EBook Gratis

APRENDIZAJE xaml

Free unaffiliated eBook created from
Stack Overflow contributors.

#xaml

Tabla de contenido

Acerca de

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

Capítulo 1: Empezando con xaml

Observaciones

E **X** tensible **Una** aplicación **M** arkup **L** idioma (XAML) es un lenguaje de marcado basado en XML desarrollado por Microsoft. Se utiliza en varias tecnologías de Microsoft como Windows Presentation Foundation (WPF), Silverlight, WinRT, Universal Windows Platform, etc. para definir la interfaz de usuario para las aplicaciones.

Versiones

Versión	Fecha de lanzamiento
WPF XAML	2006-11-21
Silverlight 3	2009-07-09
Silverlight 4	2010-04-15
Windows 8 XAML	2011-09-01

Examples

Instalación o configuración

La forma más fácil de escribir tu primer XAML es instalar Microsoft Visual Studio. Esto está disponible gratuitamente de Microsoft.

Una vez instalado, puede crear un nuevo proyecto, de tipo Aplicación WPF, ya sea con un código VB.NET o C #.

Esto es similar a los formularios de Windows en el sentido de que tiene una serie de ventanas, la principal diferencia es que estas ventanas están escritas en XAML y son mucho más sensibles a los diferentes dispositivos.

Todavía se necesita mejorar.

Hola Mundo

Aquí hay un ejemplo simple de una página XAML en WPF. Consiste en una `Grid` , un `TextBlock` y un `Button` , los elementos más comunes en 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>
```

Sintaxis	Descripción
`<Window>`	El contenedor raíz que aloja el contenido que visualiza los datos y permite a los usuarios interactuar con él. Una ventana de WPF es una combinación de un archivo XAML (.xaml), donde el elemento es la raíz, y un archivo CodeBehind (.cs).
`<Grid>`	Un panel de diseño que organiza sus elementos secundarios en una estructura tabular de filas y columnas.
`<TextBlock>`	Proporciona un control ligero para mostrar texto de cadena en su propiedad de texto o elementos de contenido de flujo en línea, como Negrita, Hipervínculo e InlineUIContainer, en su propiedad Inlines.
`<Button>`	Representa un botón de control que reacciona con el usuario haciendo clic en él.

Propiedad	Descripción
`Title`	Obtiene o establece el título de una ventana.
`Height`	Obtiene o establece la altura de un elemento.
`Width`	Obtiene o establece el ancho de un elemento.
`Text`	Obtiene o establece el contenido de texto de un elemento de texto.
`FontSize`	Obtiene o establece el tamaño de fuente de nivel superior para el texto.
`Background`	Obtiene o establece el color del pincel que pinta el fondo de un elemento.

Propiedad	Descripción
`Foreground`	Obtiene o establece el color del pincel que pinta la fuente de un texto en un elemento.
`Margin`	Obtiene o establece el valor que describe el espacio exterior entre un elemento y los otros.
`HorizontalAlignment`	Obtiene o establece las características de alineación horizontal aplicadas al elemento cuando se compone dentro de un elemento principal, como un panel o control de elementos.
`VerticalAlignment`	Obtiene o establece las características de alineación vertical aplicadas al elemento cuando se compone dentro de un elemento principal, como un panel o control de elementos.

Lea Empezando con xaml en línea: https://riptutorial.com/es/xaml/topic/903/empezando-con-xaml

Capítulo 2: Controles de diseño

Examples

Lona

`Canvas` es el más simple de los paneles. Coloca elementos en las coordenadas `Top/Left` especificadas.

```
<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` alinea el control de acuerdo con la propiedad de acoplamiento, en el orden en que se coloca en el control.

NOTA: `DockPanel` es parte del marco WPF, pero no viene con Silverlight / WinRT / UWP. Sin embargo, las implementaciones de código abierto son fáciles de encontrar.

```
<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 coloca sus controles uno tras otro. Actúa como un panel de acoplamiento con todos los muelles de su control establecidos en el mismo valor.

```
<!-- 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>
```

Para apilar elementos de abajo hacia arriba, use un panel de acoplamiento.

Cuadrícula

`Grid` se utiliza para crear diseños de tablas.

Definiciones básicas de filas y columnas.

```
<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>
```

NOTA: Todos los ejemplos siguientes usarán solo columnas, pero también son aplicables a las filas .

Definiciones de tamaño automático

Las columnas y filas se pueden definir con "Automático" como su ancho / alto. El tamaño automático tomará *tanto espacio como sea necesario* para mostrar su contenido, y no más.
Las definiciones de tamaño automático se pueden utilizar con definiciones de tamaño fijo.

```
<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>
```

Definiciones de estrellas simples

Las columnas y filas se pueden definir con `*` como su ancho / alto. Las filas / columnas de tamaño estrella ocuparán *todo el espacio que tengan* , independientemente de su contenido.
Las definiciones de tamaño estrella se pueden usar con definiciones de tamaño fijo y automático.
El tamaño de estrella es el predeterminado y, por lo tanto, se puede omitir el ancho de columna o el alto de fila.

```
<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>
```

Definiciones de tamaño de estrella proporcional

Además del hecho de que la estrella ocupa tanto espacio como puede, las definiciones de las estrellas también son proporcionales entre sí. Si no se menciona nada más, cada definición de estrella ocupará tanto espacio como las otras en la cuadrícula actual.

Sin embargo, es posible definir una relación entre los tamaños de diferentes definiciones simplemente agregándole un multiplicador. Por lo tanto, una columna definida como `2*` tendrá el doble de ancho que una columna definida como `*` . El ancho de una sola unidad se calcula dividiendo el espacio disponible por la suma de los multiplicadores (si no hay, se cuenta como 1). Por lo tanto, una cuadrícula con 3 columnas definidas como `*` , `2*` , `*` se presentará como 1/4, 1/2, 1/4.
Y una con 2 columnas definidas como `2*` , `3*` se presentará 2/5, 3/5.

Si hay definiciones automáticas o fijas en el conjunto, éstas se calcularán primero, y las definiciones en estrella tomarán el espacio restante después de eso.

```
<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>
```

Columna / Fila

Es posible hacer que un control se extienda más allá de su celda configurando Row / ColumnSpan. El valor establecido es el número de filas / columnas th

```
<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

El panel de envoltura actúa de manera similar al panel de apilamiento. Excepto cuando reconoce que los elementos excederán su tamaño, luego los envolverá en una nueva fila / columna, dependiendo de su orientación.

Orientación horizontal

```
<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>
```

Panel de envoltura 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

La cuadrícula uniforme colocará a todos sus hijos en un diseño de cuadrícula, cada niño en su propia celda. Todas las celdas tendrán el mismo tamaño. Se puede pensar que es una abreviatura de una cuadrícula donde todas las definiciones de filas y columnas se configuran en `*`

Filas y columnas predeterminadas

Por defecto, UniformGrid intentará crear un número igual de filas y columnas. Cuando una fila sea

demasiado larga, agregará una nueva columna.

Este código producirá una cuadrícula de 3x3 con las primeras 2 filas llenas y la última con un botón:

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

Filas / columnas especificadas

Puede decirle a UniformGrid exactamente cuántas filas y / o columnas desea tener.

```
<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>
```

NOTA: en caso de que ambas filas y columnas estén configuradas, y haya más elementos secundarios que celdas, no se mostrarán los últimos elementos secundarios de la cuadrícula

---

Propiedad FirstColumn

Una vez que se establece la propiedad Columnas, puede establecer la propiedad FirstColumn. Esta propiedad ingresará x celdas vacías en la primera fila antes de que se muestre el primer hijo. FirstColumn se debe establecer en un número más pequeño que la propiedad Columnas.

En este ejemplo, el primer botón se mostrará en la segunda columna de la primera fila:

```
<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>
```

Panel relativo

`RelativePanel` se introdujo en Windows 10 y se usa principalmente para admitir diseños adaptativos, donde los elementos secundarios del panel se distribuyen de manera diferente según el espacio disponible. `RelativePanel` se usa generalmente con estados visuales , que se usan para cambiar la configuración del diseño, adaptarse al tamaño de la pantalla o ventana, la orientación o el caso de uso. Los elementos secundarios utilizan propiedades adjuntas que definen dónde están en relación con el panel y entre sí.

```
<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>
```

Lea Controles de diseño en línea: https://riptutorial.com/es/xaml/topic/3634/controles-de-diseno

Capítulo 3: Convertidores

Parámetros

Parámetro	Detalles
valor	El valor para convertir de
tipo de objetivo	El tipo que se está convirtiendo a
parámetro	Valor opcional para controlar cómo funciona la conversión.
cultura	Objeto CultureInfo - requerido si se necesita localización

Observaciones

El método `Convert` convierte el valor del origen (generalmente el modelo de vista) al objetivo (generalmente una propiedad de un control).

El método `ConvertBack` convierte el valor del destino a la fuente. Sólo es necesario si la unión es `TwoWay` o `OneWayToSource` .

Cuando no se `ConvertBack` un `ConvertBack` , es decir, no hay una asignación uno a uno entre el valor de pre-conversión y el valor de post-conversión, es una práctica común que el método `ConvertBack` devuelva `DependencyProperty.UnsetValue` . Es una mejor opción que lanzar una excepción (por ejemplo, `NotImplementedException`) ya que evita errores de tiempo de ejecución inesperados. Además, los enlaces pueden beneficiarse de su `FallbackValue` cuando `DependencyProperty.UnsetValue` es devuelto por un convertidor.

Examples

Cadena a IsChecked Converter

En 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 clase de 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;
        }
    }
}
```

Convertidores 101

Los controles XAML pueden tener propiedades de dependencia que pueden vincularse a objetos de `DataContext` u otros controles. Cuando el tipo de objeto que se está enlazando es diferente del tipo de la propiedad de `DependencyProperty` destino, se puede usar un **convertidor** para adaptar un tipo a otro.

Los convertidores son clases que implementan `System.Windows.Data.IValueConverter` o `System.Windows.Data.IMultiValueConverter` ; WPF implementa algunos convertidores listos para usar, pero los desarrolladores pueden ver el uso en implementaciones personalizadas, como suele ser el caso.

Para usar un convertidor en XAML, se puede crear una instancia de una instancia en la sección `Resources` . Para el ejemplo a continuación, se utilizará `System.Windows.Controls.BooleanToVisibilityConverter` :

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

Observe el elemento `x:Key` definido, que luego se usa para hacer referencia a la instancia de `BooleanToVisibilityConverter` en el enlace:

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

En el ejemplo anterior, una propiedad `IsVisible` booleana se convierte a un valor de la enumeración `System.Windows.Visibility` ; `Visibility.Visible` if true, o `Visibility.Collapsed` contrario.

Creación y uso de un convertidor: BooleanToVisibilityConverter e InvertibleBooleanToVisibilityConverter

Para ampliar y ampliar la experiencia de enlace, tenemos convertidores para convertir un valor de un tipo en otro valor de otro tipo. Para aprovechar los Convertidores en un enlace de datos, primero debe crear una clase DataConverter que extienda

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

o

- `IMultiValueConverter` **(WPF)**

Si quieres convertir varios tipos en un solo tipo.
En este caso, nos centramos en convertir un valor `boolean True/False` a las correspondientes Visibilidades `Visibility.Visible` y `Visibility.Collapsed` . `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 **`Convert`** método se llama cada vez que **`GET`** datos **`FROM`** el **`ViewModel`** .
El **`ConvertBack`** está llamada a **`SET`** ing datos **`TO`** del **`ViewModel`** para **`BindingMode.TwoWay`** enlaces.

Por supuesto, también puede utilizar propiedades dentro de su convertidor. Echa un vistazo a este:

```
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 desea usar un convertidor en un `Binding` , simplemente declare que es un recurso en su página, ventana u otro elemento, dele una clave y proporcione las propiedades potencialmente necesarias:

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

y usarlo como `StaticResource` en un enlace:

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

Lea Convertidores en línea: https://riptutorial.com/es/xaml/topic/2996/convertidores

Capítulo 4: Diferencias en los diversos dialectos XAML.

Observaciones

XAML se usa en las aplicaciones Silverlight, Windows Phone, Windows RT y UWP. Compartir código o convertir código entre estos es a veces más difícil de lo deseable debido a diferencias sutiles entre los diversos dialectos XAML. Este tema se esfuerza por ofrecer una visión general de estas diferencias con una breve explicación.

Examples

Enlaces de datos compilados: la extensión de marcado {x: Bind}

Las bases de datos son esenciales para trabajar con XAML. El dialecto XAML para aplicaciones UWP proporciona un tipo de enlace: la extensión de marcado {x: Bind}.

Trabajar con {Binding XXX} y {x: Bind XXX} es casi todo equivalente, con la diferencia de que la extensión x: Bind funciona en tiempo de compilación, lo que permite mejores capacidades de depuración (por ejemplo, puntos de ruptura) y mejor rendimiento.

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

La extensión de marcado x: Bind solo está disponible para aplicaciones UWP. Obtenga más información sobre esto en este artículo de MSDN: https://msdn.microsoft.com/en-us/windows/uwp/data-binding/data-binding-in-depth .

Alternativas para Silverlight, WPF, Windows RT: use la sintaxis estándar {Binding XXX}:

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

Importando espacios de nombres en XAML

La mayoría de las veces necesita importar espacios de nombres en su archivo XAML. Cómo se hace esto es diferente para las diferentes variantes de XAML.

Para Windows Phone, Silverlight, WPF usa la sintaxis de espacio de nombres clr:

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

Windows RT, UWP usa la sintaxis de uso:

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

Multi Binding

Multi Binding es una característica exclusiva para el desarrollo de WPF. Permite vincular varios valores a la vez (generalmente se utiliza con un MultiValueConverter).

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

Las plataformas distintas de WPF no admiten el enlace múltiple. Debe encontrar soluciones alternativas (como mover el código de la vista y los convertidores al modelo de visualización) o recurrir a comportamientos de terceros como en este artículo: http://www.damirscorner.com/blog/posts/20160221-MultibindingInUniversalWindowsApps.html)

Lea Diferencias en los diversos dialectos XAML. en línea: https://riptutorial.com/es/xaml/topic/4498/diferencias-en-los-diversos-dialectos-xaml-

Capítulo 5: El enlace de datos

Sintaxis

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

Observaciones

Todas estas etiquetas producen el mismo resultado.

Examples

Enlace de cadena a propiedad de texto

Para cambiar el contenido de la interfaz de usuario en tiempo de ejecución, puede utilizar `Binding` . Cuando se cambia la propiedad enlazada del código, se mostrará a la interfaz de usuario.

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

Para notificar a la UI sobre los cambios, la propiedad debe generar el evento `PropertyChanged` desde la interfaz `INotifyPropertyChanged` o puede usar la `Dependency Property` .

El enlace funciona si la propiedad "Título" está en el archivo xaml.cs o en la clase Datacontext de `XAML` .

El Datacontext se puede configurar en el XAML directamente

```
<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>
```

Formato de enlaces de cadena

Al hacer un enlace de algo, por ejemplo, una fecha, es posible que desee mostrarlo en un formato específico sin alterarlo en el código.

Para hacer esto podemos usar la propiedad StringFormat.

Aquí hay unos ejemplos:

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

Esto formatea mi fecha a la siguiente:

Martes 16 de agosto de 2016

Aquí hay otro ejemplo para la temperatura.

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

Este formato para:

25 ° C

Los fundamentos de INotifyPropertyChanged

Si no solo desea mostrar objetos estáticos, sino que su interfaz de usuario responde a los cambios para correlacionar objetos, debe comprender los conceptos básicos de la interfaz `INotifyPropertyChanged` .

Suponiendo que tenemos nuestro `MainWindow` definido como

```
<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>
```

Con nuestro modelo de vista de clase `MainWindowViewModel` definido como

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

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

```
    }
}
```

El TextBlock de nuestra aplicación mostrará el texto *Hello World* debido a su enlace. Si nuestro ApplicationStateText cambia durante el tiempo de ejecución, nuestra interfaz de usuario no será notificada de dicho cambio.
Para implementar esto, nuestro DataSource, en este caso nuestro `MainWindowViewModel` , necesita implementar la Interfaz `INotifyPropertyChanged` . Esto hará que nuestros `Bindings` puedan suscribirse al `PropertyChangedEvent` .
Todo lo que tenemos que hacer es invocar `PropertyChangedEventHandler` cada vez que cambiemos nuestra propiedad `ApplicationStateText` esta forma:

```
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!";
        }
    }
}
```

y asegúrese de que nuestro `Binding` de `TextBlock.Text` realmente escucha 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>
```

Enlace a una colección de objetos con INotifyPropertyChanged y INotifyCollectionChanged

Supongamos que tiene un `ListView` que se supone que muestra todos los objetos de `User` enumerados en la Propiedad de `Users` de `ViewModel` donde las Propiedades del objeto de Usuario pueden actualizarse mediante programación.

```
<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>
```

A pesar de que `INotifyPropertyChanged` implementó correctamente para el objeto `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));
    }
}
```

y para su objeto `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));
    }
}
```

su interfaz de usuario no se actualizará, si se realiza un cambio a un usuario mediante programación.

Esto es simplemente porque solo ha configurado INotifyPropertyChanged en la instancia de la lista. Solo si vuelve a crear una instancia completa de la Lista si una propiedad de un elemento cambia, su interfaz de usuario se actualizará.

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

Sin embargo, esto es muy pesado e increíblemente malo para el rendimiento.
Si tiene una lista de 100,000 elementos que muestran tanto el ID como el nombre del usuario,

habrá 200,000 enlaces de datos en su lugar y cada uno tendrá que volver a crearlos. Esto da como resultado un retraso notable para el usuario cuando se realiza un cambio en cualquier cosa.

Para resolver parcialmente este problema, puede usar `System.ComponentModel.ObservableCollection<T>` lugar de `List<T>` :

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

`ObservableCollection` nos proporciona el evento `CollectionChanged` e implementa el propio `INotifyPropertyChanged` . Según MSDN, el evento aumentará, "[..] cuando un elemento se *agregue* , *elimine* , *cambie* , *mueva* o *se actualice la lista completa* ".
Sin embargo, pronto se dará cuenta de que, con .NET 4.5.2 y anteriores, `ObservableCollection` no generará un evento CollectionChanged si la propiedad de un elemento en la colección cambia, como se explica aquí .

Siguiendo esta solución, simplemente podemos implementar nuestra propia `TrulyObservableCollection<T>` sin la restricción `INotifyPropertyChanged` para `T` teniendo todo lo que necesitamos y exponiendo si `T` implementa `INotifyPropertyChanged` o no:

```
/*
 * 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);
    }
}
```

y defina nuestros `Users` Propiedad como `TrulyObservableCollection<User>` en nuestro ViewModel

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

Ahora se notificará a nuestra interfaz de usuario una vez que una propiedad de INPC de un elemento dentro de la colección cambie sin la necesidad de volver a crear cada `Binding` .

Lea El enlace de datos en línea: https://riptutorial.com/es/xaml/topic/3329/el-enlace-de-datos

Capítulo 6: Herramientas de desarrollo XAML

Examples

Microsoft Visual Studio y Microsoft Expression Blend

Cree interfaces de usuario atractivas para las aplicaciones de escritorio de Windows con Blend para Visual Studio, la principal herramienta de diseño profesional para aplicaciones XAML. Cree transiciones y visualizaciones hermosas con el conjunto completo de herramientas de dibujo vectorial de Blend, potentes funciones de edición de plantillas, animación en tiempo real, administración del estado visual y más.

Descargar Visual Studio

Inspector de WPF

WPF Inspector es una utilidad que se adjunta a una aplicación WPF en ejecución para solucionar problemas comunes con el diseño, el enlace de datos o el estilo. WPF Inspector le permite explorar una vista en vivo del árbol lógico y visual, leer y editar los valores de propiedad de los elementos, observar el contexto de los datos, desencadenadores de depuración, rastrear estilos y mucho más.

Descargar WPF Inspector desde Codeplex

Fisgonear

Snoop es una herramienta de código abierto disponible que le permite navegar por el árbol visual de una aplicación WPF en ejecución sin la necesidad de un depurador y cambiar las propiedades.

Descargar WPF Snoop desde GitHub

WPF Performance Suite

Windows SDK incluye un conjunto de herramientas de creación de perfiles de rendimiento para las aplicaciones de Windows Presentation Foundation (WPF) llamada WPF Performance Suite. WPF Performance Suite le permite analizar el comportamiento en tiempo de ejecución de sus aplicaciones WPF y determinar las optimizaciones de rendimiento que puede aplicar. WPF Performance Suite incluye herramientas de perfilado de rendimiento llamadas Perforator y Visual Profiler. Este tema describe cómo instalar y usar las herramientas Perforator y Visual Profiler en WPF Performance Suite.

Leer más MSDN

Lea Herramientas de desarrollo XAML en línea:
https://riptutorial.com/es/xaml/topic/6800/herramientas-de-desarrollo-xaml

Capítulo 7: Plantillas de control

Examples

Plantillas de control

Las interfaces de usuario predeterminadas para los controles WPF se construyen normalmente a partir de otros controles y formas. Por ejemplo, un botón está compuesto de los controles ButtonChrome y ContentPresenter. ButtonChrome proporciona la apariencia estándar del botón, mientras que ContentPresenter muestra el contenido del botón, según lo especificado por la propiedad Contenido. A veces, la apariencia predeterminada de un control puede ser incongruente con la apariencia general de una aplicación. En este caso, puede usar un ControlTemplate para cambiar la apariencia de la interfaz de usuario del control sin cambiar su contenido y comportamiento.

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>
```

Código 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!");
            }
        }
    }
```

Lea Plantillas de control en línea: https://riptutorial.com/es/xaml/topic/6782/plantillas-de-control

Capítulo 8: Plantillas de datos

Examples

Usando DataTemplate en un ListBox

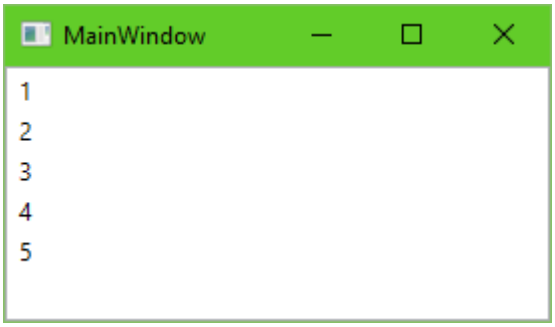
Supongamos que tenemos el siguiente fragmento de código XAML:

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

Luego, en el código subyacente para este archivo XAML, escribimos lo siguiente en el constructor:

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

Al ejecutar la aplicación, obtenemos una lista de números que ingresamos.



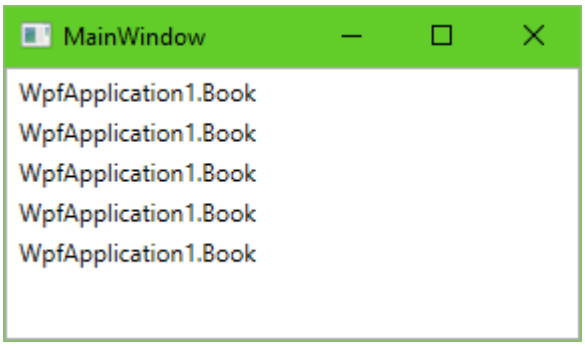
Sin embargo, si intentamos mostrar una lista de objetos de un tipo personalizado, como este

```
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" }
};
```

asumiendo que tenemos una clase llamada `Book`

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

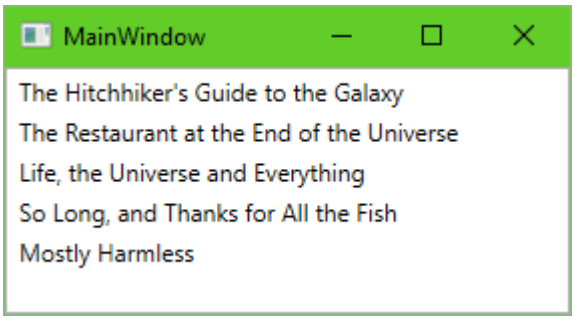
entonces la lista se vería algo como esto:



Si bien podemos suponer que el ListBox será "lo suficientemente inteligente" para mostrar los objetos de nuestro libro de forma correcta, lo que realmente vemos es el nombre completo del tipo de `Book` . Lo que ListBox realmente hace aquí es llamar al método `ToString()` en los objetos que desea mostrar, y mientras eso produce el resultado deseable en el caso de los números, al llamar a `ToString()` en los objetos de clases personalizadas se obtiene el nombre de Su tipo, como se ve en la captura de pantalla.

Podríamos resolverlo escribiendo `ToString()` para nuestra clase de libros, es decir,

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



Sin embargo, eso no es muy flexible. ¿Y si también queremos mostrar al autor? También podríamos escribir eso en el `ToString` , pero ¿qué `ToString` si no queremos eso en todas las listas de la aplicación? ¿Qué tal una cubierta de libro agradable para mostrar?

Ahí es donde DataTemplates puede ayudar. Son fragmentos de XAML que pueden ser "instanciados" según sea necesario, rellenados con detalles de acuerdo con los datos de origen para los que fueron creados. En pocas palabras, si extendemos nuestro código ListBox de la siguiente manera:

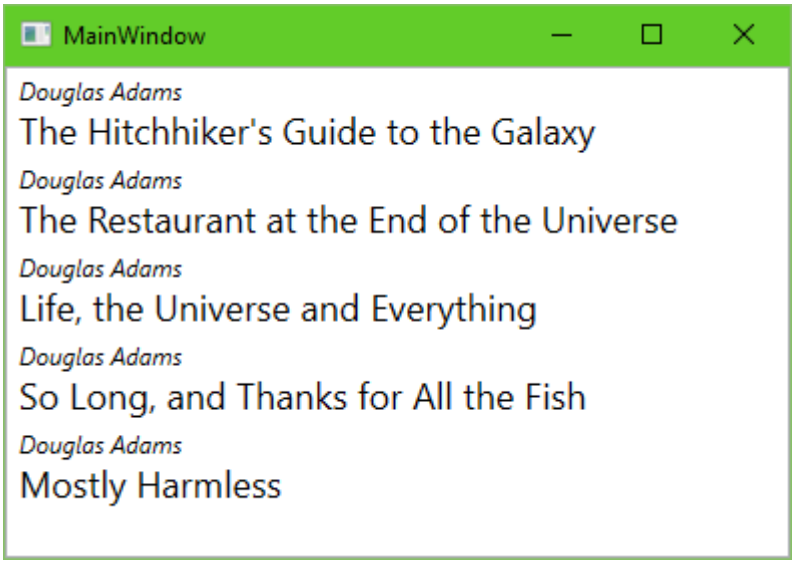
```
<ListBox x:Name="MyListBox">
    <ListBox.ItemTemplate>
        <DataTemplate>
            <TextBlock Text="{Binding Title}" />
        </DataTemplate>
    </ListBox.ItemTemplate>
</ListBox>
```

luego, la lista creará un `TextBox` para cada elemento en su fuente, y esos TextBoxes tendrán sus propiedades de `Text` "rellenadas" con valores de la propiedad `Title` de nuestros objetos.

Si ejecutamos la aplicación ahora, obtendremos el mismo resultado anterior *, incluso si eliminamos la implementación personalizada de `ToString` . Lo que es beneficioso de esto es que podemos personalizar esta plantilla mucho más allá de las capacidades de una `string` simple (y `ToString`). Podemos usar cualquier elemento XAML que queramos, incluidos los personalizados, y podemos "vincular" sus valores a los datos reales de nuestros objetos (como `Title` en el ejemplo anterior). Por ejemplo, extienda la plantilla de la siguiente manera:

```
<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>
```

¡Entonces obtenemos una buena vista formateada de nuestros libros!



Lea Plantillas de datos en línea: https://riptutorial.com/es/xaml/topic/3858/plantillas-de-datos

Capítulo 9: Trabajando con archivos personalizados XAML

Examples

Leyendo un objeto de XAML

Considere que una estructura de las siguientes clases se debe construir en XAML y luego leer en un objeto 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; }
    }
}
```

Las clases no deben tener un constructor explícito o proporcionar un vacío. Para mantener limpio el XAML, las colecciones deben inicializarse. Sin embargo, también es posible inicializar colecciones en XAML.

Para leer XAML se puede usar la clase `XamlServices` . Se define en `System.Xaml` que debe agregarse a las referencias. La siguiente línea luego lee el archivo `test.xaml` del disco:

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

El método `XamlServices.Load` tiene varias sobrecargas para cargar desde flujos y otras fuentes. Si se lee XAML desde un archivo incrustado (como se hace en WPF), la propiedad `Build Action` que se establece en `Page` de forma predeterminada debe cambiarse a, por ejemplo, `Embedded Resource` . De lo contrario, el compilador pedirá referencias a los ensamblados de WPF.

El contenido del archivo XAML para leer debe tener este aspecto:

```
<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 definición de `xmlns` pura permite el uso de clases en el mismo espacio de nombres sin prefijo. La definición de los `xmlns:x` es necesaria para usar construcciones como `{x:Null}` . Por supuesto, los prefijos para otros espacios de nombres o ensamblajes se pueden definir según sea necesario.

Escribiendo un objeto a XAML

Considere que una estructura de las siguientes clases se debe construir en XAML y luego leer en un objeto 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; }
    }
}
```

Para escribir XAML se puede usar la clase `XamlServices` . Se define en `System.Xaml` que debe agregarse a las referencias. La siguiente línea escribe la `test` instancia que es de tipo `Test` en el archivo `test.xaml` en el disco:

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

El método `XamlServices.Save` tiene varias sobrecargas para escribir en secuencias y otros destinos. El XAML resultante debería verse algo como esto:

```
<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 definición de `xmlns` pura permite el uso de clases en el mismo espacio de nombres sin prefijo. La definición de los `xmlns:x` es necesaria para usar construcciones como `{x:Null}` . El escritor agrega automáticamente `xmlns:scg` para inicializar una `List<TestChild>` para la propiedad `Children` del objeto `Test` . No se basa en que la propiedad sea inicializada por el constructor.

Lea Trabajando con archivos personalizados XAML en línea:
https://riptutorial.com/es/xaml/topic/6693/trabajando-con-archivos-personalizados-xaml

Creditos

S. No	Capítulos	Contributors
1	Empezando con xaml	ChrisF, Community, Explisam, Raamakrishnan A., Rafael Costa , RenDishen, Ryan Thomas
2	Controles de diseño	CKII, Filip Skakun
3	Convertidores	Adi Lester, AhammadaliPK, ChrisF, Jirajha, Rafael Costa
4	Diferencias en los diversos dialectos XAML.	HeWillem, stefan.s
5	El enlace de datos	Jirajha, RenDishen, Ryan Thomas, SeeuD1, Tobias
6	Herramientas de desarrollo XAML	Vimal CK
7	Plantillas de control	Vimal CK
8	Plantillas de datos	Dániel Kis-Nagy
9	Trabajando con archivos personalizados XAML	Alexander Mandt