Filling the Gaps:

HOW TO BUILD A CUSTOM CONTROL IN XAMARIN.FORMS

KELLEY RICKER

GrapeCity®

Abstract

Xamarin.Forms provides a flexible, code-once option for developers to create native mobile apps, and it provides a nice array of controls to drop into your app. Third-party products like Xuni can extend app customization with complex controls like data grids, charts, and calendars. But if your requirements include a control that's not available on the market, you're going to need to build it yourself.

Custom controls can make your apps look better, gain novel functionality, or fill in some gap in the functionality that the Xamarin.Forms API provides. This document walks you through the big picture of what's possible with several different approaches.

You'll learn how to:

1. Extend the out-of-the-box Xamarin.Forms controls
2. Create controls using the Xamarin Platform Projects
    a. Use custom renderers for basic UI tweaks on a per-platform basis
    b. Use a DependencyService to access platform-specific features
3. Create a fully native control and connect it to Xamarin.Forms
    a. Develop a native iOS control
    b. Bind a native control to Xamarin
    c. Use a custom renderer to interact with custom native controls

We'll examine the pros and cons of each approach along with the time and experience necessary for each. We'll also cover some basic examples that visualize how each approach works.

Table of Contents

1. Introduction

It might not be clear why you'd ever want to delve into the realm of developing your own Xamarin.Forms controls, since many common ones are already included. While Xamarin provides a large amount of out-of-the-box functionality, there are definite gaps in what they've made available. More complex controls such as chart, calendar, rich textbox, and data grid aren't present, which limits your app's potential.

Xamarin.Forms targets native platforms that are distinct entities. Android and iOS have very different APIs underneath Xamarin.Forms, and if you pull back the curtain, you'll see a great deal of unevenness in terms of what each platform provides. Xamarin.Forms goes out of its way to smooth this over, but what if you need to fill in a gap that they haven't filled? Sometimes you'll actually want (or even need) different behavior between platforms, and building your own control may be your only option.

Custom controls fill in some of these gaps to provide more advanced input, data visualization, and data manipulation (among an even greater list of possibilities). They don't always have to be advanced constructions; sometimes it's as simple as combining existing objects to get novel behavior from a hybrid control. You may also find that one of the native platforms has exactly what you need, while the other platform has no equivalent. In this case, a custom control may be able to fill in the gap.

In this article, we'll address some of the different avenues for creating custom controls for the Xamarin.Forms developer. We'll examine three possible options for creating these controls, and we'll try to address where each option makes the most sense, moving from basic tweaks in Xamarin.Forms to creating your own native control.

Note: When we touch on native development, the focus will be on iOS and Android controls since those are likely to be the least familiar to the average Xamarin.Forms developer.

1.1. Custom Control Possibilities in Xamarin.Forms

Xamarin.Forms provides many built-in controls for basic UI design, but some areas are definitely lacking. Since Xamarin.Forms maps to each platform's native controls, they skew toward the more basic and common controls shared among platforms. Complex controls enabling advanced data visualizations and data management aren't represented in this group of controls, either. Fortunately, the ingenious developer can take multiple approaches to add any missing functionality.

1. **Basic**: Extend the out-of-the-box Xamarin.Forms controls
2. **Intermediate**: Create controls using the Xamarin Platform Projects
    a. Use custom renderers for basic UI tweaks on a per-platform basis
    b. Use a DependencyService to access platform-specific features
3. **Advanced**: Create a fully native control and connect it to Xamarin.Forms

The best option will vary according to practicality and intent.

At the most **basic** level, Xamarin allows you to extend their controls, giving you the ability to add to and combine existing controls without writing any platform-specific code.

At the **intermediate** level, a developer can use a custom renderer to provide a generic control API in Xamarin.Forms that maps to specific native control behaviors in each platform project. This approach is more advanced and allows more freedom for providing platform-specific UI customization. It also allows you to stay within the comfort of C#, provided you have some knowledge of the native platform APIs you wish to use.

Additionally, you can use a DependencyService to access platform-specific features in your shared code. This comes into play when you need to perform a behind-the-scenes action (such as saving a file or accessing the device hardware), which is implemented differently on each platform.

The most **advanced** approach combines using a custom renderer with your own custom native control. This is the most powerful means of customization, but also requires that you get your hands dirty with some Java and/or Objective-C/Swift coding. Anything that can be accomplished natively can be done using this approach.

	Basic	**Intermediate**		**Advanced**
Approach	Extend	Custom renderer	DependencyService	Build a custom native control
Language	C#	C#	C#	Objective-C, Swift, Java
Platform Limitations	Xamarin.Forms	Xamarin.iOS / Xamarin.Android	Xamarin.iOS / Xamarin.Android	iOS, Android
Capabilities	Some ability for UI tweaks and access to device	Greater flexibility for UI tweaks	Access to most device APIs	Access to all UI and device APIs
Time Investment	Low	Medium	Medium	High

Table 1: A comparison of the different approaches to developing custom controls in Xamarin.Forms.

The Xamarin.Forms API is a subset of what's available in each of the Xamarin Platforms. If functionality is missing from Xamarin.Forms, you may be able to find it a step deeper in Xamarin.iOS or Xamarin.Android. Anything you can do in Swift, Objective-C, and Java you can do with Xamarin since they support 100% of the core platforms.

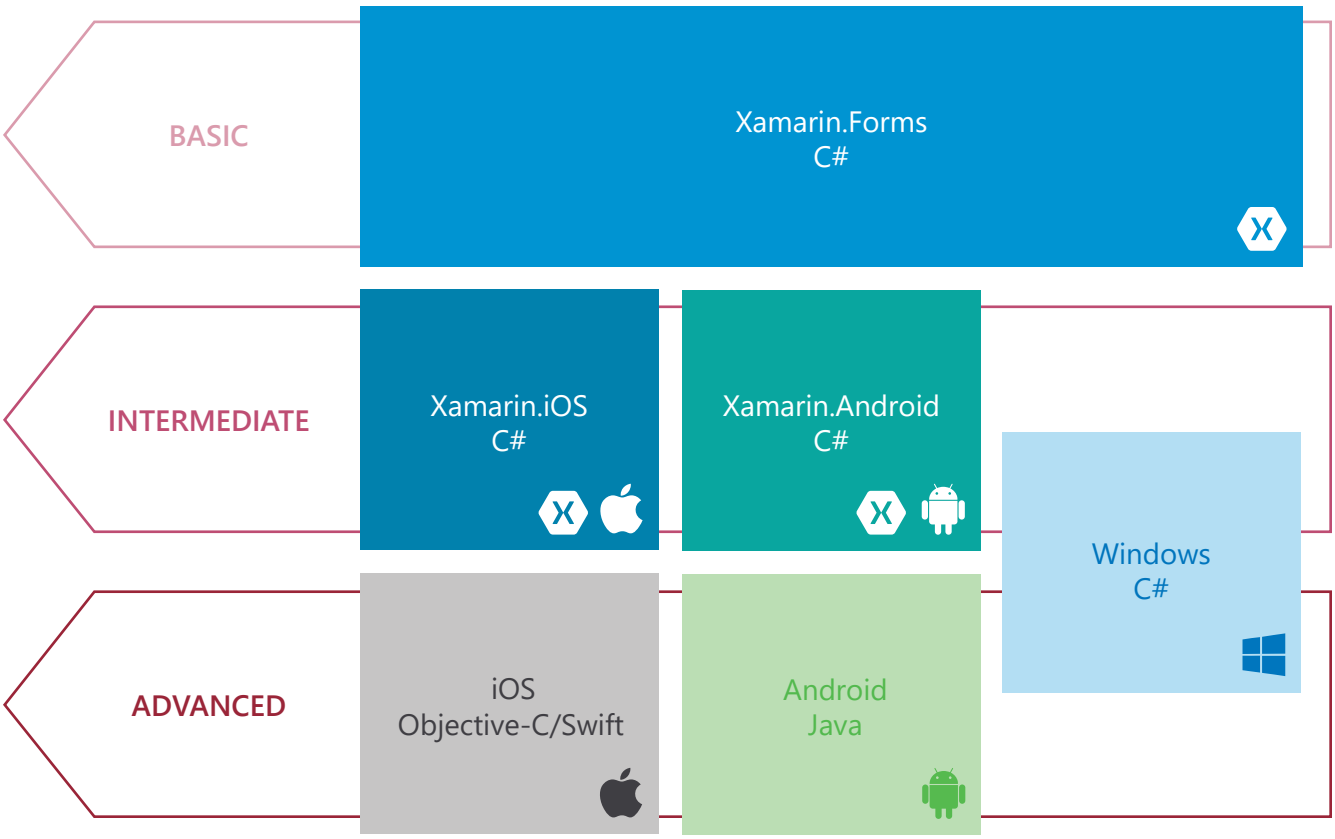


Figure 1: The relationship hierarchy of Xamarin.Forms, the Xamarin.Platforms, and the native platforms. The basic approach is applied at the Xamarin.Forms level, written entirely in C#. The intermediate approach is applied at the Xamarin.Platform level, written in C# with some knowledge of the native platform's API. The advanced approach is applied at the pure native level, and is written in the appropriate native platform language.

2. Basic Approach: Extending Out-of-the-Box Controls

Creating a custom control at the most basic level only requires you to extend what's available through Xamarin.Forms. This option is inherently limited, since you can only work with the preexisting controls in Xamarin.Forms, but it does allow you to combine Xamarin.Forms elements to create new hybrid controls.

A list of all Xamarin.Forms controls is available in the controls reference portion of Xamarin's documentation, so you can see what controls already exist and what can be extended (See section 6 for a link to "Xamarin.Forms Controls Reference.").

A new control such as a numeric text box can be created by combining the preexisting Entry and Stepper controls (Figure 2). We can extend these through a distinct numeric text box class that becomes a reusable object.

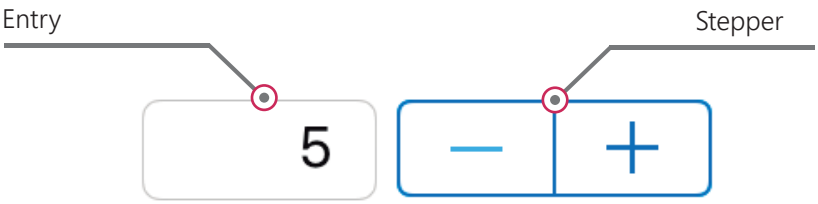


Figure 2: The individual controls that constitute a numeric textbox

Extending existing controls also allows you to work in C# and XAML with the Xamarin.Forms API. If you're new to the platform, or the most comfortable with .NET development, this is the easiest path for getting started. This type of control is built entirely in the portable or shared library project, so there's no need to venture into writing any platform-specific code in the iOS or Android modules. We'll start with this approach and cover its limitations later.

2.1. Example: Custom Color Picker

I'll illustrate one possibility using this approach: a custom color picker control. Xamarin.Forms doesn't have a color picker control, but by combining BoxView, Image, and Picker controls we can create a reusable component with the desired behavior. We can extend a basic View control to inherit all of the necessary layout and rendering capabilities.

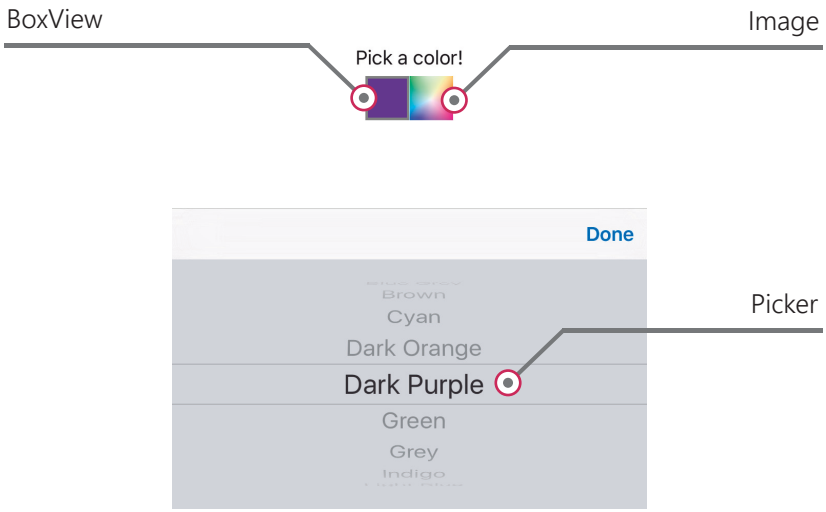


Figure 3: The individual controls that constitute a color picker

2.1.1. Designing the Appearance

Designing the appearance of the control can be done in XAML, much like any other UI design is done in Xamarin.Forms. The UI is basically a Grid with a BoxView, Image, and Picker control arranged inside it (Figure 3). The XAML aspect of the control design is straightforward:

```
<?xml version="1.0" encoding="utf-8" ?>
<Grid xmlns="http://xamarin.com/schemas/2014/forms"
      xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
      x:Class="ColorPickerApp.ColorPicker"
      ColumnSpacing="0" VerticalOptions="CenterAndExpand">
  <Grid.RowDefinitions>
    <RowDefinition Height="38"></RowDefinition>
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="38"></ColumnDefinition>
    <ColumnDefinition Width="38"></ColumnDefinition>
  </Grid.ColumnDefinitions>
  <Grid Grid.Row="0" Grid.Column="0" Padding="2" BackgroundColor="Gray"
        HorizontalOptions="FillAndExpand" VerticalOptions="FillAndExpand">
    <BoxView x:Name="bvColorPicker"></BoxView>
  </Grid>
  <Image x:Name="imgColorPicker" HorizontalOptions="FillAndExpand"
         VerticalOptions="FillAndExpand" Grid.Row="0" Grid.Column="1"/>
  <Picker x:Name="pickerColorPicker" IsVisible="False" Grid.ColumnSpan="2"
          SelectedIndexChanged="pickerColorPicker_SelectedIndexChanged"/>
</Grid>
```


As we mentioned at the beginning of this section, this type of custom control needs to extend a Xamarin.Forms View to inherit important layout and rendering capabilities. Since we've made the control layout a Grid, we can extend the Grid class in the code behind.

```
public partial class ColorPicker : Grid
{

}
```

2.1.2. Configuring the Code-Behind

The next step is to define the public properties that will be available for each instance of our control. The object model for this class will be relatively simple since we're only providing a single public property that controls the selected color value. (You can add other properties if desired.)

```
public Color Color
{
    get { return this.bvColorPicker.Color; }
    set { this.bvColorPicker.Color = value; }
}
```

The color property should now be configurable in code. You can also go a step further to make the property bindable if desired, though we won't cover that here.

The color picker needs to be initialized explicitly in code (similar to the Pages in your Xamarin.Forms project). The colors we'll have available for our control can also be placed into a dictionary.

Note: These color values are Xamarin.Color objects. Both iOS and Android have their own color objects, which are very different. The great thing about this approach is that we don't have to do anything special to deal with the platforms underneath Xamarin.Forms.

```
public ColorPicker()
{
    InitializeComponent();

    // set the image from embedded resource
    imgColorPicker.Source = ImageSource.FromResource("ColorPickerApp.ColorPicker.png");

    // populate picker with available colors
    foreach (string colorName in colorDict.Keys)
    {
        pickerColorPicker.Items.Add(colorName);
    }

}

// Dictionary to get Color from color name.
Dictionary<string, Color> colorDict = new Dictionary<string, Color>
{
    { "Default", Color.Default },                   { "Amber", Color.FromHex("#FFC107") },
    { "Black", Color.FromHex("#212121") },          { "Blue", Color.FromHex("#2196F3") },
    { "Blue Grey", Color.FromHex("#607D8B") },      { "Brown", Color.FromHex("#795548") },
    { "Cyan", Color.FromHex("#00BCD4") },           { "Dark Orange", Color.FromHex("#FF5722") },
    { "Dark Purple", Color.FromHex("#673AB7") },    { "Green", Color.FromHex("#4CAF50") },
    { "Grey", Color.FromHex("#9E9E9E") },           { "Indigo", Color.FromHex("#3F51B5") },
    { "Light Blue", Color.FromHex("#02A8F3") },     { "Light Green", Color.FromHex("#8AC249") },
    { "Lime", Color.FromHex("#CDDC39") },           { "Orange", Color.FromHex("#FF9800") },
    { "Pink", Color.FromHex("#E91E63") },           { "Purple", Color.FromHex("#94499D") },
    { "Red", Color.FromHex("#D32F2F") },            { "Teal", Color.FromHex("#009587") },
    { "White", Color.FromHex("#FFFFFF") },          { "Yellow", Color.FromHex("#FFEB3B") },
};
```

2.1.3. Handling User Input

The control requires that we also add some kind input handling, since the color picker is a generic Grid containing a BoxView and an Image. We can add a TapGestureRecognizer to the GestureRecognizers collection for our UI element to handle the gesture:

```
TapGestureRecognizer tapImgColorPicker = new TapGestureRecognizer();
imgColorPicker.GestureRecognizers.Add(tapImgColorPicker);
tapImgColorPicker.Tapped += TapImgColorPicker_Tapped;

private void TapImgColorPicker_Tapped(object sender, EventArgs e)
{
    pickerColorPicker.Focus();
}
```

Finally, we'll need to handle changing the color with the picker control. We can use the picker's selectedIndexChanged Event for this. In the method that handles this event, you can simply get the chosen color name from the picker. That value is used to determine the correct color to assign to the Color property of our control.

That's all there is to this approach. Everything that we've done here is standard C# that will be very familiar to a .NET developer.

2.2. Limitations

You're limited to making novel use of the elements provided in Xamarin.Forms, which doesn't offer any deeper access to the native platforms underneath Xamarin.Forms. To create controls that take advantage of platform-specific behaviors, you'll need to look to custom renderers and DependencyServices. We'll cover that next.

3. Intermediate Approach: Creating Controls Using the Xamarin Platform Projects

Extending what's available in Xamarin.Forms sometimes isn't enough. For example:

- You want to work with a control that requires more specific and sophisticated styling differences between platforms.
- You need separate implementations to handle the very different mechanisms each platform has for an action, like file saving (or otherwise directly accessing the device hardware).

Custom renderers and DependencyServices (respectively) can help fill these gaps.

Native platforms have their own APIs that allow finer control over the UI, and dictate how your app interacts with the device hardware. Custom renderers and DependencyServices allow you to tap directly into these native APIs that Xamarin has ported. The explicit difference:

- A custom renderer deals with the UI, as it's a mechanism for customizing the appearance and behavior of your controls.
- A DependencyService allows you to call into platform-specific functionality to access the device's hardware (camera, battery status, orientation, etc.).

Both methods require that you place specific tweaks for each platform into the platform projects in your Xamarin.Forms solution. These tweaks are written in C#, but they require some knowledge of the native platform's API. While both subjects are deep concepts in their own right, we'll discuss them together. Both enable a Xamarin.Forms developer to go a step further in manipulating the individual platforms, and offer more advanced custom controls.

3.1. Customizing Look and Behavior with Custom Renderers

Xamarin.Forms provides a common API for cross-platform mobile development. The controls, layouts, and pages that Xamarin provides are actually rendered differently to each platform by Renderer classes. These classes create a native control that corresponds to the control you've specified in your code through the Xamarin.Forms API.

For example, the Entry control in Xamarin.Forms uses the EntryRenderer class to map to a number of different native controls: UITextField for iOS, EditText for Android, and TextBox for Windows. Xamarin has a full list of their controls, renderers, and corresponding native controls available in their documentation (See section 6 for a link to "Render Base Classes and Native Controls."). Custom Renderer classes allow finer control over how a control appears and behaves on a specific platform. This provides deeper access to the native APIs and tweaks the behavior of your controls so they begin to feel unique to each platform.

3.1.1. Example: Custom Button Renderer

The general process for using a custom renderer is:

1. Create your custom control class in your Xamarin.Forms project.
2. Create a custom renderer for each platform.
3. Consume the control in Xamarin.Forms.

We'll introduce the concept by customizing the color of the built-in Button control for each platform. Now, this is something we could also accomplish in Xamarin.Forms, but it's better to show how the renderers work in a simple case. Another possibility is that you want to produce something more complex (such as a dropdown control) that relies on two different native controls (Popup on iOS and Spinner on Android). In that case, custom renderers act as a bridge across platforms that allow us to expose features previously inaccessible in Xamarin.

We'll create our own CustomButton class that inherits from the Button class in Xamarin.Forms (Figure 4). After that we'll use a custom renderer in each of our platform projects to change the appearance for that platform. The iOS and Droid custom renderers are built on Xamarin.iOS and Xamarin.Android respectively, while the Windows one is built on the Windows SDK.

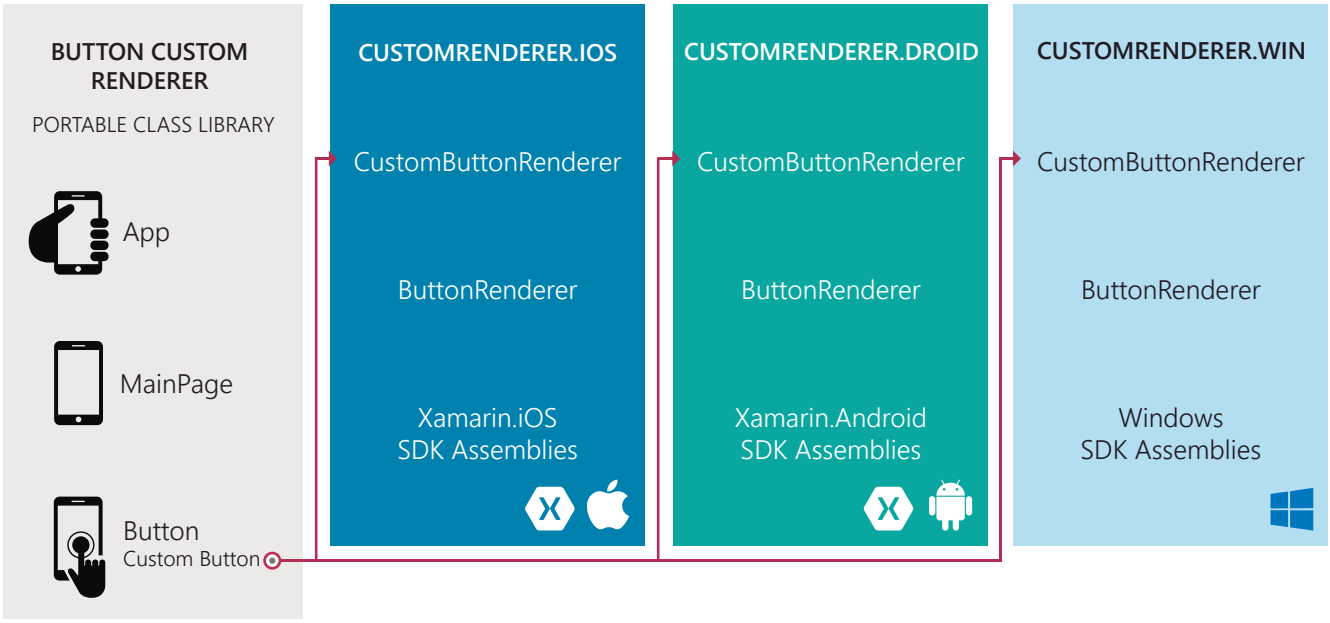


Figure 4: The relationship between custom control, custom renderer, and the individual platform projects. A CustomButton subclasses the Xamarin.Forms Button, and uses a custom renderer for each platform to dictate how it will appear.

3.1.1.1. Creating a CustomButton Class

First, create a CustomButton class inside of the Xamarin.Forms project that inherits from the Xamarin.Forms base Button control.

```
namespace CustomButtonExample
{
    public class CustomButton : Button
    {

    }
}
```

3.1.1.2. The Per-Platform Custom Renderers

The next step is to create the custom renderers for each platform. Each renderer will have some similarities:

- It will be a subclass of the ButtonRenderer.
- It overrides the OnElementChanged method to customize the control.
- It contains the ExportRenderer attribute to alert Xamarin.Forms to use this renderer when drawing the control.

3.1.1.3. Android Custom Renderer

We'll start with the Android CustomButtonRenderer, which we'll create in the Android platform project. On this platform, the button has a green background with black text.

```
[assembly: ExportRenderer (typeof(CustomButton), typeof(CustomButtonRenderer))]
namespace CustomButtonExample.Droid
{
    public class CustomButtonRenderer : ButtonRenderer
    {
        protected override void OnElementChanged (ElementChangedEventArgs<Button> e){
            base.OnElementChanged (e);
            if (Control != null) {
                Control.SetBackgroundColor (global::Android.Graphics.Color.LightGreen);
                Control.SetTextColor (global::Android.Graphics.Color.Black);
            }
        }
    }
}
```

The Android API has its own specific color objects and methods for setting color.

3.1.1.4. iOS Custom Renderer

Next, we'll move to the iOS platform project and create another CustomButtonRenderer. On this platform, the button has a blue background with white text. On iOS, we're using different methods and properties to set the color values (also platform-specific).

```
[assembly: ExportRenderer (typeof(CustomButton), typeof(CustomButtonRenderer))]
namespace CustomButtonExample.iOS
{
    public class CustomButtonRenderer: ButtonRenderer
    {
        protected override void OnElementChanged (ElementChangedEventArgs<Button> e){
            base.OnElementChanged (e);
            if (Control != null) {
                Control.BackgroundColor = UIColor.Blue;
                Control.SetTitleColor(UIColor.White, UIControlState.Normal);
            }
        }
    }
}
```

Though this is a relatively simple example, you can see the native APIs starting to show through.

3.1.1.5. Completed Control

When we use the CustomButton in Xamarin.Forms, we see two very differently styled results (Figure 5).

Figure 5: The completed buttons (iOS on left and Android on right)

Note: You can add a third custom renderer into your Windows project using the same concepts.

We'll revisit the topic of custom renderers later, when we go over creating your own native control and connecting it to Xamarin.Forms.

3.2. Using a DependencyService to Interact with the Device

DependencyServices are another mechanism for obtaining deeper access to the native platform and APIs. Xamarin.Forms gives you some ability to do per-platform configuration using the OnPlatform class, but that limits you to the Xamarin.Forms APIs. A DependencyService goes a step further into the native APIs.

Unlike a custom renderer, a DependencyService does not deal with appearance, but instead allows access to specific native platform features. They are effectively dependency resolvers, where an interface is defined in the common Xamarin.Forms project, and the DependencyService finds the appropriate interface implementation in the platform projects. They often come into play during direct interaction with a device's hardware. This allows Xamarin.Forms to produce apps that have the same abilities as native apps.

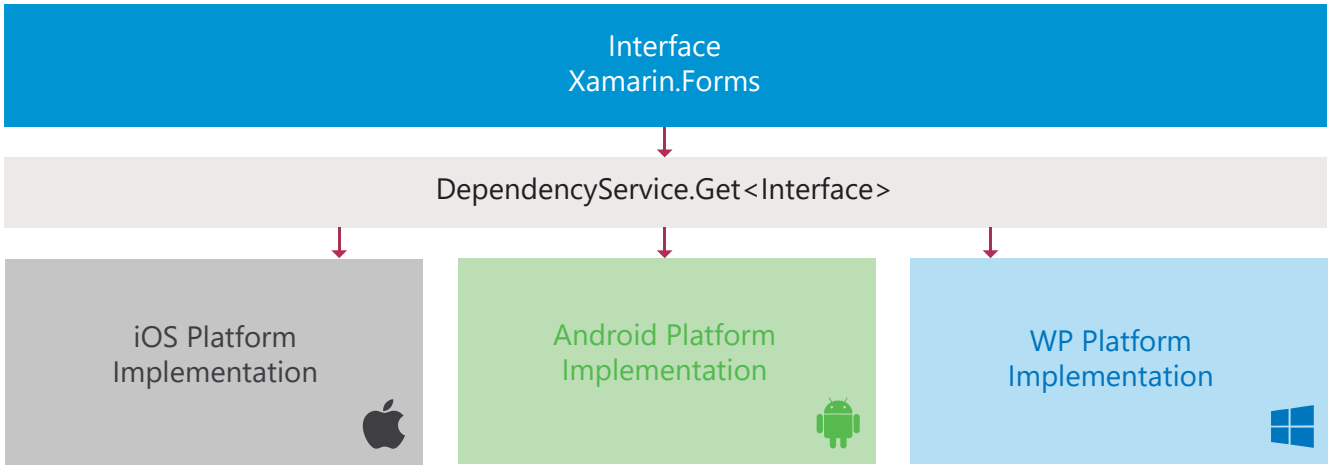


Figure 6: The relationship between Interface, DependencyService, and individual implementation. The interface is created in Xamarin.Forms and has individual implementations on each platform. A call to DependencyService.Get<Interface> will resolve to the correct implementation.

For a DependencyService to function, Xamarin.Forms apps need an interface and an implementation for each platform that is registered with the DependencyService attribute. When it's set up correctly, you can explicitly call DependencyService to access the appropriate platform-specific interface implementation (Figure 6).

3.2.1. Example: Saving an Image File Using a DependencyService in Xamarin.Forms

The platforms greatly differ in access to their file systems. File access on Android and Windows is much more open than on iOS. Apple, on the other hand, has a walled-garden approach to iOS; they allow access to only a few specific locations on the device. The variations between platforms make it a clear case for using a DependencyService.

3.2.1.1. Creating Xamarin.Forms Interface

The first step is to create an interface in the Xamarin.Forms portable or shared project. The interface doesn't need to be overly complicated since it's only going to save an image to the camera roll.

```
namespace FlexChartImageSave
{

    public interface IPicture
    {
        void SavePictureToDisk (string filename, byte[] imageData);

    }
}
```

To work, the interface needs specific implementations in each platform project. Each implementation will interact with a native API using the byte array (and potentially the filename) to save the image.

3.2.1.2. Android Implementation

We'll start with the implementation of the interface in the Android project. This is a two-step process on Android:

1. Save the picture to the external storage of the device.
2. Use the mediaScanIntent to add the saved image to the Gallery.

We'll also generate a timestamp to the file name to ensure unique image names. Finally, a Dependency attribute needs to be added at the top of the class so Xamarin.Forms can identify this as the appropriate implementation of our interface when we call the DependencyService.


```
[assembly: Xamarin.Forms.Dependency(typeof(Picture_Droid))]

namespace FlexChartImageSave.Droid
{
    public class Picture_Droid : IPicture
    {
        public void SavePictureToDisk(string filename, byte[] imageData)
        {
            var dir = Android.OS.Environment.GetExternalStoragePublicDirectory(Android.OS.Environment.DirectoryDcim);
            var pictures = dir.AbsolutePath;
            //adding a time stamp time file name to allow saving more than one image... otherwise it overwrites
            string name = filename + System.DateTime.Now.ToString("yyyyMMddHHmmssfff") + ".jpg";
            string filePath = System.IO.Path.Combine(pictures, name);
            try
            {
                System.IO.File.WriteAllBytes(filePath, imageData);
                //mediascan adds the saved image into the gallery
                var mediaScanIntent = new Intent(Intent.ActionMediaScannerScanFile);
                mediaScanIntent.SetData(Uri.FromFile(new File(filePath)));
                Xamarin.Forms.Forms.Context.SendBroadcast(mediaScanIntent);
            }
            catch(System.Exception e)
            {
                System.Console.WriteLine(e.ToString());
            }

        }
    }
}
```

Note: Xamarin.Forms requires specific permission to access storage on Android. To write an image to an Android device we'll also need to enable WRITE_EXTERNAL_STORAGE in the required permissions list of the Android Mani-

3.2.1.3. iOS Implementation

The implementation on iOS is actually a bit easier. Apple allows use of the SaveToPhotosAlbum method of the UIImage object to save your image object directly into your photos. This implementation also needs the Dependency attribute.

```
[assembly: Xamarin.Forms.Dependency(typeof(Picture_iOS))]

namespace FlexChartImageSave.iOS
{
    public class Picture_iOS: IPicture
    {
        public void SavePictureToDisk(string filename, byte[] imageData)
        {
            var chartImage = new UIImage(NSData.FromArray(imageData));
            chartImage.SaveToPhotosAlbum((image, error) =>
            {
                //you can retrieve the saved UI Image as well if needed using
                //var i = image as UIImage;
                if(error != null)
                {
                    Console.WriteLine(error.ToString());
                }
            });
        }
    }
}
```

iOS does not require any special permissions to save an image to the photo album, so this is all we need.

3.2.1.4. Using the DependencyService

Now, to actually use the interface we can call:

```
//uses the IPicture interface to use the appropriate saving mechanism from the picture class in each individual project
DependencyService.Get<IPicture>().SavePictureToDisk("ChartImage", flexChart.GetImage());
```

The DependencyService uses the appropriate implementation of IPicture for the current platform.

Note: The Windows implementation follows the same format.

3.3. Limitations

Custom renderers and DependencyServices both offer more than what's available in Xamarin.Forms, though you're still bound by what Xamarin provides. If you need something that isn't present in any of the Xamarin APIs, you need to work with some native code.

4. Advanced Approach: Creating a Fully-Native Control and Connecting to Xamarin.Forms

All of the approaches we've discussed are still inherently limited, since all of your code is within Xamarin. Native code has more freedom than Xamarin, and there comes a point where it makes sense to develop your own native control. For instance:

1. You need the greatest amount of flexibility.
2. You've inherited some preexisting native code that you need to incorporate into your project.
3. You need a native version of your library in addition to what's in your Xamarin.Forms application project.

For any of these reasons, you may find yourself developing a native control that you'd like to use in Xamarin.Forms. You can create your own control for a native platform, connect it to an individual Xamarin Platform (as in Xamarin.iOS or Xamarin.Android) via a Binding Library, and use a custom renderer to map a Xamarin.Forms control to your custom control.

Developing your own native control is the most complex approach, but it does give you the most power. You directly determine the native API and behavior, how that code is bound in Xamarin, and how the Xamarin control interacts with Xamarin.Forms. This is the most complete way to fill the gaps between Android and iOS controls.

4.1. Going Native

Many Xamarin developers won't be thrilled at the prospect of working with Java, and are absolutely repulsed by Objective-C. There are circumstances, though, where you may find yourself needing to work in the native platforms.

At the native level, you'll be deriving from objects built into the iOS and Android APIs. UIViews (iOS) and Views and ViewGroups (Android) are the most basic UI building blocks. It's a good idea to see what's available on each platform, so you may want to spend some time using the UIKit framework and Android libraries to familiarize yourself with these objects.

4.1.1. Example: Creating an iOS CheckBox

We'll examine the process of creating a control that is not included in iOS, but present in Android and Windows: the simple checkbox (Figure 7).

Figure 7: Completed iOS CheckBox controls

4.1.1.1. Planning the Control

You should consider several questions before you get started:

1. What are the most basic elements of your control?
2 What behaviors are necessary for the control to feel correct on this platform?
3 How does this control behave on other platforms?

Since we're going to be tying this control into Xamarin.Forms, carefully plan your API so that your control has the correct look and feel for iOS, but still has enough in common with the Android control that the experience feels similar in Xamarin.Forms. This means putting some care into what properties, methods, and events you'll need to make available when you create your native API later on.

A CheckBox control doesn't need to be overly complex, as it only involves a handful of elements. In our case, we'll focus on the following:

1. Box
2. Checkmark
3. Color
4. Enable/Disable
5. Touch Interaction (State Changes)

Because we're developing an iOS control, we need to work on a Mac that has Xcode installed.

4.1.1.2. Creating the Control

To get started, open up Xcode, and create a new static library project. Once created, the main CheckBox control class can subclass UIKit's built-in UIControl class to give us a basic structure. The UIControl class acts as the basis for all control objects such as buttons and sliders. We can use some of the tracking provided in the class to track touch interactions (which will help us later on).

The properties and methods for the CheckBox class correspond to elements we described above. I'm keeping the functionality limited here so that the later steps aren't overly complicated.

```
#import <Foundation/Foundation.h>
#import <UIKit/UIKit.h>

@interface CheckBox : UIControl

-(void)setChecked:(BOOL)isChecked;
-(void)setEnabled:(BOOL)isEnabled;

@property IBInspectable UIColor *color;
@property IBInspectable BOOL isEnabled;
@property IBInspectable BOOL isChecked;

@end
```

All iOS controls use the drawRect method to draw the control to the screen. These specific elements are drawn by filling in or drawing along UIBezierPaths. Two are present: one represents the box and the other the check. The position where the CheckBox is drawn must be consistent between platforms. Otherwise, when we use the control in Xamarin.Forms, we'll end up with controls that have more than minor cosmetic differences.

```
- (void)drawRect:(CGRect)rect {
    int width = 20;
    int height = 20;
    // Drawing code
    [checkColor setFill];
    [_color setStroke];
    UIBezierPath *boxPath = [UIBezierPath bezierPathWithRoundedRect:
                             CGRectMake(7, 2, width - 3, height - 3) cornerRadius:width/5];
    boxPath.lineWidth = 2;
    [boxPath fill];
    [boxPath stroke];
    if (_isChecked == YES) {
        UIBezierPath *checkPath = [UIBezierPath bezierPath];
        checkPath.lineWidth = 2;
        [checkPath moveToPoint:CGPointMake(width * 4/5 + 5, height/5)];
        [checkPath addLineToPoint:CGPointMake(width/2 + 5, height * 4/5)];
        [checkPath addLineToPoint:CGPointMake(width/5 + 5, height/2)];
        [_color setFill];
        [boxPath fill];
        [checkColor setStroke];
        [checkPath stroke];
    }
    //check if control is enabled...lower alpha if not and disable interaction
    if (_isEnabled == YES) {
        self.alpha = 1.0f;
        self.userInteractionEnabled = YES;
    }
    else{
        self.alpha = 0.6f;
        self.userInteractionEnabled = NO;
    }
    [self setNeedsDisplay];
}
```

The setter methods are a straightforward means of setting some properties.

Note: There is a method call to sendActionsForControlEvent that passes UIControlEventValueChanged. If you miss adding this method, you'll have trouble getting your valueChanged event to work in Xamarin.Forms.

```
-(BOOL)beginTrackingWithTouch:(UITouch *)touch withEvent:(UIEvent *)event{
    [self setChecked:!_isChecked];
    return true;
}

-(void)setChecked:(BOOL)isChecked{
    _isChecked = isChecked;
    [self sendActionsForControlEvents:UIControlEventValueChanged];
    [self setNeedsDisplay];
}
-(void)setEnabled:(BOOL)isEnabled{
    _isEnabled = isEnabled;
    [self setNeedsDisplay];
}
```

The CheckBox can easily be used in other native iOS projects by simply importing the project and the CheckBox header.

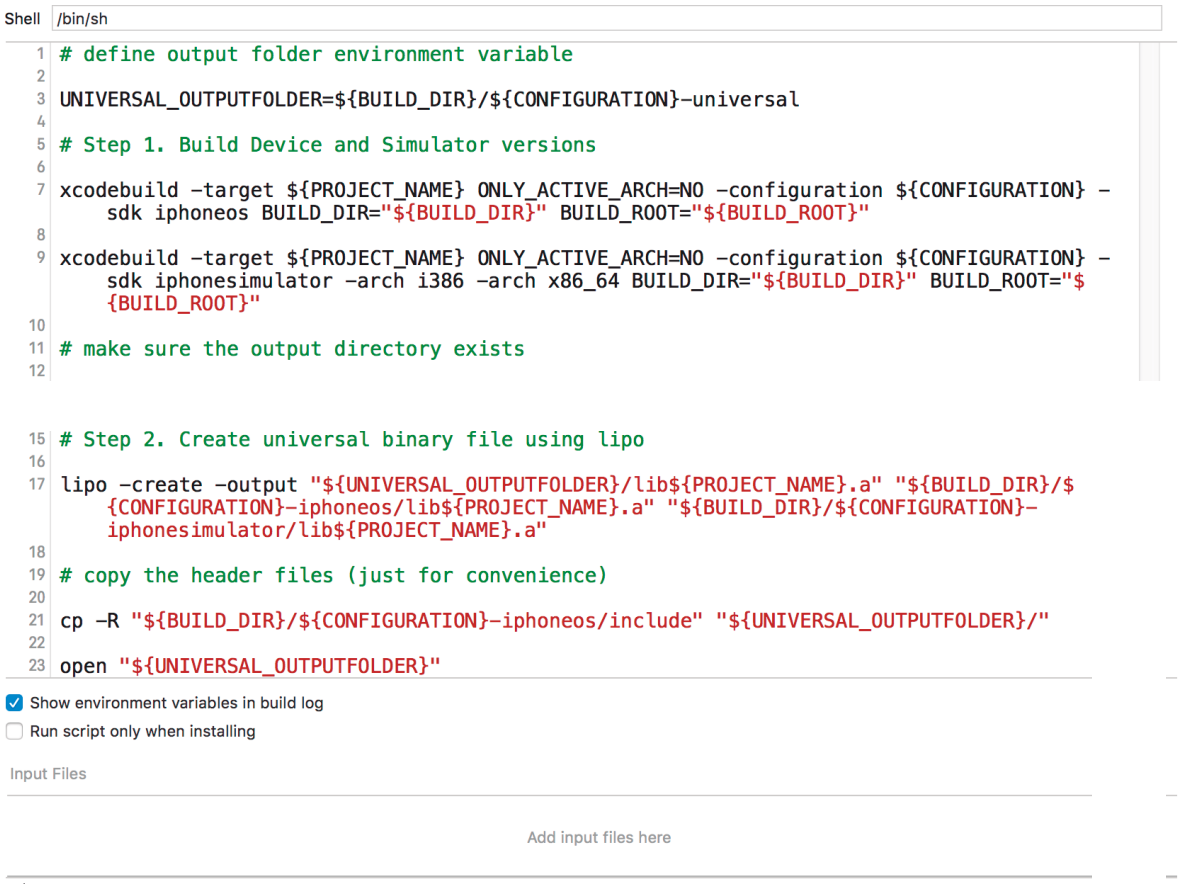
4.1.1.3. Creating an Aggregate Static Library

Since we're using this control in Xamarin, let's continue to the next step and create an aggregate target for our static library. Our compiled static library is currently only targeting one architecture (either x86 or ARM), and the final product needs to accommodate both. Apple provides a command line tool called lipo for exactly this function ("Mach-O Programming Topics"). In Xcode, adding an aggregate target allows us to use the lipo tool to combine both results into one universal library that we can connect to Xamarin.

The steps involved to create an aggregate static library include:

1. Add a new aggregate target to your project.
2. Navigate to Build Phases for the aggregate target.
3. Add a New Run Script Phase to your target.
4. Add a script that uses lipo to combine multiple version of your static library.
5. Build the project for your aggregate target.

Here's the script that you'll need to add to your project:



When you build for this aggregate target, the Finder will automatically open a folder with the Universal library archive. This archive is exactly what we need to connect the iOS control to Xamarin.

4.2. Bringing the Native Control into Xamarin

Getting the control to work in Xamarin.iOS is next. Xamarin.iOS will use a bindings library project to connect to the universal library we created earlier. A couple of files within the bindings library project enable this to work, namely:

1. **linkwith** file appears after the archive file has been copied into the project and communicates to Xamarin.iOS how linking the static library into the resulting DLL should be handled.
2. **ApiDefinitions** specifies how Xamarin.iOS and the native control interact.
3. **StructsAndEnums** contains any types, structs, or enums needed by the ApiDefinition.

Xamarin provides a tool called Objective Sharpie that automatically generates the ApiDefinition and StructsAndEnums files from the header file(s) within the native CheckBox control project.

Objective Sharpie can be obtained directly from Xamarin (See section 6 for link to "Objective Sharpie."). It's a simple command line tool (run directly from the Terminal application on a Mac) that can save a lot of time when you're initially trying to create your ApiDefinition files.

4.2.1. Example: Using Objective Sharpie

Ultimately, you can generate these files using the command:

```
sharpie bind --output=CheckBox --namespace=iOSCheckBox --
sdk=iphoneos9.2 <Path-To-Project> /CheckBox.h
```

The files may need some changes, depending on the complexity of your control. For a control as simple as a CheckBox, the ApiDefinition created by Objective Sharpie will be fine as is.

The resulting ApiDefinition file shows the relationship between the Xamarin properties/method calls and native ones. The [BaseType] attribute specifies the parent class (UIControl), and the [Export] attribute denotes the Objective-C method or property that is being bound to the C# API:

```
namespace iOSCheckBox
{
    // @interface CheckBox : UIControl
    [BaseType (typeof(UIControl))]
    interface CheckBox
    {
        // -(void)setChecked:(BOOL)isChecked;
        [Export ("setChecked:")]
        void SetChecked (bool isChecked);

        // -(void)setEnabled:(BOOL)isEnabled;
        [Export ("setEnabled:")]
        void SetEnabled (bool isEnabled);

        // @property UIColor * color;
        [Export ("color", ArgumentSemantic.Assign)]
        UIColor Color { get; set; }

        // @property BOOL isEnabled;
        [Export ("isEnabled")]
        bool IsEnabled { get; set; }

        // @property BOOL isChecked;
        [Export ("isChecked")]
        bool IsChecked { get; set; }
    }
}
```

The control can now be used in Xamarin.iOS, either by importing the bindings library project into another solution, or by sharing the resulting DLL. The usage may be closer to native iOS than most Xamarin.Forms and .NET developers are used to, but the applications can be fully developed in C#.

Now we'll make the control usable in Xamarin.Forms by revisiting custom renderers.

4.3. Using Custom Renderers to Connect Native Controls to Xamarin Forms

Custom renderers are also important when connecting your own native control to Xamarin.Forms. Now that we have a CheckBox control available in Xamarin.iOS, we can use a custom renderer to create a Xamarin.Forms CheckBox object and tie it to these two independent platform controls. It's the same concept we explored earlier, just with the additional wrinkle of working with a new native control.

4.3.1. Example: CheckBox Custom Renderer

The premise is the same as before: create a common CheckBox control class in the Xamarin.Forms project, and then create custom renderers for this control in each platform project (Figure 8).

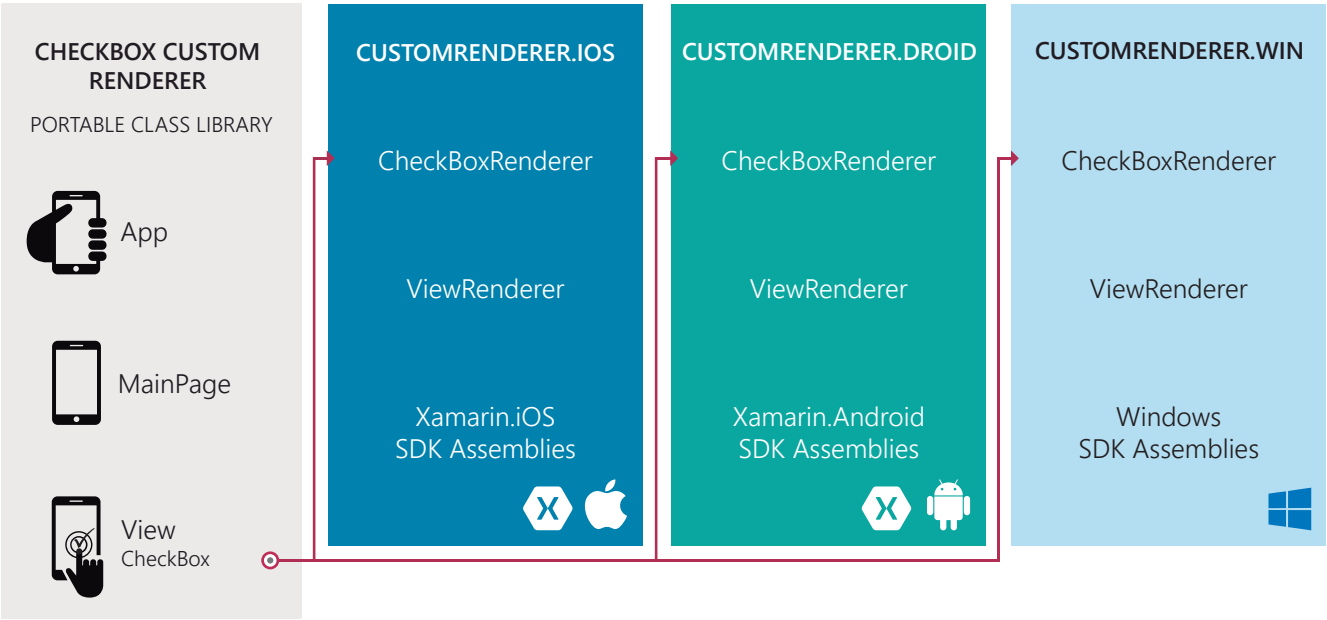


Figure 8: The relationship between the CheckBox control, custom renderer, and the individual platform projects. A CheckBox subclasses the Xamarin.Forms View and uses a custom renderer for each platform to dictate how it will appear.

4.3.1.1. Xamarin.Forms CheckBox Class

The CheckBox control class includes a couple of bindable properties (IsChecked and Color), along with an EventHandler that captures when the control becomes checked. The CheckBox will also inherit from the Xamarin.Forms View class since it affords the most flexibility.

```
namespace CheckboxCustomRenderer
{
    public class Checkbox : View
    {
        public static readonly BindableProperty IsCheckedProperty = BindableProperty.Create<Checkbox, bool>
        (p => p.IsChecked, true, propertyChanged: (s, o, n) => { (s as Checkbox).OnChecked(new EventArgs()); });
        public static readonly BindableProperty ColorProperty = BindableProperty.Create<Checkbox, Color>
        (p => p.Color, Color.Default);

        public bool IsChecked {
            get {
                return (bool)GetValue (IsCheckedProperty);
            }
            set {
                SetValue (IsCheckedProperty, value);
            }
        }

        public Color Color{
            get{
                return (Color)GetValue(ColorProperty);
            }
            set{
                SetValue(ColorProperty, value);
            }
        }

        public event EventHandler Checked;

        protected virtual void OnChecked(EventArgs e){
            if (Checked != null)
                Checked(this, e);
        }
    }
}
```

4.3.1.2. Android Custom Renderer

Starting with the Android custom renderer, we'll follow this general process:

- Subclass the ViewRenderer.
- Override the OnElementChanged method to instantiate and configure the native control.
- Use the ExportRenderer attribute to alert Xamarin.Forms to use this renderer when drawing the control.

There's some added complexity in this renderer since we need to configure the native control when the bindable properties change. We also need to provide a listener for capturing when the CheckBox is tapped. Finally, the CheckBox requires a ColorStateList with different color values for each possible CheckBox state.

```
[assembly: ExportRenderer(typeof(CheckboxCustomRenderer.Checkbox), typeof(CheckboxRenderer))]
namespace CheckboxCustomRenderer.Droid
{
    public class CheckboxRenderer : ViewRenderer<Checkbox, CheckBox>
    {
        private CheckBox checkBox;
        protected override void OnElementChanged(ElementChangedEventArgs<Checkbox> e){
            base.OnElementChanged(e);
            var model = e.NewElement;
            checkBox = new CheckBox(Context);
            checkBox.Tag = this;
            CheckboxPropertyChanged(model, null);
            checkBox.SetOnClickListener(new ClickListener(model));
            SetNativeControl(checkBox);
        }
        private void CheckboxPropertyChanged(Checkbox model, String propertyName)
        {
            if(propertyName == null || Checkbox.IsCheckedProperty.PropertyName == propertyName){
                checkBox.Checked = model.IsChecked;
            }
            if (propertyName == null || Checkbox.ColorProperty.PropertyName == propertyName){
                int[][] states = {
                    new int[] { Android.Resource.Attribute.StateEnabled}, // enabled
                    new int[] {Android.Resource.Attribute.StateEnabled}, // disabled
                    new int[] {Android.Resource.Attribute.StateChecked}, // unchecked
                    new int[] { Android.Resource.Attribute.StatePressed}  // pressed
                };
                var checkBoxColor = (int)model.Color.ToAndroid ();
                int[] colors = {checkBoxColor, checkBoxColor, checkBoxColor, checkBoxColor};
                var myList = new Android.Content.Res.ColorStateList(states, colors);
                checkBox.ButtonTintList = myList;
            }
        }
        protected override void OnElementPropertyChanged(object sender, PropertyChangedEventArgs e){
            if (checkBox != null){
                base.OnElementPropertyChanged(sender, e);
                CheckboxPropertyChanged((Checkbox)sender, e.PropertyName);
            }
        }

        public class ClickListener : Java.Lang.Object, IOnClickListener{
            private Checkbox _myCheckbox;
            public ClickListener(Checkbox myCheckbox){
                this._myCheckbox = myCheckbox;
            }
            public void OnClick(global::Android.Views.View v){
                _myCheckbox.IsChecked = !_myCheckbox.IsChecked;
            }
        }
    }
}
```

4.3.1.3. iOS Custom Renderer

The iOS custom renderer also needs to be registered with Xamarin.Forms, and most of the work is done in OnElementChanged. We do see some minor code differences between iOS and Android. Color is a single value for the iOS CheckBox, although it still requires a conversion from a Xamarin Color to a UIColor.

```
[assembly: ExportRenderer (typeof(CheckboxCustomRenderer.Checkbox), typeof(CheckboxRenderer))]
namespace CheckboxCustomRenderer.iOS
{
    public class CheckboxRenderer : ViewRenderer<Checkbox, iOSCheckBox.CheckBox>
    {
        private iOSCheckBox.CheckBox nativeCheckbox;

        protected override void OnElementChanged (ElementChangedEventArgs<CheckboxCustomRenderer.Checkbox> e)
        {
            base.OnElementChanged (e);
            var model = e.NewElement;
            if (model == null) {
                return;
            }

            nativeCheckbox = new iOSCheckBox.CheckBox ();
            CheckboxPropertyChanged (model, null);
            model.PropertyChanged += OnElementPropertyChanged;

            nativeCheckbox.ValueChanged += (object sender, EventArgs eargs) => {
                model.IsChecked = nativeCheckbox.IsChecked;
            };
            SetNativeControl (nativeCheckbox);
        }
        private void CheckboxPropertyChanged(Checkbox model, String propertyName){
            if (propertyName == null ||  propertyName == Checkbox.IsCheckedProperty.PropertyName) {
                nativeCheckbox.IsChecked = model.IsChecked;
            }
            if (propertyName == null || propertyName == Checkbox.ColorProperty.PropertyName) {
                nativeCheckbox.Color = model.Color.ToUIColor ();
            }
        }
        protected override void OnElementPropertyChanged(object sender, PropertyChangedEventArgs e)
        {
            if (nativeCheckbox != null)
            {
                base.OnElementPropertyChanged(sender, e);

                CheckboxPropertyChanged((Checkbox)sender, e.PropertyName);
            }
        }
    }
}
```

Once both custom renderers are complete, you can use your checkbox in Xamarin.Forms in XAML or C#. To use the control in XAML, create a new Xamarin.Forms page, add a namespace with a reference to the assembly, and use it as you would any other control.

```
<?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:CheckboxCustomRenderer;assembly=CheckboxCustomRenderer"
  x:Class="CheckboxCustomRenderer.TestPage">
    <ContentPage.Content>
    <StackLayout VerticalOptions="Center">
    <local:Checkbox Color="Green" WidthRequest="24" HeightRequest="24"/>
    </StackLayout>
    </ContentPage.Content>
</ContentPage>
```

Configuring the control in code-behind is equally straightforward:

```
public App ()
{
    MainPage = new ContentPage {
        Content = new StackLayout {
            VerticalOptions = LayoutOptions.Center,
            Children = {
                 new CheckboxCustomRenderer.Checkbox(){
                    WidthRequest = 50,
                    HeightRequest = 100,
                    Color = Xamarin.Forms.Color.Red
                }
            }
        }
    };
}
```

Visually, both controls will look and behave similarly (since we designed with that in mind). Figure 9 demonstrates how similar the CheckBox control appears, regardless of platform.

Figure 9: Completed CheckBox controls in Xamarin.Forms.
The Android version is in black and the iOS version is in white.

4.4. Limitations

There are no limitations to this method. Anything that is possible in native is possible through this approach.

5. Conclusion: Deciding How to Develop Your Own Controls

Which finally brings us to the question: which approach is best for your custom control requirements? There isn't a one-size-fits-all answer to the question. Your decision depends on the complexity of your requirement vs. your ability to use the native APIs in the Xamarin.Platforms, or write native Objective-C, Swift, or Java code.

We've presented three major avenues that give you varying degrees of capability. Creating a custom control in Xamarin.Forms is the quickest option, though the most limited. If your custom control isn't possible in Xamarin.Forms alone (the basic approach), then look into creating a custom renderer for each Xamarin Platform project before going the complete native approach. While the intermediate and advanced approaches allow you to augment Xamarin.Forms in powerful ways, the intermediate requires some knowledge of the native APIs, and the advanced requires comfort with writing native code.

5.1. Third-Party Controls

If developing a custom control seems like too much effort, you could go with an off-the-shelf solution. A third-party control library can potentially provide the exact functionality you need while freeing you from any maintenance concerns in the future. These controls often have better and more consistent documentation than open source, and more access to short-term and long-term support.

This is where our product, Xuni, fits into the picture (Figure 10). For the most complicated types of controls, such as charts and data grids, third-party solutions only require that the developer learns the control API, rather than worrying about anything underneath. While this may limit customization (compared to designing your own control or using open source software), it also frees you from the responsibility of any problems that may arise within the control, since bug fixes and feature additions are part of the third-party control package.

A control set like Xuni fills significant gaps in what's available within the Xamarin.Forms API, and, as such, targets specific areas where custom controls would be necessary. Some of the most advanced control behavior may already be covered in existing libraries. Fundamental issues that might cause you to develop custom controls may already be solved. While many tools exist to help solve a given development challenge, it's always worth looking to see if someone else has already solved the problem.

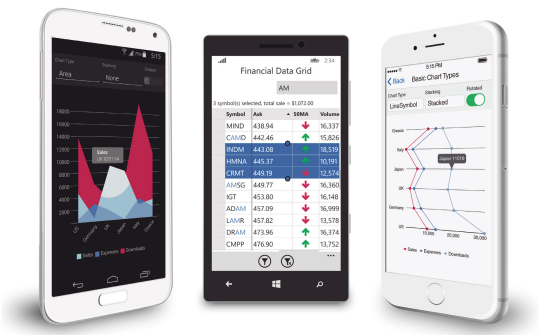


Figure 10: Xuni FlexChart and FlexGrid running on Android, Windows Phone, and IOS

6. Additional Resources

- Mach-O Programming Topics. (n.d.). Retrieved March 28, 2016, from https://developer.apple.com/library/mac/documentation/DeveloperTools/Conceptual/MachOTopics/1-Articles/building_files.html
- Objective Sharpie. (n.d.). Retrieved March 28, 2016, from https://developer.xamarin.com/guides/cross-platform/macios/binding/objective-sharpie/
- Render Base Classes and Native Controls. (n.d.). Retrieved March 28, 2016, from https://developer.xamarin.com/guides/xamarin-forms/custom-renderer/renderers/
- Xamarin.Forms Controls Reference. (n.d.). Retrieved March 28, 2016, from https://developer.xamarin.com/guides/xamarin-forms/controls/

For more information, please contact sales@goxuni.com.

