PXL IT

XAML in Xamarin.Forms

DE HOGESCHOOL
MET HET NETWERK

Hogeschool PXL – Elfde-Liniestraat 24 – B-3500 Hasselt
www.pxl.be - www.pxl.be/facebook

Agenda

- What is XAML?
- Building an app with XAML
- XAML Syntax
- XAML Markup Extensions
- Data binding
- Advanced topics in Xamarin.Forms

What is XAML?

- eXtensible Application markup Language
  - XML-based Markup
    - Describes both Look and Feel
    - Typically will be created with tools
  - Pronounced “zammel”, not XAMMEL!
- It is a markup language used to instantiate .NET objects
  - Everything you declare in XAML will be instantiated as a .NET object in the background
- Can be applied to many different problem domains, originally only WPF
  - WPF, Windows Phone, Silverlight, UWP (Universal Windows 10)

Disadvantages of XAML

- XAML cannot contain code
  - All event handlers must be defined in a code file.
- XAML cannot contain loops for repetitive processing
  - However, several Xamarin.Forms visual objects—mostly notably ListView —can generate multiple children based on the objects in its ItemsSource collection
- XAML cannot contain conditional processing
  - However, a data-binding can reference a code-based binding converter that effectively allows some conditional processing
- XAML generally cannot instantiate classes that do not define a parameterless constructor
  - However there is sometimes a way around this restriction
- XAML generally cannot call methods
  - Again, this restriction can sometimes be overcome

PXL IT

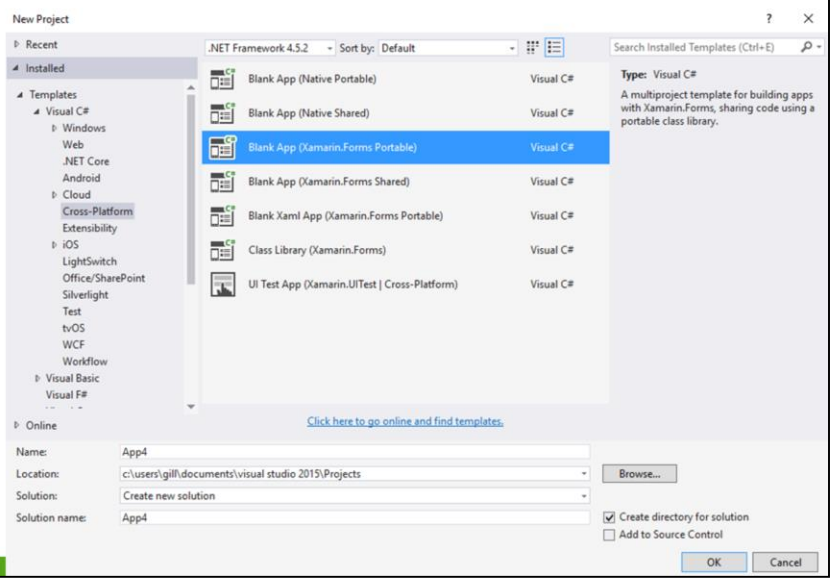
Building an app with XAML

DE HOGESCHOOL
MET HET NETWERK

Hogeschool PXL – Elfde-Liniestraat 24 – B-3500 Hasselt
www.pxl.be - www.pxl.be/facebook

Creating a XAML-based application

- Same template



Select a location for the solution and give it a name of XamlSamples (or whatever).

Visual Studio creates four projects: XamlSamples.Android, XamlSamples.iOS, XamlSamples.WinPhone, and a shared Portable Class Library (PCL) project named simply XamlSamples.

When using Xamarin Studio on the PC, only the XamlSamples.Android and XamlSamples projects are created. Xamarin Studio on the Mac also creates the XamlSamples.iOS project.

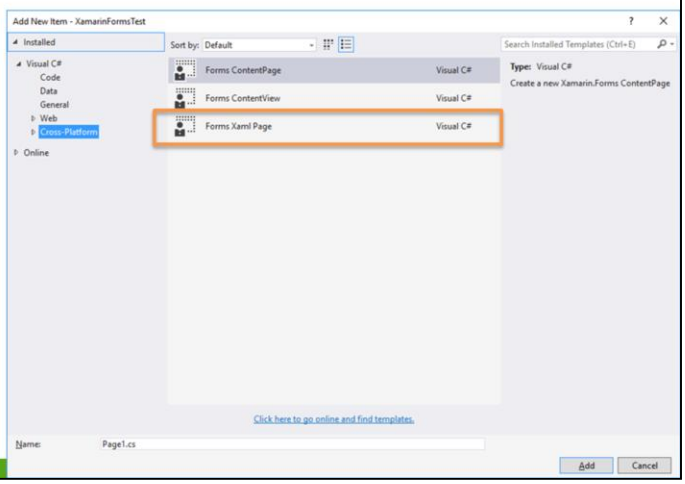
After creating the Xamarin.Forms solution, you might want to test your development environment by selecting the various platform projects as the solution startup project, and building and deploying the simple application created by the project template on either phone emulators or real devices.

Unless you need to write platform-specific code, the shared XamlSamples PCL project is where you'll be spending virtually all of your programming time.

XAML can play a role in a Xamarin.Forms application in several ways, but undoubtedly the most common is for defining the visual contents of an entire page, which is usually a class derived from ContentPage.

Creating a XAML-based application

- Add a XAML file (XAML page)
  - Also creates the code-behind



Generated code

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="XamarinFormsTest.Page1">
  <Label Text="{Binding MainText}" VerticalOptions="Center"
           HorizontalOptions="Center" />
</ContentPage>
```

- xmlns namespace declarations
- x:Class declaration
- ContentPage

PXL IT

The two XML namespace (xmlns) declarations refer to URIs, the first seemingly on Xamarin's web site and the second on Microsoft's. Don't bother checking what those URIs point to. There's nothing there. They are simply URIs owned by Xamarin and Microsoft, and they basically function as version identifiers.

The first XML namespace declaration means that tags defined within the XAML file with no prefix refer to classes in Xamarin.Forms, for example ContentPage. The second namespace declaration defines a prefix of x. This is used for several elements and attributes that are intrinsic to XAML itself and which (in theory) are supported by all implementations of XAML. However, these elements and attributes are slightly different depending on the year embedded in the URI. Xamarin.Forms supports the 2009 XAML specification, but not all of it.

Immediately after the x prefix is declared, that prefix is used for an attribute named Class. Because the use of this x prefix is pretty much universal in XAML files, XAML attributes such as Class are almost always referred to as x:Class.

The x:Class attribute specifies a fully qualified .NET class name: the HelloXamlPage class in the XamlSamples namespace. This means that this XAML file defines a new class named HelloXamlPage in the XamlSamples namespace that derives from ContentPage—the tag in which the x:Class

attribute appears.

The x:Class attribute can only appear in the root element of a XAML file to define a derived C# class. This is the only new class defined in the XAML file. Everything else that appears in the XAML file is instead simply instantiated and initialized.

Generated code-behind

- Partial class: partial with the generated file
- Derives from ContentPage
- InitializeComponent()

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

using Xamarin.Forms;

namespace XamarinFormsTest
{
    public partial class Page1 : ContentPage
    {
        public Page1()
        {
            InitializeComponent();
        }
    }
}
```

Notice the partial class definition. Although it's not indicated explicitly in this class definition, HelloXamlPage derives from ContentPage.

But there seems to be something missing. Shouldn't there be another C# file with another partial class definition for HelloXamlPage? And what is that InitializeComponent method? You'll see those shortly.

In the App.cs file in the XamlSamples project, you'll want to remove some of the existing code and use the static App.GetMainPage method to simply instantiate and return HelloXamlPage:

Generated App.cs

- Must change to

```
namespace XamlSamples
{
    public class App
    {
          public App()
          {
             // The root page of your application
             MainPage = new Page1();
          }
}
```

Generated *.xaml.g.cs

- Contains InitializeComponent() method, called from our code-behind
  - Both are partial classes

```
namespace XamarinFormsTest {
    using System;
    using Xamarin.Forms;
    using Xamarin.Forms.Xaml;

    public partial class Page1 : global::Xamarin.Forms.ContentPage {

        [System.CodeDom.Compiler.GeneratedCodeAttribute("Xamarin.Forms.Build.Tasks.XamlG", "0.0.0.0")]
        private void InitializeComponent() {
            this.LoadFromXaml(typeof(Page1));
        }
    }
}
```

PXL IT

Explain link here between all these files

This is the other partial class definition of HelloXamlPage, and it explicitly indicates that the base class is ContentPage. This class definition also contains the definition of the InitializeComponent method called from the HelloXamlPage constructor. During the build process, this code file is first generated from the XAML file, and then the two partial class definitions of HelloXamlPage are compiled together.

At runtime, code in the particular platform project calls the static App.GetMainPage method to get the initial page. This method instantiates HelloXamlPage. The constructor of that class calls InitializeComponent, which then calls the LoadFromXaml method that extracts the entire XAML file from the Portable Class Library and parses it, instantiates and initializes all the objects defined in the XAML file, connects them all together in parent-child relationships, attaches event handlers defined in code to events set in the XAML file, and sets the resultant tree of objects as the content of the page.

Although you normally don't need to spend much time with generated code files, sometimes runtime exceptions are raised on code in the generated files, so you should be familiar with them.

The parsing of the XAML file during the build process is rudimentary compared with the later parsing at runtime. The parsing at build time reveals XML syntax errors but not incorrectly spelled elements or attributes. Problems of that sort

will only be detected at runtime. Fortunately, the runtime exceptions usually provide sufficient information to locate and fix the problem.

Adding content to the pages

- All UI code we've seen so far can be edited from XAML too
  - All element declarations are resulting in an object being instantiated
  - Attributes translate into properties

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="XamlSamples.HelloXamlPage"
             Title="Hello XAML Page"
             Padding="10, 40, 10, 10">

 <Label Text="Hello, XAML!"
       VerticalOptions="Start"
       HorizontalTextAlignment="Center"
       Rotation="-15"
       IsVisible="true"
       FontAttributes="Bold"
       TextColor="Aqua" />

</ContentPage>
```

TypeConverters

- Many properties are basic data types
  - Text is of type String
  - Rotation is of type Double
  - XAlign is an enumeration
- Some conversions are more complex
  - Need TypeConverter when parsing XAML
- Examples
  - ThicknessTypeConverter for the Padding property
    - Can handle 1, 2 or 4 comma separated numbers
  - LayoutOptionsConverter for the VerticalOptions property
  - FontTypeConverter for the Font property
  - ColorTypeConverter for the TextColor property

PXL IT

At this time, the relationship between classes, properties, and XML should be obvious: A Xamarin.Forms class (such as ContentPage or Label) appears in the XAML file as an XML element, and properties of that class—including Title and Padding on ContentPage and seven properties of Label—appear as XML attributes.

Many shortcuts exist to set the values of these properties. Some properties are basic data types: For example, the Title and Text properties are of type String, Rotation is of type Double, and IsVisible (which is true by default and is set here only for illustration) is of type Boolean.

The XAlign property is of type TextAlignment, which is an enumeration. For a property of any enumeration type, all you need supply is a member name.

For properties of more complex types, however, converters are used for parsing the XAML. These are classes in Xamarin.Forms that derive from TypeConverter. Many are public classes but some are not. For this particular XAML file, several of these classes play a role behind the scenes:

- ThicknessTypeConverter for the Padding property
- LayoutOptionsConverter for the VerticalOptions property
- FontTypeConverter for the Font property
- ColorTypeConverter for the TextColor property

Adding code in the code behind

- We can link event handlers to events in the UI

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="XamlSamples.XamlPlusCodePage"
             Title="XAML + Code Page">
  <StackLayout>
    <Slider VerticalOptions="CenterAndExpand"
            ValueChanged="OnSliderValueChanged" />

    <Label Text="A simple Label"
           FontAttributes="Bold"
           HorizontalOptions="Center"
           VerticalOptions="CenterAndExpand" />

    <Button Text="Click Me!"
            HorizontalOptions="Center"
            VerticalOptions="CenterAndExpand"
            Clicked="OnButtonClicked" />
  </StackLayout>
</ContentPage>
```

PXL IT

The HelloXamlPage sample contains only a single Label on the page, but this is very unusual. Most ContentPage derivatives set the Content property to a layout of some sort, such as a StackLayout. The Children property of the StackLayout is defined to be of type IList<View> but it's actually an object of type ElementCollection<View>, and that collection can be populated with multiple views or other layouts. In XAML, these parent-child relationships are established with normal XML hierarchy. Here's a XAML file for a class named XamlPlusCodePage.

Code behind

```
namespace XamlSamples
{
    public partial class XamlPlusCodePage
    {
        public XamlPlusCodePage()
        {
            InitializeComponent();
        }

        void OnSliderValueChanged(object sender,
                                  ValueChangedEventArgs args)
        {

        }

        void OnButtonClicked(object sender, EventArgs args)
        {

        }
    }
}
```

However, it is probably deficient in functionality. It is very likely that manipulating the Slider is supposed to cause the Label to display the current value, and the Button is probably intended to do something within the program.

As you'll see in Part 4. Data Binding Basics, the job of displaying a Slider value using a Label can be handled entirely in XAML with a data binding. But it is useful to see the code solution first. Even so, handling the Button click definitely requires code. This means that the code-behind file for XamlPlusCodePage must contain handlers for the ValueChanged event of the Slider and the Clicked event of the Button. Let's add them:

Code behind

```
async void OnButtonClicked(object sender, EventArgs args)
{
    Button button = (Button)sender;
    await DisplayAlert("Clicked!",
        "The button labeled '" + button.Text + "' has been clicked",
        "OK");
}
```

The method is defined as async because the DisplayAlert method is asynchronous and should be prefaced with the await operator, which returns when the method completes. Because this method obtains the Button firing the event from the sender argument, the same handler could be used for multiple buttons.

Simple XAML Xamarin.Forms apps

DEMO

PXL IT

XAML Syntax

DE HOGESCHOOL
MET HET NETWERK

Hogeschool PXL – Elfde-Liniestraat 24 – B-3500 Hasselt
www.pxl.be - www.pxl.be/facebook

Property Elements

- Properties are normally set as XML attributes

```
<Label Text="Hello, XAML!"
       VerticalOptions="Center"
       Font="Bold, Large"
       TextColor="Aqua" />
```

- Alternative way: property-element syntax
  - Sometimes optional, sometimes required

```
<Label Text="Hello, XAML!"
       VerticalOptions="Center"
       FontAttributes="Bold">
    <Label.TextColor>
      Aqua
    </Label.TextColor>
</Label>
```

PXL IT

These two ways to specify the TextColor property are functionally equivalent, but you can't use both ways for the same property because that would effectively be setting the property twice.

With this new syntax, some handy terminology can be introduced:

Label is an *object element* . It is a Xamarin.Forms object expressed as an XML element.

Text , VerticalOptions , and Font are *property attributes* . They are Xamarin.Forms properties expressed as XML attributes.

In that final snippet, TextColor has become a *property element* . It is a Xamarin.Forms property but it is now an XML element.

Property Elements

- Can be used on more than one property at the same time

```
<Label Text="Hello, XAML!"
       VerticalOptions="Center">
     <Label.FontAttributes>
       Bold,
     </Label.FontAttributes>
     <Label.TextColor>
       Aqua
     </Label.TextColor>
</Label>
```

Property elements

- When the value is more than a simple string, it becomes required to use this syntax

```
<Label>
      <Label.Text>
        Hello, XAML!
      </Label.Text>
      <Label.FontAttributes>
        Bold, Large
      </Label.FontAttributes>
      <Label.TextColor>
        Aqua
      </Label.TextColor>
      <Label.VerticalOptions>
        <LayoutOptions Alignment="Center" />
      </Label.VerticalOptions>
</Label>
```

PXL IT

However, property-element syntax becomes essential when the value of a property is too complex to be expressed as a simple string. Within the property-element tags you can instantiate another object and set its properties. For example, you can explicitly set a property such as **VerticalOptions** to a **LayoutOptions** value with property settings:

Property Elements

- Also in grid definitions, this is required to be used

```
<Grid>
    <Grid.RowDefinitions>
      <RowDefinition Height="Auto" />
      <RowDefinition Height="*" />
      <RowDefinition Height="100" />
    </Grid.RowDefinitions>

    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="Auto" />
      <ColumnDefinition Width="*" />
      <ColumnDefinition Width="100" />
    </Grid.ColumnDefinitions>
```

Another example: The Grid has two properties named RowDefinitions and ColumnDefinitions. These two properties are of type RowDefinitionCollection and ColumnDefinitionCollection, which are collections of RowDefinition and ColumnDefinition objects. Unless there is a syntax to define all the rows and column dimensions in the collection as a single string—which is surely possible but hasn't been done—you'll need property element syntax.

Here's the beginning of the XAML file for a GridDemoPage class, showing the property element tags for the RowDefinitions and ColumnDefinitions collections:

Platform-specific values

- What if the values are to be different for different platforms?
  - Can only be solved using Property Element syntax as well

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="XamlSamples.GridDemoPage"
             Title="Grid Demo Page">

  <ContentPage.Padding>
    0, 20, 0, 0
  </ContentPage.Padding>

  ...

</ContentPage>
```

PXL IT

This XAML fragment has another potential use for property elements. The Padding on the ContentPage tag is set to 20 units on the top, but that's only to avoid overlapping the status bar on the iPhone. That padding isn't required on Android and Windows Phone. (Nor is it required when the page is navigated to through a NavigationPage as the pages in XamlSamples are, but let's continue as if this were a standalone page in a single-page application.)

Fortunately there is a way to embed some platform-specific markup in a XAML file using a class named OnPlatform<T>. This is a generic class that has three properties named iOS, Android, and WinPhone of type T. The OnPlatform<T> class also defines an implicit cast of itself to type T that returns the appropriate object depending on which platform it's running on.

It sounds complicated but the XAML syntax is actually quite straightforward. First, separate out the Padding as a property element of the ContentPage class:

Platform-specific values

- Solves the problem of different spacing at the top of the screen

```
<ContentPage.Padding>
   <OnPlatform x:TypeArguments="Thickness">
     <OnPlatform.iOS>
       0, 20, 0, 0
     </OnPlatform.iOS>
     <OnPlatform.Android>
       0, 0, 0, 0
     </OnPlatform.Android>
     <OnPlatform.WinPhone>
       0, 0, 0, 0
     </OnPlatform.WinPhone>
   </OnPlatform>
</ContentPage.Padding>
```

Platform-specific values

```
<ContentPage.Padding>
   <OnPlatform x:TypeArguments="Thickness"
               iOS="0, 20, 0, 0"
               Android="0, 0, 0, 0"
               WinPhone="0, 0, 0, 0" />
</ContentPage.Padding>
```

- If the value is only required on one platform, we can omit the others

```
<ContentPage.Padding>
   <OnPlatform x:TypeArguments="Thickness"
               iOS="0, 20, 0, 0" />
</ContentPage.Padding>
```

PXL IT

If the content of the individual property elements can be represented by simple strings (as these can) you can instead define them as property attributes:

Attached Properties

- Sometimes, we need to set properties from one control on another one
  - "Label needs to go in row #1 of the surrounding Grid"
  - Can be solved with Attached Properties

```
<Label Text="Autosized cell"
          Grid.Row="0" Grid.Column="0"
          TextColor="White"
          BackgroundColor="Blue" />

   <BoxView Color="Silver"
            HeightRequest="0"
            Grid.Row="0" Grid.Column="1" />
```

PXL IT

As you've seen, the Grid requires property elements for the RowDefinitions and ColumnDefinitions collections to define the rows and columns. However, there must also be some way for the programmer to indicate the row and column where each child of the Grid resides.

Within the tag for each child of the Grid you specify the row and column of that child using the following attributes:

Grid.Row

Grid.Column

The default values of these attributes are 0. You can also indicate if a child spans more than one row or column with these attributes:

Grid.RowSpan

Grid.ColumnSpan

These two attributes have default values of 1.

Here's the complete GridDemoPage.xaml file:

Attached properties

- Grid defines itself 4 “global” (attached) properties
  - RowProperty
  - ColumnProperty
  - RowSpanProperty
  - ColumnSpanProperty
  - →attached properties
  - Defined on Grid, used by its children
- Attached properties always are of the form <class>.<property>
  - Class is not the type on which it is being used

PXL IT

Judging solely from the syntax, these Grid.Row, Grid.Column, Grid.RowSpan, and Grid.ColumnSpan attributes appear to be static fields or properties of Grid, but interestingly enough, Grid does not define anything named Row, Column, RowSpan, or ColumnSpan.

Instead, Grid defines four bindable properties named RowProperty, ColumnProperty, RowSpanProperty, and ColumnSpanProperty. These are special types of bindable properties known as *attached properties*. They are defined by the Grid class but set on children of the Grid.

When you wish to use these attached properties in code, the Grid class provides static methods named SetRow, GetColumn, and so forth. But in XAML, these attached properties are set as attributes in the children of the Grid using simple properties names.

Attached properties are always recognizable in XAML files as attributes containing both a class and a property name separated by a period. They are called *attached properties* because they are defined by one class (in this case, Grid) but attached to other objects (in this case, children of the Grid). During layout, the Grid can interrogate the values of these attached properties to know where to place each child.

The AbsoluteLayout class defines two attached properties named LayoutBounds and LayoutFlags. Here’s something of a checkerboard pattern realized using the proportional positioning and sizing features of

AbsoluteLayout:

Content Properties

- Objects can have content or children
  - Aren't used often directly in XAML
  - Could be done, is however almost always omitted

```
<StackLayout>
    <StackLayout.Children>
     <Slider VerticalOptions="CenterAndExpand"
          ValueChanged="OnSliderValueChanged" />

     <Label x:Name="valueLabel"
          Text="A simple Label"
          FontSize="Large"
          HorizontalOptions="Center“a
          VerticalOptions="CenterAndExpand" />

     <Button Text="Click Me!"
          HorizontalOptions="Center"
          VerticalOptions="CenterAndExpand"
          Clicked="OnButtonClicked" />
    </StackLayout.Children>
  </StackLayout>
```

PXL IT

Content properties

- Why aren't they always used?
  - In XAML, each object can have one property flagged as ContentProperty
    - All "content" goes into that property automatically

```
[Xamarin.Forms.ContentProperty("Children")]
public abstract class Layout<T> : Layout ...
```

PXL IT

Content properties

Element	Content Property
ContentPage	Content
ContentView	Content
Frame	Content
Label	Text
Layout<T>	Children
ScrollView	Content
ViewCell	View

XAML Syntax

DEMO

PXL IT

Markup extensions, shared resources and styles

DE HOGESCHOOL
MET HET NETWERK

Hogeschool PXL – Elfde-Liniestraat 24 – B-3500 Hasselt
www.pxl.be - www.pxl.be/facebook

XAML Markup Extensions

- Normally, properties are set to values
- If we need processing to be done for a value, we can use a markup extension
  - Backed by code: class that implements IMarkupExtension
  - Surrounded by { and }
- You can add your own or use the provided ones

PXL IT

In general, properties of an object are set to explicit values, such as a string, a number, an enumeration member, or a string that is converted to a value behind the scenes.

Sometimes, however, properties must instead reference values defined somewhere else, or which might require a little processing by code at runtime. For these purposes, XAML *markup extensions* are available.

These XAML markup extensions are not extensions of XML. XAML is entirely legal XML. They're called "extensions" because they are backed by code in classes that implement IMarkupExtension. You can write your own custom markup extensions.

In many cases, XAML markup extensions are instantly recognizable in XAML files as attribute settings delimited by curly braces: { and }, but sometimes markup extensions appear in markup as conventional elements.

Shared Resources

- If we have many visual properties, we can use a resource dictionary to store often-used values
- Each VisualElement exposes Resources property of type ResourceDictionary
  - Dictionary<string,object>
  - Defined entries can then be referenced from XAML

PXL IT

If one of these properties needs to be changed, you might prefer to make the change just once rather than three times. If this were code, you'd likely be using constants and static read-only objects to help keep such values consistent and easy to modify.

In XAML, one popular solution is to store such values or objects in a *resource dictionary*. The VisualElement class defines a property named Resources of type ResourceDictionary, which is a dictionary with keys of type string and values of type object. You can put objects into this dictionary and then reference them from markup, all in XAML.

To use a resource dictionary on a page, include a pair of Resources property-element tags. It's most convenient to put these right at the top of the page:

Creating a resource dictionary

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"

xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="XamlSamples.SharedResourcesPage"
             Title="Shared Resources Page">
<ContentPage.Resources>
    <ResourceDictionary>
      <LayoutOptions x:Key="horzOptions"
                     Alignment="Center" />

      <LayoutOptions x:Key="vertOptions"
                     Alignment="Center"
                     Expands="True" />
      <x:Int32 x:Key="borderWidth">
        3
      </x:Int32>
    </ResourceDictionary>
  </ContentPage.Resources>
  …

</ContentPage>
```

Using the resource dictionary

```
<Button Text="Do this!"
        HorizontalOptions="{StaticResource
horzOptions}"
        VerticalOptions="{StaticResource
vertOptions}"
        BorderWidth="{StaticResource
borderWidth}"
        Rotation="{StaticResource
rotationAngle}"
        TextColor="{StaticResource textColor}"
        Font="{StaticResource font}" />
```

Using styles

```
<?xml version="1.0" encoding="UTF-8"?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="WorkingWithStyles.StyleXaml">
    <ContentPage.Resources>
        <ResourceDictionary>
            <Style x:Key="ButtonStyle" TargetType="Button">
                <Setter Property="BackgroundColor" Value="Yellow"/>
                <Setter Property="BorderRadius" Value="0"/>
                <Setter Property="HeightRequest" Value="42"/>
            </Style>
        </ResourceDictionary>
    </ContentPage.Resources>

    <Button Text="Style Me XAML" Style="{StaticResource ButtonStyle}" />

</ContentPage>
```

PXL IT

Application-wide styles

```
<Application
    xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="WorkingWithAppResources.App">
    <Application.Resources>
        <ResourceDictionary>
            <Style x:Key="labelStyle" TargetType="Label">
                <Setter Property="TextColor" Value="Green" />
            </Style>
        </ResourceDictionary>
    </Application.Resources>
</Application>
```

PXL IT

Implicit styles

```
<ResourceDictionary>
    <Style TargetType="BoxView">
        <Setter Property="Color" Value="Aqua" />
    </Style>
</ResourceDictionary>
```

Resources and styles

DEMO

Data binding

DE HOGESCHOOL
MET HET NETWERK

Hogeschool PXL – Elfde-Liniestraat 24 – B-3500 Hasselt
www.pxl.be - www.pxl.be/facebook

Data Binding Basics

- Data bindings allow properties of two objects to be linked so that a change in one causes a change in the other
  - Source provides the data
  - Target consumes and displays the data from the source
- Binding is a markup extension
- Applying a data binding is done as follows:
  - Set the BindingContext property to the object that will be bound to
    - The bound object may be any .NET object that implements the INotifyPropertyChanged interface
  - Invoke the SetBinding method on the Xamarin.Forms object once for each property or method that should be bound

Data binding is used to simplify how a Xamarin.Forms application can display and interact with its data. It establishes a connection between the user interface and the underlying application. When the user edits the value in a text box, the data binding automatically updates an associated property on an underlying object. The BindableObject class contains much of the infrastructure to support data binding.

Data binding defines the relationship between two objects. The *source* object will provide data. The *target* object is another object that will consume (and often display) the data from the source object. For example, a Label may display the name from an Employee class. In this case, the Employee object is the source, while the Label is the target.

Setting up data binding on a Xamarin.Forms object (such as a Page or a Control) follows these two steps:

Set the *BindingContext* property to the object that will be bound to. The bound object may be any .NET object that implements the INotifyPropertyChanged interface (discussed below).

Invoke the *SetBinding* method on the Xamarin.Forms object once for each property or method that should be bound.

Data Binding Basics

- SetBinding accepts 2 parameters
  - Type of the binding
  - Information on what or how to binding
    - Normally, just the name of the property we are binding to (property of the BindingContext)

```
someLabel.SetBinding(Label.TextProperty, new Binding("."));
```

PXL IT

Data bindings from code

```
public EmployeeDetailPage(Employee employeeToDisplay)
{
    this.BindingContext = employeeToDisplay;

    var firstName = new Entry()
                {
                    HorizontalOptions = LayoutOptions.FillAndExpand
                };
    firstName.SetBinding(Entry.TextProperty, "FirstName");

    // Rest of the code omitted…
}
```

PXL IT

The first line of code sets the BindingContext to a .NET object – this tells the underlying data binding API's what object to bind to. The next line of code instantiates a Xamarin.Forms.Entry control. The last line defines the binding between the Xamarin.Forms.Entry and employeeToDisplay; the Entry.Text property should be bound to the FirstName property of the object set to the BindingContext. The changes made in the Entry control will automatically be propagated to the employeeToDisplay object. Likewise, if changes are made to employeeToDisplay.FirstName, then Xamarin.Forms will also update the contents of the Entry control. This is known as *two-way binding*.

INotifyPropertyChanged

- Interface which is used to notify a client of an object that a value has changed

```
public interface INotifyPropertyChanged
{
    event PropertyChangedEventHandler PropertyChanged;
}
```

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

   string _firstName;
   public string FirstName
   {
      get { return _firstName; }
      set
      {
         if (value.Equals(_firstName,  StringComparison.Ordinal))
         {
            // Nothing to do - the value hasn't changed;
            return;
         }
         _firstName = value;
         OnPropertyChanged();

      }
   }

   void OnPropertyChanged([CallerMemberName]  string propertyName = null)
   {
      var handler = PropertyChanged;
      if (handler  != null)
      {
         handler(this,  new PropertyChangedEventArgs(propertyName));
      }
   }
}
```

Objects that implement INotifyPropertyChanged must raise the PropertyChanged event when one of their properties is updated with a new value. An example of one such class can be seen in the following class:

Data bindings in XAML

- Binding can be done entirely in XAML
- Setting the BindingContext is required to be done from code or via StaticResource

PXL IT

Data bindings connect properties of two objects, called the *source* and the *target*. In code, two steps are required: The BindingContext property of the target object must be set to the source object, and the SetBinding method (often used in conjunction with the Binding class) must be called on the target object to bind a property of that object to a property of the source object.

The target property must be a bindable property, which means that the target object must derive from BindableObject. The online Xamarin.Forms documentation indicates which properties are bindable properties.

In markup, these same two steps are also required, except that the Binding markup extension takes the place of the SetBinding call and the Binding class.

However, there is no single technique to set the BindingContext of the target object. Sometimes it's set from the code-behind file, sometimes using a StaticResource or x:Static markup extension, and sometimes as the content of BindingContext property-element tags.

View-to-view bindings

- It's possible to link two views using binding

```
<StackLayout>
    <Label Text="ROTATION"
           BindingContext="{x:Reference Name=slider}"
           Rotation="{Binding Path=Value}"
           Font="Bold, Large"
           HorizontalOptions="Center"
           VerticalOptions="CenterAndExpand" />

    <Slider x:Name="slider"
            Maximum="360"
            VerticalOptions="CenterAndExpand" />

    <Label BindingContext="{x:Reference slider}"
           Text="{Binding Value,
                          StringFormat='The angle is {0:F0} degrees'}"
           Font="Large"
           HorizontalOptions="Center"
           VerticalOptions="CenterAndExpand" />
  </StackLayout>
```

Data bindings can be defined to link properties of two views on the same page. In this case, you set the BindingContext of the target object using the x:Reference markup extension.

Here's a XAML file that contains a Slider and two Label views, one of which is rotated by the Slider value and another which displays the Slider value:

The Slider contains an x:Name attribute that is referenced by the two Label views using the x:Reference markup extension.

The x:Reference binding extension defines a property named Name to set to the name of the referenced element, in this case slider. However, the ReferenceExtension class that defines the x:Reference markup extension also defines a ContentProperty attribute for Name, which means that it isn't explicitly required. Just for variety, the first x:Reference includes "Name=" but the second does not:

The Binding markup extension itself can have several properties, just like the BindingBase and Binding class. The ContentProperty for Binding is Path, but the "Path=" part of the markup extension can be omitted only if it's the first item in the Binding markup extension. The first example has "Path=" but the second omits it:

The properties can all be on one line or separated into multiple lines. Whatever's convenient.

Notice the StringFormat in the second Binding markup extension. In Xamarin.Forms, bindings do not perform any implicit type conversions, and if you need to display a non-string object as a string you must provide a type converter or use StringFormat. Behind the scenes, the string specified in the StringFormat is used in the static String.Format method. That's potentially a problem, because .NET formatting specifications involve curly braces, which are also used to delimit markup extensions and hence have the danger of confusing the XAML parser. To avoid that, put the entire formatting string in single quotation marks:

Data binding

DEMO

Bindings and Collections

- Data binding work with Collections, for example ListView
  - Defines ItemsSource (IEnumerable)
  - Items can be of any type
  - By default, will apply just the ToString() on the items
- Works great in combination with templates: class which derives from Cell
  - Template is cloned for each item in collection
  - Each will be able to bind on item individually
  - Custom ViewCell can be created in XAML

PXL IT

Nothing illustrates the power of XAML and data bindings better than a templated ListView.

ListView defines an ItemsSource property of type IEnumerable, and it displays the items in that collection. These items can be objects of any type. By default, ListView uses the ToString method of each item to display that item. Sometimes this is just what you want, but in many cases the ToString returns only the fully-qualified class name of the object.

However, the items in the ListView collection can be displayed any way you want through the use of a template, which involves a class that derives from Cell. The template is cloned for every item in the ListView, and data bindings that have been set on the template are transferred to the individual clones.

Very often, you'll want to create a custom cell for these items using the ViewCell class. This process is somewhat messy in code, but in XAML it becomes very straightforward.

Without templates

```
<ListView ItemsSource="{x:Static local:NamedColor.All}" />
```

Defining a template

```
<ListView ItemsSource="{x:Static local:NamedColor.All}">
    <ListView.ItemTemplate>
      <DataTemplate>
        <ViewCell>
          <ViewCell.View>
            <Label Text="{Binding FriendlyName}" />
          </ViewCell.View>
        </ViewCell>
      </DataTemplate>
    </ListView.ItemTemplate>
  </ListView>
```

PXL IT

It's not much information, but the ListView is scrollable and selectable.

To define a template for the items, you'll want to break out the ItemTemplate property as a property element, and set it to a DataTemplate, which then references a ViewCell. To the View property of the ViewCell you can define a layout of one or more views to display each item. Here's a simple example:

The result


Carrier
9:08 PM
Back
ListView Demo Page
Alice Blue
Antique White
Aqua
Aquamarine
Azure
Beige
Bisque
ListView Demo Page
Alice Blue
Antique White
Aqua
Aquamarine
Azure
Beige
Bisque
Black
Blanched Almond
Alice Blue
Antique White
Aqua
Aquamarine
Azure
Beige
Bisque
Black
Blanched Almond

Binding collections

DEMO

PXL IT

Advanced topics in Xamarin.Forms

DE HOGESCHOOL
MET HET NETWERK

Hogeschool PXL – Elfde-Liniestraat 24 – B-3500 Hasselt
www.pxl.be - www.pxl.be/facebook

PXL IT

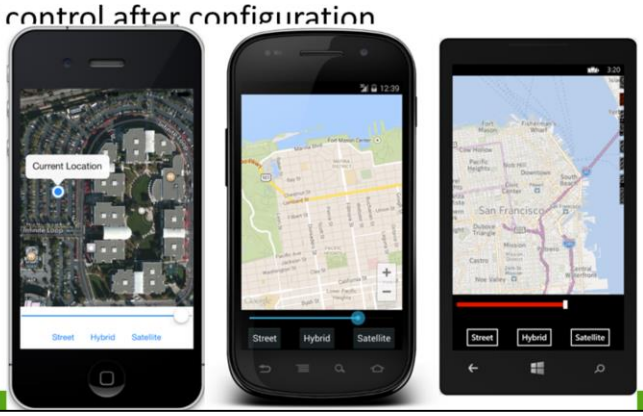
Working with maps in Xamarin.Forms

DE HOGESCHOOL
MET HET NETWERK

Hogeschool PXL – Elfde-Liniestraat 24 – B-3500 Hasselt
www.pxl.be - www.pxl.be/facebook

Working with Maps

- Uses native Map API on each platform
  - Familiar experience for all users
  - Different configuration is required on different platforms
    - Only main difference, it's a regular control after configuration
    - A custom renderer can be used for further customization



Xamarin.Forms uses the native map APIs on each platform.

Xamarin.Forms.Maps uses the native map APIs on each platform. This provides a fast, familiar maps experience for users, but means that some configuration steps are needed to adhere to each platforms specific API requirements. Once configured, the Map control works just like any other Xamarin.Forms element in common code.

Maps Initialization

- Xamarin.Forms.Maps is a separate NuGet package that needs to be installed
  - Depends (on Android) on Google Play Services just like maps in classic Xamarin → downloaded separately
- Requires initialization
  - Android: In OnCreate of MainActivity

```
Xamarin.FormsMaps.Init(this, bundle);
```

  - iOS: in FinishedLaunching of AppDelegate

```
Xamarin.FormsMaps.Init();
```

When adding maps to a Xamarin.Forms application, **Xamarin.Forms.Maps** is a a separate NuGet package that you should add to every project in the solution. On Android, this also has a dependency on GooglePlayServices (another NuGet) which is downloaded automatically when you add Xamarin.Forms.Maps.

After installing the NuGet package, some initialization code is required in each application project, *after* theXamarin.Forms.Forms.Init method call. For iOS use the following code:

Xamarin.FormsMaps.Init();On Android you must pass the same parameters as Forms.Init:

Xamarin.FormsMaps.Init(this, bundle);For the Windows Runtime (WinRT) and the Universal Windows Platform (UWP) use the following code:

Xamarin.FormsMaps.Init("INSERT_AUTHENTICATION_TOKEN_HERE");Add this call in the following files for each platform:

iOS - AppDelegate.cs file, in the FinishedLaunching method.

Android - MainActivity.cs file, in the OnCreate method.

WinRT and UWP - MainPage.xaml.cs file, in the MainPage constructor.

Once the NuGet package has been added and the initialization method called inside each applcation, Xamarin.Forms.MapsAPIs can be used in the common PCL or Shared Project code.

Maps Initialization

- On iOS (since v8.0), we need to add 2 keys to info.plist
  - NSLocationAlwaysUsageDescription
  - NSLocationWhenInUseUsageDescription

NSLocationWhenInUseUsageDescription	String	We are using your location
NSLocationAlwaysUsageDescription	String	Can we use your location

PXL IT

On iOS 7 the map control "just works", so long as the FormsMaps.Init() call has been made.

For iOS 8 two keys need to be added to the **Info.plist** file: NSLocationAlwaysUsageDescription and NSLocationWhenInUseUsageDescription. The XML representation is shown below - you should update the string values to reflect how your application is using the location information:

Maps Initialization

- On Android, we need to generate an API key
- Needs to be added to AndroidManifest.xml
  - Grey box appears if not added correctly

```
meta-data
android:name="com.google.android.maps.v2.API_KEY"
    android:value="YourKeyGoesHere" />
```

- Extra permissions are also required
  - AccessCoarseLocation
  - AccessFineLocation
  - AccessLocationExtraCommands
  - AccessMockLocation
  - AccessNetworkState
  - AccessWifiState
  - Internet

PXL IT

To use the Google Maps API v2 on Android you must generate an API key and add it to your Android project. Follow the instructions in the Xamarin doc on obtaining a Google Maps API v2 key. After following those instructions, paste the API key in the **Properties/AndroidManifest.xml** file (view source and find/update the following element):

<meta-data android:name="com.google.android.maps.v2.API_KEY" android:value="AbCdEfGhIjKlMnOpQrStUvWValueGoesHere" />Without a valid API key the maps control will display as a grey box on Android.

Go here for key creation: https://developer.xamarin.com/guides/android/platform_features/maps_and_location/maps/obtaining_a_google_maps_api_key/

Using Maps

- Sample code to display a map

```
public class MapPage : ContentPage {
    public MapPage() {
        var map = new Map(
            MapSpan.FromCenterAndRadius(
                    new Position(37,-122), Distance.FromMiles(0.3))) {
                IsShowingUser = true,
                HeightRequest = 100,
                WidthRequest = 960,
                VerticalOptions = LayoutOptions.FillAndExpand
            };
        var stack = new StackLayout { Spacing = 0 };
        stack.Children.Add(map);
        Content = stack;
    }
}
```

Using Maps

- We can change the map type:

```
map.MapType == MapType.Street;
```

- Possible values of the enumeration
  - Hybrid
  - Satellite
  - Street (the default)

The map content can also be changed by setting the MapType property, to show a regular street map (the default), satellite imagery or a combination of both.

map.MapType == MapType.Street;Valid MapType values are:

Hybrid

Satellite

Street (the default)

Using Maps

- MapSpan can be used to set initial view
  - Center point and zoom level
- MoveToRegion() can be used to change this value
- MapSpan can be created:
  - MapSpan.FromCenterAndRadius():
    - Static method to create a span from a Position and specifying a Distance
  - new MapSpan ()
    - Constructor that uses a Position and the degress of latitude and longitude to display

PXL IT

As shown in the code snippet above, supplying a MapSpan instance to a map constructor sets the initial view (center point and zoom level) of the map when it is loaded. The MoveToRegion method on the map class can then be used to changed the position or zoom level of the map. There are two ways to create a new MapSpan instance:

MapSpan.FromCenterAndRadius() - static method to create a span from a Position and specifying a Distance .

new MapSpan () - constructor that uses a Position and the degress of latitude and longitude to display.

Using Maps

- Zoom level can also be changed without changing the location
  - Set the center of the MapSpan to the VisibleRegion.Center property
  - Can then be changed using a Slider

```
var slider = new Slider (1, 18, 1);
slider.ValueChanged += (sender, e) => {
    var zoomLevel = e.NewValue; // between 1 and 18
    var latlongdegrees = 360 / (Math.Pow(2, zoomLevel));
    map.MoveToRegion(new MapSpan (map.VisibleRegion.Center, latlongdegrees, latlongdegrees));
};
```

PXL IT

To change the zoom level of the map without altering the location, create a new MapSpan using the current location from theVisibleRegion.Center property of the map control. A Slider could be used to control map zoom like this (however zooming directly in the map control cannot currently update the value of the slider):

Using Maps

- Map pins can be added using new Pin instances

```
var position = new Position(37,-122); // Latitude, Longitude
var pin = new Pin {
            Type = PinType.Place,
            Position = position,
            Label = "custom pin",
            Address = "custom detail info"
        };
map.Pins.Add(pin);
```

Maps using XAML

```
<?xml version="1.0" encoding="UTF-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:maps="clr-namespace:Xamarin.Forms.Maps;assembly=Xamarin.Forms.Maps"
    x:Class="MapDemo.MapPage">
    <StackLayout VerticalOptions="StartAndExpand" Padding="30">
        <maps:Map WidthRequest="320" HeightRequest="200"
            x:Name="MyMap"
            IsShowingUser="true"
            MapType="Hybrid"
        />
    </StackLayout>
</ContentPage>
```

Working with Maps

DEMO

PXL IT

Custom Renderers

DE HOGESCHOOL
MET HET NETWERK

Hogeschool PXL – Elfde-Liniestraat 24 – B-3500 Hasselt
www.pxl.be - www.pxl.be/facebook

Customizing controls for each platform

- Xamarin.Forms UIs are rendered using native controls for each platforms
  - This way, the apps look great on each platform
- Xamarin.Forms is internally based on renderers
  - Each control has a renderer for each supported control
- We can extend this behaviour for our own controls using custom renderers
  - Can be used for small styling changes or completely different experiences, both in layout and behavior

PXL IT

Xamarin.Forms user interfaces are rendered using the native controls of the target platform, allowing Xamarin.Forms applications to retain the appropriate look and feel for each platform. Custom Renderers let developers override this process to customize the appearance and behavior of Xamarin.Forms controls on each platform.

Custom renderers provide a powerful approach for customizing the appearance and behavior of Xamarin.Forms controls. They can be used for small styling changes or sophisticated platform-specific layout and behavior customization. This article provides an introduction to custom renderers, and outlines the process for creating a custom renderer.

Why custom renderers

- Each layout, page or control is rendered using a renderer
  - Base class is Renderer
  - Creates in turn a native control specific to the platform
    - Also takes care of positioning on screen and behaviour via code
- We can build on this system
  - All what is missing in Xamarin.Forms can be added this way
  - A control can be customized for one or all platforms
  - Simple changes can be done (missing property) but can also be added using Effects

PXL IT

Custom renderers provide a powerful approach for customizing the appearance and behavior of Xamarin.Forms controls. They can be used for small styling changes or sophisticated platform-specific layout and behavior customization. This article provides an introduction to custom renderers, and outlines the process for creating a custom renderer.

Xamarin.Forms Pages, Layouts and Controls present a common API to describe cross-platform mobile user interfaces. Each page, layout, and control is rendered differently on each platform, using a Renderer class that in turn creates a native control (corresponding to the Xamarin.Forms representation), arranges it on the screen, and adds the behavior specified in the shared code.

Developers can implement their own custom Renderer classes to customize the appearance and/or behavior of a control. Custom renderers for a given type can be added to one application project to customize the control in one place while allowing the default behavior on other platforms; or different custom renderers can be added to each application project to create a different look and feel on iOS, Android, and Windows Phone. However, implementing a custom renderer class in order to perform a simple control customization is often a heavy-weight response. Effects simplify this process, and are typically used for small styling changes. For more information, see Effects.

Why custom renderers

- Can't we just inherit from the base control?
  - Limited functionality, only what is "shared" across platforms can be used this way, we can't access platform-specific functions!

```
public class MyEntry : Entry
{
  public MyEntry ()
  {
    BackgroundColor = Color.Gray;
  }
}
```

  - Above control can now be used from XAML

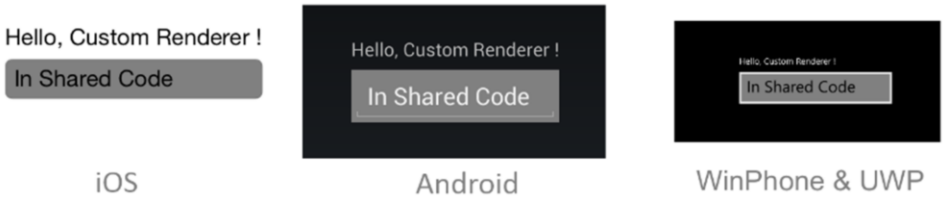
PXL IT

Changing the appearance of a Xamarin.Forms control, without using a custom renderer, is a two-step process that involves creating a custom control through subclassing, and then consuming the custom control in place of the original control. The following code example shows an example of subclassing the Entry control:

Changing the background color of the control on each platform has been accomplished purely through subclassing the control. However, this technique is limited in what it can achieve as it is not possible to take advantage of platform-specific enhancements and customizations. When they are required, custom renderers must be implemented.

Why custom renderers

```
<ContentPage
    ...
    xmlns:local="clr-namespace:CustomRenderer;assembly=CustomRenderer"
    ...>
    ...
    <local:MyEntry Text="In Shared Code" />
    ...
</ContentPage>
```



iOS Android WinPhone & UWP

PXL IT

Creating a Custom Renderer Class

- Steps required to create a custom renderer:
  - Create a subclass of the **renderer** class that renders the native control
  - Override the method that renders the native control and write logic to customize the control
    - The OnElementChanged method is used to render the native control, which is called when the corresponding Xamarin.Forms control is created
  - Add an **ExportRenderer** attribute to the custom renderer class to specify that it will be used to render the Xamarin.Forms control
    - This attribute is used to register the custom renderer with Xamarin.Forms.

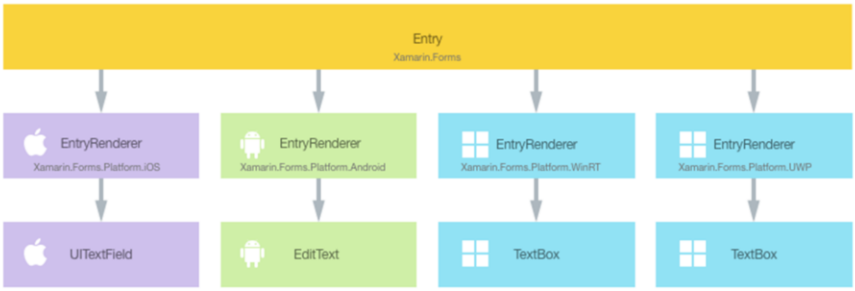
What renderers are using internally in XF

Views	Renderer	iOS	Android	Windows Phone 8	WinRT / UWP
ActivityIndicator	ActivityIndicatorRenderer	UIActivityIndicator	ProgressBar	ProgressBar	ProgressBar
BoxView	BoxRenderer (iOS and Android)	UIView	ViewGroup	Rectangle	Rectangle
Button	ButtonRenderer	UIButton	Button	Button	Button
CarouselView	CarouselViewRenderer	UIScrollView	RecyclerView	FlipView	FlipView
DatePicker	DatePickerRenderer	UITextField	EditText	DatePicker	DatePicker
Editor	EditorRenderer	UITextView	EditText	TextBox	TextBox
Entry	EntryRenderer	UITextField	EditText	PhoneTextBox/PasswordBox	TextBox
Image	ImageRenderer	UIImageView	ImageView	Image	Image
Label	LabelRenderer	UILabel	TextView	TextBlock	TextBlock

PXL IT

Creating a custom entry

- Entry is used for single line of text rendering
- Custom renderer can be used to create custom entry control
  - Entry is rendered using the EntryRenderer
  - Generates
    - UITextField on iOS
    - EditText on Android
    - TextBox on Windows



The Xamarin.Forms Entry control allows a single line of text to be edited. This article demonstrates how to create a custom renderer for the Entry control, enabling developers to override the default native rendering with their own platform-specific customization.

Every Xamarin.Forms control has an accompanying renderer for each platform that creates an instance of a native control. When an Entry control is rendered by a Xamarin.Forms application, in iOS the EntryRenderer class is instantiated, which in turns instantiates a native UITextField control. On the Android platform, theEntryRenderer class instantiates an EditText control. On Windows Phone and the Universal Windows Platform (UWP), the EntryRenderer class instantiates a TextBoxcontrol. For more information about the renderer and native control classes that Xamarin.Forms controls map to, see Renderer Base Classes and Native Controls.

The following diagram illustrates the relationship between the Entry control and the corresponding native controls that implement it:

Creating a custom entry

- Step 1: Create a custom entry control

```
public class MyEntry : Entry
{
}
```

Creating a custom entry

- Step 2: Use the custom entry control

```
<ContentPage ...
    xmlns:local="clr-namespace:CustomRenderer;assembly=CustomRenderer"
    ...>
    ...
    <local:MyEntry Text="In Shared Code" />
    ...
</ContentPage>
```

Creating a custom entry

- Can also be done from C#

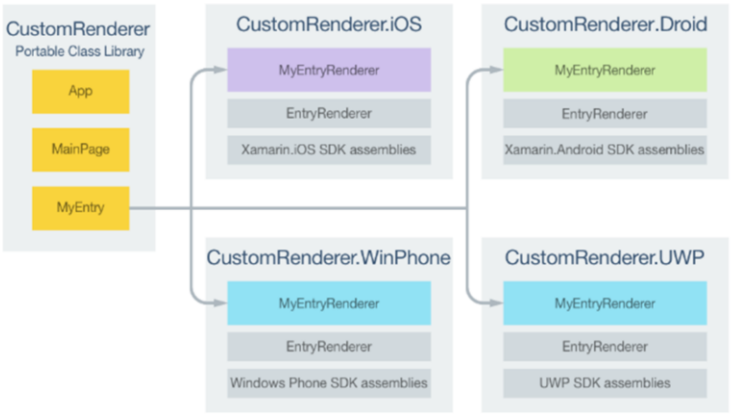
```
public class MainPage : ContentPage
{
  public MainPage ()
  {
    Content = new StackLayout {
      Children = {
        new Label {
          Text = "Hello, Custom Renderer !",
        },
        new MyEntry {
          Text = "In Shared Code",
        }
      },
      VerticalOptions = LayoutOptions.CenterAndExpand,
      HorizontalOptions = LayoutOptions.CenterAndExpand,
    };
  }
}
```

Creating a custom entry

- Step 3: Creating the Custom Renderer on each Platform
  - Create a subclass of the EntryRenderer class that renders the native control
  - Override the OnElementChanged method that renders the native control and write logic to customize the control
    - This method is called when the corresponding Xamarin.Forms control is created.
  - Add an ExportRenderer attribute to the custom renderer class to specify that it will be used to render the Xamarin.Forms control
    - This attribute is used to register the custom renderer with Xamarin.Forms.

Creating a custom entry

- Step 3 overview



The OnElementChanged method

- OnElementChanged method is exposed in custom renderer
  - Called automatically by XF when control is to be generated
  - Receives ElementChangedEventArgs
    - Contains OldElement and NewElement properties
    - OldElement representd the Xamarin.Forms element that the renderer *was* attached to, typically, this is null
    - NewElement represents the Xamarin.Forms element that the renderer *is* attached to
- In OnElementChanged override, we will write code to interact with native controls
  - Typed reference to native control is accessible through Control property

PXL IT

The EntryRenderer class exposes the OnElementChanged method, which is called when the Xamarin.Forms control is created in order to render the corresponding native control. This method takes an ElementChangedEventArgs parameter that containsOldElement and NewElement properties. These properties represent the Xamarin.Forms element that the renderer *was*attached to, and the Xamarin.Forms element that the renderer *is* attached to, respectively. In the sample application theOldElement property will be null and the NewElement property will contain a reference to the MyEntry control.

An overridden version of the OnElementChanged method in the MyEntryRenderer class is the place to perform the native control customization. A typed reference to the native control being used on the platform can be accessed through theControl property. In addition, a reference to the Xamarin.Forms control that's being rendered can be obtained through theElement property, although it's not used in the sample application.

Each custom renderer class is decorated with an ExportRenderer attribute that registers the renderer with Xamarin.Forms. The attribute takes two parameters – the type name of the Xamarin.Forms control being rendered, and the type name of the custom renderer. The assembly prefix to the attribute specifies that the attribute applies to the entire assembly.

The following sections discuss the implementation of each platform-specific MyEntryRenderer custom renderer class.

The OnElementChanged method

- ExportRenderer is added as attribute → registers the renderer with Xamarin.Forms
  - Accepts 2 parameters
    - The type name of the Xamarin.Forms control being rendered
    - The type name of the custom renderer
  - Assembly prefix indicates that attribute is applies to entire assembly

Each custom renderer class is decorated with an ExportRenderer attribute that registers the renderer with Xamarin.Forms. The attribute takes two parameters – the type name of the Xamarin.Forms control being rendered, and the type name of the custom renderer. The assembly prefix to the attribute specifies that the attribute applies to the entire assembly.

The following sections discuss the implementation of each platform-specific MyEntryRenderer custom renderer class.

Creating the Custom Renderer on iOS

- Control gives access to UITextField

```
using Xamarin.Forms.Platform.iOS;

[assembly: ExportRenderer (typeof(MyEntry), typeof(MyEntryRenderer))]
namespace CustomRenderer.iOS
{
    public class MyEntryRenderer : EntryRenderer
    {
        protected override void OnElementChanged (ElementChangedEventArgs<Entry> e)
        {
            base.OnElementChanged (e);

            if (Control != null) {
                // do whatever you want to the UITextField here!
                Control.BackgroundColor = UIColor.FromRGB (204, 153, 255);
                Control.BorderStyle = UITextBorderStyle.Line;
            }
        }
    }
}
```

Creating the Custom Renderer on Android

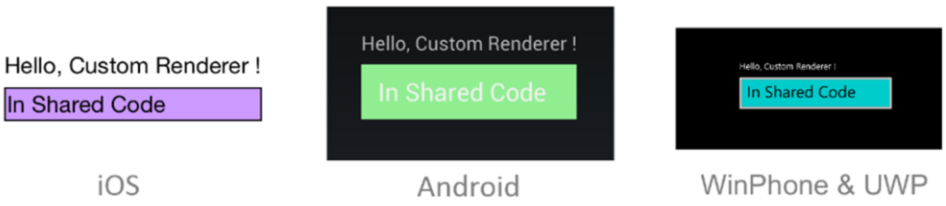
```
using Xamarin.Forms.Platform.Android;

[assembly: ExportRenderer (typeof(MyEntry), typeof(MyEntryRenderer))]
namespace CustomRenderer.Android
{
    class MyEntryRenderer : EntryRenderer
    {
        protected override void OnElementChanged (ElementChangedEventArgs<Entry> e)
        {
            base.OnElementChanged (e);

            if (Control != null) {
                Control.SetBackgroundColor
(global::Android.Graphics.Color.LightGreen);
            }
        }
    }
}
```

Creating a custom entry

- The result



iOS Android WinPhone & UWP

Custom renderers

DEMO