Quick answers to common problems

Xamarin Cross-Platform Development Cookbook

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

George Taskos

[PACKT] PUBLISHING open source* community experience distilled

Xamarin Cross-Platform Development Cookbook

Table of Contents

Xamarin Cross-Platform Development Cookbook

Credits

About the Author

About the Reviewer

www.PacktPub.com

- eBooks, discount offers, and more
  - Why Subscribe?

Preface

- What this book covers
- What you need for this book
- Who this book is for
- Sections
  - Getting ready
  - How to do it…
  - How it works…
  - There's more…
  - See also
- Conventions
- Reader feedback
- Customer support
  - Downloading the example code
  - Errata
  - Piracy
  - Questions

1. One Ring to Rule Them All

- Introduction
- Creating a cross-platform solution
  - Getting ready
  - How to do it…

How it works…
See also
Creating a cross-platform login screen
How to do it…
How it works…
There’s more…
Using common platform features
How to do it…
How it works…
See also
Authenticating with Facebook and Google providers
How to do it…
For the Android project
For the iOS project
How it works…
See also
2. Declare Once, Visualize Everywhere
Introduction
Creating a tabbed-page cross-platform application
How to do it…
How it works…
See also
Adding UI behaviors and triggers
How to do it…
How it works…
See also
Configuring XAML with platform-specific values
How to do it…
How it works…
See also
Using custom renderers to change the look and feel of views

How to do it…
How it works…
See also
3. Native Platform-Specific Views and Behavior
Introduction
Showing native pages with renderers
How to do it…
How it works…
See also
Attaching platform-specific gestures
How to do it…
How it works…
See also
Taking an in-app photo with the native camera page
How to do it…
How it works…
See also
4. Different Cars, Same Engine
Introduction
Sharing code between different platforms
How to do it…
How it works…
See also
Using the dependency locator
How to do it…
How it works…
See also
Adding a third-party Dependency Injection Container
How to do it…
How it works…
See also

Architecture design with Model-View-ViewModel (MVVM) pattern
How to do it…
How it works…
There's more…
See also
Using the event messenger
How to do it…
How it works…
See also
Adding localization
How to do it…
How it works…
There's more…
5. Dude, Where's my Data?
Introduction
Creating a shared SQLite data access
How to do it…
How it works…
See also
Performing CRUD operations in SQLite
How to do it…
How it works…
See also
Consuming REST web services
How to do it…
How it works…
Leveraging native REST libraries and making efficient network calls
How to do it…
How it works…
There's more…
6. One for All and All for One

Introduction
Creating cross-platform plugins
How to do it…
How it works…
There’s more…
See also
Taking or choosing photos
How to do it…
How it works…
There’s more…
See also
Getting the GPS location
How to do it…
How it works…
See also
Show and schedule local notifications
How to do it…
How it works…
There’s more…
7. Bind to the Data
Introduction
Binding data in code
How to do it…
How it works…
Binding data in XAML
How to do it…
How it works…
Configuring two-way data binding
How to do it…
How it works…
See also

Using value converters
How to do it…
How it works…
See also
8. A List to View
Introduction
Displaying a collection and selecting a row
How to do it…
How it works…
See also
Adding, removing, and refreshing items
How to do it…
How it works…
There's more…
See also
Customizing the row template
How to do it…
How it works…
There's more…
See also
Adding grouping and a jump index list
How to do it…
How it works…
There's more…
See also
9. Gestures and Animations
Introduction
Adding gesture recognizers in XAML
How to do it…
How it works…
There's more…

- See also
- Handling gestures with native platform renderers
  - How to do it…
  - How it works…
- Adding cross-platform animations
  - How to do it…
  - How it works…
  - There's more…

10. Test Your Applications, You Must
- Introduction
- Creating unit tests
  - How to do it…
  - How it works…
- Creating acceptance tests with Xamarin.UITest
  - How to do it…
  - How it works…
  - There's more…
  - See also
- Using the Xamarin.UITest REPL runtime shell to test the UI
  - How to do it…
  - How it works…
  - See also
- Uploading and running tests in Xamarin Test Cloud
  - How to do it…
  - How it works…
  - There's more…
  - See also

11. Three, Two, One – Launch and Monitor
- Introduction
- Using Xamarin Insights
  - How to do it…

How it works…
There's more…
Publishing iOS applications
How to do it…
How it works…
Publishing Android applications
How to do it…
How it works…
There's more…
See also
Publishing Windows Phone applications
How to do it…
How it works…
There's more…
Index

Xamarin Cross-Platform Development Cookbook

Xamarin Cross-Platform Development Cookbook

Copyright © 2016 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: March 2016

Production reference: 1210316

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-78588-053-7

www.packtpub.com

Credits

Author

George Taskos

Reviewer

Stan Okunevic

Commissioning Editor

Amarabha Banerjee

Acquisition Editor

Reshma Raman

Content Development Editor

Samantha Gonsalves

Technical Editor

Nirant Carvalho

Copy Editors

Dipti Mankame

Jonathan Todd

Project Coordinator

Sanchita Mandal

Proofreader

Safis Editing

Indexer

Priya Sane

Graphics

Kirk D’Penha

Production Coordinator

Shantanu N. Zagade

Cover Work

Shantanu N. Zagade

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.

I would like to thank my wife, Natalya Taskos, for putting up with my all night sessions in the writing of the book and trying to find the next best practice and technology to create the amazing software.

About the Reviewer

Stan Okunevic is a speaker, entrepreneur, and a true lover of All Things Mobile. Having dropped out of university to start a career in software consultancy, he entered the world of Xamarin by accident—having stumbled upon mentions of MonoTouch while looking for something new to learn— and has not looked back since. Having worked with a wide array of clients ranging from small, stealth-mode start-ups to Fortune 50 companies, he has gained invaluable insight into what it takes to build a product and run a team.

Nowadays, he spends most of his time expanding his technology consultancy, building various products of his own whenever an opportunity arises, studying for an MSc degree in Cyber Security and updating his personal blog at Nowadays, he spends most of his time expanding his technology consultancy, building various products of his own whenever an opportunity arises, studying for an MSc degree in Cyber Security and updating his personal blog at daytimehacker.com.

I'd like to thank each and every one of my clients to trust me and my team, as well as everyone I have worked with in the early stages of my career to provide invaluable advice. Special thanks to Paul F. Johnson for the review recommendation that reviewing the book has been of immense pleasure.

www.PacktPub.com

eBooks, discount offers, and more

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at <customercare@packtpub.com> for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

https://www2.packtpub.com/books/subscription/packtlib

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Preface

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 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 data-binding 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.

What you need for this book

- On Mac:
  - Xamarin Studio 5.10.1 (build 6)
  - The latest iOS SDK (Currently in version 9.2)
  - Xcode 7.1
  - OS X 10.10.5+ (Yosemite) or 10.11 (El Capitan)
- On Windows:
  - Any non-Express edition of Visual Studio 2012
  - Visual Studio 2013
  - Visual Studio 2015 (Community, Professional, and Enterprise)
- Visual Studio Extensions for iOS and Android

All examples will work with the Free Trial evaluation or Xamarin Starter licenses. Xamarin Starter is installed by default with Visual Studio 2015 and works with VS 2012, 2013, and 2015 (including Community editions).

Who this book is for

This book is for mobile developers. You must have some basic experience of C# programming, but no previous experience with Xamarin is required. If you are just starting with C# and want to use Xamarin to develop cross-platform apps effectively and efficiently, then this book is the right choice for you.

Sections

In this book, you will find several headings that appear frequently (Getting ready, How to do it…, How it works…, There’s more…, and See also).

To give clear instructions on how to complete a recipe, we use these sections as follows:

Getting ready

This section tells you what to expect in the recipe, and describes how to set up any software or any preliminary settings required for the recipe.

How to do it…

This section contains the steps required to follow the recipe.

How it works…

This section usually consists of a detailed explanation of what happened in the previous section.

There’s more…

This section consists of additional information about the recipe in order to make the reader more knowledgeable about the recipe.

See also

This section provides helpful links to other useful information for the recipe.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: “With the `Auth.GetUI()` method, we get a platform-specific object that we can use to show our authentication view.”

A block of code is set as follows:

```
Position position = new Position(40.730599, -73.986581);
Pin pin = new Pin {
  Type = PinType.Place,
  Position = position,
  Label = "New York",
  Address = "New York"
};

map.Pins.Add(pin);
```

Any command-line input or output is written as follows:

```
[0:] Exception: System.NotSupportedException: Need to fix this!
  at XamFormsMessagingCenter.MainPage.OnAppearing ()……..
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: ” Start Visual Studio and create a new **Blank App** (Xamarin.Forms Portable) project.”

Note

Warnings or important notes appear in a box like this.

Tip

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail <feedback@packtpub.com>, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this book from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on **Code Download**.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting http://www.packtpub.com/submit-errata, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to https://www.packtpub.com/books/content/support and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at <copyright@packtpub.com> with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at <questions@packtpub.com>, and we will do our best to address the problem.

Chapter 1. One Ring to Rule Them All

In this chapter, we will cover the following recipes:

- Creating a cross-platform solution
- Creating a cross-platform login screen
- Using common platform features
- Authenticating with Facebook and Google providers

Introduction

Xamarin.Forms is a cross-platform UI framework where the idea is no longer to share only your models, business logic, and data access layers similar to a traditional Xamarin solution but also the **user interface** (**UI**) across iOS, Android, and Windows Phone. With Xamarin.Forms, you can easily and quickly create great data-driven and utility applications or prototypes.

To accomplish this, Xamarin uses the super-modern C# language, the power of the .NET **Base Class Libraries** (**BCL**), these are C# bindings to the native APIs, and two great IDEs, Xamarin Studio and Microsoft Visual Studio. You can't, however, create iOS applications if you don't have a Mac connected to the network and acting as a server to build and deploy your application with the help of the Xamarin Build Host.

This book will provide you with real-world recipes and step-by-step development of the most common practices that you need to create professional cross-platform applications. You will learn how to create one UI across all platforms, customize the layout and views, and inject implementation per platform with a focus on patterns and best practices.

In this chapter, we will dive into the details of creating a cross-platform solution, adding a login screen, storing values for each platform, and using the `Xamarin.Auth` component to allow your users to log in with Facebook and Google providers. Neat! Exactly what you need to create a real-world application.

Creating a cross-platform solution

Getting started with apps for Xamarin.Forms is very easy. The installer sets everything up, the IDE creates the project, and you’re up and running in minutes! Lean!

Getting ready

Before we can start creating cross-platform apps, we need to get our tools in place using a single installer from Xamarin:

1. Go to http://xamarin.com/download

2. Enter your registration details.
3. Click the **Download Xamarin for OS X** button.
4. Once the download has completed, launch the installer, following the onscreen instructions. The setup will continue to download and install all the required components.

Once the install is finished, you will have a working installation of Xamarin Studio, the IDE designed for cross-platform development:

Default
Default
Xamarin Studio
Search
Xamarin Studio
Developer Center
Support
Forums
Georgios Taskos
New Solution...
Open...
FormsCookbook
...gtaskos/Downloads/FormsCookbook
FormsCookbook
...ergiostaskos/Projects/FormsCookbook
Xamarin Podcast: Exploring Xamarin Insights
This week on the Xamarin Podcast, James and I discuss our experiences with mobile application analytic and crash reporting tools to celebrate Xamarin Insights moving to production. Subscribe or Download...
Now Playing: iOS 9, Up Next: tvOS
Yesterday was a big day for Apple announcements. Beyond new iPhones and bigger iPads, the big news for many developers was the new Apple TV that will be joining the lineup of devices with an SDK and an...
Live Webinar: Go Mobile with Xamarin and Azure
We'd like to invite you to join us on September 23, 2015 for a joint webinar with Microsoft Azure. Kirill Gavrylyuk from Microsoft's Azure App Services team and I will cover multiple demos...
Join Xamarin for an Oracle Cloud Day Near You
We recently announced our partnership with Oracle Mobile Cloud Service, and together, we're helping organizations build better, scalable enterprise apps. Now, we're joining them for Oracle Cloud...
10000
Task List
Download Solution
Step Counter
Download Solution

How to do it…

Creating a Xamarin.Forms solution is easy using the default templates in Xamarin Studio or Visual Studio. Depending on the IDE, you will get three (Xamarin Studio in Windows), four (Xamarin Studio in Mac), and four (Visual Studio) projects. Of course, you can open the solution in the desired IDE and add the projects missing while moving forward with development. In this section, we will create a Xamarin.Forms blank application in Xamarin Studio for Mac and then add the Windows Phone project in Visual Studio.

In Xamarin Studio, choose **File | New | Solution**, and in the Cross-platform section, App category, you will see two options: **Blank Xamarin.Forms App** and **Single View App**, as shown in the following screenshot:

Choose Blank Xamarin.Forms App and click the **Next** button. You get to a screen with the options **App Name**, the name of your applications, and **Identifier**, which will be used as your package name for Android and the bundle identifier for iOS and the platforms you want to target.

The last option is if you want to use a **Portable Class Library** or a **Shared Library** as shown in the following screenshot. Shared Library will create a project that uses conditional compilation directives and it will be built in your main binary. With a Portable

Class Library, on the other hand, you get to choose a profile you want to target and also distribute it via NuGet or the Xamarin Component Store. In this book, we only focus on the option of a Portable Class Library and its corresponding patterns and practices.

Click the **Next** button. Enter the project name, which will be used to create the names for each platform-specific project with the convention `[ProjectName].[Platform]` and a solution name as shown in the following screenshot. You can choose where to save the solution and if you want to use Git version control.

Click **Create**. Notice that we have a solution with four projects. You will learn later in Chapter 10, *Test Your Applications, You Must* how to create tests for our solution. This project is created only if we create the solution with Xamarin Studio.

With no hesitation, choose your iPhone simulator and click the play button. You will see the simulator starting and a simple app running with the message **Welcome to Xamarin Forms!**. Congratulations, you can do the same for Android by right-clicking in the project and **Set As Startup Project**. For Android projects, you can use either Google's emulators or the Xamarin Android Player, https://xamarin.com/android-player, which personally I find more efficient and faster. Click the play button and enjoy the same message in your favorite emulator.

Sweet! Three clicks and we have an application running in iOS and Android! Now, before we examine the project structure, let's jump into Visual Studio and add the Windows Phone project.

1. In Visual Studio, go to **File | Open | Project/Solution** and choose the `ProjectName.sln` file; in the example's case, the `FormsCookbook.sln` file.
2. In the Solution Explorer, right-click your solution and **Add | New Project**.
3. In the section **Store Apps | Windows Phone Apps**, click in **Blank App (Windows**

Phone Silverlight) and choose a name; to be consistent, let's name it `[ProjectName].WinPhone` , then press the **OK** button.

4. Choose either **Windows Phone 8.0** or **Windows Phone 8.1**.

   Voila! Another platform is added but it's missing some parts to work with our Xamarin.Forms solution. We need to add the required packages/libraries and make a small modification to the application starting point code.

5. Right-click in **References** of the Windows Phone project we just created and choose **Manage NuGet Packages**.
6. Search for the `Xamarin.Forms` package and hit **Install** to add it to the Windows Phone project.

7. We also need to reference the PCL library. Right-click to **References | Add Reference**, and in the section **Solution** you will find the PCL with the Project Name. Check the box and hit **OK**.

We're almost done. We just need some code changes to the XAML `MainPage` to convert it to an Xamarin.Forms application.

8. Select the `MainPage.xaml` file and modify the root tag from `PhoneApplicationPage` to `FormsApplicationPage` as shown in the following screenshot:

```
<winPhone:FormsApplicationPage
    x:Class="FormsCookbook.WinPhone.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:phone="clr-namespace:Microsoft.Phone.Controls;assembly=Microsoft.Phone"
    xmlns:shell="clr-namespace:Microsoft.Phone.Shell;assembly=Microsoft.Phone"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:winPhone="clr-namespace:Xamarin.Forms.Platform.WinPhone;assembly=Xamarin.Forms.Platform.WP8"
    mc:Ignorable="d"
    FontFamily="{StaticResource PhoneFontFamilyNormal}"
    FontSize="{StaticResource PhoneFontSizeNormal}"
    Foreground="{StaticResource PhoneForegroundBrush}"
    SupportedOrientations="Portrait" Orientation="Portrait"
    shell:SystemTray.IsVisible="True">
```

9. You might need to bring the namespace in XAML. If you use any fancy tool like ReSharper you already took care of it; if not, add the following line to your project root tag that we just modified. Again, check the final result in the preceding screenshot.

```
xmlns:winPhone="clr-
```

```
namespace:Xamarin.Forms.Platform.WinPhone;assembly=Xamarin.Forms.Platfo
rm.WP8"
```

10. We need to check the **MainPage.xaml.cs** behind-code file as well and change it to inherit from `FormsApplicationPage` and the code in lines 13 and 14 as in the following screenshot:

```
FormsCookbook.WinPhone                    FormsCookbook.WinPhone.MainPage
 1  using Xamarin.Forms;
 2  using Xamarin.Forms.Platform.WinPhone;
 3
 4  namespace FormsCookbook.WinPhone
 5  {
 6      public partial class MainPage : FormsApplicationPage
 7      {
 8          // Constructor
 9          public MainPage()
10          {
11              InitializeComponent();
12
13              Forms.Init();
14              LoadApplication(new FormsCookbook.App());
15          }
16      }
17  }
```

11. Right-click to the **Windows Phone** project and set as **Start Up Project**; press *F5* or from the **Debug | Start Debugging** menu.

Congratulations! You have created a solution with three platform-specific projects (iOS, Android, and Windows Phone) and a core class library (PCL). Remember, the Windows Phone project will only be active when the solution is opened in Visual Studio.

How it works…

So, we still have the beauty of native applications but we get to share the UI code as well! Sweet!

See in the following screenshot the Xamarin.Forms platform architecture that reflects what we just created:

Our three platform-specific projects are at the top architecture layer, the shared C# user interface code below, and the Shared C# App Logic at the bottom.

Xamarin.Forms allows you to describe the UI once using a shared set of controls while still rendering a native UI. This is happening with the help of the native platforms' renderers; there is one for each supported control for every platform. So, when we add a **Label control**, the platform renderer will transform it to a **UILabel** for iOS, a **TextView** for Android, and a **TextBlock** for Windows Phone.

In the core Portable Class Library project, expand **References | From Packages**. Referenced are the libraries `Xamarin.Forms.Core` and `Xamarin.Forms.Xaml`; the first one contains all the core logic of Xamarin.Forms and the second is specific to the XAML code, which we will discuss in detail in Chapter 2, *Declare Once, Visualize Everywhere*.

Go to a platform-specific project now; let's see Android first. Expand **References | From Packages**, and we have again `Xamarin.Forms.Core` and `Xamarin.Forms.Xaml`, but also `Xamarin.Forms.Platform.Android`. This is the platform-specific library where all the Android renderers live and its responsibility is to take the UI abstractions and translate to a platform-specific UI.

The same applies to our iOS project with the platform-specific library having the name `Xamarin.Forms.Platform.iOS`.

In the references of the platform-specific projects you can see our core PCL reference too; it couldn't work without our base code, right?

That is all great! But how do these platform-specific projects connect to the Xamarin.Forms application? For that we have to look in the code behind every platform

application entry point.

Start from the Windows Phone project. We modified our `MainPage.xaml.cs` code behind, and now it is not a Windows Phone Page type but `Xamarin.Forms.Platform.WinPhone.FormsApplicationPage`. The important calls is in the constructor, `Forms.Init`, and the `LoadApplication` methods. In the latter we pass a `Xamarin.Forms.Appplication` instance located in our PCL. Open the `FormsCookbook.cs` file in the PCL project. Here is our Xamarin.Forms entry point, with some UI code in the constructor. In the next section, we discuss and change this code.

For the iOS project, you will find the related method calls in the `AppDelegate.cs`, `FinishedLaunching` method implementation. The difference is that the `AppDelegate` class inherits from `Xamarin.Forms.Platform.iOS.FormsApplicationDelegate`.

And finally, in the Android project, the `MainActivity.cs` file is our `MainLauncher` activity and inherits from `Xamarin.Forms.Platform.Android.FormsApplicationActivity`. It initializes the platform specifics and loads the Xamarin.Forms application class in the `OnCreate` method implementation. Notice the difference in the `Forms.Init` method: it requires two parameters: an Activity and Bundle instances.

See also

- https://developer.xamarin.com/guides/cross-platform/xamarin-forms/controls/views/

Creating a cross-platform login screen

Almost every application has an authentication screen. Here, we will examine the root application page of a Xamarin.Forms application by creating a login screen where the user can set his username, password and tap the login button to get access.

You can create a new solution or continue from the previous section.

How to do it…

In our core PCL project:

1. Open the `FormsCookbook.cs` file.
2. Replace the code in the constructor with the following code snippet:

```
var userNameEntry = new Entry {
  Placeholder = "username"
};
var passwordEntry = new Entry {
  Placeholder = "password",
  IsPassword = true
};
var loginButton = new Button {
  Text = "Login"
};

loginButton.Clicked += (sender, e) => {
  Debug.WriteLine(string.Format("Username: {0} - Password: {1}",
userNameEntry.Text, passwordEntry.Text));
};

MainPage = new ContentPage {
  Content = new StackLayout {
    VerticalOptions = LayoutOptions.Center,
    Children = {
      userNameEntry,
      passwordEntry,
      loginButton
    }
  }
};
```

3. Run the application for each platform; you should get the two native default text input controls and a button.
4. Set your username and password in the corresponding textboxes.
5. Click the **Login** button and watch the application output printing the message we set up in the Clicked delegate event handler.

You should see something similar to the following screenshot:

In iOS:

iOS Simulator - iPhone 6 - iPhone 6 / iOS 8.4...
Carrier
10:13 PM
username
password
Login

In Android:

Xamarin Android Player - [Nexus 5 (KitKat)]
10:15
username
password
Login

In Windows:

This is all great: the default layout looks OK and everything is aligned in the middle of the screen and to the edge of the screen. Of course, every view and layout container has properties that will align and add spacing between controls. Let's try to change things up a little bit. It's good to start organizing our code better too.

1. Right-click in the core PCL project and **Add | New Folder**; name it `Custom Pages`.
2. Right-click in the **Custom Pages** folder and **Add | New File**; choose **Empty Class**, set the name to `MainPage`, and click **New**.
3. In the constructor of the new class, write or paste the following code:

```
public class MainPage: ContentPage {
  Entry userNameEntry;
  Entry passwordEntry;
```

```
    Button loginButton;
    StackLayout stackLayout;

    public MainPage() {
      userNameEntry = new Entry {
        Placeholder = "username"
      };
      passwordEntry = new Entry {
        Placeholder = "password",
        IsPassword = true
      };
      loginButton = new Button {
        Text = "Login"
      };

      loginButton.Clicked += LoginButton_Clicked;

      this.Padding = new Thickness(20);

      stackLayout = new StackLayout {
        VerticalOptions = LayoutOptions.FillAndExpand,
        HorizontalOptions = LayoutOptions.FillAndExpand,
        Orientation = StackOrientation.Vertical,
        Spacing = 10,
        Children = {
        userNameEntry,
        passwordEntry,
        loginButton
        }
      };

      this.Content = stackLayout;
    }

    void LoginButton_Clicked(object sender, EventArgs e) {
      Debug.WriteLine(string.Format("Username: {0} - Password: {1}",
      userNameEntry.Text, passwordEntry.Text));
    }
  }
```

4. Make the `MainPage` class a subclass of `ContentPage`.

   ```
   public class MainPage : ContentPage
   ```

5. Replace the `FormsCookbook.cs` constructor with the following code:

   ```
   MainPage = new MainPage ();
   ```

6. Run the application for each platform to see the results.

You should see something similar to the following screenshot:

In iOS:

iOS Simulator - iPhone 6 - iPhone 6 / iOS 8.4...
Carrier
10:46 PM
username
password
Login

In Android:

Xamarin Android Player - [Nexus 5 (KitKat)]
10:45
username
password
Login

In Windows:

10:48
username
password
Login

How it works…

`FormsCookbook.cs` is the main application entry that contains a `MainPage` property where we assign the application initial page.

For the Content of the `MainPage` we create a `StackLayout`, and for its children we add our three Views: `userNameEntry`, `passwordEntry`, and `loginButton` with `VerticalOptions`, `HorizontalOptions`, `Orientation`, `Spacing`.

We assigned a delegate handler for the Clicked event of the `loginButton` to respond to our finger tap. Nothing fancy here, just printing a message to the Application Output window.

The Xamarin.Forms UI hierarchy is based on a more adaptive layout; we don’t specify positions with coordinates, but rather with layout containers, which we add child elements and specific rules.

To fine-tune our login screen, we used the `StackLayout's` container properties, `VerticalOptions` and `HorizontalOptions`, to determine how the child content is positioned, spacing for the space between the controls and padding for around the element.

Note

When it comes to width and height, there is no straightforward way to set these properties in Xamarin.Forms views. What we actually do is request the width and height, and with no guarantee that it will happen the layout will try to complete your request. The Width and Height are read-only properties.

There’s more…

When it comes to a native cross-platform application like Xamarin.Forms, don’t forget that we are running in different platforms, and different platforms have different screen sizes with different measurement systems: iOS units, Android and Windows Phone, device-independent pixels.

How does Xamarin.Forms deal with that? For example, if you say `Spacing=10` then it will be 10 in the specific platform’s measurement system, 10 units in iOS, and 10 dpi in Android and Windows Phone.

Using common platform features

The biggest challenge when developing a cross-platform application is the specific platform features APIs. Many common platform features exists across the platforms, but there isn't always a simple way to use them. For instance, showing an alert dialog message box or opening a URL from your application is a supported feature for iOS, Android, and Windows Phone but the APIs are completely different and platform specific, and a Portable Class Library has no access to all these features.

For our convenience, Xamarin.Forms supports some of the common platform-specific features out of the box. The architectural design and how you can create your own abstractions and implementations to use them in runtime in the portable library is discussed in Chapter 4, *Different Cars, Same Engine*.

How to do it…

We will explore the following Xamarin.Forms APIs:

- `Device.OpenUri` – opens a URL in native browser
- `Device.StartTimer` – triggers time-dependent tasks
- `Device.BeginInvokeOnMainThread` – any background code that needs to update the user interface
- `Page.DisplayAlert` – shows simple alert message dialogs
- `Xamarin.Forms.Map` – a NuGet cross-platform map library

Let's create an application to see the usage of the preceding APIs.

1. Create a cross-platform project from scratch using the default template named `XamFormsCommonPlatform`.
2. Create a new folder and add a new class named `MainPage.cs`.
3. Paste the following code:

```
public class MainPage: ContentPage {
  private Button openUriButton;
  private Button startTimerButton;
  private Button marshalUIThreadButton;
  private Button displayAlertButton;
  private Button displayActionSheetButton;
  private Button openMapButton;
  private StackLayout stackLayout;

  public MainPage() {
    openUriButton = new Button {
      Text = "Open Xamarin Evolve"
    };
    startTimerButton = new Button {
      Text = "Start timer"
    };
    marshalUIThreadButton = new Button {
      Text = "Invoke on main thread"
    };
    displayAlertButton = new Button {
      Text = "Display an alert"
    };
    displayActionSheetButton = new Button {
      Text = "Display an ActionSheet"
    };
    openMapButton = new Button {
      Text = "Open platform map"
    };

  openUriButton.Clicked += OpenUriButton_Clicked;
  startTimerButton.Clicked += StartTimerButton_Clicked;
  marshalUIThreadButton.Clicked += MarshalUIThreadButton_Clicked;
  displayAlertButton.Clicked += DisplayAlertButton_Clicked;
  displayActionSheetButton.Clicked += DisplayActionSheetButton_Clicked;
  openMapButton.Clicked += OpenMapButton_Clicked;
```

```
  stackLayout = new StackLayout {
    Orientation = StackOrientation.Vertical,
    Spacing = 10,
    Padding = new Thickness(10),
    VerticalOptions = LayoutOptions.FillAndExpand,
    HorizontalOptions = LayoutOptions.FillAndExpand,
    Children = {
    openUriButton,
    startTimerButton,
    marshalUIThreadButton,
    displayAlertButton,
    displayActionSheetButton,
    openMapButton
    }
  };

  Content = stackLayout;
}

  void OpenMapButton_Clicked(object sender, EventArgs e) {

}

  async void DisplayActionSheetButton_Clicked(object sender, EventArgs
e) {
    string action = await DisplayActionSheet("Simple ActionSheet",
"Cancel", "Delete", new string[] {
      "Action1",
      "Action2",
      "Action3",
    });

    Debug.WriteLine("We tapped {0}", action);
  }

  async void DisplayAlertButton_Clicked(object sender, EventArgs e) {
    bool result = await DisplayAlert("Simple Alert Dialog", "Sweet!",
"OK", "Cancel");
    Debug.WriteLine("Alert result: {0}", result ? "OK" : "Cancel");
  }

  void MarshalUIThreadButton_Clicked(object sender, EventArgs e) {
    Task.Run(async() => {
      for (int i = 0; i < 3; i++) {
      await Task.Delay(1000);
      Device.BeginInvokeOnMainThread(() => {
        marshalUIThreadButton.Text = string.Format("Invoke {0}", i);
      });
     }
    });
  }

  void StartTimerButton_Clicked(object sender, EventArgs e) {
    Device.StartTimer(new TimeSpan(0, 0, 1), () => {
     Debug.WriteLine("Timer Delegate Invoked");
     return true; // false if we want to cancel the timer.
```

```
    });
  }

  void OpenUriButton_Clicked(object sender, EventArgs e) {
    Device.OpenUri(new Uri("http://xamarin.com/evolve"));
  }
}
```

4. Ignore the `OpenMapButton_Clicked` method for now; we will need to make some configuration soon to make it work.
5. Run each application and test all the buttons except **Open platform map**, which has no functionality yet.

You should see something similar to the following screenshot:

In iOS:

iOS Simulator - iPhone 6 - iPhone 6 / iOS 8.4...
Carrier
9:46 PM
Open Xamarin Evolve
Start timer
Invoke on main thread
Display an alert
Display an ActionSheet
Open platform map

In Android:

In Windows Phone:

To add Cross-Maps to our application, we first need to make some platform-specific configurations.

1. For all the projects in our solution, add the `Nuget` package: `Xamarin.Form.Maps`.
2. For the iOS and Windows Phone application projects, add the following code after `Forms.Init()`. As a reminder, the call is in `MainActivity.cs` for Android and in `AppDelegate.cs` in iOS.

```
Xamarin.FormsMaps.Init();
```

3. In the Android project, we need to pass the two required parameters as the `Forms.Init` call.

```
Xamarin.FormsMaps.Init(this, bundle);
```

4. For the iOS 7 apps, there is no need for any modifications, but iOS 8 requires two keys in the `info.plist` file. Open the file and view its source, then add the following key/value pair:

NSLocationAlwaysUsageDescription	String	We always need to use your location!
NSLocationWhenInUseUsageDescription	String	We need to use your location when the application is active!

5. For Android, you need to get a Google Maps API v2 key; visit the link to complete this requirement: https://developers.google.com/maps/documentation/android/.
6. Open `Properties/AndroidManifest.xml`, view source, and add or update the following in your <application> tag:

```
<meta-dataandroid:name="com.google.android.maps.v2.API_KEY"
android:value="YOUR_API_KEY" />
```

7. Right-click the Android project and choose **Options**. Select the **Android Application** section and in **Required Permissions**, check the following:
   - Internet
   - AccessNetworkState
   - AccessCoarseLocation
   - AccessFineLocation
   - AccessLocationExtraCommands
   - AccessMockLocation
   - AccessWifiState
8. Open the `Properties/WMAppmanifest.xml` file in the Windows Phone Silverlight project and add the following in **Capabilities**:
   - `ID_CAP_MAP`
   - `ID_CAP_LOCATION`

   ## Note

   The `Xamarin.Forms.Maps` library does not currently support Windows Phone 8.1 or Windows 8.1 projects.

9. Add a new page with a right-click in the `Custom Pages` folder and **Add** | **New File**. Name it `MapPage`.
10. Make the `MapPage` class a `ContentPage` subclass.
11. Add the following code:

```
private Map map;
```

```
    private StackLayout stackLayout;

    public MapPage() {
      MapSpan span = MapSpan.FromCenterAndRadius(new Position(40.730599,
    -73.986581), Distance.FromMiles(0.4));

      map = new Map(span) {
        VerticalOptions = LayoutOptions.FillAndExpand
      };

      stackLayout = new StackLayout {
        Spacing = 0,
        Children = {
          map
        }
      };

      Content = stackLayout;
    }
```

12. In `MainPage.cs`, add the following line of code in the `OpenMapButton_Clicked` method:

    ```
    Navigation.PushModalAsync (new MapPage ());
    ```

13. Run your applications, tap the **Open platform map**, and check out Manhattan!

We can also change the map type and add pins on a map. The available pin types are the following:

- Generic
- Place
- SavedPin
- SearchResult

To add a pin on the map, use the following code:

```
Position position = new Position(40.730599, -73.986581);
Pin pin = new Pin {
  Type = PinType.Place,
  Position = position,
  Label = "New York",
  Address = "New York"
};

map.Pins.Add(pin);
```

Yes, it is that easy! The Xamarin team did a great job abstracting this common feature to accomplish the most basic operations with a map.

In iOS:

iPhone 6s - iPhone 6s / iOS 9.0 (13A340)
Carrier
FLATIRON BUILDING
9:23 AM
Factory Store
Dough
E 19th St
Park Ave S
Gramercy Park Hotel
E 24th St
E 25th St
E 23rd St
2nd Ave
ABC Kitchen
E 22nd St
E 21st St
E 17th St
Friend of A Farmer
E 20th St
E 19th St
E 18th St
Union Square Park
E 13th St
Starbucks
E 14th St
Stuyvesant Square
Broadway
4th Ave
1st Ave
Momofuku Milk Bar
E 11th St
Staples
E 13th St
E 9th St
Urban Outfitters
Momofuku Noodle Bar
E 12th St
MUJI
E 7th St
E 10th St
E 11th St
E 6th St
E 9th St
E 5th St
Merchant's House Museum
Museum of the American Gangster
E 3rd St
Tompkins Square Park
Death & Co
E 8th St
E 7th St
E 4th St
E 5th St
E 6th St
Anthology Film Archives
E 5th St
Cafe Habana
Russ & Daughters
E Houston St
E 2nd St
Avenue B
E 3rd St
Eldridge St
Allen St
Clinton Street Baking Company
Legal

In Android:

Xamarin Android Player - [Nexus 5 (KitKat)]
9:21
E 23rd St
5th Ave
Park Ave S
GRAMERCY PARK
Union Square
et - Union Square
Street - Union Square
3 Av
Broadway
4th Ave
3rd Ave
1 Av
E 14th St
NYU
Astor Pl
NOHO
2nd Ave
EAST VILLAGE
Bowery
BOWERY
1st Avenue
2nd Avenue
Allen St
E Houston St
Google

In Windows Phone:

How it works…

Xamarin has provided us with all the abstractions that we can reference in our PCL core library and the implementations of these platform-specific features for our native projects. Using the `DependencyService` (a service we will cook in Chapter 4, *Different Cars, Same Engine*), a Service Locator mechanism to resolve dependencies in runtime, Xamarin makes it possible for us to create native applications with these common features in a very short time.

`Device.OpenUri` in runtime will inject implementation that will allow it to use the iOS `UIAlertController`, in Android the `AlertDialog`, and in Windows Phone the `MessageBox` platform-specific APIs. `Device.StartTimer` will hide the equivalent timer APIs and the same practice goes for all the plugins we demonstrated.

The same rules apply for the `Xamarin.Forms.Map` library, but some configuration requirements are needed.

Note

One objective of Xamarin and especially Xamarin.Forms is to make us more productive; this is where the Xamarin and Community plugins come into play. Always check before you start reinventing the wheel!

See also

- Chapter 4, *Different Cars, Same Engine*
- https://developer.xamarin.com/guides/cross-platform/xamarin-forms/controls/layouts/
- https://components.xamarin.com/

Authenticating with Facebook and Google providers

One common feature that most of the applications utilize is to allow you to log in with your favorite login provider. We will close this chapter learning how you can leverage the `Xamarin.Auth` plugin to connect and authenticate two of the most famous login providers, Facebook and Google, to save your users from registering and using custom login steps, and to save your application from extra work in the backend and frontend tiers.

How to do it…

1. In Xamarin Studio or Visual Studio, create a new cross-platform project; if you don't remember how to, check the *Creating a cross-platform solution* recipe in this chapter. Name it `XamFormsAuthenticateProviders`.
2. Add the plugin in the iOS and Android projects, but don't bother with Windows Phone for now.
3. Right-click the `XamFormsAuthenticateProviders.iOS` components folder, **Edit Components**, search for `Xamarin.Auth`, and add it to the project. Do the same for the `XamFormsAuthenticateProviders.Droid` project.
4. To be able to authenticate your users with Facebook and Google, you will need to get your own client ID and secret from the corresponding developer consoles. For Facebook, go to https://developers.facebook.com, and for Google, got to https://console.developers.google.com.
5. The process in every platform to get your credentials is almost the same for each OAuth 2.0 provider: you create a new project and retrieve the keys. Refer to the related platform documentation for more information. The tricky part is to not forget to add the same redirect URLs to allow the application to end somewhere after a successful login.
6. In the `XamFormsAuthenticateProviders` core project, right-click and choose **Add | New Folder**; name it `Models`.
7. In the newly created folder, add two helper classes: `OAuthSettings` and `ProviderManager`.
8. Add the following code:

```
public class OAuthSettings {
  public string ClientId {
    get;
    set;
  }
  public string ClientSecret {
    get;
    set;
  }
  public string AuthorizeUrl {
    get;
    set;
  }
  public string RedirectUrl {
    get;
    set;
  }
  public string AccessTokenUrl {
    get;
    set;
  }
  public List < string > Scopes {
    get;
    set;
  }
```

```
  public string ScopesString {
    get {
      return Scopes.Aggregate((current, next) => string.Format("{0}+
{1}", current, next));
    }
  }

  public OAuthSettings() {
    Scopes = new List < string > ();
  }
}
public enum Provider {
    Unknown = 0,
    Facebook,
    Google
  }

public static class ProviderManager {
  private static OAuthSettings FacebookOAuthSettings {
    get {
      return new OAuthSettings {
        ClientId = "YOUR_CLIENT_ID",
        ClientSecret = "YOUR_CLIENT_SECRET",
        AuthorizeUrl = "https://m.facebook.com/dialog/oauth/",
        RedirectUrl =
"http://www.facebook.com/connect/login_success.html",
        AccessTokenUrl =
"https://graph.facebook.com/v2.3/oauth/access_token",
        Scopes = new List < string > {
          ""
        }
      };
    }
  }

  private static OAuthSettings GoogleOAuthSettings {
    get {
      return new OAuthSettings {
        ClientId = " YOUR_CLIENT_ID ",
        ClientSecret = " YOUR_CLIENT_ID ",
        AuthorizeUrl = "https://accounts.google.com/o/oauth2/auth",
        RedirectUrl = "https://www.googleapis.com/plus/v1/people/me",
        AccessTokenUrl = "https://accounts.google.com/o/oauth2/token",
        Scopes = new List < string > {
          "openid"
        }
      };
    }
  }

  public static OAuthSettings GetProviderOAuthSettings(Provider
provider) {
    switch (provider) {
      case Provider.Facebook:
      {
        return FacebookOAuthSettings;
```

```
      }
      case Provider.Google:
      {
        return GoogleOAuthSettings;
      }
    }
    return new OAuthSettings();
  }
}
```

9. Right-click on the core **XamFormsAuthenticate** project and **Add | New Folder**; name it **Custom Pages**.
10. Right-click on the newly created folder and add two classes, `LoginPage` and `ProvidersAuthPage`, and then write or paste-replace the file's contents with the following code:

```
public class LoginPage : ContentPage
{
  public OAuthSettings ProviderOAuthSettings { get; set; }

  public LoginPage (Provider provider)
  {
    ProviderOAuthSettings = ProviderManager.GetProviderOAuthSettings
(provider);
  }
}

public class ProvidersAuthPage : ContentPage
{
  StackLayout stackLayout;
  Button facebookButton;
  Button googleButton;

  public ProvidersAuthPage ()
  {
    facebookButton = new Button {
    Text = "Facebook"
  };
  facebookButton.Clicked += async (sender, e) =>
  await Navigation.PushModalAsync(new LoginPage(Provider.Facebook));

  googleButton = new Button {
    Text = "Google"
  };
  googleButton.Clicked += async (sender, e) =>
  await Navigation.PushModalAsync(new LoginPage(Provider.Google));

  stackLayout = new StackLayout {
    VerticalOptions = LayoutOptions.Center,
    HorizontalOptions = LayoutOptions.Center,
    Orientation = StackOrientation.Vertical,
    Spacing = 10,
    Children = {
      facebookButton,
      googleButton
```

```
    }
  };

  this.Content = stackLayout;
  }
}
```

11. In the App constructor of the `App.cs XamFormsAuthenticate` project file, add the following to set the start page:

```
MainPage = new ProvidersAuthPage();
```

Now we need to add the authentication code flow, but if you noticed when we added the `Xamarin.Auth` components in step 3, there are two different implementations: one for each platform. To accomplish our goal and run specific platform code for each project, we will use the `PageRenderer` class and an attribute that under the covers uses the Xamarin.Forms `DependencyService`. You can find more in-depth recipes in this book regarding these topics in Chapter 2, *Declare Once, Visualize Everywhere* and Chapter 4, *Different Cars, Same Engine*.

For the Android project

1. Create a new folder, name it `Platform Specific`, and add a new class file name `LoginPageRenderer`. Make it a `PageRenderer` subclass and add the following code:

```
LoginPage page;
bool loginInProgress;

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

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

    page = e.NewElement as LoginPage;

    if (page == null || loginInProgress)
    return;

    loginInProgress = true;
    try {
      // your OAuth2 client id
      OAuth2Authenticator auth = new
OAuth2Authenticator(page.ProviderOAuthSettings.ClientId,
      // your OAuth2 client secret
      page.ProviderOAuthSettings.ClientSecret,
      // scopes
      page.ProviderOAuthSettings.ScopesString,
      //the scopes, delimited by the "+" symbol
      new Uri(page.ProviderOAuthSettings.AuthorizeUrl),
      // the redirect URL for the service
      new Uri(page.ProviderOAuthSettings.RedirectUrl),
      new Uri(page.ProviderOAuthSettings.AccessTokenUrl));
```

```
      auth.AllowCancel = true;
      auth.Completed += async(sender, args) => {
        // Do something…
        await page.Navigation.PopAsync();
        loginInProgress = false;
      };

      auth.Error += (sender, args) => {
        Console.WriteLine("Authentication Error: {0}", args.Exception);
      };

      var activity = Xamarin.Forms.Forms.Context as Activity;
      activity.StartActivity(auth.GetUI(Xamarin.Forms.Forms.Context));
    } catch (Exception ex) {
        Console.WriteLine(ex);
      }
  }
```

2. We need some using statements to make all this and also a decoration attribute on the namespace.

```
using XamFormsAuthenticateProviders;
using XamFormsAuthenticateProviders.Droid;
using Xamarin.Forms.Platform.Android;
using Xamarin.Auth;
using Android.App;

[assembly: ExportRenderer(typeof(LoginPage),
typeof(LoginPageRenderer))]
namespace XamFormsAuthenticateProviders.Droid
```

For the iOS project

1. The same applies here as with the Android platform: create a folder and a new class file named `LoginPageRenderer`, and make it a subclass of `PageRenderer`.
2. This time, we need iOS-related code; you can find it in the following excerpt:

```
LoginPage page;
bool loginInProgress;

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

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

    page = e.NewElement as LoginPage;
  }

public override async void ViewDidAppear(bool animated) {
  base.ViewDidAppear(animated);

    if (page == null || loginInProgress)
```

```
        return;

      loginInProgress = true;
      try {
        // your OAuth2 client id
        OAuth2Authenticator auth = new
OAuth2Authenticator(page.ProviderOAuthSettings.ClientId,
        // your OAuth2 client secret
        page.ProviderOAuthSettings.ClientSecret,
        // scopes
        page.ProviderOAuthSettings.ScopesString,
        // the scopes, delimited by the "+" symbol
        new Uri(page.ProviderOAuthSettings.AuthorizeUrl),
        // the redirect URL for the service
        new Uri(page.ProviderOAuthSettings.RedirectUrl),
        new Uri(page.ProviderOAuthSettings.AccessTokenUrl));

        auth.AllowCancel = true;
        auth.Completed += async(sender, args) => {
        // Do something…
        await DismissViewControllerAsync(true);
        await page.Navigation.PopModalAsync();
        loginInProgress = false;
        };

        auth.Error += (sender, args) => {
          Console.WriteLine("Authentication Error: {0}", args.Exception);
        };

        await PresentViewControllerAsync(auth.GetUI(), true);

      }catch (Exception ex) {
        Console.WriteLine(ex);
       }
}
```

3. Add the using statements and the `ExportRenderer` attribute on the namespace.

```
using System;
using Xamarin.Forms.Platform.iOS;
using Xamarin.Forms;
using XamFormsAuthenticateProviders;
using XamFormsAuthenticateProviders.iOS;
using Xamarin.Auth;

[assembly: ExportRenderer(typeof(LoginPage),
typeof(LoginPageRenderer))]
namespace XamFormsAuthenticateProviders.iOS
```

The Windows Phone platform! Yes, the forgotten one. You must have noticed that there is no Windows Phone platform component in the Xamarin Component Store. Fear not, you have options. The magic answer is relying on the custom platform-specific `PageRenderer` that we add for each platform; you can create a platform-specific following the same steps and add your favorite Windows Phone OAuth library logic or you can follow the next

steps to try out the experimental alpha channel `Xamarin.Auth` NuGet package.

1. Right-click in the Windows Phone project and add the `Xamarin.Auth` NuGet package. You have to change the channel to Alpha in order to find it.
2. Create a folder named `Platform Specific` and add a class file named `LoginPageRenderer`; subclass it to `PageRenderer`.
3. Override the `OnElementChanged` method and add your code or `Xamarin.Auth`-related code.

Don’t forget to check the code available with this book for the complete picture.

That was a big milestone. Of course, it scratches the surface of access tokens, but we leave that to your preferences. Normally, you would get the access token, capture the expiration date, and persist it somewhere.

How it works…

You just had a taste of the power that Xamarin.Forms can provide you to create not only a unified application with shared code but also customize per platform when necessary. With the use of `PageRenderer`, you can completely provide your native views and pages.

The first thing we did after we created our solution is to go to the developer console of Facebook and Google, set up a new project, enable the OAuth2 system, retrieve our keys, and add the redirect URL we need to navigate the users after a successful login. In a few words, OAuth and OAuth is an authentication system to verify that this user is actually real and exists in the provider's records. It enables the user to use the same secure credentials from a provider he trusts to give access to the application he is about to use. You can also ask for various scopes and retrieve personal information that he has provided to his trusted provider.

Then we added two helper classes: one to provide us with all the common properties, `OAuthSettings`, that every provider needs in order to provide you with the desired access token; and a static class, `ProviderManager`, with a static method accepting a provider enumeration and returning an `OAuthSettings` instance for the required provider.

Our cross-platform UI consists of two pages: `ProvidersAuthPage`, which is our main page and has two buttons where in the Clicked event of each we navigate modally to the `LoginPage` passing the desired provider we need to authenticate our user.

In the `LoginPage` constructor, we use the provider value to retrieve an instance of `OAuthSettings` and set to a public property. That's it, everything else is handled in our platform-specific renderers.

Every `PageRenderer` we added has platform-specific functionality and is essentially our Xamarin.Forms page; so essentially in iOS, a `UIViewController`; in Android is almost like an Activity, but the lifecycle is a little bit different; and for Windows Phone, the same applies as a `PhoneApplicationPage` under the hood. There are more renderers that correspond to the type of page you want.

We also decorate our renderers namespace with the `ExportRenderer` attribute. With this, Xamarin.Forms knows which implementation to load in runtime.

For every `PageRenderer`, we override `OnElementChanged` and get the actual Xamarin.Forms page. For iOS, we use the `ViewDidAppear` iOS-specific method, but for Android we don't have any specific lifecycle methods so we write all our authentication code in the `OnElementChanged` method.

We capture if the login is in progress in case we dismiss the modal page and `OnElementChanged` is invoked again to avoid showing again the authentication page.

The `Xamarin.Auth` plugin has an `OAuth2Authenticator` class that we instantiate with the information from our `OAuthSettings` instance that we have in our Xamarin.Forms page `ProviderOAuthSettings` property that we have access in our renderer. The best of both worlds!

Register the completed event that will be invoked in success and in cancel cases and the error event that will notify us if any error is occurred. With the `Auth.GetUI()` method, we get a platform-specific object that we can use to show our authentication view. Last but not least, in the Completed event we close the authentication view that we presented on each platform.

See also

- Chapter 2, *Declare Once, Visualize Everywhere*
- Chapter 4, *Different Cars, Same Engine*
- https://developer.xamarin.com/guides/cross-platform/xamarin-forms/custom-renderer/
- https://components.xamarin.com/view/xamarin.auth

Chapter 2. Declare Once, Visualize Everywhere

In this chapter, we will cover the following recipes:

- Creating a tabbed-page cross-platform application
- Adding UI behaviors and triggers
- Configuring XAML with platform-specific values
- Using custom renderers to change the look and feel of views

Introduction

Xamarin.Forms is a cross-platform UI framework. So far, we have seen examples of creating the UI in code using procedural code, creating objects, wiring them to events, and adding them to collections. That's OK, but when it comes to a more complicated UI structure, this code can get very large, hard to understand, and difficult to visualize what has been created.

Another limitation is that we mix design code and behavior, which can become difficult to separate the roles between developer and designer, and also concerns like colors and fonts.

This is where XAML comes to save us from the pain. It is a markup language originally created by Microsoft to describe the user interface in **Windows Presentation Foundation** (**WPF**). Using a declarative resource-style approach to describe the UI makes it clearer to visualize it and to separate the concerns between design and development.

If you have used Microsoft XAML before, it will be very easy to get up and running. The specifications are exactly the same, although the tags are specific Xamarin.Forms controls.

At the time of writing this book, there is no UI designer option, only XAML text editors. However, XAML language is toolable and extensible, and I'm pretty sure that Xamarin will soon introduce either a designer or an extension, for example for Blend!

In this chapter, we will see examples of how to create pages using XAML, add layout controls, behavior, customize the look and feel of the native rendered controls, and apply platform-specific values. So, let's dive in!

Creating a tabbed-page cross-platform application

In Chapter 1, *One Ring to Rule Them All* recipes we set up our cross-platform shared UI layout using procedural behind code. There's nothing wrong with this, and you might need to continue doing it when creating custom controls. Xamarin.Forms has a big advantage utilizing XAML, which is a declarative language. Using XAML we can easily separate the design and the the behavior to speak a universal markup language between designers and developers while having reusable components, styles, and resources.

If you've done XAML programming with Microsoft before, you will find it exactly the same since the specifications are the same. The difference is mainly the controls and layout containers that you use.

We will take this approach in all the recipes of the remaining chapters. In this section, we will demonstrate how to use XAML and create a four-tabbed-page cross-platform application.

How to do it…

1. Create your cross-platform using Visual Studio or Xamarin Studio.
2. **Add | New Folder** in the core PCL project, and name it `Pages`.
3. **Add | New File**, choose top-section **Forms**, and then select **Forms Content Page Xaml**. Name it `FormsTabPage`.

4. The newly created page is a `ContentPage`, so we need to change the root tag to `TabbedPage`.

```
<?xml version="1.0" encoding="utf-8" ?>
<TabbedPage xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
x:Class="XamlTabbedApp.Pages.FormsTabPage"
</TabbedPage>
```

5. We also need to change the subclass type to a `TabbedPage` in the code-behind file. When we created the XAML page file, a complementary behind-code file is created, `FormsTabPage.xaml.cs`. You can click the arrow next to the XAML file and double-click to open it.

6. Change the class to inherit from `TabbedPage`.

   ```
   public partial class FormsTabPage : TabbedPage
   ```

7. Let's make `FormsTabPage` the main page of the application. Go to the `App.cs` constructor and change the `MainPage` property assignment to create a new `FormsTabPage`.

   ```
   MainPage = new FormsTabPage();
   ```

8. Now, let's create the content for each tab. This will be a XAML page for each tab item. Create four XAML content pages.
9. Repeat step 3 four times and name the pages `Soccer`, `Football`, `Basketball`, `Baseball`.
10. Change the label Text property for each one with some sport quotes. Refer to the related code of the book for the quotes.
11. There are four icons that we will use as resources for the tab items in the iOS platform. You can find them in the source code.
12. Create a folder in the iOS project `Resources` folder, **Add | New Folder**, and name it `TabIcons`.
13. Right-click in the newly created folder, **Add | Add Files…**, and choose the icons from the directory that is located. Choose the option of how you want the resources to link in the folder, in this case copying the files. You will find the tab icons in the book's code.
14. We are ready to add the pages as tab item contents, and this can happen by modifying the `FormsTabPage.xaml` with the following code:

    ```
    <TabbedPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="XamlTabbedApp.Pages.FormsTabPage" xmlns:mypages="clr-
    namespace:XamlTabbedApp.Pages;assembly=XamlTabbedApp">
        <TabbedPage.Children>
          <mypages:SoccerPage Title="Soccer" Icon="TabIcons/soccer.png"/>
          <mypages:FootballPage Title="Football"
    Icon="TabIcons/football.png"/>
          <mypages:BasketballPage Title="Basketball"
    Icon="TabIcons/basketball.png"/>
          <mypages:BaseballPage Title="Baseball"
    Icon="TabIcons/baseball.png"/>
        </TabbedPage.Children>
    </TabbedPage>
    ```

15. Run the application for every platform and witness the magic. You should see something similar to the following screenshots:

iOS bottom tab bar with icons:

iPhone 6s - iPhone 6s / iOS 9.0 (13A340)
Carrier
12:23 PM
Soccer is simple, but it is difficult to play simple.
Johan Cruyff
Soccer
Football
Basketball
Baseball

Android tabs:

Xamarin Android Player - [Nexus 5 (KitKat)]
12:24
SOCCER
FOOTBALL
BASKETBALL
BA
Football is like life - it requires perseverance, self-denial, hard work, sacrifice, dedication and respect for authority.
Vince Lombardi

Windows Phone pivot:

The tabbed page application works and respects the native look and feel. The content pages serve a purpose, but let’s see how to add navigation in a tab page:

1. Go to `FormsTabPage.xaml` and wrap a `NavigationPage` around the `BasketBallPage`.

```
<NavigationPage Title="Basketball" Icon="TabIcons/basketball.png">
  <x:Arguments>
    <mypages:BasketballPage Title="Basketball"
Icon="TabIcons/basketball.png"/>
  </x:Arguments>
```

```
</NavigationPage>
```

2. Add a Button with a name to make it visible in the behind code to `BasketballPage.xaml` and wrap it around a `StackLayout`.

```
<StackLayout Orientation="Vertical" VerticalOptions="Center"
HorizontalOptions="Center">
    <Label Text="There is no such thing as a perfect basketball player,
and I don't believe there is only one greatest player either. Michael
Jordan" VerticalOptions="Center" HorizontalOptions="Center" />
    <Button x:Name="photoButton" Text="Photo"/>
</StackLayout>
```

3. Right-click the **Pages** folder, **Add** | **New File…**, and select **Forms** | **Forms ContentPage Xaml**; name it `PhotoPage`.
4. Add an `Image` control in the `ContentPage.Content` tag. Set the source to a URL; in our case, we point to a Michael Jordan photo.

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
x:Class="XamlTabbedApp.PhotoPage">
    <ContentPage.Content><Image
Source="http://vignette4.wikia.nocookie.net/epicrapbattlesofhistory/ima
ges/3/3c/Michael_Jordan_Based_On.png/revision/latest?cb=20150823041257"
VerticalOptions="Fill"HorizontalOptions="Fill" />
    </ContentPage.Content>
</ContentPage>
```

5. In the behind code `Basketball.xaml.cs` constructor, register an event delegate handler to navigate to the `PhotoPage`.

```
photoButton.Clicked += (sender, e) => Navigation.PushAsync(new
PhotoPage());
```

6. Run the application, go to the **Basketball** tab page, and press the photo button. The result should be similar to the following screenshots:

   iOS:

iPhone 6s - iPhone 6s / iOS 9.0 (13A340)
Carrier
3:09 PM
Basketball
Soccer
Football
Basketball
Baseball

Android:

Xamarin Android Player - [Nexus 5 (KitKat)]
3:29
ER
FOOTBALL
BASKETBALL
BASEBALL

Windows Phone:

How it works…

In this recipe, we covered how you can navigate using the Xamarin.Forms `TabbedPage`. Tab applications is a common pattern in the mobile world that users from any platform are familiar with.

As we see, creating a `TabbedPage` is straightforward. Setting the title of the tab item and the icon that is used only in the iOS platform, makes sense since it is the only platform that uses the concept of tab icons at the bottom tab bar. In Android, you have the tab on top of the page with only the title text. In Windows Phone, the native equivalent control is the pivot that allows only a title text to each item.

Xamarin.Forms respects the native platform user interface guidelines and does its best to keep the look and feel of the platform's nature.

Note

`TabbedPage` is essentially a container of pages. It actually has a property, Children, that you can add pages programmatically, but since we use XAML it's implicit with adding the children inside the page's root tag, `TabbedPage`.

To accomplish referencing our pages in XAML we added a namespace with the key `mypages` and a `xmlns` statement.

Then, for every page, we need to set the Title and Icon properties for each tab item.

We added four tab items in our cross-platform application. If you try to add five or more tabs, it will produce different behavior for each platform.

The iOS design decision is to allow five tab items, if more then the last tab to the right becomes a list page where the user can choose an option.

In Android, the top tab bar becomes scrollable while adding tab items. In Windows Phone, the pivot is just adding more items.

Adding navigation in your tab page is fairly simple using the navigation container. You could add a `NavigationPage` to wrap your `FormsTabPage` and utilize navigation for every page. In our example, we added navigation in only one of our pages to maintain the concept of customization and separation.

In XAML the easy way to do this is wrap the page in a `NavigationPage`. Notice that the page is declared inside the `<x:Arguments>` tag; it is important for the instantiation of the `NavigationPage` in runtime using our page as a constructor parameter.

Tip

It is also mandatory to set the Icon to the `NavigationPage` and the Title to both the `NavigationPage` and the `ContentPage`. There is no exception if it is omitted, but one title is for the tab item and the other for the navigation bar in the iOS platform.

See also

- https://developer.xamarin.com/guides/cross-platform/xamarin-forms/user-interface/xaml-basics/
- https://developer.xamarin.com/guides/cross-platform/xamarin-forms/controls/pages/

Adding UI behaviors and triggers

You already must have come to the common problem of where do we write business logic, how to organize it, and reflect the requirement or give a visual hint to the user.

Validation rules and business flow is everywhere from the data layer to the UI layer, and we follow various practices to accomplish its level of input and data integrity.

This is where behaviors and triggers are introduced. The first attaches independent testable functionality to our UI controls (think about it as an element input validator), while the later is used in XAML to express user interface changes based on monitoring properties, events, data, and a combination of expressions.

In this section, we will take a login page, similar to the *Creating a cross-platform login screen* section of Chapter 1, *One Ring to Rule Them All*, but using XAML, and apply some validation rules.

How to do it…

There are four types of triggers, and we'll demonstrate all in this recipe.

- **Property trigger** – executed when a property on a control is set to a particular value
- **Data trigger** – same as the property trigger but uses data binding
- **Event trigger** – occurs when an event is raised on the control
- **Multi trigger** – allows multiple trigger conditions to be set before an action occurs

First, let's add the login page and a property trigger; this trigger will allow us to define some setters if the property we are monitoring is changed to the value that is required.

1. Create a new cross-platform solution; we'll name it `XamFormsBehaviorTriggers` using your preferred IDE. Let's follow the Visual Studio way for this recipe.
2. Right-click, **Add | New Folder** to the core PCL `XamFormsBehaviorTriggers` project, and name it `Pages`.
3. **Add | New Item**, find and choose the **Forms Xaml Page**, and name it `LoginPage`.
4. Delete the default `Label` element that is automatically created and add the following code:

```
<StackLayout Orientation="Vertical" Spacing="20" Padding="40"
HorizontalOptions="Center" VerticalOptions="Start">
    <Entry x:Name="usernameEntry" Placeholder="Username or email"/>
    <Entry x:Name="passwordEntry" Placeholder="Password"
IsPassword="true"/>
    <Button x:Name="loginButton" Text="Login"/>
</StackLayout>
```

5. Open the `App.cs` cross-platform entry application point and change the `MainPage` assignment to a `LoginPage` instance.

```
MainPage = new LoginPage();
```

6. Run the application and you will get the desired UI for username, password, and a button to proceed with the authentication.
7. Create a `ResourceDictionary` with a style to apply the trigger to all the page elements. This will be applied to every `TargetType` in this page; in our example, we apply the style to every Entry view. Add the following code to the `LoginPage.xaml` after the opening tag `ContentPage` at the top. This will trigger a `Setter` when any Entry element's property `IsFocused` is changed value will turn the `BackGroundColor` to yellow and the `TextColor` to black. The trigger will fire by default every time the property is changed to `true`.

```
<ContentPage.Resources>
  <ResourceDictionary>
    <Style TargetType="Entry">
      <Style.Triggers>
        <Trigger TargetType="Entry" Property="IsFocused" Value="True">
        <Setter Property="BackgroundColor" Value="Yellow" />
        <Setter Property="TextColor" Value="Black" />
        </Trigger>
      </Style.Triggers>
```

```
    </Style>
  </ResourceDictionary>
</ContentPage.Resources>
```

8. Test your Android platform.

Xamarin Android Player - [Nexus 5 (KitKat)]
8:58
yoda
Password
Login
gods
Yoda
God's
q w e r t y u i o p
a s d f g h j k l
z x c v b n m
?123
Done

Let’s add a Data Trigger now, which will disable the button if there is no password entered. This kind of trigger works with Data Binding. If you want to see how Data Binding works, see Chapter 7, *Bind to the Data*.

1. Make the one-line `Button` tag to an open-close tag and add a `DataTrigger`.

```
<Button.Triggers>
  <DataTrigger TargetType="Button" Binding="{Binding Source=
{x:Reference passwordEntry}, Path=Text.Length}" Value="0">
      <Setter Property="IsEnabled" Value="False" />
  </DataTrigger>
</Button.Triggers>
```

2. That’s it. Run the iOS application and see the result logic in action.

iPhone 6 - iPhone 6 / iOS 9.0 (13A340)
Carrier
10:04 PM
Username or email
Password
Login

When the case is taking an action when an element's event is raised, Event Trigger comes to the rescue.

Event triggers are backed by a `TriggerAction<T>` subclass with an abstract method that we have to implement.

1. **Add | New Folder** in the `XamFormsBehaviorTriggers` core PCL library; name it `Triggers`.
2. In the newly created folder, **Add | New Item…** and create a new class with the name `LengthValidationTrigger`.
3. The following code checks the `Entry.Text.Length` property and changes the background color accordingly. Modify the class with the following:

```
public class LengthValidationTrigger : TriggerAction<Entry>
{
    protected override void Invoke(Entry sender)
    {
      bool isValid = sender.Text.Length > 6;
      sender.BackgroundColor = isValid ? Color.Yellow : Color.Red;
    }
}
```

4. Open the `LoginPage.xaml` file.
5. To use our newly created `LengthValidationTrigger`, we need to define a `xmlns namespace` to the root `ContentPage` tag.

```
xmlns:local="clr-
namespace:XamFormsBahaviorTriggers.Triggers;assembly=XamFormsBahaviorTr
iggers"
```

6. Modify the `passwordEntry` element and add inside the enclosing tag the following trigger entry:

```
<Entry.Triggers>
  <EventTrigger Event="TextChanged">
    <local:LengthValidationTrigger />
  </EventTrigger>
</Entry.Triggers>
```

7. Run the Windows Phone platform application and type the first letters in the password entry control.

For multiple condition requirements and business logic, we have the MultiTrigger. Follow the next steps to accomplish a case where both username and password should be filled to enable the **Login** button:

1. Go to the `XamFormsBehaviorTriggers` PCL core library and right-click **Add | New Folder**; name it `Converters`.

2. On the new folder `Converters`, right-click, **Add | New Item…**, and create a new class file with the name `MultiTriggerConverter`. Add the following code:

```
public class MultiTriggerConverter : IValueConverter
{
  public object Convert(object value, Type targetType, object
parameter, CultureInfo culture)
    {
      if ((int)value > 0)
        return true;    // data has been entered
      else
        return false;   // input is empty
    }

  public object ConvertBack(object value, Type targetType, object
parameter, CultureInfo culture)
      {
        throw new NotSupportedException();
      }
}
```

3. Add the `namespace` for the `Converter` folder classes. Creating classes inside a Visual Studio folder adds the folder name to the created class `namespace` with the assembly name.

```
xmlns:converters="clr-
namespace:XamFormsBahaviorTriggers.Converters;assembly=XamFormsBahavior
Triggers"
```

4. Define a `MultiTriggerConverter` in the page's `ResourceDictionary` `LoginPage.xaml` file.

```
<converters:MultiTriggerConverter x:Key="multiTriggerConverter" />
```

5. For our login button in the `LoginPage.xaml` page, add the following trigger in the `Buttons.Triggers` tag:

```
<MultiTrigger TargetType="Button">
  <MultiTrigger.Conditions>
    <BindingCondition Binding="{Binding Source={x:Reference
usernameEntry}, Path=Text.Length,
      Converter={StaticResource multiTriggerConverter}}" Value="true"
/>
    <BindingCondition Binding="{Binding Source={x:Reference
passwordEntry}, Path=Text.Length, Converter={StaticResource
multiTriggerConverter}}" Value="true" />
  </MultiTrigger.Conditions>
<Setter Property="IsEnabled" Value="True" />
</MultiTrigger>
```

6. Run your Android project, or any other platform you might want. See that you need to set at least one letter in both the username and password entry to enable the login button.

Xamarin Android Player - [Nexus 5 (KitKat)]
11:36
obiwankenobi
•••••
Login
q w e r t y u i o p
a s d f g h j k l
z x c v b n m
?123
Done

Nice and easy UI validation and modification for property changes, event notifications, use with data binding, and when a combination of conditions meet the requirements.

Another exciting feature to modify the UI and attach functionality is Behaviors. Behaviors are written in code, a C# class, and attached in XAML. In this example, we will set the background property to red if the e-mail value is not a valid e-mail.

1. In Visual Studio, go to the core PCL `XamFormsBehaviorTriggers` project and **Add | New Folder**; name it `Behaviors`.
2. Right click in the created folder, **Add | New Item…**, create a new class file, and name it `EmailValidatorBehavior`.
3. Make the class a subclass of the `Behavior<Entry>` generic class since we will attach it to an Entry element.
4. Add the following code to the class:

```
const string EmailRegex = @"^(?("")("".+?(?<!\\)""@)|(([0-9a-z]((\.
(?!\.))|[-!#\$%&'\*\+/=\?\^`\{\}\|~\w])*)(?<=[0-9a-z])@))" + @"(?(\[)(\
[(\d{1,3}\.){3}\d{1,3}\])|(([0-9a-z][-\w]*[0-9a-z]*\.)+[a-z0-9][\-a-z0-
9]{0,22}[a-z0-9]))$";

  protected override void OnAttachedTo(Entry bindable)
  {
    bindable.TextChanged += HandleTextChanged;
  }

  void HandleTextChanged(object sender, TextChangedEventArgs e)
    {
      bool isValid = (Regex.IsMatch(e.NewTextValue, EmailRegex,
RegexOptions.IgnoreCase, TimeSpan.FromMilliseconds(250)));
      ((Entry)sender).BackgroundColor = isValid ? Color.Yellow :
Color.Red;
      ((Entry)sender).TextColor = Color.Black;
    }

  protected override void OnDetachingFrom(Entry bindable)
  {
    bindable.TextChanged -= HandleTextChanged;
  }
```

5. At the top in the `ContentPage` root tag, add the namespace to make the Behaviors namespace available to the Page.

```
xmlns:behaviors="clr-namespace:XamFormsBahaviorTriggers.Behaviors;
assembly=XamFormsBahaviorTriggers"
```

6. Attach the `EmailValidatorBehavior` in the XAML `usernameEntry` element.

```
<Entry.Behaviors>
  <behaviors:EmailValidatorBehavior />
</Entry.Behaviors>
```

7. Run the application and watch how the background color of the e-mail Entry is red. Android project with behavior.

Xamarin Android Player - [Nexus 5 (KitKat)]
8:33
Email
Password
Login
we
he
end
q w e r t y u i o p
a s d f g h j k l
z x c v b n m
?123
,
.
Done

8. Once you type a valid e-mail, it will turn to yellow.

Xamarin Android Player - [Nexus 5 (KitKat)]
9:37
gt@x.co
Password
Login
come
co
of
q w e r t y u i o p
a s d f g h j k l
z x c v b n m
?123
Done

How it works…

Triggers are extremely helpful. You use them to take action when a specified condition is met; triggers are declarative; and except property and data Triggers they are backed by a class that inherits from `TriggerAction<T>` when we want to use an event Trigger, and `IValueConverter` when we want to use a multi trigger.

In the case of property trigger, we created a style and in the triggers collection added a trigger to monitor the `IsFocused` property of the element, changed the `BackGroundColor` to yellow and `TextColor` to black with Setters. You can have as many setters as you want and they will all invoked when the property value is changed to the required.

Note

A data trigger is conceptually the same as the property trigger. Instead, we don't specify a property but assign a binding expression. This could be another element's property or a property in a **ViewModel**.

Event triggers hook up to an elements event and invoke the assigned custom `TriggerAction<T>` subclass when it is raised. It is a good place to add UI validation.

Finally, the triggers family has a multi trigger to cover you when your requirements demand more than one condition to be met before the setters are invoked.

They can also be backed by an `IValueConverter` subclass where you can add your additional logic.

Last but not least in this recipe, we attached behavior to an entry element validating if the e-mail is a valid e-mail; this was accomplished using a regular expression and a `Behavior<T>` subclass that requires to override two methods: `OnAttachedTo(T)`, which is fired immediately after the behavior is attached to the view, and `OnDetachingFrom(T)`, which is fired when the behavior is removed from the attached control.

In our case, in the `OnAttached(T)` method we register to the `TextChanged` event of the Entry element that the behavior is attached. We unregister the event handler in `OnDetachingFrom(T)`. The delegate method handler checks if the `Entry.Text` value is a valid e-mail and changes the background color accordingly.

Behaviors is a very exciting and powerful feature. It is possible to attach multiple behaviors in a view and utilizing data binding. With bindable properties, you can add complex validation and show some nice UI hints while the user is providing input.

See also

- https://developer.xamarin.com/guides/cross-platform/xamarin-forms/working-with/triggers/
- https://developer.xamarin.com/guides/cross-platform/xamarin-forms/working-with/behaviors/

Configuring XAML with platform-specific values

Every platform as we know has its own screen metric system, and while Xamarin.Forms does a great job of creating a cross-platform UI and following the platform specifics, we often might need some platform tweaking.

Fortunately, Xamarin.Forms has an out-of-the-box feature that we can use in XAML and in code.

How to do it…

1. In Visual Studio, create a `Xamarin Forms (Portable)` project; give it the name `XamFormsOnPlatform`.
2. Right-click and **Add | New Item…** in the PCL core project. Create a new **Forms Xaml** Page and name it `MainPage`.
3. Open the `MainPage.xaml` file, remove the default label, and add the following in the `ContentPage` tag:

   ```
   <ContentPage.BackgroundColor>
     <OnPlatform x:TypeArguments="Color">
       <OnPlatform.iOS>#FF0000</OnPlatform.iOS>
       <OnPlatform.Android>#00FF00</OnPlatform.Android>
       <OnPlatform.WinPhone>#51C0D4</OnPlatform.WinPhone>
     </OnPlatform>
   </ContentPage.BackgroundColor>
   ```

4. Go to `App.cs` now and change the default assignment of the `MainPage` property in the constructor.

   ```
   MainPage = new MainPage();
   ```

5. Run your application in all platforms and you can witness Xamarin.Forms taking care of your request and change accordingly the `BackgroundColor` of the page for each platform.

Let’s try with a different syntax for a numeric property.

1. After the `ContentPage.BackgroundColor` tag, add the following XAML code. This declares a `BoxView` and sets the `WidthRequest` value for each platform.

   ```
   <BoxView HorizontalOptions="Center" BackgroundColor="Black">
       <BoxView.WidthRequest>
         <OnPlatform x:TypeArguments="x:Double" iOS="30" Android="40"
   WinPhone="50" />
       </BoxView.WidthRequest>
   </BoxView>
   ```

2. Just run the app again and see the results. Magic in action!

   Android:

Xamarin Android Player - [Nexus 5 (KitKat)]
11:09

iOS:

iPhone 6s - iPhone 6s / iOS 9.0 (13A340)
Carrier

Windows Phone:

Using this feature in XAML is incredibly helpful, although sometimes we might want to handle a case in code.

In the next example, we will add a Label and see how to provide platform-specific values in code.

1. Open the `MainPage.xaml` and comment, or remove, the `BoxView` element.
2. Add a Label after the `ContentPage.BackgroundColor` tag.

```
<Label x:Name="label"
  Text="Xamarin Forms OnPlatform"
  HorizontalOptions="Center"
  VerticalOptions="Center"/>
```

3. In the behind-code file `MainPage.xaml.cs`, override the `OnAppearing` method and add the following code to set platform-specific values for the label we just added:

```
protected override void OnAppearing()
{
  base.OnAppearing();

  Device.OnPlatform( iOS: () =>
  {
     label.FontFamily = "HelveticaNeue-Thin";
     label.FontSize = 10;
   },
   Android: () =>
   {
      label.FontFamily = "Arial Black";
      label.FontSize = 20;
   },
   WinPhone: () =>
   {
     label.FontFamily = "Calibri";
     label.FontSize = 30;
   }
  );
}
```

4. That’s it. Run the application and notice that Xamarin.Forms is handling our requirements for each platform.

   iOS:

iPhone 6s - iPhone 6s / iOS 9.0 (13A340)
Carrier
5:29 PM
Xamarin Forms OnPlatform

Android:

Xamarin Android Player - [Nexus 5 (KitKat)]
5:30
Xamarin Forms OnPlatform

Windows Phone:

A very helpful platform-specific customization, which saves a lot of time! No abstractions with interfaces, no implementation classes, and dependency injection from our side. Set

the properties and let the Xamarin.Forms magic happen!

How it works…

`OnPlatform<T>` is a generic static class that has three properties: iOS, Android, and Windows Phone of type T. Depending on which platform we are currently running, Xamarin.Forms implicitly casts to type T and returns the appropriate object.

Its usage is really simple in XAML, by providing the `x:TypeArguments` property to the type that you target and then setting the three properties to the values that you want. Under the covers, Xamarin.Forms will provide the equivalent platform value with its default converter.

In our XAML example, we set the `BackgroundColor` of the `ContentPage` to green for Android, red for iOS, and cyan to Windows Phone. Notice how we included the `OnPlatform` in the `ContentPage.BackgroundColor` tag and specified the type of the property we target setting the `x:TypeArguments` property to color. Xamarin.Forms then handles all the casting in the platform it is running on.

To demonstrate the `OnPlatform` usage with a primitive type and another syntax, we added the `OnPlatform` tag in the `BoxView.WidthRequest` tag setting the `x:TypeArgument` to Double and the three properties to the desired values.

You can also use this feature in code with two static methods in the Device class: `OnPlatform` and `OnPlatform<T>`. In our example, we set the `FontFamily` and the `FontSize` of the label for each platform to a different value.

See also

- https://developer.xamarin.com/guides/cross-platform/xamarin-forms/working-with/platform-specifics/

Using custom renderers to change the look and feel of views

A user interface built with Xamarin.Forms looks and feels native because it is native. Each of the available controls are rendered in the equivalent currently running platform view; for example, an Entry view is a `UITextField` in iOS, an `EditText` in Android, and a `TextBox` in Windows Phone.

The great thing is that you can also create your own, or extend the existing ones. In this section, we will see how you can extend the view and customize the look and feel of the available Entry view.

How to do it…

1. Start by creating a cross-platform project in Xamarin Studio or Visual Studio. For this example, we used Visual Studio because we don't have to do extra work to create and connect the Windows Phone to Xamarin.Forms. Name the project `ViewCustomRenderer`.
2. In the core PCL project, create a class file, **Add | Class…**, name it `CustomEntry`, and make it a subclass of `Entry`.
3. We will need a main page, **Add | New Item…**, and choose to create a **Forms Xaml Page**; name it `LoginPage`.
4. Add the following code. This will just put two `CustomEntry` views and an image with height 1 point to make it look like a separator line. Note that we have to declare the `xmlns:local` namespace to make our `CustomEntry` view available 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:ViewCustomRenderer;assembly=ViewCustomRenderer"
BackgroundColor="#4A90E2" x:Class="ViewCustomRenderer.LoginPage">
    <StackLayout Orientation="Vertical" VerticalOptions="Center">
      <local:CustomEntry Placeholder="Username"
BackgroundColor="#77ABE9" />
      <Image BackgroundColor="White" HeightRequest="1" />
      <local:CustomEntry Placeholder="Password" IsPassword="True"
BackgroundColor="#77ABE9" />
    </StackLayout>
</ContentPage>
```

5. Open the `App.cs` file, and change the instantiation of the `MainPage` property in the constructor with a new `LoginPage` instance.

```
MainPage = new LoginPage();
```

6. Run the application and you should see the default look and feel of the view for each platform.

You must have noticed, for example, in the Android platform under the first `EditText` placeholder a line, then our separator, and again the same for the second `EditText`. In iOS, we get the default rounded edge `UITextField`, and a default `TextBox` in Windows Phone.

Let's try to make it a little bit flatter and provide a cross-platform look and feel with adding custom view renderers.

1. Go to the Android project and right-click, **Add | Class…**, give it the name `CustomEntryRenderer`, and make the newly created class a subclass of `EntryRenderer`.

```
public class CustomEntryRenderer : EntryRenderer
```

2. Implement the `OnElementChanged` method, which is called when the corresponding

Xamarin.Forms control is created. This will give us the opportunity to tweak the view post-initial setup.

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

    if (Control != null)
    {
      Control.SetBackgroundDrawable(null);
    }
}
```

3. A renderer requires an `[assembly]` attribute above the namespace to resolve the view type to the platform-specific renderer.

```
[assembly: ExportRenderer(typeof(CustomEntry),
typeof(CustomEntryRenderer))]
```

4. There is nothing wrong running the application in any platform at this point, Xamarin.Forms will render the default view for any platform that don't have a platform-specific renderer implementation.

   Repeat the steps 1, 2 and 3 for the iOS platform. See the platform implementation following:

```
[assembly: ExportRenderer(typeof(CustomEntry),
typeof(CustomEntryRenderer))]
namespace ViewCustomRenderer.iOS
{
  public class CustomEntryRenderer : EntryRenderer
    {
      protected override void
OnElementChanged(ElementChangedEventArgs<Entry> e)
          {
             base.OnElementChanged(e);

            if (Control != null)
              {
                Control.BorderStyle = UIKit.UITextBorderStyle.None;
              }
          }
    }
}
```

5. And the same for Windows Phone.

```
[assembly: ExportRenderer(typeof(CustomEntry),
typeof(CustomEntryRenderer))]
namespace ViewCustomRenderer.WinPhone
{
    public class CustomEntryRenderer : EntryRenderer
      {
        protected override void
OnElementChanged(ElementChangedEventArgs<Entry> e)
```

```
            {
              base.OnElementChanged(e);
                if (Control != null)
                  {
                    TextBox textBox = Control.Children[0] as TextBox;
                    textBox.BorderBrush = new
SolidColorBrush(Colors.Transparent);
                    textBox.Background = new
SolidColorBrush(Colors.Transparent);
                  }
            }
        }
}
```

6. Run all platforms and notice that the look and feel is more appropriate.

   iOS.

iPhone 6s - iPhone 6s / iOS 9.0 (13A340)
Carrier
3:03 PM
Username
Password

Android:

Xamarin Android Player - [Nexus 5 (KitKat)]
3:02
Username
Password

Windows Phone:

How it works…

Declaring our control once in XAML and customizing the implementation for each platform is really powerful, especially when the `OnPlatform` feature doesn’t cover your needs. In our case, we want to access the actual native control’s properties.

You can create a new control deriving by View or extend an existing one. Our case is straightforward: creating a class that simply derives from Entry.

Lastly, we associated the renderer with the control through `the [assembly: ExportRendererAttribute(Type view, Type renderer)]`. This provides the dependency service of Xamarin.Forms to locate the specific renderer for the currently running platform.

See also

- https://developer.xamarin.com/guides/cross-platform/xamarin-forms/custom-renderer/
- https://developer.xamarin.com/videos/cross-platform/xamarinforms-custom-renderers/

Tip

Downloading the example code

You can download the example code files for this book from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.

2. Hover the mouse pointer on the **SUPPORT** tab at the top.

3. Click on **Code Downloads & Errata**.

4. Enter the name of the book in the **Search** box.

5. Select the book for which you're looking to download the code files.

6. Choose from the drop-down menu where you purchased this book from.

7. Click on **Code Download**.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

Chapter 3. 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!

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.

```
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.

```
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 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.

```
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:

```
[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`:

```
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)
    {
      button.SetTitle(Page.RandomNumber.ToString(),
UIControlState.Normal);
    }
  }

  partial void OnButtonPressed(UIButton sender)
  {
    if (Page != null)
    {
      Page.OnButtonPressed();
    }
  }
}
```

13. Again, add the `ExportRenderer` attribute above the namespace.

```
[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`:

```
<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`:

```
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:

iPhone 6s - iPhone 6s / iOS 9.0 (13A340)
Carrier
3:11 PM
765458164

Android:

Xamarin Android Player - [Nexus 5 (KitKat)]
3:02
352089142

Windows Phone:

How it works…

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 `UpdateButonText` 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.

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

- https://developer.xamarin.com/api/type/Xamarin.Forms.BindableObject/
- https://developer.xamarin.com/api/type/Xamarin.Forms.BindableProperty/
- 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);
     }
   }
   ```

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

   ```
   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 add `UILongPressGestureRecognizer` 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:

```
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`.

Note

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.

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.

```
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.

```
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.

```
[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 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
         android:id="@+id/textureView"
         android:layout_marginTop="-95dp"
         android:layout_width="wrap_content"
         android:layout_height="wrap_content" />
      <ImageView
        android:id="@+id/snapshotView"
```

```
        android:layout_width="240dp"
        android:layout_height="260dp"
        android:layout_gravity="right|top" />
      <Button
        android:id="@+id/takePhotoButton"
        android:layout_width="match_parent"
        android:layout_height="65dp"
        android:layout_marginBottom="15dp"
        android:layout_gravity="center|bottom"
        android:text="Take Photo" />
</FrameLayout>
```

6. Go to the `InAppCameraPageRenderer.cs` class file again and add the following private fields:

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

7. Override the `OnElementChanged` method.

```
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` Android 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.

13:56
command
option
Take Photo

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.

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

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

   ```
   [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:

Automatic 〉 InAppCameraPage.h 〉 No Selection

```
// WARNING
// This file has been generated automatically by Xamarin Studio to
// mirror C# types. Changes in this file made by drag-connecting
// from the UI designer will be synchronized back to C#, but
// more complex manual changes may not transfer correctly.


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


@interface InAppCameraPage : UIViewController {
    UIView *_cameraViewContainer;
    UIImageView *_captureImageView;
    UIView *_overlayView;
    UIButton *_takePhotoButton;
}

@property (nonatomic, retain) IBOutlet UIView *cameraViewContainer;

@property (nonatomic, retain) IBOutlet UIImageView *captureImageView;

@property (nonatomic, retain) IBOutlet UIView *overlayView;

@property (nonatomic, retain) IBOutlet UIButton *takePhotoButton;

- (IBAction)CapturePhotoTouchUpInside:(UIButton *)sender;

@end
```

Action:

Automatic › InAppCameraPage.m › No Selection

```
// WARNING
// This file has been generated automatically by Xamarin Studio to
// mirror C# types. Changes in this file made by drag-connecting
// from the UI designer will be synchronized back to C#, but
// more complex manual changes may not transfer correctly.


#import "InAppCameraPage.h"

@implementation InAppCameraPage

@synthesize cameraViewContainer = _cameraViewContainer;
@synthesize captureImageView = _captureImageView;
@synthesize overlayView = _overlayView;
@synthesize takePhotoButton = _takePhotoButton;

- (IBAction)CapturePhotoTouchUpInside:(UIButton *)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.

```
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.

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

  AuthorizeCameraUseAsync ().ContinueWith ((antecedent) => {
     bool result = antecedent.Result;
     if (result)
     {
       SetupCameraLiveFeed ();
     }
  });
}
```

```
  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.ContinuousAutoExp
osure))
  {
    device.LockForConfiguration(out error);
    device.ExposureMode = AVCaptureExposureMode.ContinuousAutoExposure;
    device.UnlockForConfiguration();
  }
  else if
(device.IsWhiteBalanceModeSupported(AVCaptureWhiteBalanceMode.Continuou
sAutoWhiteBalance))
  {
    device.LockForConfiguration(out error);
    device.WhiteBalanceMode =
AVCaptureWhiteBalanceMode.ContinuousAutoWhiteBalance;
    device.UnlockForConfiguration();
  }
}
```

13. Implement the `CameraPhotoTouchUpInside` method and add an `async` method to capture the photo.

```
partial void CapturePhotoTouchUpInside(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.

```
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:

T-Mobile
13:50
100%
esc
tab
caps lock
shift
fn
control
alt
option
Take Photo

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`.

   ```
   <Image x:Name="previewImage"
     Width="Auto"
     Height="Auto" />
   ```

3. Right-click the project again and **Add | Class…**, name it `InAppCameraPageRenderer`, and click **Add**. Make it a `PageRenderer` subclass.
4. Add `ExportRender` attribute above the namespace declaration of the `InAppCameraPageRenderer.cs` file.

   ```
   [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!

9:07

How it works…

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 `LayoutParameters` 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 `CMSamplBuffer` 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/
- https://developer.xamarin.com/api/type/Android.Views.TextureView+ISurfaceTexture
- https://developer.xamarin.com/recipes/android/other_ux/textureview/display_a_strean
- https://developer.apple.com/library/prerelease/ios/documentation/AudioVideo/Concep

Chapter 4. Different Cars, Same Engine

In this chapter, we will cover the following recipes:

- Sharing code between different platforms
- Using the dependency locator
- Adding a third-party Dependency Injection Container
- Architecture design with **Model-View-ViewModel** (**MVVM**) pattern
- Using the event messenger
- Adding localization

Introduction

In this chapter, we will learn the different approaches, Shared Projects and **Portable Class Libraries** (**PCL**), to share code across all platforms.

`DependencyService` is the out-of-the-box service locator from Xamarin.Forms and we will learn how to use it to resolve implementation classes for our interfaces in runtime.

Using a service locator is a solution to register the implementation classes you need to use for your abstracted interfaces in the application. In this chapter, we will explore Xamarin.Forms' built-in `DependencyService` to map interfaces to implementation classes.

Moving forward, we will explore **aspect oriented programming** (**AOP**), which is accomplished using Dependency Injection. With Dependency Injection, the class is given its dependencies automatically via a constructor with all required parameters. The Dependency Injection Container is responsible for resolving the dependencies needed; we will use Ninject to see how we can apply Dependency Injection in our solution.

Many times, you have two components in your application where none is referencing the other, but still want to receive notifications of an action. You can accomplish this with a global event messenger. There are two types of event messaging: instance events and global events. Using the Xamarin.Forms event messenger, we will learn how to publish global events and subscribe clients that will listen for these notifications.

Mobile applications may target an international audience, meaning multiple languages will have to be supported. Xamarin.Forms supports localization, and the last section demonstrates how you can accomplish this task.

Sharing code between different platforms

Code sharing is one of the key concepts when developing applications with Xamarin. In this section, we will learn how to use the available options of sharing code between platforms when creating a Xamarin solution, Shared Project and Portable Class Library, and what the key differences are.

How to do it…

We will start with creating a solution based in Shared Project.

1. Start Visual Studio. In the top menu, select **File | New | Project…** and in the **Templates | Mobile Apps** tab choose **Blank App** (Xamarin.Forms Shared). Name it `XamFormsSharedProject`, choose your directory path, and click **OK**.
2. With that, you get four projects: the `XamoFormsSharedProject` core shared project and the corresponding Android, iOS, and Windows Phone platform projects.
3. Create a new class in the core `XamFormsSharedProject` project, right-click, **Add | Class…**, and give it the name `NameService.cs`. Make it public.

   ### Note

   An advantage of Shared Project is that the code is actually compiled with the platform project that is referenced and you can use conditional directives to use platform APIs from the core project.

4. In the `NameService` class, above the namespace add the following code to import using statements per platform:

```
#if WINDOWS_PHONE

using Windows.Storage;

#elif __IOS__

using Foundation;

#elif __ANDROID__

using Android.Content.Res;
using Android.Content;
using Xamarin.Forms;

#endif
```

5. Copy the following code of the `NameService` class. What we are essentially doing here is using each platform API to read an asset `.txt` file and adding the content to the greeting return value.

```
public class NameService
{
  public async Task<string> GetGreeting(string firstName, string
lastName)
  {
    string fullName = string.Format("{0} {1}", firstName, lastName);
    string content = string.Empty;

    #if WINDOWS_PHONE

    string platformAssetFilePath = @"Assets\PlatformAsset.txt";
    StorageFolder InstallationFolder =
```

```
Windows.ApplicationModel.Package.Current.InstalledLocation;
    StorageFile file = await
InstallationFolder.GetFileAsync(platformAssetFilePath);
    using (StreamReader stream = new StreamReader(await
file.OpenStreamForReadAsync()))
    {
      content = await stream.ReadToEndAsync();
    }

    #elif __ANDROID__

    AssetManager assets = Forms.Context.Resources.Assets;
    using (StreamReader stream = new StreamReader (assets.Open
("PlatformAsset.txt")))
    {
      content = await stream.ReadToEndAsync();
    }

    #elif __IOS__

    string path = NSBundle.MainBundle.PathForResource ("PlatformAsset",
"txt");
    using (StreamReader stream = new StreamReader (path))
    {
      content = await stream.ReadToEndAsync();
    }

    #endif
    fullName = string.Format("Hello {0} from {1}", fullName, content);
    return fullName;
  }
}
```

6. Open the `App.cs` file and change the constructor code with the following. Change first name and last name `GetGreeting` method parameters to your name.

```
public App ()
{
  // The root page of your application

  Label label = new Label
  {
    XAlign = TextAlignment.Center
  };

  NameService nameService = new NameService();

  nameService.GetGreeting("Georgios", "Taskos")
  .ContinueWith((antecedent) =>
  {
    label.Text = antecedent.Result;
  }, TaskScheduler.FromCurrentSynchronizationContext());

  MainPage = new ContentPage {
    Content = new StackLayout {
      VerticalOptions = LayoutOptions.Center,
```

```
      Children = {
        label
      }
    }
  };
}
```

7. We need to add the txt file in each platform. Start with the Android platform, right-click, **Add | New Item…**, and choose text file. Name it `PlatformAsset.txt` and click **Add**.
8. In the newly created text file, add the content Android!
9. Deploy the Android application in your favorite emulator; you should get the following result with your name:

8:20
Hello Georgios Taskos from Android!

10. In the `XamFormsSharedProject.iOS`, in the `Resources` folder, right-click, **Add | New Item…**, and choose text File. Name it `PlatformAsset.txt` and as content put iOS!
11. That's all you need. Run iOS in the simulator to get the result like in the following screenshot:

Carrier
8:21 PM
Hello Georgios Taskos from iOS!

12. Select `XamFormsSharedProject.WinPhone`, the `Asset` folder, right-click, **Add | New Item…**, and choose text File. The name is the same, `PlatformAsset.txt`, and for content add Windows Phone! Press *F4* with the newly created file selected and in the **Properties** window set **Build Action** to **Content**. Build Action indicates what Visual Studio does with a file when a build is executed; it can have several values. Setting Build Action to Content means that the text file will be copied to the target machine; we actually deploy the file with our application.
13. Run the application in the Windows Phone emulator.

The second approach is using a PCL as our core library to host the shared code.

1. Back to Visual Studio and **File | New | Project…**, but this time from **Templates | Mobile Apps** choose **Blank App** (Xamarin.Forms Portable), name it `XamFormsPortable`, and click **OK**.
2. Again, we get a core project, but you might already have noticed the difference

visually from our previous example with a Shared Project; this is an actual assembly, but more on how it works soon.

Note

When using PCL as a core code, we can't have every platform API available; it is limited in the PCL profile we choose, meaning that we will need to create abstractions and inject platform implementation. We will introduce `DependencyService` here, but for now just know that it is used to assign an implementation file for an Interface type; more on the subject in the next recipe, *Using the dependency locator*.

3. On the PCL, right-click, **Add | New Item…**, and choose **Interface** this time. Give it the name `INameService` and click **Add**. For this interface, we will have only one method signature, similar to our Shared Project example.

```
public interface INameService
{
  Task<string> GetGreeting(string firstName, string lastName);
}
```

4. Open the `App.cs` file and replace the constructor code with the following:

```
public App()
{
  // The root page of your application
  INameService nameService = DependencyService.Get<INameService>();

  Label label = new Label
  {
    XAlign = TextAlignment.Center
  };

  nameService.GetGreeting("Georgios",
"Taskos").ContinueWith((antecedent) =>
  {
    label.Text = antecedent.Result;
  }, TaskScheduler.FromCurrentSynchronizationContext());

  MainPage = new ContentPage
  {
    Content = new StackLayout
    {
      VerticalOptions = LayoutOptions.Center,
      Children = {
        label
      }
    }
  };
}
```

5. Same as before, we will add the `PlatformAsset.txt` file for each platform. Repeat steps 7, 11, and 13 from the Shared Project section earlier to create the asset file.
6. Now we can create the implementation classes that implement the `INameService`

interface and instruct `DependencyService` to inject the implementation for this interface in runtime. Start with the Android platform: right-click and choose **Add | Class…**, name it **NameService**, and insert the following code:

```
public class NameService : INameService
{
  public async Task<string> GetGreeting(string firstName, string
lastName)
  {
    string fullName = string.Format("{0} {1}", firstName, lastName);
    string content = string.Empty;
    AssetManager assets = Forms.Context.Resources.Assets;
    using (StreamReader stream = new
StreamReader(assets.Open("PlatformAsset.txt")))
    {
      content = await stream.ReadToEndAsync();
      fullName = string.Format("Hello {0} from {1}", fullName,
content);
      return fullName;
    }
  }
}
```

7. We need to add the following attribute above the `XamFormsPortable.Droid` namespace to instruct `DependencyService` about our implementation class:

```
 [assembly:
Xamarin.Forms.Dependency(typeof(XamFormsPortable.Droid.NameService))]
```

8. Repeat step 6 for the `XamFormsPortable.iOS` project. See the following equivalent implementation and Dependency attribute needed:

```
 [assembly:
Xamarin.Forms.Dependency(typeof(XamFormsPortable.iOS.NameService))]

  namespace XamFormsPortable.iOS
  {
    public class NameService : INameService
    {
      public async Task<string> GetGreeting(string firstName, string
lastName)
      {
        string fullName = string.Format("{0} {1}",firstName, lastName);
        string content = string.Empty;

        string path =
NSBundle.MainBundle.PathForResource("PlatformAsset", "txt");
        using (StreamReader stream = new StreamReader(path))
        {
          content = await stream.ReadToEndAsync();
        }
        fullName = string.Format("Hello {0} from {1}", fullName,
content);
        return fullName;
      }
```

```
    }
  }
```

9. And for Windows Phone:

```
[assembly:
Xamarin.Forms.Dependency(typeof(XamFormsPortable.WinPhone.NameService))
]

  namespace XamFormsPortable.WinPhone
  {
    public class NameService : INameService
    {
      public async Task<string> GetGreeting(string firstName, string
lastName)
      {
        string fullName = string.Format("{0} {1}", firstName,
lastName);
        string content = string.Empty;
        string platformAssetFilePath = @"Assets\PlatformAsset.txt";
        StorageFolder InstallationFolder =
Windows.ApplicationModel.Package.Current.InstalledLocation;
        StorageFile file = await
InstallationFolder.GetFileAsync(platformAssetFilePath);
        using (StreamReader stream = new StreamReader(await
file.OpenStreamForReadAsync()))
        {
          content = await stream.ReadToEndAsync();
        }
        fullName = string.Format("Hello {0} from {1}", fullName,
content);
        return fullName;
      }
    }
}
```

10. Running the projects should give you the exact same result with our previous Shared Project screenshots.

How it works…

We saw how to use two main cross-platform options of sharing code: Shared Project and PCL. There is another option called file-linking, but this is omitted since using it makes testing painful and refactoring a nightmare, I find file-linking useful when the same content file might be shared between all platforms.

Shared Project is a project that shares the files with the target project and is actually compiled in the binary. It requires Visual Studio 2013 update 2. It copies the source files and assets in the target project. It is defined by a new project type, `.shproj`. You can define the build type of the included files but no assembly DLL file is generated.

In our example, we used the conditional compilation strategy, which is an easy way to manage parts of the code that utilizes platform-specific APIs, and we want to make it available only when compiling to a specific platform.

There are five main symbol definitions you might want to use:

- `__MOBILE__`
- `__IOS__`
- `__ANDROID__`
- `WINDOWS_PHONE`
- `SILVERLIGHT`

For the Android platform, you can also define the API level. For example, `__ANDROID_11__` defines code that should only be included if Android 3.0 Honeycomb or the newer API level is supported.

Windows Phone 8.1 also defines `WINDOWS_PHONE_APP`, and for Universal Windows Apps, `WINDOWS_APP`.

Other than conditional directives, you can use class mirroring, where you use a class in your Shared Project but define the implementation in the platform-specific projects and also use partial classes and methods.

These are all valid ways to go, but it can easily become ugly and a maintainability nightmare. PCL comes to the rescue.

PCL are assemblies that are produced, and you can reference them to target projects supporting PCL libraries or distribute to other developers as well. They are tied to a set of classes depending on the target frameworks you want to support. For more information, visit the following link: https://msdn.microsoft.com/en-us/library/gg597391(v=vs.110).aspx.

Using a PCL as a core project means that any platform-specific API is missing. To make this functionality available, our code must use a loosely coupling architectural pattern defining interfaces in the core project and injecting implementation classes conforming to the core interface from our platform projects using the Inversion of Control principle. This means that an object should not know how to construct its dependencies.

In our example, we created an interface with a method signature that accepts two string

arguments and returns a string. Using the `DependencyService` service locator, we retrieved our implementation class registered in our platform-specific projects.

What kind of code can you share? Any code that you write targeting the Base Class Library .NET Framework, like POCO classes, Model classes, repositories that connect to a web service or access a local database, like SQLite, and whatever calculation, processing, and business logic are candidates for code sharing.

Any other API might be available depending on the PCL profile you choose.

See also

- *Using the dependency locator* recipe from this chapter
- *Adding a third-party Dependency Injection Container* recipe from this chapter

Using the dependency locator

When we design a loosely coupled architecture, which nowadays is a standard practice to separate concerns between our layers, we program against interfaces and not implementations.

The problem we're solving with abstractions is tight coupling, which is when implementation layers are reference one each other directly; for example, the view has a reference of the presentation layer, which in turn has a repository reference that has a service reference class, meaning that the view is coupled to the service layer at the bottom.

To achieve loose coupling, we somehow need to tell the project in runtime that for the interface we are programming against to create this implementation class.

There are three ways to achieve this: using a DI container, Dependency Injection, or with a Service Locator, where you can map an implementation to an interface.

Xamarin.Forms offers an out-of-the-box dependency resolver, `DependencyService`, that you can use to register dependencies and in runtime ask for the interface mapped to an implementation class.

How to do it…

1. In Visual Studio, create a new **Blank App** (Xamarin Forms Portable) solution, **File | New Project…**, and name it `XamFormsDependencyService`.
2. Right-click the `XamFormsDependencyService` (Portable) core project, **Add | New Item…**, choose **Interface** from the templates, give it the name `IPlatform`, and make it public.
3. Add a method signature: it will be very simple, accepting no arguments, and returning a string value with the description of the platform we're running on.

```
public interface IPlatform
{
  string GetPlatformDescription();
}
```

4. Programming against the interface introduces the need to resolve a dependency, in our case the `IPlatform` interface, in runtime with an implementation. Open the `App.cs` file and refactor the constructor like the following:

```
public App()
{
  // The root page of your application

  IPlatform platform = DependencyService.Get<IPlatform>();

  Label label = new Label
  {
    XAlign = TextAlignment.Center,
    Text = platform.GetPlatformDescription()
  };

  MainPage = new ContentPage
  {
    Content = new StackLayout
      {
        VerticalOptions = LayoutOptions.Center,
        Children = {
          label
        }
      }
  };
}
```

5. Create the Android implementation class. Right-click the Android project, **Add | Class…**, name it `PlatformAndroid` and click **Add**, and conform to the `IPlatform` interface.

```
public class PlatformAndroid : IPlatform
{
  public string GetPlatformDescription()
  {
    return "Android";
  }
```

```
}
```

6. For every native platform implementation, we have to register the class with `DependencyService`. Above the namespace declaration, add the `Xamarin.Forms.Dependency` attribute.

```
[assembly:
Xamarin.Forms.Dependency(typeof(XamFormsDependencyService.Droid.PlatformAndroid))]
```

7. Repeat steps 5 and 6 for the iOS and Windows Phone platforms. The following is the iOS code:

```
[assembly:
Xamarin.Forms.Dependency(typeof(XamFormsDependencyService.iOS.PlatformiOS))]

  public class PlatformiOS : IPlatform
  {
    public string GetPlatformDescription()
    {
      return "iOS";
    }
  }
```

For Windows Phone:

```
[assembly:
Xamarin.Forms.Dependency(typeof(XamFormsDependencyService.WinPhone.PlatformWinPhone))]

  public class PlatformWinPhone : IPlatform
  {
    public string GetPlatformDescription()
    {
      return "Windows Phone";
    }
  }
```

The result is the description of each platform in the middle of the screen.

Android:

6:39
Android

iOS:

Carrier
6:43 PM
iOS

Windows Phone:

How it works…

`DependencyService` resolves interfaces to platform-specific implementations. Under the hood is a service locator, which uses a container that maps abstractions (interfaces) to concrete registered types, then the client uses the locator to acquire these dependencies.

In this recipe, we created a simple interface (`IPlatform`) that returns a string, in our case the platform description. For each platform we created, a class implements the `IPlatform` interface and returns the name of the platform.

The key is to register the corresponding implementation to `DependencyService`. For this, Xamarin has provided us the dependency attribute that we use to decorate our class above the namespace and then from our core project we resolve the dependency implementation using the method `DependencyService.Get<T>`, which returns the corresponding implementation for the interface.

Note

Note that the interface must be implemented in all platforms and the class must have a parameterless constructor or `DependencyService` will not be able to create an instance for you.

See also

- https://developer.xamarin.com/guides/cross-platform/xamarin-forms/dependency-service/

Adding a third-party Dependency Injection Container

Dependency Injection is one of the solutions when it comes to resolving dependencies in runtime. Let's look at the Wikipedia description:

Note

Dependency Injection is a software design pattern that implements **Inversion of Control** (**IoC**) for resolving dependencies.

Working with a Dependency Injection Container, you will see that it isn't only an IoC container but a set of software design patterns and principles that solves the problem of tight coupling.

Loosely coupled code enables us to write extensible code; testing becomes much easier with mocking the Subject Under Testing component dependencies; dynamic construction in runtime with injected dependencies; and of course maintainability, code that is easy to navigate in different modules, fix bugs, and add new features.

There are different patterns to inject dependencies to modules:

- Constructor Injection
- Property Injection
- Method Injection
- Service Locator
- Others

Which one you choose depends on your architecture and the team's approach to solving the problem.

In this recipe, we will use the Ninject DI Container, Constructor Injection, and the `XLabs.IoC` Service Locator. We will program against two interfaces: `IDataService` and `ISettingsRepository`, `IDataService` will depend on `ISettingsRepository` to retrieve the username key value from a dictionary appending the platform-specific description we are running.

How to do it…

1. Start with creating a new **Blank App** (Xamarin.Forms Portable) solution in Visual Studio, set the name to `XamFormsDependencyInjection`, and click **OK**.
2. Right-click the **Portable Class Library**, `XamFormsDependencyInjection` project, and choose **Manage NuGet Packages**.
3. Search and install the following packages:
   - `Portable.Ninject`
   - `XLabs.IoC.Ninject` (will install the `XLabs.IoC` NuGet package dependency)
4. Right-click and choose **Add | New Item…**, choose **Forms Xaml Page**, give it the name `MainPage.xaml`, and click **Add**.
5. Replace the default label view with the following code:

   ```
   <Label x:Name="label" Text="Hello Xamarin Forms!"
   VerticalOptions="Center" HorizontalOptions="Center" />
   ```

6. Open `App.cs` file and in the constructor, change the `MainPage` property assignment to the following:

   ```
   MainPage = new MainPage();
   ```

7. Time to create the interfaces. Right-click the PCL project and **Add | New Item…**, choose **Interface**, provide the name `IDataService`, and click **Add**. Make the interface public.
8. The interface will have an `ISettingsRepository` property signature that we will create in the next step.

   ```
   ISettingsRepository SettingsRepository { get; set; }
   ```

9. Repeat step 7 to create the `ISettingsRepository` interface and add the method signatures like the following:

   ```
   void Save(string key, string value);
   string GetValue(string key);
   ```

10. Right-click, **Add | Class…**, name it `DataService`, conform to the `IDataService` interface, and add a default constructor accepting an `ISettingsRepository` parameter.

    ```
    public class DataService : IDataService
    {
      public ISettingsRepository SettingsRepository { get; set; }

      public DataService(ISettingsRepository settingsRepository)
      {
        SettingsRepository = settingsRepository;
      }
    }
    ```

11. Go to the `MainPage.xaml.cs` behind code and override the `OnAppearing` method.
12. In the `OnAppearing` method, we will use `Xlabs.Ioc.Resolver` to resolve our

`IDataService` dependency and invoke the `ISettingsRepository Save` method to store a key/value in memory and `GetValue` to retrieve the key's value assigning it to the `MainPage.Label.Text` property.

```
protected override void OnAppearing()
{
  base.OnAppearing();

  IDataService dataService = Resolver.Resolve<IDataService>();
  dataService.SettingsRepository.Save ("Username", "George");
  label.Text = dataService.SettingsRepository.GetValue("Username");
}
```

That's all we need to be able to code against our interfaces in our shared code.

Let's configure the platform-specific implementations and the Ninject DI Container starting with the Android platform.

1. Right-click the `XamFormsDependencyInjection.Droid` project and create a new class, **Add | Class…**, name it **SettingsDroidRepository.cs**, and click **Add**.
2. The new class will implement `ISettingsRepository`. The implementation classes will be the same for all platforms with the difference of hardcoding the platform description, and the naming (not mandatory).

```
public class SettingsDroidRepository : ISettingsRepository
{
  Dictionary<string, string> _containerDictionary = new
Dictionary<string, string>();

  public void Save(string key, string value)
  {
    if (!_containerDictionary.ContainsKey(key))
    {
      _containerDictionary.Add(key, value);
    }
  }

  public string GetValue(string key)
  {
    string value = string.Empty;
    if (_containerDictionary.TryGetValue(key, out value))
    {
      value = string.Format("{0} - Droid", value);
    }

  return value;
  }
}
```

Note

Help from the `XLabs.IoC` container is needed. It is another service locator that will encapsulate our Ninject DI dependency resolver and enable us to access the Ninject

IoC from our PCL library. This might not be the best practice; for example, when you introduce the MVVM architectural pattern in your applications, you want your `ViewModels` to be injected in your views automatically and utilize data-binding instead of using service location. To learn how to do this, refer to this chapter recipe, *Architecture design with Model-View-ViewModel (MVVM)* pattern and Chapter 7, *Bind to the Data*.

3. Set up the Resolver, the `NinjectContainer`, and register our dependencies. Go to `MainActivity.cs` and create the following `SetupDependencyContainer` method:

```
private void SetupDependencyContainer()
{
  StandardKernel standardKernel = new StandardKernel();
  NinjectContainer resolverContainer = new
NinjectContainer(standardKernel);

  standardKernel.Bind<ISettingsRepository>
().To<SettingsDroidRepository>().InSingletonScope();
  standardKernel.Bind<IDataService>().To<DataService>
().InSingletonScope();

  Resolver.SetResolver(resolverContainer.GetResolver());
}
```

4. Call the method after `Forms.Init` in the `OnCreate` method of the `MainActivity` class.

```
global::Xamarin.Forms.Forms.Init(this, bundle);

SetupDependencyContainer();

LoadApplication(new App());
```

For the platforms iOS and Windows Phone, the code is repetitive. In step 1, name the repository implementations; in this book's source code, we created `SettingsTouchRepository` and `SettingsPhoneRepository` respectively. In step 2, change the hardcoded description from Droid to iOS and WinPhone. In step 3, set up the dependencies in `AppDelegate.cs` and `MainPage.xaml.cs`. For details, please refer to the sample code of this book.

Run the application for each platform and you should get results similar to the following screenshots:

Android:

4:36
George - Droid

iOS:

Carrier
4:29 PM
George - iOS

Windows Phone:

How it works…

A Dependency Injection with an IoC is very similar to a service locator, but using a DI we don't need the client to constantly ask for the dependencies; the container creates all the objects needed and injects them to the component that requires them through the constructor. This happens automatically in runtime when a component is constructed via the container.

As we saw in the recipe, we have a `DataService` class that implements the `IDataService` interface and has a dependency of an `ISettingsRepository` object. Setting up the DI container means registering your dependencies. We mapped for each platform an interface to a concrete implementation class that we want the container to instantiate every time it is needed. Using Ninject, we bind the `IDataService` interface to the platform-specific implementation class; we do the same for the `ISettingsRepository` dependency in our `DataService` class. Both are defined as Singleton, a pattern that will create the object once and reuse it through the application lifecycle. That way, the container knows every time that an `IDataService` or `ISettingsService` is needed and it will construct it, injecting or returning it to the client.

To separate the concerns and since we need to use Ninject to resolve dependencies, we will have to introduce another Service Locator; this is resolved from Xamarin with the `XLabs.IoC.Ninject Nuget` package. Using this plugin, we can set the resolver to our Ninject container and request a dependency from our portable class library. Remember, the Ninject DI container has to be used from the platform-specific projects to register the implementations.

With this in place and setting the resolver to the `NinjectContainer`, we can request `IDataService`. When Ninject tries to create a `DataService` class, since this is what we registered for this interface in our setup, it finds a constructor accepting an `ISettingsRepository` argument, checks its internal dictionary if such an implementation class is registered, and in turn creates one and passes it to the constructor. Magic!

See also

- http://www.ninject.org/
- https://github.com/XLabs/Xamarin-Forms-Labs
- https://github.com/XLabs/Xamarin-Forms-Labs/wiki/IOC
- *Architecture design with Model-View-ViewModel (MVVM) pattern* recipe in this chapter

Architecture design with Model-View-ViewModel (MVVM) pattern

The key aspect of using Xamarin is sharing code, classic approach or Xamarin.Forms you'll have to put all the business logic, navigation, dialogs, and data representation in a place that you can reuse and share. This problem is solved using the MVVM pattern and Xamarin.Forms is a perfect fit for it.

The MVVM architectural pattern is a variation of the **Model-View-Controller** (**MVC**) pattern and both are presentation patterns. Although MVVM can be achieved without data binding, the real power is when the platform supports it; the code is much simplified when we are using MVVM and data binding, hence it is natural to use MVVM with XAML-based applications like Xamarin.Forms, Windows Store, Windows Phone, and WPF.

The goal using MVVM is a clean separation of the different modules in our code. A major achievement is the abstraction between designer and developer.

> *Ensure that any code that manipulates presentation only manipulates presentation, pushing all domain and data source logic into clearly separated areas of the program.*
>
> *—Martin Fowler*

This recipe demonstrates how to create a solution using the MVVM pattern and data binding. For more details on data binding, go to Chapter 7, *Bind to the Data*.

How to do it…

1. Start Visual Studio and create a new **Blank App** (Xamarin.Forms Portable) project. Provide the name `XamFormsMVVM`.
2. Every project in the solution has NuGet dependencies, the `XLabs.Forms` package and its dependencies, right-click the solution, choose **Manage NuGet Packages…**, and search for the package and install it.
3. In the `XamFormsMVVM` portable class library, create four folders: `Data`, `Models`, `ViewModels`, and `Views`. To create a folder, right-click the project and select **Add | New Folder**.
4. In the `Models` folder, right-click, **Add | Class…**, and name it `Contact.cs`. This will be the model, theoretically constructed with values from a database or a web service. It is a simple POCO class with three properties.

   ```
   public class Contact
   {
     public string FirstName { get; set; }
     public string LastName { get; set; }
     public string Profession { get; set; }
   }
   ```

5. In the `Data` folder, right-click, **Add | Class…**, and give it the name `DataService.cs`. This is a static class to mock some data; in our case, we will return two instances of contact type.

   ```
   public static class DataService
   {
     public static IEnumerable<Contact> GetAll()
     {
       return new List<Contact>
       {
         new Contact()
         {
           FirstName = "Chris",
           LastName = "Smith",
           Profession = "Computer Scientist"
         },
         new Contact()
         {
           FirstName = "Georgios",
           LastName = "Taskos",
           Profession = "Software Engineer"
         }
       };
     }
   }
   ```

6. Right-click the `ViewModels` folder and **Add | Class…**; name it `ContactViewModel`. Repeat the same for `ContactListViewModel` and `ContactDetailsViewModel`. All `ViewModels` will inherit from `XLabs.Forms.Mvvm.ViewModel`.

   ```
   public class ContactViewModel : XLabs.Forms.Mvvm.ViewModel
   ```

```
{
  private readonly Contact _contact;

  public string Profession
  {
    get { return _contact.Profession; }
    set
    {
      if (_contact.Profession != value)
      {
        _contact.Profession = value;
        NotifyPropertyChanged();
      }
    }
  }

  public string FullName
  {
    get { return string.Format("{0} {1}", _contact.FirstName,
_contact.LastName); }
  }

  public ContactViewModel(Contact contact)
  {
    _contact = contact;
  }
}

public class ContactListViewModel : XLabs.Forms.Mvvm.ViewModel
  {
    private IList<ContactViewModel> _contacts;
    public IList<ContactViewModel> Contacts
    {
      get
      {
        return _contacts;
      }
      set
      {
        _contacts = value;
        NotifyPropertyChanged();
      }
    }

  public ICommand ItemSelectedCommand { get; set; }

  private void OnItemSelected(ContactViewModel contactVM)
  {
    Navigation.PushAsync<ContactDetailsViewModel>((viewmodel, page) =>
    {
      viewmodel.Contact = contactVM;
    });

  }

  public ContactListViewModel()
```

```
  {
    ItemSelectedCommand = new Command<ContactViewModel>
(OnItemSelected);
    Contacts = new ObservableCollection<ContactViewModel>(
    DataService.GetAll().Select(p => new ContactViewModel(p)));
  }
}

public class ContactDetailsViewModel : XLabs.Forms.Mvvm.ViewModel
  {
    private ContactViewModel _contact;
    public ContactViewModel Contact
    {
      get { return _contact; }
      set
      {
        _contact = value;
        NotifyPropertyChanged();
      }
    }
  }
```

7. The last piece of our MVVM parts is the views. In the `Views` folder, right-click and **Add | New Item…**, choose **Forms Xaml Page**, and create `ContactDetailsPage`. Repeat this for `ContactListPage`. For the contents, copy the following XAML code in each page:

```
<!-- ContactDetailsPage -->
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml" x:Class="
XamFormsMVVM.Views.ContactDetailPage" Title="Contact Details">
    <Label Text="{Binding Contact.FullName}" VerticalOptions="Center"
HorizontalOptions="Center" />
</ContentPage>


<!-- ContactListPage -->
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
x:Class="XamFormsMVVM.Views.ContactListPage" Title="Contacts">

    <ListView ItemsSource="{Binding Contacts}"
ItemSelected="HandleItemSelected">
      <ListView.ItemTemplate>
        <DataTemplate>
          <TextCell Text="{Binding FullName}"
           Detail="{Binding Profession}">
          </TextCell>
        </DataTemplate>
      </ListView.ItemTemplate>
    </ListView>

</ContentPage>
```

8. Open the `ContactListPage.xaml.cs` behind-code file and add the following code.

Why we need this is explained in the *How it works…* section.

```
public const string ItemSelectedCommandPropertyName =
"ItemSelectedCommand";
  public static BindableProperty ItemSelectedCommandProperty =
BindableProperty.Create(
  propertyName: "ItemSelectedCommand",
  returnType: typeof(ICommand),
  declaringType: typeof(ContactListPage),
  defaultValue: null);

  public ICommand ItemSelectedCommand
  {
    get { return (ICommand)GetValue(ItemSelectedCommandProperty); }
    set { SetValue(ItemSelectedCommandProperty, value); }
  }

  protected override void OnBindingContextChanged()
  {
    base.OnBindingContextChanged();

    RemoveBinding(ItemSelectedCommandProperty);
    SetBinding(ItemSelectedCommandProperty, new
Binding(ItemSelectedCommandPropertyName));
  }

  private void HandleItemSelected(object sender,
SelectedItemChangedEventArgs e)
  {
    if (e.SelectedItem == null)
    {
      return;
    }

    var command = ItemSelectedCommand;
    if (command != null && command.CanExecute(e.SelectedItem))
    {
      command.Execute(e.SelectedItem);
    }
  }
```

Note

An important step is to connect all these pieces together. Model and `ViewModel` is the easy part. `ContactViewModel` depends on a Contact, so we make sure whenever you create a `ContactViewModel` instance, you pass a Contact instance to the default constructor. To connect `ViewModel` to the corresponding View needs a little bit of architecture. There are some ways to achieve this, and none is wrong, but we should always go after the cleanest way possible. You can use a Service Locator to register your `ViewModels` and create a static `ViewModelLocator` class to access them, or just create a `ViewModel` in the View behind code and assign it to the `BindingContext`. Wouldn't it be great to just register your `ViewModel` to the View and magically have it assigned? Fear not! XLabs to the rescue! *How it works…*soon!

9. Go to `App.cs`, change the constructor, and add a `RegisterViews` method like the following. This will provide us with the desired witchcraft.

```
public App()
{
  // The root page of your application
  RegisterViews();
  MainPage = new
NavigationPage((ContentPage)ViewFactory.CreatePage<ContactListViewModel
, ContactListPage>());
}

private void RegisterViews()
{
  ViewFactory.Register<ContactListPage, ContactListViewModel>();
  ViewFactory.Register<ContactDetailPage, ContactDetailsViewModel>();
}
```

10. For the platform-specific projects, the code is the same. We just need to register the IoC container needed by `XLabs.Forms.Mvvm`. Go to the Android project and open the `MainActivity.cs`. In the `OnCreate` method and after the `Forms.Init` method call, insert the following code:

```
if (!Resolver.IsSet)
{
  Resolver.SetResolver(new SimpleContainer().GetResolver());
}
```

11. Go to the `XamFormsMVVM.iOS` project, open `AppDelegate.cs`, and in the `FinishedLaunching` method, after the `Forms.Init` call add the same line of code in step 10.
12. In `XamFormsMVVM.WinPhone`, open the `MainPage.xaml.cs` file and in the constructor after the `Forms.Init` call, add the line of code in step 10.
13. Run the application for all platforms and admire your MVVM architecture live!

Android:

6:52
Contacts
Chris Smith
Computer Scientist
Georgios Taskos
Software Engineer

iOS:

Carrier
6:54 PM
Contacts
Chris Smith
Computer Scientist
Georgios Taskos
Software Engineer

Windows Phone:

Tapping a row will take you to the `ContactDetailsPage` showing the `FullName` property value.

6:55
Chris Smith

How it works…

Implementing the MVVM pattern in Xamarin.Forms, we achieved a clean cross-platform mobile solution. The concept is to abstract as much as we can from the platform specifics, program against interfaces, use Dependency Injection, separation of concerns and single responsibility of components.

We started creating a simple class that represents our Model, Contact, that is a POCO class with the data we would normally retrieve from a local database or from a web service.

The data is retrieved with the help of the `DataService` static class; in this recipe, we just hardcoded two contacts for demonstration purposes.

For the Views, the requirements are two pages: one to list all the contacts and one to show the selected contact details when we tap a row. The first and main View is the `ContactListPage`, and for the details we added the `ContactDetailsPage`.

The last part of the MVVM pattern is `ViewModels`. Every View is backed by a `ViewModel`; it's the View's Model, classes that are responsible to present data and execute actions. For this solution, we need three `ViewModels`, so let's take a small tour of the implementation.

One `ViewModel` is `ContactListViewModel` and its responsibility is to show the list of contacts, has one property of `IList<ContactViewModel>`, in reality it's an observable collection. Xamarin.Forms understands this in runtime. Each row is a `ContactViewModel`, a wrapper around our Contact model to present data to the View. As you see here, we have a composition of `ViewModels`, which is totally valid. To add some behavior, we added an `ICommand` property that will invoke its assigned delegate method when we select a list item, in our case `ContactViewModel`. In the constructor of `ContactListViewModel`, we use `DataService` to retrieve all contacts; we also initialize `ICommand<ContactViewModel>` with the method `OnItemSelected` that navigates to `ContactDetailsViewModel`. We set the selected `ContactViewModel` to the public contact property when the Action parameter is invoked. Look at that! More helper goodies from Xamarin.Forms Labs, and navigation is already implemented for each platform; we can use it because it is part of the `XLabs.Forms.Mvvm.ViewModel` class. Note that this property will be null if `NavigationPage` is not used.

But why did we use that code behind in `ContactListPage.xaml.cs`? In `ContactListViewModel`, we have a property that is a list of `ContactViewModel`, so every list row is binded to a `ContactViewModel` instance, which means that a row tap command will need to be handled in that `ViewModel`. Unfortunately, at the moment of this writing there is no out-of-the-box command that will delegate the tap row to the page's binded `ViewModel`, and for that we need a workaround code that handles the delegation of the event to a command in `ContactListViewModel`. This is achieved via the `ItemSelected` event handler of `ListView` and a custom `BindableProperty`. Refer to the book's code for a closer look at the code to reuse it if needed in your solutions.

The MVVM implementation in this recipe is one of the ways to do it, MVVM is a family of patterns; for example, instead of copying the properties of the Contact Model in

`ContactViewModel`, you could inherit the Model from the `ObservableObject` class and expose it directly to the View. Although I've worked fine in previous projects before with this practice and it's completely valid, it breaks the encapsulation and makes available the Model to the View. In the MVVM pattern, the concept of keeping a clean separation of concerns is very important and to maintain this we should let `ViewModel` coordinate everything between the View and the Model.

To make data binding work correctly, we need to implement the `INotifyPropertyChanged` interface; this interface contains an event, `PropertyChanged`. When raised, it notifies the View's bindable properties to update with the new value. Since this interface is already implemented in the `XLabs.Forms.Mvvm.ViewModel` class, actually in the `XLabs.Data.ObservableObject`, we don't need to do anything else than inherit from this class. Boring boilerplate code work is ready for us! Sweet!

The last step in our core cross-platform library is to register our `ViewModels` to the Views; the method `RegisterViews` handle this. The XLabs `ViewFactory` class is used to register which `ViewModel` we want as a `BindingContext` in our View, and we also use it to construct our main page in the App constructor.

The platform-specific projects are really simple: we just have to register `XLabs.IoC.Resolver`; this simple Dependency Injection Container is used by XLabs internally to offer cool things like navigation, injecting our `ViewModels`, and other abstractions.

There’s more…

Nothing feels better than a nice and clean architecture, right? We accomplished a great code foundation in our application implementing the MVVM pattern, binding our `ViewModels` to pages with no major additional work, and data binding to update our UI on a property value change.

One thing can get better though: we use a static `DataService` class to retrieve data. In the real world, we would abstract the data access components and program against an interface and inject the actual implementation using Dependency Injection. For more information how to do this, refer to *Adding a third-party Dependency Injection Container* recipe of this chapter and Chapter 5, *Dude, Where’s My Data?*

Data binding is key in MVVM XAML applications. To learn more about it, go to Chapter 7, *Bind to the Data*.

See also

- https://github.com/XLabs/Xamarin-Forms-Labs
- https://developer.xamarin.com/guides/cross-platform/xamarin-forms/user-interface/xaml-basics/data_bindings_to_mvvm/

Using the event messenger

When writing C# code, we rely a lot on instance events. A class might have some events that another class instance can register a handler and get notified when something happened; for example, classes that implement the `INotifyPropertyChanged` interface raise the `PropertyChanged` event when a property value is changed, or maybe we start a lengthy asynchronous task in another thread from a component we reference and we want to get notified about the progress. It's a great feature, but to make use of events the component that hooks to an instance event of another component must depend on it. Imagine if we have many cases of this communication: we will end up with spaghetti code and dependency references from component to component.

A common requirement is the communication of unrelated components, a `ViewModel` to a `ViewModel` or a service to a `ViewModel` is common use cases.

An event messenger is solving this problem to send loosely coupled messages between components that are not aware of each other with the Publish/Subscribe pattern and a messaging key contract.

A good example would be when a configuration setting has changed and some components need to act on it, maybe refresh data, or have a central component that will take action for all the handled exceptions caught in the application. In this recipe, we will do exactly that.

How to do it…

1. Open Visual Studio and create a new cross-platform solution from the top menu **File | New | Project**, find the **Blank App** (Xamarin.Forms Portable), name it `XamFormsMessagingCenter`, and click **OK**.
2. Right-click the `XamFormsMessagingCenter` PCL, **Add | New Item…**, choose the **Forms Xaml Page** template, set the name to `MainPage`, and click **Add**.
3. Change the label's `Text` property of the newly created page to Hello Messenger!

```
<Label Text="Hello Messenger!" VerticalOptions="Center"
HorizontalOptions="Center" />
```

4. Go to the `App.cs` constructor and set the `MainPage` property to a new instance of our newly created `MainPage` page.

```
public App()
{
  // The root page of your application
  MainPage = new MainPage();
}
```

5. The messaging key for subscribers and publishers is a string parameter. You can of course use hardcoded strings but this practice is error prone and typos could cost you debugging time. Create an `enum` to use as a key contract.

```
public enum MessagingKey
{
  HandledException = 0
}
```

6. Open the `MainPage.xaml.cs` behind-code file and override the `OnAppearing` method. In this method, we will intentionally throw a `NotSupportedException`, catch it, and publish a message to all the subscribers of the `MessagingKey.HandledException.ToString()` key using the `MessegingCenter.Send` method.

```
protected override void OnAppearing()
{
  base.OnAppearing();

  try
  {
    throw new NotSupportedException("Need to fix this!");
  }
  catch(NotSupportedException ex)
  {
    MessagingCenter.Send<object, Exception>(this,
MessagingKey.HandledException.ToString(), ex);
  }
}
```

7. Right-click the core portable library project and create a class with **Add | Class…**, rename to `CrashManager`, and click **Add**. The class has a default constructor,

subscribing to an event, and a `Dispose` method to unsubscribe from the event. It's always good to help the system with cleaning resources.

```
public class CrashManager
{
  public void Dispose()
  {
    MessagingCenter.Unsubscribe<object, Exception>(this,
MessagingKey.HandledException.ToString());
  }

  public CrashManager ()
  {
    MessagingCenter.Subscribe<object, Exception>(this,
MessagingKey.HandledException.ToString(), (obj, ex) =>
    {
      Debug.WriteLine("Exception: {0}", ex);
    });

}
```

8. Go to the `App.cs` file and in the constructor before creating the `MainPage`, add the following code that creates a new instance of `CrashManager`:

```
new CrashManager();
```

9. Run the project to any platform you want; the behavior is exactly the same. When the screen appears, you will see in the Output window a message like the following:

```
[0:] Exception: System.NotSupportedException: Need to fix this!
  at XamFormsMessagingCenter.MainPage.OnAppearing ()……….
```

How it works…

Basically, the `MessagingCenter` is the mediator between our components. These classes don’t know anything about each other except for the messenger and a signature key, the message that they are subscribing or sending.

The `MessagingCenter` is simple: the Send method takes a mandatory generic parameter, the type of the sender, and a second optional generic parameter, the type of the argument. Then we pass two or three parameters. The first is the sender instance, the second is the messaging key signature, and if we have arguments we pass them in the third parameter.

The Subscribe method works almost the same, with a difference in the third parameter that is an Action delegate; we can access the sender instance and any arguments if applicable, meaning if you defined a second generic parameter.

See also

- https://developer.xamarin.com/guides/cross-platform/xamarin-forms/messaging-center/

Adding localization

Creating a mobile application is an opportunity to reach out to the whole world; no barriers! This means that you will want to talk to your users' language, so you need a way to globalize your application.

Xamarin.Forms uses a built-in mechanism for localizing `.NET` applications. You can use RESX files, add your strings for each language code, and depending on the system language settings your application will load the appropriate value.

This is feasible in code and from XAML pages. In this recipe, we will explore the two options and present the string value Hello World! in English, and in French Bonjour le monde!.

How to do it…

1. Start by opening Visual Studio and creating a cross-platform solution, **File | New | Project…**, choose **Blank App** (Xamarin.Forms Portable) from the templates, name it `XamFormsLocalization`, and click **OK**.
2. Create a default RESX file and a French RESX file in the portable library, right-click and **Add | New Item…**, from the templates choose `Resources` File, name it `AppResources.resx` , click **Add**. This is the base resource language strings. Repeat the same step but this time give the file the name `AppResources.fr.resx`.
3. For the `AppResources.resx` file, add the Name and Value strings like the following screenshot:

4. And for our French resources, the following are the **Name** and **Value** strings:

5. We need a XAML page for our application `MainPage`. Right-click, **Add | New Item…**, choose **Forms Xaml Page** from the templates, rename to `MainPage.xaml`, and click **Add**.
6. We added the resource files with the strings, but we somehow need to know the system language that the user selected; this needs some abstractions and the use of `DependencyService` to get this information in the platform-specific application layer. In our `XamFormsLocalization` portable library, right-click, **Add | New Item…**, choose **Interface**, rename to `ILocalize.cs`, and add a method signature. You can find the details in the next code snippet:

```
public interface ILocalize
{
  CultureInfo GetCurrentCultureInfo();
}
```

7. Add an interface implementation for each platform, right-click on the platform projects and **Add | Class…**, name it `Localize.cs`, implement the `ILocalize` interface, and append the `Dependency` attribute to register the implementation with the interface.

```
// Android
```

```
[assembly: Dependency(typeof(XamFormsLocalization.Droid.Localize))]

  namespace XamFormsLocalization.Droid
  {
  public class Localize : ILocalize
  {
    public CultureInfo GetCurrentCultureInfo()
    {
      Locale androidLocale = Locale.Default;
      string netLanguage = androidLocale.ToString().Replace("_", "-");
      Debug.WriteLine("NetLanguage: {0}", netLanguage);
      return new CultureInfo(netLanguage);
    }
  }
}

// iOS
[assembly: Dependency(typeof(XamFormsLocalization.iOS.Localize))]

  namespace XamFormsLocalization.iOS
  {
    public class Localize : ILocalize
    {
      public CultureInfo GetCurrentCultureInfo()
      {
        string netLanguage = "en";
        string prefLanguageOnly = "en";

        if (NSLocale.PreferredLanguages.Length > 0)
        {
          var pref = NSLocale.PreferredLanguages[0];
          prefLanguageOnly = pref.Substring(0, 2);
          netLanguage = pref.Replace("_", "-");
          Debug.WriteLine("Preferred language:" + netLanguage);
        }

        CultureInfo ci = null;
        try
        {
          ci = new CultureInfo(netLanguage);
        }
        catch
        {
        // iOS locale not valid .NET culture (eg. "en-ES" :
English in Spain)
        // fallback to first characters, in this case "en"
        ci = new CultureInfo(prefLanguageOnly);
        }

        return ci;
      }
    }
  }

// Windows Phone
```

```
[assembly: Dependency(typeof(XamFormsLocalization.WinPhone.Localize))]

  namespace XamFormsLocalization.WinPhone
  {
    public class Localize : ILocalize
    {
      public CultureInfo GetCurrentCultureInfo()
      {
        return Thread.CurrentThread.CurrentUICulture;
      }
    }
  }
```

8. For the iOS platform, now we need to declare the languages we support in the `Info.plist`, so add the following key/values:

Key	Type	Value
▼ Information Property List	Dictionary	(11 items)
▼ Localizations	Array	(1 item)
Item 0	String	French
Localization native development region	String	en

9. In Windows Phone, we have to set the supported languages in the options. Right-click the project and **Properties**. In the section **Application | Supported Cultures**, check **French** (France) along with the **English** options.

10. Open the `XamFormsLocalization` portable library `App.cs` file and replace the constructor default code with the following snippet:

```
public App()
{
  if (Device.OS != TargetPlatform.WinPhone)
  {
    AppResources.Culture = DependencyService.Get<ILocalize>
().GetCurrentCultureInfo();
  }

  MainPage = new MainPage();
}
```

Note

You must have noticed that we only retrieve `CultureInfo` for the platforms iOS and Android, and that is because the Windows Phone resource framework gets the selected language automatically.

11. You’re ready to test the localization setup we did in code. Open the `MainPage.xaml` and set a name for the `Label` control so that you can access it in the behind code.

```
<Label x:Name="label" Text="" VerticalOptions="Center"
HorizontalOptions="Center" />
```

12. Go to `MainPage.xaml.cs` and add the following method:

```
protected override void OnAppearing()
{
  base.OnAppearing();

  label.Text = AppResources.HelloWorld;
}
```

13. Run the code and in all the platforms, you should get the same result. The following is a screenshot of the Android application:

3:37
Hello World!

14. Comment the code we added in step 10. We will enable our application to use the resource files in XAML.
15. To translate the user interface controls directly in XAML, we need to create a class implementing `IMarkupExtension`, which will expose the RESX resources. In the `XamFormsLocalization` portable library, right-click, **Add | Class…**:

```
// You exclude the 'Extension' suffix when using in Xaml markup
[ContentProperty("Text")]
public class TranslateExtension : IMarkupExtension
{
  readonly CultureInfo cultureInfo;
  const string ResourceId = "XamFormsLocalization.AppResources";

  public TranslateExtension()
  {
    cultureInfo = DependencyService.Get<ILocalize>
().GetCurrentCultureInfo();
  }

  public string Text { get; set; }

  public object ProvideValue(IServiceProvider serviceProvider)
  {
    if (string.IsNullOrEmpty(Text))
    return "";

    ResourceManager resmgr = new ResourceManager(ResourceId,
typeof(TranslateExtension).GetTypeInfo().Assembly);
    var translation = resmgr.GetString(Text, cultureInfo);

    if (translation == null)
    {
      throw new ArgumentException(
      String.Format("Key '{0}' was not found in resources '{1}' for
culture '{2}'.", Text, ResourceId, cultureInfo.Name),"Text");
    }
    return translation;
  }
}
```

16. Go to `MainPage.xaml` and add the namespace needed to make available `TranslateExtension` and use the binding syntax to access the resource strings by name. The following is the `MainPage.xaml` code changed to use the `HelloWorld` string resource in the `Text` property:

```
<?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:resource="clr-
namespace:XamFormsLocalization;assembly=XamFormsLocalization"
x:Class="XamFormsLocalization.MainPage">
    <Label x:Name="label" Text="{resource:Translate HelloWorld}"
```

```
VerticalOptions="Center" HorizontalOptions="Center" />
</ContentPage>
```

17. Run the application for each platform and you should get the same results. We changed the language preference to French (France) in the iOS and Windows Phone platforms.

iOS:

Opérateur
16:39
Bonjour le monde!

Windows Phone:

How it works…

Fairly easy, you must say! Xamarin.Forms using the .NET mechanism of globalization made it possible for us to create our RESX files in a PCL and share it between platforms. Of course there are couple of extra steps, but it's a small price to pay for major advantages.

The solution supports English, the default `AppResources.resx` file, and French (France), `AppResources.fr.resx`. All the languages-supporting RESX files follow a naming convention defining the language-specific code after the name and before the extension, `AppResources.{language-code}.resx`. You can use the two-letter language code but there are languages that you will want to be more specific, like Chinese or Brazilian Portuguese, for example.

In the `AppResources` static class, there is a property, Culture, where we need to set to the system's language preference. The implementation of retrieving this information is platform specific, so we will follow the service location pattern using the available `Xamarin.Forms DependencyService`. In our portable class library, we create an `ILocalize` interface to program with, and for all the platforms' Localize implementations we implement this interface and register to `DependencyService`. In the Xamarin.Forms application creation, `App.cs` constructor, we set the culture retrieving the platform implementation `ILocalize` via `DependencyService`. We don't have to do this for the Windows Phone platform as the framework sets the culture automatically, but we still need the implementation as we use it for the `IMarkupExtension` implementation.

So far, we've set up everything needed to access the values in code; if you followed the steps, you've already seen it in action. Xamarin.Forms has the power of XAML, and instead of boilerplate behind code or wiring up binding properties in `ViewModels`, we want to set the value directly in XAML. By implementing the `IMarkupExtension` interface and using the binding syntax in XAML, Xamarin.Forms can retrieve the value.

In the `TranslateExtension` class, an `IMarkupExtension`, we retrieve using `DependencyService` the `CultureInfo`, decorate the class with the `ContentPropertyAttribute` and set it to Text. In the `ProvideValue` implemented method, we retrieve a `ResourceManager` instance using the `ResourceId` constant private field; this is important and it has to be in the format `{Namespace}.{AppResources}` , with the second parameter the assembly that we get with the help of reflection, and with the `ResourceManager` we retrieve the translation string. We add some troubleshooting help when in debug to throw an exception to make sure the key exists in `AppResources`.

Last step: we declare the namespace to access the `TranslateExtension` class and request a value for the specified key using the `{Binding}` syntax.

Happy globalizing!

There’s more…

It is possible in Xamarin.Forms to localize platform-specific elements like images or the application name. For more information on how to do this, refer to the Xamarin documentation.

https://developer.xamarin.com/guides/cross-platform/xamarin-forms/localization/.

Chapter 5. Dude, Where’s my Data?

In this chapter, we will cover the following recipes:

- Creating a shared SQLite data access
- Performing CRUD operations in SQLite
- Consuming REST web services
- Leveraging native REST libraries and making efficient network calls

Introduction

There are three types of applications: online, offline, and both, where in some way the device synchronizes its local data with some remote data in a server.

In the disconnected mobile world, we have three local storage options to store data in your own sandbox space, the areas where you have the required permission to read and write data:

- Preferences (simple key\value pairs user settings)
- Direct access to the filesystem (JSON, XML, text, binary)
- Local database (SQLite, NoSQL)

You can use one, none, or all of these options depending on your needs and your architecture decisions. Usually, my personal preference for local storage is to save user settings in preferences and when I need to query data, manipulate, and save back, the standard is a SQLite database built-in in the iOS and Android platforms and some minimal effort to make it available in Windows Phone.

For connected applications, you will have the need of a server, which will query a database and return the data in a formatted representation structure, usually JSON or XML. There are different formats you might work with over the network, but the standard and most famous are JSON and XML querying a web service.

For the first half of the chapter, we will learn how to create a shared data access SQLite connection, create tables, query data, insert, update, and delete, known as **CRUD** operations.

We will get our hands on the available HTTP client API and perform CRUD operations.

In the end, we will see how, with the help of some open source libraries we can improve the performance, schedule the priority, and handle the concurrency of network requests.

Dive in!

Creating a shared SQLite data access

This recipe demonstrates how to set up a cross-platform Xamarin.Forms solution to use SQLite and create the core most important object: a SQLite connection.

How to do it…

1. Create a **Blank App** (Xamarin.Forms Portable) named `XamFormsSQLiteDataAccess` in Visual Studio from **File | New | Project…**
2. For each project, we will add two NuGet packages: `SQLite.Net-PCL` and `SQLite.Net.Async-PCL`, a cross-platform SQLite client and ORM. Right-click in the `XamFormsSQLiteDataAccess` portable core library and choose **Manage NuGet Packages**, search for the packages, and add them.
3. Repeat step two for the Android, iOS, and Windows Phone projects.
4. iOS and Android support SQLite out of the box, whereas Windows Phone needs an extra step. Go to the Visual Studio top menu and click **Tools | Extensions and Updates** and search for `sqlite for windows phone`. In the results returned, choose the extension **SQLite for Windows Phone**.

5. In the Windows Phone project, right-click the `References` folder and choose **Add Reference** In the templates, click **Windows Phone SDK 8.0**, then **Extensions**, and check the **SQLite for Windows Phone** extension available. Ready to go!

6. If you try to build the Windows Phone project, you will receive an error. It has to do with the platform architecture: only x86 architecture is supported by SQLite for the Windows Phone library. Right-click the solution and choose **Configuration Manager**; change the **XamFormsDataAccess.WinPhone** platform cell to **x86**.

7. Right-click the `XamFormsSQLiteDataAccess` library and **Add | New Item…**, choose **Interface**, and name it `ISQLiteConnection.cs`. See the code in the following snippet:

```
public interface ISQLiteConnection
{
  string GetDataBasePath();
  SQLiteAsyncConnection GetConnection();
}
```

8. Go to the Android project and create a class named `SQLiteConnectionDroid.cs` that will implement the `ISQLiteConnection` interface. Right-click in the project and choose **Add | Class…**. Find the implementation code next:

```
[assembly:
Dependency(typeof(XamFormsSQLiteDataAccess.Droid.SQLiteConnectionDroid)
)]

  namespace XamFormsSQLiteDataAccess.Droid
  {
    public class SQLiteConnectionDroid : ISQLiteConnection
    {
      private SQLiteAsyncConnection _connection;

      public string GetDataBasePath()
      {
        string filename = "MyDb.db3";
        string path =
System.Environment.GetFolderPath(System.Environment.SpecialFolder.Perso
```

```
nal);
        return Path.Combine(path, filename);
      }

      public SQLiteAsyncConnection GetConnection()
      {
        if (_connection != null)
        {
          return _connection;
        }

        SQLiteConnectionWithLock connectioonWithLock =
        new SQLiteConnectionWithLock(new SQLitePlatformAndroid(), new
SQLiteConnectionString(GetDataBasePath(), true));
        return _connection = new SQLiteAsyncConnection(() =>
connectioonWithLock);
      }
    }
  }
```

9. In the iOS project, create a class named `SQLiteConnectionTouch.cs`. See the following implementation:

```
[assembly:
Dependency(typeof(XamFormsSQLiteDataAccess.iOS.SQLiteConnectionTouch))]

  namespace XamFormsSQLiteDataAccess.iOS
  {
    public class SQLiteConnectionTouch : ISQLiteConnection
    {
      private SQLiteAsyncConnection _connection;

      public string GetDataBasePath()
      {
        string filename = "MyDb.db3";
        string docFolder =
Environment.GetFolderPath(Environment.SpecialFolder.Personal);
        string libFolder = Path.Combine(docFolder, "..", "Library",
"Databases");

        if (!Directory.Exists(libFolder))
        {
          Directory.CreateDirectory(libFolder);
        }

        return Path.Combine(libFolder, filename);
      }

      public SQLiteAsyncConnection GetConnection()
      {
        if (_connection != null)
        {
          return _connection;
        }

        SQLiteConnectionWithLock connectioonWithLock =
```

```
        new SQLiteConnectionWithLock(
        new SQLitePlatformIOS(),
        new SQLiteConnectionString(GetDataBasePath(), true));
        return _connection = new SQLiteAsyncConnection(() =>
connectioonWithLock);
      }
    }
  }
```

10. And for the Windows Phone platform, create a class named `SQLiteConnectionWinPhone.cs`.

```
[assembly:
Dependency(typeof(XamFormsSQLiteDataAccess.WinPhone.SQLiteConnectionWin
Phone))]

  namespace XamFormsSQLiteDataAccess.WinPhone
  {
    public class SQLiteConnectionWinPhone : ISQLiteConnection
    {
      private SQLiteAsyncConnection _connection;

      public string GetDataBasePath()
      {
        string filename = "MyDb.db3";
        string path =
Windows.Storage.ApplicationData.Current.LocalFolder.Path;
        return Path.Combine(path, filename);
      }

      public SQLiteAsyncConnection GetConnection()
      {
        if (_connection != null)
        {
          return _connection;
        }

        SQLiteConnectionWithLock connectioonWithLock =
        new SQLiteConnectionWithLock(
        new SQLitePlatformWP8(),
        new SQLiteConnectionString(GetDataBasePath(), true));
        return _connection = new SQLiteAsyncConnection(() =>
connectioonWithLock);
      }
    }
  }
```

11. To test that we are ready to create and work against a local SQLite database in our application from our core portable class library, replace the `App.cs` constructor code with the following:

```
public App()
{
  // The root page of your application
  ISQLiteConnection connection =
DependencyService.Get<ISQLiteConnection>();
```

```
    if (connection.GetConnection () != null) {
      Debug.WriteLine ("SQLite connection is ready!");
    }

    MainPage = new ContentPage
    {
      Content = new StackLayout
      {
        VerticalOptions = LayoutOptions.Center,
        Children = {
          new Label {
            XAlign = TextAlignment.Center,
            Text = connection.GetDataBasePath()
          }
        }
      }
    };
  }
```

12. Run the application for each platform. You will get a message in the output window that says **SQLite connection is ready**!” and the database path in the center of the screen.

iOS:

Opérateur
07:19
/Users/georgiostaskos/Library/Developer/
CoreSimulator/Devices/5B25FC1F-4348-4F5B-
B95A-B81E7DDA34B6/data/Containers/Data/
Application/E0B306B7-
FB1D-4EC9-9A06-0B7D876EA276/
Documents/../Library/Databases/MyDb.db3

Android:

Windows Phone:

8:55
C:\Data\Users\DefApps\AppData
\{443FE440-7D29-4EDF-8A2F-348667E24248}\local
\MyDb.db3

How it works…

This is the basic setup you need to create a shared SQLite data access connection. The challenge here is the different filesystem APIs for each platform, but we solved this with the out-of-the-box `DependencyService` injecting the implementation and programming against the interface. In a real-world application, you might want to leverage Dependency Injection and push the `ISQLiteConnection` instance in your repository classes via constructor; you can see how to do this in the recipe *Adding a third-party Dependency Injection Container* of Chapter 4, *Different Cars, Same Engine*.

For Windows Phone, an extra step is needed, SQLite is not built-in for the platform but an extension is available providing the necessary library; steps 4 and 5 explain how to do it.

`SQLite.Net-PCL` has an object called `SQLiteAsyncConnection`; you need an instance of this object to perform any operation on the database. To create an instance of this class, we need access to the native platform to retrieve the database folder path and specify the platform with an `ISQLitePlatform` implementation. To accomplish this, we created a class for each platform implementing an interface and using `DependencyService` to resolve it. If you want to know more about how to use `DependencyService`, go to *Using the dependency locator* recipe of Chapter 4, *Different Cars, Same Engine*.

See also

- https://github.com/oysteinkrog/SQLite.Net-PCL

Performing CRUD operations in SQLite

CRUD operations (create, read, update, delete) are the four basic functions in persistent storage. Whatever the storage option, you will need to perform these actions.

There are patterns and best practices to achieve reusable data access components and the most common implementation is the Repository Pattern. In this recipe, we create a generic repository that we use to perform CRUD operations against a SQLite database.

How to do it…

1. In Visual Studio, create a **Blank App** (Xamarin.Forms Portable) project named `XamFormsCRUDSQLite` from the top menu, **File** | **New** | **Project…**.
2. For all the projects in the solution, we need the `SQLite.Net-PCL` and SQLite.Net.Async-PCL NuGet packages. Right-click on every project and choose **Manage NuGet Packages**; search and install the packages.
3. Create a folder named `Data`: in the `XamFormsCRUDSQLite` PCL, right-click **Add** | **New Folder**.
4. In the newly created folder, we assume that you created `ISQLiteAsyncConnectionService` and also the platform implementations to retrieve a SQLite connection via `DependencyService`. Refer to the code of this recipe to see the details or go to the recipe *Creating a shared SQLite data access* in this chapter.
5. Create a generic interface named `IRepository.cs`: right-click the `Data` folder, **Add** | **New Item…**, choose **Interface**, and click **Add**. See the following method signatures:

```
public interface IRepository<T> where T : class, new()
{
  Task<List<T>> GetAllAsync();
  Task<int> InsertAsync(T entity);
  Task<int> UpdateAsync(T entity);
  Task<int> DeleteAsync(T entity);
}
```

6. One generic implementation is needed for this repository. Create a class by right-clicking the `Data` folder, **Add** | **Class…**, and name it `Repository.cs`. Copy the following implementation:

```
public class Repository<T> : IRepository<T> where T : class, new()
  {
    ISQLiteAsyncConnectionService _connectionService;
    SQLiteAsyncConnection _connection;

    public Repository(ISQLiteAsyncConnectionService connectionService)
    {
      _connectionService = connectionService;
      _connection = _connectionService.GetConnection();
      _connection.CreateTableAsync<T>();
    }

    public Task<List<T>> GetAllAsync()
    {
      return _connection.Table<T>().ToListAsync();
    }

    public Task<int> InsertAsync(T entity)
    {
      return _connection.InsertAsync(entity);
    }

    public Task<int> UpdateAsync(T entity)
    {
```

```
      return _connection.UpdateAsync(entity);
    }

    public Task<int> DeleteAsync(T entity)
    {
      return _connection.DeleteAsync(entity);
    }
  }
```

7. Create a folder named `Models`: right-click **Add** | **New Folder**.
8. Right-click the folder `Models` and **Add** | **Class…**. Name it `Contact.cs`, make it public, and add three simple properties: `FirstName`, `LastName`, and `Id`. See the following implementation:

```
public class Contact
{
  [PrimaryKey]
  public Guid Id { get; set; }
  public string FirstName { get; set; }
  public string LastName { get; set; }
}
```

Note

Notice the `PrimaryKeyAttribute` decoration in the `Id` property? This is needed for the database. We define a `primary key` in the database to uniquely identify the row, for indexing, but also you will not be able to delete a row without it.

9. Create a folder named `Views`: right-click and **Add** | **New Folder**.
10. Right-click the `Views` folder and **Add** | **New Item…**, choose **Forms Xaml Page** from the templates, name it `MainPage.xaml`, and click `Add`.
11. Repeat step 11 for a page named `ContactDetailPage.xaml`.
12. Go to the `App.cs` constructor and replace the code with the following single line of code:

```
MainPage = new NavigationPage(new MainPage());
```

Note

Almost ready. All we need is to wire up some user interface and add functionality in the pages' behind code. Here, we omit the XAML code. If you want to see the XAML pages' contents, refer to this recipe code. To learn more about Xamarin.Forms XAML, go to Chapter 2, *Declare Once, Visualize Everywhere*; for data binding, visit Chapter 7, *Bind to the Data*; and for the `ListView` control, go to Chapter 8, *A List to View*.

13. In `MainPage.xaml.cs`, we retrieve the connection and pass it to the contacts repository constructor. We keep a field of the repository to invoke the CRUD methods.

```
private IRepository<Contact> _contactRepo;

public MainPage()
```

```
{
  InitializeComponent();

  ISQLiteAsyncConnectionService connectionService =
DependencyService.Get<ISQLiteAsyncConnectionService>();
  _contactRepo = new Repository<Contact>(connectionService);
}
```

14. Add the following code to refresh the list from the database, add a row, delete a row, and go to the details of a row to update a contact:

```
private async void OnAddContactClick(object sender, EventArgs e)
  {
    await _contactRepo.InsertAsync(new Contact
    {
      Id = Guid.NewGuid(),
      FirstName = "FirstName: " + new Random().Next(10),
      LastName = "LastName: " + new Random().Next(10)
    });
    await RefreshAsync();
  }

  public async void OnItemSelected(object sender,
SelectedItemChangedEventArgs e)
  {
    if (e.SelectedItem == null) return;
    ((ListView)sender).SelectedItem = null;
    await Navigation.PushAsync(new
ContactDetailPage((Contact)e.SelectedItem));
  }

  public async void OnCellClicked(object sender, EventArgs e)
  {
    Button button = (Button)sender;
    Guid id = (Guid) button.CommandParameter;
    await
_contactRepo.DeleteAsync(((List<Contact>)BindingContext).FirstOrDefault
(p => p.Id == id));
    await RefreshAsync();
  }

  protected async override void OnAppearing()
  {
    base.OnAppearing();

    await RefreshAsync();
  }

  private async Task RefreshAsync()
  {
    BindingContext = await _contactRepo.GetAllAsync();
  }
```

15. Go to `ContactDetailsPage.xaml.cs` and add the following code snippet. In this page, we allow the user to update a contact's details.

```
private IRepository<Contact> _contactRepo;
public ContactDetailPage(Contact contact)
  {
    InitializeComponent();

    BindingContext = contact;
    ISQLiteAsyncConnectionService connectionService =
DependencyService.Get<ISQLiteAsyncConnectionService>();
    _contactRepo = new Repository<Contact>(connectionService);
  }

  private async void OnSaveContactClick(object sender, EventArgs e)
  {
    await _contactRepo.UpdateAsync((Contact)BindingContext);
    await Navigation.PopAsync();
  }
```

16. Ready! See next the screenshots of running the solution. Add, delete, and update a row.

Android:

Android contact details:

Windows Phone:

6:58
FirstName: 8
LastName: 8
Delete
FirstName: 5
LastName: 5
Delete

iOS:

Carrier
11:53 AM
Add
FirstName: 5
LastName: 5
Delete

How it works…

With Xamarin.Forms, a little help from NuGet and the community, a bit of architecture setup, and working with the SQLite local database, a shared repository for all platforms becomes easy.

Having created a service for each platform to retrieve the connection and injected via `DependencyService`, we create a generic interface, `IRepository<T>`, and a generic implementation repository class, `Repository<T>`. With this solution, we encapsulate all the data access in a reusable single responsibility container.

The `IRepository<T>` interface defines four methods: fetching all the related entities of the corresponding table and inserting, updating, and deleting an entity.

In the `Repository<T>` generic implementation, we pass the I`SQLiteAsyncConnectionService` dependency in our constructor, keeping the important instance of the `SQLiteAsyncConnection` in a private field and creating asynchronously the T table; if the tables exist, nothing happens.

For each `IRepository<T>` implemented method, we use the `SQLiteAsyncConnection` instance and its corresponding methods for CRUD operations.

SQLite is a non-thread-safe database. Fear not! The `SQLite.Net-PCL` NuGet package is utilizing a connection with a lock that makes it thread safe. Neat!

In this recipe, we used only one model class, contact, mapping against the database to a contact table. Decorating the Id property with `PrimaryKeyAttribute` gave the necessary instruction to the `SQLite-Net-PCL` library to create a column Id as the primary column key. If you want to see how you can define relationships and load records with their corresponding children, refer to https://bitbucket.org/twincoders/sqlite-net-extensions.

The XAML UI is out of the scope of the recipe in this topic, so we removed the code. You can easily check it out in the book code or visit the chapters in the *See also* section to learn more details about how it works. To see how `ListView` works, go to Chapter 8, *A List to View*.

In the behind pages' constructors, we retrieve the I`SQLiteAsyncConnectionService` dependency and create a `Repository<Contact>`; this will create a contact table if it does not exist and provide us with all the CRUD operations.

In `MainPage.xaml.cs`, we have wired event handlers for navigation, retrieving all the contact records, insert, and delete a record.

The `ContactDetailPage.xaml.cs` constructor accepts a contact argument and contains an event handler to update the contact and go back to the parent page.

See also

- https://github.com/oysteinkrog/SQLite.Net-PCL
- https://bitbucket.org/twincoders/sqlite-net-extensions
- *Creating a shared SQLite data access* recipe in this chapter
- Chapter 2, *Declare Once, Visualize Everywhere*
- Chapter 7, *Bind to the Data*
- Chapter 8, *A List to View*

Consuming REST web services

Almost all applications today connect to a server to persist data. Some are connected, meaning an active Internet connection is required, whereas others have only offline data, with a local database on the device. Many utilize a combination of offline-online applications with an implementation of data syncing.

In this recipe, we will demonstrate how to connect to a REST web service and perform CRUD operations in a connected architecture application.

REST (short for **Representational State Transfer**) is not a protocol or a specific implementation of a design pattern, but you can think of it as an architectural style that relies on a stateless, client-server, cacheable communications protocol, and in most of the cases the HTTP protocol is used. REST is the standard architectural style of most web services that you will see today.

How to do it…

1. In Visual Studio, create a new **Blank App** (Xamarin.Forms Portable) named `XamFormsREST`. In the top menu, select **File** | **New** | **Project…**.

   ### Note

   There are numerous libraries to install and work with REST web services. One of the most well-known, and one that works great with Xamarin, is the `Microsoft.Net.Http` NuGet package. For the payload of our requests, we are using JSON format, and to serialize objects to JSON representation, we are using the `Newtonsoft.Json` NuGet package.

2. Right-click the core `XamFormsREST` library and choose **Manage NuGet Packages**. Search for `Microsoft.Net.Http` and install the package. Repeat the search and install step for the `Newtonsoft.Json` NuGet package.
3. Repeat step 2 only for the `XamFormsREST.WinPhone` project and install the `Microsoft.Net.Http` NuGet package.
4. Back to the `XamFormsREST` core library, create two folders named `Models` and `Views` by right-clicking the project and choosing **Add** | **New Folder**.
5. Right-click the `Models` folder and create a new class, **Add** | **Class…**, named `Order.cs`. The class has three simple properties. See the following implementation:

   ```
   public class Order
   {
     public string ObjectId { get; set; }
     public string OrderNumber { get; set; }
     public string ClientComments { get; set; }
   }
   ```

   ### Note

   A REST web service is needed. For the sake of our recipe, we are using Parse, an easy to set up, simple, and powerful cloud database that exposes a REST API to work with. For information on how to get it up and running (in a few minutes), go to https://www.parse.com/. Just make sure you create a class named `Order` in your Parse application.

6. Right-click the `Models` folder and create a class named `DataService.cs`. This is the data access class to perform all required CRUD operations against the REST web service and the `Order` class endpoint. Check the full implementation next:

   ```
   public class DataService
   {
     private readonly JsonSerializerSettings _jsonSerializerSettings = new
   JsonSerializerSettings
     {
     ContractResolver = new CamelCasePropertyNamesContractResolver()
     };
     const string Url = "https://api.parse.com/1/classes/Order";
   ```

```
  private HttpClient GetClient()
  {
    HttpClient client = new HttpClient();
    client.DefaultRequestHeaders.Add("X-Parse-Application-Id",
"YOUR_APPLICATION_ID");
    client.DefaultRequestHeaders.Add("X-Parse-REST-API-Key",
"YOUR_REST_API_KEY");
    client.DefaultRequestHeaders.Accept.Add(new
MediaTypeWithQualityHeaderValue("application/json"));
    return client;
  }

  public async Task<IEnumerable<Order>> GetAllAsync()
  {
    using (HttpClient client = GetClient())
    {
      HttpResponseMessage httpResponseMessage = await
client.GetAsync(Url);
      httpResponseMessage.EnsureSuccessStatusCode();
      string content = await
httpResponseMessage.Content.ReadAsStringAsync();
      JObject jsonObject = JObject.Parse(content);
      string ordersJson = jsonObject["results"].ToString();
      return JsonConvert.DeserializeObject<IEnumerable<Order>>
(ordersJson);
    }
  }

  public async Task<Order> InsertAsync(Order order)
  {
    using (HttpClient client = GetClient())
    {
      HttpResponseMessage httpResponseMessage = await
client.PostAsync("https://api.parse.com/1/classes/Order",
      new StringContent(JsonConvert.SerializeObject(order,
_jsonSerializerSettings),
      Encoding.UTF8, "application/json"));
      httpResponseMessage.EnsureSuccessStatusCode();
      string content = await
httpResponseMessage.Content.ReadAsStringAsync();
      return JsonConvert.DeserializeObject<Order>(content);
    }
  }

  public async Task UpdateAsync(Order order)
  {
    using (HttpClient client = GetClient())
    {
      HttpResponseMessage httpResponseMessage =
      await client.PutAsync(Url + "/" + order.ObjectId,
      new StringContent(JsonConvert.SerializeObject(order,
_jsonSerializerSettings),
      Encoding.UTF8, "application/json"));
      httpResponseMessage.EnsureSuccessStatusCode();
    }
  }
```

```
  public async Task DeleteAsync(string objectId)
  {
    using (HttpClient client = GetClient())
    {
      HttpResponseMessage httpResponseMessage = await
client.DeleteAsync(Url + "/" + objectId);
      httpResponseMessage.EnsureSuccessStatusCode();
    }
  }
}
```

7. We will create two views to interact with the data. Right-click the `Views` folder and **Add | New Item…**; from the templates, choose **Forms Xaml Page** and name it `MainPage.xaml`. Repeat this for a page named `OrderDetailsPage.xaml`.

   ### Note

   Almost ready. All we need is to wire up some user interface and add functionality in the pages' behind code. Here, we omit the XAML code. If you want to see the XAML pages' contents, refer to this recipe code. To learn more about Xamarin.Forms XAML, go to Chapter 2, *Declare Once, Visualize Everywhere*; for data binding, visit Chapter 7, *Bind to the Data*; and for `ListView` control, go to Chapter 8, *A LIst to View*.

8. In the `MainPage.xaml.cs` behind code, we add the following implementation:

```
public partial class MainPage : ContentPage
{
  private readonly DataService _dataService = new DataService();

  public MainPage()
  {
    InitializeComponent();
  }

  private async void OnAddUserClick(object sender, EventArgs e)
  {
    await _dataService.InsertAsync(new Order
    {
      OrderNumber = new Random().Next(100).ToString()
    });
    await RefreshAsync();
  }

  public async void OnItemSelected(object sender,
SelectedItemChangedEventArgs e)
  {
    if (e.SelectedItem == null) return;
    ((ListView)sender).SelectedItem = null;
    await Navigation.PushAsync(new
OrderDetailsPage((Order)e.SelectedItem));
  }

  public async void OnCellClicked(object sender, EventArgs e)
```

```
    {
      Button button = (Button)sender;
      string id = (string)button.CommandParameter;
      await
_dataService.DeleteAsync(((List<Order>)BindingContext).FirstOrDefault(p
=> p.ObjectId == id).ObjectId);
      await RefreshAsync();
    }

    protected async override void OnAppearing()
    {
      base.OnAppearing();

      await RefreshAsync();
    }

    private async Task RefreshAsync()
    {
      BindingContext = await _dataService.GetAllAsync();
    }
  }
```

9. Go to `OrderDetailsPage.xaml.cs` and set up the class like the following snippet:

```
public partial class OrderDetailsPage : ContentPage
{
  private readonly DataService _dataService = new DataService();

  public OrderDetailsPage(Order order)
  {
    InitializeComponent();

    BindingContext = order;
  }

  private async void OnSaveUserClick(object sender, ventArgs e)
  {
    await _dataService.UpdateAsync((Order)BindingContext);
    await Navigation.PopAsync();
  }
}
```

10. Go to `App.cs` constructor and replace the code with the following:

```
MainPage = new NavigationPage(new MainPage());
```

11. Run the applications and interact with the REST web service to fetch, insert, update, and delete order records.

Android:

Android – order details:

Windows Phone:

iOS:

Carrier
4:05 PM
Add
75
Please deliver to doorman.
Delete

How it works…

Working against REST web services has a lot of detail, but it can be simplified with the use of one of the client libraries out there: one of the most well-known and proven to work great with Xamarin is the `Microsoft.Net.Http` NuGet package.

Many backend platforms provide SDKs too. In our case, using Parse simplifies even the database design details and provide us with the advantage of getting up and running in two minutes. They also provide an SDK, depending on the platform, but we utilized the plain REST endpoints, flexible for prototyping your Xamarin application and then switch to your server implementation by changing two or three lines of code.

Our user interface is simple: we have a `MainPage` with a `ListView` to present rows of Order data, and a toolbar button to add an order with a random `OrderNumber` in the database through the REST web service endpoint. Tapping a row navigates to the `OrderDetailsPage` where you can update the `OrderComments` of an order.

In the behind code of the pages, we construct and use a `DataService` instance class to do all the work between the client and the REST web service.

`DataService` is the core of the data exchange with the server. For this recipe, we kept it simple, but you could implement the Repository pattern to make it generic for all the additional classes added in the future. To see how to do this, go to the section Performing CRUD operations in SQLite. You might want to also eliminate behind code and make your code reusable for additional future platforms implementing the MVVM pattern. Details on how to do this are in the *Architecture design with Model-View-ViewModel (MVVM) pattern* recipe of Chapter 4, *Different Cars, Same Engine*.

The class has a constant string property named `Url`, which is the base URL to reach the order resource URI.

We added a private method, `GetClient`, returning an `HttpClient` instance, the main object we need to make requests to the web service.

In the `GetAllAsync` method, we retrieve all the rows from the server database through the REST endpoint making a `GET` request, we invoke the `EnsureSuccessStatusCode` method to make sure everything went ok with our request call, and in the end we use the `JSON.Net` (`Newtonsoft.Json` NuGet package) library to deserialize the JSON string representation format to `Order` class instances.

The `InsertAsync` method accepts an `Order` instance; here, we do a POST with the `HttpClient.PostAsync` method, which accepts two arguments: the URL and an `HttpContent`. For the latter, we create `StringContent` serializing the Order with a `JsonSerializerSettings` instance to meet the `camelCase` JSON representation format standard, set the encoding to UTF8 and the content-type to application/json, ensure the success status code, and retrieve the response payload, which is the order with `objectId` set by the server.

In the `UpdateAsync` method, we pass an `Order` instance. We only use the `ObjectId`

property in the URL endpoint and with a `PUT` method, `HttpClient.PutAsync`, we pass the `order` serialized as in the `InsertAsync` method. We ensure the success status code and that’s it, the order has been updated.

For `DeleteAsync`, we just need `ObjectId` and a `DELETE` method, `HttpClient.DeleteAsync`, passing `ObjectId` in the URL endpoint. Ensure the success status code and the order is deleted.

This is all you need to work against any REST web service out there. Happy data consuming!

Leveraging native REST libraries and making efficient network calls

In a connected world, while your application grows, you will see that calls to web services are getting stressed and often, and all these calls will be asynchronous concurrent network requests.

Issuing a lot of requests will end up making your app slow, and managing these requests by hand, such as who has priority and when, or how many calls at a time are allowed, will break a lot of encapsulation between different components, resulting in spaghetti code.

It’s ok! We can fix this. There are two libraries out there, created by Paul Betts, to save the day: `modernhttpclient` and `Punchclock`. Let’s see how to install and use them in our network calls.

How to do it…

1. In Visual Studio, create a **Blank App** (Xamarin.Forms Portable) solution named `XamFormsEfficientNetworking` from the top menu, **File | New | Project…**.
2. Right-click the portable class library, `XamFormsEfficientNetworking`, and choose **Manage NuGet Packages**. Search and install the following packages:
   - `Microsoft.Net.Http`
   - `Newtosoft.Json`
   - `modernhttpclient`
   - `Punchclock`
3. Install only the `modernhttpclient` NuGet package in the `XamFormsEfficientNetworking.Droid` and `XamFormsEfficientNetworking.iOS` projects.
4. Right-click the `XamFormsEfficientNetworking.WinPhone` project and choose **Manage NuGet Packages**. Find and install the `Microsoft.Net.Http` package.
5. There is no `modernhttpclient` specific Windows Phone project, but it is easy to support it. All we need is to create an empty `NativeMessageHandler` class derived from `HttpClientHandler`. The `modernhttpclient` library will handle picking it up and using it in runtime. Right-click the project and **Add | Class…**, name it `NativeMessageHandler`, and click **Add**. Find the class details next:

   ```
   public class NativeMessageHandler : HttpClientHandler
     {
     }
   ```

6. Create a folder named `Models`. Right-click `XamFormsEfficientNetworking`, **Add | New Folder**.
7. Right-click the newly created folder `Models` and choose **Add | Class…**. Give it the name `Order.cs` and click **Add**. The following is the simple implementation:

   ```
   public class Order
   {
     public string ObjectId { get; set; }
     public string OrderNumber { get; set; }
     [JsonIgnore]
     public List<Item> Items { get; set; }
   }
   ```

8. Right-click the `Models` folder and choose **Add | Class…**. Name it `Item.cs` and click **Add**. Find the implementation details in the following snippet:

   ```
   public class Item
   {
     public string ObjectId { get; set; }
     public string OrderNumber { get; set; }
     public string Code { get; set; }
   }
   ```

9. Right-click the portable class library and create a folder named `Views` with **Add |**

New Folder.

10. Right-click the newly created `Views` folder and choose **Add | New Item…**. From the templates, choose **Forms Xaml Page**, name it `MainPage.xaml`, and click Add.

Note

The XAML code for `MainPage` is omitted for brevity. Please refer to the recipe's code to see the `ListView` implementation or refer to Chapter 8, *A List to View* to learn how to work with the `ListView` control.

11. Go to the `App.cs` constructor and change the `MainPage` property assignment to the following:

```
MainPage = new NavigationPage(new MainPage());
```

12. Right-click the `Models` folder and create a class named `DataService.cs`. This is the class where all the interesting things are happening. Check the following implementation details:

```
using ModernHttpClient;
using Newtonsoft.Json;
using Newtonsoft.Json.Linq;
using Newtonsoft.Json.Serialization;
using Punchclock;
using System.Collections.Generic;
using System.Net.Http;
using System.Net.Http.Headers;
using System.Text;
using System.Threading.Tasks;

public class DataService
{
  private readonly OperationQueue _opQueue = new OperationQueue(2);
  private readonly JsonSerializerSettings _jsonSerializerSettings = new
JsonSerializerSettings
  {
    ContractResolver = new CamelCasePropertyNamesContractResolver()
  };
  const string OrdersUrl = "https://api.parse.com/1/classes/Order";
  const string ItemsUrl = "https://api.parse.com/1/classes/Item";

  private HttpClient GetClient()
  {
    HttpClient client = new HttpClient(new NativeMessageHandler());
    client.DefaultRequestHeaders.Add("X-Parse-Application-Id",
"YOUR_APPLICATION_KEY");
    client.DefaultRequestHeaders.Add("X-Parse-REST-API-Key",
"YOUR_API_KEY");
    client.DefaultRequestHeaders.Accept.Add(new
MediaTypeWithQualityHeaderValue("application/json"));
    return client;
  }

  public async Task<IEnumerable<Order>> GetAllOrdersAsync(int skip = 0)
  {
```

```
    using (HttpClient client = GetClient())
    {
      Dictionary<string, string> values = new Dictionary<string,
string>()
      {
        { "skip", skip.ToString() }
      };
      FormUrlEncodedContent encodedContent = new
FormUrlEncodedContent(values);
      string param = await
encodedContent.ReadAsStringAsync().ConfigureAwait(false);
      HttpResponseMessage httpResponseMessage = await
_opQueue.Enqueue(10, () => client.GetAsync(OrdersUrl + "?" +
param)).ConfigureAwait(false);
      httpResponseMessage.EnsureSuccessStatusCode();
      string content = await
httpResponseMessage.Content.ReadAsStringAsync().ConfigureAwait(false);
      JObject jsonObject = JObject.Parse(content);
      string ordersJson = jsonObject["results"].ToString();
      IEnumerable<Order> orders =
JsonConvert.DeserializeObject<IEnumerable<Order>>(ordersJson);
      List<Task> tasks = new List<Task>();
      foreach (Order order in orders)
      {

tasks.Add(GetItemsByOrderNumberAsync(order.OrderNumber).ContinueWith((a
ntecedent) =>
        {
           order.Items = new List<Item>(antecedent.Result);
        }));
      }
      await Task.WhenAll(tasks).ConfigureAwait(false);
      return orders;
    }
  }

  public async Task<IEnumerable<Item>>
GetItemsByOrderNumberAsync(string orderNumber)
  {
    using (HttpClient client = GetClient())
    {
      string orderNumberJson = "{\"orderNumber\": \"" + orderNumber +
"\"}";
      Dictionary<string, string> values = new Dictionary<string,
string>()
      {
        { "where", orderNumberJson }
      };
      FormUrlEncodedContent encodedContent = new
FormUrlEncodedContent(values);
      string param = await
encodedContent.ReadAsStringAsync().ConfigureAwait(false);
      HttpResponseMessage httpResponseMessage =
      await _opQueue.Enqueue(15, () => client.GetAsync(ItemsUrl + "?" +
param)).ConfigureAwait(false);
      httpResponseMessage.EnsureSuccessStatusCode();
```

```
      string content = await
    httpResponseMessage.Content.ReadAsStringAsync().ConfigureAwait(false);
      JObject jsonObject = JObject.Parse(content);
      string ordersJson = jsonObject["results"].ToString();
      IEnumerable<Item> items =
    JsonConvert.DeserializeObject<IEnumerable<Item>>(ordersJson);
      return items;
    }
  }

  public async Task<Order> InsertOrderAsync(Order order)
  {
    using (HttpClient client = GetClient())
    {
      HttpResponseMessage httpResponseMessage = await
    _opQueue.Enqueue(1, () => client.PostAsync(OrdersUrl,
      new StringContent(JsonConvert.SerializeObject(order,
    _jsonSerializerSettings),
      Encoding.UTF8, "application/json"))).ConfigureAwait(false);
      httpResponseMessage.EnsureSuccessStatusCode();
      string content = await
    httpResponseMessage.Content.ReadAsStringAsync().ConfigureAwait(false);
      Order newOrder = JsonConvert.DeserializeObject<Order>(content);
      List<Task> tasks = new List<Task>();
      foreach (Item item in order.Items)
      {
        tasks.Add(InsertItemAsync(item));
      }
      await Task.WhenAll(tasks).ConfigureAwait(false);
      return newOrder;
    }
  }

  public async Task<Item> InsertItemAsync(Item item)
  {
    using (HttpClient client = GetClient())
    {
      HttpResponseMessage httpResponseMessage = await
    _opQueue.Enqueue(3, () => client.PostAsync(ItemsUrl,
      new StringContent(JsonConvert.SerializeObject(item,
    _jsonSerializerSettings), Encoding.UTF8,
    "application/json"))).ConfigureAwait(false);
      httpResponseMessage.EnsureSuccessStatusCode();
      string content = await
    httpResponseMessage.Content.ReadAsStringAsync().ConfigureAwait(false);
      return JsonConvert.DeserializeObject<Item>(content);
    }
  }
}
```

13. Go to `MainPage.xaml.cs` and add the following code:

```
private readonly DataService _dataService = new DataService();

  public MainPage()
  {
```

```
    InitializeComponent();
  }
  private void OnRefreshClick(object sender, EventArgs e)
  {
    RefreshAsync();
  }

  protected override void OnAppearing()
  {
    base.OnAppearing();
   //RandomOrdersHelper.InsertOrdersAsync(100);
  // Add some records to the remote database
  }

  private Task RefreshAsync()
  {
    int skip = BindingContext != null ?
((IEnumerable<Order>)BindingContext).Count() : 0;
    return
_dataService.GetAllOrdersAsync(skip).ContinueWith((antecedent) =>
    {
      Device.BeginInvokeOnMainThread(() =>
      {
        if (BindingContext == null)
        {
          BindingContext = new ObservableCollection<Order>
(antecedent.Result);
        }
        else
        {
          foreach (Order item in antecedent.Result)
          {
            ((ObservableCollection<Order>)BindingContext).Add(item);
          }
        }
      });
    }, TaskContinuationOptions.OnlyOnRanToCompletion);
  }
```

14. You need some records. The recipe code has a helper method to insert random Order and Item records to your database. Refer to this book's code to see the implementation, uncomment the code line in the `OnAppearing` method, and you will start creating some records. You can also apply your own dummy insert class; it's up to you!
15. After adding some records, this is what the application will look like, if you followed the random `OrderNumber` implementation of this book's code:

Android:

11:03
REFRESH
25766
88245
16872
51780
84641
56462
78882
15842
18614

iOS:

Windows Phone:

11:02
25766
88245
16872
51780
84641
56462
78882
15842
18614

How it works…

Starting with the models, we use two POCO classes with simple properties: Order and Item. An Order has many Items that are ignored when serialized in JSON. Depending on your architecture, this is something you may or may not want. If you use a document database and you want a JSON field in your Order table, it's fine; for this recipe we used a separate table for the Item data, as it gives us the opportunity to make more calls for demonstration purposes.

In `MainPage.xaml.cs`, the important method is `RefreshAsync`. Here, we check if we have loaded any orders. If yes, call the `DataService.GetAllOrdersAsync(Skip)` method with the count of the list items or 0 if there are no items yet fetched. The continuation delegate will run only on successful completion of the Task. In the completion action, we check if `BindingContext` is null and if true, create an `ObservableCollection<Order>` with the results; if `false,` we append items in the collection.

All the efficiency and handling of concurrent calls happen in the interesting `DataService` class. Let's examine why this class handles network requests better as if we are using the default configuration.

In the `GetClient` method, we create and return an `HttpClient` instance; it's a factory method we use for each `HttpClient` call because we immediately dispose the instance. In this method, we use the `ModernHttpClient` NuGet package `NativeMessageHandler` class and pass an instance of the handler in the `HttpClient` constructor. What essentially this handler does is to bypass the default `HttpClient` web protocol calls on each platform and make use of two native libraries: `NSURLSession` in the iOS platform and `OkHttp` in the Android platform. There is no specific `ModernHttpClient` Windows Phone handler implementation, but it is easy to instruct the library to pick up our custom `NativeMessageHandler` by creating a `HttpClientHandler` derived class. This makes our calls drastically faster!

`GetAllOrdersAsync` will fetch a default limit of 100 records from the remote database. We also have an optional parameter where you can skip records and fetch the next 100. Pagination is something you should also consider when creating apps that load large lists of data. We make the `HttpClient.GetAsync` call, deserialize the list of orders, and fetch the corresponding order items calling `GetItemsByOrderNumber`, in parallel.

Notice the wrapper of `OperationQueue`, `_opQueue.Enqueue<T>`(Priority, and `Func<Task<T>>`) around `HttpClient.GetAsync`. This is a class in the `Punchclock` library, the orchestrator that will schedule and prioritize our network requests via a priority integer parameter. When we created the `OperationQueue` instance, we set the maximum calls at two at a time. In this method, we fetch the orders with a priority of 10, then in the same method we make multiple calls of `GetItemsByOrderNumberAsync`, which consequently issues internally another `HttpClient.GetAsync` method wrapped in `OperationQueue.Enqueue` has a higher priority than `GetAllOrdersAsync`, meaning whatever is happening in the queue, I will cut in to get priority, if of course there's no equivalent or higher-priority items in the queue.

The `InsertOrderAsync` and `InsertItemAsync` methods have low priority. This was for our recipe demonstration purposes, but in your application based on your requirements, you might want inserts to cut in the queue while fetching data requests.

With these two simple changes in our `DataService`, we have major performance increase and queue management of network requests, which gives us powerful handling of priority and parallel execution of the number of simultaneous calls.

There’s more…

There are a couple of other libraries from Paul Betts that you might find interesting:

Fusillade, https://github.com/paulcbetts/fusillade, inspired by Volley, an Android asynchronous HTTP requests library, and Picasso, image downloading and caching for Android.

Refit, https://github.com/paulcbetts/refit, a type-safe REST library inspired by Retrofit, HTTP client for Android and Java. You can decorate an interface with attributes and it will automatically generate a class implementation for this interface.

It is worth noting Flurl, https://tmenier.github.io/Flurl/, created by Todd Menier, a fluent URL builder and testable HTTP library. This will simplify your `HttpClient` constructs in a fun and simple fluent way.

There is not one solution to rule them all, but you can install and mix and match for every architecture that suits your applications best!

Chapter 6. One for All and All for One

In this chapter, we will cover the following recipes:

- Creating cross-platform plugins
- Taking or choosing photos
- Querying the GPS location
- Querying the OS contacts

Introduction

With Xamarin.Forms you can also create a cross-platform UI and customize it or mix and match native-Xamarin.Forms. Amazing!

That’s all good, but you can’t access the native platform features from shared code and in our shared code we have all our POCO data objects, networking service classes, data access components, Models, ViewModels, and business logic. If you try to reference shared code in your native code and start applying logic then you start repeating code between platforms, messing with spaghetti code, and that leads to complex, non-readable, and hard to debug code.

All platforms have common capabilities. Android, iOS, and Windows have battery, GPS, notifications, settings, Bluetooth, text to speech, and maps, but all are used with their corresponding native APIs. What are the solutions? Abstractions, service location, and dependency injection! You program against an interface where the implementation is loaded in runtime for each platform.

In the first recipe, *Creating cross-platform plugins*, we will explore a technique called Bait and Switch used by almost all the plugins out there, at least most of those created by Xamarin, and it’s an exciting way of thinking when programming cross-platform applications.

In the remaining three recipes, we will be using some of the currently available plugins to demonstrate how simple it is to produce your forms app with help from the community. It is worth bearing in mind that as with Xamarin.Forms, plugins will change over time and therefore you may have to alter the examples to enable the code to compile.

Creating cross-platform plugins

Okay, you're tired of repeating the same copy and paste code in every project you create for this native implementation that there is no available plugin from Xamarin or a Xamarin developer out there! You create your interface, create the implementation classes, register to a DI container or just a service locator, and resolve the dependencies in runtime. It works!

Now, for a minute imagine creating your own plugin. If the client remembers to register your plugin with your implementation for each platform then everything's OK, and you will be depended in a custom internal or third-party service locator and DI container? Doesn't sound very practical, right?

Yes, there is a better way to do things. A good friend named it *Bait and Switch PCL*, and Miguel De Icaza from Xamarin has called it the *Advanced PCL* pattern.

What this pattern essentially solves is the process of creating interface abstractions to program against and injecting implementation classes conforming to the interface in your application via Dependency Injection or a Service Locator.

The importance of this pattern is not to think of a PCL as an entity on its own, so let's see some facts to understand this better.

- There is no such thing as a portable application.
- The platform that the PCL profile defines doesn't actually exist. An application will always run under a profile with more features
- A critical aspect of NuGet for the pattern to work is that it will always prefer a platform library to a PCL. Meaning that if you write a NuGet dll for each platform, NuGet will always choose the platform-specific one when added to the project.
- The switch of the assembly is accomplished via method invocation, which means, as long as the platform binary matches the assembly name, version, and class structure, we can replace it with the platform binary.

Let's see how to do it.

How to do it…

1. In Visual Studio, go to the top menu, select **Tools | Extensions and Updates** and search for `plugin for Xamarin templates`; select and click **Download** and then **Close**.

2. Select **File | New | Project…** and find the template plugin for Xamarin; give it the name `XamFormsCrossPlugin` and click **OK**.

3. Right-click the solution and select **Add | New Solution Folder**; give it the name `Client`.
4. Right-click the newly created solution folder `Client` and choose **Add | New Project…**, select the template **Blank App** (Xamarin.Forms Portable), name it `XamFormsPluginClient` and click **OK**.
5. Go to the `Plugin.XamFormsCrossPlugin.Abstractions` library, open the `IXamFormsCrossPlugin.cs` file and add one method signature in the `IXamFormsCrossPlugin` interface.

```
string PlatformHelloWorld();
```

6. In the `Plugin.XamFormsCrossPlugin` portable library, notice that the `CrossXamFormsCrossPlugin.cs` class file is linked to every native platform project library. Inside there is a lazy property that creates a new `XamFormsCrossPluginImplementation` class if the platform is not `PORTABLE` or else `null`.
7. Go to `Plugin.XamFormsCrossPlugin.Android`, open the `XamFormsCrossPluginImplementation` and implement the `IXamFormsCrossPlugin` interface.

```
public class XamFormsCrossPluginImplementation : IXamFormsCrossPlugin
  {
    public string PlatformHelloWorld()
    {
```

```
      return "Hello from Android";
    }
}
```

8. Repeat step 7 for `Plugin.XamFormsCrossPlugin.iOSUnified`.

```
public class XamFormsCrossPluginImplementation : IXamFormsCrossPlugin
  {
    public string PlatformHelloWorld()
    {
      return "Hello from iOS";
    }
}
```

9. And for `Plugin.XamFormsCrossPlugin.WindowsPhone8`:

```
public class XamFormsCrossPluginImplementation : IXamFormsCrossPlugin
  {
    public string PlatformHelloWorld()
    {
      return "Hello from Windows Phone 8";
    }
}
```

10. Go to the `Client` solution folder in the `XamFormsPluginClient` portable class library and right-click the `References` folder. Choose **Add Reference** and in the **Projects | Solution** section, check the `Plugin.XamFormsCrossPlugin` and `Plugin.XamFormsCrossPlugin.Abstractions` libraries and click **OK**.

11. Open `App.cs` and in the constructor, replace the assignment of the `Text` property in the `ContentPage` initialization.

    ```
    Text = CrossXamFormsCrossPlugin.Current.PlatformHelloWorld()
    ```

12. Right-click the `References` folder in `XamFormsPluginClient.Droid`, choose **Add Reference** and in the **Project | Solution** section check the `Plugin.XamFormsCrossPlugin.Android` and `Plugin.XamFormsCrossPlugin.Abstractions` libraries and click **OK**.
13. Repeat step 12 for `XamFormsPluginClient.iOS` and check the `Plugin.XamFormsCrossPlugin.iOSUnified` and `Plugin.XamFormsCrossPlugin.Abstractions` libraries.
14. Repeat step 12 for `XamFormsPluginClient.WinPhone` and this time add the `Plugin.XamFormsCrossPlugin.WindowsPhone8` and `Plugin.XamFormsCrossPlugin.Abstractions` libraries.
15. Run the application for each platform and admire your awesome new feature of printing a hello message for every platform.

iOS:

Carrier
10:02 PM
Hello from iOS

Android:

10:03
Hello from Android

Windows Phone:

How it works…

Do you need to create a plugin that works in various platforms? No problem. James Montemagno from Xamarin created this helper template based on the Bait and Switch pattern to make development rapid.

The template creates two portable class libraries. The abstraction one contains our core shared types like interfaces, enums, and any constants or models dependencies; in our case, we only have one interface, `IXamFormsCrossPlugin`. The second portable contains only one class, the bait, `CrossXamFormsCrossPlugin`, that lazy creates an instance of our class implementing the `IXamFormsCrossPlugin` interface, `XamFormsCrossPluginImplementation`, the switch!

We have access to the platform instance from the current static property where if the `Implementation.Value` is `null` it throws a `NotImplementedException`. You can also see where the switch is happening with an `#ifdef` conditional check in the `CreateXamFormsCrossPlugin` method, if `PORTABLE` then returns `null` or else the native platform implementation.

How is the switch happening? In the five native platform libraries we get from the template, there is a link to the `CrossXamFormsCrossPlugin.cs` file and a `XamFormsCrossPluginImplementation.cs` file for each native platform library.

Well, that saved us from some complexity. It might seem confusing in the beginning, but if you explore the code for some time it makes a lot of sense. Open `XamFormsCrossPluginImplementation` of the `Plugin.XamFormsCrossPlugin.Android` library; notice that the instance creation of a class is now highlighted, so when this library is compiled the native class implementation is exposed to the platform project even if we reference the PCL and work from our core shared library with the abstractions. In runtime, the platform-specific implementation will be used.

Take note that the namespace, version, and assembly name are the same for all the native platform libraries and the portable library; this is key for the switch to work.

In the end, we reference the abstractions and portable library to our core portable client library, and for every platform the native equivalent library and the abstractions library.

There’s more…

You’re almost ready to ship your cool plugin! But you need to package it and upload it to NuGet. No problem. Xamarin has a template for this too.

- Right-click the solution and **Add | New Item…**, find the template plugin for **Xamarin NuSpec** and name it exactly the same with our plugin solution. In the recipe, the name is `XamFormsCrossPlugin`. Click **Add**.

That’s all you need. The Nuspec is ready; edit as needed if you removed or added any supported platforms and then build the plugin in release mode to get ready to ship your cross-platform Xamarin plugin.

See also

- https://developer.xamarin.com/guides/cross-platform/advanced/nuget/

Taking or choosing photos

It’s very common to work with the camera in applications: you want to take a photo or video, post on social media, or just keep it as records, from expenses to business cards.

For this recipe, we will use a Xamarin plugin to simplify the process and achieve the goal of using the camera for capturing photos and videos with a couple of code lines.

How to do it…

1. Create a **Blank App** (Xamarin.Forms Portable) solution from **File | New | Project…**, named `XamFormsTakingPhotos` and click **OK**.
2. For each project created, we will add the `Xam.Plugin.Media` NuGet package. Right-click the projects one by one, choose **Manage NuGet Packages** and search and install the `Xam.Plugin.Media` package.
3. Three permissions are required in the Android Manifest. These are added automatically when installing the NuGet package, but just check to make sure. Right-click on the Android package and choose Properties, select the **Android Manifest** tab and check for the following permission:
   - `WRITE_EXTERNAL_STORAGE`
   - `READ_EXTERNAL_STORAGE`
   - `CAMERA`

4. Go to the Windows Phone platform, expand the **Properties** folder and double-click `WMAppManifest.xml`. Select the **Capabilities** tab and check the `ID_CAP_ISV_CAMERA` to allow camera access.
5. The Android project should contain a `MainApplication.cs` file. If not, make sure to upgrade to the latest Xamarin.Forms NuGet package. Open `MainApplication.cs` and in the `OnCreate` method, add the following code to register the dependency:

   ```
   Xamarin.Forms.DependencyService.Register<MediaImplementation>();
   ```

6. Add the same line of code from step 5 in the `AppDelegate.cs` of the iOS platform after the `Forms.Init` method.
7. Repeat the same for the Windows Phone platform in the `MainPage.xaml.cs` behind code after the `Forms.Init` method.
8. Go to the `XamFormsTakingPhotos` PCL, right-click and **Add** | **New Item…**; select **Forms Xaml Page**, name it `MainPage.xaml` and click **Add**.
9. Replace the ContentPage tag contents with the following:

   ```
   <StackLayout HorizontalOptions="Center" VerticalOptions="Center">
       <Button Text="Take Photo" Clicked="OnTakePhotoButtonClicked"/>
       <Button Text="Take Video" Clicked="OnTakeVideoButtonClicked"/>
       <Image x:Name="imagePhoto" />
   </StackLayout>
   ```

10. Open `App.cs` and in the constructor, replace the `MainPage` property assignment with the following code:

    ```
    MainPage = new MainPage();
    ```

11. Go to `MainPage.xaml.cs` and at the top of the file, add a field to retrieve the `IMedia` implementation.

    ```
    private readonly IMedia Media = DependencyService.Get<IMedia>();
    ```

12. To capture a photo, we will add the corresponding `OnTakePhotoButtonClicked` event handler.

    ```
    async void OnTakePhotoButtonClicked(object sender, EventArgs args)
      {
        if (!Media.IsCameraAvailable ||
        !Media.IsTakePhotoSupported)
        {
          DisplayAlert("No Camera", "No camera available.", "OK");
          return;
        }

        MediaFile file = await Media.TakePhotoAsync(new
    StoreCameraMediaOptions
        {
          Directory = "Sample",
          Name = "test.jpg"
        });
    ```

```
    if (file != null)
    {

    ImageSource imageSource = ImageSource.FromStream(() =>
    {
      var stream = file.GetStream();
      return stream;
    });
    imagePhoto.Source = imageSource;
    Debug.WriteLine("Photo File Path: {0}", file.Path);
    file.Dispose();
  }
}
```

13. To capture a video, add the `OnTakeVideoButtonClicked` event handler:

```
async void OnTakeVideoButtonClicked(object sender, EventArgs args)
  {
    if (!Media.IsCameraAvailable ||
    !Media.IsTakeVideoSupported)
    {
      DisplayAlert("No Camera", ":( No camera available.", "OK");
      return;
    }

    MediaFile file = await Media.TakeVideoAsync(new StoreVideoOptions
    {
      Directory = "Sample",
      Name = "test.mp4"
    });

    if (file != null)
    {

    ImageSource imageSource = ImageSource.FromStream(() =>
    {
      var stream = file.GetStream();
      return stream;
    });
    Debug.WriteLine("Video File Path: {0}", file.Path);
    file.Dispose();
  }
}
```

14. The app is now ready to take photos and videos. With this plugin, your Xamarin app is created easily. Compile the code and test it. You need an Android emulator that supports a camera or a physical device, an iOS device, and the Windows Phone emulator or a device to make sure it will work.

iOS:

T-Mobile
12:10
71%
Take Photo
Take Video
esc
F1
F2
F3
F4
F5
F6
Q
W
E
R
T
A
S
D
F
G
Z
X
C
V
alt
control
option
command

Windows Phone:

How it works…

You don’t have to reinvent the wheel or write custom code with renderers for every case. In this recipe, with the use of `Xam.Plugin.Media`, we can take or choose photos easily with minimal code.

First, you install the `Xam.Plugin.Media` NuGet package and add the required permissions for Android and the capabilities in Windows Phone, and then you register the dependency in every native platform.

iOS has required permissions but it is automatically requested in code from the plugin implementation. The user has to only allow the use of the camera at that point of action.

In our simple XAML user interface, we have two buttons: one for taking a photo and one for recording a video.

In the behind-code `OnTakePhotoButtonClicked` event handler, we check if a camera is available and taking a photo is supported and invoke the `IMedia.takePhotoAsync` method passing a `StoreCameraMediaOptions` instance setting the photo directory and filename. When the user returns to the page, we create an `ImageSource` from the returned `MediaFile` and the method `GetStream` and assign it to the `Image.Source` view property.

Taking a video uses the exact same steps, only we check the property `IsTakeVideoSupported` and invoke the `IMedia.TakeVideoAsync` method passing a `StoreVideoOptions` instance. This returns a `MediaFile`, which we can use to get the stream.

There are two ways to access the plugin: from the static `CrossMedia.Current` property or registering the interface and program against it. To keep the best practices of cross-platform mobile development, we program against the interface.

There’s more…

We have demonstrated how to take photos. You can also choose photos from the device library. If you examine the `IMedia` signature or go to the GitHub `Xam.Plugin.Media` source code (see the following link), you can find out the `IsPickPhotoSupported`, `IsPickVideoSupported`, `PickPhotoAsync`, and `PickVideoAsync` methods. It is repetitive code like our example, only replacing the conditional checks and the method call.

See also

- https://github.com/jamesmontemagno/Xamarin.Plugins/tree/master/Media

Getting the GPS location

Getting and monitoring the GPS location for each platform is a common capability. Every smartphone in the market right now has a GPS sensor built in, but each OS has its own native APIs to retrieve the position.

The good news is that you don’t have to create a plugin as there is one already created by James Montemagno from Xamarin that you can freely use.

How to do it…

1. Create a **Blank App** (Xamarin.Forms Portable) cross-platform mobile solution named `XamFormsQueryGps` in Visual Studio with **File** | **New** | **Project…**.
2. Right-click all the projects one by one and choose **Manage NuGet Packages** to search and install `Xam.Plugin.Geolocator`.
3. Go to the `XamFormsQueryGps` portable library and open the `App.cs` file; add the following code to the file:

```
Label gpsLabel;

public App()
{
  IGeolocator locator = DependencyService.Get<IGeolocator>();
  locator.DesiredAccuracy = 50;
  locator.AllowsBackgroundUpdates = true;
  locator.StartListening(1, 1, true);
  locator.PositionChanged += LocatorPositionChanged;

  Button button = new Button
  {
    Text = "Get Position!"
  };
  button.Clicked += async (sender, e) => await
GetCurrentLocationAsync(locator);
  gpsLabel = new Label
  {
    Text = "GPS Coordinates"
  };

  // The root page of your application
  MainPage = new ContentPage
  {
    Content = new StackLayout
    {
      VerticalOptions = LayoutOptions.Center,
      Children = {
        gpsLabel,
        button
      }
    }
  };
}

async Task GetCurrentLocationAsync(IGeolocator locator)
{
  Position position = await locator.GetPositionAsync(10000);
  PrintPosition(position);
}

private void LocatorPositionChanged(object sender, PositionEventArgs e)
  {
    PrintPosition(e.Position);
}
```

```
private void PrintPosition(Position position)
{
  gpsLabel.Text = string.Format(
  "Time: {0} \n" +
  "Lat: {1} \n" +
  "Long: {2} \n " +
  "Altitude: {3} \n" +
  "Altitude Accuracy: {4} \n" +
  "Accuracy: {5} \n " +
  "Heading: {6} \n " +
  "Speed: {7}",
  position.Timestamp, position.Latitude, position.Longitude,
  position.Altitude, position.AltitudeAccuracy, position.Accuracy,
  position.Heading, position.Speed);
}
```

4. In the Android project, we have to add two permissions. Right-click `XamFormsQueryGps.Droid` and choose **Properties**. In the required permissions section, add the following:
   - `ACCESS_COARSE_LOCATION`
   - `ACCESS_FINE_LOCATION`

   ### Note

   With Android 6.0 Marshmallow, you will have to ask for Fine and Course Location runtime permission: https://blog.xamarin.com/requesting-runtime-permissions-in-android-marshmallow/.

5. Open `MainActivity.cs` and after the `Forms.Init` method call in the `OnCreate` method, register the `GeolocatorImplementation` type.

   ```
   DependencyService.Register<GeolocatorImplementation>();
   ```

6. For the iOS to support background updates, meaning we will receive a notification even if our application is not currently in the foreground, go to `XamFormsQueryGps.iOS`. We need to make some changes in the `Info.plist` file. Double-click the file, go to the **Background Modes** tab section, check **Enable Background Modes** and check the **Location update** too.

7. We next need to add two new keys to the `Info.plist` file. If you are using Xamarin Studio, this can be achieved by double-clicking the `Info.plist` file, selecting **Advanced** and then selecting the add new key icon. If you are using Visual Studio, you can edit the file using a text editor (such as notepad) or the built-in source code editor. To do this, right-click the `Info.plist` file and choose **Open With…**, select **Source Code (Text) Editor With Encoding** and click **OK**. Add the following key value pairs inside the `<dict>` tag:

```
<key>NSLocationWhenInUseUsageDescription</key>
  <string>Need Location!</string>
  <key>NSLocationAlwaysUsageDescription</key>
<string>Need Location Always!</string>
```

8. Go to the `AppDelegate.cs` file and after the `Forms.Init` method call in the `FinishedLaunching` method, add the following dependency registration:

```
DependencyService.Register<GeolocatorImplementation>();
```

9. In the Window Phone project, expand the **Properties** folder, double-click the `WMAppManifest.xml` file, go to the **Capabilities** section tab and check the `ID_CAP_LOCATION` capability.
10. Open the `MainPage.xaml.cs` file and after the `Forms.Init` method call in the constructor, add the dependency registration of the plugin.

```
DependencyService.Register<GeolocatorImplementation>();
```

11. Run the application to the simulator or device and you will start receiving updates. Pressing the **Get Location** button will refresh the values as well. With Xamarin Android player, you can tap the settings icon and change the location. With iOS

simulator, you can go to the **Debug** menu and in **Location**, choose a location; in the Windows Phone emulator, you are able to change the location by expanding the tools.

iOS:

Carrier 1:26 PM

Time: 12/6/2015 6:26:00 PM +00:00
Lat: 50.785834
Long: 122.406417
Altitude: 0
Altitude Accuracy: 0
Accuracy: 5
Heading: 0
Speed: 23.81

Get Position!

Android:

1:24
Time: 12/7/2015 4:24:24 AM +00:00
Lat: 50.7976983333333
Long: 122.402
Altitude: 7.345
Altitude Accuracy: 0
Accuracy: 5
Heading: 359.989990234375
Speed: 0
GET POSITION!

Windows Phone:

How it works…

Each plugin has a `GeolocatorImplementation` class. In that file, the plugin utilizes the platform native API to provide location updates. The majority of Xamarin plugins that you will see follow the Bait and Switch technique. To learn how to create Xamarin cross-platform plugins, go to the section *Creating cross-platform plugins*.

The Android platform is using `LocationManager`, in iOS the `CLLocationManager` API, and Windows Phone `GeoCoordinateWatcher`. Of course, the full implementation has more detail. You can refer to the source code and explore how each platform works.

We can access the plugin through a static property, the `CrossGeolocator.Current` property, but as a best practice we register the dependency implementation class and request the interface from the core PCL library using `DependencyService`.

Having an `IGeolocator` interface implementation instance, we set `DesiredAccuracy` to `50`. `DesiredAccuracy` provides more granularity and control of the accuracy of the position results; the higher the value, the more your application will require the most accurate data available. Since we want to allow background updates, we set `AllowsBackgroundUpdates` to `true`, invoke the `StartListening` method, and register an event handler for the `PositionChanged` event.

Our UI is very simple, with a Label showing the GPS position details when the `LocatorPositionChanged` event handler is invoked or if you request it when clicking the **Get Position!** button

See also

- https://github.com/jamesmontemagno/Xamarin.Plugins/tree/master/Geolocator

Show and schedule local notifications

Notifications are everywhere in today’s mobile applications. In our example, for this recipe we will explore how to use a cross-platform NuGet package and minimize the amount of code to show local notifications to the user.

How to do it…

1. Open Visual Studio and create a **Blank App** (Xamarin.Forms Portable) cross-platform solution from the top menu, **File | New | Project…**, named `XamFormsLocalNotifications`.
2. For each native platform project, right-click and choose **Manage NuGet Package**; search and install the `Xam.Plugins.Notifier` NuGet package and click **Install**.
3. Go to the `App.cs` file in the `XamFormsLocalNotifications` class library and replace the constructor with the following code:

```
public App()
{
  ILocalNotifications localNotifications =
DependencyService.Get<ILocalNotifications>();

  Button showNotificationButton = new Button();
  showNotificationButton.Text = "Show Local Notification";
  showNotificationButton.Clicked += (sender, e) =>
localNotifications.Show("Test", "Local notification alert", 1);

  Button cancelNotificationButton = new Button();
  cancelNotificationButton.Text = "Cancel Local Notification";
  cancelNotificationButton.Clicked += (sender, e) =>
localNotifications.Cancel(1);

  MainPage = new ContentPage
  {
    Content = new StackLayout
    {
      VerticalOptions = LayoutOptions.Center,
      Children = {
        showNotificationButton,
        cancelNotificationButton
      }
    }
  };
}
```

4. Open `MainActivity.cs` in the `XamFormsLocalNotifications.Droid` project and register the plugin implementation dependency after the `Forms.Init` method call.

```
DependencyService.Register<LocalNotificationsImplementation>();
```

5. Go to `AppDelegate.cs` in the `XamFormsLocalNotifications.iOS` project and in the `FinishedLaunching` method, add the same dependency plugin registration from step 5 after the `Forms.Init` method call.
6. For the iOS platform, an extra step is required to request access for showing notifications. In `AppDelegate.cs`, add the following code in the `FinishedLaunching` method:

```
var settings = UIUserNotificationSettings.GetSettingsForTypes(
    UIUserNotificationType.Alert
    | UIUserNotificationType.Badge
```

```
        | UIUserNotificationType.Sound,
        new NSSet());
    UIApplication.SharedApplication.RegisterUserNotificationSettings(settin
    gs);
```

Run the applications and press the button **Show Local Notification** to show a notification.

iOS:

Carrier
8:31 PM
Today
Notifications
Today
XamFormsLocalNotifications
now
Local notification alert

Android:

8:15 PM
Monday, December 7
Test
Local notification alert
8:15 PM
SHOW LOCAL NOTIFICATION

How it works…

Two lines of code and the local notification APIs for each platform are available at your command!

Starting from the cross-platform UI, we retrieved the `ILocalNotifications` interface implementation and created two buttons: one to show a notification and one to cancel it.

The `ILocalNotifications.Show` method accepts three arguments: title, description, and an ID. To cancel a notification, by invoking the `ILocalNotifications.Cancel` method, the ID has to match.

There's more…

`Xam.Plugins.Notifier` only supports Windows Phone 8.1+. By default, at the time of writing this book, creating a Visual Studio Xamarin.Forms solution will target the Windows Phone Silverlight 8.0 version. Follow the next steps to upgrade to Windows Phone Silverlight 8.1 and add local notification:

1. Right-click the `XamFormsLocalNotifications.WinPhone` project and select **Retarget to Windows Phone 8.1**.

2. `System.Windows.Interactivity` has to be included after this change to the project. Right-click and select `Add Reference`, go to the section **Assemblies | Extensions**, check the desired library and click **OK**.

3. Right-click the project and select **Manage NuGet Packages**. Click the **Browse** tab and search for `Xam.Plugins.Notifier`. Select it and click the **Install** button.
4. Open `MainPage.xaml.cs`. After the `global::Xamarin.Forms.Init() method`, add the registration of the `LocalNotificationsImplementation` Windows Phone

platform implementation.

5. `DependencyService.Register<LocalNotificationsImplementation>();`
6. Run the Windows Phone 8.1 application and click the **Show Local Notification** button. You will receive a message in the notification bar and on the top of the screen.

Chapter 7. Bind to the Data

In this chapter, we will cover the following recipes:

- Binding data in code
- Binding data in XAML
- Configuring two-way data binding
- Using value converters

Introduction

As a mobile developer, you create different types of applications and the majority will interact with data; they are data-driven. These apps display and manipulate data from a source, local database, filesystem, or from a remote server. From any type of persistent storage, you create classes that represent the data and in many cases transform and present them to a view. In Xamarin applications, the MVVM pattern is a common approach to design such applications. To learn more about this pattern, refer to Chapter 5, *Dude, Where's my Data?* and Chapter 4, *Different Cars, Same Engine*.

How are these data values presented to the view and also pushed back to the source (`ViewModel`, Model) when a value is changed? Traditionally, you might set the value to the UI control property and register to a value-changed event of the control to set the new value back to the model property. This might not sound terrifying, but when your application starts growing, maintenance and business logic changes can become a nightmare, and a small change will result in a chain of refactoring changes.

Data binding solves this problem with an intermediary object that will handle updating the value between the source (your model public property) and the target (a UI control property), creating a loose coupling between the components.

Binding data in code

This recipe introduces the concept of data binding: binding properties of an object (source) to the view controls (target) in the behind code of a page.

How to do it…

1. Start by creating a Visual Studio **Blank App** (Xamarin.Forms Portable) solution. In the top menu, click **File | New | Project…** and give it the name `XamFormsCodeBinding`.
2. In the `XamFormsCodeBinding` PCL library, right-click and choose **Add | Class…**; give it the name `Person.cs` and click **Add**.
3. Copy the following code to add two simple properties in `Person.cs`:

```
public class Person
{
  public string FirstName { get; set; }
  public string LastName { get; set; }
}
```

4. Right-click `XamFormsCodeBinding` PCL and **Add | New Item…**; choose **Forms Xaml Page**, name it `MainPage.xaml`, and click **Add**. In the `Content` tag, add the following code that adds two Label controls:

```
<StackLayout HorizontalOptions="Center" VerticalOptions="Center">
    <Label x:Name="firstNameLabel" />
    <Label x:Name="lastNameLabel" />
</StackLayout>
```

5. Go to `MainPage.xaml.cs` and change the constructor with the following code to bind a Person class instance:

```
public MainPage(Person person)
{
  InitializeComponent();

  Binding firstNameBinding = new Binding
  {
    Path = "FirstName",
    Source = person
  };
  firstNameLabel.SetBinding(Label.TextProperty, firstNameBinding);

  Binding lastNameBinding = new Binding
  {
    Path = "LastName",
    Source = person
  };
  lastNameLabel.SetBinding(Label.TextProperty, lastNameBinding);
}
```

6. Go to the `App.cs` constructor and change the code like the following; change the `FirstName` and `LastName` properties with your own information:

```
public App()
{
  Person person = new Person
  {
    FirstName = "George",
```

```
    LastName = "Taskos"
  };

  MainPage = new MainPage(person);
}
```

7. Ready to go! Running the application will present your name in the center of the screen. See the following iOS platform screenshot:

iOS:

Carrier
4:14 PM
George
Taskos

How it works…

To create a binding, we need three pieces:

- the Source
- the Path
- the Target

The Source can be any object, the Path is any public property of the object, and the Target has to be a `BindableProperty` in a `BindableObject` to bind with the source path.

In our example, the Source (a `Person` class instance) binds to two Label controls, the Target, and in the `MainPage.xaml.cs` behind code we create the Binding setting the Source and the Path; the Path is represented as a string. In runtime, Xamarin.Forms will use reflection to read the property value.

The source Path can be expressed in other ways too. If you want to bind to a navigation child property then you use `NavigationProperty.Child`. If you have a dictionary `Property[Key]`, with an array you use the `Property[Index]` syntax and there is the `.`binding path that references the object directly; in the recipe example, `.` will reference the person instance set as the Source and invoke the `ToString()` method for the presentation value.

The `SetBinding` method on the Label element associates our binding setup. We must set the Target property, which you can find as a static `BindableProperty` on the element class, and pass the binding we created.

Binding data in XAML

This recipe introduces the concept of data binding some object instance properties to view controls in XAML.

How to do it…

1. Start by creating a Visual Studio **Blank App** (Xamarin.Forms Portable) solution. In the top menu, click **File | New | Project…** and give it the name `XamFormsXamlBinding`.
2. In the `XamFormsXamlBinding` PCL library, right-click and choose **Add | Class…**; give it the name `Person.cs` and click **Add**.
3. Copy the following code to add two simple properties in `Person.cs`:

```
public class Person
{
  public string FirstName { get; set; }
  public string LastName { get; set; }
}
```

4. Right-click the `XamFormsCodeBinding` PCL and **Add | New Item…**; choose **Forms Xaml Page**, name it `MainPage.xaml`, and click **Add**. In the `Content` tag, add the following code that adds two `Label` controls to bind to `Person` properties directly in XAML:

```
<StackLayout HorizontalOptions="Center" VerticalOptions="Center">
    <Label Text="{Binding FirstName}" />
    <Label Text="{Binding LastName}" />
</StackLayout>
```

5. Go to `MainPage.xaml.cs` and change the constructor with the following code to set a `Person` class instance to the `BindingContext` property:

```
public MainPage(Person person)
{
  InitializeComponent();

  BindingContext = person;
}
```

6. Go to the `App.cs` constructor and change the code to the following; change the `FirstName` and `LastName` properties with your own information:

```
public App()
{
  Person person = new Person
  {
    FirstName = "George",
    LastName = "Taskos"
  };

  MainPage = new MainPage(person);
}
```

7. Run the application in your Android emulator or device. See the following screenshot:

Android:

4:00
George
Taskos

How it works…

In XAML, Xamarin.Forms simplified how we bind our UI controls to a source instance object with the `{Binding}` expression and with the help of `BindingContext`. The binding expression will create a Binding object and assign the required association. In our labels, the binding expressions assigned to the property Text, which is the Target property, and the Path are `FirstName` and `LastName`. XAML knows the source object instance because we set `BindingContext`.

A lot of times, you will have to bind an object to a page and create bindings for some properties and children of the instance. That would require you to create a lot of binding objects and assign them to the elements, but the Page and the UI controls classes have a shortcut instance property, `BindingContext`, which you can use to define the source of the binding in the level you want. In our case, we bind to the page level a Person instance, which makes our expressions search `BindingContext` and set the `FirstName` and `LastName` properties to the Target property.

Configuring two-way data binding

Binding data to a view via code or XAML is easy; nothing special is needed to present data from an object to the view. Binding a POCO class will only give you that though. Updating the source object and expecting to reflect the change to the target view or trying to push the values changed by the user from the target to the source will not work out of the box.

How to do it…

1. Start by creating a Visual Studio **Blank App** (Xamarin.Forms Portable) solution. In the top menu, click **File | New | Project…** and give it the name `XamFormsBindingModes`.
2. In the `XamFormsBindingModes` PCL library, right-click and choose **Add | Class…**; give it the name `Person.cs` and click **Add**.
3. We will create an extension method extending the `PropertyChangedEventHandler` delegate handler type. Right-click the portable class library again and choose **Add | Class…**; name the class `PropertyChangedEventHandlerExtensions.cs` and click **Add**.
4. Make the class `static` and add the following three helper extension methods:

```
public static class PropertyChangedEventHandlerExtensions
{
  public static void Raise<T>(this PropertyChangedEventHandler handler,
object sender, Expression<Func<T>> propertyExpression)
  {
    var body = propertyExpression.Body as MemberExpression;
    if (body == null)
    throw new ArgumentException("'propertyExpression' should be a
member expression");

    var expression = body.Expression as ConstantExpression;
    if (expression == null)
    throw new ArgumentException("'propertyExpression' body should be a
constant expression");

    handler.Raise(sender, body.Member.Name);
  }

  public static void Raise<T>(this PropertyChangedEventHandler handler,
object sender, params Expression<Func<T>>[] propertyExpressions)
  {
    foreach (var propertyExpression in propertyExpressions)
    {
      handler.Raise<T>(sender, propertyExpression);
    }
  }

  public static void Raise(this PropertyChangedEventHandler
propertyChangedHandler, object sender, [CallerMemberName] string
propertyName = "")
  {
    var handler = propertyChangedHandler;
    if (handler != null)
    {
      handler(sender, new PropertyChangedEventArgs(propertyName));
    }
  }
}
```

5. Go to the `Person.cs` class and implement the `INotifyPropertyChanged` interface.

We will add an extra method utilizing the `PropertyChangedEventHandlerExtensions` class we created earlier.

```
private string _firstName;
public string FirstName
{
  get
  {
    return _firstName;
  }
  set
  {
    SetFieldAndRaise(ref _firstName, value,
    () => FirstName, () => FullName);
  }
}

private string _lastName;
public string LastName
  {
  get
  {
    return _lastName;
  }
  set
  {
    SetFieldAndRaise(ref _lastName, value,
    () => LastName, () => FullName);
  }
}
public string FullName
{
  get
  {
    return string.Format("{0} {1}", FirstName, LastName);
  }
}

public event PropertyChangedEventHandler PropertyChanged;

protected virtual bool SetFieldAndRaise<T>(ref T field, T value, params
Expression<Func<T>>[] propertyExpressions)
  {
    if (EqualityComparer<T>.Default.Equals(field, value)) return false;
    field = value;
    PropertyChanged.Raise(propertyExpressions);
    return true;
}
```

6. Right-click the `XamFormsBindingModes` PCL and choose **Add | New Item…**. Select `Xaml Forms Page`, give it the name `MainPage.xaml`, and click **Add**.
7. Double-click `MainPage.xaml` and add the following code inside the `ContentPage` tag:

```
<StackLayout HorizontalOptions="FillAndExpand">
    <StackLayout.Padding>
```

```
      <OnPlatform x:TypeArguments="Thickness">
        <OnPlatform.iOS>
          0, 20, 0, 0
        </OnPlatform.iOS>
        <OnPlatform.Android>
          0, 0, 0, 0
        </OnPlatform.Android>
        <OnPlatform.WinPhone>
          0, 0, 0, 0
        </OnPlatform.WinPhone>
      </OnPlatform>
    </StackLayout.Padding>
    <Entry Text="{Binding FirstName, Mode=TwoWay}" Placeholder="First
name" />
    <Entry Text="{Binding LastName, Mode=TwoWay}" Placeholder="Last
name" />
    <Label Text="{Binding FullName}" HorizontalOptions="Center" />
</StackLayout>
```

8. Go to the `MainPage.xaml.cs` code-behind file and change the constructor to accept a `Person` parameter and setting the `BindingContext`.

```
public MainPage(Person person)
{
  InitializeComponent();

  BindingContext = person;
}
```

9. In `App.cs`, change the constructor to the following code:

```
public App()
{
  // The root page of your application
  MainPage = new MainPage(new Person());
}
```

10. Run the application, insert your first and last name, and you should see the full name label updating on the property has changed.

Android:

4:01
George
Taskos
George Taskos

iOS:

Carrier
8:49 AM
George
Taskos
George Taskos

Windows Phone:

How it works…

The important piece needed to accomplish two-way data binding is implementing the `INotifyPropertyChanged` interface and enable two-way binding mode scenarios.

The `INotifyPropertyChanged` interface has only one member, an event of delegate type `PropertyChangedEventHandler` named `PropertyChanged`. As the name implies, this event should be raised when a property value is changed and any component is registered to this event will be notified.

In this recipe, we took an extra step in the name of best practices and created an extension class for the `PropertyChangedEventHandler` type. The extension class has three methods, all named Raise but with different signatures.

The first Raise extension method accepts the sender that the event is raised and an `Expression<Func<T>>` parameter that we use to extract from the expression body the property name; with this practice, we avoid typos that may easily occur when dealing with string values.

The second method is a helper method that accepts the sender and a params of `Expression<Func<T>>` array iterating for each expression and call the first overload to extract the property name, which in turn calls the Raise overload that actually raises the event. This is very handy in cases like this recipe where we need to raise the `PropertyChanged` event for more than one property, `FirstName` and `FullName`.

Creating the extension methods makes our code reusable. You can have one or more base classes that implement `INotifyPropertyChanged` and use the methods to raise the `PropertyChanged` event easily using expressions for each property name.

In the `Person` class, we implement `INotifyPropertyChanged`, and add the protected virtual generic method `SetFieldAndRaise<T>` so that you can access it from derived classes and also provide your own version if needed. This is a practice that you can transfer to your base classes and all derived models/viewmodels can invoke in the properties setters.

The XAML code in the `MainPage.xaml` page is simple. We use `OnPlatform` in `StackLayout.Padding` to set `20` points from the top in the iOS platform. If we leave it at `0,` our UI controls will start from the very top edge of the device screen and the status bar will overlap with our controls.

We continue by setting the binding expressions for the Entry controls and also set the Mode property of Binding to `TwoWay`. For input controls such as Entry, this is the default value, but for code clarity and demonstration purposes we explicitly set it. For Label, the default value is `OneWay`, meaning one-way updates from the source to the target; we omitted setting it explicitly.

Last but not least, we need to set `BindingContext` to a Person instance passed as a parameter in the default constructor, one that we create in the `App.cs` constructor.

This chapter focuses on data binding using a custom object class instance as a source, but

with binding you can do more things in XAML. For example, View-To-View binding binds a slider's value property to a label's text property, or create animations while the value is changed.

You can have bindings to more than one view's properties, just by adding a binding expression for each property of the view.

`BindingContext` is available in the page level or you can set it up in the element level. An example would be to have an object `BindingContext` set in the page level and then set an object's property child to `BindingContext` of an element in the page.

There are also other ways to set `BindingContext`. One is using `StaticResource` or the `x:Static` markup extension.

See also

- https://developer.xamarin.com/guides/cross-platform/xamarin-forms/user-interface/xaml-basics/data_binding_basics/

Using value converters

Often when we consume data from the network or a local database, the values we retrieve aren’t always in the appropriate format for user presentation, such as dates as we demonstrate in this recipe example. You have a property type of `DateTime`, and you want to display it to the user’s screen in a detailed form such as `MM/dd/yyyy HH:mm:ss.fff`.

To achieve this in the following recipe, we are using a value converter.

How to do it…

1. Create a Visual Studio **Blank App** (Xamarin.Forms Portable) solution named `XamFormsValueConverter`, from the top menu **File | New | Project…**.
2. Right-click the `XamFormsValueConverter` portable class library and choose **Add | Class…**; name it `NotifyPropertyChangedExtension.cs` and click **Add**.
3. Make the newly created class static and add the contents like the following:

```
public static class NotifyPropertyChangedExtension
{
  public static void Raise<T>(this PropertyChangedEventHandler handler,
object sender, Expression<Func<T>> propertyExpression)
  {
    var body = propertyExpression.Body as MemberExpression;
    if (body == null)
    throw new ArgumentException("'propertyExpression' should be a
member expression");

    var expression = body.Expression as ConstantExpression;
    if (expression == null)
    throw new ArgumentException("'propertyExpression' body should be a
constant expression");

    handler.Raise(sender, body.Member.Name);
  }

  public static void Raise<T>(this PropertyChangedEventHandler handler,
object sender, params Expression<Func<T>>[] propertyExpressions)
  {
    foreach (var propertyExpression in propertyExpressions)
    {
      handler.Raise<T>(sender, propertyExpression);
    }
  }

  public static void Raise(this PropertyChangedEventHandler
propertyChangedHandler, object sender, [CallerMemberName] string
propertyName = "")
  {
    var handler = propertyChangedHandler;
    if (handler != null)
    {
      handler(sender, new PropertyChangedEventArgs(propertyName));
    }
  }
}
```

4. Right-click the PCL core library again and **Add | Class…** to create a model class named `Order.cs` and click **Add**. Insert the following code:

```
public class Order : INotifyPropertyChanged
{
  private DateTime _dateOrdered;
  public DateTime DateOrdered
```

```
  {
    get
    {
      return _dateOrdered;
    }
    set
    {
      SetFieldAndRaise(ref _dateOrdered, value);
    }
  }

  public event PropertyChangedEventHandler PropertyChanged;

  protected virtual bool SetFieldAndRaise<T>(ref T field, T value,
[CallerMemberName] string propertyName = "")
  {
    if (EqualityComparer<T>.Default.Equals(field, value)) return false;
    field = value;
    PropertyChanged.Raise(this, propertyName);
    return true;
  }
}
```

5. Right-click to add another class, **Add | Class…**, named `DateTimeToStringConverter.cs` and click **Add**. Add the following code to convert the values when the source to target or target to source value changed is triggered:

```
public class DateTimeToStringConverter : IValueConverter
{
  public object Convert(object value, Type targetType, object
parameter, CultureInfo culture)
  {
    DateTime dateValue = (DateTime)value;
    string stringDate = dateValue.ToString("MM/dd/yyyy HH:mm:ss.fff",
CultureInfo.InvariantCulture);
    Debug.WriteLine("DateTime to string: {0}", stringDate);
    return stringDate;
  }

  public object ConvertBack(object value, Type targetType, object
parameter, CultureInfo culture)
  {
    DateTime dateValue;

    if (!DateTime.TryParse(value as string, out dateValue))
    {
      throw new NotSupportedException();
    }
    Debug.WriteLine("string value to DateTime: {0}",
dateValue.ToString());
    return dateValue;
  }
}
```

6. Right-click the core PCL again, **Add | New Item…**, choose **Xaml Forms Page**, name it `MainPage.xaml`. and click **Add**. Find the contents of the page next:

```
<?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:XamFormsValueConverter;assembly=XamFormsValueConverter"
  x:Class="XamFormsValueConverter.MainPage">
  <ContentPage.Resources>
    <ResourceDictionary>
      <local:DateTimeToStringConverter x:Key="dateConverter" />
    </ResourceDictionary>
  </ContentPage.Resources>
  <Entry Text="{Binding DateOrdered, Converter={StaticResource
dateConverter}"
  VerticalOptions="Center" HorizontalOptions="Center" />
</ContentPage>
```

7. Go to `MainPage.xaml.cs` and change the constructor to the following code:

```
public MainPage(Order order)
{
  InitializeComponent();

  BindingContext = order;
}
```

8. Open the `App.cs` file and change the constructor to the following code instantiating our newly created `MainPage`:

```
public App()
{
  MainPage = new MainPage(new Order { DateOrdered = DateTime.Now });
}
```

9. Run the application and you should immediately see debug messages to the Output windows like the following. You can test and change the value; the messages will appear every time the value is changed from the source and before updating the source.

```
DateTime to string: 12/16/2015 21:16:09.807
…
string value to DateTime: 12/16/2015 9:16:09 PM
```

Android:

9:36
12/16/2015 21:36:05.861

iOS:

Carrier
11:45 AM
12/16/2015 21:37:04.754

Windows Phone:

How it works…

If you come from the .NET world to the Xamarin.Forms world, you already must have noticed that there's no difference in the way you use value converters and data binding. The crucial piece of the puzzle is a class implementing the `IValueConverter` interface.

For this recipe, we created an `Order` class with only one property, `DateOrdered`, of type `DateTime`. We implemented the `INotifyPropertyChanged` interface and added a helper method to raise property-changed events. For more details on how to do this and how it works, refer to the section *Configuring two-way data binding*.

To get access in the flow of the interaction between the source and the target when a value-changed event is raised, and the binding is triggered with the new value, we created the class `DateTimeToStringConverter`. This class implements the `IValueConverter` interface and will act as a proxy between the source and the target. There are two methods we need to implement:

- `Convert`
- `ConvertBack`

The `Convert` method will be invoked when the value is updated in the source, in our example when the `DateOrdered` property is changed. When the value is returned from the method, it will be assigned to the element property. This is where we convert the `DateTime` property to a string with the format `MM/dd/yyyy HH:mm:ss.fff`.

The `ConvertBack` method is invoked when the target value is changed and will update the source, changing the element's value, will invoke this method before assigning the value to the source.

In both of the methods, we added debug messages to monitor this behavior in the output window.

To use the `DateTimeToStringConverter` class in our `MainPage.xaml`, we create a resource in the `ContentPage.Resources ResourceDictionary`. Don't forget that we need to import the namespace that the class lives in to make XAML aware of the class. To complete the pieces, we assign the static resource to the Converter property of the binding expression of the `Entry` element.

The last part is standard, setting the `BindingContext` to an `Order` instance in the `MainPage` constructor passed from the `App.cs` constructor. To learn more about `BindingContext`, go to the section *Configuring two-way data binding*.

Using an `IValueConverter` implementation is not the only way to achieve formatting. You can use the `StringFormat` property available in the Binding expression. Check the following example code for how to do this:

```
<StackLayout BindingContext="{x:Static sys:DateTime.Now}"
  HorizontalOptions="Center"
  VerticalOptions="Center">
    <Label Text="{Binding Year, StringFormat='The year is {0}'}" />
    <Label Text="{Binding StringFormat='The month is {0:MMMM}'}" />
```

```
    <Label Text="{Binding Day, StringFormat='The day is {0}'}" />
    <Label Text="{Binding StringFormat='The time is {0:T}'}" />
</StackLayout>
```

See also

- https://developer.xamarin.com/guides/cross-platform/xamarin-forms/user-interface/xaml-basics/data_bindings_to_mvvm/

Chapter 8. A List to View

In this chapter, we will cover the following recipes:

- Displaying a collection and selecting a row
- Adding, removing, and refreshing items
- Customizing the row template
- Adding grouping and a jump index list

Introduction

99 percent of applications are using a control that you can scroll and view a collection of data. This collection of data, sometimes more than one type, can be represented in many ways to show the corresponding information. In this chapter, we will see how the Xamarin.Forms `ListView` control works with our collections.

A collection is a stream of data and the `ListView` control in Xamarin.Forms helps significantly with how we present the data to the user in a vertical scrollable area, and enable actions such as adding, removing, or editing rows. This is certainly one of the most important controls that you will spend a lot of time on during the development of any project.

The Xamarin.Forms `ListView` is translated in the native control for each platform: a `UITableView` for iOS, a `ListView` for Android, and a `LongListSelector` for the Windows Phone platform.

We will investigate the simple setup of a `ListView` setting a collection and its default layout; how to select, add, and remove a row; and the out-of-the-box pull-to-refresh feature. Next, we take a step further and see how to customize a row using one of the built-in cell types and also provide your complete cell layout using the `ViewCell` type.

Finally, we explore features such as as grouping and the jump list supported by the iOS platform only.

Displaying a collection and selecting a row

The main requirement for `ListView` to work is to provide a collection of data where each item will be translated to a row.

`ListView` exposes an event that will notify you when a row is tapped; usually, you'll want to register an event handler to navigate to another page.

How to do it…

1. In Visual Studio, go to the top menu and select **File | New | Project**, choose the **Blank App** (Xamarin.Forms Portable) template, name it `XamFormsDisplayCollections`, and click **OK**.
2. The first thing we need is to create our model class and some dummy data to load in `ListView`. Right-click the PCL, **Add | Class…**, name it `Character`, and click **Add**. Find next the abbreviated version of the `Character` class; refer to the book code for this section for the list of characters returned in the `Characters` property:

```
public class Character
{
  public string Name { get; set; }
  public string Species { get; set; }
  public string ImageUrl { get; set; }

  public override string ToString()
  {
    return string.Format("{0}, {1}", Name, Species);
  }

  public static IList<Character> Characters
  {
    get
    {
      return new List<Character>
      {
        new Character
        {
          …
        },
        …….
      };
    }
  }
}
```

3. Right-click again and select **Add | New Item…**. Choose **Forms Xaml Page**, name it `MainPage.xaml`, and click **Add**. Find the contents of the newly created page next:

```
<?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:XamFormsDisplayCollections;assembly=XamFormsDisplayCollection
s"
  x:Class="XamFormsDisplayCollections.MainPage"
  BindingContext="{x:Static local:Character.Characters}"
  Title="Star Wars">
    <ListView ItemsSource="{Binding .}" ItemTapped="OnItemTapped"/>
</ContentPage>
```

4. Repeat step 3 and create a page named `CharacterPage`. Find the contents next:

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
  x:Class="XamFormsDisplayCollections.CharacterPage"
  Title="{Binding Name}">
    <Label Text="{Binding Name}" VerticalOptions="Center"
HorizontalOptions="Center" />
</ContentPage>
```

5. Go to `CharacterPage.xaml.cs` and change the constructor like the following code:

```
public CharacterPage(Character character)
{
  InitializeComponent();

  BindingContext = character;
}
```

6. Go to `MainPage.xaml.cs` and add the `OnItemTapped` event handler.

```
private async void OnItemTapped(object sender, ItemTappedEventArgs
args)
  {
    Character character = args.Item as Character;
    await Navigation.PushAsync(new CharacterPage(character));
  }
```

7. Go to the `App.cs` file and change the constructor like the following code:

```
public App()
{
  // The root page of your application
  MainPage = new NavigationPage(new MainPage());
}
```

8. Run the application to all the platforms and may the force be with you!

iOS:

Carrier
8:03 PM
Star Wars
Owen Lars, Human
C-3PO, Droid
R2-D2, Droid
Darth Vader, Human
Luke Skywalker, Human
Obi-Wan Kenobi, Human

Android:

7:55
Owen Lars, Human
C-3PO, Droid
R2-D2, Droid
Darth Vader, Human
Luke Skywalker, Human
Obi-Wan Kenobi, Human

Tapping a row will navigate you to `CharacterPage` showing the character's name in the center of the page.

Windows Phone:

How it works…

To set up a `ListView` and show some data in its simplest form like in our recipe doesn't need more than a few lines of code.

All you need is to set a list of some type to the `ListView.ItemsSource` property. There are a couple of ways to set the list to `ItemsSource`, and in this example we used the `x:Static` XAML extension and data binding to bind an `IList<Character>` of some Star Wars characters.

For simplicity, we created a static property that returns the list. With the help of the `x:Static` XAML extension, we retrieved the data and set it to `BindingContext` of `MainPage` in XAML; convenient in our case but in a real-world scenario you would create a data repository, fetch the data in `ViewModel`, and bind the list property of `ViewModel` on `ListView.ItemsSource`. Please refer to Chapter 5, *Dude, Where's My Data?* and *Architecture design with Model-View-ViewModel (MVVM) pattern* recipe from Chapter 4, *Different Cars, Same Engine* to learn how to apply these practices.

Having a list set to `BindingContext`, we can access the data with a Binding expression on the `ListView.ItemsSource` property. To learn more about data binding, go to Chapter 7, *Bind to the Data*.

Without further row customization, the default cell will invoke the `ToString()` method of the list item type we overridden this method and returned a string representation of `Name, Species`.

Finally, to navigate to `CharacterPage`, we wired up an event handler for the `ListView.ItemTapped` event. When this method is invoked, we have access to the current item that is tapped in the `ItemTappedEventArgs` method argument. With that, we cast the `ItemTappedEventArgs.Item` property to a `Character` and pass it to `CharacterPage` where we navigate using the `Navigation.PushAsync` method.

Using the MVVM pattern, the `ItemTapped` event will not be a great help; fortunately, `ListView` exposes the `SelectedItem` property that you can programmatically get and set a selected item. You could use binding on the `SelectedItem` property in XAML and track the selected item of `ListView`.

Note

Don't forget that the `Navigation` property will be null if we don't wrap `MainPage` in `NavigationPage`.

See also

- https://developer.xamarin.com/guides/xamarin-forms/user-interface/listview/data-and-databinding/

Adding, removing, and refreshing items

It is very common that you want to manipulate the data while your application is running; change data on your server, or locally; do calculations; refresh the representation of your data; and allow your user to add items and update or delete an item. This is as easy as working with a collection type in C#.

How to do it…

1. In Visual Studio, go to the top menu and select **File | New | Project**. Choose the **Blank App** (Xamarin.Forms Portable) template, name it `XamFormsAddingRemovingItems`, and click **OK**.
2. In this recipe, we demonstrate the pull-to-refresh feature added in Xamarin.Forms 1.4. Make sure you have updated the **Xamarin.Forms NuGet** package for the solution. To do this, right-click the solution and select **Manage NuGet Packages** for **Solution**, go to the Updates tab, check the **Xamarin.Forms** package, and click **Install** for the latest stable version.
3. Right-click the PCL, **Add | Class…**, name it `Character`, and click **Add**. Find next the abbreviated version of the `Character` class; refer to the book's code for this section for the list of characters returned in the `Characters` property:

```
public class Character
{
  private static IList<Character> _characters;

  public string Name { get; set; }
  public string Species { get; set; }
  public string ImageUrl { get; set; }

  public override string ToString()
  {
    return string.Format("{0}, {1}", Name, Species);
  }

  public static IList<Character> Characters
  {
    get
    {
      return _characters ?? (_characters = new
ObservableCollection<Character>
      {
        new Character
        {
          …
        },
        …….
       };
    }
  }
}
```

4. Right-click again and select **Add | New Item…**. Choose **Forms Xaml Page**, name it `MainPage.xaml`, and click **Add**. Find the contents of the newly created page next:

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage
xmlns="http://xamarin.com/schemas/2014/forms"xmlns:x="http://schemas.mi
crosoft.com/winfx/2009/xaml"
  xmlns:local="clr-
namespace:XamFormsAddingRemovingItems;assembly=XamFormsAddingRemovingIt
```

```
ems"
  x:Class="XamFormsAddingRemovingItems.MainPage"
  BindingContext="{x:Static local:Character.Characters}"
  Title="Star Wars">
    <ContentPage.ToolbarItems>
      <ToolbarItem Text="Add" Activated="OnToolbarClick"
Order="Primary" Priority="0">
        <ToolbarItem.Icon>
          <OnPlatform x:TypeArguments="FileImageSource"
          WinPhone="Toolkit.Content/ApplicationBar.Add.png"/>
        </ToolbarItem.Icon>
      </ToolbarItem>
    </ContentPage.ToolbarItems>
    <ListView ItemsSource="{Binding .}" IsPullToRefreshEnabled="True"
Refreshing="OnRefreshing">
    <ListView.ItemTemplate>
      <DataTemplate>
        <TextCell Text="{Binding Name}">
          <TextCell.ContextActions>
            <MenuItem Clicked="OnDelete" Text="Delete"
IsDestructive="True"/>
          </TextCell.ContextActions>
        </TextCell>
      </DataTemplate>
    </ListView.ItemTemplate>
  </ListView>
</ContentPage>
```

5. Go to `MainPage.xaml.cs` and add the `OnToolbarClick`, `OnRefreshing`, and `OnDelete` event handlers.

```
private void OnToolbarClick(object sender, EventArgs args)
{
  if (Character.Characters.SingleOrDefault(p => p.Name.Equals("Anakin
Skywalker")) == null)
  {
    Character.Characters.Add(new Character
    {
      Name = "Anakin Skywalker",
      Species = "Human",
      ImageUrl =
"http://static.comicvine.com/uploads/original/11125/111250671/4775035-
a1.jpg"
    });
  }
}

private void OnDelete(object sender, EventArgs args)
{
  MenuItem menuItem = sender as MenuItem;
  Character character = menuItem.BindingContext as Character;
  Character.Characters.Remove(character);
}

private async void OnRefreshing(object sender, EventArgs args)
  {
```

```
    ListView listView = sender as ListView;
    listView.IsRefreshing = true;
    // Code to refresh…
    // Called on the UI thread.
    Debug.WriteLine("Refreshing!");
    await Task.Delay(2000);
    listView.IsRefreshing = false;
}
```

6. In the `App.cs` file, change the constructor with the following code:

```
public App()
{
  // The root page of your application
  MainPage = new NavigationPage(new MainPage());
}
```

7. Run the application in your Android emulator or device. Click the **ADD** toolbar button and Anakin Skywalker will be added to `ListView`.

   Android:

2:51
Star Wars
ADD
Owen Lars
C-3PO
R2-D2
Darth Vader
Luke Skywalker
Obi-Wan Kenobi
Anakin Skywalker

8. We have added a destructive `MenuItem` with the text `Delete` in the `TextCell.ContextActions`. In iOS, if you swipe the row to the left, it will show the button to delete the row; in Android and Windows Phone, a long tap is required.

   iOS:

Carrier
6:05 PM
Star Wars
Add
Owen Lars
C-3PO
R2-D2
der
Delete
Luke Skywalker
Obi-Wan Kenobi

9. Pull `ListView` down and you will get an indicator that something is happening. See the following Android screenshot:

   Android:

2:51
Star Wars
ADD
Owen Lars
C-3PO
R2-D2
Darth Vader
Luke Skywalker
Obi-Wan Kenobi
Anakin Skywalker

Note

Note that as of Xamarin.Forms 1.4.3, pull-to-refresh is not supported on Windows Phone 8.1. On Windows Phone 8, pull-to-refresh is not a native platform feature, so an implementation of pull-to-refresh is provided by Xamarin.Forms. Finally, be aware that pull-to-refresh will not work on Windows Phone if all elements in the list can fit on the screen (in other words, if vertical scrolling isn’t required).

How it works…

The static property `Characters`, in the `Character` class, returns `ObservableCollection<Character>`. `ObservableCollection<T>` is a list that provides methods to add and delete items. Whenever you make a change to the collection, it is reflected in `ListView`; this is possible due to `ObservableCollection` raising event notifications when the collection is changed with implementing the `INotifyCollectionChanged` interface.

Note

Note that this applies to collection changes and not for any updates of underlying items; for example, updating the properties of a collection item, a Character, will not trigger such a notification. For this, we need to implement the `INotifyPropertyChanged` interface for the item's class, in our case the `Character` class.

You might already have noticed that `ListView` doesn't support any actions manipulating its internal list of items. Instead, you always work directly with the underlying data source bind to the `ItemsSource` property.

In the recipe, we are using data binding, setting the `Character.Characters` in the `BindingContext` property of `MainPage` in XAML with the `x:Static` XAML extension, and using the `{Binding}` expression for the `ListView.ItemsSource` property binding to the actual `BindingContext` object. To learn more about data binding, go to Chapter 7, *Bind to the Data*. For more information about the architecture in fetching data and showing to the view, check out Chapter 5, *Dude, Where's My Data?* and *Architecture design with Model-View-ViewModel (MVVM) pattern* recipe from Chapter 4, *Different Cars, Same Engine*.

`ContentPage` has a property of `ToolbarItems`. Adding a toolbar item will provide an equivalent action for each platform. We register an event handler named `OnToolbarClicked` for the `Activated` event to programmatically add a `Character`.

For `Toolbar.Icon`, we utilize the handy `OnPlatform` object to instruct a specific icon for the Windows Phone platform. This is needed due to the default representation of the Windows Phone toolbar icon of an X icon, which misleads the functionality to the user.

For the deletion of an item, we have modified the row layout by adding `MenuItem` that is set as destructive and the text to `Delete` in the `TextCell.ContextActions` collection. Providing `DataTemplate` in `ListView.ItemTemplate`, we return the built-in `TextCell`. This `DataTemplate` will be used for each item in our collection, and for every platform it will provide a way of accessing the action. In iOS, you get the familiar button option when you swipe the row cell to the left; in Android and Windows Phone, you need to long tap the row and the option will appear.

For more details on how to customize a row template, refer to the section in this chapter, *Customizing the row template*.

`ListView` provides a very easy way for the very common pull-to-refresh gesture of a

scrolling control. It is as easy as setting `IsPullToRefreshEnabled` to `true` and registering an event handler for the `Refreshing` event in the behind code. In the event handler, you show and hide the indicator by setting the `IsRefreshing` property of `ListView`.

Note

Note that with the preceding information, in the Windows Phone platform there are version support limitations: refresh will not fire if the collection is not large enough where a vertical scroll is not required and Windows Phone 8.1 is not supported.

There’s more…

As a side note, sometimes you might modify a collection from another thread, but this must always be done on the UI thread or you will receive a runtime exception and your application will crash.

There are some ways to resolve this issue. Using Task, if a call is initiated asynchronously from the UI thread, for example, you begin to fetch data from the network or a local database, you can define a continuation delegate and when Task is completed, the delegate will be invoked in the synchronization context that it started.

When the async/await keywords are used, by default the call returns to the current context that initiated, but you can explicitly define this with the `ConfigureAwait(bool)` method of the Task object setting the Boolean argument to `true`, which is the default value.

Another way is by using the `Device.BeginInvokeOnMainThread(Action)` static method and providing an Action that will be invoked on the UI thread. If we needed to do this in our code, the `OnToolbarClicked` handler would look like the following:

```
Device.BeginInvokeOnMainThread(() =>
{
  Character.Characters.Add(new Character
  {
    Name = "Anakin Skywalker",
    Species = "Human",
    ImageUrl =
"http://static.comicvine.com/uploads/original/11125/111250671/4775035-
a1.jpg"
  });
});
```

The binding system also provides a `static` method where you can pass the collection and a delegate to ensure that your collection is modified on the UI thread. In our solution, you would do something like the following code in the `MainPage.xaml.cs` constructor:

```
BindingBase.EnableCollectionSynchronization(
  Character.Characters,
  null,
  (list, context, action, writeAccess) =>
  {
    lock (list)
    {
      action();
    }
  });
```

See also

- https://developer.xamarin.com/guides/xamarin-forms/user-interface/listview/interactivity/

Customizing the row template

Xamarin.Forms has four built-in cell types:

- `EntryCell` – a cell with a Label and a single-line text Entry field
- `SwitchCell` – a cell with a Label and an on/off switch
- `TextCell` – a cell with primary and secondary text
- `ImageCell` – a cell that also includes an image

But this is not enough for every case that you need, or sometimes will not provide you with a nice result. Xamarin.Forms has another type, `ViewCell`, that you can use to customize the layout as desired.

You are welcome to extend any of these types to introduce custom functionality.

How to do it…

1. In Visual Studio, go to the top menu and select **File | New | Project**. Choose the **Blank App** (Xamarin.Forms Portable) template, name it `XamFormsCustomizingRows`, and click **OK**.
2. Right-click the PCL, **Add | Class…**, name it `Character`, and click **Add**. Find next the abbreviated version of the `Character` class. Refer to this section in the book code for the list of characters returned in the `Characters` property.

```
public class Character
{
  public string Name { get; set; }
  public string Species { get; set; }
  public string ImageUrl { get; set; }

  public override string ToString()
  {
    return string.Format("{0}, {1}", Name, Species);
  }

  public static IList<Character> Characters
  {
    get
    {
      return new List<Character>
      {
        new Character
        {
          …
        },
        …….
      };
    }
  }
}
```

3. Right-click again and select **Add | New Item…**. Choose **Forms Xaml Page**, name it `MainPage.xaml`, and click **Add**. Find the contents of the newly created page next:

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage
xmlns="http://xamarin.com/schemas/2014/forms"xmlns:x="http://schemas.mi
crosoft.com/winfx/2009/xaml"
  xmlns:local="clr-
namespace:XamFormsCustomizingRows;assembly=XamFormsCustomizingRows"
  x:Class="XamFormsCustomizingRows.MainPage"
  BindingContext="{x:Static local:Character.Characters}"
  Title="Star Wars">
    <ListView ItemsSource="{Binding .}" RowHeight="60">
      <ListView.ItemTemplate>
        <DataTemplate>
          ImageCell Text="{Binding Name}"
          Detail="{Binding Species}"
          ImageSource ="{Binding ImageUrl}"/>
```

```
        </DataTemplate>
      </ListView.ItemTemplate>
    </ListView>
</ContentPage>
```

4. Go to the `App.cs` file and change the constructor like the following:

```
public App()
{
  // The root page of your application
  MainPage = new NavigationPage(new MainPage());
}
```

5. Run the application for each platform.

Android:

6:48
Star Wars
Owen Lars
Human
C-3PO
Droid
R2-D2
Droid
Darth Vader
Human
Luke Skywalker
Human
Obi-Wan Kenobi
Human

Windows Phone:

iOS:

Carrier 9:04 PM

Star Wars

Owen Lars
Human

C-3PO
Droid

R2-D2
Droid

Darth Vader
Human

Luke Skywalker
Human

Obi-Wan Kenobi
Human

Using the built-in `ImageCell`, the layout of the image and details look fine on the Android platform, but not so great in Windows Phone, which crops the images, and with misalignment in iOS. This is due to the lack of control in `ImageCell`.

Fortunately, using the `ViewCell` type, we can customize the layout and behavior of the row. You can also derive from it or any other cell type and extend the functionality.

Let's add `ViewCell` and customize the layout in XAML.

1. Go to the `MainPage.xaml` file and replace the `ImageCell` tag inside `DataTemplate` with the following contents:

```
<ViewCell>
  <Grid>
    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="75"/>
      <ColumnDefinition Width="*"/>
    </Grid.ColumnDefinitions>
    <Image Grid.Column="0" WidthRequest="75" Source ="{Binding
ImageUrl}" Aspect="AspectFit" />
    <StackLayout Grid.Column="1">
      <Label Text="{Binding Name}" FontSize="Medium"
FontAttributes="Bold"/>
      <Label Text="{Binding Species}" FontSize="Small"/>
    </StackLayout>
  </Grid>
</ViewCell>
```

2. In `ListView`, add the following property to set the height of the row:

```
RowHeight="60"
```

3. Replace the `App.cs` constructor with the following code:

```
public App()
{
  // The root page of your application
  MainPage = new NavigationPage(new MainPage());
}
```

4. Run the application and you will notice that for each platform, the results are much better. It's exactly what we instructed inside `ViewCell`. See the following iOS screenshot:

   iOS:

Carrier 3:31 PM

Star Wars

Owen Lars

Human

C-3PO

Droid

R2-D2

Droid

Darth Vader

Human

Luke Skywalker

Human

Obi-Wan Kenobi

Human

How it works…

In order to customize the visual appearance of the `ListView` rows, you need to provide an `ItemTemplate` of type `DataTemplate`. `DataTemplate` must always return a `Cell`.

Presenting for each Star Wars character a row with an image, the name, and the species can be easily supported with the out-of-the-box `ImageCell` type. In `ImageCell`, we bind to the properties of a single item of the `ItemsSource` underlying data source. `Text` property is assigned to `Name`, `Detail` to `Species`, and `ImageSource` to the `ImageUrl` properties.

Working with the built-in cell types is perfect for simple customization but not always a good fit for our needs. In our solution, we load some images from some Internet URLs regarding each Star Wars character. Each image size is not equal, so for every platform it will behave differently by default. Android has a descent resizing feature, Windows Phone crops images, and iOS resizes the image and messes the alignment.

The solution to this problem is not to use `ImageCell`, but the `ViewCell` type, which allows us to return a single view. In our case, we return a Grid and inside that we have Image and `StackLayout` that contains two `Label` controls. We now have full control over the layout representation of the row with respect to resizing and layout. For example, note the `Image.Aspect` property set to `AspectFit` or `Label.FontSize` to medium or small.

`ListView` sets `RowHeight` automatically, but we set the value explicitly. If you don’t set `RowHeight`, `ListView` will go through the data and set the height based on the highest element that it finds.

You are always welcome to also derive from any built-in cell type and provide the implementation required in code.

There’s more…

`ListView` supports a header and footer via the the properties `Header` and `Footer`. You can set these properties or bind them to a string value where it will automatically render a Label with the value on the top and bottom of `ListView`.

One solution is to provide a `View` to `ListView.Header` and `ListView.Footer` like the following simple example:

```
<ListView.Header>
  <ContentView>
    <Label Text ="Simple Header Customized!" />
  </ContentView>
</ListView.Header>
```

That introduces another problem when implementing the MVVM pattern. To learn how to apply the pattern in your project, go to *Architecture design with Model-View-ViewModel (MVVM) pattern* recipe from Chapter 4, *Different Cars, Same Engine*.

Binding a `ViewModel` property to `ListView.Header` is unacceptable because it is violating the pattern principle that `ViewModel` shouldn’t be aware of any View-related code and dependencies; well, the good news is that there is a way to overcome this.

`ListView` has a property named `HeaderTemplate` and you can use Binding expressions. `BindingContext` is what you set to the `ListView.Header` property; see the following example:

```
<ListView ItemsSource="{Binding .}" Header="{Binding StarWarsEpisode}">
  <ListView.HeaderTemplate>
    <DataTemplate>
      <Label Text="{Binding .}" />
    </DataTemplate>
  </ListView.HeaderTemplate>
</ListView>
```

In this scenario, the `DataTemplate` returns a `View` and not a cell type. The `{Binding}` expression in the `Label.Text` property inside the `DataTemplate` corresponds to the `ListView.Header` value as the `BindingContext`.

See also

- https://developer.xamarin.com/guides/xamarin-forms/user-interface/listview/customizing-cell-appearance/

Adding grouping and a jump index list

Grouping is essential in many cases. Most often, you will see grouping in a settings page that breaks the rows into categories. It can be applied for anything that makes sense in a parent-child fashion; maybe you want to show grouped orders with their corresponding items.

In iOS, we enable the jump list built-in support, which will make visible a small list to the right of the screen that you can tap and navigate to the corresponding index of the collection.

How to do it…

1. In Visual Studio, go to the top menu and select **File | New | Project**. Choose the **Blank App** (Xamarin.Forms Portable) template, name it **XamFormsAddGrouping**, and click **OK**.
2. Right-click the PCL, **Add | Class…**, name it `Character.cs`, and click **Add**. Find next the abbreviated version of the `Character` class; refer to this section in the book code for the list of characters returned in the `Characters` property:

```
public class Character
{
  public string Name { get; set; }
  public string Species { get; set; }
  public string ImageUrl { get; set; }

  public override string ToString()
  {
    return string.Format("{0}, {1}", Name, Species);
  }

  public static IList<Character> Characters
  {
    get
    {
      return new List<Character>
      {
        new Character
        {
          …
        },
        …….
      };
    }
  }
}
```

3. Right-click again and select **Add | New Item…**. Choose **Forms Xaml Page**, name it `MainPage.xaml`, and click **Add**. Find the contents of the newly created page next:

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage
xmlns="http://xamarin.com/schemas/2014/forms"xmlns:x="http://schemas.mi
crosoft.com/winfx/2009/xaml"
  x:Class="XamFormsAddGrouping.MainPage"
  Title="Star Wars">
    <ListView ItemsSource="{Binding .}" RowHeight="60"
      IsGroupingEnabled="True"
      GroupDisplayBinding="{Binding Key}"
      GroupShortNameBinding="{Binding Key}">
    <ListView.ItemTemplate>
      <DataTemplate>
        <ViewCell>
          <Grid>
            <Grid.ColumnDefinitions>
```

```
              <ColumnDefinition Width="75"/>
              <ColumnDefinition Width="*"/>
            </Grid.ColumnDefinitions>
            <Image Grid.Column="0" WidthRequest="75" Source ="{Binding
ImageUrl}" Aspect="AspectFit" />
            <StackLayout Grid.Column="1">
              <Label Text="{Binding Name}" FontSize="Medium"
FontAttributes="Bold"/>
              <Label Text="{Binding Species}" FontSize="Small"/>
            </StackLayout>
          </Grid>
        </ViewCell>
      </DataTemplate>
    </ListView.ItemTemplate>
  </ListView>
</ContentPage>
```

4. Right-click in the `XamFormsAddGrouping` PCL and select **Add | Class…**. Name it `GroupingObservableCollection.cs` and click **Add**. See next the implementation of the newly created class:

```
public class GroupingObservableCollection<K, T> :
ObservableCollection<T>
  {
    public K Key { get; private set; }
    public GroupingObservableCollection(K key, IEnumerable<T> items) :
base(items)
    {
      Key = key;
    }
}
```

5. Go to `MainPage.xaml.cs` and add the following code in the constructor:

```
BindingContext = new
ObservableCollection<GroupingObservableCollection<string, Character>>(
    Character.Characters
    .OrderBy(c => c.Name)
    .GroupBy(c => c.Name[0].ToString(), c => c)
    .Select(g => new GroupingObservableCollection<string, Character>
(g.Key, g)));
```

6. Replace the `App.cs` constructor with the following code:

```
public App()
{
  MainPage = new NavigationPage(new MainPage());
}
```

7. Run the application and you will notice that only the iOS platform shows a small list to the right of your screen that you can use to jump to a letter; Android and Windows Phone are not currently supported.

iOS:

Carrier 4:19 PM

Star Wars

C

C-3PO

Droid

D

Darth Vader

Human

L

Luke Skywalker

Human

O

Obi-Wan Kenobi

Human

Owen Lars

Human

R

C
D
L
O
R

Android:

4:06
Star Wars
Darth Vader
Human
L
Luke Skywalker
Human
O
Obi-Wan Kenobi
Human
Owen Lars
Human
R
R2-D2
Droid

Windows Phone:

How it works…

`ListView` supports grouping out of the box, setting the `IsGroupingEnabled` property to true. Additionally, it is needed to set the `GroupDisplayBinding` property, which will define the parent group key.

Each item in the list that you will provide has to be a list. In our solution, we applied a practice of defining our custom `GroupingObservableCollection<K, T>`, where `K` is the key type and derives from `ObservableCollection<T>`. In this way, we can extend the collection and add additional properties like the `Key` property. The `Key` property is the parent value and the collection items are the children.

As an extra feature for iOS users, we bind the `GroupShortNameBinding` property to the `Key` property, which will automatically add a small list on the right of `UITableView` with the key letters available; tapping to a letter will jump directly to the corresponding section. Unfortunately, at the moment this is supported only in iOS.

For each row, we provided a custom `DataTemplate`. For details on how to do this, refer to the recipe *Customizing the row template*.

Finally, in the `MainPage.cs` constructor, we set the page's `BindingContext` to `ObservableCollection` of `GroupingObservableCollection<string, Character>`. In the constructor of `GroupingObservableCollection<string, Character>`, we pass `IEnumerable<GroupingObservableCollection<string, Character>>` ordered by character name and grouped by the first letter of the name using LINQ.

There’s more…

Setting `GroupDisplayBinding` to the key will automatically create a header for each grouping elements as a Label. There is a way that you can customize the header for each section in `ListView` by setting `GroupHeaderTemplate` to `DataTemplate` where you have to return a `Cell` type.

```
<ListView.GroupHeaderTemplate>
  <DataTemplate>
    <TextCell Text="{Binding Key}"
    Detail="{Binding Count, StringFormat='{0} characters'}"/>
  </DataTemplate>
</ListView.GroupHeaderTemplate>
```

See also

- https://developer.xamarin.com/guides/xamarin-forms/user-interface/listview/customizing-list-appearance/#Grouping

Chapter 9. Gestures and Animations

In this chapter, we will cover the following recipes:

- Adding gesture recognizers in XAML
- Handling gestures with native platform renderers
- Adding cross-platform animations

Introduction

Mobile devices today are controlled, almost exclusively, by touches. It is the main interaction between the user and the device for input.

As a developer, you understand the importance for the success of an application. It’s all in the user’s hands when it goes from the store to a mobile device, and for a demanding user, any touches or gestures have to be essential and intuitive.

Users also value fireworks. They like fancy and crisp apps, the ones that are alive, and how to make an app alive is through animations!

It is also, I believe, the reason for the success of the two top platforms today, with the weight more to Apple. After all, the iPhone’s success is what made the gesture touch system part of our life; not that this means it was the inventor of it.

The touch and animation system for each native platform is significantly different in how it works, from its philosophy to the API.

In this chapter, we will explore built-in gesture recognizers, add native platform gesture recognizers and touch events for each platform, and add cross-platform animations.

Adding gesture recognizers in XAML

The most common gestures in every mobile platform are tap, pinch, swipe, and long tap. All these can be combined. Maybe you have also encountered applications that use more than one finger; for example, the two-finger swipe.

Every platform has a completely different system to handle touches and gestures. I find Apple’s philosophy the best architected, and it seems Xamarin.Forms does too. If you are coming from iOS, you will find the same naming of the gesture recognizers.

You are able to add multiple gesture recognizers to a UI element. Unfortunately, at the time of this writing there is support for the tap and pinch gestures only, but the Xamarin.Forms team is adding gesture recognizers, as they proved in the framework releases, and so I believe they will support more if not all in the future.

How to do it…

1. In Visual Studio, go to the top menu and select **File | New | Project**. Choose the **Blank App** (Xamarin.Forms Portable) template, name it `XamFormsGestures`, and click **OK**.
2. Right-click on the solution and select **Manage NuGet Packages** for **Solution**. Go to the **Updates** tab and update the **Xamarin.Forms NuGet** package to the latest version.
3. Right-click the `XamFormsGesture` PCL project, select **Add | Class…**, and choose the **Forms Xaml Page** template. Give it the name `MainPage.xaml` and click **Add**.
4. Replace the XAML code with the following code:

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
  x:Class="XamFormsGestures.MainPage">
    <Image x:Name="image"
      Source="http://cdn-media.extratv.com/2015/06/22/anakin-skywalker-
1-825x580.jpg">
          <Image.GestureRecognizers>
            <TapGestureRecognizer Tapped="OnImageTapped" />
            <PinchGestureRecognizer PinchUpdated="OnPinchUpdated" />
          </Image.GestureRecognizers>
    </Image>
</ContentPage>
```

5. Go to `MainPage.xaml.cs` and copy the following event handlers for the `Tapped` and `PinchUpdated` events of the gesture recognizers:

```
private void OnImageTapped(object sender, EventArgs args)
{
  Debug.WriteLine("Image double-tapped!");
}

private void OnPinchUpdated(object sender, PinchGestureUpdatedEventArgs
e)
  {
    Debug.WriteLine("Image pinched!");
}
```

6. Change the constructor to show the `MainPage`.

```
public App()
{
  // The root page of your application
  MainPage = new MainPage();
}
```

7. Run the application and you should get a page with an Anakin Skywalker image like the following iOS screenshot.
8. Try to tap on the image or simulate the pinch gesture. If you are testing on a simulator, the output console will print similar messages to the following:

```
2016-01-08 19:22:46.209 XamFormsGesturesiOS[1618:46444] Image double-
tapped!
2016-01-08 19:22:51.433 XamFormsGesturesiOS[1618:46444] Image pinched!
```

Carrier 7:12 PM

How it works…

Every UI element has a `GestureRecognizers` property that you can use to add or remove gesture recognizers.

A gesture recognizer will handle the gesture event on the view and invoke your delegate action that you provided.

Xamarin.Forms will translate the gesture recognizer to the corresponding platform APIs. For more on how to manually add native platform gestures and touches behavior, go to the *Handling gestures with native platform renderers* recipe of this chapter.

There’s more…

`TapGestureRecognizer` also supports the number of taps required by setting the property `NumberOfTapsRequired`.

If your application uses the MVVM pattern and leveraging data-binding you would like to bind your `ICommand` properties and execute when the gesture is applied. You can easily do this with the `Command` and `CommandParameter` properties of the gesture recognizer.

```
<Image Source="your_image_uri">
  <Image.GestureRecognizers>
    <TapGestureRecognizer
      Command="{Binding ViewModelCommand}"
      CommandParameter="param" />
    </Image.GestureRecognizers>
</Image>
```

See also

- You might find this commercial plugin that handles more cross-platform gestures useful: http://www.mrgestures.com/

Handling gestures with native platform renderers

Built-in gesture recognizers work well for a simple tap and pinch on a view, but sometimes you need more flexibility and control over the touches behavior.

Xamarin.Forms custom renderers will save the day. Creating a custom renderer for a view will provide you with the native functionality that is needed in runtime. There is a custom renderer for every Xamarin.Forms view. To learn more about how to create custom renderers, refer to *Using custom renderers to change the look and feel of views* recipe from Chapter 2, *Declare Once, Visualize Everywhere*.

For this section we will create a custom `Image` class and add custom renderers with native gestures implementation printing messages in the output for testing purposes.

How to do it…

1. In Visual Studio, go to the top menu and select **File | New | Project**. Choose the **Blank App** (Xamarin.Forms Portable) template, name it `XamFormsRendererGestures` and click **OK**.
2. Right-click the `XamFormsRendererGestures` PCL and select **Add | Class…**, name it `GestureImage` and click **Add**. The class has to derive from `Image`; no custom code is required for our example. Find the `GestureImage` code next:

```
public class GestureImage : Image
  {
}
```

3. Right-click again and select **Add | New Item…**. Choose the **Forms Xaml Page** template, name it `MainPage.xaml` and click **Add**. Find the contents of the page next:

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage
xmlns="http://xamarin.com/schemas/2014/forms"xmlns:x="http://schemas.mi
crosoft.com/winfx/2009/xaml"
  xmlns:local="clr-
namespace:XamFormsRendererGestures;assembly=XamFormsRendererGestures"
  x:Class="XamFormsRendererGestures.MainPage">
    <local:GestureImage Source="http://cdn.playbuzz.com/cdn/852d461b-
a454-41ee-8b38-ef10baac9161/bc89be98-c6f3-4c13-9c8e-1d5b6409ac55.jpg"
    VerticalOptions="CenterAndExpand"
    HorizontalOptions="CenterAndExpand" />
</ContentPage>
```

4. Go to the `XamFormsRendererGestures.Droid` project, right-click and select **Add | Class…**. Name the class `GestureImageListener.cs` and click **Add**. This will be a `SimpleOnGestureListener`.

```
public class GestureImageListener :
GestureDetector.SimpleOnGestureListener
  {
    public override void OnLongPress(MotionEvent e)
    {
      Console.WriteLine("OnLongPress");
      base.OnLongPress(e);
    }

    public override bool OnDoubleTap(MotionEvent e)
    {
      Console.WriteLine("OnDoubleTap");
      return base.OnDoubleTap(e);
    }

    public override bool OnDoubleTapEvent(MotionEvent e)
    {
      Console.WriteLine("OnDoubleTapEvent");
      return base.OnDoubleTapEvent(e);
    }
```

```
    public override bool OnSingleTapUp(MotionEvent e)
    {
      Console.WriteLine("OnSingleTapUp");
      return base.OnSingleTapUp(e);
    }

    public override bool OnDown(MotionEvent e)
    {
      Console.WriteLine("OnDown");
      return base.OnDown(e);
    }

    public override bool OnFling(MotionEvent e1, MotionEvent e2, float
velocityX, float velocityY)
    {
      Console.WriteLine("OnFling");
      return base.OnFling(e1, e2, velocityX, velocityY);
    }

    public override bool OnScroll(MotionEvent e1, MotionEvent e2, float
distanceX, float distanceY)
    {
      Console.WriteLine("OnScroll");
      return base.OnScroll(e1, e2, distanceX, distanceY);
    }

    public override void OnShowPress(MotionEvent e)
    {
      Console.WriteLine("OnShowPress");
      base.OnShowPress(e);
    }

    public override bool OnSingleTapConfirmed(MotionEvent e)
    {
      Console.WriteLine("OnSingleTapConfirmed");
      return base.OnSingleTapConfirmed(e);
    }
}
```

5. Right-click again, select **Add | Class…**, name it `GestureImageDroidRenderer.cs` and click **Add**. Copy the implementation from the following code snippet:

```
[assembly: ExportRenderer(typeof(GestureImage),
typeof(GestureImageDroidRenderer))]

namespace XamFormsRendererGestures.Droid
{
  public class GestureImageDroidRenderer : ImageRenderer
  {
    private readonly GestureImageListener _imageGestureListener;
    private readonly GestureDetector _gestureDetector;

    public GestureImageDroidRenderer()
    {
      _imageGestureListener = new GestureImageListener();
      _gestureDetector = new GestureDetector(_imageGestureListener);
```

```
    }

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

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

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

    void HandleTouch(object sender, TouchEventArgs e)
    {
      _gestureDetector.OnTouchEvent(e.Event);
    }

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

6. Run the `XamFormsRendererGestures.Droid` application. For every gesture interaction on the Jedi image, you will receive messages printed in the output console.

8:56

```
01-08 20:58:32.184 I/mono-stdout( 1730): OnDown
01-08 20:58:32.269 I/mono-stdout( 1730): OnSingleTapUp
01-08 20:58:32.476 I/mono-stdout( 1730): OnSingleTapConfirmed
```

7. Go to the `XamFormsRendererGestures.iOS` project, right-click, select **Add** | **Class…**, name it `GestureImageTouchRenderer.cs` and click **Add**.

```
[assembly: ExportRenderer(typeof(GestureImage),
typeof(GestureImageTouchRenderer))]

namespace XamFormsRendererGestures.iOS
{
  public class GestureImageTouchRenderer : ImageRenderer
  {
    UILongPressGestureRecognizer longPressGestureRecognizer;
    UIPinchGestureRecognizer pinchGestureRecognizer;
    UIPanGestureRecognizer panGestureRecognizer;
    UISwipeGestureRecognizer swipeGestureRecognizer;
    UIRotationGestureRecognizer rotationGestureRecognizer;

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

      longPressGestureRecognizer = new UILongPressGestureRecognizer(()
=> Console.WriteLine("Long Press"));
      pinchGestureRecognizer = new UIPinchGestureRecognizer(() =>
Console.WriteLine("Pinch"));
      panGestureRecognizer = new UIPanGestureRecognizer(() =>
Console.WriteLine("Pan"));
      swipeGestureRecognizer = new UISwipeGestureRecognizer(() =>
Console.WriteLine("Swipe"));
      rotationGestureRecognizer = new UIRotationGestureRecognizer(() =>
Console.WriteLine("Rotation"));

      if (e.NewElement == null)
      {
        if (longPressGestureRecognizer != null)
        {
          RemoveGestureRecognizer(longPressGestureRecognizer);
        }
        if (pinchGestureRecognizer != null)
        {
          RemoveGestureRecognizer(pinchGestureRecognizer);
        }
        if (panGestureRecognizer != null)
        {
          RemoveGestureRecognizer(panGestureRecognizer);
        }
        if (swipeGestureRecognizer != null)
        {
          RemoveGestureRecognizer(swipeGestureRecognizer);
```

```
        }
        if (rotationGestureRecognizer != null)
        {
          RemoveGestureRecognizer(rotationGestureRecognizer);
        }
      }

      if (e.OldElement == null)
      {
        AddGestureRecognizer(longPressGestureRecognizer);
        AddGestureRecognizer(pinchGestureRecognizer);
        AddGestureRecognizer(panGestureRecognizer);
        AddGestureRecognizer(swipeGestureRecognizer);
        AddGestureRecognizer(rotationGestureRecognizer);
      }
    }
  }
}
```

8. Go to the `XamFormsRendererGestures.WinPhone` project, right-click and select **Add | Class…**. Name it `GestureImagePhoneRenderer.cs` and click **Add**.

```
[assembly: ExportRenderer(typeof(GestureImage),
typeof(GestureImagePhoneRenderer))]

namespace XamFormsRendererGestures.WinPhone
{
  public class GestureImagePhoneRenderer : ImageRenderer
  {
    protected override void
OnElementChanged(ElementChangedEventArgs<Image> e)
    {
      base.OnElementChanged(e);

      if (e.NewElement == null)
      {
        Tap -= OnTap;
        DoubleTap -= OnDoubleTap;
        ManipulationStarted -= OnManipulationStarted;
        ManipulationDelta -= OnManipulationDelta;
        ManipulationCompleted -= OnManipulationCompleted;
      }

      if (e.OldElement == null)
      {
        Tap += OnTap;
        DoubleTap += OnDoubleTap;
        ManipulationStarted += OnManipulationStarted;
        ManipulationDelta += OnManipulationDelta;
        ManipulationCompleted += OnManipulationCompleted;
      }
    }

    // ManipulationStarted is the same thing as DragCompleted.
    private void OnManipulationCompleted(object sender,
ManipulationCompletedEventArgs e)
```

```
    {
      Debug.WriteLine("OnManipulationCompleted");
    }

    // ManipulationDelta represents either a drag or a pinch.
    private void OnManipulationDelta(object sender,
ManipulationDeltaEventArgs e)
    {
      Debug.WriteLine("OnManipulationDelta");
    }

    // ManipulationStarted is the same thing as DragStarted.
    private void OnManipulationStarted(object sender,
System.Windows.Input.ManipulationStartedEventArgs e)
    {
      Debug.WriteLine("OnManipulationStarted");
    }

    private void OnDoubleTap(object sender,
System.Windows.Input.GestureEventArgs e)
    {
      Debug.WriteLine("OnDoubleTap");
    }

    private void OnTap(object sender,
System.Windows.Input.GestureEventArgs e)
    {
      Debug.WriteLine("OnTap");
    }
  }
}
```

9. Run the the application and you will see messages printed in your output like the following when applying gestures on the image:

```
2016-01-08 21:17:34.509 XamFormsRendererGesturesiOS[2619:120824] Pinch
2016-01-08 21:17:35.880 XamFormsRendererGesturesiOS[2619:120824] Pan
```

How it works…

Creating `GestureImage` that is derived from the `Image` class and adding it to the `MainPage` XAML page means we are able to create our platform renderers and add native platform code for the custom view.

Having access to the native application layer, in Android, we created a `SimpleOnGestureListener` derive class, `GestureImageListener`, and overridden methods that will be invoked when gestures are applied such as long press, single tap, double tap, and so on.

For every Xamarin.Forms view, there is a renderer. In this case, we derive from `Xamarin.Forms.Platform.Android.ImageRenderer`. In the `GestureImageDroidRenderer` constructor, we create `Android.Views.GestureDetector` passing a `GestureImageListener` instance. Finally, we register handlers for the `GenericMotion` and `Touch` events of `Android.Views.View`; the view in our case is `GestureImage` rendered native view.

In the handler methods, we pass the `Event` object from the arguments to `GestureDetector`. All Android goodies here!

For iOS, things are very similar with the built-in Xamarin.Forms gesture recognizers. Every view has a `GestureRecognizers` property where you can add one or more gesture recognizers. In this example, we added the long press, pinch, pan, swipe, and rotation gesture recognizers.

The Windows Phone `UIElement` class provides all the necessary events to register your handlers and do any manipulation needed, something we implemented in `GestureImagePhoneRenderer`.

Adding cross-platform animations

Creating a great data-driven cross-platform application is a challenge, and Xamarin.Forms is the key to such success. It will provide everything needed to deliver a cross-platform mobile solution to your users.

Users are demanding, and as a user yourself you understand how critical it is to provide a rich user experience, fine-tune your app performance, and provide crisp graphics and animations! Putting all these together means your users intuitively understand how much you care for what you built, and they will always reward you with ratings, membership, coming back, and advertisement to other potential users.

Xamarin.Forms offers a cross-platform animation API. Next, we will create an example using the built-in extension methods.

How to do it…

1. In Visual Studio, go to the top menu and select **File** | **New** | **Project**. Choose the **Blank App** (Xamarin.Forms Portable) template, name it `XamFormsAddAnimations` and click **OK**.
2. Right-click the `XamFormsAddAnimations` PCL select **Add** | **New Item…** and choose the **Forms Xaml Page**. Name it `MainPage.xaml` and click **OK**.

```
<?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:XamFormsAddAnimations;assembly=XamFormsAddAnimations"
  x:Class="XamFormsAddAnimations.MainPage">
    <StackLayout>
      <StackLayout VerticalOptions="CenterAndExpand">
        <Image
          Source="{local:ImageResource XamFormsAddAnimations.yoda-
2.png}"
          Aspect="AspectFit" x:Name="image" Opacity="0.0"
          VerticalOptions="Center" HorizontalOptions="Start" />
    </StackLayout>
    <StackLayout Orientation="Horizontal" VerticalOptions="End">
      <Button Text="Animate" Clicked="OnButtonClicked"
HorizontalOptions="FillAndExpand"/>
    </StackLayout>
  </StackLayout>
</ContentPage>
```

Note

We used a local embedded resource image of Master Yoda in this example. If you want to learn more about working with images, refer to the following documentation link: https://developer.xamarin.com/guides/xamarin-forms/working-with/images/.

3. Go to `MainPage.xaml.cs` and add the `OnButtonClicked` event handler method.

```
private async void OnButtonClicked(object sender, EventArgs args)
  {
    await Task.WhenAll(image.FadeTo(1.0, 400, Easing.BounceIn),
image.RotateTo(360, 400, Easing.BounceOut),image.LayoutTo(new
Rectangle(200, image.Y,image.Width, image.Height), 2000,
Easing.Linear)).ContinueWith(async (antecedent) =>{
         if (antecedent.Status == TaskStatus.RanToCompletion)
         {
           await image.ScaleTo(1.5, 200);
                   await image.ScaleTo(1.0, 200);
                   await image.LayoutTo(new Rectangle(0, image.Y,
image.Width, image.Height), 2000, Easing.Linear);
         }
       },
           TaskScheduler.FromCurrentSynchronizationContext());
  }
```

4. Replace the `App.cs` constructor with the following code to initialize the app with the `MainPage.xaml` page:

```
public App()
{
  MainPage = new MainPage();
}
```

5. Run the application to any platform; the result is exactly the same: cross-platform goodies!
6. Click the bottom **ANIMATE** button and watch Master Yoda fading, rotating, and moving across the screen, with the X axis kicking in and returning back to the left edge of the screen, similar to the following Android screenshot:

   Android:

4:21
ANIMATE

iOS:

Windows Phone:

How it works…

Xamarin.Forms provides three built-in animation types: fading in/out, rotation, and translation of a view via the extension methods `FadeTo`, `RotateTo`, `LayoutTo`, and `ScaleTo`.

All the extension methods accept two last arguments as optional parameters: one is the length of the animation with a default value of 250 ms and an Easing parameter with a default value of `null`.

Easing will define how the motion of the animation will be executed. At the time of writing this book, there are 11 easing modes that you can choose from. Check the following link to learn more about these modes:

https://developer.xamarin.com/api/type/Xamarin.Forms.Easing/.

Note

With `FadeTo`, you can pass the opacity level you want as a result of the completion of the animation.

`RotateTo` accepts a first argument of the rotation value in degrees of the view.

In `LayoutTo`, pass an instance of `Rectangle`, which has a constructor accepting the *X* and *Y* coordinates where you want to change the location of the view and the width and height.

`ScaleTo` accepts a scale value, which will scale the view from the center.

All the extension methods are of type `Task`. To accomplish concurrent animations, just add the animations to the `Task.WhenAll`, `Task.WaitAny`, or `Task.WaitAll` methods. Depending on your scenario, you can mix and match concurrency.

Master Yoda in our example fades in, moves across the *X* axis of the screen, rotates, scales, and moves back to point $X=0$.

The first three animations (fade-in, rotate, and layout translation) are concurrent and when the layout finishes, we continue with another two scale and one layout animations, all awaiting serial execution.

The animating force is with you!

There’s more…

Fade is a simple animation on the view’s opacity and it makes things stable, but this might not be the case with all your scenarios; for example, when moving or scaling a view inside relative or stack layouts.

Choosing the correct layout is key for animations to work as expected. In this recipe, we worked with `StackLayout`, but the example is mostly linear: moving the view from point A to point B in the *X* axis of the screen, scaling it in a steady position, and then moving it back to point A from point B worked great. The problem comes for more complex layout operations, for example translating the *X* and *Y* position of a view from point A to point B.

To make sure your views are aligned correctly and the desired effect is achieved, or if you experience strange results, you need to change your layout container. The safest choice is `AbsoluteLayout`, which the position of views is free of relative view calculations and repositioning elements on the screen.

Chapter 10. Test Your Applications, You Must

In this chapter, we will cover the following recipes:

- Creating unit tests
- Creating acceptance tests with Xamarin.UITest
- Using the Xamarin.UITest REPL runtime shell to test the UI
- Uploading and running tests in Xamarin Test Cloud

Introduction

You definitely must! Testing your applications is very important. Is there anyone who never had bugs in core components of the layers of the application or UI behavior meet the requirements for all the features supported with one pass? We need to ensure that our code delivered is of high quality and the application does what the requirements acceptance criteria define.

In traditional testing, we have software testers perform testing on the device, but with automated testing the risk of missing a problem in a component is minimized; also, we don't fully depend on a tester's skills.

With test automation, we ensure that new code will work as expected and that we didn't break any existing part of the code even if we didn't work on that part. Remind you of something? I know. In mobile development, the problem is bigger with traditional testing: there are so many devices from different vendors with different characteristics and OS versions, so how will everything work as expected in all these different form factors? You have to take into consideration other factors as well: connectivity, service availability, and hardware device APIs or battery. Manually testing and covering these factors with abstractions and mocking frameworks is time consuming and error-prone.

There are two types of testing approaches using **test-driven development** (**TDD**): Inside-Out and Outside-In, known also as Classic school (bottom-up) and London school (top-down). In this chapter, we will learn how to create tests for both approaches, then take our acceptance tests and upload them to **Xamarin Test Cloud** and run them against real physical mobile devices.

Creating unit tests

Inside-Out testing, commonly known as unit testing, performs tests in individual components and its methods are completely independent of the UI. So, if you take a Classic-school architecture, bottom-up, you might have classes in layers like the data access repositories, models, viewmodels, and business logic components.

Unit tests are making sure that individual parts (units) of the application work independently, with any outside dependencies stubbed out and made irrelevant.

First, we will create a unit test project for the shared portable code and then two projects for specific platform device capabilities.

How to do it…

1. We'll start with Classic-school testing. In Visual Studio, go to top menu and select **File** | **New** | **Project**. Choose the **Blank App** (Xamarin.Forms Portable) template, name it `XamFormsUnitTesting` and click **OK**.
2. Right-click the `XamFormsUnitTesting` PCL and select **Add** | **Class…**, name it `ReverseService.cs` and click **Add**. The following is the code for the newly created class:

```
public class ReverseService
{
  public string ReverseWord(string word)
  {
    char[] arr = word.ToCharArray();
    Array.Reverse(arr);
    return new string(arr);
  }
}
```

3. Right-click the solution and select **Add** | **New Project…**. In **Templates**, select the **Test** section and choose **Unit Test Project**, give it the name `UnitTestCore` and click **OK**.
4. Right-click the `UnitTestCore` project and select **Add** | **Class…**. Name it `ReverseServiceTests.cs` and click **Add**. Find the code for the newly created class next:

```
[TestClass]
public class ReverseServiceTests
{
  [TestMethod]
  public void Should_Reverse_Word()
  {
    / Arrange
    string word = "ABC";
    string reversedWord = "CBA";
    ReverseService reverseService = new ReverseService();

    // Act
    string result = reverseService.ReverseWord(word);

    // Assert
    Assert.AreEqual(result, reversedWord);
  }
}
```

5. Right-click the `References` folder in the `UnitTestCore` project and choose **Add Reference**. Go to the **Projects** | **Solution** section, check the `XamFormsUnitTesting` PCL project and click **OK**.
6. Build the `XamFormsUnitTesting` and `UnitTestCore` projects.
7. In the Visual Studio menu, go to **Test** | **Windows** | **Test Explorer**. In the **Test Explorer** window, press the **Run All** button link; the test `Should_Reverse_Word` should become green.

8. A green checkmark indicator will also appear on top of the `Should_Reverse_Word` test method; clicking it will provide you with the results of the test completion.

9. Continuing, we will create two platform-specific libraries for testing. Right-click on the solution and select **Add | New Project…**. In **Templates**, select the **Test** section and choose **Unit Test App** (Android), give it the name `UnitTestDroid` and click **OK**.
10. We will not add any tests for this example, but go to the `TestSample.cs` file and explore all the `SetUp`, `TearDown`, and `Test` methods. Run the `UnitTestDroid` project in the simulator or device and click **Run Tests**; you will see tests information on the screen. Clicking on the screen, you can go to individual tests. In this project, you can add any platform-specific tests you want.

10:25
UnitTestDroid
Test Suites
Commands
Run Tests
Options
/storage/emulated/0/Android/data/
UnitTestDroid.UnitTestDroid/files/.__override__/
UnitTestDroid.dll
Failed
1 passed 1 failed 1 ignored1 inconclusive
One or more child tests had errors
for NUnitLite, see http://www.nunitlite.org

11. Right-click the solution and select **Add | New Project…**. In **Templates**, select the **Test** section and choose **Unit Test App** (iOS), give it the name `UnitTestTouch` and click **OK**.
12. At the time of writing this book, creating a **Unit Test App** (iOS) project in Visual Studio lacked the startup code to set a `TouchRunner ViewController` as `RootViewController`. Go to `AppDelegate.cs` and replace the `FinishedLaunching` method with the following code. This is an iOS-specific testing library.

```
TouchRunner runner;

public override bool FinishedLaunching(UIApplication application,
NSDictionary launchOptions)
  {
  // Override point for customization after application launch.
  // If not required for your application you can safely delete this
method

  // create a new window instance based on the screen size
  Window = new UIWindow (UIScreen.MainScreen.Bounds);
  runner = new TouchRunner (Window);

  // register every tests included in the main application/assembly
  runner.Add (System.Reflection.Assembly.GetExecutingAssembly ());

  Window.RootViewController = new UINavigationController
(runner.GetViewController ());

  // make the window visible
  Window.MakeKeyAndVisible ();

  return true;
}
```

13. Right-click the `UnitTestTouch` project and select **Add | Class…**. Give it the name `Tests.cs` and click **Add**. Next, find the contents of the class:

```
[TestFixture]
public class Tests {

  [Test]
  public void Pass ()
  {
    Assert.True (true);
  }

  [Test]
  public void Fail ()
  {
    Assert.False (true);
  }

  [Test]
```

```
  [Ignore ("another time")]
  public void Ignore ()
  {
    Assert.True (false);
  }
}
```

14. Run the `UnitTestTouch` project on the simulator or your device and click the **Run Everything** button.

Carrier
10:33 PM
Test Runner
UnitTestTouch.exe
1 success, 1 failure, 1 ignored
Run Everything
Options
Credits

How it works…

We created three unit test projects; the first one is a full .NET framework class library.

Note

My machine uses the current .NET 4.5.2 version.

Note that Visual Studio by default creates a project using **MSTest**. If you are using Xamarin Studio, **NUnit** is the default testing framework.

NUnit has an advantage that works in a PCL, but with the full .NET framework, you can use mocking frameworks; one of my favorites is **Moq**.

In our very simple sample component class, `ReverseService`, we have one method, `ReverseWord`. In the UnitTestCore project, we created a test class named `ReverseServiceTests` to test the unit under the testing component `ReverseService`.

Every test has to validate one or more conditions and is structured in three parts:

- Arrange: this is where we do any setup for our test like initialize some variables or instantiate your unit under a testing component passing any dependencies
- Act: is the stage that you are performing the test; this is the place to invoke the method you want to test and get results
- Assert: is where you verify that the conditions are met and the method works as expected; you should at least have one assertion

The test will be considered successful if it is passes and no unexpected exception is thrown.

Assert is a static class with many useful validation methods. Refer to the class documentation for a complete list: https://msdn.microsoft.com/en-us/library/microsoft.visualstudio.testtools.unittesting.assert.aspx.

We tested the PCL code with the .NET 4.5.2 testing class library. To test the platform-specific components and features, the **NUnitLite** framework is used, a unit testing framework that allows you to run unit tests on simulators or devices. There is a template for creating an Android unit test project and an iOS unit test project. Working with NUnitLite is the same as working with NUnit.

Be aware that simulators don't have all the device capabilities available; for example, you won't be able to test a sensor API that is not available.

Creating acceptance tests with Xamarin.UITest

Outside-In testing is the acceptance testing we are performing to make sure that the application meets the specifications. These tests start from the user interface testing and might go to the bottom of the architecture. These behavior tests are responsible for testing the user interface functionality and behavior. In this section, we will look into the Xamarin.UITest acceptance testing framework and create a cross-platform UITest for the Android and iOS platforms just like a user would interact with it.

Xamarin.UITest is based on NUnit, and on the open source framework Calabash, one of the world's most popular behavior-driven development acceptance frameworks for mobile applications. Beware, though, as these are not unit tests.

In this recipe, we are using Xamarin Studio on a Mac. Creating a Xamarin.UITest cross-platform project is supported in Visual Studio; however, you will be able to test your Android tests only. iOS UI testing is not yet supported in Visual Studio.

Note

Windows Phone is not supported by Xamarin.UITest. The team is looking into supporting the platform, but no specific ETA has been announced at the time of this writing.

How to do it…

1. Open Xamarin Studio on your Mac and go to **File | New | Solution…**. In the dialog that appears, choose the template in the section **Cross-platform | App | Xamarin.Forms App** and click **Next**.

2. In the next screen, you need to configure your Xamarin.Forms app. Set the app name to `XamFormsAcceptanceUITesting`. Target platforms by default are iOS and Android and the **Shared Code** option is **Use Portable Class Library**. Click **Next**.

3. Continuing in the next screen, configuring the project, uncheck the **Use Xamarin Insights** option. We will not be using this feature for this example. If you want to learn how to monitor your application for live insights, go to the *Using Xamarin Insights* recipe from Chapter 11, *Three, Two, One – Launch and Monitor*. Check the option **Add an automated UI test project** and click **Create**.

4. The solution is created. Go to the `XamFormsAcceptanceUITesting.UITests` project and expand the folder `Packages`. Right-click the `Xamarin.UITest` package and click **Update**.
5. Right-click the `XamFormsAcceptanceUITesting` PCL project and select **Add | New File…**. In the **Forms** template section, select **Forms Content Page Xaml**, give it the name `MainPage` and click **New**.
6. Open the newly created `MainPage.xaml` file and replace the contents with the following code:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
x:Class="XamFormsAcceptanceUITesting.MainPage"StyleId="mainPage">
    <ContentPage.Content>
      <Button x:Name="addContactButton"
        StyleId="addContactButton" HorizontalOptions="FillAndExpand"
VerticalOptions="Center" Text="Add Contact" Clicked="OnAddContactClick"
/>
    </ContentPage.Content>
</ContentPage>
```

7. Open the `MainPage.xaml.cs` behind-code file and add the `OnAddContactClick` event handler method.

```
private async void OnAddContactClick(object sender, EventArgs args)
  {
```

```
    await Navigation.PushAsync (new ContactPage ());
  }
```

8. Repeat step 5 for a **Forms Content Page Xaml** page named `ContactPage.xaml`. Add the following XAML contents to the newly created page:

```
<?xml version="1.0" encoding="UTF-8"?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
x:Class="XamFormsAcceptanceUITesting.ContactPage"
StyleId="addContactPage">
    <ContentPage.Content>
      <StackLayout Orientation="Vertical">
        <Label Text="Name:" />
        <Entry x:Name="nameEditText" StyleId="nameEditText" />
        <Label Text="Email:" />
        <Entry x:Name="emailEditText" StyleId="emailEditText" />
                <Button Text="Save" x:Name="saveButton"
StyleId="saveButton" Clicked="OnSaveClick" />
       </StackLayout>
    </ContentPage.Content>
</ContentPage>
```

9. Open `AddCotactPage.xaml.cs` and add the `OnSaveClick` event handler method.

```
private async void OnSaveClick(object sender, EventArgs args)
  {
    if (!string.IsNullOrWhiteSpace (nameEditText.Text) &&
string.IsNullOrWhiteSpace (emailEditText.Text)) {
      await Navigation.PopAsync (true);
      }
  }
```

10. Go to the `XamFormsAcceptanceUITesting.cs` file and replace the constructor with the following code:

```
public App ()

{
  MainPage = new NavigationPage(new MainPage());
}
```

11. Open the `Tests.cs` file created automatically in the `XamFormsAcceptanceUITesting.UITests` project and add the following tests:

```
  [Test]
  public void Add_Contact_Button_Should_Navigate_To_Contact_Page ()
  {
    // Act
    // Screenshot names are only supported in Test Cloud.
    app.Screenshot ("MainPage screen.");
    app.Tap (p => p.Button ("addContactButton"));
    AppResult[] results = app.Query (p => p.Marked ("addContactPage"));
    // Assert
    Assert.True (results.Any());
  }
  [Test]
```

```
  public void Save_Contact_Button_Should_Navigate_Back_To_MainPage()
  {
    // Act
// Screenshot names are only supported in Test Cloud.
    app.Screenshot ("MainPage screen.");
    app.Tap (p => p.Button ("addContactButton"));
    app.WaitForElement (p => p.Marked ("addContactPage"));
    app.EnterText (p => p.Marked ("nameEditText"), "George Taskos");
    app.EnterText (p => p.Marked ("emailEditText"),
"george@xplatsolutions.com");
    app.Tap (p => p.Button ("saveButton"));
    app.WaitForElement (p => p.Marked ("mainPage"));
    AppResult[] results = app.Query (p => p.Marked ("mainPage"));
    // Assert
        Assert.True (results.Any());
  }
```

12. Go to the `XamFormsAcceptanceUITesting.iOS` project and open the `AppDelegate.cs` file. In the `FinishedLaunching` method and after the `global:Xamarin.Forms.Forms.Init()` method call and inside the `ENABLE_TEST_CLOUD` if a directive condition, add the following code:

```
global::Xamarin.Forms.Forms.ViewInitialized += (sender, e) => {
  // http://developer.xamarin.com/recipes/testcloud/set-
accessibilityidentifier-ios/
  if (null != e.View.StyleId) {
    e.NativeView.AccessibilityIdentifier = e.View.StyleId;
  }
};
```

13. Go to the `XamFormsAcceptanceUITesting.Droid` project and open the `MainActivity.cs` file. In the `OnCreate` method and after the `global:Xamarin.Forms.Forms.Init(this, bundle)` method call, add the following code:

```
// http://forums.xamarin.com/discussion/21148/calabash-and-xamarin-
forms-what-am-i-missing
  global::Xamarin.Forms.Forms.ViewInitialized += (object sender,
Xamarin.Forms.ViewInitializedEventArgs e) => {
    if (!string.IsNullOrWhiteSpace(e.View.StyleId)) {
      e.NativeView.ContentDescription = e.View.StyleId;
    }
};
```

14. Rebuild the solution.
15. Start your favorite Android emulator. In this book, we test the Android applications in the Xamarin Android Player, Nexus 5 (Lollipop) (API 22) emulator.
16. In the Xamarin Studio top menu, click **View | Pads | Unit Tests**. The **Unit Tests** window will appear.

17. Right-click **Tests**(Android) and select **Run Test**. After watching the Android acceptance tests playing, repeat the same for **Tests**(iOS).

How it works…

The UITests are contained in a class library and described as NUnit tests. To execute the UITests against your app, it utilizes an automation library named Xamarin Test Cloud Agent.

Xamarin.UITest uses a client/server architecture that communicates over HTTP and JSON. There is a Xamarin Test Cloud Server installed in your device or simulator that drives the app using platform automation APIs. On Android, the server is a separate process signed with the same keystore as the app, which is bypassing the normal sandbox conditions. In iOS, the server is bundled into the application binary. This server component is started in the `AppDelegate.cs` file, the `FinishedLaunching` method.

Tip

As a best practice, you should use a conditional compilation directive so that you don't deploy the Calabash server to your final release binary. By default, Xamarin uses the directive `ENABLE_TEST_CLOUD`.

UITests are based on Calabash, and Xamarin.UITest provides a C# abstraction on top of the framework. You can find more information about the open source framework at http://calaba.sh/.

Note

Note that the acceptance tests can only test your application and cannot interact in any way with the system. You can't, for example, automate pressing the home button or going to the system settings and changing a value.

To drive the application when testing, we use the `IApp` interface. Xamarin provides two implementations of the interface, one for iOS and one for Android. The concrete implementation is provided to us via the `AppInitializer.StartApp(Platform)` static method.

The `IApp` interface is the bread and butter of Xamarin.UITest. To interact directly with your user interface, you can query the interface and tap a button, enter text in input fields, wait for an element to appear or disappear on the screen, apply gestures, and so on. Find the complete reference of the interface at the following link: https://developer.xamarin.com/api/type/Xamarin.UITest.IApp/.

You may have noticed that in the XAML code we set the `StyleId` property to identify the elements when we are querying the interface via the `IApp` interface. This is necessary to define the UI elements of the native-built application from Xamarin.Forms. Setting `StyleId` in Android will set `ContentDescription` and on iOS, the `Accessibility` identifier. This does not work out of the box, so a little bit of code is needed: an event handler to the static event `Xamarin.Forms.Forms.ViewInitialized` for both platforms in steps 12 and 13. This event will be raised each time an element is initialized and we set the corresponding property to the value of `StyleId` so that we can query the views appropriately.

In the **Unit Tests** window, you have the option to choose where you want to test the application: simulator/emulator or a device. Right-click a platform in the **Test Apps** section and choose the desired target. To change from simulator to a device in iOS, you will have to select **Debug | iPhone** in the project option.

Note

Note that to run the Xamarin.UITest acceptance tests in your device, you have to make sure that some settings are correct. You can find a guide that will instruct you with all the related information at the following link: https://developer.xamarin.com/guides/testcloud/uitest/working-with/testing-on-devices/.

There’s more…

You can use Xamarin.UITest to test any native Android or iOS application. You have the option to create a UITest project, write your tests, and run them on the binaries provided to you. You can configure this in the `AppInitializer.cs` file `StartApp` static method. There are three ways to identify the application we want to test against:

1. Specify the full path on the local disk to a built iOS or Android binary.

   ```
   ConfigureApp.Android.ApkFile ("../../path/myapp.apk");
   ConfigureApp.iOS.AppBundle ("../../path/mybundle.app");
   ```

2. Identify a specific package or bundle identifier for an installed application.

   ```
   ConfigureApp.iOS.InstalledApp ("com.packtpub.xamformscookbook");
   ConfigureApp.Android.InstalledApp ("com.packtpub.xamformscookbook");
   ```

3. Associate a project in the solution, which must be in the same folder as the UITest project (Xamarin Studio support only). This is the default when creating a solution in Xamarin Studio with the option to create a UITest project, like in this recipe.

Hybrid HTML applications are supported in the Xamarin.UITest framework; for example, you can query the interface using `IApp.CssClass("#mycssclass")`.

There is another helpful method in `AndroidAppConfigurator` and `iOSAppConfigurator` that enables local screenshot saving. `EnableLocalScreenshots` is a feature enabled by default in **Test Cloud**. If you want to enable this feature locally, invoke this method for each platform configurator in the `AppInitializer.StartApp(Platform)` method. This will save a screenshot in the format `screenshot-1.png` in your `bin` folder. Title is only available in **Test Cloud**. In this recipe, we invoke the `IApp.Screenshot(String)` method to capture screenshots.

See also

- https://developer.xamarin.com/guides/testcloud/uitest/intro-to-uitest/
- https://developer.xamarin.com/guides/testcloud/uitest/cheatsheet/

Using the Xamarin.UITest REPL runtime shell to test the UI

Creating your acceptance tests is challenging. Identifying the controls, structuring your commands, and replicating the desired user testing gestures is not easy, especially when you are testing a binary. Even if you are writing the source code and you have set your StyleId properties to your views, it’s impossible to remember them or stop and start a test until you have the correct commands in place.

To find all the information about the controls and interact with the interface in real time, there is a command-line tool named REPL, which stands for Read-Evaluate-Print-Loop.

In this recipe, we will see how in real time you can start a test project, query the interface for the available views, interact with the views, and copy the history in the memory to start off your test method.

How to do it…

1. Open Xamarin Studio on your Mac and go to **File | New | Solution…**. In the dialog that appears, choose the template in the section **Cross-platform | App | Xamarin.Forms App** and click **Next**.

2. In the next screen, you need to configure your Xamarin.Forms app. Set the app name to `XamFormsReplTest`. The target platforms by default are iOS and Android and the **Shared Code** option is **Use Portable Class Library**. Click **Next**.

3. Continuing in the next screen, configuring the project, uncheck the **Use Xamarin Insights** option. We will not be using this feature for this example. If you want to learn how to monitor your application for live insights, go to the *Using Xamarin Insights* recipe from Chapter 11, *Three, Two, One – Launch and Monitor*. Check the option **Add an automated UI test project** and click **Create**.

4. The solution is created, so go to the `XamFormsReplTest.UITests` project and expand the folder `Packages`. Right-click the `Xamarin.UITest` package and click **Update**.
5. Right-click the `XamFormsReplTest` PCL project and select **Add | New File…**. In the **Forms** template section, select **Forms Content Page Xaml**, give it the name `MainPage` and click **New**.
6. Open the newly created `MainPage.xaml` file and replace the contents with the following code:

```
<?xml version="1.0" encoding="UTF-8"?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
x:Class="XamFormsReplTest.MainPage">
    <ContentPage.Content>
      <StackLayout HorizontalOptions="FillAndExpand"
VerticalOptions="CenterAndExpand">
        <Entry StyleId="usernameEntry" x:Name="usernameEntry"
Placeholder="Username" />
        <Entry StyleId="passwordEntry" x:Name="passwordEntry"
Placeholder="Password" IsPassword="true" />
        <Button StyleId="logInButton" Text="Log In"
Clicked="OnLogInClick" />
      </StackLayout>
    </ContentPage.Content>
</ContentPage>
```

7. Open the `MainPage.xaml.cs` behind-code file and add the `OnLogInClick` event handler method.

```
private async void OnLogInClick(object sender, EventArgs args)
  {
    if (string.IsNullOrWhiteSpace (usernameEntry.Text) ||
string.IsNullOrWhiteSpace (passwordEntry.Text)) {
      await DisplayAlert ("Log In Error", "Username or password is
empty!", "OK", "Cancel");
    } else {
      await DisplayAlert ("Log In", "Nice!", "OK", "Cancel");
    }
}
```

8. Go to the `XamFormsReplTest.iOS` project and open the `AppDelegate.cs` file. In the `FinishedLaunching` method and after the `global:Xamarin.Forms.Forms.Init()` method call and inside the `ENABLE_TEST_CLOUD`, if a directive condition, add the following code:

```
global::Xamarin.Forms.Forms.ViewInitialized += (sender, e) => {
    // http://developer.xamarin.com/recipes/testcloud/set-
accessibilityidentifier-ios/
    if (e.View.StyleId != null) {
    e.NativeView.AccessibilityIdentifier = e.View.StyleId;
  }
};
```

9. Go to the `XamFormsReplTest.Droid` project and open the `MainActivity.cs` file. In the `OnCreate` method and after the `global:Xamarin.Forms.Forms.Init(this, bundle)` method call, add the following code:

```
// http://forums.xamarin.com/discussion/21148/calabash-and-xamarin-
forms-what-am-i-missing
  global::Xamarin.Forms.Forms.ViewInitialized += (object sender,
Xamarin.Forms.ViewInitializedEventArgs e) => {
  if (!string.IsNullOrWhiteSpace(e.View.StyleId)) {
    NativeView.ContentDescription = e.View.StyleId;
  }
};
```

10. Go to the `XamFormsReplTest.UITests` project and open the `Tests.cs` file. There is only one test created by default; rename it to the `DummyTest` method that just passes.

```
[Test]
public void DummyTest()
{
  Assert.IsTrue (true);
}
```

11. In the `Tests.cs` file, go to the `BeforeEachTest` method and add the method call `app.Repl()` like the following:

```
[SetUp]
public void BeforeEachTest ()
{
```

```
  app = AppInitializer.StartApp (platform);
  // REPL is ignored when running on the cloud
  app.Repl ();
}
```

12. Open the **Unit Tests** pad from the **View** | **Pads** | **Unit Tests** top menu, expand the tree, right-click the **Tests** (Android) and select **Run Tests**. This will start the Android application and open a terminal window.

13. Type `tree` and hit *Enter*. This will give you the interface controls hierarchy on the screen with plenty of information like the text and ID.

```
georgiostaskos — Xamarin.UITest REPL — mono /var/folders/0w/...
Full log file: /var/folders/0w/gvln4g0158722_pblktpc9gw0000gn/T/uitest/log-2016-
01-22_18-50-16-410.txt
Skipping IDE integration as important properties are configured. To force IDE in
tegration, add .PreferIdeSettings() to ConfigureApp.
Android test running Xamarin.UITest version: 1.2.0
Initializing Android app on device 10.71.34.101:5555.

App has been initialized to the 'app' variable.
Exit REPL with ctrl-c or see help for more commands.

>>> tree
[[object CalabashRootView] > PhoneWindow$DecorView]
  [ActionBarOverlayLayout] id: "decor_content_parent"
    [FrameLayout > ... > RendererFactory_DefaultRenderer] id: "content"
      [EntryRenderer]
        [EntryEditText] label: "usernameEntry"
      [EntryRenderer]
        [EntryEditText] label: "passwordEntry"
      [ButtonRenderer]
        [Button] label: "logInButton",  text: "Log In"
>>>
```

14. Write `app`. As soon as you hit the ‘ . ‘ character, you will notice that you get IntelliSense.

```
georgiostaskos — Xamarin.UITest REPL — mono /var/folders/0w/...
App has been initialized to the 'app' variable.
Exit REPL with ctrl-c or see help for more commands.

>>> tree
[[object CalabashRootView] > PhoneWindow$DecorView]
  [ActionBarOverlayLayout] id: "decor_content_parent"
    [FrameLayout > ... > RendererFactory_DefaultRenderer] id: "content"
      [EntryRenderer]
        [EntryEditText] label: "usernameEntry"
      [EntryRenderer]
        [EntryEditText] label: "passwordEntry"
      [ButtonRenderer]
        [Button] label: "logInButton",  text: "Log In"
>>> app.
Back, ClearText, Config, Device, DismissKeyboard, DoubleTap, DoubleTapCoordinate
s, DragAndDrop, DragCoordinates, EnterText, Equals, Flash, GetHashCode, GetType,
 Invoke, PinchToZoomIn, PinchToZoomInCoordinates, PinchToZoomOut, PinchToZoomOut
Coordinates, PressEnter, PressMenu, PressUserAction, PressVolumeDown, PressVolum
eUp, Print, Query, Repl, Screenshot, ScrollDown, ScrollDownTo, ScrollLeft, Scrol
lLeftTo, ScrollRight, ScrollRightTo, ScrollTo, ScrollToHorizontalEnd, ScrollToHo
rizontalStart, ScrollToVerticalEnd, ScrollToVerticalStart, ScrollUp, ScrollUpTo,
 SetOrientationLandscape, SetOrientationPortrait, SwipeLeft, SwipeRight, Tap, Ta
pCoordinates, TestServer, ToString, TouchAndHold, TouchAndHoldCoordinates, WaitF
or, WaitForElement, WaitForNoElement
```

15. Using the `Query` method and passing a selector, you can get more information for a specific interface control. Type `app.Query(p => p.Marked("logInButton"))` and hit *Enter*. Look at the information regarding the Android button widget.

```
georgiostaskos — Xamarin.UITest REPL — mono /var/folders/0w/...
        [Button] label: "logInButton",  text: "Log In"
>>> app.Query(p => p.Marked("logInButton"))
Query for Marked("logInButton") gave 1 results.
[
    [0] {
        Id => null,
        Description => "android.widget.Button{2d67589d VFED..C. ........ 0,0-108
0,144}",
        Rect => {
            Width => 1080,
            Height => 144,
            X => 0,
            Y => 1007,
            CenterX => 540,
            CenterY => 1079
        },
        Label => "logInButton",
        Text => "Log In",
        Class => "android.widget.Button",
        Enabled => true
    }
]

>>>
```

16. Tap the **Log In** button using the `Tap` command, `type app.Tap(p => p.Button("logInButton"))` and hit *Enter*. Notice that we get an alert dialog with the message that the **Username or Password is empty!** It's like we are actually interacting with the application using our fingers.

```
georgiostaskos — Xamarin.UITest REPL — mono /var/folders/0w/...
Full log file: /var/folders/0w/gvln4g0158722_pblktpc9gw0000gn/T/uitest/log-2016-
01-22_18-53-19-848.txt
Skipping IDE integration as important properties are configured. To force IDE in
tegration, add .PreferIdeSettings() to ConfigureApp.
Android test running Xamarin.UITest version: 1.2.0
Initializing Android app on device 10.71.34.101:5555.

App has been initialized to the 'app' variable.
Exit REPL with ctrl-c or see help for more commands.

>>> app.Tap(p => p.Button("logInButton"))
Using element matching Button("logInButton").
Tapping coordinates [ 540, 1079 ].
>>>
```

Android:

6:25
Us
Pa
Log In Error
Username or password is empty!
CANCEL
OK

17. Type the command `tree` again and you will notice that `button1` has the text `OK`. Type `app.Tap(p => p.Button("button1"))`, hit *Enter*, and the dialog goes away.

```
georgiostaskos — Xamarin.UITest REPL — mono /var/folders/0w/...
>>> tree
[[object CalabashRootView] > ... > FrameLayout]
  [FrameLayout] id: "content"
    [LinearLayout] id: "parentPanel"
      [LinearLayout] id: "topPanel"
        [LinearLayout] id: "title_template"
          [DialogTitle] id: "alertTitle" text: "Log In Error"
      [FrameLayout] id: "contentPanel"
        [ScrollView > LinearLayout] id: "scrollView"
          [TextView] id: "message" text: "Username or password is empty!"
      [LinearLayout] id: "buttonPanel"
        [Button] id: "button2" text: "Cancel"
        [Button] id: "button1" text: "OK"
>>> app.Tap(p => p.Button("button1"))
Using element matching Button("button1").
Tapping coordinates [ 873, 1065 ].
>>>
```

18. Now, enter some text to the input fields using REPL commands. Type `app.EnterText(p => p.TextField("usernameEntry"), "George")` and hit *Enter*. Notice the word `George` is automatically inserted into the username field.

Note

Note that if you have connected the simulator keyboard to your Mac keyboard, the iOS keyboard might not appear. If this is the case then the value will not be typed in the field. To avoid this happening, uncheck the **Connect Hardware Keyboard** option from the **Hardware** | **Keyboard** menu of the iOS simulator.

19. Type `app.EnterText(p => p.TextField("passwordEntry"), "aPassword")` and hit *Enter*. The password entry field is filled with the word `aPassword`.

```
georgiostaskos — Xamarin.UITest REPL — mono /var/folders/0w/...
    [LinearLayout] id: "parentPanel"
      [LinearLayout] id: "topPanel"
        [LinearLayout] id: "title_template"
          [DialogTitle] id: "alertTitle" text: "Log In Error"
      [FrameLayout] id: "contentPanel"
        [ScrollView > LinearLayout] id: "scrollView"
          [TextView] id: "message" text: "Username or password is empty!"
      [LinearLayout] id: "buttonPanel"
        [Button] id: "button2" text: "Cancel"
        [Button] id: "button1" text: "OK"
>>> app.Tap(p => p.Button("button1"))
Using element matching Button("button1").
Tapping coordinates [ 873, 1065 ].
>>> app.EnterText(p => p.TextField("usernameEntry"), "George")
Using element matching TextField("usernameEntry").
Tapping coordinates [ 540, 767 ].
>>> app.EnterText(p => p.TextField("passwordEntry"), "aPassword")
Using element matching TextField("passwordEntry").
Tapping coordinates [ 540, 921 ].
>>>
```

20. Let’s tap the button from the command line now, `app.Tap(p => p.Button("logInButton"))`, and notice the success dialog message now that the entry fields have values.

6:52
Log In
Nice!
CANCEL
OK

21. Type the command `copy` in the REPL runtime shell. This command will copy the history in the clipboard. Go to Xamarin Studio and paste the contents from your clipboard in `DummyTest`. This will give you a start for your test function.

```
[Test]
  public void DummyTest()
  {
    app.Tap(p => p.Button("logInButton"));
    app.Tap(p => p.Button("button1"));
    app.EnterText(p => p.TextField("usernameEntry"), "George");
        app.EnterText(p => p.TextField("passwordEntry"), "aPassword");
    app.Tap(p => p.Button("logInButton"));
    Assert.IsTrue (true);
  }
```

22. Go to the `BeforeEachTest` method and comment `app.Repl()`. Run the test again. You will see the acceptance-testing movie playing!

How it works…

To launch this runtime shell and interact with the connected application, you just need to invoke the `IApp.Repl()` method in the test. If you'd like to use the terminal for more than one test then add the `IApp.Repl()` call in the `BeforeEachTest` overridden method and after `AppInitializer.StartApp(platform)`, as we did in this recipe.

When you execute tests, the REPL runtime shell will appear with your application connected and ready to accept commands.

Xamarin.UITest has a rich API to accomplish almost everything as a user would normally interact with the application. You can find the documentation for the IApp interface at the following link: https://developer.xamarin.com/api/type/Xamarin.UITest.IApp/.

To identify elements on our interface, we have the `Query` method that accepts a selector This returns an `AppResults[]` array with zero or more results that match the predicate filter. The selector syntax is similar to the C# lambda expression. In the expression, you will usually filter for a specific view, passing the ID or text value in strongly typed methods like `Button(id)` or `TextField(id)`, using `Marked(id/text)` or `Id(id)`, though there are other ways to query elements like using class queries for specific types, `Class("UITextField")`.

Querying hybrid web applications are supported as well; you can use the `Id(id).Css("#cssclass")` method.

See also

- https://developer.xamarin.com/guides/testcloud/uitest/working-with/repl/

Uploading and running tests in Xamarin Test Cloud

Your tests are ready and are testing all your screens, setting text values, tapping buttons, and interacting with gestures. Everything is great, tested in the simulator and in your device. Powerful! But this is not enough. Even with two or three devices you might have, the nicest and most polished application you've finished is tested in a managed environment. Developers tend to forget about the changes they made to settings or the OS version is updated to the latest one. For Android, it gets even more complicated; scenarios such as different vendors with different versions of the OS, where some are updated some not, and various API levels on each device are encountered.

Xamarin Test Cloud is a cloud-based Acceptance Testing service where you can deploy your tests exactly as is from your cross-platform and execute them in parallel in over 2,000 physical different mobile devices. See the list of devices at the following link: https://testcloud.xamarin.com/devices.

Note

For this recipe, we will create a Xamarin.Forms project and deploy the tests directly from Xamarin Studio.

How to do it…

1. To start with Xamarin Test Cloud, you have to log in with your Xamarin account in https://testcloud.xamarin.com/. You can also register with a different account if this is what you want.

   ### Note

   Xamarin Test Cloud provides you with 60 minutes/month if you are a licensed Xamarin developer and a 25-hour time gift that will start counting after your monthly time if you are a Xamarin University subscriber. You may use your trial evaluation Xamarin account, or try a free demo at the following link: https://xamarin.com/test-cloud-demo. In this recipe, we will show how you can push your tests as if you have an account.

2. Open Xamarin Studio on your Mac and go to **File | New | Solution…**. In the dialog that appears, choose the template in the section **Cross-platform | App | Xamarin.Forms App** and click **Next**.

3. In the next screen, you need to configure your Xamarin.Forms app. Set the app name to `XamFormsTestCloud`. The target platforms by default are iOS and Android, and the **Shared Code** option is **Use Portable Class Library**. Click **Next**.

4. Continuing in the next screen, configuring the project, uncheck the Use Xamarin Insights option. We will not be using this feature for this example. If you want to learn how to monitor your application for live insights, go to the *Using Xamarin Insights* recipe from Chapter 11, *Three, Two, One – Launch and Monitor*. Check the option **Add an automated UI test project** and click **Create**.

5. The solution is created. Go to the `XamFormsTestCloud.UITests` project and expand the folder `Packages`. Right-click the `Xamarin.UITest` package and click **Update**.
6. Right-click the `XamFormsTestCloud` PCL project and select **Add | New File…**. In the **Forms** template section, select **Forms Content Page Xaml**, give it the name `MainPage` and click **New**.
7. Open the newly created `MainPage.xaml` file and replace the contents with the following code:

```
<?xml version="1.0" encoding="UTF-8"?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
x:Class="XamFormsTestCloud.MainPage" StyleId="mainPage">
    <ContentPage.Content>
        <Button StyleId="detailsPageButton" Text="Go to details"
Clicked="OnDetailsClick"
      HorizontalOptions="Center" VerticalOptions="CenterAndExpand" />
    </ContentPage.Content>
</ContentPage>
```

8. Open the `MainPage.xaml.cs` file and add the `OnDetailsClick` event handler.

```
private async void OnDetailsClick(object sender, EventArgs args)
  {
    await Navigation.PushAsync (new DetailsPage ());
  }
```

9. Right-click the **XamFormsTestCloud** PCL project and select **Add | New File…**. In the **Forms** template section, select **Forms Content Page Xaml**, give it the name `DetailsPage` and click **New**.
10. Open the newly created `DetailsPage.xaml` file and replace the contents with the following code:

```
<?xml version="1.0" encoding="UTF-8"?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
x:Class="XamFormsTestCloud.DetailsPage" StyleId="detailsPage">
      <ContentPage.Content>
        <Button StyleId="mainPageButton" Text="OK" Clicked="OnOkClick
"HorizontalOptions="Center" VerticalOptions="CenterAndExpand" />
    </ContentPage.Content>
</ContentPage>
```

11. Open the `DetailsPage.xaml.cs` file and add the `OnOkClick` event handler.

```
private async void OnOkClick(object sender, EventArgs args)
{
  await Navigation.PopAsync ();
}
```

12. Go to the `XamFormsTestCloud.iOS` project and open the `AppDelegate.cs` file. In the `FinishedLaunching` method and after the `global:Xamarin.Forms.Forms.Init()` method call and inside the `ENABLE_TEST_CLOUD`, if a directive condition, add the following code:

```
global::Xamarin.Forms.Forms.ViewInitialized += (sender, e) => {
    // http://developer.xamarin.com/recipes/testcloud/set-
accessibilityidentifier-ios/
    if (null != e.View.StyleId) {
    e.NativeView.AccessibilityIdentifier = e.View.StyleId;
    }
};
```

13. Go to the `XamFormsTestCloud.Droid` project and open the `MainActivity.cs` file. In the `OnCreate` method and after the `global:Xamarin.Forms.Forms.Init(this, bundle)` method call, add the following code:

```
// http://forums.xamarin.com/discussion/21148/calabash-and-xamarin-
forms-what-am-i-missingglobal::Xamarin.Forms.Forms.ViewInitialized +=
(object sender, Xamarin.Forms.ViewInitializedEventArgs e) => {
    if (!string.IsNullOrWhiteSpace(e.View.StyleId)) {
      e.NativeView.ContentDescription = e.View.StyleId;
    }
  };
```

14. In the `XamFormsTestCloud.UITests` project, open `Tests.cs` and paste the following `test` method:

```
[Test]
  public void Should_Navigate_To_Details_And_Back_To_MainPage ()
  {
    // Act
```

```
    app.Screenshot ("On mainPage");
    app.Tap(p => p.Button("detailsPageButton"));
    app.WaitForElement(p => p.Marked("detailsPage"));
    app.Screenshot ("Navigated to detailsPage");
    app.WaitForElement(p => p. Marked ("mainPageButton"));
    app.Tap(p => p.Button("mainPageButton"));
    AppResult[] results = app.WaitForElement(p => p. Marked
("mainPage"));
    app.Screenshot ("Navigated back to mainPage");
    // Assert
    Assert.True(results.Any());
  }
```

15. Open the **Unit Tests** pad from the top menu **View | Pads | Unit Tests**, right-click the test, and choose **Run Test** to run the unit tests in your simulator/emulator or devices for both platforms to ensure that everything is working fine. If you want to learn more about how to create and execute UITests, go to **Creating acceptance tests** with **Xamarin.UITest** and *Using the Xamarin.UITest REPL runtime shell to test the UI* recipe in this chapter.
16. From the configuration build option in the upper-left corner, choose **Debug | iPhone**.
17. To upload and execute the test in **Test Cloud**, open the **Unit Tests** pad and in the **Tests(iOS)** section, right-click and choose **Run in Test Cloud**.

18. A window will appear with the option **XamFormsTestCloud.iOS** selected. As you can see, **Release build** has to be selected to run Android tests in the cloud. Click **Upload and Run**.

19. Xamarin Studio will now compile the package and upload the tests to **Test Cloud**. You can see details in the **Test Cloud** output window that appears.

20. When the upload is complete, you will be transferred to the web browser with the option to select devices that you would like the tests to run on. In the following screenshot, you can see I selected top 10 phones. Click the button in the bottom-right corner to continue.

21. In the next screen, we will leave the default settings and click **Done**.

22. Next, your tests will prepare and run. When the tests are finished, you will also receive an e-mail.

23. Click on the test when finished and you will transfer to the **Overview** page.

24. Clicking the **Should Navigate To Details And Back To Main Page** test will show all the devices that the test is executed on. Select a device and you will get all the related insights. You can also select to see the screenshots from the left menu.

How it works…

To upload your Xamarin.UITest acceptance tests in Xamarin Test Cloud and choose to run them in the physical devices of your choice is as simple as right-clicking and selecting **Run in Test Cloud** on a test or multiple tests.

There is no special directives or code that is needed for your tests. The only option to take into consideration is that the build configuration has to be the same setup as a local device: Debug-iOS Device build for iOS and Release build for Android.

Xamarin Studio will take care of all the details of building and uploading the tests in Test Cloud. When this step is complete, you will be redirected to the Xamarin Test Cloud website and there is just the small matter of some configuration setup.

There’s more…

Test Cloud is organized by teams and members. You can add a new team or add members to an existing team by navigating to your **Account settings** by clicking your profile name in the top-right corner and then clicking the section **Teams & Apps** on the left menu.

You have the option to show the API key, configure an existing team, or create a new one. Each team can manage different apps and members.

You can add admins to manage teams. Click the section **Subscription overview**, and on the right of the page there is the option to create a new admin.

There are some CI systems that supports Test Cloud such as Team City, Jenkins, Visual Studio Online, or any other system with custom post-build commands. The Xamarin.UITest NuGet package includes command-line tools to upload and execute your tests; you can find more information at the following link:

https://developer.xamarin.com/guides/testcloud/uitest/cheatsheet/#Running_UITests_Using

See also

- https://xamarin.com/test-cloud/

Chapter 11. Three, Two, One – Launch and Monitor

In this chapter, we will cover the following recipes:

- Using Xamarin Insights
- Publishing iOS applications
- Publishing Android applications
- Publishing Windows Phone applications

Introduction

After all these character bytes, logic, best practices, reading, researching, meetings, and pursuing of the quality, time, and budget triangle, the time has come to release your creation to the world! Deployment it is.

Yes, it comes with that feeling of fear, as no application is ever released without bugs.

In this chapter, we will explore how to use the Xamarin Insights solution and track issues, send events, and extra data to gain a better understanding of how your application is performing and what your users like the most.

Next, we will go through the configuration to prepare your iOS, Android, and Windows Phone applications for deployment.

Using Xamarin Insights

Did you ever deploy an application and wonder how it works for the user? Is it performing well? Unhandled exceptions causing frustration to users? What features are the users using most?

It’s pretty scary when you know nothing about what’s going on after deploying the code to the world. You can of course depend on a custom support feature you have embedded in your application, but 95 percent of your users will not send you a message; it’s easier to uninstall it and give you a bad rating. Most probably you will never make it again on their home screen!

Worry not! Xamarin has a product in the family named Xamarin Insights, which is a solution for performance monitoring.

With Xamarin Insights, you can:

- Track unhandled exceptions and report handled exceptions
- Receive runtime device information
- Attach custom values to retrieve users or application information
- Send events to track anything you want
- Monitor network performance

Note

The preceding features describe the platform capabilities; however, there is a pricing structure when it comes to Xamarin Insights: all Xamarin subscribers get the basic plan with some features, and then you can upgrade (if it makes sense for your team) to a business account or to the enterprise plan. Learn more at the following link: https://xamarin.com/insights.

How to do it…

1. Open Xamarin Studio and create a new cross-platform solution from the top menu **File | New | Solution…**.
2. On the **New Project** initial window screen, select **Cross-platform | App | Xamarin.Forms App** and click **Next**.

3. Set the app name to `XamFormsInsights`, leave the default platforms to both iOS and Android, the default selection **Use Portable Class Library**, and click **Next**.

4. Next, choose a directory location for the project. Make sure to uncheck the option **Use Xamarin Insights**. We will set up Xamarin Insights manually and learn how it works. Click the **Create** button.

5. Go to https://insights.xamarin.com and log in with your Xamarin account.
6. After logging in, click the **New App** button on the right of the website.

7. You will be transferred to a page to enter a name for this insights application in the first step. Enter a name and click the **Create new app** button.

8. The second step with your API key appears. You will use this key in all your platform applications. The API key is unique and therefore different from the following screenshot:

9. Go to your application and add the `NuGet` package. For Xamarin Forms, you need to add the package to each project, including the PCL. In Xamarin Studio, right-click each project, select **Add | Add Nuget Packages…**, search for `Xamarin.Insights`, check the box, and click **Add Package**.

Note

If you have a signed application, you should use the `Xamarin.Insights.Signed` NuGet package.

10. Go to `XamFormsInsights.cs` and add the the API key as a public constant at the top of the `App` class.

```
public const string InsightsApiKey = @"<YOUR_API_KEY_HERE>";
```

11. Now, for each platform, you need to add the initialization code. Check the following iOS application entry point, `Main.cs`, where we initialize the component:

```
static void Main (string[] args)
{
  Xamarin.Insights.Initialize (App.InsightsApiKey);
  UIApplication.Main (args, null, "AppDelegate");
}
```

12. In Android, initialization happens either in the application class or in your main activity launcher in the `OnCreate` overridden method. There is an extra argument to pass the `Context` instance.

```
protected override void OnCreate (Bundle bundle)
{
  base.OnCreate (bundle);
  Xamarin.Insights.Initialize (App.InsightsApiKey, this);
  global::Xamarin.Forms.Forms.Init (this, bundle);
```

```
    LoadApplication (new App ());
}
```

13. In the Windows Phone project, you add the initialize method call in the `App.cs`, `App()` constructor, and it is the same as iOS. Refer to Chapter 1, *One Ring to Rule Them All* to learn how to add a Windows Phone project in an existing cross-platform solution created in Xamarin Studio.
14. Enable the Internet permission in Android. Right-click the `XamFormsInsights.Droid` project, click **Options**, select the **Android Application** section, find the **Internet** option in **Required permissions** and check the checkbox.

Note

Android requires version 4.0 – Ice Cream Sandwich (API level 14) or higher.

15. Don't leave the required permissions window. Although optional, you should allow Xamarin Insights to capture more details regarding your Android application. See the following list of permissions and what data Insights is enabled to retrieve.
    - `BindNotificationListenerService (BIND_NOTIFICATION_LISTENER_SERVICE)` – catch Android native crashes
    - `AccessNetworkState & AccessWifiState (ACCESS_NETWORK_STATE & ACCESS_WIFI_STATE)` – check the state of the network before making a request to the Xamarin Insights service

- `BatteryStats (BATTERY_STATS)` – get the battery statistics
- `ReadExternalStorage (READ_EXTERNAL_STORAGE)` – get the available storage of any external storage
- `ReadPhoneState (READ_PHONE_STATE)` – get the device ID

16. In Windows Phone 8.0/8.1 Silverlight versions only, the Windows RT API is different and doesn't require any capabilities to be enabled. You need to open `WMAppManifest.xml`, go to the **Capabilities** tab, find `ID_CAP_IDENTITY_DEVICE` and check it. With this in place, Insights is enabled to uniquely identify the device and hardware.
17. Let's try to catch unhandled and handled exception. Right-click on the `XamFormsInsights` PCL project, select **Add** | **New File…**, choose **Forms** | **Forms ContentPage Xaml**, set the name to `MainPage` and click **New**. Find the XAML code next:

```
<?xml version="1.0" encoding="UTF-8"?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
x:Class="XamFormsInsights.MainPage">
    <ContentPage.Content>
      <StackLayout HorizontalOptions="CenterAndExpand"
VerticalOptions="CenterAndExpand">
        <Button Text="Unhandled Exception"
        Clicked="OnUnhandledExceptionClick" />
        <Button Text="Handled Exception"
        Clicked="OnHandledExceptionClick" />
      </StackLayout>
    </ContentPage.Content>
</ContentPage>
```

18. Open `MainPage.xaml.cs` and add the event handlers.

```
private void OnUnhandledExceptionClick(object sender, EventArgs args)
  {
    int i = 0;
    var result = 1 / i;
  }
  private void OnHandledExceptionClick(object sender, EventArgs args)
  {
    try
    {
      string aString = null;
      aString.Equals("will_throw");
    }
    catch (NullReferenceException ex) {
      Xamarin.Insights.Report (ex);
    }
}
```

19. Open the `XamFormsInsights.cs` file and replace the constructor by setting a new `MainPage` instance to the `MainPage` property.

```
public App ()
{
```

```
    MainPage = new MainPage();
}
```

20. Run the application in your preferred platform and click the **Unhandled Exception** button. The application should crash immediately. You're in debug mode, so hit *command* + *return* to continue execution; the application exits immediately. Run the application one more time, which will give Xamarin Insights the chance to upload any pending cached reports. As a developer, you will receive an e-mail with the crash details.
21. Click the button **Handled Exception**. This will immediately send a caught exception report to Insights or it will be cached until an Internet connection is available.
22. Go to https://insights.xamarin.com and in the `XamFormsInsights` project, you will see two issues.

23. Click `DivideByZeroException` and it will transfer you to the details page.

24. Explore all the details available.

Note

`stacktrace` is the most important part of a crash and the nice thing with Xamarin Insights is that it provides the managed code `stacktrace`. Native `stacktrace` is supported and you might receive one if it wasn’t possible to bubble up the managed environment.

How it works…

Xamarin Insights is a full, detailed product of the Xamarin family for performance monitoring. Insights will help you track issues; collect runtime information like the operating system, version, and device type; and analytics like occurrences per day, users affected, and which version is top affected; add extra data; add custom events; identify users, and more.

To enable Xamarin Insights in your project, you need the client SDK, which is available via NuGet. Officially, the SDK supports Xamarin iOS/Android, Xamarin Forms and Windows Phone 8.0/8.1. Essentially, it is a PCL library connecting to a REST service and it can theoretically work with any .NET platform.

As a minimum, you need to have a valid Xamarin subscription to have access to the Xamarin Insights dashboard.

In the dashboard, you register an app, then you include the NuGet library in your applications and finally initialize and deploy with Xamarin Insights support.

When an event happens, the SDK will try to send it. If there is no Internet connection, it will be cached until network connectivity is available.

The SDK provides a large API to use. The complete documentation is at the following link: https://developer.xamarin.com/api/type/Xamarin.Insights/.

There’s more…

Xamarin Insights allows you to add custom information to handled exceptions by passing a `Dictionary<string, string>` as an argument in `Insights.Report(Exception, Dictionary<string, string>)`.

```
Xamarin.Insights.Report (ex, new Dictionary<string, string>
{
  { "userId", "user-id" }
});
```

To symbolicate the crash report, you need to map the symbol addresses with the `dSYM` bundle containing the DWARF debug information and it will be used to create a human-readable `stacktrace`.

Note

`dSYM` files store the debug symbols for your app. Xamarin Insights uses it to replace the symbols in the crash logs with the appropriate methods names so it will be human readable and make sense for the developer to understand the origin of the crash and debug the application. The benefit of using dSYM is that you don’t need to ship your app with debug symbols reducing the binary size.

You can manually upload the `dSYM` file, which is generated for every build of your application in Xamarin Insights, from the section **Settings | DSYMS**.

In the preceding screenshot, you can see another option to upload `dSYM` using the web API. Notice the `curl` command to the API address.

To find the `dSYM` file on a Mac, go to the `Debug/Release/bin` folder. If you are using Visual Studio on a Windows machine connected to a Mac host build server, go to **Tools | iOS | Show IPA file** on the build server.

You can turn off data collection through static properties on the Insights class. See the following properties and their description:

- `DisableCollection` – disables all Insights automated behavior (bool)
- `DisableDataTransmission` – turns off transmission to the dashboard (bool)
- `DisableCollectionTypes` – bit flag from `Insights.CollectionTypes` to turn off specific collection types

```
Insights.DisableCollectionTypes =  // Do not track:
  Insights.CollectionTypes.HardwareInfo  // Hardware device types
  | Insights.CollectionTypes.Jailbroken    // Jailobroken devices
  | Insights.CollectionTypes.Locale        // User locale
  | Insights.CollectionTypes.OSInfo        // OS version type
  | Insights.CollectionTypes.NetworkInfo    // Network connection
```

If you have a business or enterprise license, there are some additional features you can leverage:

- Track users
- Add additional user information
- Events tracking
- Third-party integrations (GitHub issues, Slack, HipChat)

For more information on the preceding features, refer to the official Xamarin Insights website: https://xamarin.com/insights.

Publishing iOS applications

After all this hard work coding and polishing your app, delivering requirements, and figuring out best practices for your new iOS project, the moment has arrived: your creation has to reach users' pockets, and this can be done by getting into the Apple Store submission process.

To submit an iOS application, it is necessary to have a Mac with Xcode installed, Xamarin iOS uses the same tool that Objective-C and Swift native applications use.

You also need an Apple Developer account; to enroll in the program, go to https://developer.apple.com/membercenter.

Following this recipe, you will learn how to prepare your application and package it for submission to the Apple Store.

How to do it…

1. You can start off with an already created Xamarin Forms cross-platform solution, but if you want to it is easy to just create a new one. In the Xamarin Studio top menu, choose **File | New | Solution…** and choose **Xamarin.Forms App** in the **Cross-platform | App** section. Click **Next**.

2. Next, choose an app name; we simply set it to `PublishDemo`. Also notice the organization identifier, which is important when publishing. Don't worry though, you can change it later. Click **Next**.

3. In the last screen, check the **Use Xamarin Insights** checkbox; it is always a good option to include insights in your application. To learn more about how to use Xamarin Insights, go to the section Using Xamarin Insights. Click **Next**.

4. Expand the `PublishDemo.iOS` project and double-click the `Info.plist` file. Here, you will find some application settings. This file is the same file with a native iOS application. Make sure the **Application Name**, **Bundle Identifier**, and **Version** are set, as these identify your app uniquely to the user and device. **Bundle Identifier** has to match your Apple distribution provisioning profile.

Note

`Info.plist` contains various information regarding the application like what devices you support, the deployment target SDK, icons, launch images, maps integration, and background modes. Make sure you provide all the application assets needed; the

application icon is very important for our example.

5. Set the deployment target to 6.0, which is the minimum that Xamarin.Forms uses. If you know that your application uses APIs that don't support a lower-version SDK then feel free to change this to your needs.
6. Double-click `PublishDemo.iOS` and go to the **iOS Build tab**. This tab has some important settings that you should configure. In **Configuration**, choose **Release mode**, and for **Platform**, choose **iPhone**.
7. In **Supported architectures**, choose **ARMv7 + ARM64**.
8. Now, check the **Use the LLVM optimizing compiler** checkbox and under that **Use Thumb-2** instruction set for **ARMv7** and **ARMv7s**.
9. In the iOS SDK version option, choose the latest one; in my case, I would choose 9.2, which is the latest.
10. In the **Linker behavior** option, choose **Link SDK** assemblies only.
11. Go to the **iOS Bundle Signing** tab. Here, you will have to choose **Signing Identity** and **Provisioning Profile**. Apple provides information about how to create provisioning profiles for distribution at this link: https://developer.apple.com/library/ios/documentation/IDEs/Conceptual/AppDistribut In a nutshell, find next the steps you will commonly follow to install a distribution profile in your machine. Of course, you must enroll to the developer program to be able to submit an application to the Apple Store.

    1. In the iOS Dev Center, https://developer.apple.com/membercenter, click **Certificates, Identifiers & Profiles**.
    2. In the **iOS Apps** panel, click **Provisioning Profiles**.
    3. Click **+**.
    4. Select **App Store** and click **Continue**.
    5. Select an **App ID** to associate with the provisioning profile and click **Continue**.
    6. To be able to use one development provisioning profile across multiple apps, select a wildcard App ID, if available.
    7. Select one or more certificates for production to include in the provisioning profile and click **Continue**.
    8. Only certificates for production are listed.
    9. Enter a name for the profile and click **Generate**.
    10. Click **Download** to download the provisioning profile.

12. Next, you need to set up an Apple Store entry in the iTunes website. Go to https://itunesconnect.apple.com. To learn more regarding iTunes Connect and how to set up an application, refer to the Apple documentation at https://developer.apple.com/library/ios/documentation/LanguagesUtilities/Conceptual CH12-SW1.
13. In the Xamarin Studio top menu, choose **Build | Archive for Publishing**. The setting of the build on the upper-left section of Xamarin Studio has to be **Release | iOS Device**. The list of the archives window will open.

14. Click **Sign and Distribute**, and in the modal window, choose **AppStore**.

15. Next, you will choose the distribution provisioning profile and click **Next**.
16. Click the **Publish** button and save the `.ipa` file on your hard disk.
17. After saving the file, you can choose **Open Application Loader** and click the button.
18. Log in with your Apple Developer credentials. Click **Deliver Your App** and click **Choose**. Select the **.ipa** file you previously saved on your hard disk and click **Open**.
19. Application Loader will connect to iTunesConnect and provide information that matches the application package. Click **Next** to upload the application.
20. You now need to return to https://itunesconnect.apple.com and find the build you just uploaded. Select the **build**, click **Save**, and then click **Submit for Review**.
21. There are a couple more steps answering questions and that's it, you are officially waiting for Apple to review your application and respond.

How it works…

First things first, no application can be submitted if you are not a registered Apple developer. There is a fee and with it the power of deploying your cool app ideas in the Apple Store is right in front of you.

It’s not the most straightforward process when it comes to submitting apps in the Apple Store. The identity part is especially cumbersome, but believe me, it is better than five years ago, and it gets better after a while working with the development and distribution profiles.

In application settings, choosing only ARM64 might make sense if you just want to support the latest devices; this may be something you want, but to target a larger audience we simply choose ARMv7 + ARM64, which will target 32 bit devices as well.

The LLVM compiler will generate code that will be smaller and perform better. Note that this option works only in release builds; it is not supported with the debugger.

Choosing the latest SDK is a requirement from Apple. It is not a good practice to leave as default, as this selects the latest and it may cause you build problems in case Apple releases an SDK version that might introduce exceptions to your application. Always build and test your application when Apple releases a new SDK to make sure there are no surprises.

This option has nothing to with compatibility; the SDK does not define the platforms the app will run. You can choose 9.2 and still have a deployment target of iOS 8.0. This option can be found in the `Info.plist` configuration file.

The linker checks all your code and chooses which methods are not used and should be removed from the build. The **Don’t link** option will not remove anything and results in a bigger size of the binary. **Link SDK assemblies only** will filter the SDK assemblies for the removal of redundant code, and **Link all assemblies** will go through everything that you’re using in the applications. This is pretty dangerous, especially if reflection is used in an assembly, as the linker will not detect which methods you are using and will remove them, which will of course cause crashes. The safest option to go with is what we’ve selected: **Link SDK assemblies only**.

Publishing Android applications

You did it! The Android world is the major market users in percentage in the world. It is polished and ready, so now you have to publish it to the store.

Don’t hurry. There are a couple of settings that need to be reviewed and configured, the distributed application package needs to be created, and then the package can be submitted to the store.

You may choose any Android store you want. In this recipe, we will introduce Google Play Store, which is Google’s official Android store, the most well known in the world.

How to do it…

1. You can start off with an already created Xamarin.Forms cross-platform solution, but if you want it is easy to just create a new one. In the Xamarin Studio top menu, choose **File | New | Solution…** and choose **Xamarin.Forms App** in the **Cross-platform | App section**. Click **Next**.

2. Next, choose an app name; we simply set it to `PublishDemo`. Also notice **Organization Identifier**, which is important for publishing. Don't worry though, you can change it later. Click **Next**.

3. In the last screen, check the Use Xamarin Insights checkbox, as it is always good to include insights in your application. To learn more about how to use Xamarin Insights, go to the *Using Xamarin Insights* recipe. Click **Next**.

4. Right-click the `PublishDemo.Droid` project and select the **Android Build** section.
5. Set **Configuration** to **Release**.
6. Check the **Enable ProGuard** checkbox.

7. Select the **Advanced** tab in the **Android Build** section. Here, select the CPU architecture binaries you want to include in the package, select **armeabi** and **armeabi-v7a** options, but bear in mind that you can always select another, for example if you are planning to target **arm64-v8a** CPU architectures like the new Nexus 9.
8. Select the Android Application section and make sure that you have set **Package name**. The package name must be lowercase, no spaces, and follow a reverse-DNS format.
9. Next, in the Application icon, select a PNG image from the list. This listing is from the `Resources/drawable` folder.
10. **Version number** is an integer and it is used as a build number. You can change this value to whatever makes sense for you; it is never shown to the user.
11. **Version name** is the release version identifier from which the user is able to identify the application version.
12. For **Minimum Android version**, set the minimum API level that your application supports.
13. In **Target Android version**, define what your application is tested and guarantee that runs best.

14. There is a third setting regarding the Android version. Select the **General** section, you will find at the top the **Target framework** option. Here, we usually select the latest released framework, which instructs the API level that your application will compile against.

15. We're ready to create a package. In the Xamarin Studio top menu, select **Build | Archive for Publishing** and wait until the **Archives** window is shown.
16. In the **Archives** window, select the archive created and click **Sign and Distribute…**.

17. There are two options: you can save the APK to your disk or upload it directly from Xamarin Studio to Google Play Store.

18. Click the **Ad-Hoc** option. You will need to create a new signing identity to move forward.
19. Click **Create a New Key** and fill out all the fields.

20. Click **OK**, select the newly created key and click **Next**.

21. This is the last screen to export an APK to the disk. Click **Publish**, choose a location to save the file, and then enter the key password of the signing identity.

22. Having the APK file, you can now manually upload it to the Google Developer Console, manually install it to your device, or send it via e-mail to beta testers.
23. To upload the application from Xamarin Studio to Google Play via the Xamarin Studio IDE, refer to the detailed guide in the following link that will take you through the process of the Google Developer Console: https://developer.xamarin.com/guides/android/deployment,_testing,_and_metrics/publ _publishing_an_application_on_google_play/.

How it works…

The Enable ProGuard setting will obfuscate and remove unnecessary code from your application byte code, meaning any unused methods from your binary and libraries that you are referencing will be stripped out creating a smaller application package. This option is supported only in **Release configuration** mode.

Choosing many architectures in you package will increase the size of the APK file. Consider the option in the **General** tab of the **Android Build** section, **Generate one package (.apk) per selected ABI**, if you want to create a package for each CPU architecture build.

We assume that you don’t have an existing key. If you already have one, you already have submitted or exported an application package before and you can reuse this key for all your applications.

You can install an APK manually on a device by copying the file to the device storage using the device file browser, the adb, or several GUI tools available on the Internet. Also, in **Device Settings | Security**, you need to enable the **Unknown sources** option for installation to begin.

There’s more…

Right-click the Android project and in the **Android Build** section, **Genera**l tab, there’s a setting, **Enable Multi-Dex**. Android applications contain executable byte code. The Dalvik executable file (`.dex`) format has a limitation of 65k of total native methods referenced. The **Enable ProGuard** setting will dramatically help reduce the application method references by removing unused code, but it is possible for an application to hit this limitation. Enabling this option will split the `.dex` file into separate files.

For enterprise license users only, Xamarin provides a feature that you can select in the **Android Build** section, **General** tab, **Embed assemblies in native code**. This option will take any .NET assemblies and wrap them into a native Android library that makes it harder for someone who wants to de-obfuscate and read your code.

On the **Android Build**, **General** settings, there is a section at the bottom named **Code Generation** with the option **Enable AOT** (Experimental), available to business subscribers. This is a feature that is in the testing phase, at the time of writing this book, by Xamarin. Xamarin.Android supports JIT code generation, and enabling this option means you will have pre-compiled code that results in higher performance and faster startup times. Bear in mind that with the new ART Android runtime, there is support for Ahead of Time compilation on the device, but this has no relation to the option discussed here. The Xamarin option is targeting the managed code of the application and ART is meant for the Android native part of the application.

See also

- https://developer.xamarin.com/guides/android/deployment,_testing,_and_metrics/publ

Publishing Windows Phone applications

Maybe the Windows Store is not as popular as its iOS and Android competitors, and the multitude of applications available is not so great, but this is an opportunity!

Now that you finished your Xamarin.Forms Windows Phone platform, let’s see how to prepare it for publishing.

Microsoft currently has some different phone/tablet versions starting from Windows Phone 8.0/8.1 Silverlight, Windows Phone 8.1 (WinRT), Universal Windows, Windows 8.1, and Windows 10.

The Visual Studio template, as of the writing of this book, will create Windows Phone 8.1 (WinRT), Windows 8.1, and Universal Windows projects along with a PCL core library, the iOS, and the Android projects. Xamarin additionally continues supporting the Windows Phone 8.0/8.1 Silverlight project types.

For this recipe, we will configure and package the Windows Phone 8.1 application, but the settings and packaging is the same for all the Windows platforms.

How to do it…

1. Start Visual Studio and in the top file menu, select **File | New | Project…**. In the **Cross-Platform** section, choose **Blank App** (Xamarin.Forms Portable), set the name to `PublishDemoPhone` and click **OK**.

2. Right-click the `PublishDemoPhone.WinPhone (Windows Phone 8.1)` project and select **Set as StartUp Project**.
3. From the top toolbar menu, change the configuration to **Release**.

4. Right-click the `PublishDemoPhone.WinPhone (Windows Phone 8.1)` project and select **Properties**.

5. In the **Application** tab, click the **Assembly Information…** button. Here, try to fill out as much metadata information as you can; it's straightforward what input is expected.

6. Microsoft took care of the build process for you and haven't left you with much to configure. The most important setting is in the **Build** tab, the **Optimize code** checkbox. Changing the configuration mode to **Release** notice that it is already checked by default.

7. Go to the **Application** tab and click the **Package Manifest…** button.
8. In the `Package.appxmanifest` **Application** tab, change **Display name** and **Description** to something more meaningful than a namespace.

9. Click the **Visual Assets** tab. Here, you will need to add the application icon, tile icons, badge logo, splash logo, and so on. Assets is a big topic and there are specific UX guidelines from Microsoft in detail. Click the **More information** link button, which will take you to the following link: https://msdn.microsoft.com/en-us/library/windows/apps/mt412102.aspx. The default images are located in the `Assets` folder of the project.
10. For the **Requirements and Capabilities** tabs during development and testing, you have already added what is needed. Just make sure you check one more time.
11. Go to the **Packaging** tab. Here, you should change **Publisher** display name and **Application version**.
12. Right-click `PublishDemoPhone.WinPhone` and select **Build**. Go to the `bin/Release` folder, grab the `.xap` file, and upload it to the Windows Store.

How it works…

Packaging an application in Visual Studio is simple when it comes to comparing with iOS and Android submission preparations of the binary.

Setting some metadata information takes five minutes. Adding the visual assets is more complicated, but fortunately, Microsoft has documented this part very well.

Tip

Pay attention to the **Optimize code** option we checked in the **Application** tab. This option will optimize your code and make it smaller and faster in runtime. This hasn't caused any problems in my experience so far, but if you notice that your application behaves differently in Release mode than Debug mode then you should consider unchecking it.

To have access in Windows Store and upload applications, you need to enroll to the Windows Developer program, which you can easily do by paying the one-time fee of $19/$99 USD (always subject to change) for single developer and enterprise licenses. See more details and information at the following link: https://dev.windows.com/en-us/programs/join.

There’s more…

Microsoft has provided two ways to beta test your application. It is always nice to fix any major issues detected by users that were not part of the development process before your actual deployment. You might of course want to regularly update the beta testers with a version of a beta release cycle.

One way is with Beta Publication, which supports sending the application via e-mail invitation in up to 10,000 devices.

An alternative is uploading the app to the Windows Store, but not make it visible in customers’ searches. This way, only users with a specified URL link can install the application.

Also, consider using the Windows App Certification Kit, a utility that you should use to validate that your application binary passes all the tests before submitting to the Windows Store. More details are at the following link: https://dev.windows.com/en-us/develop/app-certification-kit.

Windows Store is similar to Apple Store and Google Play. It has submission policies and manual live testing for the application submission process. Visit the following link for an overview of the policies: https://msdn.microsoft.com/en-us/library/windows/apps/dn764944.aspx.

- Have a good description for customers to download the app
- No customer misleading
- No abuse of the ecosystem or customers
- No offensive or illegal activity
- The application must be testable

Index

A

- acceptance tests
  - creating, with Xamarin.UITest / Creating acceptance tests with Xamarin.UITest, How to do it…, How it works…, There's more…
- Android 6.0 Marshmallow
  - URL / How to do it…
- Android applications
  - publishing / Publishing Android applications, How to do it…, How it works…
- Apple documentation
  - URL / How to do it…
- application types
  - about / Introduction
  - online / Introduction
  - offline / Introduction
- aspect oriented programming (AOP)
  - about / Introduction
- assert
  - about / How it works…
  - URL / How it works…

B

- Base Class Libraries (BCL)
  - about / Introduction
- bug, with hardware
  - URL / How to do it…

C

- Calabash
  - URL / How it works…
- code
  - sharing, between different platforms / Sharing code between different platforms, How to do it…, How it works…
  - data, binding / Binding data in code, How to do it…, How it works…
- collection
  - about / Introduction
  - displaying / Displaying a collection and selecting a row, How to do it…, How it works…
- common platform features
  - using / Using common platform features, How to do it…, How it works…
  - references / How to do it…, How it works…
- cross-platform animations
  - adding / Adding cross-platform animations, How to do it…, How it works…
- cross-platform login screen
  - creating / Creating a cross-platform login screen, How to do it…, How it works…, There's more…
- cross-platform plugins
  - creating / Creating cross-platform plugins, How to do it…, How it works…, There's more…
  - references / There's more…
- cross-platform solution
  - creating / Creating a cross-platform solution, Getting ready, How to do it…, How it works…
- CRUD
  - about / Introduction
- CRUD operations
  - performing, in SQLite / Performing CRUD operations in SQLite, How to do it…, How it works…
- custom renderers
  - used, for changing user interface / Using custom renderers to change the look and feel of views, How to do it…, How it works…

D

- data
  - binding, in code / Binding data in code, How to do it…, How it works…
  - binding, in XAML / Binding data in XAML, How to do it…, How it works…
- data binding
  - about / Introduction
- demo, Xamarin
  - URL / How to do it…
- dependency locator
  - using / Using the dependency locator, How to do it…, How it works…
  - references / How it works…
- devices, Test Cloud
  - URL / Uploading and running tests in Xamarin Test Cloud

E

- efficient network calls
  - performing / Leveraging native REST libraries and making efficient network calls, How to do it…, How it works…, There's more…
- event messenger
  - using / Using the event messenger, How to do it…
  - references / How it works…

F

- Facebook
  - URL / How to do it…
- Facebook and Google providers
  - authenticating with / Authenticating with Facebook and Google providers, How to do it…
  - for iOS project / For the Android project
  - for Android project / For the iOS project, How it works…
  - references / How it works…
- Flurl
  - URL / There’s more…
- Fusillade
  - URL / There’s more…

G

- Geolocator
  - references / How it works…
- gesture recognizers
  - adding, in XAML / Adding gesture recognizers in XAML, How to do it…, How it works…, There's more…
  - URL / See also
- gestures
  - handling, with native platform renderers / Handling gestures with native platform renderers, How to do it…
- Google
  - URL / How to do it…
- Google Developer Console
  - URL / How to do it…
- GPS location
  - obtaining / Getting the GPS location, How to do it…, How it works…
- grouping
  - adding / Adding grouping and a jump index list, How to do it…, How it works…
  - references / There's more…

I

- images, Xamarin.Forms
  - URL / How to do it…
- in-app photo
  - obtaining, with native camera page / Taking an in-app photo with the native camera page, How to do it…, How it works…
- Inversion of Control (IoC)
  - about / Adding a third-party Dependency Injection Container
- iOS applications
  - publishing / Publishing iOS applications, How to do it…, How it works…
  - references / Publishing iOS applications
- iOS Dev Center
  - URL / How to do it…
- items
  - adding / Adding, removing, and refreshing items, How to do it…, How it works…, There's more…
  - removing / Adding, removing, and refreshing items, How to do it…, How it works…, There's more…
  - refreshing / Adding, removing, and refreshing items, How to do it…, How it works…, There's more…
- iTunes website
  - URL / How to do it…

J

- jump index list
  - adding / Adding grouping and a jump index list, How to do it…, How it works…, There’s more…

L

- libraries, from Paul Betts
  - URL / There’s more…
- localization
  - adding / Adding localization, How to do it…, How it works…
  - references / There’s more…

M

- Model-View-Controller (MVC)
  - about / Architecture design with Model-View-ViewModel (MVVM) pattern
- Model-View-ViewModel (MVVM) pattern
  - architecture design, defining with / Architecture design with Model-View-ViewModel (MVVM) pattern, How to do it…, How it works…
  - references / There's more…
- modes, Xamarin.Forms
  - URL / How it works…
- Moq
  - about / How it works…

N

- native camera page
  - in-app photo, obtaining with / Taking an in-app photo with the native camera page, How to do it…, How it works…
- native pages
  - displaying, with renderers / Showing native pages with renderers, How to do it…, How it works…
- native platform renderers
  - gestures, handling with / Handling gestures with native platform renderers, How to do it…
- native REST libraries
  - leveraging / Leveraging native REST libraries and making efficient network calls, How to do it…, How it works…, There's more…
- notifications
  - displaying / Show and schedule local notifications, How to do it…, How it works…, There's more…
  - scheduling / Show and schedule local notifications, How to do it…, How it works…, There's more…
- NUnit
  - about / How it works…
  - advantages / How it works…
- NUnitLite framework
  - about / How it works…

P

- Parse application
  - URL / How to do it…
- patterns
  - used, for injecting dependencies to modules / Adding a third-party Dependency Injection Container
- PCL
  - references / How it works…
- photos
  - taking / Taking or choosing photos, How to do it…, How it works…
  - selecting / Taking or choosing photos, How to do it…, How it works…
- platform-specific gestures
  - attaching / Attaching platform-specific gestures, How to do it…, How it works…
- platform-specific values
  - XAML, configuring with / Configuring XAML with platform-specific values, How to do it…, How it works…
- Portable Class Libraries (PCL)
  - about / Introduction
- provisioning profiles, for distribution
  - URL / How to do it…

R

- Refit
  - URL / There’s more…
- renderers
  - native pages, displaying with / Showing native pages with renderers, How to do it…, How it works…
- REST (Representational State Transfer)
  - about / Consuming REST web services
- REST web services
  - consuming / Consuming REST web services, How to do it…, How it works…
- row
  - selecting / Displaying a collection and selecting a row, How to do it…, How it works…
- row template
  - customizing / Customizing the row template, How to do it…, How it works…, There’s more…
  - references / There’s more…

S

- shared SQLite data access
  - creating / Creating a shared SQLite data access, How to do it…, How it works…
- SQLite
  - CRUD operations, performing / Performing CRUD operations in SQLite, How to do it…, How it works…
  - about / How it works…
  - references / How it works…
- SQLite.Net-PCL
  - URL / How it works…
- stacktrace
  - about / How to do it…

T

- tabbed-page cross-platform application
  - creating / Creating a tabbed-page cross-platform application, How to do it…, How it works…
  - references / How it works…
- TabbedPage
  - about / How it works…
- test-driven development (TDD)
  - about / Introduction
- Test Cloud
  - references / There's more…
- third-party Dependency Injection Container
  - adding / Adding a third-party Dependency Injection Container, How to do it…, How it works…
  - references / How it works…
- trigger types
  - defining / How to do it…
  - Property trigger / How to do it…
  - Data trigger / How to do it…
  - Event trigger / How to do it…
  - Multi trigger / How to do it…
- two-way data binding
  - configuring / Configuring two-way data binding, How to do it…, How it works…
  - references / How it works…

U

- UI
  - testing, Xamarin.UITest REPL runtime shell used / Using the Xamarin.UITest REPL runtime shell to test the UI, How to do it…, How it works…
- UI behaviors and triggers
  - adding / Adding UI behaviors and triggers, How to do it…, How it works…
  - references / How it works…
- unit testing
  - about / Creating unit tests
- unit tests
  - creating / Creating unit tests, How to do it…, How it works…
- user interface
  - changing, custom renderers used / Using custom renderers to change the look and feel of views, How to do it…, How it works…
- user interface (UI)
  - about / Introduction

V

- value converters
  - using / Using value converters, How to do it…, How it works…
- ViewModel
  - about / How it works…

W

- Windows Phone applications
  - publishing / Publishing Windows Phone applications, How to do it…, How it works…, There's more…
- Windows Presentation Foundation (WPF)
  - about / Introduction

X

- Xamarin
  - URL / Getting ready
  - creating / How to do it…
  - references / How it works…
- Xamarin.Forms
  - about / Introduction, Introduction, Introduction
  - limitation / Introduction
  - built-in cell types / Customizing the row template
- Xamarin.Forms event messenger
  - using / Introduction
- Xamarin.UITest
  - acceptance tests, creating with / Creating acceptance tests with Xamarin.UITest, How to do it…, How it works…, There's more…
  - URL / How it works…, How it works…
  - references / How it works…, How it works…
- Xamarin.UITest REPL runtime shell
  - used, for testing UI / Using the Xamarin.UITest REPL runtime shell to test the UI, How to do it…, How it works…
- Xamarin Android Player
  - URL / How to do it…
- Xamarin Insights
  - using / Using Xamarin Insights, How to do it…, There's more…
  - URL / Using Xamarin Insights, How to do it…, How it works…, There's more…
- Xamarin Test Cloud
  - about / Introduction
  - tests, running / Uploading and running tests in Xamarin Test Cloud, How to do it…, How it works…, There's more…
  - tests, uploading / Uploading and running tests in Xamarin Test Cloud, How to do it…, There's more…
  - URL / How to do it…
  - references / There's more…
- XAML
  - configuring, with platform-specific values / Configuring XAML with platform-specific values, How to do it…, How it works…
  - references / How it works…
  - data, binding / Binding data in XAML, How to do it…, How it works…
  - gesture recognizers, adding / Adding gesture recognizers in XAML, How to do it…, How it works…, There's more…