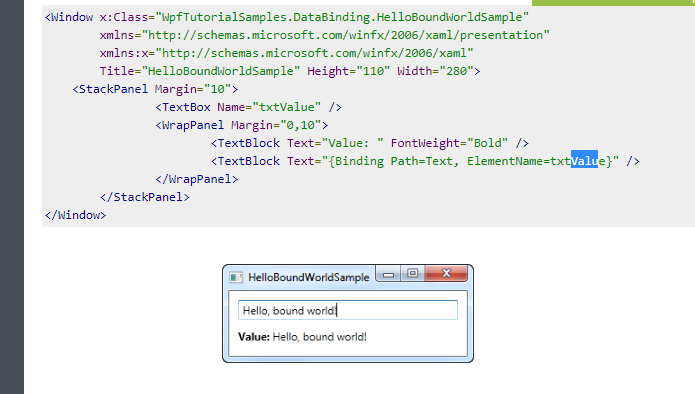
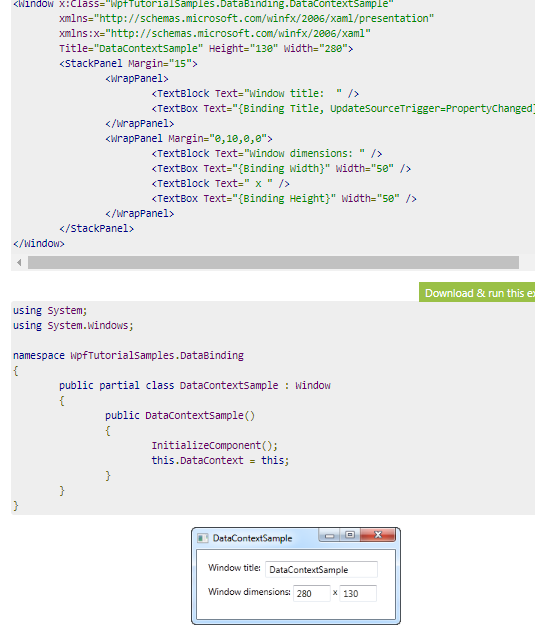
Data binding is general technique that binds two data/information sources together and maintains synchronization of data.



Using the DataContext

**Importante:!!!!!!**

The DataContext property is the default source of your bindings, unless you specifically declare another source, like we did in the previous chapter with the ElementName property



However, this doesn't mean that you have to use the same DataContext for all controls within a Window. Since each control has its own DataContext property, you can easily break the chain of inheritance and override the DataContext with a new value.

The UpdateSourceTrigger property

In the previous article we saw how changes in a TextBox was not immediately sent back to the source. Instead, the source was updated only after focus was lost on the TextBox. This behavior is controlled by a property on the binding called **UpdateSourceTrigger**. It defaults to the value "Default", which basically means that the source is updated based on the property that you bind to. As of writing, all properties except for the Text property, is updated as soon as the property changes (PropertyChanged), while the Text property is updated when focus on the destination element is lost (LostFocus).

Default is, obviously, the default value of the UpdateSourceTrigger. The other options are **PropertyChanged**, **LostFocus** and **Explicit**. The first two has already been described, while the last one simply means that the update has to be pushed manually through to occur, using a call to UpdateSource on the Binding.

Default-🡪 Lost focus 🡪 Sabemos como funciona

PropertyChanged 🡪 Cambio instantáneo…sabemos como funciona

Explicit🡪 Por ejemplo, pulsando un botón, que se actualice el texto del textbox :

<WrapPanel>

<TextBlock Text="Window title: " />

<TextBox Name="txtWindowTitle" Text="{Binding Title, UpdateSourceTrigger=Explicit}" Width="150" />

<Button Name="btnUpdateSource" Click="btnUpdateSource\_Click" Margin="5,0" Padding="5,0">\*</Button>

</WrapPanel>

public partial class DataContextSample : Window

{

public DataContextSample()

{

InitializeComponent();

this.DataContext = this;

}

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

{

BindingExpression binding = txtWindowTitle.GetBindingExpression(TextBox.TextProperty);

binding.UpdateSource();

}

}

Responding to changes

So far in this tutorial, we have mostly created bindings between UI elements and existing classes, but in real life applications, you will obviously be binding to your own data objects. This is just as easy, but once you start doing it, you might discover something that disappoints you: **Changes are not automatically reflected, like they were in previous examples.**

<https://www.wpf-tutorial.com/data-binding/responding-to-changes/>

1. Reflecting changes in the list data source

The first step is to get the UI to respond to changes in the list source (ItemsSource), like when we add or delete a user. What we need is a list that notifies any destinations of changes to its content, and fortunately, WPF provides a type of list that will do just that. It's called ObservableCollection, and you use it much like a regular List<T>, with only a few differences.

1. Reflecting changes in the data objects

The second step is to let our custom User class implement the INotifyPropertyChanged interface. By doing that, our User objects are capable of alerting the UI layer of changes to its properties

**Quedando así:**

MainWindow.xaml 🡪 ListUsers

MainWindow.cs 🡪 ListUsers.source = ObservableCollection<User>

Model -> User implements INotifyPropertyChanged

<Window x:Class="WpfTutorialSamples.DataBinding.ChangeNotificationSample"

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

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

Title="ChangeNotificationSample" Height="135" Width="300">

<DockPanel Margin="10">

<StackPanel DockPanel.Dock="Right" Margin="10,0,0,0">

<Button Name="btnAddUser" Click="btnAddUser\_Click">Add user</Button>

<Button Name="btnChangeUser" Click="btnChangeUser\_Click" Margin="0,5">Change user</Button>

<Button Name="btnDeleteUser" Click="btnDeleteUser\_Click">Delete user</Button>

</StackPanel>

<ListBox Name="lbUsers" DisplayMemberPath="Name"></ListBox>

</DockPanel>

</Window>

namespace WpfTutorialSamples.DataBinding

{

public partial class ChangeNotificationSample : Window

{

private ObservableCollection<User> users = new ObservableCollection<User>();

public ChangeNotificationSample()

{

InitializeComponent();

users.Add(new User() { Name = "John Doe" });

users.Add(new User() { Name = "Jane Doe" });

lbUsers.ItemsSource = users;

}

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

{

users.Add(new User() { Name = "New user" });

}

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

{

if(lbUsers.SelectedItem != null)

(lbUsers.SelectedItem as User).Name = "Random Name";

}

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

{

if(lbUsers.SelectedItem != null)

users.Remove(lbUsers.SelectedItem as User);

}

}

public class User : INotifyPropertyChanged

{

private string name;

public string Name {

get { return this.name; }

set

{

if(this.name != value)

{

this.name = value;

this.NotifyPropertyChanged("Name");

}

}

}

public event PropertyChangedEventHandler PropertyChanged;

public void NotifyPropertyChanged(string propName)

{

if(this.PropertyChanged != null)

this.PropertyChanged(this, new PropertyChangedEventArgs(propName));

}

}

}

Value conversion with IValueConverter

So far we have used some simple data bindings, where the sending and receiving property was always compatible. **However, you will soon run into situations where you want to use a bound value of one type and then present it slightly differently.**

For cases like this, you can use a value converter. These small classes, which implement the IValueConverter interface, will act like middlemen and translate a value between the source and the destination. So, in any situation where you need to transform a value before it reaches its destination or back to its source again, you likely need a converter.

<https://www.codeproject.com/Tips/868163/IValueConverter-Example-and-Usage-in-WPF>

Veamos:

So basically, we are binding string value from textbox to Ischecked property of the combobox. But IsChecked requires a Boolean value. So here is where the Value converter concept kicks in where we convert text value from textbox to boolean value to set IsChecked property.

<Window x:Class="IValueConverterExample.MainWindow"

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

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

Title="IValueConverterExample" Height="100" Width="300"

xmlns:local="clr-namespace:IValueConverterExample">

<Window.Resources>

<local:YesNoToBooleanConverter x:Key="YesNoToBooleanConverter"/>

</Window.Resources>

<Grid>

<StackPanel Margin="10">

<TextBox Name="txtValue" /> **<SOURCE>**

<CheckBox IsChecked="{Binding ElementName=txtValue,

**<TARGET>** Path=Text,

Converter={StaticResource YesNoToBooleanConverter}}"

Content="Yes" />

</StackPanel>

</Grid>

</Window>

public class YesNoToBooleanConverter : IValueConverter

{

public object Convert(object value, Type targetType, object parameter,

System.Globalization.CultureInfo culture)

{

switch (value.ToString().ToLower())

{

case "yes":

return true;

case "no":

return false;

default:

return Binding.DoNothing;

}

}

public object ConvertBack(object value, Type targetType, object parameter,

System.Globalization.CultureInfo culture)

{

if (value is bool)

{

if ((bool)value == true)

return "yes";

else

return "no";

}

return "no";

}

}

El target sera el element que tenga el {Binding}, en este caso el checkbox

**Convert-🡪 El valor del textbox actualize el checkbox**

**Convert** method gets called **when source updates target** object.

The Convert() methods assume that it receives a string as the input (the value parameter) and then converts it to a Boolean true or false value

**ConvertBack** 🡪 El checkbox pondrá yes o no en string

**ConvertBack** method gets called **when target updates source** object.

The ConvertBack() method obviously does the opposite: It assumes an input value with a Boolean type and then returns the English word "yes" or "no" in return, with a fallback value of "no".

Introduction to WPF Commands

To have interactions in application, we use either events or commands. La ventaja de los commandos frente a los eventos:

For example, a Save Command we can be executed on Save Button Click, or by Pressing Ctrl+S or by Choosing Save Option from Menu bar.

Commands help you to respond to a common action from several different sources, using a single event handler. It also makes it a lot easier to enable and disable user interface elements based on the current availability and state.

Commands make it simpler. You associate these actions with a single command. Command will be linked a single event handler. Which will contain the logic.

En proyectos que no sigan el MVVM podremos usar eventos…En cambio, si estamos siguiendo el MVVM, deberíamos usar commands.

Ver simple Command Binding

<https://code.msdn.microsoft.com/windowsdesktop/Learn-Simple-MVVM-and-8beb6bf0>

**Implementing ICommand Interface**

Normalmente, para controlar los eventos, añadíamos un event\_handler directamente en la vista (mainWindow.xaml) y desarrollábamos su código en el mainWindow.cs-🡪

(click)=”clickMethod()”

public void clickMethod(){}

Lo que se pretende con el MVVM es quitar el codeBehind…Es decir, que no se ponga código en el mainWindow.cs … es decir, en el archivo c# asociado a cada vista o xaml.

Ver 2 proyectos de ImplementingICommandInterface e ImplementingIcommandWithParameters