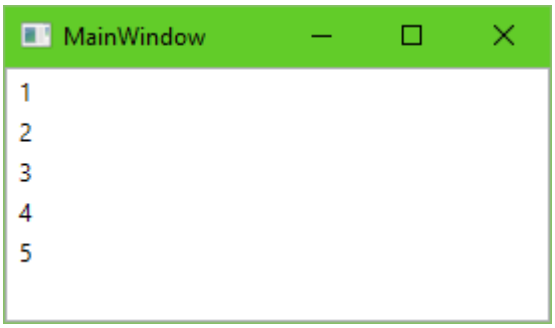
Supposons que nous ayons l'extrait de code XAML suivant:

```
<ListBox x:Name="MyListBox" />
```

Ensuite, dans le code-behind de ce fichier XAML, nous écrivons ce qui suit dans le constructeur:

```
MyListBox.ItemsSource = new[]
{
    1, 2, 3, 4, 5
};
```

En exécutant l'application, nous obtenons une liste de numéros que nous avons entrés.



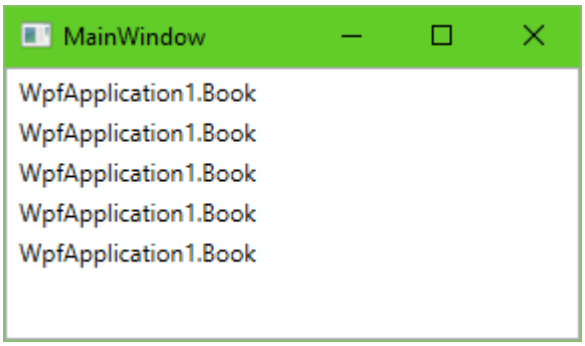
Cependant, si nous essayons d'afficher une liste d'objets d'un type personnalisé, comme ceci

```
MyListBox.ItemsSource = new[]
{
    new Book { Title = "The Hitchhiker's Guide to the Galaxy", Author = "Douglas Adams" },
    new Book { Title = "The Restaurant at the End of the Universe", Author = "Douglas Adams"
},
    new Book { Title = "Life, the Universe and Everything", Author = "Douglas Adams" },
    new Book { Title = "So Long, and Thanks for All the Fish", Author = "Douglas Adams" },
    new Book { Title = "Mostly Harmless", Author = "Douglas Adams" }
};
```

en supposant que nous avons une classe appelée `Book`

```
public class Book
{
    public string Title { get; set; }
    public string Author { get; set; }
}
```

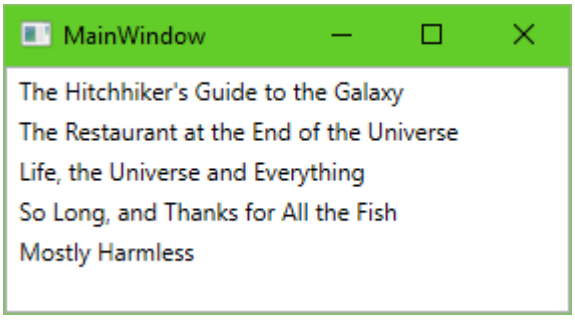
alors la liste ressemblerait à ceci:



Bien que nous puissions supposer que ListBox sera suffisamment "intelligent" pour afficher nos objets de livre correctement, nous voyons en réalité le nom complet du type de `Book` . Ce que ListBox fait en réalité ici, c'est d'appeler la méthode `ToString()` intégrée sur les objets à afficher et, bien que cela produise le résultat souhaitable dans le cas des nombres, l'appel à `ToString()` sur des objets de classes personnalisées entraîne l'obtention du nom leur type, comme on le voit sur la capture d'écran.

Nous pourrions résoudre ce problème en écrivant `ToString()` pour notre classe de livre, c.-à-d.

```
public override string ToString()
{
    return Title;
}
```



Cependant, ce n'est pas très flexible. Et si on veut aussi afficher l'auteur? Nous pourrions aussi écrire cela dans le `ToString` , mais que se passe-t-il si nous ne voulons pas cela dans toutes les listes de l'application? Que diriez-vous d'une belle couverture de livre à afficher?

C'est là que DataTemplates peut vous aider. Ce sont des extraits de XAML qui peuvent être "instanciés" si nécessaire, avec des détails correspondant aux données source pour lesquelles ils ont été créés. Autrement dit, si nous étendons notre code ListBox comme suit:

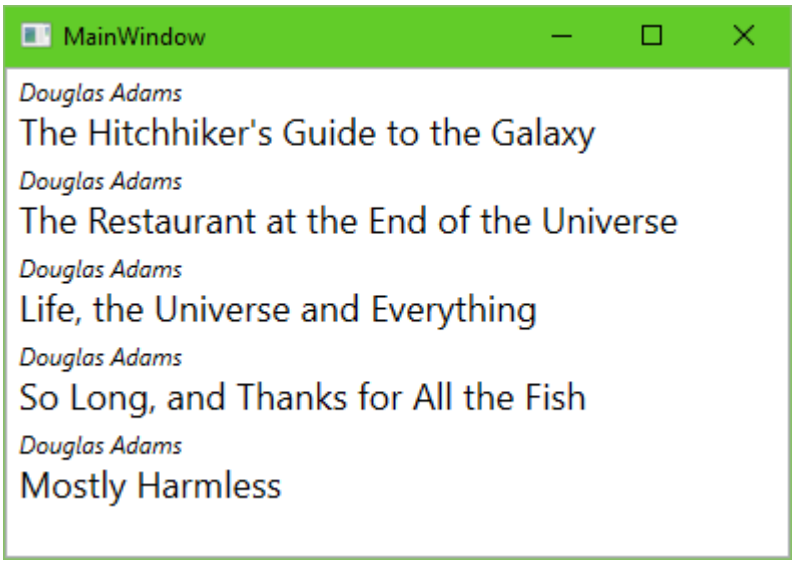
```
<ListBox x:Name="MyListBox">
    <ListBox.ItemTemplate>
        <DataTemplate>
            <TextBlock Text="{Binding Title}" />
        </DataTemplate>
    </ListBox.ItemTemplate>
</ListBox>
```

La liste créera alors un `TextBox` pour chaque élément de sa source, et les propriétés `Text` ces TextBox seront "remplies" avec les valeurs de la propriété `Title` de nos objets.

Si nous exécutons l'application maintenant, nous obtenons le même résultat que ci-dessus, * même si nous `ToString` implémentation personnalisée de `ToString` . Ce qui est avantageux à ce sujet, c'est que nous pouvons ensuite personnaliser ce modèle bien au-delà des capacités d'une simple `string` (et de `ToString`). Nous pouvons utiliser n'importe quel élément XAML souhaité, y compris les éléments personnalisés, et pouvons "lier" leurs valeurs aux données réelles de nos objets (comme `Title` dans l'exemple ci-dessus). Par exemple, étendez le modèle comme suit:

```
<ListBox x:Name="MyListBox">
    <ListBox.ItemTemplate>
        <DataTemplate>
            <StackPanel>
                <TextBlock FontStyle="Italic" Text="{Binding Author}" />
                <TextBlock FontSize="18" Text="{Binding Title}" />
            </StackPanel>
        </DataTemplate>
    </ListBox.ItemTemplate>
</ListBox>
```

Ensuite, nous obtenons une belle vue formatée de nos livres!



Lire Modèles de données en ligne: https://riptutorial.com/fr/xaml/topic/3858/modeles-de-donnees

Chapitre 8: Outils de développement XAML

Examples

Microsoft Visual Studio et Microsoft Expression Blend

Créez des interfaces utilisateur attrayantes pour les applications Windows Desktop avec Blend for Visual Studio, le premier outil de conception professionnel pour les applications XAML. Créez de magnifiques transitions et visualisations à l'aide de la suite complète d'outils de dessin vectoriel de Blend, de puissantes fonctions d'édition de modèles, d'animations en temps réel, de gestion des états visuels, etc.

Télécharger Visual Studio

Inspecteur WPF

WPF Inspector est un utilitaire qui se connecte à une application WPF en cours d'exécution pour résoudre les problèmes courants de mise en page, de liaison de données ou de style. WPF Inspector vous permet d'explorer une vue en direct de l'arborescence logique et visuelle, de lire et de modifier les valeurs de propriété des éléments, de surveiller le contexte des données, les déclencheurs de débogage, les styles de trace et bien plus encore.

Téléchargez l'inspecteur WPF de Codeplex

Espionner

Snoop est un outil open source disponible qui vous permet de parcourir l'arborescence visuelle d'une application WPF en cours d'exécution sans avoir besoin d'un débogueur et de modifier les propriétés.

Téléchargez WPF Snoop depuis GitHub

Suite de performance WPF

Windows SDK comprend une suite d'outils de profilage des performances pour les applications Windows Presentation Foundation (WPF) appelées WPF Performance Suite. WPF Performance Suite vous permet d'analyser le comportement d'exécution de vos applications WPF et de déterminer les optimisations de performances que vous pouvez appliquer. WPF Performance Suite comprend des outils de profilage des performances appelés Perforator et Visual Profiler. Cette rubrique explique comment installer et utiliser les outils Perforator et Visual Profiler dans WPF Performance Suite.

En savoir plus MSDN

Lire Outils de développement XAML en ligne: https://riptutorial.com/fr/xaml/topic/6800/outils-de-developpement-xaml

Chapitre 9: Travailler avec des fichiers XAML personnalisés

Examples

Lire un objet depuis XAML

Considérez qu'une structure des classes suivantes devrait être construite en XAML puis lue dans un objet CLR:

```
namespace CustomXaml
{
    public class Test
    {
        public string Value { get; set; }
        public List<TestChild> Children { get; set; } = new List<TestChild>();
    }

    public class TestChild
    {
        public string StringValue { get; set; }
        public int IntValue { get; set; }
    }
}
```

Les classes doivent soit ne pas avoir de constructeur explicite, soit fournir un constructeur vide. Pour garder le XAML propre, les collections doivent être initialisées. L'initialisation des collections dans XAML est également possible.

Pour lire XAML, la classe `XamlServices` peut être utilisée. Il est défini dans `System.Xaml` et doit être ajouté aux références. La ligne suivante lit ensuite le fichier `test.xaml` à partir du disque:

```
Test test = XamlServices.Load("test.xaml") as Test;
```

La méthode `XamlServices.Load` a plusieurs surcharges à charger depuis les flux et autres sources. Si vous lisez XAML à partir d'un fichier incorporé (comme c'est le cas dans WPF), la propriété `Build Action` génération définie sur `Page` par défaut doit être modifiée en une `Embedded Resource` . Sinon, le compilateur demandera des références aux assemblys WPF.

Le contenu du fichier XAML à lire devrait ressembler à ceci:

```
<Test xmlns="clr-namespace:CustomXaml;assembly=CustomXaml"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      Value="test">
    <Test.Children>
        <TestChild StringValue="abc" IntValue="123"/>
        <TestChild StringValue="{x:Null}" IntValue="456"/>
    </Test.Children>
</Test>
```

La définition pure `xmlns` permet l'utilisation de classes dans le même espace de noms sans préfixe. La définition de `xmlns:x` est nécessaire pour utiliser des constructions comme `{x:Null}` . Bien sûr, les préfixes pour d'autres espaces de noms ou assemblages peuvent être définis selon les besoins.

Écrire un objet dans XAML

Considérez qu'une structure des classes suivantes devrait être construite en XAML puis lue dans un objet CLR:

```
namespace CustomXaml
{
    public class Test
    {
        public string Value { get; set; }
        public List<TestChild> Children { get; set; } = new List<TestChild>();
    }

    public class TestChild
    {
        public string StringValue { get; set; }
        public int IntValue { get; set; }
    }
}
```

Pour écrire XAML, la classe `XamlServices` peut être utilisée. Il est défini dans `System.Xaml` et doit être ajouté aux références. La ligne suivante écrit alors le `test` instance de type `Test` dans le fichier `test.xaml` sur le disque:

```
XamlServices.Save("test.xaml", test);
```

La méthode `XamlServices.Save` a plusieurs surcharges à écrire dans les flux et autres cibles. Le XAML résultant doit ressembler à ceci:

```
<Test Value="test" xmlns="clr-namespace:CustomXaml;assembly=CustomXaml"
                   xmlns:scg="clr-namespace:System.Collections.Generic;assembly=mscorlib"
                   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Test.Children>
    <scg:List x:TypeArguments="TestChild" Capacity="4">
      <TestChild IntValue="123" StringValue="abc" />
      <TestChild IntValue="456" StringValue="{x:Null}" />
    </scg:List>
  </Test.Children>
</Test>
```

La définition pure `xmlns` permet l'utilisation de classes dans le même espace de noms sans préfixe. La définition de `xmlns:x` est nécessaire pour utiliser des constructions comme `{x:Null}` . Le rédacteur ajoute automatiquement le `xmlns:scg` pour initialiser une `List<TestChild>` pour la propriété `Children` de l'objet `Test` . Il ne repose pas sur la propriété en cours d'initialisation par le constructeur.

Lire Travailler avec des fichiers XAML personnalisés en ligne:

https://riptutorial.com/fr/xaml/topic/6693/travailler-avec-des-fichiers-xaml-personnalises

Crédits

S. No	Chapitres	Contributeurs
1	Démarrer avec xaml	ChrisF, Community, Explisam, Raamakrishnan A., Rafael Costa , RenDishen, Ryan Thomas
2	Contrôles de disposition	CKII, Filip Skakun
3	Convertisseurs	Adi Lester, AhammadaliPK, ChrisF, Jirajha, Rafael Costa
4	Différences dans les différents dialectes XAML	HeWillem, stefan.s
5	Liaison de données	Jirajha, RenDishen, Ryan Thomas, SeeuD1, Tobias
6	Modèles de contrôle	Vimal CK
7	Modèles de données	Dániel Kis-Nagy
8	Outils de développement XAML	Vimal CK
9	Travailler avec des fichiers XAML personnalisés	Alexander Mandt