Arnaud Weil

Learn WPF MVVM

XAML, C# and the MVVM pattern

Learn WPF MVVM - XAML, C# and the MVVM pattern

Be ready for coding away next week using WPF and MVVM

Arnaud Weil

This book is for sale at http://leanpub.com/learnwpf

This version was published on 2020-02-19

To my parents, for teaching me freedom and making sure I can enjoy it.

To my wonderful family. Your love and support fueled this book.

To my readers who suggested improvements to this book, especially Doğan Kartaltepe.

Contents

1. Introduction

1.1 What this book is not

I made my best to keep this book small, so that you can learn WPF quickly without getting lost in petty details. If you're looking for a reference book where you'll find answers to all the questions you may have within the next 4 years of your WPF practice, you'll find other heavy books for that.

My purpose is to swiftly provide you with the tools you need to code your first WPF application using the MVVM pattern and be able to look for more by yourself when needed. While some authors seems to pride themselves in having the thickest book, in this series I'm glad I achieved the thinnest possible book for my purpose. Though I tried my best to keep all of what seems necessary, based on my 14 years experience of teaching.

I assume that you know what WPF is and when to use it. In case you don't, read the following *Why WPF ?* chapter.

1.2 Prerequisites

In order for this book to meet its goals, you must :

- Have basic experience creating an application with C# (any type of application is alright).
- Have working knowledge of Visual Studio.
- Have basic knowledge of XML syntax.
- Have basic knowledge of SQL Server.

> You could as well use VB.NET to code a WPF application. I chose to include only C# code in that book because I want it small and my field experience shows that almost every teams chooses C# over VB.NET nowadays.

1.3 How to read this book

This book's aim is to make you productive as quickly as possible. For this we'll use some theory, several demonstrations, plus exercises. Exercises appear like the following:

Do it yourself: Time to grab your keyboard and code away to meet the given objectives.

1.4 Tools you need

The only tool you'll need to work through that book is Visual Studio 2015. You can get any of those editions:

- Visual Studio 2015 Community (free)
- Visual Studio 2015 Professional

1.5 Source code

All of the source code for the demos and do-it-yourself solutions is available at https://bitbucket.org/epobb/learnwpfexercises

It can be downloaded as a ZIP file[1], or if you installed GIT you can simply type:

```
git clone https://bitbucket.org/epobb/learnwpfexercises\
.git
```

[1] https://bitbucket.org/epobb/learnwpfexercises/downloads

2. Why WPF ?

If you're in a hurry, you can safely skip this chapter and head straight to the Creating a WPF application chapter. This *Why WPF* chapter is there for those that want to know why WPF should be used.

WPF is a .NET development framework for desktop applications that solves several problems encountered with previous development frameworks.

Applications were counter-intuitive

Do you remember Windows XP? In order to turn off the computer, you had to press a button titled *Start*. Not really intuitive, is it? While this is a mainstream example, most software suffered from bad user experience: it was simply too complicated for a user to find her way around.

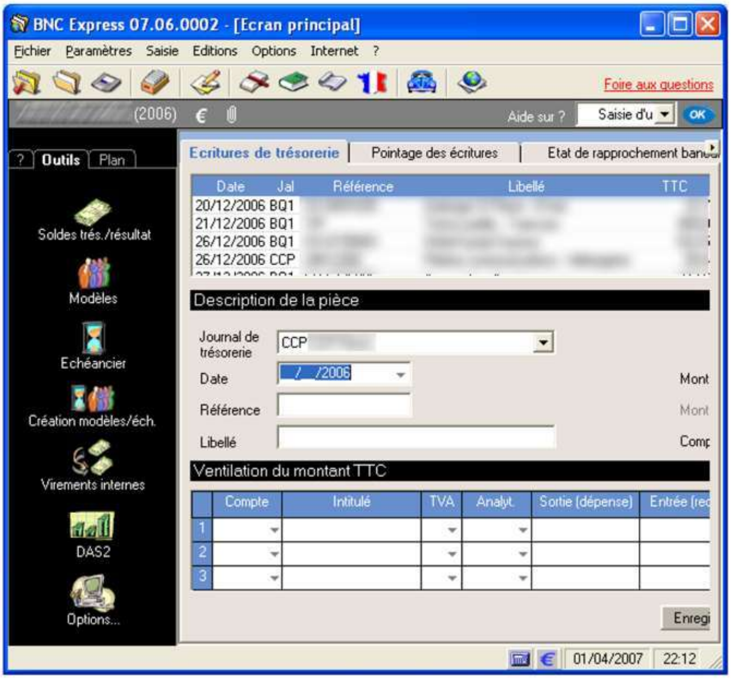
Some programs were even so complicated to use that a *wizard* was run when launched in order to guide users through the process of using the application. Wait: "process"? Using an application shouldn't be a *process*. It shouldn't be complicated to use an application, in fact the application should adapt to the user. While that debate would be nice for UX experts, my point is: why were applications so complicated to use?

The answer to that question is rooted within that simple fact: developers were asked to design the user-experience using code. Why? Because in many technology stacks, the user experience is *coded* by a developer, not *drawn* by a UX expert. Now, how can we expect someone to work correctly using inappropriate tools and lacking the necessary knowledge? A developer has little knowledge of user experience, and a programming language is not an appropriate way to create a user interface.

It all comes down to this: user-experience shouldn't be created using code, and it should be designed by a UX expert.

Applications were dull

Here is an application I used for accounting:



Date	Jal	Référence	Libellé	TTC
20/12/2006	BQ1			
21/12/2006	BQ1			
26/12/2006	BQ1			
26/12/2006	CCP			

Journal de trésorerie: CCP
Date: / /2006
Référence:
Libellé:

	Compte	Intitulé	TVA	Analyt.	Sortie (dépense)	Entrée (rec
1						
2						
3						

Alright, the program did the job it was supposed to do. But oh my, what a dull interface. Plus it’s not appealing at all. It doesn’t handle resizes correctly, doesn’t fill the available screen estate, and it looks like the icons were randomly picked. No need to tell you that I wasn’t eager to use that program as often as I needed to.

But it would be too easy to blame the developer for that application’s dullness. Coding a nice UI could easily double development time when using frameworks like Windows Forms, because you have to *code* in order to handle:

- Resizing
- Homogeneity
- Styling
- Elements positioning

Another source for that dullness is the fact that few developers have design skills. And vice-versa.

Nice GUIs could be dreamed of but not implemented

When you watch a movie or your favorite series, look at the user interface when people use computers. Did you notice how well designed, fluid and attractive they are? When some evil hacker tries to enter a system, he just has to press a big shiny “Hack” button. And when the hero shows the President some exceptional event live, she just slides through the information, zooms in and out in a fluid manner.

Same goes with Tom Cruise in Minority Report: in order to browse through files, he just moves around the pictures and movies using gestures:

What does it mean? That people who can create attractive, intuitive, user-friendly IHMs exist. However they work for movies, not for the computer industry. Why, you ask? Well, they were fed up with us developers.

Just think about the latest time some designer (or any creative people for that matter) came in and asked “hey, it would be great if there was a floating unicorn and when you pulled the hair it would float around and [add whatever you need here]”. What did you answer? Probably something that goes along those lines: “it’s not possible”. But what you really meant was: “it’s not possible to do so in a time that is reasonable, since it would take more time to code than the business logic itself”. And you were right, because your framework didn’t allow you to do so.

So do you know what happened? Those creative people got tired of seeing their ideas teared down and they went to work somewhere else. At some place where they wouldn’t hear “no” as the only answer to their ideas. Movies, series, you name it.

Appearance and logic separation

When Windows Forms, MFC C++, Java Swing or other client application frameworks were designed, the developers did what seemed natural to them: use the coding language in order to describe the user interface. For instance, a UI in Windows Forms is described using C# or VB.NET:

Windows Forms example of a UI description

```
public class Form1 : Form
{
  public Form1()
  {
    Button b = new Button();
    b.Text = "Buy stocks";
    b.Left = 20;
    b.Top = 40;
    b.Click += new EventHandler(b_Click);
  }
  void b_Click(object sender, EventArgs e)
  {
      // ...
  }
}
```

In the example above the button creation, position and appearance are set using C#. Which brings two problems:

1. A designer cannot edit this code. Even if she had the knowledge to do so, would you allow a designer to edit C# code?
2. A quick look at that code doesn't give a clue about the button appearance. Which makes any design work harder.

In fact, we all know that **presentation code and logical code should not be mixed**. But Windows Forms made the mistake. And many other frameworks did.

The WPF solution

In the HTML world, problems aren't so tough: designers work on the appearance while coders work on the business logic. Why is it so? Simply because things are separated: designers work in HTML and CSS files that describe the appearance, while developers work in JavaScript files. Plus HTML and CSS are quite adequate for describing an appearance.

Microsoft took the same approach with WPF. But HTML would have been too limited for desktop applications so they simply created XAML. XAML (XML application markup language) is XML, and you can think of it as HTML on steroids.

Since mixing presentation code and logical code was an error, WPF separates them. For each screen we have two files:

- a XAML file, describing the appearance, including any animation;
- a C# file, describing the functional logic of the screen. That file is called *code-behind*.

Practically, when you create a screen named *MyScreen*, it will be made of two files: MyScreen.xaml (appearance) and MyScreen.xaml.cs (code-behind).

Using separate files makes everything better: designers and developers can work on the same project, each on their own files.

Apart from this separation, WPF also introduced the following features:

- Controls composition: most controls can host other controls. For instance you can have buttons inside a ListBox control , or any shape and even video inside a Button control.
- Adaptation to any screen resolution: when working with pixels as in Windows Forms, programs get smaller as the resolution rises. WPF uses device-independent pixels, that state the real size independently of the screen resolution.

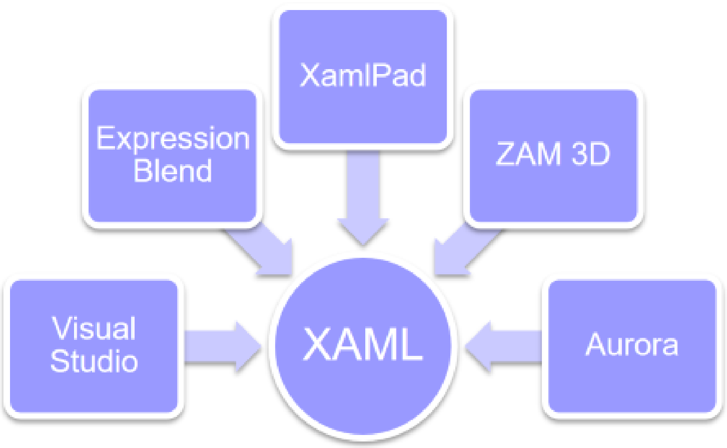
What does it all mean?

WPF simply allows for gorgeous user interfaces, which can be created before, during, or after the business logic is written. This allows for instance for a prototype to be turned into an application just by adding the business logic in C#.

XAML being very flexible, most of the design work that would have taken weeks using previous frameworks is done in hours. For instance, adding close buttons to tabs in Windows Forms takes 5 days, but doing so in WPF is a matter of minutes even though the TabControl didn’t include them.

XAML

Though it can easily be used by a designer, XAML is an extremely powerful tool. Being XML-based, it can cope with several XAML-specific or XML tools:

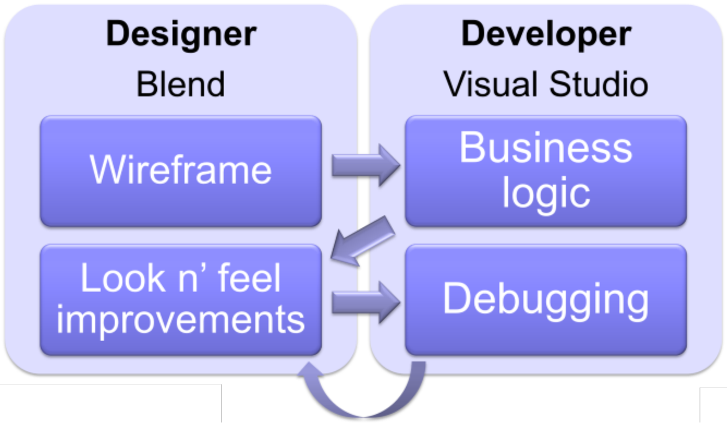


One feature that makes XAML so powerful is that it is a very easy way to instantiate .NET classes. More about that in the Understanding XAML chapter.

3. Creating a WPF application

3.1 Developer - designer workflow

When working on a WPF application there are two roles:



The designer is in charge of creating the wireframe and then high quality version of the user interface. The developer is in charge of coding the business logic, connecting with data and, well, um… debugging.

I'm talking about *roles* here. On a small team, the developers themselves could take the the *designer* role. On a large team however, it’s a good idea to have separate people in charge of those roles. Simply because a developer is not a designer. Though this book is going to teach you how to use the designer tools, you’ll realize that using a hammer doesn’t make you a craftsman. Designing nice user interfaces has its own learning curve.

In fact, I’ve been very impressed by the designer work on the large WPF projects I took part in. In a few days they were able to provide XAML files that made the application really appealing to the users. Without impacting business code.

3.2 Editors

Since there are two roles, there are two tools. Though you could use any of those tools in order to do all the work, each of them makes specific parts of the job faster and more convenient.

Visual Studio targets developers. Use it when adding controls, manually editing XAML, and writing the business logic.

Blend for Visual Studio targets designers. It is used when changing the appearance of controls and creating animations.

> *Blend for Visual Studio* is now installed together with Visual Studio. It was previously sold as a separate

program, *Expression Blend*. When working with WPF with versions of Visual Studio that are older than Visual Studio 2013, you'll need to separately install Expression Blend if you need to use it.

3.3 Adding a control

There are two ways to add a control to a screen[^screen]:

1. drag and drop the control from the toolbox;
2. simply add an XML element in the XAML file.

In case you manually add the XML element, its position and size depends on the container. We'll talk about containers in the Layout chapter, but for now just be aware that the control will take all the screen size when added to a *Grid*, or remain at the top-left corner when added to a *Canvas*.

For instance, the following XAML code will display a *Button* control that spans the whole screen:

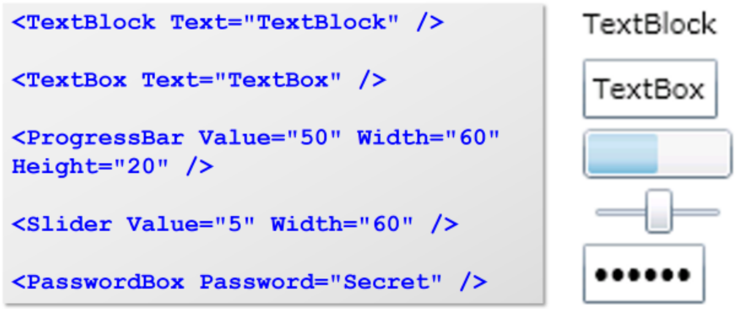
```
<Grid xmlns="...">
  <Button Content="Hello world" />
</Grid>
```

3.4 Simple controls

WPF provides relatively few controls. This is due to the fact that their appearance can be easily and completely revamped using pure XAML as we'll see in the Change a control's look chapter. Let's review the basic ones.

Basic controls

There is almost no need to explain much about those controls. On the left, you can see their declaration in XAML; on the right, their *default*[^basicdefault] appearance.



```
<TextBlock Text="TextBlock" />

<TextBox Text="TextBox" />

<ProgressBar Value="50" Width="60"
Height="20" />

<Slider Value="5" Width="60" />

<PasswordBox Password="Secret" />
```

Those controls are symmetrical. While *TextBlock* and *TextBox* allow for a *string* to be displayed or input as their *Text* property, *ProgressBar* and *Slider* allow for a *double* to be displayed or input as their *Value* property.

Note that in order to display text, the *TextBlock* control should be preferred to the *Label* control. The *Label* control is a much more flexible content control, which means it can display much more than text. Since it can display anything, the *Label* control lacks properties like *TextWrapping* that enable long text to be wrapped, and can be found on the *TextBlock* control.

Multimedia controls

The *Image* control displays, well, any picture, and the *MediaElement* displays movies. Both share a common resizing behavior:

- they resize their content to fit the size assigned to the control;
- they provide a *Stretch* property that enables you to specify how the content is resized.

The most interesting values of the *Stretch* property are:

- Uniform (default): Image is resized proportionally, leaving transparent margins on the sides as needed.
- Fill: Image is resized proportionally, filling up the whole space assigned to the *Image* control.

The following code will display a picture resized to be 50 tall (width is automatically computed since it is not provided) and a movie with the same characteristics.

```
<Image Source="fleurs.jpg" Height="50" />
```

```
<MediaElement Source="ic09.wmv" Height="50" />
```

As stated earlier, sizes in WPF are not provided in pixels, since specifying pixels doesn't scale well when the screen resolution increases. Sizes are provided in *device-independent pixels*. If a screen is correctly calibrated, one device-independent pixel is about half a millimeter. This means that *50* represents around 2.5 centimeters on screen. This size would remain the same whatever the screen resolution you chose. Great news: that enables your application to perform well on nowadays' high screen resolutions.

Drawing controls

The *Ellipse*, *Rectangle* and *Path* controls are basic shape drawing controls. They all share common properties:

- Fill: a *Brush* used to paint the inside of the control;
- Strike: a *Brush* used to paint the outline of the control;
- Stretch: how the control should resize its shape when resized, just like we saw for multimedia controls.

The *Path* control is very flexible: it enables you to provide a list of points and have them connected using segments or Bezier curves. Manually providing the points is too tedious so you have two options: draw the shape using *Blend for Visual Studio* or export the shape from a drawing or converter tool that generates XAML.

They are not container control so they can't have a child, but who cares? Should you need to add text to them, you can place a *TextBlock* over them, grouping both in a *Grid* control so they have the same size.

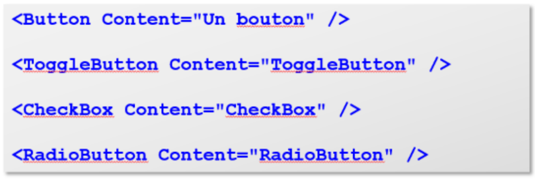
Apart from placing them anywhere on a screen, you can use them inside templates in order to give outstanding new appearances to existing controls. More about that a little later.

Content controls

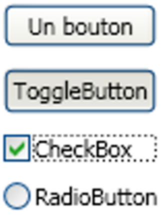
Content controls have a content that can be anything. For this, they expose a *Content* property. The following are content controls:

- Button
- Border
- ScrollViewer
- ViewBox

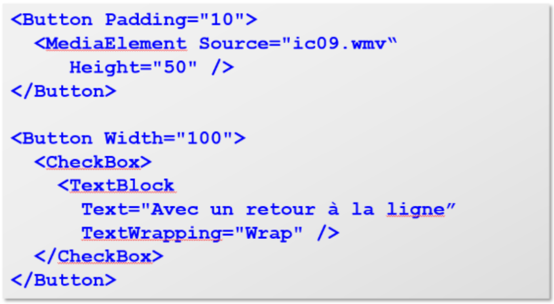
Here are some buttons. Again, their *default* appearance displayed on the right:



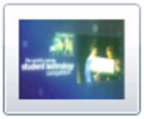
```
<Button Content="Un bouton" />

<ToggleButton Content="ToggleButton" />

<CheckBox Content="CheckBox" />

<RadioButton Content="RadioButton" />
```

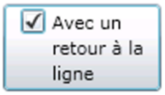


Note that the *Content* property is assigned using a *Content* attribute. That works well with simple content. When you need to assign more complex content, you can provide a child element to your content control instead of using the *Content* attribute. Here are two examples:



```
<Button Padding="10">
  <MediaElement Source="ic09.wmv"
     Height="50" />
</Button>

<Button Width="100">
  <CheckBox>
    <TextBlock
      Text="Avec un retour à la ligne"
      TextWrapping="Wrap" />
  </CheckBox>
</Button>
```

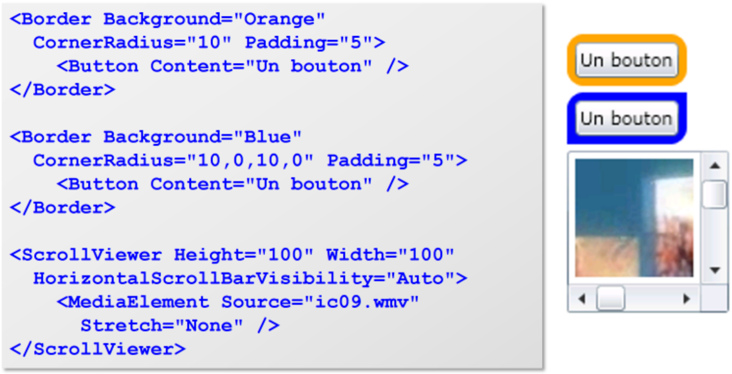




As I wrote, the content can be *anything*. Did you note how the example above adds a checkbox to a button? This simply wasn't possible with frameworks like Windows Forms because the *Button* control didn't have an *EnableCheckBox* property. Using WPF, you can simply combine controls in order to get the functionality you need. Plus you can also change their appearance, as we'll see later.

That gives you a great deal of flexibility. For instance, you may add scrolling around any control by just wrapping it inside a *ScrollViewer* control. Or a border to any control by wrapping it inside a *Border* control: don't look for a *Border* property on e.g. a *TextBlock* control: simply wrap it inside a *Border* control.

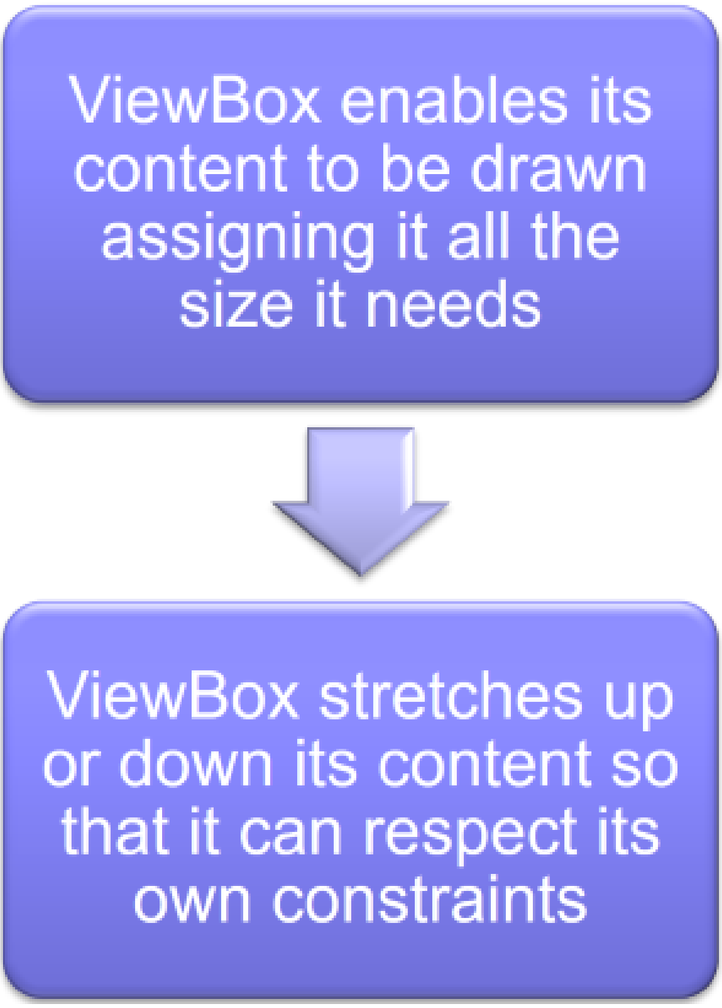
Here are examples of using the *Border* control and adding scrollbars to a movie using the *ScrollViewer* control.



```
<Border Background="Orange"
  CornerRadius="10" Padding="5">
    <Button Content="Un bouton" />
</Border>

<Border Background="Blue"
  CornerRadius="10,0,10,0" Padding="5">
    <Button Content="Un bouton" />
</Border>

<ScrollViewer Height="100" Width="100"
  HorizontalScrollBarVisibility="Auto">
    <MediaElement Source="ic09.wmv"
      Stretch="None" />
</ScrollViewer>
```

Now is time to introduce one of my preferred WPF controls to you: *ViewBox*. I love the *ViewBox* control because it shows the flexibility of WPF. It is able to resize any content just as if it were a picture, and the content remains usable. That means you can quickly have any kind of screen resized to the available width and height. It will come in very handy in control templates and many parts of your application.

Here is how the *ViewBox* control works:

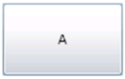

ViewBox enables its content to be drawn assigning it all the size it needs
ViewBox stretches up or down its content so that it can respect its own constraints

Now guess what? The *ViewBox* control has a *Strech* property that states how its content should be resized.

And it behaves exactly like the *Strech* property of *Image* and *MediaElement* controls.

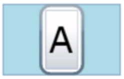
Let me show you simple uses of the *ViewBox* control together with their resulting display.

```
<Grid Height="60" Width="100" Background="LightBlue">
    <Button Content="A" />
</Grid>
```



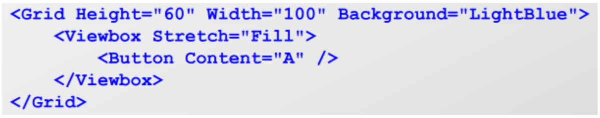
In the above example, there is no *ViewBox* control. As we'll see later, a *Grid* control will stretch its content to fill in all of this space. So the *Button* control takes up all of the *Grid* control size.

```
<Grid Height="60" Width="100" Background="LightBlue">
    <Viewbox>
        <Button Content="A" />
    </Viewbox>
</Grid>
```

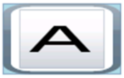


In that second example above, I just inserted a *ViewBox* control between the *Grid* control and the *Button* Control. The *Button* control is thus drawn using the size it needs (since there are no other constraints here, the size necessary to display its text), and then stretched up by the *ViewBox* control in order to fill all of the *Grid* control size. Note how the *Button* borders look thicker: all of the control was proportionally stretched.

Now, let me add just one attribute to the *ViewBox* control we used:



```
<Grid Height="60" Width="100" Background="LightBlue">
    <Viewbox Stretch="Fill">
        <Button Content="A" />
    </Viewbox>
</Grid>
```



Notice the result? The *Button* control is distorted.

Best part is that since *ViewBox* is a content control it can be used in order to resize a full screen. Suppose you have the following screen:

```
<Grid xmlns="...">
  <Button Content="Hello world" ... />
  <ListBox ... />
  <DataGrid ... />
</Grid>
```

You can have that whole screen resize to any dimension just adding a *ViewBox* control:

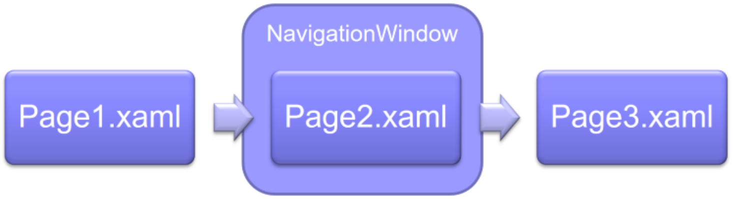
```
<ViewBox xmlns="...">
  <Grid>
    <Button Content="Hello world" ... />
    <ListBox ... />
    <DataGrid ... />
  </Grid>
</ViewBox>
```

This method is quick to implement but has its drawbacks: it resizes all of the content. If you want some more complex resizing like providing more space to the ListBox control, you should use layout controls.

3.5 Navigation

Users are now used to navigating inside an application. Going back to the previous screen, and back again in the history, is likely to be part of your application's requirements. WPF comes in with a navigation framework that may come handy, though you are free to use another one.

When using WPF navigation system, screens are Pages, and they are displayed within a single *Frame* control. Think of the *Frame* control as a Web browser and of the Pages as Web pages.



Pages are XAML files, and you can consider them just like Windows except they have no borders or window-related properties. They are a subclass of user controls, so you could also think of them as user controls. Anyway, in order to create a page you just add a *Page* element using Visual Studio, and get roughly the following XAML:

```
<Page x:Class="..." Title="...">
  <Grid>
      ...
  </Grid>
<Page>
```

You will create as many pages as your application needs screens, and then you'll add a *Frame* control that will serve as the page browser. A natural place to put the *Frame* control is the *MainWindow.xaml* window that has been created by default. Next, you tell the *Frame* control which page to display using the *Source* property.

You get something like that (probably inside *MainWindow.xaml*):

```
<Frame Source="/Welcome.xaml">
<Frame>
```

Don't forget the "/" in front of the page name.

This code would display the Welcome page. Now, you need a way for the user to move from one page to another. You can do so using XAML or C#.

Navigate to another page using code-behind

```
NavigationService.Navigate(
  new Uri("/Payment.xaml", UriKind.Relative)
);
```

Link to another page using XAML

```
<Label>
  <Hyperlink NavigateUri="/Payment.xaml">
    Pay now
  </Hyperlink>
</Label>
```

3.6 It’s your turn to code: do-it-yourself

Now is your turn to grab the keyboard and code away. Oh, just let me explain you how that works, in case you’re not familiar with my *Learn collection* books.

About exercises in this book

All of the exercises are linked together: you’re going to build a small e-commerce application. You’ll allow users to browse through your products, add them to their basket, and you’ll also create a full back-end where the site administrators will be able to list, create, modify, and delete products.

In case you get stuck

You should be able to solve the exercise all by yourself. If you get stuck or don't have a computer at hand (or you don't have the prerequisites for that book, which is fine with me!), no problem. I'll provide the solution for all of the exercises in this book, right after each of them.

3.7 Exercise - Create the application and contact page

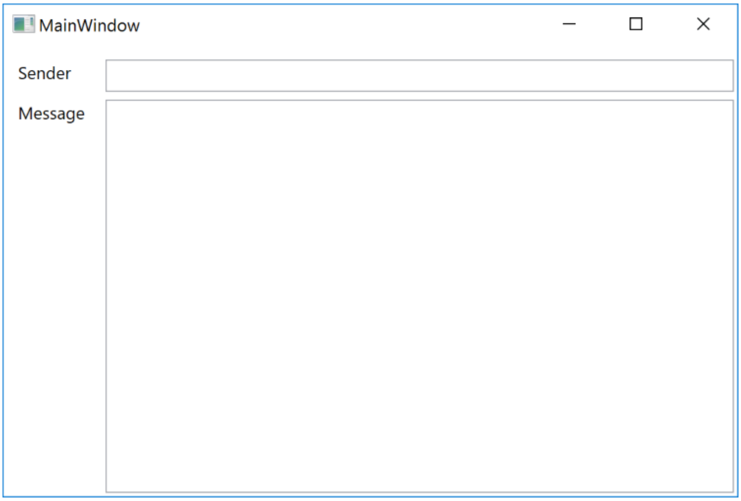
Create a new WPF application named *BikeShop*.

Add a new page named *Contact.xaml* to the application.

Add two *TextBox* controls and two *TextBlock* controls to the *Contact* page so that a user can input a message.

Make sure that the *Contact* page is displayed by default on the *MainWindow.xaml* screen

Your application should look like the following:



I know, it's basic, but you need to learn some more things before you can do more.

Beginner badge unlocked: let's proceed to the next level.

3.8 Exercise solution

- Start Visual Studio.
- Click on the *File / New / Project...* menu entry.
- In the *New Project* dialog box, select the *WPF Application* template making sure that you select *Templates / Visual C# / Windows* on the left-hand side. In the *Name* zone at the bottom, type "BikeShop". Click the *OK* button.
- Open the *Solution Explorer* clicking on the *View / Solution Explorer* menu entry.

- In the *Solution Explorer*, right-click the project (not the solution), and select *Add / Page* from the context menu.
- In the *Add New Item* dialog box, look for the *Name* zone at the bottom, and type "Contact". Click the *Add* button.
- Open the *Toolbox* clicking on the *View / Toolbox* menu entry.
- Drag and drop two *TextBlock* controls and two *TextBox* controls from the toolbox to the design surface. Position them and resize them so that the screen looks as expected.
- Make sure that the *Properties* window is displayed clicking on the *View / Properties Window* menu entry.
- Click the first *TextBlock* control and change its *Text* property to *Sender*.
- Click the second *TextBlock* control and change its *Text* property to *Message*.
- Click the first *TextBox* control and change its *Text* property to be an empty string.
- Click the second *TextBox* control and change its *Text* property to be an empty string.
- In the *Solution Explorer*, double-click the *MainWindow.xaml* file.
- Inside the *Grid* element, add a *Frame* element. The *MainWindow.xaml* code should look like this:

```
<Window ...>
  <Grid>
    <Frame Source="/Contact.xaml" />
  </Grid>
</Window>
```

- Run the application (click on the *Debug / Start Debugging* menu entry).
- Close the application.

3.9 Understanding XAML

XAML namespaces

Object creation

This is just a sample of the full book.

If you like it, get your full version here: https://leanpub.com/learnwpf

Properties definition

This is just a sample of the full book.

If you like it, get your full version here: https://leanpub.com/learnwpf

Naming

This is just a sample of the full book.

If you like it, get your full version here: https://leanpub.com/learnwpf

3.10 Events

3.11 Exercise - Create the menu page

3.12 Exercise solution

3.13 Layout

Why our screens don't resize

Size allocation

Panel controls

Canvas

StackPanel

DockPanel

WrapPanel

UniformGrid

Grid

Summary of panel controls

3.14 List controls

Selection controls

This is just a sample of the full book.

If you like it, get your full version here: https://leanpub.com/learnwpf

3.15 Exercise - Create the discussion page

This is just a sample of the full book.

If you like it, get your full version here: https://leanpub.com/learnwpf

3.16 Exercise solution

This is just a sample of the full book.

If you like it, get your full version here: https://leanpub.com/learnwpf

4. Managing data in a WPF application

4.1 Data binding

This is just a sample of the full book.

If you like it, get your full version here: https://leanpub.com/learnwpf

Binding examples

This is just a sample of the full book.

If you like it, get your full version here: https://leanpub.com/learnwpf

Binding Mode

This is just a sample of the full book.

If you like it, get your full version here: https://leanpub.com/learnwpf

Extra properties

This is just a sample of the full book.

If you like it, get your full version here: https://leanpub.com/learnwpf

Binding errors

This is just a sample of the full book.

If you like it, get your full version here: https://leanpub.com/learnwpf

4.2 DataContext

This is just a sample of the full book.

If you like it, get your full version here: https://leanpub.com/learnwpf

4.3 Converters

4.4 Displaying collections using list controls

4.5 Customizing list controls

4.6 Exercise - Display messages from a data object

4.7 Exercise solution

4.8 INotifyPropertyChanged

4.9 INotifyCollectionChanged

5. Making it shine: customize the look

5.1 Change a control's look

Template

TemplateBinding

ItemsPresenter

5.2 Resources

ResourceDictionaries

5.3 Exercise - Set the background

This is just a sample of the full book.

If you like it, get your full version here: https://leanpub.com/learnwpf

5.4 Exercise solution

This is just a sample of the full book.

If you like it, get your full version here: https://leanpub.com/learnwpf

5.5 Styles

This is just a sample of the full book.

If you like it, get your full version here: https://leanpub.com/learnwpf

5.6 Themes

5.7 Transforms

5.8 Control states

5.9 Animations

This is just a sample of the full book.

If you like it, get your full version here: https://leanpub.com/learnwpf

6. MVVM pattern for WPF

6.1 Spaghetti code

This is just a sample of the full book.

If you like it, get your full version here: https://leanpub.com/learnwpf

6.2 MVC

This is just a sample of the full book.

If you like it, get your full version here: https://leanpub.com/learnwpf

6.3 MVVM

6.4 Recommended steps (simple)

6.5 Example

6.6 Example, more complex

6.7 Commands and methods

Commands: the apparently easy way

Methods: the straightforward way

6.8 Recommended steps (complete)

6.9 Exercise - Display products and details using MVVM

6.10 Exercise solution

6.11 MVVM frameworks in short

A word from the author

I sincerely hope you enjoyed reading this book as much as I liked writing it and that you quickly become proficient enough with WPF and the MVVM pattern.

If you would like to get in touch you can use :

- email: books@aweil.fr
- Facebook: https://facebook.com/learncollection
- Twitter: @epo

In case your project needs it, I'm also available for speaking, teaching, consulting and coding, all around the world.

If you liked this book, you probably saved a lot of time thanks to it. I'd be very grateful if you took some minutes of your precious time to leave a comment on the site where you purchased this book. Thanks a ton!