Understanding In and Out of XAML in WPF

1. What is XAML? Extensible Application Markup Language and pronounced “zammel” is a markup language used to instantiate .NET objects. Although XAML is a technology that can be applied to many different problem domains, its primary role in life is to construct WPF user interfaces.

If we’re a graphic designer, that tool is likely to be a graphical design and drawing program such as Microsoft Expression Blend. If we’re a developer, we’ll probably start with Visual Studio.

2. Why to learn XAML: Understanding XAML is critical to WPF application design. WPF applications are quite different from Windows Forms applications in this respect—with Windows Forms applications, we could safely ignore the automatically generated UI code, while in WPF applications the XAML often takes center stage. Important task they perform are below

a. Wiring up an event handler: Attaching event handler in the most cases, for example Click on Button is easy to do in visual studio. However once we understand how events are wired up in XAML,we’ll be able create more sophisticated connections.

b. Defining resources: Resources are the object which once we define in XAML can be re-used in the various places inside markup. Resources allow us to centralize and standardize formatting, and create nonvisual objects such as templates and animations.

c. Defining control template: WPF controls are designed to be *lookless*, which means we can substitute our custom visuals in place of the standard appearance. To do so, we must create our own control template, which is nothing more than a block of XAML markup

d. Writing data binding expressions: data binding allows us to extracts data from a object and display it in a linked element. To setup this we can add a data binding expression in XAML.

e. Defining animation: Animations are a common ingredient in XAML applications. Usually, they’re defined as resources, constructed using XAML markup, and then linked to other controls (or triggered through code).

3. Life before XAML: With traditional display technologies, there’s no easy way to separate the graphical content from the code. The key problem with Windows Forms application is that every form we create

Is defined entirely in C# code. As we drop controls onto the design surface and configure them, Visual Studio quietly adjusts the code in the corresponding form class. Sadly, graphic designers don't have any tools that can work with C# code.

Then solution to above problem was that the designer forced to take their content and export it to a bitmap format. These bitmaps can then be used to skin windows, buttons, and other controls. This approach works well for straightforward interfaces that don't change much over time, but it's extremely limiting in other scenarios.

Even if the interface is designed from scratch by a graphic designer, we'll need to re-create it with C# code. Usually, the graphic designer will simply prepare a mock-up that we need to translate painstakingly into our application.

Solution to above issue is XAML: When designing a WPF application in Visual Studio*, the window we're designing isn't translated into code. Instead, it's serialized into a set of XAML tags. When we run the application, these tags are used to generate the objects that compose the user interface.*

4. Type of XAML: The way XAML is been used in reference with are many, which is nothing but is an all-purpose XML-based syntax for representing a tree of .NET objects and these objects can be button in a window or a custom class we have defined. However XAML could be used on the other platform to re-present non .NET objects.
Below I have discussed about subsets of XAML
a. WPF XAML: encompasses the elements that describe WPF content, such as vector graphics, controls, and documents
b. XPS XAML: is the part of WPF XAML that defines an XML representation for formatted electronic documents.
c. Silverlight XAML: is a subset of WPF XAML that's intended for Silverlight applications. Silverlight is a cross-platform browser plug-in that allows us to create rich web content.
d. WF XAML: encompasses the elements that describe Windows Workflow Foundation.

5. Understanding XMAL compilation: Solving the pain for designed cannot be a sole reason to be happy about XAML, well it also needs to be fast in performance. XAML achieves this with BAML (binary application markup language), which is binary representation of XMAL.

What happen when we compile XAML in visual studio, all XAML is converted into BAML and that BAML is then embedded as resource inside the final DLL or EXE assembly. BAML is *tokenized*, which means lengthier bits of XAML are replaced with shorter tokens. Not only is BAML significantly smaller, it's also optimized in a way that makes it faster to parse at runtime.

So do we require or is it must that this compilation happens, well no. because there might be an scenario where it is required some of the user interface to be supplied just in time, so here possible to use XAML without compiling it.

6. Knowing XAML Skelton:

Basic:

a. Every element in a XAML document maps to an instance of a .NET class. The name of the element matches the name of the class *exactly*. For example, the element <Button> instructs WPF to create a Button object.

b. As with any XML document, we can nest one element inside another. A button element inside a Grid element, our user interface probably includes a Grid that contains a button inside. However, Nesting is usually a way to express containment

c. We can set the properties of each class through attributes

Create a blank WPF page, and we will notice that visual studio adds up few lines of code, lets us understand them.

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

    </Grid>
</Window>
```

a. This XAML includes only two elements—the top-level Window element, which represents the entire window, and the Grid, in which we can place all our controls. Although we could use any top-level element, WPF applications rely on just a few:

- Window
- Page (which is similar to Window, but used for navigable applications)
- Application (which defines application resources and startup settings)

b. In all XML documents, there can only be one top-level element. In the previous example, that means that as soon as we close the Window element with the </Window> tag, we end the document. No more content can follow.

Namespace:

It's not enough to supply just the name, XAML parser also needs to know the .Net namespace where this class is located. As in window can exist in several places, it might refer to system.windows.window class or it could refer to a window class in a third party component. To figure out which class we really want, the XAML parser examines the XML namespace that's applied to the element.

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

XMLNS: The xmlns attribute is a specialized attribute in the world of XML that's reserved for declaring namespaces. This snippet of markup declares two namespaces that we'll find in every WPF XAML document we create:

`xmlns=http://schemas.microsoft.com/winfx/2006/xaml/presentation` Is the core WPF namespace. It encompasses all the WPF classes, including the controls we use to build user Interfaces.

`xmlns:x=http://schemas.microsoft.com/winfx/2006/xaml` It includes various XAML utility features that allow we to influence how our document is interpreted. This namespace is mapped to the prefix *x*.

By convention, XML namespaces are often URIs (as they are here). These URIs look like they point to a location on the Web, but they don't. It is just a naming convention to avoid any conflict.
The namespace information allows the XAML parser to find the right class. For example, when it looks at the Window and Grid elements, it sees that they are placed in the default WPF namespace. It then searches the corresponding .NET namespaces, until it finds System.Windows. Window and System.Windows.Controls.Grid.

Code behind Class:

XAML allows we to construct a user interface, but in order to make a functioning application

We need a way to connect the event handlers that contain our application code. XAML makes this easy using the Class attribute that's shown here:

```
<Window x:Class="TireControl.Window1"
```

The ***x* namespace** prefix places the Class attribute in the XAML namespace, which means this is a more general part of the XAML language. In fact, the Class attribute tells the XAML parser to generate a new class with the specified name. The Window1 class is generated automatically at compile time

Visual Studio helps us out by automatically creating a partial class where we can place our event handling code.

```
namespace TireControl
{
    /// <summary>
    /// Interaction logic for Window1.xaml
    /// </summary>
    public partial class Window1 : Window
    {
        public Window1()
        {
            InitializeComponent();
        }
    }
}
```

When we compile the application, XAML that defines our user interface as such window1.xaml is translated into CLR tyre declaration that is merged with logic in the code behind window1.xaml.cs to form the single unit.

The InitializeComponent() Method:

Window1 .cs contains a default constructor, which call this method, when we create new instance of the class. The InitializeComponent()method plays a key role in WPF applications. For that reason, we should never delete the InitializeComponent() call in our window's constructor. Similarly, if we add another constructor, make sure it also calls InitializeComponent().

It work like this way that when we compile our application essentially, all InitializeComponent() does is call the **LoadComponent**() method of the System.Windows.Application class. The LoadComponent () method extracts the BAML (the compiled XAML) from our assembly and uses it to build our user interface. As it parses the BAML, it creates each control object, sets its properties, and attaches any event handlers.

7. Ways to create a WPF application: There are three distinct coding styles that we can use to create a WPF application.

a. **Code-only.** This is the traditional approach used in Visual Studio for Windows Forms applications. It generates a user interface through code statements. Code-only development is a less common (but still fully supported) avenue for writing a WPF application without any XAML. The obvious disadvantage to code-only development is that it has the potential to be extremely tedious.

Let's write little code in code behind to understand this; add a class and not an existing item template.

[To create this example, we must code the Window1 class from scratch (right-click the Solution Explorer, and choose Add ➤ Class to get started). We can't choose Add ➤ Window, because that will Add a code file and a XAML template for our window, complete with an automatically generated Initialize-Component () method]

```
public partial class Window1 : Window
    {
        private Button button1;
        public Window1()
        {
            InitializeComponent();
            // Configure the form.
            this.Width = this.Height = 285;
            this.Left = this.Top = 100;
            this.Title = "Code-Only Window";
            // Create a container to hold a button.
            DockPanel panel = new DockPanel();
            // Create the button.
            button1 = new Button();
            button1.Content = "Please click me.";
            button1.Margin = new Thickness(30);
            // Attach the event handler.
            button1.Click += button1_Click;
            // Place the button in the panel.
            IAddChild container = panel;
            container.AddChild(button1);
            // Place the panel in the form.
            container = this;
            container.AddChild(panel);

        }
        private void button1_Click(object sender, RoutedEventArgs e)
        {
            button1.Content = "Thank you.";
        }
```

```
    }
```

To get this application started, we can use a Main() method with code like this:

```
public class Program : Application
    {
        [STAThread()]
        static void Main()
        {
            Program app = new Program();
            app.MainWindow = new Window1();
            app.MainWindow.ShowDialog();
        }
    }
```

b. Code and uncompiled markup (XAML). This is a specialized approach that makes sense in certain scenarios where we need highly dynamic user interfaces. We load part of the user interface from a XAML file at runtime using the **XamlReader** class from the System.Windows.Markup namespace.
One of the most interesting ways to use XAML is to parse it on the fly with the **XamlReader**. It's like at runtime, we can load this content into a live window to create the same window.

c. **Code and compiled markup (BAML).** This is the preferred approach for WPF, and the one that Visual Studio supports. We create a XAML template for each window and this XAML is compiled into BAML and embedded in the final assembly. At runtime the compiled BAML is extracted and used to regenerate the user interface. This is the method used by Visual Studio, and it has several advantages.
a. Reading BAML at runtime is faster than reading XAML.
b. Deployment is easier. Because BAML is embedded in our assembly as one or more resources, there's no way to lose it.

Let us understand how **compilations** happen.
Visual Studio uses a two-stage compilation process when we're compiling a WPF application.

Step 1*. The first step is to compile the XAML files into BAML using the* ***xamlc.exe*** *compiler. For example, if our project includes a file name Window1.xaml, the compiler will create a temporary file named Window1.baml and place it in the obj\Debug subfolder (in our project folder). At the same time, a partial class is created for our window, using the language of our choice. For example, if we're using C#, the compiler will create a file named Window1.g.cs in the obj\Debug folder. The g stands for generated.*

The partial class includes three things:

• Fields for all the controls in our window.

• Code that loads the BAML from the assembly, thereby creating the tree of objects. This happens when the constructor calls Initialize Component ().

• Code that assigns the appropriate control object to each field and connects all the event handlers. This happens in a method named Connect (), which the BAML parser calls every time it finds a named object.

Step 2::*When the XAML-to-BAML compilation stage is finished, Visual Studio uses the appropriate language compiler to compile our code and the generated partial class files. In the case of a C# application, it's the csc.exe compiler that handles this task. The compiled code becomes a single assembly Window1.exe) and the BAML for each window is embedded as a separate resource.*

8 What is Loose XAML: Loose XAML files can be opened directly in Internet Explorer? (Assuming we've installed the .NET Framework 3.0 or are running Windows Vista, which has it preinstalled?)

To try out a loose XAML page, take a .xaml file and make these changes:

• Remove the Class attribute on the root element.

• Remove any attributes that attach event handlers (such as the Button.Click attribute).

• Change the name of the opening and closing tag from Window to Page. Internet Explorer can only show hosted pages, not stand-alone windows.

Thanks for reading

http://www.vishalnayan.wordpress.com