brief contents

PART 1  GETTING STARTED WITH XAMARIN ........................................ 1

1  ■  Introducing native cross-platform applications with Xamarin  3
2  ■  Hello MVVM—creating a simple cross-platform app using MVVM  25
3  ■  MVVM—the model-view–view model design pattern  50
4  ■  Hello again, MVVM—understanding and enhancing our simple MVVM app  81
5  ■  What are we (a)waiting for? An introduction to multithreading for Xamarin apps  113

PART 2  BUILDING APPS ............................................................. 153

6  ■  Designing MVVM cross-platform apps  155
7  ■  Building cross-platform models  195
8  ■  Building cross-platform view models  241
9  ■  Building simple Android views  288
10  ■  Building more advanced Android views  319
11  ■  Building simple iOS views  345
12  ■  Building more advanced iOS views  385
PART 3  FROM WORKING CODE TO THE STORE........................................409

13  ■  Running mobile apps on physical devices   411
14  ■  Testing mobile apps using Xamarin UITest   436
15  ■  Using App Center to build, test, and monitor apps  467
16  ■  Deploying apps to beta testers and the stores   505
The traditional way to build a mobile app is to write it twice: once in Objective-C or Swift for iOS, and then again in Java for Android. This is a huge waste of time, duplicating code across two languages. Luckily some of the most innovative engineers in the world (according to *Time* magazine) have a solution—Xamarin.

Xamarin is a platform from Microsoft that allows you to build and ship iOS and Android apps using .NET. It’s also part of a thriving mobile ecosystem containing everything from mobile-specific cloud resources from Microsoft, DevOps tools, and a huge community of open source software. At its most basic, it’s a way to use the same language and technology across iOS and Android, allowing you to reuse large amounts of code and third-party libraries across two very different mobile platforms. The best practices around Xamarin are focused on keeping this amount of code-sharing as large as possible.

This first part of the book covers the architectural concepts behind a well-written cross-platform Xamarin app, focusing on the incredibly popular MVVM design pattern. A good architecture will help you reuse the most code possible, so it’s worth investing the time to learn these concepts, avoiding wasting time writing swathes of code twice. Patterns such as MVVM allow you to test your code faster and easier using unit tests, catching bugs earlier in the development cycle and reducing the time manually testing (and bug fixing) further down the development cycle. These are the foundations you’ll need to build production-quality mobile apps.
Chapter 1 starts by discussing Xamarin and the benefits of building Xamarin mobile apps. It also looks at the development lifecycle, covering all the steps in building production-quality mobile apps.

Chapter 2 looks at the MVVM design pattern as a way to increase your code reuse, and to build a well-architected, testable app. Then it covers the creation of a Hello World app that uses a popular MVVM framework.

Chapter 3 dives into MVVM in more detail, looking at the different layers from model, through view model, to view. It then covers the application layer and navigation patterns.

Chapter 4 revisits the example Hello World app from chapter 2, diving deeper into how the MVVM design pattern was used to build the app. It then looks at expanding the app, using cross-platform Xamarin plugins.

Chapter 5 is all about multithreading, covering the threading considerations involved in building mobile apps. It also introduces async and await, a feature of C# that makes it very easy to build clean and easy-to-read multithreaded code.
Introducing native cross-platform applications with Xamarin

This chapter covers
- What a Xamarin app is
- The mobile-optimized development lifecycle
- Building production-ready cross-platform apps

Back in 2000 Microsoft announced a new software framework called .NET, along with a new programming language called C#. Not long after this, a company called Ximian (founded by Miguel de Icaza and Nat Friedman) started working on Mono, a free implementation of the .NET framework and the C# compiler that could run on Linux.

Fast forward 16 years, and Nat Friedman is standing on stage at the Xamarin Evolve conference giving the keynote talk—physically in front of sixteen hundred mobile developers and virtually in front of tens of thousands more. He’s speaking about how Xamarin enables a mobile-optimized development lifecycle. Xamarin (the company that grew out of the ashes of Ximian and that provides tools and technology to build cross-platform mobile apps) had just been bought by Microsoft.
for a rumored half a billion U.S. dollars, and had become a key part of Microsoft’s “mobile first, cloud first” strategy.

Xamarin is now a well-known term among the mobile developer community, and it’s starting to be well known in other Microsoft-based developer circles. But what do we mean when we talk about Xamarin mobile apps, and what does Xamarin give us above and beyond other tools?

### 1.1 Introducing Xamarin mobile apps

To really see the benefits of Xamarin mobile apps, we first need to look at how apps are built using vendor-provided development environments, or other cross-platform tools like Cordova, and compare them to what Xamarin offers. We can do this by looking at two main types of developers—an indie developer working on an app in their spare time, and a corporate development team building an app for their customers. We’ll start by considering what their differing needs are in terms of platform support, and then we’ll compare the possible options.

Our example indie developer has come up with the idea of the millennium for a killer app, FlappyFoo, that they want to sell to consumers on an app store. Our example large corporation, FooCorp, wants to build a DailyFoo app to help their customers.

Figure 1.1 outlines the four different mobile development platforms you could choose from:

- Vendor-specific apps using the development environments from Apple and Google
- Cordova
- Xamarin native using Xamarin.iOS and Xamarin.Android
- Xamarin.Forms

![Figure 1.1 A comparison of the different mobile-development platforms](image-url)
Introducing Xamarin mobile apps

This diagram shows the programming languages used and where code can be shared for each layer of the app—from the application layer (the thin wrapper around the rest of the app that makes it into something that can be run on each platform), down through the UI layer to the business logic layer. The boxes are not to scale—they're just a representation of the layers. Your app could be heavy on UI but light on logic, so the UI layer would be bigger, or vice versa. Let’s look at each of these in more detail.

1.1.1 Vendor-specific native apps

Each OS comes with a different set of APIs, a different paradigm for building the user interface, a different way of handling user interactions, and, most frustratingly, a different programming language (or choice of languages) for you to use. If you want to build an app for Apple’s iOS-based devices such as iPhones and iPads, you need to program in either Objective-C or Swift. For Android phones and tablets, you need to program in Java (with Kotlin support coming soon).

For each platform you’ll end up building the entire app from the user interface layer right down to any in-app business logic all in the vendor’s preferred language, as shown in figure 1.2.

![Figure 1.2 Vendor-specific apps use the same language for all layers but different languages on each platform](image)

For our indie developer, this is a big problem. For FlappyFoo to be a success, it will need maximum reach, and this means both iOS and Android versions. To achieve this, the indie developer will have to learn two programming languages, and there’s more to learn than just the language syntax—they’ll have to learn different tools, different ways of getting access to third-party code, the different words developers use to express each concept, and the different design patterns that make up standard apps. This is a big task.

Even if the indie developer is a polyglot and is happy in multiple environments, there’s still the issue of time. They’ll have to code the same app twice, implementing the same logic in different languages. Time to market is key, and if the developer hits only one platform to start with, there’s nothing to stop copycats from flooding the other platform quickly. FlappyFoo may dominate the iOS app store but could lose serious revenue to FlappyBar from another developer on Android.
CHAPTER 1  Introducing native cross-platform applications with Xamarin

For our corporate team, the biggest issue is cost. To reach multiple platforms usually means one team per platform with the associated developer and organizational costs. This can be especially problematic if you consider the difficulties in finding, hiring, and retaining good developers. Ideally you want to be able to release simultaneously on all platforms, and to replicate each new feature to both platforms and release them simultaneously. This is hard if you’ve managed to employ five Android developers but only two iOS developers (a common scenario as it’s much easier to find Java developers in the corporate environment to help with Android versions than it is to find Objective-C or Swift developers).

Thinking of the corporation’s customers who use DailyFoo every day to track their Foo, the last thing we want is for them to change platform, find out that the new platform’s version is missing a killer feature from DailyFoo, and jump ship to MyBar from BarCorp.

It’s not all bad, though. The one thing you can always be sure of when writing an app using the vendor-provided tools is that you’re always building a truly native application that will be as high performance as possible and that supports everything the OS and devices have to offer. Whenever an OS update is released, the tooling is always updated to match, giving you access to all the newest features that your users will want to have. This is an important consideration, as app users are fickle. They’ll quickly drop an app for a competitor if it’s not up to scratch, it’s slow, clunky, or just not well integrated into their device.

1.1.2 Cordova

As already mentioned, using multiple languages and development tools is a headache. One popular way around this is using Cordova. This is a set of tools that allows you to create web applications using HTML, JavaScript, and CSS to build a mobile website, which is then wrapped in an app and packaged up for each platform, as shown in figure 1.3.

This has the big upside of a common language and development environment—one toolset for the indie developer to learn, or one team in a corporate environment. The downsides, though, can seriously outweigh this upside. First, you aren’t creating a native app—you’re creating a web app. This means that the widgets you see in the user interface are HTML widgets styled to look like native components. This might fool your users now, but if an OS update changes the style, your apps won’t keep up without a rebuild and will look out of date. Second, the OS and device-specific features that are available to the native developer won’t be available to a Cordova developer.
Introducing Xamarin mobile apps

The tooling does its best to provide some lowest-common-denominator plugins to allow hardware and OS access, but these are written with the aim of being cross-platform, so they only support the features common to both platforms. They’re also later to market. If the vendor releases a new feature you want to take advantage of, you’ll have to wait for the Cordova plugin to be created to support it, and this may never happen.

Thinking of our indie developer, if they use Cordova to build FlappyFoo, it could easily run slowly, especially on older devices. This can lead to a swath of one-star reviews, a lack of sales, and the developer going hungry. Cordova apps also run in a browser, so they’re limited by the speed and feature set of that browser—newer versions of the OS might have a fully featured, fast browser but older versions might be lacking. This can lead to different capabilities or different levels of performance on the same device but with different OS versions—something that’s very hard to test on the hugely fragmented ecosystem of Android.

For our corporate development team building DailyFoo, an app that’s slow or that looks out of date once an OS update comes out can create a negative image of the FooCorp brand. If the MyBar app from the rival BarCorp supports 3D touch on iOS, and DailyFoo doesn’t due to a lack of support from Cordova plugins, our fickle customers might easily be tempted to switch.

1.1.3 Xamarin native apps

In my mind, Xamarin is the clear winner because it combines the best of both the previous methods. Fundamentally, Xamarin provides a set of .NET wrappers around the native OS APIs based on Mono—the cross-platform implementation of .NET that grew out of Ximian. This provides a .NET framework for Android and iOS, with libraries and a C# compiler for each platform. It means you can write apps in C# that target each mobile platform natively, and because you’re using a single programming language, you can easily abstract out all your business logic (anything that doesn’t interact with the device directly) into a set of libraries that can be shared between platforms. You can even abstract out a lot of the UI logic by using design patterns like MVVM (model-view–view model, which you’ll learn about in more detail in chapter 2). Figure 1.13 shows the code split and sharing between each layer.

Let’s take a closer look at those last points, as this is important and is the key reason in my mind for using Xamarin:

- Xamarin provides wrappers around native APIs.
- Xamarin provides a compiler for each platform to produce native code.
This is key. The native APIs are wrapped in C# code so you can call them from your C# code. You write your apps using the same idioms and classes as pure native code, but using C#. On iOS you have a `UIViewController` class for each screen, but this is a C# class, not the Objective-C one from the iOS SDK that you code against. On Android, each screen is derived from a class called `Activity`, but it’s a C# class that wraps the Java `Activity` class from the Android SDK.

The code you write is compiled code as well—this isn’t sitting inside some emulator on the device; it’s compiled to native code that interacts with the same libraries as an app written in the vendor’s language of choice and compiled with their tools. This means your app is truly native. It uses native widgets on the UI, has access to every device and OS feature the native API has access to, and is as fast as a native app.

**XAMARIN APPS == NATIVE APPS** This is the killer feature of Xamarin apps. They’re written in C# and they have access to all the features of that language, to a large part of the .NET framework that desktop developers are used to, and to a whole host of third-party code. But the end result is native code—the same as that created in Objective-C or Swift on iOS, or Java on Android.

On iOS the C# compiler takes your code and produces a native iOS binary using an Ahead-Of-Time (AOT) compiler (figure 1.5).

On Android it creates IL code (the same as for C# apps running on Windows), which is compiled at runtime using just-in-time (JIT) compilation (figure 1.6). This is provided by a Mono runtime that’s built into your app and installed with it (but don’t worry, you only get the bits of the Mono runtime you need, thanks to a very good linker). Xamarin also has an AOT compiler for Android, but at the time of writing, it’s still very much experimental.

**WHAT ABOUT OTHER LANGUAGES?** You can also write your apps using F# if you prefer a more functional style of programming. F# is fully supported for iOS and Android apps. If VB.NET is your thing, you can build .NET Standard class libraries using it and call these from your iOS and Android apps built using C# or F#. Those options are outside the scope of this book, though—we’ll just focus on C#.
Because the language of choice is C#, the code libraries written to share code between iOS and Android can also be shared with a UWP (Universal Windows Platform) app, so you can easily target Windows 10 devices from desktops to tablets to phones to the Xbox One if you so desire.

For our indie developer, this is good news. They only have one language to learn, and they only have to write the bulk of their app once, and then write the device-specific layer once per platform they want to support. This gives a faster time to market, which is vital for consumer apps. It also means the core logic code is tested the same way on all platforms, bugs are fixed once, and improvements and new features are created with fewer changes.

For our corporate development team, this is also a good thing as it means fewer developers and less cost. Ideally there would be some developers who specialize in the platform-specific idioms of each supported OS who can work on the UI or device-specific logic, but the core of the development team can build the business logic once in a single language. It’s also easier to build the development teams because C# developers are easy to find—much easier than Objective-C developers. The advantages for the indie developer also apply here—less code to test and faster to market with bug fixes and new features.

This is not a total utopia. Xamarin developers still have to write the UI layer and anything that interacts with the device using platform-specific C# code and they still need to understand the idioms of each platform, but they only have one language to support. One syntax, one toolset, one way of using third-party code.

It’s easy to look at this and think of it as a partial failure—something that misses the mark by not being totally cross-platform—but that’s really one of its strengths. By having C# platform-specific APIs, you get the best of what the device has to offer. You aren’t limited to a common subset; instead you can write each platform’s app in a way that makes the most of the features of those devices. It also means you have access to everything—when iOS adds a new feature, Xamarin wraps its API and it’s available to you pretty much the same day. Your apps can be targeted to each platform, so they look and feel like a pure, native app and take advantage of the unique features that
make Android and iOS so different, but behind the scenes you’re sharing around 75% of your code base. Table 1.1 shows some examples of this code sharing.

Table 1.1  The amount of code in two popular apps reused between iOS and Android

<table>
<thead>
<tr>
<th></th>
<th>iCircuit (<a href="http://icircuitapp.com">http://icircuitapp.com</a>)</th>
<th>TouchDraw (<a href="http://elevenworks.com/home">http://elevenworks.com/home</a>)</th>
</tr>
</thead>
<tbody>
<tr>
<td>iOS</td>
<td>70%</td>
<td>61%</td>
</tr>
<tr>
<td>Android</td>
<td>86%</td>
<td>72%</td>
</tr>
</tbody>
</table>

There’s one downside to using Xamarin for your mobile apps—you’re dependent on them wrapping the SDKs and ensuring that the compilers work on all required platforms. There’s an overhead to wrapping the SDKs, and although Xamarin has got very, very good at it, there can still be a gap between an API being made available from Apple and Google and Xamarin having it wrapped. This is usually not an issue, as both Apple and Google release beta versions early enough for Xamarin to have time to deal with any quirks.

The only thing that has been a problem is when the underlying compiler requirements change. This happened recently with Apple Watch apps: originally they were compiled native code, but for watchOS 2 the Apple compiler changed to output bytecode instead of native code. It took a long time for Mono to catch up and be able to compile working watchOS 2 apps. This is the biggest risk with Xamarin—that Apple or Google could completely change how they build apps, and by the time Xamarin catches up, your app could have been late to market with a cool new feature or device support.

Now that Xamarin is owned by Microsoft, I can see this being less of an issue as they’ll have more resources to throw against such a problem.

1.1.4 Xamarin.Forms

Xamarin also offers a more cross-platform solution called Xamarin.Forms that attempts to bring code reuse up to 95–98% by abstracting out the UI and device-specific code layers. Unlike Cordova apps that use HTML, Xamarin.Forms apps are still native apps. It uses an abstraction that sits on top of the iOS and Android platforms and provides a lowest common denominator experience, providing features that are common to both platforms. By doing this, you can get up to 98% code reuse. This is shown in figure 1.7.

This abstraction is done using a set of UI classes that represent features common to both, and when
Introducing Xamarin mobile apps

the app is run, these are translated to the native equivalents behind the scenes. For example, each screen you see is a Page, and this is rendered on iOS using a UIViewController and on Android using an Activity. If you add a Button to this page, it’s a UIButton on iOS and a Button widget on Android. Unlike Cordova, which uses HTML to provide the cross-platform capability, Xamarin.Forms uses the actual, native controls, so you get a true native experience. If the OS updates the look of the buttons, your Forms apps will look like the new version. This abstraction is exposed not only as a set of C# classes you can use from your C# code, but you can also define your UI using XAML—a variant of XML originally defined by Microsoft for building UIs.

XAML allows you to define your UI using a more declarative syntax, similar in nature to HTML, and it’s very familiar to developers from a Windows desktop background who are used to building apps with WPF. If you’ve built WPF or Windows 10 apps, you’ll probably have come across XAML before. Xamarin.Forms uses a slightly different variant of XAML than WPF/Windows 10, but most of the concepts are the same. This similarity will increase over time because Microsoft is in the process of defining XAML Standard, a single XAML syntax that will be used across all the Microsoft XAML tooling.

The downside is that you’re building one app for all platforms. Although it tries to be as native as possible by using native controls, you can’t easily get around platform-specific idioms. For example, if you have an app that has two screens to work on, you’d navigate on Android using a drawer exposed by a hamburger menu, whereas on iOS you’d use tabs. This difference isn’t easy to implement in Forms without a lot of custom logic and custom UIs. If you want to go further than the lowest common denominator (for example, adding platform-specific behavior to one control on one platform) then you’d need to write a custom renderer for it—code that maps from the Forms controls to the underlying control.

Forms does try to abstract away device-specific features like maps or the camera using plugins, but again it’s a lowest common denominator model. The camera plugin won’t give you live photos on iOS, and the maps plugin doesn’t give you the same amount of control as Google Maps on Android.

For our indie developer, Forms might not be the best choice—the amount of work it would take to make an app look and feel like a true native experience might outweigh the time savings by maximizing code reuse.

For corporate developers, it might be a better option. Certainly for in-house apps, where you don’t always need a killer native experience, it’s a great tool, but for consumer apps it might not provide all the features needed. I’m sure over time it will carry on getting better and better—it’s under heavy development at the moment—but it’s not quite there yet for a great consumer app.

This book focuses on native Xamarin mobile apps, but the principles of MVVM that we’ll cover also apply to Xamarin.Forms apps.
1.1.5 Xamarin developer tools

As I’ve shown, Xamarin is far and away the best choice for mobile development—it gives you the power and performance of a native app, providing access to everything in the SDKs and on the devices, and it uses C# as a common language on all platforms so you can share the majority of your code base. So how do you go about building a Xamarin app?

For pure native apps, tooling is provided by the vendors: iOS apps are built using Xcode on the Mac, and Android apps are built using Android Studio on Mac/Windows/Linux.

For Xamarin apps, the best IDE around is Microsoft’s 20-year-old Visual Studio. It comes with a ridiculous number of features and tools, and it has a huge range of extensions to provide all manner of new features. It’s available as a community edition for indie developers and small teams for free, and it tiers up from there depending on how big your team is, what your support needs are, and whether you want enterprise features like profilers or embedding assemblies (you can compare the different tiers at www.visualstudio.com/vs/compare/). Xamarin is fully built into Visual Studio, providing a totally native experience where you can create a new app that targets iOS or Android just as easily as you can create a desktop WPF app or a class library. You can easily reference other projects, add in NuGet packages, and do everything with these project types that you can do with any native Windows project. From there you can build your Xamarin Android app and run it on an emulator (Visual Studio provides a number of built-in Android emulators) or on a real device. You can also build and run a Xamarin iOS app, albeit with some Apple-related restrictions.

Apple’s licensing rules for its SDK, compiler, and build tools require that you build on a Mac. Seeing as our Xamarin apps wrap the SDKs and compile down to native code using the Apple toolchain, you have to have a Mac. Luckily Xamarin iOS on Windows takes away the pain of this and provides support inside Visual Studio on Windows for building and debugging iOS apps using a remote Mac—all you need is a Mac with Xamarin installed that you can connect to, and the magic just happens. Visual Studio connects to the Mac to compile your code. The iOS SDK on the Mac includes an iPhone/iPad simulator, which you can use to test your app, and a debugger that allows you to debug apps running on a device connected via USB to your Mac, so initially you still had to test your apps either by running the simulator on the Mac or using a device plugged into it. But Xamarin now has that covered as well—at least for simulators. It can share the screen from the simulator to your Windows box so you can debug on a simulator as if it were all available on Windows. This means the Mac you need for building need not be next to you, or even on the same network. There are cloud services that can rent you time on Macs, such as Mac In Cloud (www.macincloud.com). You can use these for building your apps, and you can test these apps by debugging through Visual Studio on a simulator that’s screen-shared back to your Windows box. Figure 1.8 shows an overview of this process. You only need access to a physical Mac if you want to test on a real device.
Introducing Xamarin mobile apps

So far, so cool. We’re building cross-platform mobile apps on Windows. But one of the founding principles of the Mono project that inspired the Xamarin we know and love is being able to run on different platforms, and Xamarin has you covered there. Visual Studio is now available on the Mac, albeit in a cut-down version compared to Visual Studio on Windows. Xamarin used to have an IDE called Xamarin Studio, and this became the basis of Visual Studio for Mac. Visual Studio for Mac supports building iOS and Android apps, as well as macOS apps, tvOS apps, and ASP.NET Core websites. It has Azure integration allowing you to build both the mobile and web components of your app ecosystem, and even to debug both mobile and web components inside the same debugging session. Visual Studio on the Mac has the same licensing as for Windows, so it’s free for indie developers and small teams, with paid plans available for larger teams.

Which tool you use really depends on personal preference and the platforms you want to support. In this book we’ll be covering Visual Studio on both Windows and Mac.

CROSS-PLATFORM ALL THE THINGS! One other awesome thing to note is that Microsoft has changed recently from a closed company that was Windows only to one that supports open source and multiple platforms. They’ve even open sourced parts of the .NET framework and the compiler and have made it cross-platform. This means that bits of Mono are slowly being replaced with the Microsoft implementations from their .NET framework. It also means that the compiler in Visual Studio is the same on Windows as on Mac, with both using the open source Roslyn compiler. When you compile on the Mac, it’s the same compiler as on Windows.

1.1.6 Mobile-optimized development lifecycle

So far we’ve covered Xamarin apps, and, to a lot of people, this is what Xamarin is—a .NET framework and compiler for iOS and Android based on Mono. But as well as
providing the tools to build cross-platform apps, Xamarin also provides the tooling you need to do a lot more than just write the code.

One of the biggest concepts in the development world in recent years is DevOps—the cultural shift to a model where development and operations are combined. Some of the aims of DevOps include enabling individuals to be involved in all parts of the development and release cycle, automating as much as possible, and moving to a continuous delivery model where code can be checked in, built, and tested automatically and shipped to production with minimal human input. DevOps is a massive topic, well outside the scope of this book, but there are a number of tools, either provided by Xamarin or well integrated with other Xamarin tools, that can be used to help implement a good DevOps strategy.

During the Xamarin Evolve conference in April 2016, one of the main themes of the keynote was the mobile-optimized development lifecycle (as illustrated in figure 1.9). During this keynote, a number of tools, both from Xamarin and their new parent company Microsoft, were discussed. It was pretty clear that this was a key focus for Xamarin as a company, and it’s only been growing with the introduction of Visual Studio App Center and the greater push towards DevOps. This is important as we consider how to build production-quality mobile apps.

![Diagram of mobile-optimized development lifecycle](image)

**Figure 1.9** The mobile-optimized development lifecycle is a continuously iterating cycle of develop, test, build, distribute, and monitor.

### 1.2 Creating production-quality mobile apps

It’s a long journey from a back-of-the-napkin idea to a fully working, deployed app of sufficient quality to be usable and not get bad reviews. It’s easy for developers to jump straight into coding, as this is the part we love, but if you want to build an app that’s successful, you have to consider the whole software-development lifecycle. There’s no point in diving into the code and building something that doesn’t look good or work well because you haven’t considered the design of your finished app. During coding, you have to keep testing and monitoring in mind so that you code in a way that supports them. For anything more than a prototype, you have to think about the whole lifecycle before you write a single line of code. This lifecycle is very similar to the mobile-optimized development lifecycle talked about at Xamarin Evolve, but it adds a few more steps.
In this book we’ll be building a production-quality app, so let’s look at the stages a mobile app will need to go through on this journey. We’ll see what Xamarin can (or can’t) help with.

### Starting with an MVP

> If you are not embarrassed by the first version of your product, you’ve released too late.

—Reid Hoffman

It’s good practice when building a mobile app to start with an MVP—a minimum viable product. This is the smallest, simplest, fastest-to-market version you can deliver. Once this is in consumers’ hands, you can monitor how it’s used and deliver features based on what real people want. A lot of people think an app must be full-featured, based on their idea of what a full feature set is, to be successful, but your users might know better. It’s better to get an app out quickly and iterate based on real-world feedback, because it’s very easy to be wrong about what an app should have.

For example, Flickr started out as an online role-playing game with a photo-sharing tool, and only the photo-sharing part now survives. Be prepared to pivot!

### Design

Designing an app is hard, especially for developers with no formal design training. We’ve all seen some pretty shocking UIs, mainly for in-house apps where developers have thrown all the content and controls onto the screen and left it at that. In the consumer mobile world, this is no longer an option. Users can jump ship to another app that does the same things as yours in the time it takes to download a few megabytes of data from an app store. They have no loyalty to your app, and a bad app can remove loyalty to your business.

For example, if you’re a bank and people use your app to interact with their accounts every day and the experience is bad, they’d rather change their bank than keep using your bad app. You can get away with it in a corporate environment where your users are in-house and have to use whatever you put in front of them, but be prepared for complaints that may not be good for your career progression—especially if the CEO is one of the users.

There are several things to consider when designing an app:

- **Consistency**—Does your app look and work like other apps on the same platform, especially the ones provided by the OS vendor.

  For example, Android apps should follow the activity stack with the Back button doing what you’d expect. iOS apps should use tabs to switch between popular actions.
- **User experience**—Is your app easy to use and intuitive? A user should be able to just pick it up and know how to use it without any training. Being consistent with other apps can help with this.

  For example, avoid custom icons for buttons or menu items. Instead, use ones that are industry standard or just use text. No one cares that you think having your own custom icons will help promote your brand and make an app look like it’s yours. Instead, they’ll dump it if they can’t understand how to use it.

- **Flow**—Does your app flow well? Is there an easy flow for a user to use the app? When one action naturally leads to another, the journey between the actions should be short and concise.

  For example, if your app is for taking photos, the options for editing or sharing a photo should be on the same screen where you view the photo you’ve taken, not buried in a menu that involves multiple steps to navigate.

- **Good looks**—Does your app look good, are any images well drawn and appropriate for the device size, is all text clear and readable, and are the colors consistent and appropriate?

  For example, an app could be run on a small phone, large phone, “phablet,” small tablet, or large tablet. Any text on the screen must be readable in all formats, images must be sized to look good on all device sizes, and on-screen items should be spaced so that it’s clear what the user is looking at without UI elements blending into each other due to lack of space.

- **Accessibility**—Is your app accessible to users with differing abilities?

  For example, if a user increases the default font size, is the app still usable? Are any audio alerts also available as visual alerts? Some of this is dependent on your target audience (for example, there is not much you can do to make a music player accessible to a deaf person), but a well-designed app will consider all possible users.

It might seem odd to introduce design now, at the start of the book, but it’s an important thing to think about when you build your app. Although Xamarin provides you with the tools you need to write cross-platform apps sharing your core code, you still have to build the platform-specific layer, which includes different UI code for iOS and Android. As part of this UI layer, you need to consider what makes each platform different, and design each UI accordingly. For your app to be a success, it needs to be intuitive and look good on each platform, and part of this is consistency with what users of each platform are used to. I can’t overemphasize the word *consistency* enough—your app shouldn’t only be consistent with the platform but with itself. Any difference will cause user confusion, leading to a bad experience.

Ideally you either need skill as a designer, or access to someone with that skill. This can be easy in a corporate environment, but maybe not so easy for an indie developer doing everything on their own. The good news, though, is that the different OS vendors have you covered. They’ve all published a set of guidelines on how to build apps that not only look and work well, but are also consistent with other apps on that
platform. Google has Material Design, Apple has its Human Interface Guidelines. We’ll come back to these later in this book when we look at building UIs.

**USABILITY**

One of the key things about design is how usable your app is. An app that looks slick but is impossible to use is probably worse than an app that is bad to look at but works well. When you are designing your app the relevant platform guidelines can help ensure some consistency with other apps, but you are still responsible for ensuring a great user experience. While you are thinking of design also try out your user experience virtually—either with online tools (of which there are plenty) or simple tools like paper prototypes. With these you can mock up the UI and how it works and actually try it out—have people use the virtual or paper version as if it was a real app and see if it is natural to them. If they see the first screen of your app and have no idea what to do then you could lose a customer. Sometimes you only have seconds to draw a user in before they decide your app is no good and delete it, so it’s vital to make those first user interactions simple and obvious. One very popular book on user experience design sums up the most important principle in its title: *Don’t make me think!*.

1.2.2 **Develop**

This is the fun part—the bit we as developers love the most. Despite it being the best bit, it can also be less fun if we don’t have good tools to help. A good developer can code in a raw text editor, but it’s painful when you’re used to a full-featured IDE. Luckily, as Xamarin developers, we’re spoiled. On Windows there is Visual Studio, which is in my mind the best IDE around, especially when coupled with extensions like ReShaper from JetBrains. On Mac there’s Visual Studio for Mac, which uses the same compiler platform as Visual Studio on Windows. These IDEs give you code completion, easy-to-use refactorings, and in-editor indications of suspect or erroneous code. They also provide full debugging support for Xamarin apps running either in an emulator/simulator or on an actual device.

Seeing as all Xamarin apps are .NET apps using a platform-specific .NET framework, you have access to a whole host of libraries built by third-party developers that are also built on the .NET framework. Despite the differences between .NET on Windows, iOS, and Android, there’s a core subset that’s common to all platforms, so any libraries that target this subset can be used in all your apps, and any libraries that target a particular platform can be used against that platform. This gives you access to a wealth of code that does all manner of things, from connecting to databases, handling JSON, and constructing unit tests to providing frameworks for application development. Access to these is provided by a packaging tool called NuGet (pronounced NewGet)—these libraries are packaged into a zip file with multiple libraries separated by whichever platform they target. At the time of writing, there are almost 57,000 unique packages available on NuGet.org, and the tools to use these packages are built into Visual Studio. You simply right-click, select Manage NuGet Packages, and from there install whatever you need. We’ll look at these a bit more later because they’ll be used in the apps built throughout this book.
Testing is an important part of coding—all good coders will write unit tests as they code, if not before. Luckily Visual Studio on Windows and Mac helps in this endeavor, providing a way to run or debug tests. With live unit testing in Visual Studio on Windows, or extensions like ReSharper with dotCover or NCrunch, you can even see in the editor which lines of code are covered by tests, color-coded to indicate which tests pass and which fail, and with the tests continuously running so it moves from red to green as you write code. You can also get IDE extensions to use things like behavior-driven design (BDD), which allows you to write your tests in natural language.

When you code your app, you need to think about testing all the time, to the point of choosing design patterns that help keep your code separated enough that it can be tested easily and thoroughly. When we look at how to actually build an app later in this book, we’ll be using MVVM, a design pattern that enables this, and we’ll think about testing at every step.

All these tools make coding a lot of fun and reduce the drudge work by making it easy to automate writing boilerplate code and easy to refactor, so you’re never fighting with your code to improve it.

1.2.3 Test

Testing really goes hand in hand with coding. It’s something that should be continuous, and ideally automated. Testing every feature of your app takes a long time, and sometimes it’s very difficult to test every scenario, including the edge cases. If you can automate this, not only does it save time but it means you can fully test your app at every stage of development. That way you can catch bugs as soon as they appear, so you know what changes introduced them and you can fix them while these changes are fresh in your mind. If you don’t know about them till the end of development, it’s a lot harder to determine what caused the bug and find a fix.

The ability to run unit tests inside your IDE is a good thing because you have to think about how to test your code as you write. There are three types of testing to think about: unit testing, UI testing, and manual testing.

Unit testing

Unit testing is testing units of code, with a unit being the smallest possible isolatable piece of code. These are black-box tests against the contract of a class, designed to test that class in isolation. If that class has dependencies on other classes, those dependencies should be mocked out and given predefined behavior to ensure you’re just testing the one class in isolation.

For example, say you have a Counter class that has a Count property and an IncrementCount method. The behavior of the class is that when you call IncrementCount, the Count value goes up by 1. Here you can write a test that creates the class, calls IncrementCount, and verifies that the Count has gone up by 1 and only 1. If it doesn’t go up, the test fails; if it goes up by anything other than 1, the test fails. You don’t care about the implementation of the class—how it increments is of no interest, as this could change at any time. You just care about the contract—that IncrementCount increments the Count by 1. Once this test is written, you can be sure this method
Creating production-quality mobile apps

works, and if a bug appears in your app that looks like the `Count` is incremented by 2, you can easily see that if the unit test passes, the error is elsewhere in your app.

Another example would be a `SaveCount` method on your `Counter` class that saves the count to a web server by making a REST call passing some JSON. If your class is well written, it shouldn’t talk to the web server directly but abstract that out to another class (we’ll call it `WebService`) that actually makes the call. Your class just needs to construct the JSON and tell the other class which REST endpoint to use, passing it the JSON. In this case when you construct your `Counter` class instance, you have to pass it the `WebService` instance so it has something to call. As is, this isn’t well separated into a unit for testing, but we can change that.

If the `WebService` class implements an interface, `IWebService`, that defines the method to make the server call, you can instead pass the interface in when you construct your `Counter`. By doing this, you can mock the interface in your unit test—that is, have inside the test another object that implements the interface that you have control of. This way, you can call the `SaveCount` method and then inspect the call that was made to your interface and verify that the correct endpoint was called with the correct JSON.

**UI testing**

UI testing is the complete opposite of unit testing. Here you’re considering your app as a whole and testing it as if you were a real user interacting with the app. Xamarin provides a tool called `UITest` to enable this. It’s a library that allows you to write tests that look like unit tests and that are run using NUnit (a popular unit-testing tool), but these tests will launch your app on an emulator/simulator or physical device and perform interactions like tapping or swiping and allow you to query the UI to verify that everything works as expected.

For example, in an app that has a count shown in a label and a button that you tap to increment the count, you could write a UI test that launches the app, reads the value of the count label, taps the button, then re-reads the label, ensuring that the value has increased. Xamarin `UITest` does this by finding items inside the visual tree (the representation of the UI on screen) based on their name, ID, or contents. Once it finds these, you can read data or perform actions like tap, so a test could find the
CHAPTER 1  Introducing native cross-platform applications with Xamarin

count label based on it having an ID of Count defined inside the Android layout or iOS storyboard, and it could read the text property from there. UI tests can also call backdoors—these are special methods embedded in your app to allow you make your tests more easily. You can use these to do things like prepopulate data to avoid performing lots of repetitive steps in the UI, or to emulate situations that are hard to do through a UI test, like switching off WiFi on Android to test connectivity issues.

Once you have UI tests that run and pass on an emulator or your physical device, it would be nice to run them on more devices. One of the downsides to mobile development is the large number of possible devices and OS versions. On iOS this isn’t such an issue because most people keep their OS up to date, and there’s only a small range of devices. On Android it’s a massive problem as there are thousands of possible devices, and OS updates aren’t available to all due to manufacturer and carrier-provided tweaks. If Google updates Android, the device manufacturer needs to take that update and apply it to their version of Android and give it to the carrier, who then needs to apply it to their version before it’s available to be installed on the device. In a lot of cases, the manufacturer or carrier won’t do this, especially for older devices, meaning there’s massive fragmentation of OS versions on Android. At the time of writing, 85% of iOS devices are on the latest version of iOS. On Android, only 7.5% are on the latest, with 20% one version behind, 16% two versions behind, 33% three versions behind, and the remainder on even earlier versions.

What Xamarin provides to get around this is a service called Test Cloud. This is thousands of physical devices from different manufacturers with different OS versions set up in a data center, and you can rent time on these devices to run your UI tests. This way you can cover a wide range of device sizes and OS versions, and when you review the results you see not only which tests pass or fail, but you can get screenshots of every step, so you can see how the UI looks. This can be invaluable when you have a bug that only manifests itself on one OS or one screen size and you don’t have an emulator or physical device available that replicates it. This is integrated into Visual Studio—one click to deploy your test and run it in the cloud.

MANUAL TESTING

Yup, you’re on your own with this one. Manual testing means you have to interact with your app to try everything out. Ideally, if you’ve implemented good unit and UI tests, you’ve verified that your app works correctly. Manual testing should then be a quick sanity check to ensure any edge cases that can’t be tested using automation (such as launching external apps) are working. You should also manually test as you go along, to verify the usability of the app, verifying the user experience. Automated tests can verify whether something works correctly, but you still need to interact with the app yourself to see if things work intuitively. After implementing each feature in the app, you should try it out to make sure it follows your app’s design (as well as the design guidelines for each app), and to ensure it’s easy to use and gives good feedback.

For usability testing you should also consider hallway testing—going up to people and getting them to try the app out and see what feedback they give. When you do
this, you should try to mimic the real-world experience as much as possible. Just give them the app and leave them to it with no help, much like an end user who has just downloaded it from the app store. If they can’t work out how to use your app, you may need to reconsider the user experience.

### 1.2.4 Build

Continuous integration (CI) is the process whereby you continuously integrate your code changes into the core codebase and test it each time. In its simplest form, it’s having something that detects when code is changed inside your source code repository (such as on GitHub or BitBucket), builds your app, and runs your tests so you can see straightaway if you’ve broken the build or introduced a bug. This is a huge topic so I won’t cover it in much detail here, but I’ll touch on some areas that are relevant to Xamarin developers. There are a number of different CI tools around, and they all have some degree of support for Xamarin apps (even the most basic ones support Xamarin, because the tooling works from the command line).

There are hundreds of possibilities for the kinds of builds you could set up from a CI system. For example, you could have a check-in build that monitors your source code repo, and every time new code is checked in, it builds it and runs all your unit tests. You could then have another build that runs at the same time every night, getting the latest code, building it, running the unit tests, and then running all of the UI tests locally. Finally, you could have a release build that’s triggered manually, which gets the code, builds it, runs the unit tests, runs the UI tests, and (if all passes) packages the build up and deploys it to the app store.

The main thing you want with these builds is the continuous feedback loop—every check-in should be verified to see that it builds and the tests pass; if there are any errors, the person checking in the code should be notified so they can fix the error directly. Some CI systems can even take this further and provide precommit builds—the code you want to commit is built and tests are run, and only if everything passes is the code committed. If the build or tests fail, the commit is rolled back.

When choosing a CI system, you need to consider how good their support is for Xamarin apps and how much time you want to spend configuring them. Jenkins, for example, is a free tool and is fantastic for Java apps, but its Xamarin support is nonexistent at the time of writing, so setting up builds is a lot of work. Other tools have Xamarin support out of the box, so it’s easy to set up. The main one for Xamarin apps is Visual Studio App Center.

App Center ([https://appcenter.ms](https://appcenter.ms)) is described as “Mission Control for your apps.” You can connect to it using a GitHub or Microsoft account, point it to a Git repository in GitHub, VSTS, or BitBucket, and then it’s trivial to set up builds. You choose which type of app to build (iOS or Android), choose a branch to build from, point it at your solution or project file, choose the build configuration, and away it goes. You can also add signing certificates to allow your builds to run on real devices, and even launch your app on a device once it’s built as a sanity check to ensure that it works.
1.2.5 Distribute
You’ve designed your app, coded it up, tested it, and built it. Now you need to get it into the hands of your users. You could submit it to the relevant app store, but first it’s good to put it in the hands of beta testers.

App Center allows you to set up alpha and beta users and distribute your app directly to them. Once they have your app, you can push out updated builds as you fix bugs or make tweaks, and when they relaunch the app your users will be prompted to install these updates. This is direct to the users you want to do the testing, it’s not an open marketplace. Your users will only be able to download your app if they’re registered against it, so you have complete control of the distribution.

1.2.6 Monitor
Once your app is released and being used, you should monitor it. If your users are experiencing crashes, you can expect a slew of one-star reviews that will drive potential new users away. Your app will have bugs in it (that’s a fact of software development), but if you can monitor for these and fix the issues as soon as possible, you can minimize the impact. If you know that crashes have happened, you can do something about it immediately, and that will help with your app downloads. Remember, your customers won’t file bug reports and eagerly await a fix. They’ll just download another app that does the same thing as yours.

For the Xamarin developer, it’s easy to monitor for crashes using App Center. You can integrate the App Center SDK into your app as easily as installing a NuGet package and adding one line of code. This will track all crashes and upload stack traces to the App Center site so you can quickly see the line of code where it happened, get it fixed, and get a new version deployed. This is an invaluable tool—the quicker you can fix a crash, the less chance you have of losing users.

In addition, App Center allows you to do user and event tracking so you can see not only the demographics of your audience but also how they’re using your app. Again, this can be important in making your app as good as possible. If a particular feature of your app is being used regularly, then it’s something to work on and improve. If a feature is never used, then either it’s not wanted by your users or not obvious, so you can strip it out or make it easier to discover. If your app is popular in a particular country, you can add native language support for that country if it’s not there already. You can also track the path a user takes through your app, and if popular features are hidden behind a lot of interactions, you can change the user experience to surface those features more quickly.

All this is easy to add to your Xamarin app—just one line of code per user action to track what they’re doing. Demographic data comes as soon as you enable the SDK. If you capture the right data and use it correctly, you’ll have a powerful tool to help shape your app.
1.3 *Rinse and repeat…*

Monitoring is the final step in the mobile-optimized development lifecycle, a cycle that repeats with every iteration of your app. It’s no good resting on your laurels after a release; it’s time to fix bugs and add new features. Figure 1.11 sums up the steps.

- **Design**
  - On Android, follow Google’s material design guidelines
  - On iOS, follow Apple’s human interface guidelines
  - Think about usability and accessibility

- **Source code control**
  - Use source code control
  - Commit your code often
  - Git is well integrated with Visual Studio

- **Develop**
  - Use Visual Studio on either Mac or Windows to develop your app
  - For iOS, you need a Mac to build, for debugging, and to host the simulator
  - Visual Studio on Windows can use a remote Mac for building iOS apps. You can debug on a remote simulator screen shared over the network, so the Mac doesn’t need to be physically accessible—it can be in the cloud.

- **Test**
  - Code your app in a way that allows testing
  - Unit test as much as possible
  - Create UI tests to automatically test the user interface
  - Use Test Cloud to run your tests or debug on devices you don’t have

- **Build**
  - Use continuous integration (CI) to ensure that your code builds and that tests pass
  - Run your unit tests and UI tests on a regular basis to get a fast feedback loop
  - Consider continuous deployment—deploying builds that pass all tests to users regularly, via tools like App Center

- **Distribute**
  - Deploy early versions of your app to alpha and beta testers to get as much real-world testing as possible
  - Allow users to provide feedback and raise bugs
  - Respond to this feedback and fix bugs quickly, then redeploy to the alpha and beta testers

- **Monitor**
  - Monitor for crashes so you can quickly fix the bugs that cause them
  - Monitor the details of your users so you can ensure you’re targeting the right audience
  - Track the usage patterns of your app so you know what areas to focus on

*Figure 1.11* A summary of all the steps for each cycle of a production app
Keep your cycles small so it’s easy to change direction based on feedback from your monitoring or your users. But don’t make them so small that your users are updating their apps too often (next-day release is important for fixing bugs, but keep features updates at least a week apart). Regular updates are important because they make your users feel like your app is here to stay, and they’re good for promoting your app, as the stores highlight recently updated apps.

Now that you’ve seen this lifecycle in detail, it’s time to put some of this into practice and write some code that demonstrates the power of Xamarin apps. In the next chapter we’ll look at a design pattern that can help you build cross-platform Xamarin apps by increasing the amount of cross-platform code that can be shared across iOS and Android apps. Then we’ll follow tradition and build a cross-platform Hello World application.

**Summary**

In this chapter you learned

- Xamarin native apps are apps built in C# using a version of the .NET framework based on Mono that’s been customized to run on iOS and Android and using libraries that wrap the native device SDKs.
- Xamarin apps are better than native apps written using the vendor tools because you get all the power of a native app with all the features of the device and OS, but they’re written in a common language, allowing you to share common logic and code between apps on different platforms.
- Xamarin has tools for the mobile-optimized development lifecycle, covering developing, testing, building, distributing, and monitoring.
- There’s more to a production-quality mobile app than just coding. You first need to consider the design of your app to ensure that it’s suitable for the platform you’re targeting. You also need to code it well, ensure it’s fully tested, build it in a reproducible way, deploy it, and monitor it for issues once it’s in the wild.
Rewriting the same app for iOS and Android is tedious, error-prone, and expensive. Microsoft’s Xamarin drastically reduces dev time by reusing most application code—typically 70% or more. The core of your iOS and Android app is shared; you write platform-specific code only for the UI layer. And because Xamarin uses C#, your apps benefit from everything this modern language and the .NET ecosystem have to offer.

**Xamarin in Action** teaches you to build cross-platform mobile apps using Xamarin and C#. You’ll explore all the layers of a Xamarin app, from design to deployment. Xamarin expert Jim Bennett teaches you design practices that maximize code reuse and isolate device-specific code, making it a snap to incorporate the unique features of each OS.

**What’s Inside**

- Understanding MVVM to maximize code reuse and testability
- Creating cross-platform model and UI logic layers
- Building device-specific UIs
- Unit and automated UI testing
- Preparing apps for publication with user tracking and crash analytics

Readers should have some experience with C#. Mobile development experience is helpful, but not assumed.

**Jim Bennett** is a Senior Cloud Developer Advocate at Microsoft, specializing in Xamarin mobile apps.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/books/xamarin-in-action

“A guide to best practices for building production-quality mobile applications.”
—From the Foreword by James Montemagno, Microsoft

“True to its name, this book helps you get going and doing!”
—Prabhuti Prakash, Atos

“A well-rounded, up-to-date companion for your journey into native cross-platform development.”
—Mario Solomou, Appointai.com

“Your search for the best Xamarin book ends here!”
—Lokeshwar Reddy Vangala Micro Focus