Xamarin Cross-Platform Development Cookbook

You can create native mobile applications using the Xamarin.Forms platform for the three major platforms: iOS, Android, and Windows Phone. The advantage of this is sharing as much code as possible.

We start with a simple creation of a Xamarin.Forms solution with the three major platforms. Moving on, you will acquire more advanced knowledge and techniques while implementing views and pages for each platform and calling native UI screens such as the native camera page. Further on, you will see the power of architecting a cross-platform solution and inject platform-specific implementations. You will learn how to handle user interactions with the device and take actions on particular events. With all the work done and your application ready, you will master getting the app ready and publishing it in the app store.

What this book will do for you...

- Create and customize your cross-platform UI
- Understand and explore cross-platform patterns and practices
- Use the out-of-the-box services to support third-party libraries
- Find out how to get feedback while your application is used by your users
- Bind collections to ListView and customize its appearance with custom cells
- Create shared data access using a local SQLite database and a REST service
- Test and monitor your applications

Inside the Cookbook...

- A straightforward and easy-to-follow format
- A selection of the most important tasks and problems
- Carefully organized instructions to solve problems efficiently
- Clear explanations of what you did
- Solutions that can be applied to solve real-world problems

Xamarin Cross-Platform Development Cookbook

A recipe-based practical guide to get you up and running with Xamarin cross-platform development

George Taskos

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 3 'Native platform-specific views and pages'
- A synopsis of the book’s content
- More information on Xamarin Cross Platform Development Cookbook
About the Author

George Taskos is a senior software engineer. He has been creating applications professionally since 2005, and he started coding at the age of 8. George has worked as a consultant for the past 10 years in the enterprise and consumer domains.

George is a Microsoft Certified Solutions Developer since 2009 and Xamarin Certified Mobile Developer. Throughout his career, he has created multitier interoperable applications with various technologies, including Windows Forms, WPF, ASP.NET MVC, SOAP, and REST web services, focusing on native iOS/Android and Xamarin Cross Platform Mobile applications for the past 5 years.

As a professional, he worked with small and large enterprises in Greece, Cyprus, South Africa, UK, and USA where he is currently based in New York City.

George is a passionate engineer involved in the start-up community by contributing his free time. He was a member of the BugSense mobile analytics team that was acquired by a NASDAQ big data analytics corporation in 2013.

Currently, he is working with Verisk Analytics, a NASDAQ 100 company, leading the engineering of Xamarin iOS/Android platforms and working in general software architecture of products.
There is no better time for advancing your toolbox with cross-platform mobile development using Xamarin.Forms. Xamarin has over 1,300,000 registered developers and 15,000 clients, it is now the standard in enterprise mobility, and the demand for cross-platform Xamarin mobile developers is very high. The idea behind Xamarin.Forms is that you're no longer only able to share your business logic but the UI code across iOS, Android and Windows Phone. With Xamarin, you can open your favorite IDE, Visual Studio, or use Xamarin Studio, to create cross-platform applications using C# while still generating and deploying a 100% native platform application.

If you know C#, you know how to create native cross-platform, and this book will just make it easier for you.

You will learn everything available to combine your application and build RAD mobile applications. Even if there is a requirement to work on the native application layers, you will find step-by-step recipes that provide you with the knowledge and understanding of how to accomplish your goal.

Work with the UI in XAML or in code, and learn how every control and page is mapped to each equivalent native UI component. Create custom views, call platform APIs, and explore all the cross-platform types of pages, layouts and controls. Create your own cross-platform plugin, and deploy it to NuGet for other developers or commercial purposes.

At the core of the book, the focus is on cross-platform architecture: how to efficiently share code between all platforms, use the built-in dependency service locator, and configure your solution to use a third-party dependency injection using aspect-oriented programming.

Using a cross-platform UI framework doesn't mean that your UI should be coupled to the rest of your code; in this book, the MVVM architecture is demonstrated by injecting the ViewModel in the XAML and using data binding to sync your data between UI controls and models, keeping clean the separation of concerns.
What is programming? Art? Are we the crafts people? No. We process data, that's what we do from the backend to the frontend and vice versa, fetching, transforming, accepting input changes and persisting back to a storage. Data access is the most important layer in an application and there are recipes to cook the repository pattern for local and remote data with many tips, tricks, and best practices for performance and efficiency.

These sequences of data are often presented in a list control. Continuing from this, the book covers the practices needed to create and customize a Xamarin.Forms ListView, adding grouping, jump list, and custom cells.

No modern application is built today without unit testing; in addition, you will find recipes that will help you understand and leverage acceptance UI testing locally and uploading them in Xamarin Test Cloud testing in thousands of physical devices.

Towards the end, we will focus on monitoring an application using Xamarin Insights and preparing and packaging each native platform to upload to the corresponding marketplaces.

What this book covers

Chapter 1, One Ring to Rule Them All, shows you how to work with Xamarin Forms, how to get started with a simple project, understand the structure, and make use of available controls. Starting out the first chapter with creating a cross-platform solution using Xamarin Studio and Visual Studio, create your first page, use Xamarin's out-of-the-box common platform APIs, and learn how to access the native platform page renderers.

Chapter 2, Declare Once, Visualize Everywhere, shows you how to use the Xamarin Forms XAML declarative language to create cross-platform UI. This chapter introduces you to how to create a tabbed application and add UI behaviors and triggers; the main focus is user interface development and providing knowledge in view renderers.

Chapter 3, Native Platform-Specific Views and Behavior, teaches you how to create different page layouts and custom controls per platform. Cross-platform development has all the benefits of sharing code across all three platforms, though there are many scenarios in which you need to access the native platform layer. This chapter will give you all the recipes to create custom platform views, add platform-specific gestures, and access the camera platform APIs.

Chapter 4, Different Cars, Same Engine, shows you how to apply cross-platform architecture, patterns, and practices. Leverage the power of Xamarin.Forms with six sections with all the ins and outs of best practices for sharing code between platform. Use the built-in dependency locator to resolve implementation classes and publish messages to components. Design an MVVM solution, add a third-party DI container for Aspect Oriented Programming and localize your applications for any languages.
Chapter 5, *Dude, Where's my Data?*, shows you how to create a cross-platform data access component fetching data from a local SQLite database and a REST web service. Separation of concerns in a cross-platform solution is critical, learn how to create efficient web and local database repositories. Leverage the native platform performant HTTP SDKs adding and configuring a NuGet package.

Chapter 6, *One for All and All for One*, shows you how to use Xamarin plugins to access native platform capabilities like the camera, GPS, and showing local notifications. The Xamarin community is very generous, there are plugins for Xamarin almost for anything you might want or learn how to do it adding your own implementation, and then share it with giving back or maybe as a commercial library. Learn how to create your own cross-platform plugins and how to use plugins for photos, GPS and local notifications.

Chapter 7, *Bind to the Data*, shows you how to leverage the built-in Xamarin Forms databinding mechanism. Databinding is not a concept that every native platform is providing out of the box, and if you can't actually use data-binding the MVVM architecture doesn't sound such a great idea. Xamarin.Forms provides an out-of-the-box mechanism to bind your data in code or declarative XAML.

Chapter 8, *A List to View*, teaches you how to bind collections to ListView, customize its appearance with custom cells and apply grouping.

Chapter 9, *Gestures and Animations*, shows you how to add cross-platform animations shared between iOS, Android, and Windows Phone and handle user gestures in XAML and in native platform renderers.

Chapter 10, *Test Your Applications, You Must*, shows you how to create unit tests for your portable shared code, and platform-specific unit tests. Create UI acceptance tests and run them locally or in Xamarin Test Cloud. Learn how to use Calabash and REPL.

Chapter 11, *Three, Two, One – Launch and Monitor*, shows you how to add real-time monitoring to get detailed error reports. Prepare and package your applications for submission in iOS, Android, and Windows Phone stores.
Native Platform-Specific Views and Behavior

In this chapter, we will cover the following recipes:

- Showing native pages with renderers
- Attaching platform-specific gestures
- Taking an in-app photo with the native camera page

Introduction

Xamarin.Forms cross-platform UI framework will provide you with all the mechanisms to write your code once and deliver to all three major mobile platforms on the market. You can write your cross-platform pages in the PCL shared project and everything will work.

If you started this book from the beginning recipe by recipe, then you already encountered the usage of PageRenderers for platform-specific functionality and look-and-feel views customization.

In this chapter, we will take the customization of Xamarin.Forms platform-specific pages, views, and behavior to its maximum usage. We start with adding native views for each platform, adding native behavior in views with a delay tap functionality in a view, adding additional views per platform in a page, and in the last recipe, using the native in-app photo capture pages.

In the end, you will have gained all the skills and understanding about how you can customize specific scenarios and utilize the native APIs and features blending with the Xamarin.Forms framework: the best of both worlds!
Native Platform-Specific Views and Behavior

Showing native pages with renderers

While having all this cross-platform user interface behavior and code sharing, sometimes you still need to completely customize a page and blend it with native pages.

In this recipe, we will create an example of loading native platform views for iOS XIB, Android AXML, and Windows Phone UserControl interfaces.

How to do it...

1. Create your cross-platform Xamarin.Forms application using a PCL to share code in Visual Studio or Xamarin Studio; let’s name it XamFormsNativePages. We used Visual Studio in this example, as it’s so easy to have all three projects ready for development!

2. Create a page to use it as our host MainPage. Right-click in the core PCL library and Add | Class…, and name it MainPage.

3. Use the following code. Here, we create a BindableProperty and a ButtonPressed event.

```csharp
public class MainPage : Page
{
    public static readonly BindableProperty RandomNumberProperty =
        BindableProperty.Create("RandomNumber", typeof(int),
        typeof(MainPage), 0);

    public int RandomNumber
    {
        get { return (int)GetValue(RandomNumberProperty); }
        set { SetValue(RandomNumberProperty, value); }
    }

    public event EventHandler ButtonPressed;

    public void OnButtonPressed()
    {
        if (ButtonPressed != null)
        {
            ButtonPressed(this, EventArgs.Empty);
        }
    }
}
```
4. Go to App.cs and replace the code that initializes the MainPage property with instantiating our MainPage class we just created and register a handler to the ButtonPressed event.

```csharp
MainPage = new MainPage();
((MainPage)MainPage).ButtonPressed += MainPageButtonPressed;

private void MainPageButtonPressed(object sender, EventArgs e)
{
    MainPage page = MainPage as MainPage;
    page.RandomNumber = new Random().Next();
}
```

5. In the Android project, go to the Resources/layout folder. In case it doesn't exist, create a new folder, right-click and Add | New Folder. In the layout folder, right-click and Add | New Item…. From the templates, choose Android Layout, name it MainDroidLayout.axml, and hit Add.

6. Now that we have our Android layout, paste the following code; it simply adds a button to the user interface:

```xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
    <Button
        android:id="@+id/button"
        android:gravity="center"
        android:layout_gravity="center_horizontal|center_vertical"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="0" />
</LinearLayout>
```

7. We need to add a renderer where we will make it possible to inflate the native view. Right-click, Add | Class..., name it MainPageRenderer, make the new class a PageRenderer subclass, and add the following code. I know it's a lot, but we will see how it works later.

```csharp
public class MainPageRenderer : PageRenderer
{
    private Android.Widget.Button _button;
```
private Android.Views.View _view;

private MainPage Page
{
    get { return Element as MainPage; }
}

public MainPageRenderer()
{
    Activity activity = (Activity)Forms.Context;
    _view =
        activity.LayoutInflater.Inflate
        (Resource.Layout.MainDroidLayout, this, false);
    _button =
        _view.FindViewById<Android.Widget.Button>
        (Resource.Id.button);
    _button.Click += OnButtonClick;
    AddView(_view);
}

protected override void
OnElementChanged(ElementChangedEventArgs<Page> e)
{
    base.OnElementChanged(e);

    var oldPage = e.OldElement as MainPage;
    if (oldPage != null)
    {
        oldPage.PropertyChanged -= OnPagePropertyChanged;
    }

    var newPage = e.NewElement as MainPage;
    if (newPage != null)
    {
        newPage.PropertyChanged += OnPagePropertyChanged;
    }

    UpdateButtonText();
}

private void OnPagePropertyChanged(object sender,
    PropertyChangedEventArgs e)
{
}
if (e.PropertyName ==
    MainPage.RandomNumberProperty.PropertyName)
{
    UpdateButtonText();
}

protected override void OnMeasure(int widthMeasureSpec,
    int heightMeasureSpec)
{
    base.OnMeasure(widthMeasureSpec, heightMeasureSpec);
    _view.Measure(widthMeasureSpec, heightMeasureSpec);
    SetMeasuredDimension(_view.MeasuredWidth,
        _view.MeasuredHeight);
}

protected override void OnLayout(bool changed, int l, int
t, int r, int b)
{
    base.OnLayout(changed, l, t, r, b);
    _view.Layout(l, t, r, b);
}

private void UpdateButtonText()
{
    if (Page != null)
    {
        _button.Text = Page.RandomNumber.ToString();
    }
}

private void OnButtonClick(object sender, EventArgs e)
{
    if (Page != null)
    {
        Page.OnButtonPressed();
    }
}
8. Don’t forget, with PageRenderer there is always the need to instruct the Xamarin dependency service which implementation to use for each platform. If you miss it, no problem, but the default page will appear and this is not what you want! Add the following ExportRenderer above the namespace declaration:

```csharp
[assembly: ExportRenderer(typeof(MainPage), typeof(MainPageRenderer))]
```

9. There are some using statements of course, you can easily resolve with Ctrl+. in Visual Studio.

10. Let’s jump on the iOS project. Right-click and Add | New Item…. From the templates, choose iPhone View Controller and create UIViewController with a XIB interface file with the name MainPageRenderer. Change the MainPageRenderer UIViewController derive class to PageRenderer; remember, PageRenderer in iOS is UIViewController.

11. Double-click the MainPageRenderer.xib file, which will open the user interface in the Xcode interface builder. At the center of the View, add a UIButton control and link it in the behind .h file as an outlet with the name button. Also, add a TouchUpInside action handler that will be invoked when the button is pressed; name it ButtonPressed. This process is exactly as you would do with the classic Xamarin iOS/Android application or creating a native iOS application.

12. Find next the class implementation of MainPageRenderer:

```csharp
public partial class MainPageRenderer : PageRenderer
{
    private MainPage Page
    {
        get { return Element as MainPage; }
    }

    protected override void OnElementChanged(VisualElementChangedEventArgs e)
    {
        base.OnElementChanged(e);

        var oldPage = e.OldElement as MainPage;
        if (oldPage != null)
        {
            oldPage.PropertyChanged -= OnPagePropertyChanged;
        }
    }
}
```
var newPage = e.NewElement as MainPage;
if (newPage != null)
{
    newPage.PropertyChanged += OnPagePropertyChanged;
}
}

public override void ViewDidLoad()
{
    base.ViewDidLoad();
    UpdateButtonText();
}

private void OnPagePropertyChanged(object sender,
PropertyChangedEventArgs e)
{
    if (e.PropertyName ==
        MainPage.RandomNumberProperty.PropertyName)
    {
        UpdateButtonText();
    }
}

private void UpdateButtonText()
{
    if (IsViewLoaded && Page != null)
    {
        buttonSetTitle(Page.RandomNumber.ToString(),
            UIControlState.Normal);
    }
}

partial void OnButtonPressed(UIButton sender)
{
    if (Page != null)
    {
        Page.OnButtonPressed();
    }
}
13. Again, add the `ExportRenderer` attribute above the namespace.
   ```csharp
   [assembly: ExportRenderer(typeof(MainPage), typeof(MainPageRenderer))]
   ```

14. And for Windows Phone, right-click, **Add** | **New Item**... From the templates, choose **Windows Phone User Control**, give it the name `WindowsPhoneControl.xaml`, and add the following content in the Grid tag to add a `Button`:
   ```xml
   <StackPanel>
      <Button x:Name="button" Content="0"
             HorizontalAlignment="Center" VerticalAlignment="Center"/>
   </StackPanel>
   ```

15. **Add** | **Class**... and name it `MainPageRenderer.cs`. Copy the following code where we instantiate the newly created `UserControl` and add it to the `Children` property of the `Windows Phone ViewGroup`:
   ```csharp
   public class MainPageRenderer : PageRenderer
   {
      private System.Windows.Controls.Button _button;
      private XamFormsNativePages.MainPage Page
      {
         get { return Element as XamFormsNativePages.MainPage; }
      }
      
      protected override void OnElementChanged(ElementChangedEventArgs<Page> e)
      {
         base.OnElementChanged(e);

         var oldPage = e.OldElement as XamFormsNativePages.MainPage;
         if (oldPage != null)
         {
            oldPage.PropertyChanged -= OnPagePropertyChanged;
         }

         var newPage = e.NewElement as XamFormsNativePages.MainPage;
         if (newPage != null)
         {
            newPage.PropertyChanged += OnPagePropertyChanged;
         }
   ```
WindowsPhoneControl ctrl = new WindowsPhoneControl();
    _button = ctrl.button;
    _button.Click += OnButtonClick;
    Children.Add(ctrl);
    UpdateButtonText();
}

private void OnPagePropertyChanged(object sender, PropertyChangedEventArgs e)
{
    if (e.PropertyName == XamFormsNativePages.MainPage.RandomNumberProperty.PropertyName)
    {
        UpdateButtonText();
    }
}

private void UpdateButtonText()
{
    if (Page != null)
    {
        _button.Content = Page.RandomNumber.ToString();
    }
}

private void OnButtonClick(object sender, EventArgs e)
{
    if (Page != null)
    {
        Page.OnButtonPressed();
    }
}

16. No exception for the Windows platform. Add ExportRenderer above the namespace declaration.

[assembly:
    ExportRenderer(typeof(XamFormsNativePages.MainPage),
    typeof(MainPageRenderer))]}
17. Voila! Run the application for each platform and press the button to get random numbers as the button's text.

iOS:
Android:
Xamarin.Forms might not be the perfect framework to create highly UI customizable applications, but it is definitely highly flexible. With the help of renderers, we can mix a cross-platform page with native views.

We started the implementation creating a Page class, MainPage, adding a BindableProperty, RandomNumberProperty, and an event, ButtonPressed, that we can raise when a native button is pressed. Bindable properties are backing stores for properties that allow binding and raising property-changed events when you set a value. Our BindableProperty backing store is RandomNumber of type int.
In the App.cs constructor, we set the MainPage property, the root page of our application, to our MainPage class and we register an event handler for the ButtonPressed event. When this event is raised, we set the backing store of our BindableProperty, RandomNumber, to a random number using the Random class. Simple stuff for our recipe purposes.

In our Android platform, Resources/layout folder, we added an Android UI interface layout, AXML file. If you are familiar with native Android applications, this is a declarative XML-style file that we describe as a user interface. There are only two tags: the LinearLayout main tag that includes a button with the resource name button.

We are ready to create a PageRenderer class, MainPageRenderer, which exposes Activity methods, but remember that the lifecycle events we receive are similar but not fully supporting the Android activity lifecycle events. We customize the native appearance and behavior in the class by declaring two fields: one for the native Android Button, _button, and one for the Android main View, _view, of our AXML UI interface layout, and a helper property to access the cross-platform Page instance.

In the constructor, we grab the native Activity instance using the Forms.Context static property and with this, we inflate our AXML user interface and assign it to our _view property. Having the main View of the layout interface, we use it to grab a reference of the Button instance and registering a handler for its OnClick event. In the end, we use the method AddView(View) to present our custom native UI interface for this Xamarin.Forms page to the client!

We override the OnElementChanged method, where we have the chance to check if there is an OldElement MainPage reference. This gives us the chance to clean any event handlers registered to avoid memory leaks or other resources. In our case, we unregister the PropertyChanged event notification. We then check for a NewElement MainPage reference and assign the handler to the PropertyChanged event.

A Page class inherits from VisualElement, which inherits Element, which in turn inherits BindableObject, which implements the INotifyPropertyChanged interface and gives us the opportunity to get notified for any BindableProperty assignment. In the OnPagePropertyChanged handler, we check if the property event raised from RandomNumberProperty and we update the native Button text property with a method called UpdateButtonText using the Page.RandomNumber property value. The Button. OnClick handler raises in its turn the Page.ButtonPressed event using the method helper, Page.OnButtonPressed, where it fires the series of events again and updates the native Button.Text property.

The preceding logic is pretty much the same for the iOS and Windows Phone platforms. Let's take a quick look at the iOS platform.
Native Platform-Specific Views and Behavior

The native user interface layout in iOS is represented by XIB files or Storyboards. We added in our platform project a XIB file, MainPageRenderer.xib, backed with a UIViewController, MainPageRenderer.cs. Since PageRenderer in iOS is UIViewController, we change the class to a PageRenderer subclass.

In the XIB file, we add a UIButton view with the name button and create an action method for the TouchUpInside notification with the signature OnButtonPressed(UIButton sender), which we implement in our MainPageRenderer class. The rest of the implementation is the same as the Android platform.

In the Windows Platform, we created a UserControl XAML file, WindowsPhoneControl.xaml, added a Button, and in the equivalent MainPageRenderer class, we created an instance of the WindowsPhoneControl class and added it in the ViewGroup.Children collection of the native page.

Happy customization (if needed)!

See also

- Chapter 7, Bind to the Data
- Using custom renderers to change the look and feel of views recipe from Chapter 2, Declare Once, Visualize Everywhere

Attaching platform-specific gestures

Every mobile platform provides us with a way of handling gestures. The approach is not the same for each platform, but the point is the same: accept touch events from the user.

Xamarin.Forms, as of this writing, has cross-platform support for the tap gesture, but worry not, view renderers come to the rescue once again. In this recipe, we will demonstrate how to add long press behavior to BoxView. The equivalent for each platform is iOS CGContext, Android ViewGroup, and Rectangle for the Windows Phone equivalent.
How to do it...

1. Create a Xamarin cross-platform application using Visual Studio. Xamarin Studio works too of course; just add the Windows platform if you need to use Visual Studio later. Let's give it the name XamFormsPlatformGesture.

2. In the PCL project, right-click and Add | New Item..., choose Forms Xaml Page, and name it MainPage.

3. In the App.cs constructor, change the assignment of the MainPage property with a new instance of our newly created MainPage.xaml page.

4. We will need to create our custom control. Since in this example we use the BoxView control, right-click again and Add | Class..., name it CustomBoxView, and make it a subclass of BoxView.

5. Open the MainPage.xaml file and in the ContentPage root tag, add the project namespace so that we can use our new CustomBoxView control.

   xmlns:local="clr-namespace:XamFormsPlatformGesture;assembly=XamFormsPlatformGesture"

6. Now, let's add CustomBoxView control to the page.

   <local:CustomBoxView VerticalOptions="Center" HorizontalOptions="Center" Color="Red" WidthRequest="150" HeightRequest="150"/>

7. In the Android platform, right-click and Add | Class.... We will add a renderer class for CustomBoxView, so give it the name CustomBoxViewRenderer.

8. To catch gestures in Android, we will use SimpleOnGestureListener. So right-click and choose Add | Class...; give it the name CustomBoxViewGestureListener. Make it a subclass of SimpleOnGestureListener and override the OnLongPress method. Check the following implementation:

   public class CustomBoxViewGestureListener : GestureDetector.SimpleOnGestureListener
   {
       public override void OnLongPress(MotionEvent e)
       {
           Console.WriteLine("OnLongPress");
           base.OnLongPress(e);
       }
   }
Native Platform-Specific Views and Behavior

9. CustomBoxViewRenderer has to derive from BoxRenderer. Add the following code to attach the native view to our CustomBoxViewGestureListener:

```csharp
public class CustomBoxViewRenderer : BoxRenderer
{
    private readonly CustomBoxViewGestureListener _listener;
    private readonly GestureDetector _detector;

    public CustomBoxViewRenderer()
    {
        _listener = new CustomBoxViewGestureListener();
        _detector = new GestureDetector(_listener);
    }

    protected override void OnElementChanged(ElementChangedEventArgs<BoxView> e)
    {
        base.OnElementChanged(e);

        if (e.NewElement == null)
        {
            if (this.GenericMotion != null)
            {
                this.GenericMotion -= HandleGenericMotion;
            }
            if (this.Touch != null)
            {
                this.Touch -= HandleTouch;
            }
        }

        if (e.OldElement == null)
        {
            this.GenericMotion += HandleGenericMotion;
            this.Touch += HandleTouch;
        }
    }

    void HandleTouch(object sender, TouchEventArgs e)
    {
        _detector.OnTouchEvent(e.Event);
    }
```
void HandleGenericMotion(object sender, 
GenericMotionEventArgs e) 
{ 
_detector.OnTouchEvent(e.Event); 
} 

10. And of course a common mistake for a renderer is to forget adding the dependency ExportRendererAttribute above the namespace declaration. 

[assembly: ExportRenderer(typeof(CustomBoxView),
typeof(CustomBoxViewRenderer))] 

11. For the iOS platform, add a new class, right-click and Add | Class.... You guessed it right: name it CustomBoxViewRenderer and find the code next to addUILongPressGestureRecognizer to the view;

public class CustomBoxViewRenderer : BoxRenderer 
{ 
UILongPressGestureRecognizer longPressGestureRecognizer;

protected override void 
OnElementChanged(ElementChangedEventArgs<BoxView> e) 
{

base.OnElementChanged(e); 

longPressGestureRecognizer = new 
UILongPressGestureRecognizer(() =>
Debug.WriteLine("Long Press")); 

if (e.NewElement == null) 
{ 
if (longPressGestureRecognizer != null) 
{

this.RemoveGestureRecognizer 
(longPressGestureRecognizer);
}
}

if (e.OldElement == null) 
{ 
this.AddGestureRecognizer 
(longPressGestureRecognizer);
}
} 
}
12. Repeat step 10 for the iOS platform dependency registration of CustomBoxViewRenderer.

13. Add a class to the Windows Phone platform project. Right-click and Add | Class.... This is the Windows renderer, so give it the name CustomBoxViewRenderer. The difference is in the Windows platform is that we derive from BoxViewRenderer, a slight name difference to the Android and iOS platforms. Find the implementation of registering to the following Hold event:

```csharp
public class CustomBoxViewRenderer : BoxViewRenderer
{
    protected override void OnElementChanged(ElementChangedEventArgs<BoxView> e)
    {
        base.OnElementChanged(e);

        if (e.NewElement == null)
        {
            this.Hold -= OnHold;
        }

        if (e.OldElement == null)
        {
            this.Hold += OnHold;
        }
    }

    private void OnHold(object sender, GestureEventArgs e)
    {
        Debug.WriteLine("OnHold");
    }
}
```

14. Run your applications and long press in the rectangle in the center of the screen. You can see a message in the application output window that is the equivalent debug message.

How it works...

As we saw in the preceding example, while Xamarin.Forms doesn’t provide us with all types of gestures, it’s easy to attach listeners and events using platform-specific renderers as we would do with Xamarin iOS and Xamarin Android classic approach in the native application layer.
To start, we need the view that the behavior will be attached. For this recipe, we used the simple BoxView element. To extend it, we create an empty subclass of BoxView, CustomBoxView, and then for each platform, we added the equivalent renderer.

In Android, we added a SimpleOnGestureListener, CustomBoxViewGestureListener, and implemented the OnLongPress method. In the CustomBoxViewRenderer constructor, we create an instance of our listener and then create GestureDetector passing the listener.

In the OnElementChanged method, we check if the e.NewElement is null and unregister the event handler of the GenericMotion and Touch properties of the view. If there is an instance of the e.OldElement property, we only register the event handlers; this is how we make sure we’re not reusing the control.

The handlers are simply sending the e.Event in the GestureDetector.OnTouchEvent method, then the application output prints the message OnLongpress.

iOS is working with gesture recognizers that you can add in a view. There are numerous recognizers that you can use for every case that you want to handle, or you can also implement the TouchesBegan, TouchesEnded, TouchesCancelled, and TouchesMoved methods to own the whole process of touches on the screen.

Here, we simply attach UILongPressGestureRecognizer to our view, passing an action delegate in the constructor to print the message Long press in the output window.

The Windows platform is no exception: CustomBoxViewRenderer registers a handler to the OnHold event of the element and prints the message OnHold.

See also

- Using custom renderers to change the look and feel of views recipe from Chapter 2, Declare Once, Visualize Everywhere
- Adding gesture recognizers in XAML recipe from Chapter 9, Gestures and Animation

Taking an in-app photo with the native camera page

This chapter is all about mix and match cross-platform UI and native platform UI. In the last recipe of the chapter, we will create a Xamarin.Forms solution and utilize the native APIs of each platform to capture a photo.
Native Platform-Specific Views and Behavior

How to do it...

1. As usual, create a Xamarin.Forms project in Visual Studio using a PCL class library for our core shared project. Give it the name XamFormsInAppPhoto.

2. Right-click our core PCL project and Add | Class..., name it InAppCameraPage, and click Add.

3. Make it a ContentPage subclass.

4. Go to the App.cs constructor and replace the code in the MainPage property assignment with a new InAppCameraPage instance.

   ```csharp
   MainPage = new InAppCameraPage();
   ```

That's all we need for our core project setup. We'll move now to the Android platform.

1. To get access in the related camera APIs of Android, we will need a custom PageRenderer for each platform. Right-click the Android project, Add | Class..., name the class InAppCameraPageRenderer, and click Add.

2. Make it a subclass of PageRenderer and implement the TextureView.ISurfaceTextureListener interface.

   ```csharp
   public class InAppCameraPageRenderer : PageRenderer, TextureView.ISurfaceTextureListener
   ```

3. Let's add ExportRender attribute on top of the namespace declaration now to instruct DependencyService in Xamarin.Forms that we want to use this PageRenderer for our InAppCameraPage. It's a vital piece. If you find yourself in a situation that your custom PageRenderer has not loaded, the first thing to check is if you're missing ExportRender attribute.

   ```csharp
   [assembly: ExportRenderer(typeof(InAppCameraPage),
   typeof(InAppCameraPageRenderer))]
   ```

4. To load our camera live feed, we will need an Android layout file. Right-click in the Resources/layout folder and Add | New Item..., choose Android Layout, name it InAppPhotoLayout.axml, and click Add. If there is no layout folder in the Resources folder, create one by right-clicking in Resources and Add | New Folder.

5. In the newly created camera feed layout, add the following code:

   ```xml
   <?xml version="1.0" encoding="utf-8"?>
   <FrameLayout
   xmlns:android="http://schemas.android.com/apk/res/android"
   android:orientation="vertical"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:layout_weight="1">
   <TextureView
   ```
6. Go to the `InAppCameraPageRenderer.cs` class file again and add the following private fields:

```csharp
    Android.Hardware.Camera camera;
    Android.Widget.Button takePhotoButton;
    Activity activity;
    TextureView textureView;
    Android.Views.View view;
    ImageView snapshotImageView;
```

7. Override the `OnElementChanged` method.

```csharp
protected override void
    OnElementChanged(ElementChangedEventArgs<Page> e)
{
    base.OnElementChanged(e);

    if (e.OldElement != null || Element == null)
        return;

    activity = this.Context as Activity;
    view =
        activity.LayoutInflater.Inflate
            (Resource.Layout.InAppPhotoLayout, this, false);

    textureView =
        view.FindViewById<TextureView>
            (Resource.Id.textureView);
```
textureView.SurfaceTextureListener = this;

takePhotoButton = 
view.FindViewById<Android.Widget.Button>
(Resource.Id.takePhotoButton);
takePhotoButton.Click += OnTakePhoto;

snapshotImageView = view.FindViewById<ImageView>
(Resource.Id.snapshotView);

AddView(view);
}

You might have already noticed a warning regarding the Android.Hardware.Camera API. This API is marked as obsolete and is deprecated as of Android 5.0. However, it is fine to use the API until your minSdkVersion is 21 or higher where you will need to use the replacement Android.Hardware.Camera2.

1. Add the OnTakePhoto method to capture the photo and set it to ImageView.
   private void OnTakePhoto(object sender, EventArgs e)
   {
       camera.StopPreview();
       snapshotImageView.SetImageBitmap (textureView.Bitmap);
       camera.StartPreview();
   }

2. Now, let's implement the TextureView.ISurfaceTextureListener methods.
   public void OnSurfaceTextureAvailable(SurfaceTexture surface, int width, int height)
   {
       camera = 
       Android.Hardware.Camera.Open((int)CameraFacing.Back);
       textureView.LayoutParameters =
       new FrameLayout.LayoutParams(width, height);
       camera.SetPreviewTexture(surface);
       PrepareAndStartCamera();
   }

   public bool OnSurfaceTextureDestroyed(SurfaceTexture surface)
   {
       camera.StopPreview();
       camera.Release();
   }
return true;
}

public void OnSurfaceTextureSizeChanged(SurfaceTexture surface, int width, int height)
{
  PrepareAndStartCamera();
}

public void OnSurfaceTextureUpdated(SurfaceTexture surface)
{
  // Nothing
}

3. Add the PrepareAndStartCamera method used just now. We need this configuration to overcome a bug with certain hardware. For details, you can refer to https://code.google.com/p/android/issues/detail?id=1193.

private void PrepareAndStartCamera()
{
  camera.StopPreview();

  var display = activity.WindowManager.DefaultDisplay;
  if (display.Rotation == SurfaceOrientation.Rotation0)
  {
    camera.SetDisplayOrientation(90);
  }

  if (display.Rotation == SurfaceOrientation.Rotation270)
  {
    camera.SetDisplayOrientation(180);
  }

  camera.StartPreview();
}

4. To make our native layout visible, we need to override the OnLayout method of PageRenderer and set the size and position.

protected override void OnLayout(bool changed, int l, int t, int r, int b)
{
  base.OnLayout(changed, l, t, r, b);
}
var msw = MeasureSpec.MakeMeasureSpec(r - l, MeasureSpecMode.Exactly);
var msh = MeasureSpec.MakeMeasureSpec(b - t, MeasureSpecMode.Exactly);

view.Measure(msw, msh);
view.Layout(0, 0, r - l, b - t);

5. Ready! Now you can test the project on a device or an emulator that supports a camera. The next screenshot is taken with my Android tablet device I use to test hardware APIs. It is always better to test hardware capabilities and APIs on a device to make sure everything works as we scheduled.

It's Apple's turn. Focus on the XamFormsInAppPhoto.iOS project. We will use Xamarin Studio to accomplish the next steps since we will use Xcode Interface Builder to create our native UI.

1. Right-click the iOS platform project and Add | New File.... In the iOS tab, select iPhone View Controller, name it InAppCameraPage, and click Add.
2. Derive from `PageRenderer` and you can also remove the constructor overload safely, since there is no such overload for our `PageRenderer` base class.

```csharp
public partial class InAppCameraPage : PageRenderer
```

3. Add `ExportRenderer` attribute to make our custom `PageRenderer` load in runtime.

```csharp
[assembly: ExportRenderer(typeof(InAppCameraPage), typeof(XamFormsInAppPhoto.iOS.InAppCameraPage))]
```

4. For this step, we will polish our native UI skills and open `XamFormsInCameraPage.xib` created with our `UIViewController` in Xcode. Double-click the file.

5. Now, in the `UIView` of Interface Builder, drop two `UIViews` inside. Be careful, because both of the `UIViews` must be children of the main `UIView` and siblings to each other; this is important in creating an overlay view to make other views visible on top of the camera live stream. Check the following screenshot of our example and also refer to the code of this book for more details. We added constraints to make the layout appear appropriately in all different-sized devices.
6. In addition to `UIViews`, we add a `UIImageView` to preview our captured photo and a `UIButton` to capture one.

7. Next, let's add the outlets needed in the code and an action for the `UIButton`; the action will be invoked when the event `TouchUpInside` is raised.

8. If you are familiar with Xcode Interface Builder, holding down the control button on a view, dragging next to the assistant editor window with the left mouse button pressed, and releasing the mouse button creates an outlet if we are editing the header file (.h), or an action in the implementation file (.m). See next the outlets we created in our header file and the action added in the implementation file for our example.

Outlets:

```c
@interface InAppCameraPage : UIViewController

UITapGestureRecognizer *tapGestureRecognizer = nil;

- (IBAction)takePhoto:(id)sender;
@end
```
9. Returning to Xamarin Studio, it will automatically update the Xcode changes made. Now our outlets are available as properties and the action is a partial method we need to implement in our custom PageRenderer, InAppCameraPage.

10. Open InAppCameraPage.cs and add the following private fields. These are the classes we need to work with the iOS camera API.

```csharp
AVCaptureSession captureSession;
AVCaptureDeviceInput captureDeviceInput;
AVCaptureStillImageOutput stillImageOutput;
```

11. Add the following method, AuthorizeCameraUseAsync. iOS requires that you get the consent of the user to get access to some APIs; camera is one of them. Let's also put some code in ViewDidLoad to ask the user for permission and continue setting up the camera live feed.

```csharp
public override void ViewDidLoad ()
{
    base.ViewDidLoad ();

    AuthorizeCameraUseAsync ().ContinueWith ((antecedent) =>
    {
        bool result = antecedent.Result;
        if (result)
        {
```
Native Platform-Specific Views and Behavior

```csharp
public async Task<bool> AuthorizeCameraUseAsync()
{
    var authorizationStatus =
        AVCaptureDevice.GetAuthorizationStatus
        (AVMediaType.Video);
    if (authorizationStatus !=
        AVAuthorizationStatus.Authorized)
    {
        return await
            AVCaptureDevice.RequestAccessForMediaTypeAsync
            (AVMediaType.Video);
    }
    else if (authorizationStatus ==
        AVAuthorizationStatus.Authorized)
    {
        return true;
    }
    return false;
}

12. Add the SetupCameraLiveFeed method and a helper
    ConfigureCameraForDevice method.
    public void SetupCameraLiveFeed()
    {
        captureSession = new AVCaptureSession();
        AVCaptureVideoPreviewLayer videoPreviewLayer =
            new AVCaptureVideoPreviewLayer(captureSession)
        {
            Frame = cameraViewContainer.Bounds
        };
        cameraViewContainer.Layer.AddSublayer(videoPreviewLayer);
        AVCaptureDevice captureDevice =
            AVCaptureDevice.DefaultDeviceWithMediaType
            (AVMediaType.Video);
        ConfigureCameraForDevice(captureDevice);
        captureDeviceInput =
            AVCaptureDeviceInput.FromDevice(captureDevice);
```
NSMutableDictionary dictionary =
new NSMutableDictionary();
dictionary[AVVideo.CodecKey] =
new NSNumber((int)AVVideoCodec.JPEG);
stillImageOutput = new AVCaptureStillImageOutput()
{
    OutputSettings = new NSDictionary()
};
captureSession.AddOutput(stillImageOutput);
captureSession.AddInput(captureDeviceInput);
captureSession.StartRunning();
}

public void ConfigureCameraForDevice(AVCaptureDevice
    device)
{
    NSError error = new NSError();
    if (device.IsFocusModeSupported
        (AVCaptureFocusMode.ContinuousAutoFocus))
    {
        device.LockForConfiguration(out error);
        device.FocusMode =
            AVCaptureFocusMode.ContinuousAutoFocus;
        device.UnlockForConfiguration();
    }
    else if (device.IsExposureModeSupported
        (AVCaptureExposureMode.ContinuousAutoExposure))
    {
        device.LockForConfiguration(out error);
        device.ExposureMode =
            AVCaptureExposureMode.ContinuousAutoExposure;
        device.UnlockForConfiguration();
    }
    else if (device.IsWhiteBalanceModeSupported
        (AVCaptureWhiteBalanceMode.ContinuousAutoWhiteBalance))
    {
        device.LockForConfiguration(out error);
        device.WhiteBalanceMode =
            AVCaptureWhiteBalanceMode.ContinuousAutoWhiteBalance;
        device.UnlockForConfiguration();
    }
}
13. Implement the `CameraPhotoTouchUpInside` method and add an async method to capture the photo.

```csharp
partial void CameraPhotoTouchUpInside(UIKit.UIButton sender)
{
    CapturePhotoAsync();
}

public async Task CapturePhotoAsync()
{
    var videoConnection = 
        stillImageOutput.ConnectionFromMediaType
            (AVMediaType.Video);
    var sampleBuffer =
        await stillImageOutput.CaptureStillImageTaskAsync
            (videoConnection);
    var jpegImageAsNsData = 
        AVCaptureStillImageOutput.JpegStillToNSData
            (sampleBuffer);
    var image = new UIImage (jpegImageAsNsData);
}
```

14. Let's clean up resources by overriding the `Dispose` method.

```csharp
protected override void Dispose(bool disposing)
{
    captureSession.Dispose();
    captureDeviceInput.Dispose();
    stillImageOutput.Dispose();
    base.Dispose(disposing);
}
```

15. The iOS platform is ready. See next a screenshot from my iPhone 6 device:
Last but not least. Microsoft Windows Phone offers, of course, APIs to work directly with the camera, or use one of the chooser tasks. In our example, we will use the CameraCaptureTask API to capture a photo and assign it to an image control of our native UserControl.

1. Back to Visual Studio, right-click in XamFormsInAppPhoto.WinPhone, Add | New Item..., select Windows Phone User Control, give it the name PreviewImageUserControl, and click Add.

2. Double-click the PreviewImageUserControl.xaml file and add an Image view inside the Grid layout view.

```xml
<Image x:Name="previewImage"
      Width="Auto"
      Height="Auto" />
```
Native Platform-Specific Views and Behavior

3. Right-click the project again and Add | Class..., name it InAppCameraPageRenderer, and click Add. Make it a PageRenderer subclass.


   [assembly: ExportRenderer(typeof(InAppCameraPage), typeof(InAppCameraPageRenderer))]

5. Add two private member fields to have a reference of our native PreviewImageUserControl and CameraCaptureTask.

   CameraCaptureTask cameraCaptureTask;
   PreviewImageUserControl previewImageUserControl;

6. Override the OnElementChanged method and add the following code. We simply instantiate the CameraCaptureTask field, the custom PreviewImageUserControl, and add it in the Children collection of the view. At the end, we Show the CameraCaptureTask.

   protected override void OnElementChanged(ElementChangedEventArgs<Page> e)
   {
       base.OnElementChanged(e);

       if (e.OldElement != null || Element == null)
           return;

       cameraCaptureTask = new CameraCaptureTask();
       cameraCaptureTask.Completed +=
           OnCameraCaptureTaskCompleted;

       previewImageUserControl =
           new PreviewImageUserControl();
       Children.Add(previewImageUserControl);

       cameraCaptureTask.Show();
   }

7. Add the OnCameraCaptureTaskCompleted event handler.

   private void OnCameraCaptureTaskCompleted(object sender, PhotoResult e)
   {
       if (e.TaskResult == TaskResult.OK)
BitmapImage bmp = new BitmapImage();
bmp.SetSource(e.ChosenPhoto);
previewImageUserControl.previewImage.Source = bmp;
}
}

8. You can of course use a device to test the Windows Phone platform, but the emulator supports a camera. For this example, we used the Windows Phone 8.1 emulator. When we start the project, CameraCaptureTask immediately shows the following:
9. Click the camera button at the bottom. It will close the task chooser and return to our page showing the image captured!
For the Android and iOS platforms, we used native APIs to access the stream of the camera.

In Android, we used the `Android.Hardware.Camera` API and not the new `Android.Hardware.Camera2` because we want to have backward compatibility lower to Android SDK 21. We created a layout with a root `FrameLayout`, a `TextureView` to display our camera's stream, an `ImageView` to preview a snapshot captured, and a `Button` to capture a snapshot. We implemented `TextureView.ISurfaceTextureView` to get notified when the surface texture associated with this `TextureView` is available.

In the `OnElementChanged` method, we get our `TextureView` instance and set `SurfaceTextureListener` to our `InAppCameraPageRenderer` instance. When is available the `OnSurfaceTextureAvailable` implemented method is invoked and we open the camera, set the `LayoutParams` of the `TextureView`, set the camera's preview texture with the provided surface, and in the end we prepare and start the camera. In `OnSurfaceTextureDestroyed`, we stop the stream preview and release it from memory, and in `OnSurfaceTextureSizeChanged`, we again call `PrepareAndStartCamera` to make sure that the stream appears correctly to orientation and size changes.

For `takePhotoButton`, we subscribed a delegate handler to the `Click` event where we stop the stream, assign `TextureView.Bitmap` to `snapshotImageView`, and start the stream again.

Since we load our native Android XML layout in our `InAppCameraPageRenderer`, we override the `OnLayout` method to provide the inflated root View with the appropriate layout size.

iOS native APIs live in the AVFoundation framework kit. In the `ViewDidLoad` method, we check and request for authorization if needed and then set up the live feed if it is allowed using `AVCaptureSession` passed to `AVCaptureVideoPreviewLayer`. Adding it to our `cameraViewContainer.Layer` as a sublayer, configure the default video media `AVCaptureDevice` and get `AVCaptureDeviceInput` from `AVCaptureDevice`. We then create an `AVCaptureStillImageOutput` instance and pass it to `AVCaptureSession.AddOutput`. Pass `AVCaptureDeviceInput` to `AVCaptureSession.AddInput` and invoke the `AVCaptureSession.StartRunning()` method.

Implementing the partial `CapturePhotoTouchUpInside` method, we use `AVCaptureStillImageOutput` to get `AVCaptureConnection`, passing it to the `CaptureStillImageTaskAsync` method and then translate the returned `CMSampleBuffer` to an `NSData` instance with the `AVCaptureStillImageOutput.JpegStillToNSData` static method. Then we just create a `UIImage` passing the `NSData` instance and set to the `captureImageView.Image` property.
We also override the Dispose method to free some memory upon release of the InAppCameraPageRenderer instance.

Using the iOS camera live stream is much more complicated than the Android example and it is strongly recommended to review the native APIs for a greater understanding and control of the AVFoundation framework.

The Windows Phone CameraCaptureTask is straightforward: creating an instance, registering to the completed event, and showing the task. When we capture a photo, the completed event is invoked and we set the photo to our loaded UserControl image property.

**See also**

- [https://developer.xamarin.com/recipes/android/other_ux/textureview/display_a_stream_from_the_camera/](https://developer.xamarin.com/recipes/android/other_ux/textureview/display_a_stream_from_the_camera/)
Where to buy this book

You can buy Xamarin Cross Platform Development Cookbook from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.