THE EXPERT'S VOICE® IN MOBILE APPLICATION DEVELOPMENT

Xamarin Mobile Application Development

Cross-Platform C# and Xamarin.Forms Fundamentals

BRIDGE THE GAP BETWEEN .NET, IOS AND ANDROID FOR YOUR MOBILE APPS

Dan Hermes

Apress®

For your convenience Apress has placed some of the front matter material after the index. Please use the Bookmarks and Contents at a Glance links to access them.

Apress®

Contents at a Glance

Introduction

This book is a hands-on Xamarin.Forms primer and a cross-platform reference for building native Android, iOS, and Windows Phone apps using C# and .NET.

If you think of the Xamarin platform as a pyramid with Xamarin.Android and Xamarin.iOS at its base and Xamarin.Forms on top, that's what this book covers with C#. Mobile UI makes up the lion's share of the pyramid, and this book explores the important concepts, elements, and recipes using Xamarin layouts, controls, and lists.

The burning question in many new Xamarin projects is this: is Xamarin.Forms right for my project? This book covers the salient considerations in the comparison of the Xamarin.Forms option vs. a platform-specific approach with Xamarin.Android or Xamarin.iOS.

When you've reached the limits of what Xamarin.Forms can do out of the box, you'll want to customize your Xamarin.Forms controls by using custom renderers to leverage platform-specific features.

You'll also learn all of the key Xamarin UI navigation patterns: hierarchical and modal, drill-down lists, tabs, navigation drawer, and others. You can use the provided navigation code to build out the skeleton of just about any business app.

This book is a guide to SQLite data access. We'll cover the most common ways to access a SQLite database in a Xamarin app and how to build a data access layer (DAL). Once you have a database set up, you'll want to bind your data to your UI. You can do this by hand or use Xamarin.Forms data binding to bind UI elements to data sources. We'll cover many techniques for read and write data binding to both data models and to view models for a Model-View-ViewModel (MVVM) architecture.

Building an app requires more than a UI and data access; you'll also need to organize your code into a professional-grade architecture. We'll explore solution-building techniques from starter to enterprise to help you decouple your functional layers, manage your platform-specific code, and share your cross-platform classes for optimal code reuse, testability, and maintainability.

Who This Book Is For

If you're a developer, architect, or technical manager who can read C# examples to learn about cross-platform mobile development using the Xamarin platform, then this book is for you. C# developers will probably be most at home with this book because that's what I am, but I've made an effort to point out when Microsoft or .NET lingo is in use. The Xamarin platform has a way of bringing technologists from different backgrounds together.

How to Download Code Examples

All of the code for this book, the C# and Extensible Application Markup Language (XAML) solutions, can be found in two places online:

Apress web page for this book, on the Source Code/Downloads tab (www.apress.com/9781484202159)

GitHub at https://github.com/danhermes/xamarin-book-examples

XAML

This book was written in the same way that the Xamarin platform is built, code-first in C#, so all of the book examples are in C#. All of the C# UI examples were ported to XAML as well, and made available in the downloadable code. Look for the XAML boxes for tips on where to find them.

XAML The XAML version of this example can be found at the Apress web site (www.apress.com), or on GitHub at https://github.com/danhermes/xamarin-book-examples. The Xamarin.Forms solution for this chapter is ThisChapterSolution.Xaml.

The hardest decision I made in writing this book was not including XAML examples in the book proper. Including XAML would have meant doing away with much or all of the material on Xamarin.Android and Xamarin.iOS, topics that are indispensible for a complete understanding of the Xamarin platform. I chose to adhere to my mission for this book: cross-platform C# code-first coverage of the foundations of the Xamarin platform. That said, I understand that there is a strong need for good XAML documentation and examples. So although I wish that there had been enough time and room to include XAML examples in the text of the book, I'm proud to say that we were able to provide downloadable XAML equivalents for *all* of the C# UI examples.

Get Started with Xamarin.Forms Right Now!

No time for reading? Browse Chapter 2 for ten minutes, and then download the navigation code for Chapter 6. Rip off some of my Chapter 6 navigation patterns to use immediately in your app and get started coding right now. Leave the book open to Chapter 3 so you can build some layouts inside your navigation pages. Good luck!

Chapter Contents

All of the chapters in this book are cross-platform, weighted in favor of Xamarin.Forms. The UI chapters (Chapters 3–6) are written with Xamarin.Forms, Xamarin.Android, and Xamarin.iOS elements side by side in a mini-index at the beginning of the chapter to facilitate understanding of concepts across platforms, and to make it easier to consider custom renderers when you need them. The first part of those UI chapters is Xamarin.Forms, the second part is Xamarin.Android, and the third is Xamarin.iOS.

Chapter 1—Mobile Development Using Xamarin

An introduction to the Xamarin platform covering all the key topics in this book.

Chapter 2—Building Mobile User Interfaces (Xamarin.Forms Intro)

A Xamarin.Forms primer and a comparison of Xamarin.Forms vs. platform-specific approaches, such as Xamarin.iOS and Xamarin.Android. Covers Xamarin.Forms pages, layouts, and views.

Chapter 3—UI Design Using Layouts

Layouts help us organize the positioning and formatting of controls, allowing us to structure and design the screens of our mobile app.

Chapter 4—User Interaction Using Controls (Views)

Pickers, sliders, switches, and other mobile UI controls facilitate user interaction and data entry that is unique to the mobile user interface and differs from the PC/mouse interface, largely because of the use of gestures.

Chapter 5—Making a Scrollable List

Lists are one of the simplest and most powerful methods of data display and selection in mobile apps.

Chapter 6—Navigation

Navigation lets a user traverse an app, move from screen to screen, and access features. Hierarchical, modal, navigation drawers, drill-down lists, and other key patterns make up the core of mobile UI navigation. State management is the handling of data passed between screens as the user navigates through the app.

Chapter 7—Data Access with SQLite and Data Binding

SQLite is the database of choice for many Xamarin developers. Store and retrieve data locally by using SQLite.NET or ADO.NET. Using Xamarin.Forms data binding, fuse UI elements to your data models. Use the MVVM pattern by binding to a view model.

Chapter 8—Custom Renderers

Extend the stock Xamarin.Forms controls and take advantage of platform-specific UI feature sets while maintaining a cross-platform approach using custom renderers.

Chapter 9—Cross-platform Architecture

Architect your cross-platform application by managing platform-specific code. Project-level options in Xamarin include Portable Class Libraries (PCLs) and shared projects. Cross-platform coding techniques include conditional compilation, dependency injection (DI), and file linking.

How to Read This Book

This book covers quite a breadth of material, and there are a few ways to approach it. Here are the main navigation paths built into this book and how to use them. If you are interested in

- **Cross-platform Development: Xamarin.Forms, Xamarin.Android, and Xamarin.iOS**
  Read the book from cover to cover. Every word in this book was written for you. After you read Chapters 1 and 2, peruse the first few pages of Chapters 3–6 to understand the UI material covered in the book.
- **Xamarin.Forms**
  Read Chapters 1 and 2, and then read the first third of Chapters 3–6, which covers Xamarin.Forms UI. Then read Chapters 7, 8, and 9 for data access, custom renderers, and architecture.

- **Xamarin.Android**
  First, make sure you are up to speed on Android basics. See "Prerequisites" later in this Introduction. Then read Chapter 1 and the first half of Chapter 2. Read the middle section of Chapters 3–6, which covers the Android UI. Then read Chapters 7 and 9 for data access and architecture.
- **Xamarin.iOS**
  First, make sure you are up to speed on iOS basics. See "Prerequisites" later in this Introduction. Then read Chapter 1 and the first half of Chapter 2. Read the end section of Chapters 3–6, which covers the iOS UI. Then read Chapters 7 and 9 for data access and architecture.
- **General Reference**
  Read Chapter 1 and the first half of Chapter 2 to get oriented. Then peruse the first few pages of Chapters 3–6 to understand the UI material covered in the book and how it is organized. Use the beginning of each of these UI chapters as a cross-reference.

So many paths to pick from. This is a choose-your-own-adventure book.

CODE COMPLETE

There is a "Cliff's Notes" navigation path through this book too. If you just want the bottom line on a topic, find the section you're interested in and jump right to the *CODE COMPLETE* section. This is a complete code listing at the end of many (but not all) major topics. Many times all we want is a quick code recipe on a topic, and that's how to get it here in this book. If you need explanation about the code, turn back to the beginning of the section and step through the detailed construction of that code.

What Platform Am I Reading About Now?

Stay oriented while reading about different platforms by using the *platform headings* beneath major topic headings. Here's what the headings look like:

XAMARIN.FORMS

This heading denotes Xamarin.Forms topics.

ANDROID

This heading denotes Xamarin.Android topics.

IOS

This heading denotes Xamarin.iOS topics.

WINDOWS PHONE

This heading denotes Xamarin.Forms Windows Phone topics.

CROSS-PLATFORM

this heading denotes topics that are useful for Xamarin.Forms, Xamarin.iOS, and Xamarin.Android.

Look for the platform headings beneath the major topic headings and you'll always know what platform you're reading about.

Prerequisites

If you're mainly using Xamarin.Forms, you should be able to pick up this book without much else in the way of background, besides some C#. However, if you want to get serious about using Xamarin.Android, Xamarin.iOS, or Xamarin.Forms custom renderers, please note the following:

This is not an Android 101 primer. Although it provides some introduction to key topics and then moves on to "202" material, you'll need to consult other sources for mastery of Android and Xamarin.Android fundamentals. Consult a Xamarin.Android primer on topics such as the following:

- Creating a Xamarin.Android solution using Xamarin Studio or Visual Studio
- Xamarin Designer for Android
- Activities, XML layouts, and views
- The activity life cycle
- The basic UI controls: `TextView`, `EditText`, `Button`, and `ImageView`
- Fragments
- Images and screen sizes
- Local resources
- Gestures

This book is not an iOS 101 primer. Although it provides some introduction to key topics and then moves on to "202" material, you'll need to consult other sources for mastery of iOS and Xamarin.iOS fundamentals. Consult a Xamarin.iOS primer on topics such as the following:

- Creating a Xamarin.iOS solution using Xamarin Studio or Visual Studio
- Xcode Interface Builder or Xamarin Designer for iOS
- Storyboards and segues
- `UIView` and `UIViewController`
- The basic UI controls: `UILabel`, `UITextField`, `UIButton`, and `UIImageView`
- Images and screen sizes
- Local resources
- Gestures

What's Not In This Book

The Xamarin platform is a monumental project, spanning technologies and APIs of several operating systems. Writing about all of it in enough detail to be useful would require not hundreds but thousands of pages, which means that not everything could be addressed in one book. There is no coverage of the following:

- *Extensible Application Markup Language* (*XAML*) for Xamarin.Forms in the book proper, but there are complete, downloadable XAML code examples.
- *Integrated development environments* (*IDEs*) including Visual Studio and Xamarin Studio.
- *UI Designer* tools including Xcode Interface Builder, Xamarin Designer for iOS, and Xamarin Designer for Android.
- *Some introductory Xamarin.Android topics* (see "Prerequisites")
- *Some introductory Xamarin.iOS topics* (see "Prerequisites")

Windows Phone

Xamarin.Forms apps will run on Windows Phone. Windows Phone projects can be built to support Silverlight or WinRT. This book was written for Windows Phone Silverlight implementations, but much of what is written here applies to WinRT as well (and some WinRT differences are pointed out).

System Requirements

Whether you're developing on a Mac or Windows workstation, you will need to download and run the Xamarin unified installer at `http://xamarin.com/download`. This will allow you to install and configure the Xamarin platform including the Xamarin Android SDK, Xamarin iOS SDK, Xamarin Studio, and Xamarin's plug-in for Visual Studio, as appropriate.

Here are the OS and software requirements for Xamarin development.

Mac

These requirements must be met to build Xamarin apps on a Mac.

- Xamarin Studio 5+ is required to use Xamarin.Forms on OS X.
- To develop iOS apps:
  - Latest Xcode version
  - Mac OS X 10.9.3+ (Mavericks) or 10.10+
- Windows Phone apps cannot be developed on a Mac.

Windows

These requirements must be met to build Xamarin apps on a Windows workstation.

- Latest Xamarin Studio or Visual Studio 2012+ (not Express)
- Windows 7+
- For iOS development, a networked Mac is required.
- For Windows Phone development, you'll need the Windows Phone SDK.
- To use PCLs, you'll need Visual Studio 2013+ or the Portable Reference Library Assemblies 4.6. For PCLs using Xamarin Studio without Visual Studio, download the Portable Library Tools 2.

Xamarin.Forms

To use Xamarin.Forms, your app builds must target the following platforms.

- iOS 6.1+
- Android 4.0+
- Windows Phone 8 (at the time of this writing, and newer versions later)

Errata

The author, the technical reviewers, and many Apress staff have made every effort to find and eliminate all errors from this book's text and code. Even so, there are bound to be one or two glitches left. To keep you informed, there's an Errata tab on the Apress book page (www.apress.com/9781484202159). If you find any errors that haven't already been reported, such as misspellings or faulty code, please let us know by e-mailing support@apress.com.

Customer Support

Apress wants to hear what you think—what you liked, what you didn't like, and what you think could be done better next time. You can send comments to feedback@apress.com. Be sure to mention the book title in your message.

Contacting the Author

You can follow me on Twitter at @lexiconsystems, read my blog at www.mobilecsharpcafe.com, or e-mail me at dan@lexiconsystemsinc.com.

If you are seeking general Xamarin product support, please use the Xamarin support page at http://xamarin.com/support or the Xamarin forums at http://forums.xamarin.com/.

Summary

That is everything you need to get started with this book: contents, prerequisites, disclaimers, trailheads, maps, and signposts. If you're new to the Xamarin platform, welcome to the community! If you're experienced with Xamarin, thanks for reading, and there is plenty in here for you as well.

I promise you that reading this book, understanding the concepts, and practicing the code techniques herein will enrich your Xamarin acumen and raise your coding skills to greater heights. I also hope to give you a deeper appreciation for the amazing accomplishment that is the Xamarin platform.

CHAPTER 1

Mobile Development Using Xamarin

Mobile development in C# is unlike anything most of us have done with the language before. We are using it to develop apps for non-Windows platforms, namely Android and iOS. This is both an opportunity and a challenge. The opportunity is to expose ourselves to the rich breadth of technology that comprises the new business application ecosystem made up of phones and tablets of different platforms and sizes. The challenge is that so much about these devices and platforms is new to many of us, and there is much to learn. Of course, we can build Windows Phone and tablet apps in C# also. The essence of cross-platform development is building apps that will work on more than one mobile operating system: for example, on Android and iOS; or on iOS and Windows Phone; or on iOS, Android, *and* Windows Phone. Using the cross-platform techniques covered in this book, you will be equipped to develop for all the major mobile platforms!

The most exciting/terrifying part of this journey is learning the ins and outs of several operating systems. Lucky for us, Xamarin shields us from many of the details, wrapping platform-specific APIs and exposing familiar .NET using C#. Conversely, exposed to us in detail are C#-wrapped user interface (UI) APIs for each platform, giving us precise control over the visual design of our app. The trick is to understand which aspects of each operating system are important during development and which can be left up to Xamarin. Although it never hurts to delve deeper in our understanding, there are only so many hours in a day, and the bottom line is that we need to ship working software.

These are the key questions: How do we approach the development of a cross-platform mobile application? Given the history and background that we already have in C# development, how do we carry forward this knowledge and leverage it in the mobile space? Finally, given the multitude of things to learn about these operating systems, what do we need to get started and help solve the important challenges?

While writing apps for more than one platform, a key goal is the reuse of code. The more we reuse, the quicker and cheaper our projects become, and the more we lower our maintenance costs. Xamarin refers to this as the *unicorn* of mobile development: *write once, deploy anywhere.* Any quest for a unicorn begins with a fair maiden to entice it to appear. Our fair maiden is cross-platform design.

Let's explore how Xamarin helps us solve our mobile puzzles while pursuing cross-platform design.

What Is Xamarin?

Xamarin is a development platform that allows us to code native, cross-platform iOS, Android, and Windows Phone apps in C#.

How does it do that? Read on.

Wrapped Native APIs

Descended from the open source Mono Project that brought .NET to Linux, the Xamarin platform is a port of .NET to the iOS and Android operating systems with support for Windows Phone (see Figure 1-1). Underlying Xamarin.Android is Mono for Android, and beneath Xamarin.iOS is MonoTouch. These are C# bindings to the native Android and iOS APIs for development on mobile and tablet devices. *This gives us the power of the Android and iOS user interface, notifications, graphics, animation, and phone features such as location and camera—all using C#*. Each new release of the Android and iOS operating systems are matched by a new Xamarin release that includes bindings to their new APIs. Xamarin's port of .NET includes features such as data types, generics, garbage collection, Language-Integrated Query (LINQ), asynchronous programming patterns, delegates, and a subset of Windows Communication Foundation (WCF). Libraries are managed with a linker to include only the referenced components. Xamarin.Forms is a layer on top of the other UI bindings and the Windows Phone API, which provides a fully cross-platform UI library.

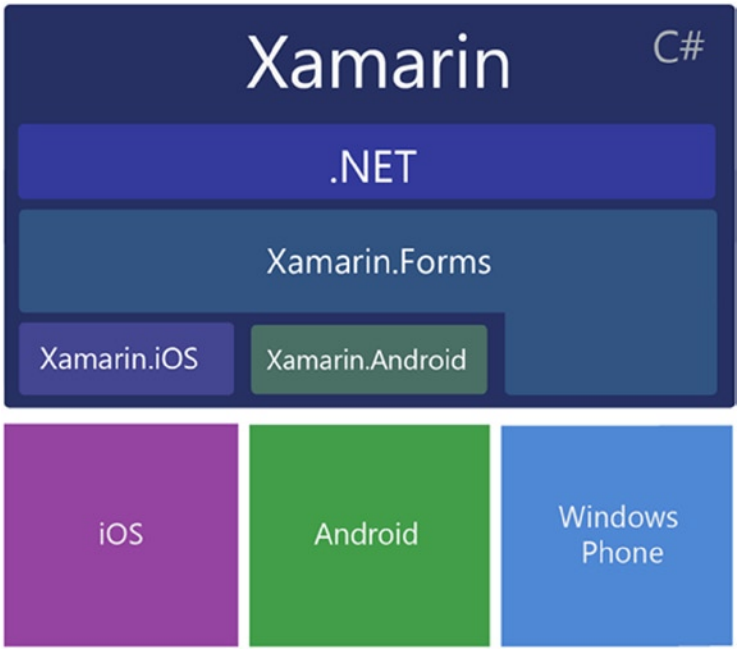

Figure 1-1. *Xamarin C# libraries bind to native OS SDKs as well as .NET*

So we have a .NET environment with iOS and Android C#-bound libraries with support for Windows Phone running on the mobile OS of our choice. Fantastic. Now how do we build UIs and write code using these libraries to build mobile apps? By using development environments and UI designers, of course.

Development Environments

Xamarin provides development environments and designers to help us build mobile apps on Windows or Mac. The two main choices for Xamarin development environments are Xamarin Studio on Mac or Windows, or Visual Studio on Windows with the Xamarin for Windows plug-in. A Mac is always required for the compilation of iOS apps, even if Visual Studio is used as the development environment.

UI Designers

The tools we use to create mobile user interfaces are called designers. These generate Extensible Markup Language (XML) files in their respective proprietary file formats. Two designers are available from Xamarin:

- Xamarin Designer for Android
- Xamarin Designer for iOS

With the availability of these designers, the need for the original, native XML editors has diminished. Anything you might need to build Android or iOS UIs can be found in Xamarin's tools. However, iOS developers still frequently use the Xcode Interface Builder, and Android developers (less frequently) use XML editors such as the Android Development Tools (ADT) plug-in for Eclipse. An XML layout is an XML layout, and the tool is largely a matter of taste and personal preference, even the decision to use a designer tool at all. Some Xamarin developers are opting to code UIs by hand in C# for all platforms with no designer use whatsoever. I recommend the designers to help learn the file formats, UI elements, and their properties. At the very least, use a designer tool like training wheels until you're ready to ride freestyle.

■ **Note** This book focuses on code rather than tools. Refer to the Xamarin online docs for more information on the designers and development environments at http://developer.xamarin.com.

What's Old: Familiar C# and .NET Techniques

Xamarin development allows us to leverage many things that we already know about C# development. We can use our high-level knowledge of the following:

- HTML-based pages
- Extensible Application Markup Language (XAML)
- UI controls
- Event-driven logic
- View life cycles
- State management
- Data binding
- Web services

We also can use many .NET-specific techniques directly and immediately, including these:

- .NET data types
- C# classes, methods, and properties
- Lambda expressions
- WCF (a subset)
- Generics (a subset)
- Local file access

- Streams
- Async/Await
- ADO.NET (a subset)

I've named just a few, so you can see there's plenty of familiar ground for the C# developer to help us make the leap into this new territory.

What's New: Mobile Development Techniques

Throughout this book, you will explore common C# techniques and patterns in mobile development. Some of them are exactly the same as the approaches we are accustomed to in traditional .NET development, some changed a little, and a few changed a lot. Here are the key topics:

- *Mobile UI* is the largest area of new learning for C# mobile development. Xamarin.Forms provides a cross-platform UI toolkit containing ready-to-use forms, pages, layouts, views, and controls. Xamarin.iOS and Xamarin.Android provide bindings to their respective native UIs. (See Chapters 2–6 and Chapter 8.)
- The *Data Access Layer* in a mobile app typically binds controls and pages to data models populated from a local database that is synced with a remote data server using web services. (See Chapter 7.)
- *Local database access* via SQLite is a change from the usual database vendors, though ADO.NET access provides a familiar inroad and the SQLite.NET component is a featureful option. (See Chapter 7.)
- *Data binding* is central to Xamarin.Forms development and is often accomplished using the Model-View-ViewModel (MVVM) pattern. (See Chapter 7.)
- *Cross-platform architecture* is a collection of code-sharing strategies to further our goal of *write once, run anywhere*. These include Portable Class Libraries (PCLs) with dependency injection, shared files and projects, and conditional compilation. (See Chapter 9.)

Let's touch on each of these topics.

Mobile UI

You have a formidable amount of new material to absorb when moving from web development to cross-platform mobile development, mostly in the area of the user interface. There is first the issue of new operating system UI APIs. Xamarin helps with this by providing platform-specific C# bindings to the major OSs with Xamarin.Android and Xamarin.iOS, while Xamarin.Forms provides cross-platform bindings to both of these plus Windows Phone. The other set of challenges involves the design differences between web apps and mobile apps. The compact screen, touch sensitivity, and handheld form factor team up to offer a fundamentally novel user experience (UX). This requires a fundamentally novel approach to design and development, leading us to explore mobile UI design.

Xamarin.Forms and Platform-Specific UI

There are two main approaches to mobile UI development using C#, which we can use separately, interchangeably, or in tandem:

- *Xamarin.Forms* is a cross-platform UI toolkit for Android, iOS, and Windows Phone.
- *Platform-specific UI* uses Xamarin.Android, Xamarin.iOS, and Windows Phone SDK.

Xamarin.Forms contains a fully cross-platform toolkit providing a single set of UI controls, layouts, and pages that map cleverly to respective native UI bindings on iOS, Android, and Windows Phone. Since Xamarin.Forms is newer than platform-specific libraries, it is also less full-featured. Each release brings us closer to full-featured cross-platform goodness, but sometimes we need more than the out-of-the-box Xamarin.Forms classes have to offer. In those cases, we use the platform-specific libraries, either for the entire page or for just parts of a page using Xamarin.Forms custom renderers called `PageRenderers`.

The platform-specific approach is older and more established and therefore quite detailed and full-featured. This involves libraries that bind directly to platform-specific UI APIs: Xamarin.Android for Android, and Xamarin.iOS for iOS. For Windows Phone, we use the Windows Phone SDK, a native API requiring no Xamarin bindings. These platform-specific libraries give us deep access to native UIs for providing a visually stunning, interactively rich user experience. This comes at a cost: platform-specific code requiring a separate UI project for each platform with little code reusability.

■ **Note** Xamarin.Forms is the thrust of this book, augmented by custom renderers, which use platform-specific UI. However, developers creating platform-specific UI projects using Xamarin.iOS and Xamarin.Android without Xamarin.Forms can make excellent use of the iOS and Android sections in Chapters 2–7. If you are taking the platform-specific approach, be sure to consult other sources (such as the Xamarin online documentation) for Xamarin.Android and Xamarin.iOS solution setup and fundamentals.

Mobile UI Design

UI techniques make up the core of most mobile software development. Current hardware limitations of small devices encourage us to leave the heavy lifting to the PCs and servers on the far end of our web services. Most of the components running in a mobile business application are there to support the visible user interface. Mobile business and data access layers are often abbreviated versions of their server-side brethren. That means that what we need most often are UI components to help us design screens using layouts, implement controls for data entry and selection, build lists and tables for data display and editing, create user navigation, and use images for backgrounds and icons. The UI topics in this book cover the functions used most frequently in mobile app development. In each chapter, we start with the simplest, most cross-platform approaches available, and then delve into platform-specifics for granularity and detail. These are the mobile UI topics are covered in this book:

- *Screens, views, or pages* are similar to the web and desktop equivalents in C#, using controls with methods and properties and firing events that we handle in our controllers. (See Chapter 2.)
- *Layouts* help us organize the positioning and formatting of controls, allowing us to structure and design the screens of our mobile app. (See Chapter 3.)

- *Controls* facilitate user interaction and data entry which is unique to the mobile user interface and differs substantially from the PC/mouse interface, largely due the use of gestures. (See Chapter 4.)
- *Lists* are one of the most powerful methods of data display and selection in mobile apps. (See Chapter 5.)
- *Navigation* lets a user traverse an app, move from screen to screen, and access features. Hierarchical navigation, modal screens, navigation drawers, alerts, drilldown lists, and other key patterns make up the core of mobile UI navigation. (See Chapter 6.)
- *State Management* is the handling of data passed between screens as the user navigates through the app. (See Chapter 6.)
- *Images* are central to the mobile experience, in menus, lists, grids, carousels, and other layouts. (See Chapters 2, 4, 6.)

Xamarin.Forms Custom Renderers

Custom Renderers allow us to go deeper than the out-of-the-box Xamarin.Forms UI controls and take advantage of platform-specific UI feature sets while keeping a cross-platform approach. Xamarin.Forms applications are inherently cross-platform, running on all three major platforms using the same code base. This works well for basic designs and using certain controls. However, many projects will develop a need to go deeper with the UI, such as design nuances on a single control, native modal dialog boxes, additional graphics or animations on a page, or any requirements that go beyond the scope of what Xamarin.Forms provides in the current release. This is accomplished by subclassing native controls and implementing `PageRenderers` to create custom controls that give full access to platform-specific UI functionality using Xamarin.iOS and Xamarin.Android. These platform-specific controls can be employed within Xamarin.Forms pages to help maintain a cross-platform architecture.

Data Access Layer

The mobile data access layer departs from the designs we are accustomed to in web apps and more closely resembles those found in desktop apps. Approaches range in sophistication from the popular MVVM pattern to Model-View-Controller (MVC) to basic CRUD (Create/Insert/Update/Delete). Data-bound pages typically feed into a local database on the device, which syncs with a remote data server using web services.

Web services in C# mobile development are a foundational aspect of code reuse. Many of these service patterns remain the same for mobile applications as what we are accustomed to when building web apps. However, mobile web services more closely resemble those found in desktop applications, differing from web app services primarily in the importance of data synchronization and offline use. Create, Update, and Delete interfaces are exposed online for RESTful calls from a multitude of platforms and devices. The fairest of maidens dwell here in mobile web service patterns, a perfect place for *cross-platform, write-once* code.

The server-side component of web services remains the same for mobile applications, compared to what we are accustomed to with web apps—except for the addition of data synchronization with local mobile data stores for both online and offline use. Offline use requires a basic data set to be kept on hand in the local database and synced when a connection is available. Not all apps support offline use.

Recounting our experiences with data access layers in older, related technologies, we will explore the architectural options for the data access layer in C# mobile apps.

Local Data Access Using SQLite

SQLite holds the title as the most stable and reliable cross-platform database product for mobile development, an open source project that works on iOS, Android, and Windows devices. Xamarin recommends it over a number of other third-party projects, and it is the one covered in this book. Xamarin provides access to and creation of SQLite databases within the development environment, provisions ADO.NET with support for SQLite. There is also a SQLite.NET Xamarin component, a C# wrapper around the C-based SQLite data layer offering low-level access to a SQLite database which includes async transactions. All this makes it easy and painless to connect to the database, create and index tables, and read and write rows.

Data Binding

Data binding is consistent and cross-platform when using Xamarin.Forms. Modeled after data binding in Windows Presentation Foundation (WPF), the MVVM pattern is central to its implementation. In code we bind control fields to our data model and the binding mechanism is automatic. A manual implementation of a `PropertyChanged` event allows your code to stay in sync with the data source. Binding is done in code or in Extensible Application Markup Language XAML and can be one- or two-way. Controls, lists, and text are tethered to a data source or to one another's properties. A growing number of third party vendors are providing Xamarin. Forms control suites which include data-bound charts and grids, such as Telerik, Infragistics, Syncfusion, and DevExpress.

Data-binding is not built into Xamarin.Android and Xamarin.iOS. Platform-specific implementations of data binding are typically achieved using open-source third party libraries such as MvvmCross and MVVM Light.

Cross-platform Development

In the same way that .NET provided a unifying infrastructure spanning many operating systems and languages, Xamarin bridges the gaps between mobile operating systems and their respective development languages: iOS and Objective-C, Android and Java, and Windows Phone and tablets and C#. In this way, Xamarin extends .NET into the mobile realm, far beyond Windows operating systems. Aside from the fact that this is eminently cool, the real value here is the opportunity to share and reuse code between and across projects and platforms. The greatest benefits of Xamarin tools are found in the cross-platform code; therefore, a cross-platform approach to mobile patterns will produce the highest yield. Xamarin tools have provided us with the means to catch a glimpse of the unicorn of *write once, deploy anywhere*.

The greatest foe we face in our quest for cross-platform implementation is platform-specific code. This code must be implemented differently depending on the platform, whether iOS, Android, or Windows Phone. Cross-platform patterns are the same regardless of operating system. Cross-platform code is sometimes referred to as *shared code*, or *core code*, as it is shared between projects for different mobile operating systems.

Xamarin.Forms addresses the thorniest cross-platform challenge: the user interface. Developers using Xamarin have a fully cross-platform data solution, which is local data access using ADO.NET or SQLite.NET with SQLite and then web services. Even so, there will always be platform-specific code, as follows:

- In the UI
- Device-specific functionality, such as camera and location
- Graphics and animations
- Security, file, and device permissions

Once we've identified the platform-specific and cross-platform code, the question is then how to organize it into a cross-platform architecture.

We have quite a few options, ranging from PCLs, shared projects, and linked files, interfaces, abstraction, and conditional compilation. PCLs provide the means for a C# component to be built with a limited, platform-specific subset of the .NET library to be compiled into a Dynamic Link Library (DLL) that can be used in a Xamarin project for any platform specified by the PCL's profile. Data access layers, client-side web services, and platform-independent business logic live happily here. Platform-specific functionality can still be introduced into these libraries by using dependency injection with interfaces. A looser, more flexible approach is to use shared files or projects that contain core files recompiled for each platform. Conditional compilation, an ancient technique well-suited to small platform-specific customizations, permits blocks of code within a shared file to be included in a platform-specific compile.

We will delve into these techniques and their related patterns as they bear the mark of the unicorn, helping us to maximize our code's cross-platform footprint.

Summary

Business applications are undergoing a sea change in hardware transformation; we have not seen this magnitude of change since the commercialization of the personal computer. The momentum of consumer mobile devices has reached a tipping point, affecting the devices upon which business applications must now function. With the battle continuing to rage between mobile operating systems, it is no longer enough to just get a mobile app up and working on a single platform.

We must think cross-platform from the get-go.

Within the world of .NET, Xamarin has provided us with the tools to make cross-platform development the norm instead of a special case, so we have no excuse. If the proper approach is taken, business logic, data access layer, and, increasingly, even the UI are mostly platform-independent. So whether you are building an Android, Windows Phone, or iOS app, the approach can be largely the same for many components of the app.

Let's take a look at the code!

CHAPTER 2

Building Mobile User Interfaces

In mobile UI development using Xamarin, our screens and their controls, images, animations, and user interactions run natively on a handheld device. Various synonyms for mobile UI *screens* exist, such as *views* and *pages*, and these are used interchangeably here. A *view* can mean a *screen* but can also refer to a *control* in certain contexts.

Two standard approaches apply to building mobile UIs with Xamarin:

- *Xamarin.Forms* is a cross-platform UI library for Android, iOS, and Windows Phone.
- *A platform-specific (or native) UI* approach uses Xamarin.Android, Xamarin.iOS, and the Windows Phone SDK.

This chapter covers both approaches and defines the platform-specific components that make up each of them. We will talk about when Xamarin.Forms is useful and when a more platform-specific approach might be better. Then we'll delve into building a Xamarin.Forms UI using pages, layouts, and views. We will create a Xamarin.Forms solution containing shared projects and platform-specific ones. While adding Xamarin.Forms controls to a project, we will touch upon basic UI concepts such as image handling and formatting controls in a layout.

Let's start by discussing Xamarin.Forms.

Understanding Xamarin.Forms

Xamarin.Forms is a toolkit of cross-platform UI classes built atop the more foundational platform-specific UI classes: Xamarin.Android and Xamarin.iOS. Xamarin.Android and Xamarin.iOS provide mapped classes to their respective native UI SDKs: iOS UIKit and Android SDK. Xamarin.Forms also binds directly to the native Windows Phone SDK. This provides a cross-platform set of UI components that render in each of the three native operative systems (see Figure 2-1).

Figure 2-1. *Xamarin libraries bind to native OS libraries*

Xamarin.Forms provides a cross-platform toolkit of pages, layouts, and controls and is a great place to start to begin building an app quickly. These Xamarin.Forms elements are built with Extensible Application Markup Language (XAML) or coded in C#, using `Page`, `Layout`, and `View` classes. This API provides a broad range of built-in cross-platform mobile UI patterns. Beginning with the highest-level `Page` objects, it provides familiar menu pages such as `NavigationPage` for hierarchical drilldown menus, and `TabbedPage` for tab menus, a `MasterDetailPage` for making navigation drawers, a `CarouselPage` for scrolling image pages, and a `ContentPage`, a base class for creating custom pages. Layouts span the standard formats we use on various platforms including `StackLayout`, `AbsoluteLayout`, `RelativeLayout`, `Grid`, `ScrollView`, and `ContentView`, the base layout class. Used within those layouts are dozens of familiar controls, or views, such as `ListView`, `Button`, `DatePicker`, and `TableView`. Many of these views have built-in data binding options.

Xamarin.Forms comprises platform-independent classes that are bound to their native platform-specific counterparts. This means we can develop basic, native UIs for all three platforms with almost no knowledge of iOS and Android UIs. Rejoice but beware! Purists warn that trying to build apps for these platforms without an understanding of the native APIs is a reckless undertaking. Let's heed the spirit of their concerns. We must take a keen interest in Android and iOS platforms, their evolution, features, idiosyncrasies, and releases. We can also wallow in the convenience and genius of the amazing cross-platform abstraction that is Xamarin.Forms!

■ **Note** Basic pages such as login screens, simple lists, and some business apps are well-suited to out-of-the-box Xamarin.Forms at the time of this writing. Platform-specific code can be utilized in Xamarin.Forms projects for added functionality but each subsequent release of this library will allow us to build more complex screens without utilizing platform-specific code.

Xamarin.Forms Solution Architecture

One of the greatest benefits of Xamarin.Forms is that it gives us the ability to develop native mobile apps for several platforms simultaneously. Figure 2-2 shows the solution architecture for a cross-platform Xamarin.Forms app developed for iOS, Android, and Windows Phone. In the spirit of good architecture and reusability, a Xamarin.Forms cross-platform solution often uses shared C# application code containing the business logic and data access layer, shown as the bottom level of the diagram. This is frequently referred to as the Core Library. The cross-platform Xamarin.Forms UI layer is also C# and is depicted as the middle layer in the figure. The thin, broken layer at the top is a tiny amount of platform-specific C# UI code in platform-specific projects required to initialize and run the app in each native OS.

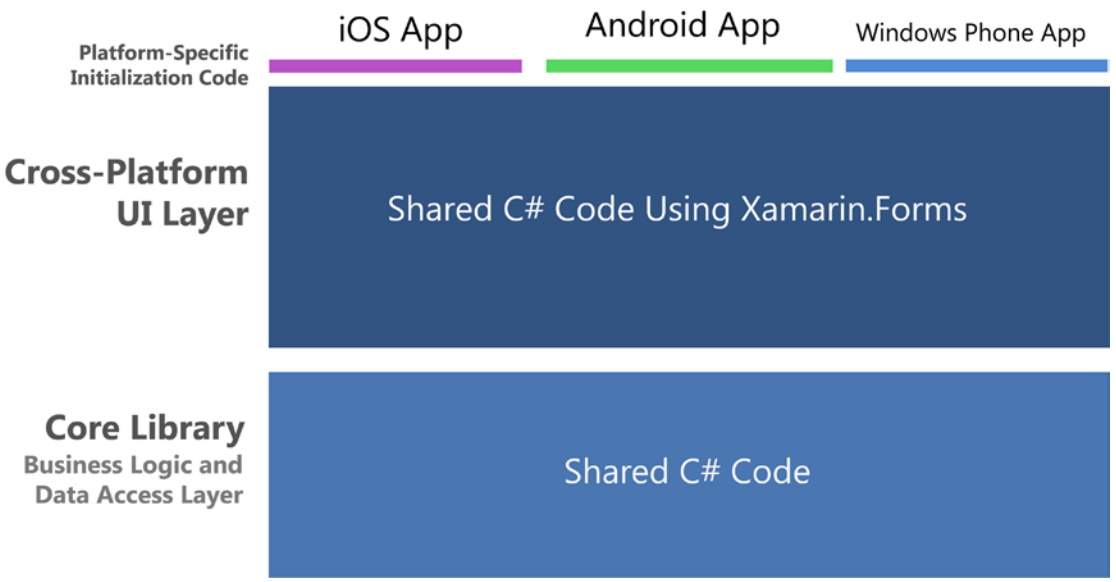

Figure 2-2. *Xamarin.Forms solution architecture: One app for three platforms*

Figure 2-2 is simplified to communicate the fundamentals of Xamarin.Forms. The reality is that hybridization between Xamarin.Forms and platform-specific code is possible, useful, and encouraged. It can happen at a number of levels. First, within a Xamarin.Forms Custom Renderer, which is a platform-specific class for rendering platform-specific features on a Xamarin.Forms page. Hybridization can also happen within platform-specific Android activities and iOS view controllers that run alongside Xamarin.Forms pages, or within platform-specific classes that are called as-needed to handle native functionality such as location, camera, graphics, or animation. This sophisticated approach can lead to a more complex architecture, such as Figure 2-3, and must be handled carefully. Note the addition of the Platform-specific UI Layer.

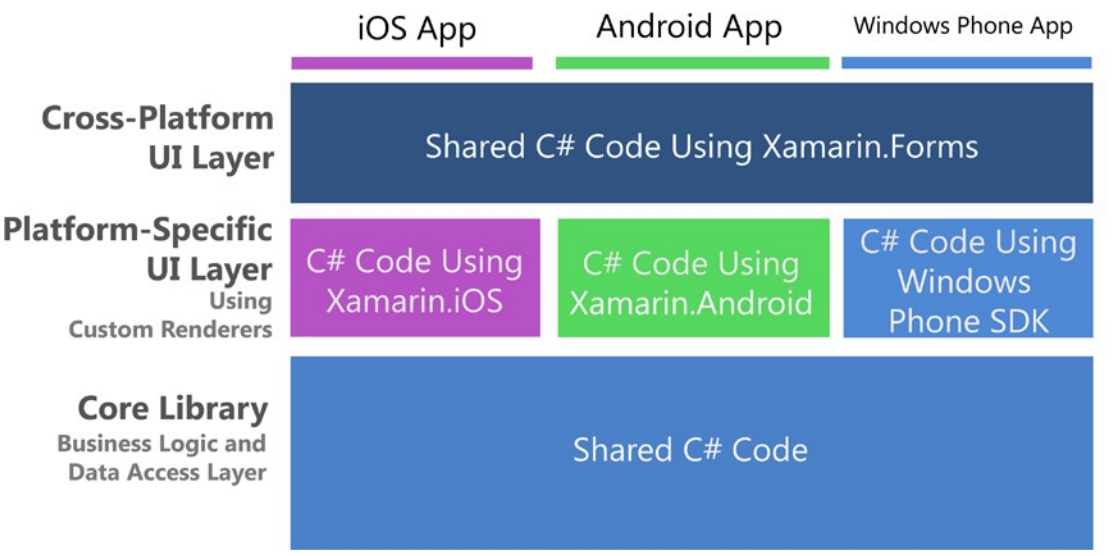

Figure 2-3. *Xamarin.Forms architecture with custom renderers*

■ **Note** Chapter 8 provides more on the use of custom renderers and platform-specific code in Xamarin.Forms solutions.

When are Xamarin.Forms appropriate to use and when do we consider other Xamarin options? I'll address this key question a bit later in the chapter, but first let's define Xamarin's platform-specific UI options.

Understanding the Platform-Specific UI Approach

Before Xamarin.Forms, there were the platform-specific (or native) UI options, which consist of the Xamarin.Android, Xamarin.iOS, and Windows Phone SDK libraries. Building screens using platform-specific UIs requires some understanding of the native UIs exposed by these libraries. We don't need to code directly in iOS UIKit or Android SDK, as we're one layer removed when using Xamarin bindings in C#. Using the Windows Phone SDK is, of course, coding natively in C# against the Windows Phone SDK, a C# library. The advantage of using Xamarin's platform-specific UIs is that these libraries are established and full-featured. Each native control and container class has a great many properties and methods, and the Xamarin bindings expose many of them out-of-the-box.

■ **Note** We're not talking about native UI development using Objective-C or Java here but use of Xamarin C# platform-specific bindings to native UI libraries. To avoid such confusion, this book favors the term *platform-specific* over *native* when referring to Xamarin libraries but Xamarin developers will sometimes use the term *native* to refer to the use of platform-specific libraries `Xamarin.iOS` and `Xamarin.Android`.

Platform-specific UI Solution Architecture

Figure 2-4 shows how a platform-specific solution designed to be cross-platform shares C# application code containing the business logic and data access layer, just like a Xamarin.Forms solution. The UI layer is another story: It's all platform-specific. UI C# code in these projects uses classes that are bound directly to the native API: iOS, Android, or the Windows Phone API directly sans binding.

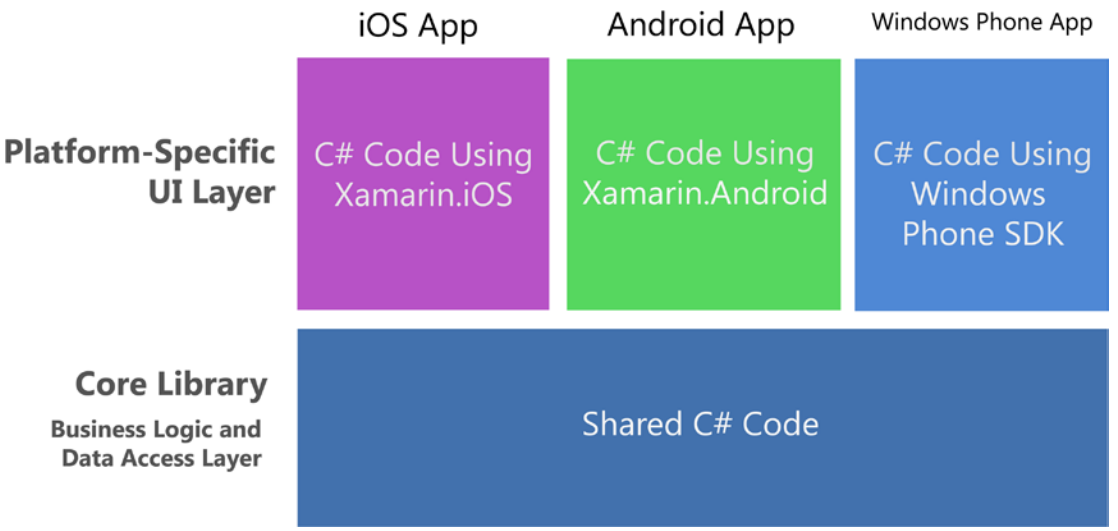

Figure 2-4. *Platform-specific UI solution architecture*

If you compare this diagram to the Xamarin.Forms diagram in Figure 2-2, you'll see that there's a lot more coding to be done here: a UI for every platform as opposed to one for all. Why would anyone bother to do it this way? There are quite a few good reasons why some or even all of the code might be done better this way. To address the burning question *Which do I use, Xamarin.Forms or Xamarin platform-specific UIs?,* see the section "Choosing Xamarin.Forms or a Platform-Specific UI" later in this chapter.

But first let's delve into the Android and iOS bindings and then look at the Windows Phone SDK.

Xamarin.Android

The Xamarin.Android C# bindings tie us into the Android API. Android apps are made up of layouts and activities, roughly translated as views and controllers. Layouts are typically XML files (.axml) edited using a UI designer that define the controls displayed on a screen. An `Activity` is a class that typically manages the life cycle of a single layout, although smaller layouts, called fragments, can be combined to comprise a screen.

Controls are called *views* in Android-ese: `Buttons`, `TextViews`, `ListViews`, and so forth. We place these on another kind of layout, controls that contain controls, which work like `<div>` in HTML: `LinearLayout`, `RelativeLayout`, `FrameLayout`, and `WebViews`. These layouts, inherited from the `ViewGroup` class are assembled manually or generated dynamically using data-binding classes called *adapters*. Inheriting from `AdapterView`, widgets such as `ListView` and `GridView` are populated by data-binding.

■ **Note** Android parlance uses "layout" to mean two different things: an XML file containing a UI screen (.axml) and a container control that houses and formats other controls, such as `LinearLayout`.

There are two ways to build Android layouts: using XML resource files or coding them in C# by hand. XML files (`.axml`) are highly readable and elegant, which encourages direct editing of the XML code, even when also using the Xamarin Designer for Android, the XML resource file editor. Most mobile developers prefer Android XML resource files (`.axml`) to hand-coding the UI in C#. Coding Android UI by hand in C# is not as comfortable, because the necessary methods and parameters have been deemed by the developer community to be clumsy and difficult to use. Additionally, most online examples of Android UI use XML resource files, even in the Xamarin online documentation. Those are the reasons that most of the Android development community is using XML for UI development. This practice has extended to the Xamarin development community, not to mention that using XML resource files is the Xamarin-recommended method.

Xamarin.iOS

The Xamarin.iOS C# bindings hook us up with the iOS native UI API, called UIKit. `Views` and `ViewControllers` are the equivalent of views and controllers in iOS. Views are typically constructed using a designer tool and result in an XML file (`.xib` or `.storyboard`). `ViewController` is a controller class that manages the views. In iOS we work with layers: tab bar view, navigation view, and images overlaying our main view, all nested inside a `UIWindow`. Controls include `UILabel`, `UIButton`, `UITextField`, and `UISlider`. These controls reside in a view class called `UIView`, which is inherited to create useful data-bound views such as `UITableView` for lists and `UICollectionView` for grids and groupings. iOS layouts are built using a technique called *AutoLayout*, based on constraints between views that move and size dynamically depending on the display context. The older layout approach, *AutoSizing*, involves creating frames and masking them, also called *springs and struts*. This is all part of UIKit, the development framework of the iOS user interface.

■ **Tip** Why the `UI` in `UILabel`, `UIButton`, `UIThis`, and `UIThat`? iOS's Objective-C has no namespaces, so they are concatenated with the class name.

There are two ways to build screens in iOS: the first is using a designer tool, such as the Xamarin Designer for iOS or the Xcode Interface Builder, and the second is hand-coding in C#. The designer tools create a storyboard XML file or `.xib` (pronounced and written: *nib*), and hand-coding is done straight in the iOS view-controller C# classes (these are then called *nib-less views*). Storyboards and nibs are sometimes difficult to read. This tightly couples them to the tools we use to construct them and discourages manual editing. Nibs are useful for simple forms such as modals and login pages, and the storyboard is the workhorse for prototyping, complex transitions, and multiple interconnected pages. Dynamic data-binding, data flow between pages, and visual effects and complexity are often best accomplished with hand-coded C#.

■ **Important Tip** The focus of this book is Xamarin.Forms and cross-platform code-first development with some of the most useful platform-specific techniques available. That's a lot to cover in one book, so the discussion of some Android and iOS basics and all designer tools is out of scope. *If you're doing platform-specific development on Android or iOS for the first time, you'll need to consult additional sources.* See the Introduction of this book for Prerequisites and consult the Xamarin, Google, and Apple online docs or one of the many fine platform-specific books to fill you in.

Windows Phone SDK

The Windows Phone SDK is a C# library with a built-in .NET API. Screens are defined by `Frame` classes that handle navigation and contain pages, loaded and unloaded like conventional views. Within these are layout containers called *panels*, such as `Canvas` for absolute positioning, and `StackPanel` and `Grid` for relative layout with autosizing. There are familiar controls such as `TextBox` for editing and `TextBlock` for labels, and `Button`, `Image`, and `MediaElement` for videos and music. For lists there is `LongListSelector` and the older `ListBox`. We build the UI using C#, XAML, Blend, and/or the Visual Studio UI designer.

■ **Tip** Because the Windows Phone SDK is already using C# and .NET, the Xamarin platform is not necessary to code a Windows Phone app in C#. Cross-platform development is the primary consideration that brings Xamarin together with Windows Phone: A Xamarin.Forms app can run on a Windows Phone.

Xamarin.Forms currently supports Windows Phone Silverlight, WinRT and Windows Store support have been announced.

Choosing Xamarin.Forms or a Platform-Specific UI

As developers, we are faced with this decision:

Which do I use, Xamarin.Forms or a Xamarin platform-specific UI?

The trade-off is portability of Xamarin.Forms versus the full-featured functionality of Xamarin's platform-specific UIs, namely Xamarin.Android and Xamarin.iOS. At the time of this writing, the platform-specific Xamarin APIs have considerably more features than Xamarin.Forms. The answer to our question will range from one, to the other, to both, depending on your needs. Here are suggested guidelines:

Use Xamarin.Forms for the following:

Learning Xamarin: If you're new to mobile development using C# then Xamarin.Forms is a great way to get started!

Cross-platform scaffolding: When building cross-platform apps, Xamarin.Forms is useful to build out the scaffolding of your app, as a rapid-application development toolset.

Basic Business Apps: Xamarin.Forms does these things well: basic data display, navigation, and data entry. This is a good fit for many business apps.

Basic design: Xamarin.Forms provides controls with baseline design features, facilitating basic visual formatting.

Simple cross-platform screens: Xamarin.Forms is great for creating fully functional basic screens. For more complex screens, leverage Xamarin.Forms custom renderers for platform-specific details.

Use a platform-specific UI (Xamarin.iOS or Xamarin.Android) for:

Complex screens: When an entire screen (or an entire app) requires a nuanced and complex design and UI approach, and Xamarin.Forms isn't quite up to the task, go with a platform-specific UI using Xamarin.Android and Xamarin.iOS.

Consumer Apps: Platform-specific UI has everything a developer needs to create a consumer app with complex visual design, nuanced gesture sensitivity, and high-end graphics and animation.

High-design: This approach provides complete native UI APIs with low-level access to design properties on each control, allowing for a high visual standard of design. Native animation and graphics are also available with this approach.

Single-platform apps: If you're building for only one platform, and a cross-platform approach for your app is not important in the foreseeable future (a rare case even if you're starting with one platform), consider using a platform-specific UI.

Unsupported platforms: Mac OS X, Windows Store, and WinRT apps are not currently supported by Xamarin.Forms at this time.

However, Xamarin moves fast, and these recommendations are likely to change. Here's how: With each new release of Xamarin.Forms, more properties and methods will be included in the bindings, bringing this library closer to the platform-specific ones and giving us increased control over our cross-platform UI. Also, third-party vendors and open source projects such as Xamarin Forms Labs are swiftly extending the options available with added controls, charts, and datagrids. Currently, there is no visual designer for Xamarin.Forms, but I expect there will be one soon.

When complex tasks or high design are required by Xamarin.Forms, virtually anything is possible using Custom Renderers.

Use Both Approaches with Custom Renderers

Custom Renderers provide access to the lower-level, platform-specific, screen-rendering classes called Renderers, which use platform-specific controls to create all Xamarin.Forms screens. Any Xamarin.Forms screen can be broken into platform-specific screens and classes using this approach. This means we can write a Xamarin.Forms page or app, and customize it by platform whenever necessary. More about this in Chapter 8.

Use Custom Renderers sparingly, or risk a fragmented UI code base that probably should have been written entirely as a platform-specific UI.

In each of the following chapters, we will explore the Xamarin.Forms options and then examine platform-specific implementations of the same functionality. You will be able to see how they compare at the time of this writing and how to use them together using custom renderers. As time marches on, Xamarin.Forms may progress from a scaffolding technology to fully featured building blocks for cross-platform apps. If it does

not, or until it does, the platform-specific approach will remain necessary to build highly functional apps without heavy reliance on Xamarin.Forms custom renderers.

Yesterday's award-winning Xamarin apps were created using the platform-specific approach, but the key question is, What will *you* create today?

Let's explore the building blocks of the C# mobile user interface.

Exploring the Elements of Mobile UIs

Xamarin is a unifying tool serving several platforms, many of which can have different names for the same things. Here are some unifying terms, weighted heavily in the direction of Xamarin.Forms:

> *Screens, views, or pages in mobile apps are made up of several basic groups of components*: pages, layouts, and controls. Pages can be full or partial screens or groups of controls. In Xamarin.Forms, these are called pages because they derive from the `Page` class. In iOS, they are views; and in Android, they're screens, layouts, or sometimes loosely referred to as activities.
>
> *Controls are the individual UI elements we use to display information or provide selection or navigation. Xamarin.Forms calls these views, because a* `View` *is the class that controls inherit from. Certain controls are called widgets in Android. More on these shortly and in Chapter 4.*
>
> *Layouts are containers for controls that determine their size, placement, and relationship to one another. Xamarin.Forms and Android use this term, while in iOS everything is a view. More on this in Chapter 3.*
>
> *Lists, typically scrollable and selectable, are one of the most important data display and selection tools in the mobile UI. More on these in Chapter 5.*
>
> *Navigation provides the user with a way to traverse the app by using menus, tabs, toolbars, lists, tappable icons, and the up and back buttons. More on this in Chapter 6.*
>
> *Modals, dialog boxes, and alerts are usually popup screens that provide information and require some response from the user. More on these in Chapter 6.*

Now that we have context and some terminology to work with, let's get started with Xamarin.Forms!

Using the Xamarin.Forms UI

Pages, layouts, and views make up the core of the Xamarin.Forms UI (Figure 2-5). Pages are the primary container, and each screen is populated by a single `Page` class. A page may contain variations of the `Layout` class, which may then hold other layouts, used for placing and sizing their contents. The purpose of pages and layouts is to contain and present views, which are controls inherited from class `View`.

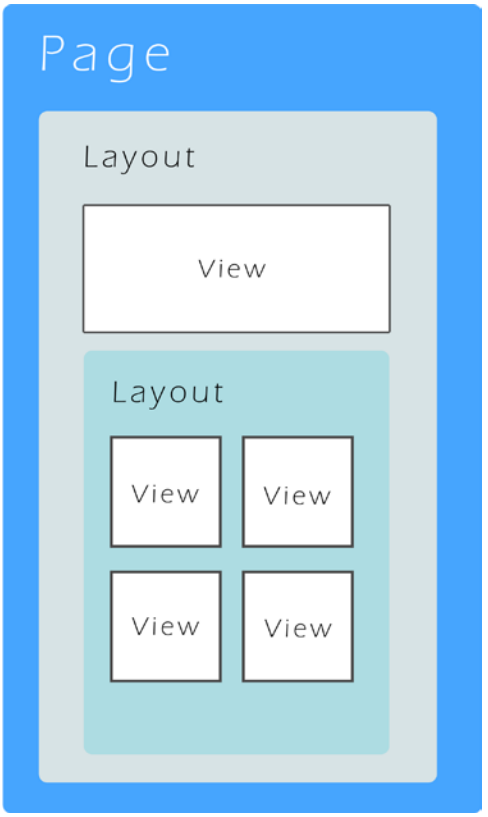

Figure 2-5. *Page, layouts, and views on a Xamarin.Forms screen*

Page

The `Page` class is the primary container of each main screen in the app. Derived from `Xamarin.Forms.VisualElement`, `Page` is a base class for the creation of other top-level UI classes. Here are the primary pages:

- `ContentPage`
- `MasterDetailPage`
- `NavigationPage`
- `TabbedPage`
- `CarouselPage`

In addition to serving as containers for layouts and views, pages provide a rich menu of prefabricated screens with useful functionality that includes navigation and gesture responsiveness. More on these in Chapters 6.

Layout

Views are placed and sized by their container class, `Layout`. Layouts come in a variety of flavors with different features for formatting their views. These containers allow views to be formatted precisely, loosely, absolute to the coordinate system, or relative to one another. Layouts are the soft tissue of the page, the cartilage that holds together the solid, visible aspects of the page(views). Here are the main layouts:

- `StackLayout`
- `AbsoluteLayout`
- `RelativeLayout`
- `Grid`
- `ScrollView`
- `Frame`
- `ContentView`

The layout's `Content` and/or `Children` properties contain other layouts and views. Horizontal and vertical alignment is set by the properties `HorizontalOptions` and `VerticalOptions`. Rows, columns, and cells within a layout can be padded with space, sized to expand to fill available space, or shrunk to fit their content. More on layouts in the next chapter.

■ **Tip** Xamarin.Forms layouts are derived from the `View` class, so everything contained by a page is actually some form of a view.

View

Views are controls, the visible and interactive elements on a page. These range from the basic views like buttons, labels, and text boxes to the more advanced views like lists and navigation. Views contain properties that determine their content, font, color, and alignment. Horizontal and vertical alignment is set by properties `HorizontalOptions` and `VerticalOptions`. Like layouts, views can be padded with space, sized to expand to fill available space, or shrunk to fit their content. Later in this chapter, we'll code some views, then visit them again in Chapter 4 and throughout the book. These are the primary views grouped by function:

- Basic – fundamental views
  - `Label`
  - `Image`
  - `Button`
  - `BoxView`
- List – make a scrollable, selectable list
  - `ListView`
- Text Entry – user entry of text strings using a keyboard
  - `Entry`
  - `Editor`