Customizing Controls for Each Platform

Subclassing Xamarin.Forms Control Renderers to access native SDK features

Overview

This article provides a simple of example of creating a custom control renderer for Xamarin.Forms, allowing developers to override the default native rendering of a Xamarin.Forms control with their own platform-specific code. Custom renderers provide complete flexibility in how controls look and behave, and can include platform-specific code to implement native SDK features.

How Renderers Work

Xamarin.Forms Pages, Layouts and Controls represent 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 which 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.

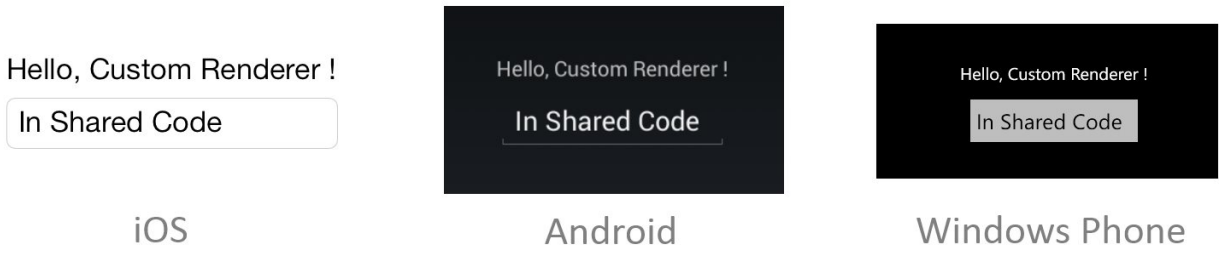
Note: The renderer APIs are not yet final. The team is still working on refining the experience to make it even easier and more powerful.

Writing a Custom Renderer

The simplest custom renderer can just update visual properties of a control specific to the platform, such as slightly altering the appearance of a text input box (Xamarin.Forms.Entry class). Before any customization, this Xamarin.Forms code:

```
new Entry {Text = "In Shared Code"}
```

is rendered like this on iOS, Android, and Windows Phone:



iOS Android Windows Phone

Steps to Implement a Custom Renderer

First, create a subclass of the control you wish to customize. Because the customization will be done in the renderer for this class, no additional implementation is required so the class body is empty.

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

Then use that `MyEntry` class in a layout. The code below will instantiate a new `ContentPage` object that will display the custom class centered both vertically and horizontally on the page (along with a standard `Label`).

```
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,
    };
  }
```

```
}
```

With the shared code written, a custom renderer can be added to each application project to customize the control's appearance on each platform.

iOS Implementation

```
public class MyEntryRenderer : EntryRenderer
{
  // Override the OnElementChanged method so we can tweak this renderer post-initial setup
  protected override void OnElementChanged (ElementChangedEventArgs<Entry> e)
  {
    base.OnElementChanged (e));
    if (e.OldElement == null) {   // perform initial setup
      // lets get a reference to the native control
      var nativeTextField = (UITextField) Control;
      // do whatever you want to the UITextField here!
      nativeTextField.BackgroundColor = UIColor.LightGray;
      nativeTextField.BorderStyle = UITextBorderStyle.Line;
    }
  }
}
```

Finally, add this `[assembly]` attribute above the class (and outside any namespaces that have been defined). Add `using` statements if required so that the `MyEntry` and `MyEntryRenderer` types are resolved.

```
[assembly: ExportRenderer (typeof (MyEntry), typeof (MyEntryRenderer))]
```

Android Implementation

```
public class MyEntryRenderer : EntryRenderer
{
  // Override the OnElementChanged method so we can tweak this renderer post-initial setup
  protected override void OnElementChanged (ElementChangedEventArgs<Entry> e)
  {
    base.OnElementChanged (e));
```

```
    if (e.OldElement == null) {   // perform initial setup
      // lets get a reference to the native control
      var nativeEditText = (global::Android.Widget.EditText) Control;
      // do whatever you want to the textField here!

nativeEditText.SetBackgroundColor(global::Android.Graphics.Color.DarkGray);
    }
  }
}
```

Once again add this `[assembly]` attribute above the class (and outside any namespaces that have been defined), including any required `using` statements.

```
[assembly: ExportRenderer (typeof (MyEntry), typeof (MyEntryRenderer))]
```

Windows Phone Implementation

The Windows Phone custom renderer for `Entry` demonstrates the need to understand the built-in renderers when subclassing. The Windows Phone `EntryRenderer` uses two native controls to provide plain text as well as password masking functionality, which affects how the `OnElementChanged` method obtains a native reference to modify. This sample code demonstrates that the `Control` isn't a `PhoneTextBox` as you might expect, it's a `Canvas` with two children; the first child is the `PhoneTextBox` that is modified.

```
public class MyEntryRenderer : EntryRenderer
{
  protected override void OnElementChanged(ElementChangedEventArgs<Entry> e)
  {
    base.OnElementChanged(e);

    if (e.OldElement == null)
    {
      var nativePhoneTextBox = (PhoneTextBox)Control.Children[0];
      //var nativePasswordBox = (PhoneTextBox)Control.Children[1]; // modify if
you're using IsPassword=true in Xamarin.Forms code
      nativePhoneTextBox.Background = new SolidColorBrush(Colors.Yellow);
    }
  }
}
```

Once again add this `[assembly]` attribute above the class (and outside any namespaces that have been defined), including any required `using` statements.

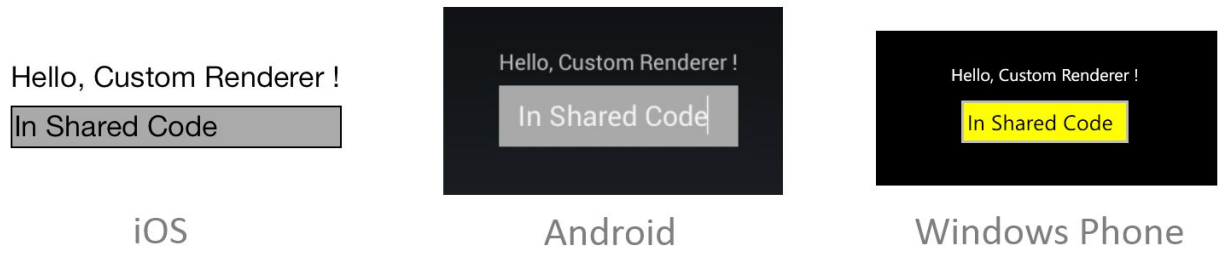
```
[assembly: ExportRenderer (typeof (MyEntry), typeof (MyEntryRenderer))]
```

Implementing in Xamarin.Forms Shared Code

Now we can update our Xamarin.Forms shared code to use the custom class:

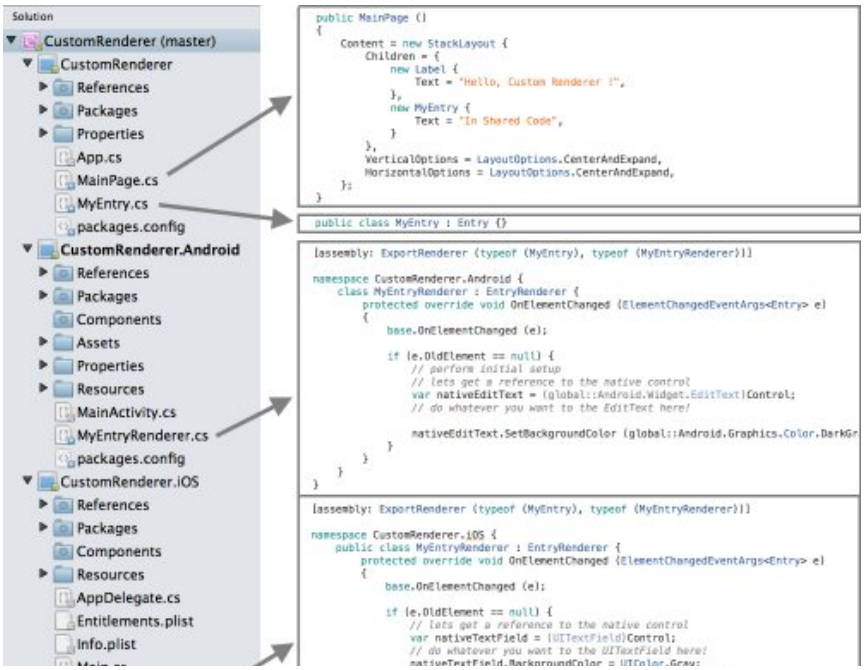
```
new MyEntry {Text = "In Shared Code"}
```

and it will be rendered in a platform-specific way on iOS and Android, as shown here:



Solution Structure

The [sample CustomRenderer solution] is shown below for iOS and Android, with the code changes outlined above highlighted.



Note that it is optional to provide a custom renderer in each platform project. If no custom renderer is registered then the default renderer for the control's base class will be used.

Custom Controls in Xaml

Custom controls (such as those with a custom renderer) can be referenced in Xaml by declaring a namespace for their location and using that prefix on the control element. For the sample CustomRenderer solution the namespace definition is shown below. The `local` prefix can be anything you choose; the `namespace` and `assembly` values should match the details of the custom controls (they are not necessarily identical, as they are in this example).

```
xmlns:local="clr-namespace:CustomRenderer;assembly=CustomRenderer"
```

Once the namespace has been declared, use the prefix to specify the correct type in the Xaml, as shown here:

```
<local:MyEntry Text="In Shared Xaml" />
```

The complete Xaml for the sample app looks like this:

```
<?xml version="1.0" encoding="UTF-8" ?>
<ContentPage
    xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:CustomRenderer;assembly=CustomRenderer"
    x:Class="CustomRenderer.MainPageXaml">
    <StackLayout
        VerticalOptions="CenterAndExpand"
        HorizontalOptions="CenterAndExpand">
        <Label Text="Hello, Custom Renderer!" />
        <local:MyEntry Text="In Shared Xaml" />
    </StackLayout>
</ContentPage>
```

Defining the `xmlns` is much simpler in PCLs than Shared Projects. A PCL is compiled into an assembly so it's easy to determine what the `assembly=CustomRenderer` value should be. When using Shared Projects all the shared assets (including the Xaml) are compiled into each of the referencing projects, which

means that if the iOS, Android and Windows Phone projects have their own *assembly names* it is impossible to to write the `xmlns` declaration because the value needs to be different for each application. Custom controls in Xaml for Shared Projects will require every application project to be configured with the same assembly name.

Summary

Custom renderers are a powerful way to customize Xamarin.Forms controls to implement native user interface features. They can be used for small styling changes or sophisticated platform-specific layout and behavior customization.