Xamarin.Forms Solutions

—

Gerald Versluis

Steven Thewissen

Foreword by David Ortinau

Apress®

Xamarin.Forms Solutions

Gerald Versluis

Steven Thewissen

Foreword by David Ortinau

Apress®

Xamarin.Forms Solutions

Gerald Versluis
Hulsberg, The Netherlands

Steven Thewissen
Hulsberg, Limburg, The Netherlands

ISBN-13 (pbk): 978-1-4842-4133-2
ISBN-13 (electronic): 978-1-4842-4134-9
https://doi.org/10.1007/978-1-4842-4134-9

Library of Congress Control Number: 2018964067

Copyright © 2019 by Gerald Versluis and Steven Thewissen

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Jonathan Gennick
Development Editor: Laura Berendson
Coordinating Editor: Jill Balzano

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit http://www.apress.com/rights-permissions.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484241332. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

Table of Contents

About the Authors ... ix

About the Technical Reviewer ... xi

Acknowledgments ... xiii

Foreword by David Ortinau ... xv

Introduction ... xvii

Chapter 1: Fundamentals ... 1

- Custom Renderers ... 1
  - Built-in Renderers ... 1
  - Implementing a Custom Renderer ... 3
- Effects ... 10
  - Effects Versus Custom Renderers ... 11
  - Implementing an Effect ... 11
- MessagingCenter ... 18
  - How MessagingCenter Works ... 19
  - Using MessagingCenter ... 19
- DependencyService ... 21
  - Using the DependencyService ... 22
  - DependencyFetchTarget ... 25
- Behaviors ... 25
  - Xamarin.Forms Behaviors ... 26
  - Attached Behaviors ... 29
- Summary ... 33

Chapter 2: User Interface..35
Exploring the Layout Options 35
Using the Right Layout Controls 36
Determining the Right Alignment and Expansion 47
Margin and Padding 49
Introducing the FlexLayout 50
Changing Our UI Based on Data 66
Property Triggers 66
Data Triggers 67
Event Triggers 68
Multi Triggers 69
Using Triggers with Style 70
Using EnterActions and ExitActions...71
Working with Animations 72
Adding Simple Animations..73
Combining Animations..81
Working with Easing...83
Working with Custom Animations...84
Using Custom Fonts 88
Adding a Custom Font in iOS 88
Adding a Custom Font in Android 89
Finding the PostScript Name of the Font...90
Tying It All Together in Shared Code 92
Creating Reusable Controls...94
Introducing a Simple Use Case...95
Creating a Custom Control..96
Creating a Bindable Property..96
Using a Custom Control 101

Exploring Gestures 103
Adding a TapGestureRecognizer 103
Adding a PinchGestureRecognizer 104
Adding a PanGestureRecognizer 105
Summary 107
Chapter 3: Working with Data 109
Getting Data Onto the Screen 109
Data Binding Basics 110
Notifying the App of Data Changes 115
Converting Data Bound Values 120
Getting to Know the ListView 123
Understanding the Basics 125
Adding a Pull-to-Refresh 136
Using a Context Action to Act on a List Item 138
Adding a Jump List for Easy Navigation 140
Using Data Templates 144
Creating a Data Template 145
Dynamically Selecting a Data Template 148
Caching Data Using the ListView 150
Summary 151
Chapter 4: Network and Security 153
Connecting with REST APIs 153
The Old-Fashioned Way 153
HTTP Requests with Refit 155
Other Libraries 158
Connectivity 158
Caching 163
Akavache 163
Monkey Cache 169

Network Resilience 171
Retry Policy 173
Timeout 175
PolicyWrap 176
Conditional HTTP Requests 178
Sensitive Data 185
Storing Sensitive Data 185
Transferring Data Securely 186
Tying It All Together 188
Summary 189
Chapter 5: Business App Concepts 191
Showing a PDF File 191
Shared Project 192
iOS Project 193
Android Project 195
Showing the PDF File 197
Scanning Barcodes 200
Installing ZXing 201
Implementing Barcode Scanning 204
Implementing Generating Barcodes 208
App Center Integrations 209
What Is App Center? 209
Creating Your App in App Center 210
Building, Testing, and Distributing 211
Crash Reporting 212
Analytics 214
App Versioning 217
iOS 218
Android 219
Setting the Version Number with Azure Pipelines 220

Localization 222
Localize Using Resource Files 223
Localize Using i18n Portable 238
Summary 241
Chapter 6: Advanced Topics 243
Converting a PCL to .NET Standard 244
Why Should We Upgrade? 244
Upgrading to .NET Standard 245
Reducing the Size of Our App 246
What Does Xamarin Already Do for Us? 246
Fixing Errors Caused by the Linker 248
Using an Icon Font Instead of Images 250
Replacing Images with an Icon Font 251
Additional File Size Improvements 252
Signing and Deploying Our Code 253
Automatic Provisioning on iOS 253
Signing and Publishing on iOS 256
Signing and Publishing on Android 262
Contributing to Xamarin.Forms 271
The Human Side 271
How to Get Started 272
Summary 274
Index 275

About the Authors

Gerald Versluis is a developer and Microsoft MVP from the Netherlands with years of experience working with Xamarin, Azure, ASP.NET, and other .NET technologies. He has been involved in numerous projects, in various roles. A great number of his projects are Xamarin apps. Not only does Gerald like to code, but he is keen on spreading his knowledge as well as gaining some in the bargain. He speaks, provides training sessions, and writes blogs and articles in his spare time.

Steven Thewissen is a developer and Microsoft MVP from the Netherlands focusing on Xamarin, Azure, and other .NET technologies. He started working with Xamarin in 2014, and has been in love with it ever since. Steven shares his knowledge by regularly writing blogs about topics that interest him. He loves to push the boundaries of what's graphically possible with Xamarin.Forms due to his interest in UI design. He loves to create kick-ass user interfaces to accompany his mobile apps.

About the Technical Reviewer

Pieter Nijs is a Belgian .NET architect with a passion for mobile and cloud development. He has played a key role in several projects, ranging from large consumer-facing telecom and media apps to smaller LOB applications. As he is primarily interested in the Microsoft stack. His interest and expertise translate to technologies like .NET, C#, Xaml, Xamarin, UWP, Azure, Visual Studio, TFS, VSTS, and more. Both at work as well as in his spare time, Pieter is constantly working and playing with these and other new technologies. He likes to tell everybody about the things he does, sharing his knowledge. You can find him speaking at conferences, giving trainings, and blogging at blog.pieeatingninjas.be. In 2017, Pieter received a Microsoft MVP Windows Development Award for sharing his passion and expertise with the community.

Acknowledgments

We would like to take some time to thank all of the people who made it possible for us to write this book.

First and foremost, our loving spouses Nadia and Laurie. Without them keeping the little munchkins out of our hair so we had the time to write, we would not be where we are today. They were just as important in completing this book as we were.

Also, our strict but very fair technical reviewer, Pieter. We know you have put in a lot of time to check and recheck every word, even in the code blocks. What really stood out to us is that you did not just point out mistakes and errors, but you also gave positive notes on the parts you liked. Thank you for taking the effort to make this book better.

A proper book cannot do without a wonderful foreword. For that, we were honored to have none other than David Ortinau write that for us. We are thankful that you could find time for us in your busy schedule to read our drafts. It really is the icing on the cake to have your foreword in this book.

From Apress, we would like to thank the entire publishing team, but Jill and Jonathan in particular. They have been there to guide us through this entire process and have been looking out for us along the way.

Foreword by David Ortinau

I love working with teams to build mobile apps, and to help make the whole process better. Working with Microsoft on Xamarin and mobile developer tools is the perfect fit for me. The work keeps me up at night laboring to solve some nagging issue blocking a customer, and it gets me up early in the morning to push forward. Day in and day out, that work ends up being a lot of screen time!

A few years back, I started venturing into the great outdoors where I could for a brief time leave all my work behind. I recall my first time hiking on a single-track trail only 1.5 miles from the nearest road, house, or human; a wave of panic came over me. I looked at my phone and prayed to see bars. Phew, I was okay. But what if I sprained my ankle? What if I ran out of water? What if I saw a bobcat? (I live in Missouri —that's about as dangerous as it gets at the foothills of the Ozark Mountains.)

I realized if I was going to spend more time in the woods and get farther and farther from civilization, I needed to know what to do in emergency situations and how to sustain myself. As my goals went from two miles to 10 to eventually 50 miles, the planning and the knowledge I needed to be successful grew and grew.

The first book I picked up was a survival handbook with practical advice such as how to escape a mountain lion, and not so practical advice such as how to survive jumping from a building into a dumpster. I talked to others whom I met out on the trails, I read a lot of blogs, and I watched a lot of YouTube. I learned valuable lessons all along the way.

A while later, I did see a coyote and it was enormous! I quickly texted my hiking friend who had hiked part of the Appalachian Trail the previous year about what to do, and he replied: "Back away slowly, makes some noise so he knows you're there, and just keep moving. Make a lot of noise if comes toward you." Phew! Good advice, and the coyote went bounding away.

Perhaps for you the questions are more like: "How do I make my Entry field look and work like this design?" or "How do I do barcode scanning in Xamarin.Forms?" or any number of other common questions. As I look at his book, I see a great success guide for mobile developers. It will answer many of your questions and help you like those survival guides have helped me.

Steven is a long-time Xamarin developer and community advocate known for blogging about how to make beautiful apps with Xamarin.Forms. Gerald is a recognized Microsoft MVP and an active contributor to Xamarin.Forms. Steven and Gerald should be on the top of anyone's list to call when that "coyote" surprises you in the middle of your day.

If you heed the guidance in this book, you'll not only survive your project, but you'll thrive! As you master the fundamental lessons, the book becomes a great resource to grab code samples from that help solve the common challenges along your way to building and shipping better and bigger apps more quickly.

—David Ortinau

Introduction

First of all, we would like to thank you for acquiring this book. Hopefully you will enjoy reading it as much as we enjoyed writing it.

This book is all about Xamarin.Forms. We have tried to provide you with a very complete reference on this matter. Whether you are a starting developer or a more seasoned one, this book should hold useful information for anyone.

We will start you off by going over the basic fundamentals of Xamarin.Forms. You will learn about renderers, which are at the heart of Forms. These renderers perform the translation between abstract controls and the native controls. You also have the possibility to hook into that and write so-called custom renderers. You will also learn how to leverage platform-specific code in a shared code context. This is accomplished with the `DependencyService` that is built-in right into Xamarin.Forms. This and much more is just in the first chapter.

From there, we will target the user interface. Contrary to popular belief, you can create beautiful UIs in Xamarin.Forms. We will provide you with all the information and tools to do that for yourself. We will see what options there are to create layouts and how the elements work in-depth. Of course, data is something you cannot do without, so we will show you how you can influence the way your interface looks based on that data. Other things that you will learn are working with animations and custom fonts and how to create reusable controls that you can not only share within your project, but even across projects!

In the remaining chapters you will find all kinds of solutions for various scenarios. All of them conveniently categorized into a couple of chapters. As already mentioned: we can't do without data. Therefore, a good understanding of how to get that data into our app and update it through data-binding is crucial. We learn how to show data, how to transform data through value converters, and how to show repeatable data in the `ListView` control.

To retrieve this data, a network connection is very important. Because of that, we have also included a whole chapter full of solutions that cover network and security. It teaches you how to connect to a backend and how to save code while doing so. We learn how to check our connectivity, cache data so we don't impose unnecessary costs on our users, save data in a secure place, and deal with an unstable network.

Toward the end of this book you will encounter more common real-life scenarios. Think about showing a PDF file and scanning or generating barcodes but also how to version your app and collect crash reports and analytics from your app when it is released. To top it off, there are some advanced topics. Xamarin.Forms app can be a bit bulkier in file size than true native apps. You will learn how to shave off some of that extra size. Also, with the transition from PCL libraries to .NET Standard, you might want to look into converting your shared code in that area. We will teach you how to do just that and much, much more.

It would be impossible to cover everything about everything, but with this book in hand, we hope to provide you with a solid base. Based on our years of hands-on experience, we have tried to identify the most frequently asked questions and capture the solutions in this book. Hopefully we will be able to transfer all this knowledge to you through the means of this book and enable you to do amazing things and build great apps.

CHAPTER 1

Fundamentals

In this chapter, we look at some Xamarin.Forms fundamentals. The concepts explained here will be used throughout this book, so it is important that you have an understanding of how these fundamentals work. If you already have some knowledge of Forms and more specifically, custom renderers, the DependencyService, the MessagingCenter, behaviors and effects, then it is probably safe to skip this chapter. If not, please keep reading.

Custom Renderers

At the heart of Xamarin.Forms are the renderers. Xamarin.Forms is a library that allows you to define the user interface (UI) as an abstract layer as opposed to implementing a UI per platform. To make this work, Xamarin has provided a set of so-called renderers that will take these abstract controls and translate them into the native counterparts for you.

Built-in Renderers

For example, if you define a (Xamarin.Forms) `Button` at runtime, this will be translated to a `UIButton` and `Android.Widget.Button` for iOS and Android, respectively. All the properties you have set on the Forms button will be mapped by the renderer to the native equivalent of that property. Or, if a similar property is not available, code is injected to simulate the intended behavior.

A schematic overview of this process is shown in Figure 1-1.

© Gerald Versluis and Steven Thewissen 2019
G. Versluis and S. Thewissen, *Xamarin.Forms Solutions*, https://doi.org/10.1007/978-1-4842-4134-9_1

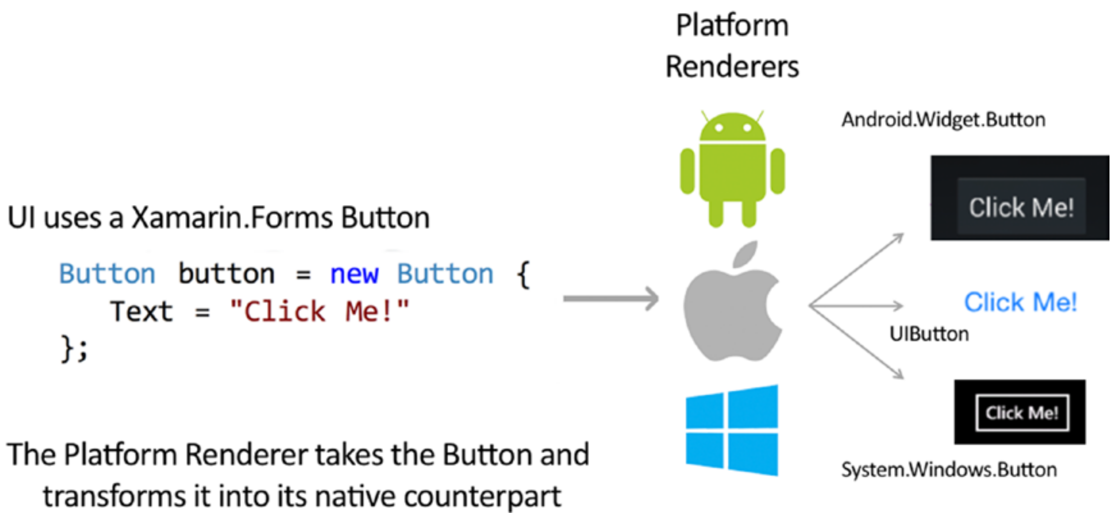

Figure 1-1. *Schematic overview of a renderer in Xamarin.Forms*

You might be wondering why a custom renderer would be needed at all. The team at Xamarin has mostly focused on implementing the most basic properties of each control. More specifically the properties that are supported by all the platforms that were supported by Xamarin.Forms at that time. But the platform-specific controls usually have more properties and capabilities that you might want to use but aren't supported by Xamarin.Forms right now. To make the platform-specific properties available to you, you can write your own renderer, typically referred to as a *custom renderer*.

So, if the default renderer, implemented by Xamarin, doesn't cut it for you, you can always choose to implement your own. I wouldn't recommend writing it from scratch, but if that is what you want, it is possible. Creating a full custom renderer on your own can be done by inheriting from the `ViewRenderer`. When you do implement your own renderer from scratch, you are responsible for creating a platform control from start to finish. That means translate the abstract Xamarin. Forms control into the control that belongs on the targeted platform. This includes setting all the necessary properties.

Instead, typically you would inherit the default renderer and just tweak the bits that are relevant to you. That way, you can normally use 99% of the code that the Xamarin team wrote for you and you just have to set that one property that you would want to change. To explain the basics, let's look at an example in the next section.

Implementing a Custom Renderer

Let's say that we want to customize a Xamarin.Forms `Entry`. In Figure 1-2, you can see how the entry control is rendered for each specific platform if we are creating an app for iOS, Android, and Windows.

At the top level, we define an `Entry` either in XAML or code. Whenever this code is executed, through reflection Xamarin.Forms will go through the associated renderer for this control and instantiate a new control. The newly created control is the platform-specific equivalent of the `Entry`.

In the case of the entry control, the renderer is named `EntryRenderer`. This is the typical naming convention: <*control name*>`Renderer`. There are some exceptions to this. For a full list of renderers and when to use which, refer to the documentation page: `https://docs.microsoft.com/en-us/xamarin/xamarin-forms/app-fundamentals/custom-renderer/renderers`.

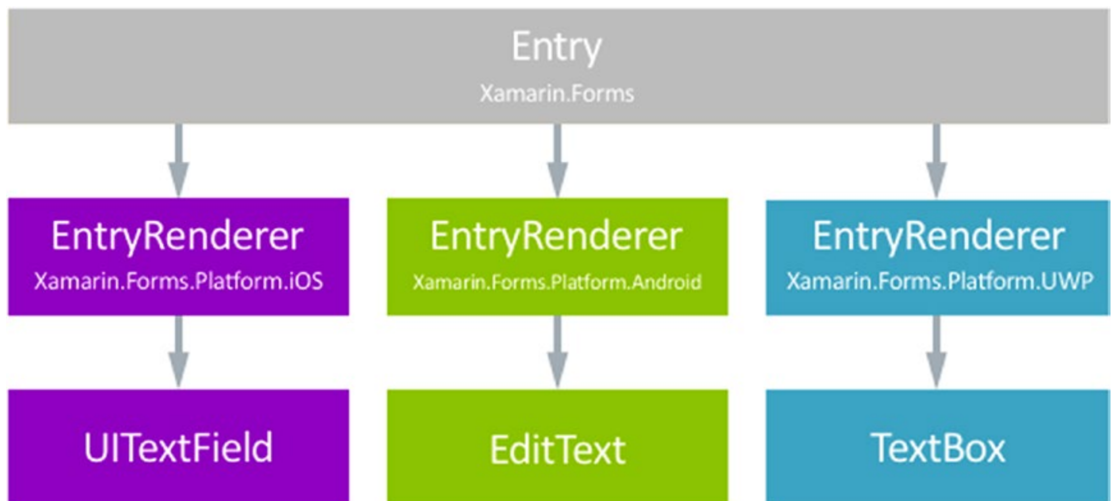

Figure 1-2. *Default path for rendering an Entry*

Imagine that we want to change something in the default behavior of the Forms renderer. For instance, we want to change the background color for this control. Note that this is possible with Forms out of the box, but for this example we will do it with our own renderer.

Creating a custom renderer takes roughly three steps:

1. Create a custom control—This needs to be created in the shared Xamarin.Forms project. Typically you would want to create an inheritance of the control that you want to create a renderer for. We will see why in a minute.

2. Use the control in your app—This also happens in the shared Forms project. You can use your control from either XAML or code, use it anywhere in your screens and layout.

3. Create a custom renderer for your control—Finally, you have to implement the custom renderer in each platform that you want to add something custom. Implementing it for each platform is not absolutely necessary. We will see this a little later.

The first step isn't absolutely necessary, but I would recommend it. If you do not create a custom control, the renderer will be invoked for *all* the default controls. That could be something that you would want, but typically it isn't. Imagine you want to add color your `Entry` control's border. By implementing a custom renderer for the `Entry`, all entries in your project will have a colored border. By sub-classing the `Entry` into `MyEntry` and applying the renderer to that, you can apply the color to only those controls.

Let's proceed with the first step, creating a custom control. This does not have to be too hard; it can be just an inheritance of the default control without adding anything new. At least, if in the future your requirements change, you have the possibility to still add something to your custom control.

The entry for this example is shown in Listing 1-1 and is called `MyEntry`.

Listing 1-1. The Custom Entry to Which We Will Apply the Custom Renderer

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

We have now simply created a new type with the sole purpose of not rendering all entry controls, but only the ones of the `MyEntry` type.

For the second step, we need to implement this control somewhere in our app. Since we do not have one sample app that we will work with in this book, refer to Listing 1-2 for a sample code snippet.

Listing 1-2. A Basic Page in XAML Using the MyEntry 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:SolutionsSampleApp"
             x:Class="SolutionsSampleApp.MainPage">
    <local:MyEntry Text="Welcome to Xamarin Forms!"
    VerticalOptions="Center" HorizontalOptions="Center" />
</ContentPage>
```

The code in Listing 1-2 is *XAML* (Extensible Application Markup Language). With this XML-like syntax you can define your UI design. For this book, we assume that you already have worked with Xamarin.Forms before and that you know what XAML is. If not, refer to this link for more information: `https://docs.microsoft.com/en-us/xamarin/xamarin-forms/xaml/xaml-basics/`.

Note Throughout the course of this book, we will be defining the UI in XAML. Everything that can be done in XAML can also be done in code. Whether you want to use XAML or code is mostly a matter of taste.

There are two things notable in the code in Listing 1-2. First, make sure that you included a new namespace declaration in the root of the page. In Listing 1-2, it is this line: `xmlns:local="clr-namespace:SolutionsSampleApp"`. This is similar to adding a `using` statement in your class. Now, the local prefix is available to declare controls in the namespace that is associated to it. You will see this prefixing of tags more often in the remainder of this book, now you know that this is because of a namespace declaration. This is what the `<local:MyEntry />` tag does. It defines a new control, our custom control, and takes it from the namespace that we have abbreviated as `local`. Since our `MyEntry` control is just an inheritance of the `Entry` control, all properties of the `Entry` are also available in our control.

Now that we have our own custom control and consumed it in our app, it is time for the final step: create the custom renderer for it. The renderer classes are to be created for each platform that we want to support. But note that you are not required to implement a custom renderer for each platform if you decide to implement it for one platform.

For example, if you want to implement something specific to iOS, you can create a renderer for iOS only. On Android (or any other platform), the default renderer will stay in effect and nothing will change there.

To create a custom renderer, create a new class in the targeted platform project. In this example, I will name it `MyEntryRenderer`. Let's first focus on iOS. The full implementation can be seen in Listing 1-3.

Listing 1-3. The Implemented iOS Custom Renderer

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

        if (Control != null)
        {
            // Platform native control is created here, have at it
            Control.BackgroundColor = UIColor.FromRGB (42, 42, 42);
            Control.BorderStyle = UITextBorderStyle.Line;
        }
    }
}
```

On the first line, you can see how we inherit from the default renderer, `EntryRenderer`. This is the renderer implemented by the Xamarin team. From that renderer, we override the `OnElementChanged` method. This pattern is what you will see for the most custom renderers.

The `ElementChangedEventArgs` contains a couple of properties that will tell you something about the lifecycle of the rendered control. During runtime, the `OnElementChanged` method can be called a couple of times. Here is a list of important properties and an explanation of what they tell you.

- `OldElement`—This can be null when a control is created. When it is not null, a control is being destroyed and you should unsubscribe from events and clean up any other resources.

- NewElement—This can also be null whenever a control is being destroyed. When it is not null, a control is being created and you should initialize it as needed.

Both OldElement and NewElement will contain a reference to the Xamarin.Forms variant of the control. In case of this example, it would be an Entry.

Inside a renderer, there are a couple of notable properties. I will explain them next.

- Control—This property contains a reference to the *native* control. In our example, when the renderer is initialized, it will contain a typed reference to a UITextField.
- Element—This will hold a reference to the Xamarin.Forms control. So, in our case that would be the Entry.

All these properties and objects together should give you enough information to understand the lifecycle of a control. Basically, you can use the boilerplate code, as seen in Listing 1-4.

Listing 1-4. Boilerplate for Any Custom Renderer

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

    if (Control == null)
    {
        // Instantiate the native control and assign it to the Control
        property
    }

    if (e.OldElement != null)
    {
        // Unsubscribe from event handlers and clean up any resources
    }
```

```
    if (e.NewElement != null)
    {
        // Configure the control and subscribe to event handlers
    }
}
```

Typically, you are probably only interested whenever the Control property is not null. That is, when the platform control is created and you can do your custom magic.

There is one very important piece of code missing to make this work. We have to register this renderer as the renderer for our MyEntry control. This is done through an attribute on the namespace level. To match our example, this would look like the code in Listing 1-5.

Listing 1-5. Register the Custom Renderer with the Runtime

```
[assembly: ExportRenderer (typeof(MyEntry), typeof(MyEntryRenderer))]
namespace CustomRenderer.iOS
{
    public class MyEntryRenderer : EntryRenderer
    {
        // Code from Listing 1-3
    }
}
```

The important line is at the top. With the ExportRenderer attribute, we register with this attribute we inform Xamarin that we want to use this type of renderer when it needs to render a view of type MyEntry. It takes two parameters—the type of the control that we want to render and the type of the renderer we want to use for it. Failing to add this attribute will result in the default renderer taking over and your renderer having no effect at all.

When we now run this code, the entry on iOS will have a background color.

At this stage, we don't have a custom renderer for the other platforms. As MyEntry inherits from Entry, Xamarin will use the default EntryRenderer to render this control on the other platforms. Hence there will be no custom background or anything—a native control will be rendered as if it were a standard Entry.

If we also want to give this control a background color on Android, we need to create a custom renderer in our Android project. Listing 1-6 shows the code that we need to implement for Android to give our `MyEntry` object a red background color.

Listing 1-6. Implementation of the Custom Renderer on Android

```
[assembly: ExportRenderer(typeof(MyEntry), typeof(MyEntryRenderer))]
namespace CustomRenderer.Android
{
    public class MyEntryRenderer : EntryRenderer
    {
        public MyEntryRenderer(Context context) : base(context)
        {
        }

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

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

You can see the concept is mostly the same. Only the code to set the background color on an Android control is different. As we learned earlier, the `Control` property contains a reference to the native control, in this case of type `EditText`. Since the renderers are implemented in the platform-specific projects, we can access the native APIs directly and set the background on the Android natives `EditText`. This really is how Xamarin.Forms works, inside these renderers—either made by you or the Xamarin team—the translation happens from the Forms control to its native counterpart.

Figure 1-3 shows a schematic overview of how all the components are arranged.

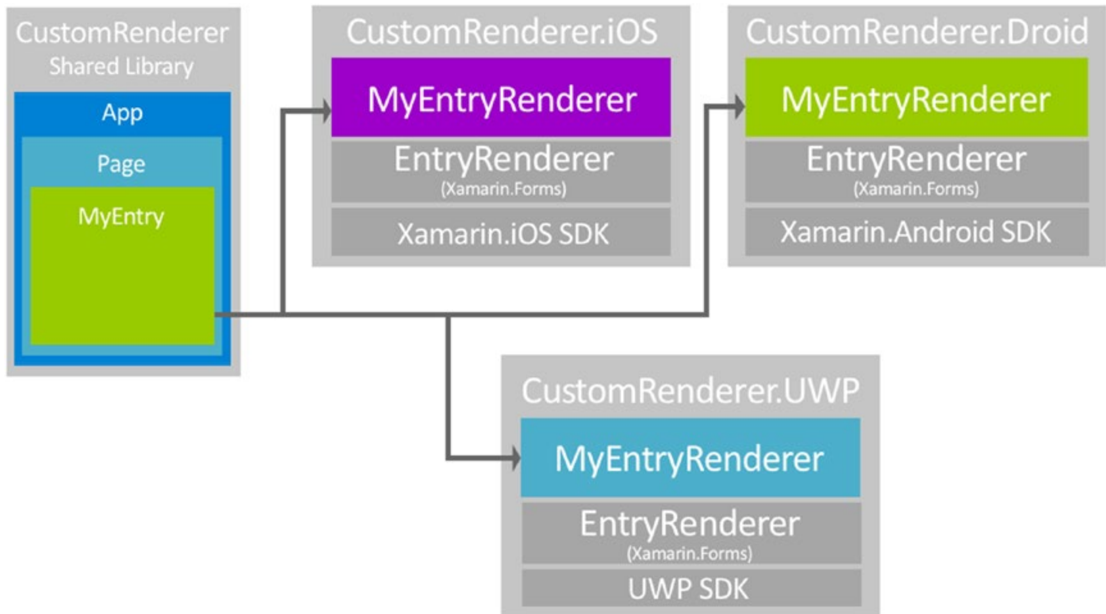

Figure 1-3. *The situation after implementing a custom renderer on each platform*

While UWP is depicted in Figure 1-3, we will not show you this example in code. It is mostly the same as the other examples or Android and iOS you saw earlier.

Tip For more information on how to create custom renderers, take a look at the documentation at `https://docs.microsoft.com/en-us/xamarin/xamarin-forms/app-fundamentals/custom-renderer/`.

Effects

Somewhat relatable to the custom renderers are *effects*. With effects, you do not need to subclass a whole renderer, instead you can write just enough code to customize the control a little bit.

Effects Versus Custom Renderers

Of course, then the question arises: when do you use an effect and when do you use a custom renderer? The short answer is that it's totally up to you. There are however a few differences between renderers and effects. While effects can be reused more easily, custom renderers offer more flexibility in terms of what you can achieve with them.

In the official documentation, this list can be found on when to choose effects over custom renderers:

- An effect is recommended when changing the properties of a platform-specific control will achieve the desired result.
- A custom renderer is required when there's a need to override methods of a platform-specific control.
- A custom renderer is required when there's a need to replace the platform-specific control that implements a Xamarin.Forms control.

All documentation on effects can be found at `https://docs.microsoft.com/en-us/xamarin/xamarin-forms/app-fundamentals/effects/introduction`.

Implementing an Effect

To create our own effect, we need to create a subclass of the `PlatformEffect` class. Each platform has its own `PlatformEffect` class and has three important properties:

- `Container`—Holds a reference to the parent the control is added to.
- `Control`—References the platform-specific control.
- `Element`—Contains a reference to the Xamarin.Forms control.

These properties are not typed and will contain references to high-level objects. You can see these objects per platform underneath:

- *iOS*—`UIView` both for `Container` and `Control`
- *Android*—`ViewGroup` (`Container`) or View (`Control`)
- *UWP*—`FrameworkElement` for `Container` and `Control`

The Container and Control properties are implemented as high-level properties so that they can be applied to all elements. You can strongly type the Container and Control properties by inheriting from the generic PlatformEffect<TContainer, TControl> class. For instance, on iOS if you know the Container should be a UIView and the Control should be a UILabel, you can declare your effect like this:

```
public class YourCustomEffect : PlatformEffect<UIView, UILabel>
```

Therefore, it is important to know for which control you are creating an effect so you can strongly type it or cast it to the right type yourself.

In addition to these properties, there are two methods that need to be overridden when creating an effect.

- OnAttached—This method is called when the effect is attached to the Forms control. In here you need to implement your custom styling, hooking up events, etc. You also need to take into account any error handling for when the effect fails to attach.
- OnDetached—Basically, this is the opposite of the attached method. You need to clean up any resources and unhook events, etc.

The PlatformEffect class also has a OnElementPropertyChanged method that can be overridden. This method is invoked whenever a property on the element changes. Please note that this method can be called many times potentially.

In order to implement an effect, we need to take five steps:

1. Create a new class that inherits from PlatformEffect in the platform project.
2. Override the OnAttached method to apply our new effect.
3. Override the OnDetached method and implement code to clean up after our effect, which is not always required.
4. Add a ResolutionGroupName attribute as part of the unique identifier of this effect. Think of this as the namespace for a class.
5. Add a ExportEffect attribute to register the effect with the runtime, equal to the way we register the custom renderer.

If we wanted to achieve the same result as with the custom renderer from the previous part of this chapter, we have to implement an effect like in Listing 1-7.

Listing 1-7. Implementing an Effect to Set the Background Color

```
[assembly:ResolutionGroupName ("com.MyApp")]
[assembly:ExportEffect (typeof(BackgroundColorEffect),
nameof(BackgroundColorEffect))]
namespace MyApp.iOS
{
    public class BackgroundColorEffect : PlatformEffect
    {
        protected override void OnAttached ()
        {
            try
            {
                Control.BackgroundColor = UIColor.FromRGB (42, 42, 42);
            }
            catch (Exception ex)
            {
                Console.WriteLine ("Cannot set property on attached
                control. Error: ", ex.Message);
            }
        }

        protected override void OnDetached ()
        {
            // No need to do anything in this example
        }
    }
}
```

This code shows you an example of how we can implement this effect for iOS. It shows a couple of similarities with the custom renderer. An effect also needs to be implemented at platform project level. Also, just like with the renderers, you will have to register the effect with an attribute. In the code from Listing 1-7, you can see it being done with the `ExportEffect` attribute above the namespace declaration. The additional `ResolutionGroupName` attribute is needed to be able to uniquely identify the effects in your project. The `ResolutionGroupName` together with the class name will make for

the fully qualified name. We will see this in a little bit. Also, you just need to set the ResolutionGroupName once per project. It will carry over to all other effects defined in the project.

For completeness, we implement the same effect on Android. You can see it in Listing 1-8.

Listing 1-8. Implementing the BackgroundEffect on Android

```
[assembly:ResolutionGroupName ("com.MyApp")]
[assembly:ExportEffect (typeof(FocusEffect), "FocusEffect")]
namespace MyApp.Droid
{
    public class BackgroundColorEffect: PlatformEffect
    {
        protected override void OnAttached ()
        {
            try
            {
                Control.SetBackgroundColor (Android.Graphics.Color.Red);
            } catch (Exception ex) {
                Console.WriteLine ("Cannot set property on attached
                control. Error: ", ex.Message);
            }
        }

        protected override void OnDetached ()
        {
        }
}
```

You can see the gist of the code is the same, but here you can also use the platform-specific APIs just like with renderers. This is because we are implementing the effects in the platform code.

Consuming an effect is a bit harder than with a custom renderer. The biggest difference here is that you have to keep in mind that each control will have its own instance of the effect, whereas a custom renderer will be used for all the controls going through. This also means that you don’t need to create your own inheritance of a control; you can just attach the effect to one instance of a regular control.

To be able to consume an effect in XAML, you need to create a subclass of the `RoutingEffect` class in your shared library. The class itself doesn't really do that much except for pointing to the right implementation on the platform. Listing 1-9 shows the `RoutingEffect` for our example `BackgroundColorEffect`.

Listing 1-9. RoutingEffect in Shared Code To Be Able to Consume Our Effect in XAML

```
public class BackgroundColorEffect : RoutingEffect
{
    public BackgroundColorEffect () : base ("com.MyApp.
    BackgroundColorEffect")
    {
    }
}
```

Notice how the effect in our shared code is named exactly the same as the effect in the platform project. Although this is not absolutely necessary, it helps identifying them easily. The real identification of the effect happens on this line:

```
public BackgroundColorEffect () : base ("com.MyApp.BackgroundColorEffect")
```

In the base call you have to specify the value from the `ResolutionGroupName` and the class name combined. From that, Xamarin.Forms will know which effect to route to.

With this in place we are now ready to use the effect in our XAML. This is shown in Listing 1-10.

Listing 1-10. Consuming the Effect in XAML

```
<?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:SolutionsSampleApp.Effects"
             x:Class="SolutionsSampleApp.MainPage">
    <Entry Text="Welcome to Xamarin Forms!" VerticalOptions="Center"
    HorizontalOptions="Center">
        <Entry.Effects>
            <local:BackgroundColorEffect />
        </Entry.Effects>
```