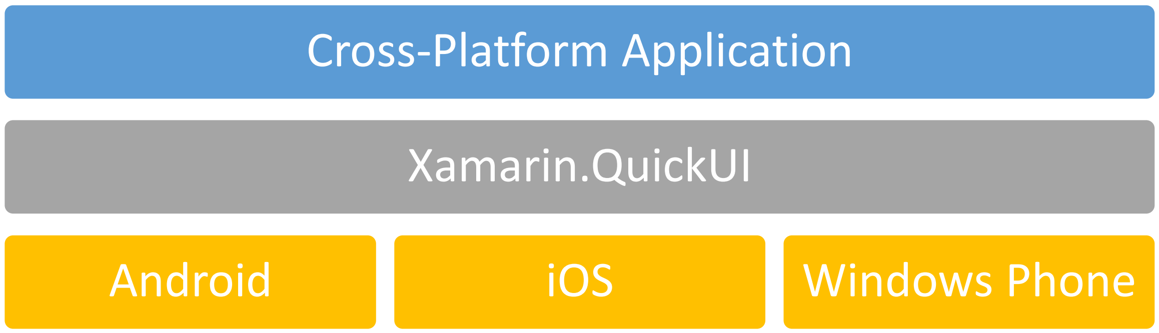
Xamarin.QuickUI Introduction

Draft Document

Introduction

Xamarin.QuickUI is a cross platform UI abstraction layer targeting iOS, Android, and Windows Phone. QuickUI enables sharing of almost 100% of all code across all target platforms, however all QuickUI types are mapped transparently down to native types for each platform. QuickUI is capable of producing almost any UI you can make with conventional methodology, however it excels at rapid prototyping as well as data intensive apps.



It is important to note that QuickUI is not a toolkit. It does not re-implement drawing code or event systems. It simply abstracts the existing UI's into a common framework.

Core Concepts

Basic Elements

QuickUI contains three primary classes which contain the majority of the useful code.

- View – The base class which all controls in QuickUI derive from.
- Layout – A View which can contain children and is responsible for arranging them.
- Page – A View used to create different pages or screens in an App.

Pages are similar to Pages on WPF or a UIViewController on iOS. They contain lifecycle events and are used for the purposes of tabs and navigation. All QuickUI App's will contain at least one Page as its root element.

Usage Scenarios

There are three primary ways in which QuickUI can be utilized

1. 100% QuickUI with fully shared code.
2. QuickUI app with custom platform specific renderers.
3. Traditional native app with embedded QuickUI pages.

Appearance

Appearance in QuickUI can be configured to utilize CSS engines like pixate or native css. Additionally there are some basic theming elements build into most QuickUI Views. Background colors, images, fonts, text colors, and simple theme options can usually be set without a css engine or a custom renderer.

Basic Usage

Creating a View

Views are created simply by invoking their constructors. They are best created using initializer syntax.

```
    var button = new Button {Text = “My Button”};
    var label = new Label {Text = “My Label”};
```

Creating a Layout

Layout initialization is not different from views. Adding elements to Layout’s is done via the Add method, which enables the usage of collection initializer syntax.

```
Using initializer syntax:
    var stackLayout = new StackLayout {button, label};
Or using the add method directly:
    var stackLayout = new StackLayout ();
    stackLayout.Add (button);
    stackLayout.Add (label);
```

Creating a Page

Page creation mirrors that of views and layouts. The most important page is the ContentPage which is used to embed content into other pages.

```
    var myPage = new ContentPage {
        Content = new Label {“Hello World”};
    };
```

Combining all of the initialization syntax seen thus far allows for simple pages to be arranged in a single statement. It is also common to set the title and an icon for a page.

```
    var simplePage = new ContentPage {
        Title = “My Page”,
        Icon = “mainIcon.png”,
        Content = new StackLayout {
            new Label {Text = “Hello World”},
            new Button {Text = “Click Me!”}
        }
    };
```

Layouts

Common Concepts

All Views contain a VerticalOptions and HorizontalOptions property. These properties provide behavior preferences that parent layouts should respect. These options specify where the view should be placed in the available space, start, center, end, or fill available space. They also specify if the View would desire to be expanded to all available overflow space as well.

If multiple Views in a single layout request to expand, overflow space is evenly distributed between them.

StackLayout

StackLayout is the simplest of all layouts. It lays out all of its children in either a vertical or horizontal stack. Views can be added to a StackLayout with the Add method. StackLayout's are similar to the Android LinearLayout or the WPF StackPanel.

```
	var stack = new StackLayout {
		Orientation = StackOrientation.Horizontal
	};
	stack.Add (new Button {Text = “Foo”});
	stack.Add (new Label {Text = “Bar”});
```

```
This can also be expressed as:

	var button = new Button {Text = “Foo”};
	var label = new Label {Text = “Bar”};
	var stack = new StackLayout {
		Orientation = StackOrientation.Horizontal
	}.WithChildren (button, label);
```

GridLayout

Absolute Layout

Navigation

The QuickUI Navigation model is provided by either global navigation or through the usage of a NavigationPage.

Dealing with UX Differentiation

On the Windows Phone platform there is no equivalent of the iOS UINavigationController. All navigation is performed at the application level, replacing the entire contents of the screen. This means that at times you will need to write code which treats some platforms different from others. The View.OnPlatform methods are useful for this.

On Windows Phone it is considered good practice to not use a NavigationPage. It is not a requirement however a warning will be produced.

Creating a NavigationPage

NavigationPages work by pushing and popping ContentPages. The initial page can be provided to the constructor of the NavigationPage or be Pushed before it is added to any other page.

```
        var navigationPage = new NavigationPage (myPage);
```

Accessing a NavigationPage from a View

All View's contain a Navigation property which returns an INavigation object. It is worth noting this object is not the NavigationPage. The Navigation property is valid and usable at construction time of the View, however any pushed Pages will not be pushed until the View is added to a tree with a navigation provider.

```
        var myView = new Button {Text = “Button”};
        myView.Activated += (s, e) => myView.Navigation.Push (myPage);
```

```
On Windows Phone the Navigation property will work without the usage of a
NavigationPage.
```

Modal Pages

Modal pages can be pushed and popped from the navigation property. Once a Modal is pushed, the previous pages will become unreachable until it is popped. Attempting to interact with non-modal Pages when a modal Page is presented will result in an error. Pushing a navigation page as a modal page results in undefined behavior.

Pages

Page is the base class of all content containing elements. In an MVC setup, a Page most closely represents the Controller. Pages are similar to UIViewControllers and WPF Pages. Different subclasses of pages serve different purposes.

NavigationPage

Navigation pages provide an encapsulation of navigation into a page. These pages are not effected on Windows Phone and fall back to global navigation. This is due to platform restrictions.

Navigation pages work by pushing and popping other Pages from them. Generally a user will want to be pushing their own subclasses of ContentPage.

ContentPage

ContentPage is the intersection between normal View's and Pages. ContentPage contains a property Content which can be set to any View.

MasterDetailPage

A page which contains two child pages called Master and Detail. This Page maps closely to a UISplitViewController.

TabbedPage
A page which contains multiple children pages that are then displayed as tabs.

CarouselPage
A page which contains multiple children pages which are displayed in a carousel which can be swiped through one at a time by the user.

Animations
Animations are done through the View class as extension methods. Animations performed on a view may not be respected by the parent layout when a relayout occurs. In order to animate views effectively, they should either be used in an absolute position context or from within a Layout. Example usage:

```
	view.MoveTo (10, 20); // Move to X = 10, Y = 20;
```

`Length and Easing values can also optionally provided to modify the length and easing of the animation. Custom easing functions can also be implemented.`

```
	view.RotateTo (180, length: 300, easing: Easing.CubicInOut);
```

Bindings
Simple Usage

Bindings in Cell Templates
Bindings provide a convenient way to map data from objects into views. This is used both for mapping data and for supporting the MVVM pattern. A simple usage of bindings in a ListView:

```
	var items = new [] {
		new {Name = “Xander”, Title = “Monkey”},
		new {Name = “John”, Title = “Boss”},
		new {Name = “Fred”, Title = “Pleb”}
	};
	var listView = new ListView {
		ItemSource = items,
		Template = new CellTemplate (typeof (TextCell))
	};
	listView.Template.AddBinding (new Binding (TextCell.TextProperty, “Name”));
	listView.Template.AddBinding (new Binding (TextCell.DetailProperty, “Title”));
```

`This code results in the production of a list which has bound the Name and Title properties into the Text and Detail properties of the TextCells.`

Simple App Walkthrough
Creating your first hello world application is as simple as:

```
public class MyApp : ContentPage
{
	Public MyApp ()
	{
		Content = new Button {"Hello World"};
	}
}
```

In this example we have a subclass of ContentPage and assigned the Content in the constructor. When designing your own app, it is advised to not subclass any Page except for Page or ContentPage. This will enable maximum flexibility for mixing in different navigation models for different platforms as needed.

To add a new page to our app we first need to make our root page content contain a NavigationPage.

```
public class FirstPage : ContentPage
{
	Public FirstPage ()
	{
		Content = new Button {"Hello World"};
	}
}

public class HelloWorldApp : ContentPage
{
	Public HelloWorldApp ()
	{
		Content = NavigationPage (new FirstPage ());
	}
}
```

Our application now contains to classes. One which is the container for the entire app, and then a page which we push into a NavigationPage. In order to add a second page to our app we will fist define that page.

```
public class SecondPage : ContentPage
{
	Public SecondPage ()
	{
		var btn = new Button {"Go Back To First"};
		btn.Activated += (s, e) => Navigation.Pop ();
		Content = btn;
	}
}
```

And then push it when the button on the first page is pressed.

```
public class FirstPage : ContentPage
```

```
{
        Public FirstPage ()
        {
                var btn = new Button {“Hello World”};
                btn.Activated += (s, e) => Navigation.Push (new SecondPage ());
                Content = btn;

        }
}
```

Advanced Topics

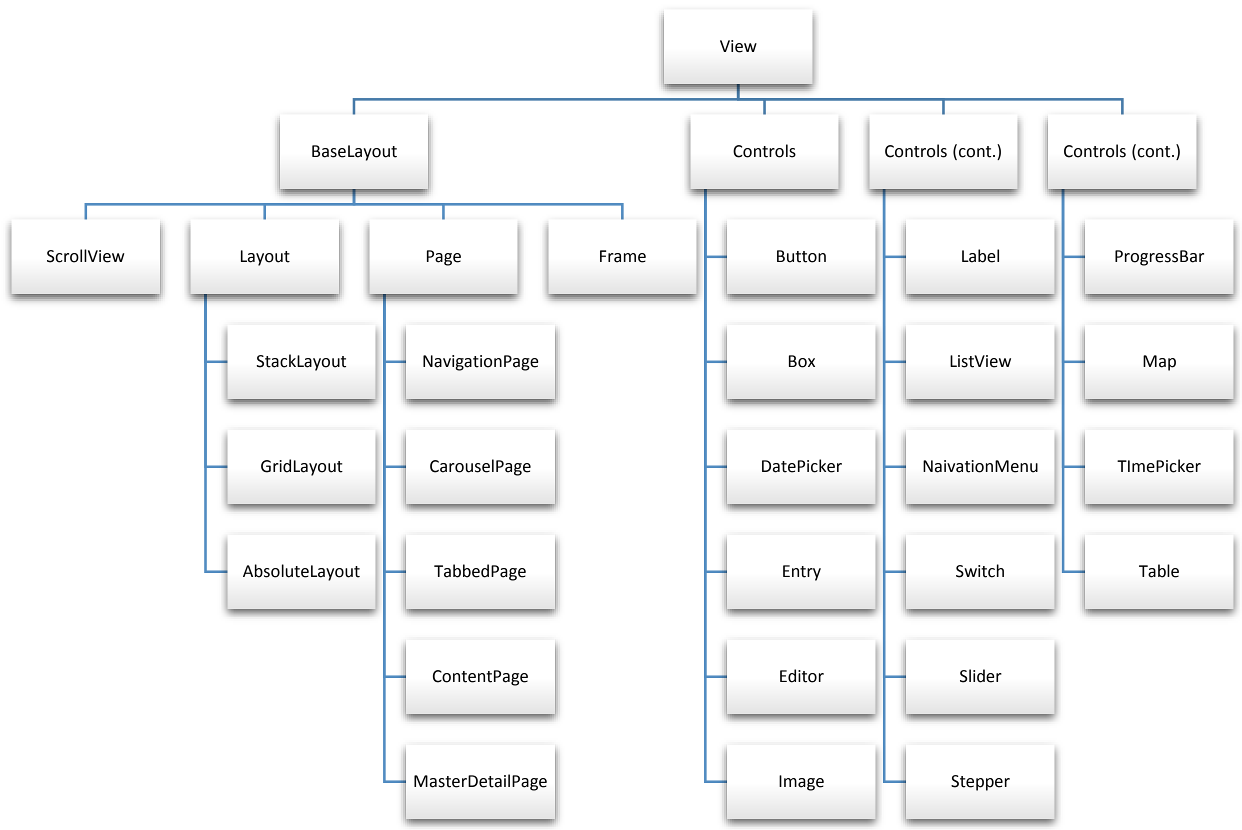
Custom Renderers

Custom renderer’s provide the ability to use platform specific code to either modify the way existing QuickUI views are rendered or to provide renderers for custom subclasses of a QuickUI.View. Creating a custom renderer requires implementing code in either a platform specific lib or directly in the platform specific App project. An example an iOS custom renderer will look like:

```
[assembly: ExportRenderer (typeof (MyQuickUIView), typeof (MyCustomRenderer))]
public class MyCustomRenderer : NativeRenderer
{
        protected override void OnModelSet (View view)
        {
                base.OnModelSet (view);
                var myNativeObject = new CustomControl ();
                SetNativeControl (myNativeObject);
        }
}
```

More complicated renderers will need to override methods to handle property change events and communicate data back with the set model. All three platforms follow the same basic model for custom renderers with adaptions for the platform specfic needs. The attribute provided would normally be placed in the AssemblyInfo.cs file. The attribute associates the renderer with the custom View the consumer has defined.

View Hierarchy



View
BaseLayout
Controls
Controls (cont.)
Controls (cont.)
ScrollView
Layout
Page
Frame
Button
Label
ProgressBar
StackLayout
NavigationPage
Box
ListView
Map
GridLayout
CarouselPage
DatePicker
NaivationMenu
TImePicker
AbsoluteLayout
TabbedPage
Entry
Switch
Table
ContentPage
Editor
Slider
MasterDetailPage
Image
Stepper