Extracted from:

Seven Mobile Apps in Seven Weeks

Native Apps, Multiple Platforms

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Seven Mobile Apps in Seven Weeks

Native Apps, Multiple Platforms

Tony Hillerson

Series editor: *Bruce A. Tate*
Development editor: *Jacquelyn Carter*

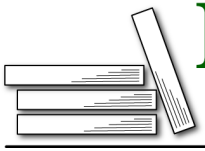
Seven Mobile Apps in Seven Weeks

Native Apps, Multiple Platforms

Tony Hillerson

The Pragmatic Bookshelf

Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

The team that produced this book includes:

Jacquelyn Carter (editor)
Potomac Indexing, LLC (index)
Candace Cunningham, Molly McBeath (copyedit)
Gilson Graphics (layout)
Janet Furlow (producer)

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-68050-148-3
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—July 2016

The Story So Far...

In the beginning there was a mobile platform, and it was good. It ran on the best mobile hardware available, both phones and tablets. It had an app store, where consumers could buy new software for their mobile devices. Developers liked to develop for this platform because the devices had fancy UIs, powerful SDKs were available, and people could make money from the app store to support their development.

There was a competing platform, though, and a lot of those devices looked pretty good, too. But it wasn't that easy to develop on. Its marketplace wasn't as good and the toolset was really hard to work with. If you were a developer on the first platform, it didn't seem very appealing to try to learn a whole different platform. If you had a good thing going, there was not a lot of value in learning all new tools instead of spending time building apps you already knew how to build. The best bet was to stick with what was comfortable and just ride that gravy train to an early retirement.

And then everything changed.

The Only Constant

In case you didn't already guess, the mobile platform I was just talking about was Windows Mobile. Back in the days before the iPhone changed the mobile space, Windows Mobile had a pretty good thing going. It already had everything the iPhone introduced: an app store (by a different name) and good tools (for the time), and it ran on pretty sweet devices (by those days' standards). It had a big competitor, or maybe more than one—who can remember back that far? The point is that if you were making your living by developing Windows Mobile apps then, there's a really good chance you no longer are. Not that it would have helped to have been proficient at more than one platform when iOS came out and made them all pretty much obsolete. The point is not how many platforms you're proficient on, but rather how proficient you are at learning new platforms.

As software developers, we've learned that the platforms we build software on change pretty often. If you pick a durable language to learn really well, chances are you can get some mileage out of it, but you're always having to learn new SDKs, new APIs, new libraries. And of course, even languages can become obsolete. With apologies to anyone still using it, I no longer include Cold Fusion on my resumé.

The *Seven in Seven* series has always been about expanding your horizons, about becoming a polyglot and a connoisseur of new ideas in software. You don't learn new languages or frameworks just because that's a fun pastime, although it can be, but because learning a new language, framework, or platform exposes you to new paradigms and idioms, different ways to solve problems. At some point you realize that recognizing the similarities and differences between languages or platforms makes you think about building software at a higher level, no matter what tool you're using at the time. At some point in your career you are no longer an "*X* developer"—where *X* is whatever platform you prefer—and instead you have become a "*software* developer."

Who Should Read This Book?

To get the most out of this book, you need to buy in to the premise of the *Seven in Seven* series. The best way I can express that is that you fully believe that continually exploring and learning about software development is your responsibility as a developer. What you get in return is the inspiration, guidance, and interesting problems that can help you maximize your time spent learning. That means that this book isn't a replacement for the official documentation; it doesn't spend a lot of time telling you how to get set up, and the platform coverage isn't comprehensive. Instead, this book provides you with a very useful tool: problems to solve in order to help you learn. Each week you'll build a nontrivial app over three days, and you'll also get some ideas of how you can take the app further yourself.

This book is focused on modern mobile platforms. Those are the "official" platforms: mobile web, iOS, Android, and Windows. Closely related are three cross-platform tools: RubyMotion, Xamarin, and React Native. The selling point of these tools is that, to some greater or lesser extent, building apps for one or more of the official platforms is easier, and you can share code between the platforms.

If you've never done any mobile development before, it's best if you have software-development experience of some kind. This book is most helpful for answering the following questions about each platform:

- How does it feel to develop on this platform?
- How does this platform compare to the others?
- And, for the cross-platform tools, what are their pros and cons over developing for one or more of the official platforms?

If you're a software developer who wants to dive into mobile development, this book will help you make a solid comparison between the platforms. If you're already a mobile developer for one platform, this book will give you a great framework for evaluating another platform. And if you've ever wondered or been asked by a client or manager if a cross-platform development tool is the right one for the job, this book will help you answer that question about three popular choices.

Seven Mobile Apps

Over the course of this book, we'll build seven mobile apps. The point of building the apps is to explore the platforms. To that end each of the apps have a number of design goals.

App-Design Goals

Not every app hits every goal, but each reaches a large enough subset to give you a really good taste of building for the target platform. Here is a list of design goals.

- The apps should have enough view code that you get a feel for how interface-building works.
- All apps have some level of navigation, multiple screens, or at least multiple views.
- All apps deal with displaying collections of data. This is something that happens all the time, so we cover it.
- The apps all make HTTP requests to a JSON API, so you should have a great idea of how to request data from the web. In some cases we'll also post data.
- Most apps include some information on designing and building interfaces for multiple screen sizes and orientations.
- Most apps give you an idea of how to store data locally.

These design goals won't make you an expert on the platform, but they will give you an idea of what it's like to work with common use cases.

How a Week Works

Each platform is designed to be explored over a week. There are chapters for three days of guided development, and each day ends with some suggested next steps for you to study or try on your own.

On the first day we start out by creating a project from scratch, so you get an idea of how that works and if there's anything interesting to consider there. We don't spend a lot of time on setting up the tools beforehand, though. We'll then start by building a simple feature, which will explore a few of the APIs.

On the second day we build on what we started and get deeper into the SDK, or possibly just deeper into the problem space of the app. Either way, you're thinking more deeply about the features of the app and we're covering a potential way to build those features.

On the third day we wrap up anything left over and implement a feature that covers one or more of the more interesting APIs on the platform.

What the Apps Aren't

There are a few things that we won't cover, and a few caveats to be aware of.

- We don't cover much, if anything, about animation. It's hard to describe in prose.
- The apps aren't meant to be visually appealing. You may even think they're downright ugly. That's fine, though, because you'll soon be proficient enough to make them look however you want!
- Whenever it makes sense, we spend time refactoring some of the first experimental steps we take. However, the finished products aren't necessarily examples of the best architecture. Instead, they contain enough good architectural choices to build a believable app of moderately small size so that you can focus on learning the platform.
- We focus very little on error-checking data from the API, validating user input, or dealing with edge cases. We stick to the happy path in order to learn. If you see some obvious bug waiting to happen, good! Design a way to handle the problem and implement it!
- In some cases we use third-party libraries because they're common, because not using them would cause unnecessary distraction, or because it's worth highlighting the package-management solution for the platform. Any third-party libraries we use aren't necessarily being endorsed, nor are they necessarily the only choices out there.

Approach these apps as a guide for exploring the platforms and tools, and you'll get the most out of the next seven weeks.

The Platforms

Let's take a look at each platform now and cover a little about what requirements each has.

Mobile Web

We felt like we just had to cover mobile web. There's continually debate about whether natively compiled apps are better than browser-based apps, or vice versa. That debate will probably never be resolved, but in the meantime there are apps to build and features to ship, and knowing where best to leverage the desktop browser, the mobile browser, natively compiled apps, or a combination of all three is important.

For the mobile web platform we'll build a responsive world-clock app completely in the browser. It will make a call to a separate API to get a list of world time zones. The only thing you'll need to build this app is a modern browser.

We start out with web first, because even though it's likely you've done HTML and JavaScript development before, it's possible you've never done responsive, mobile-first design. It's a good place to start and get your feet wet before we move on to the official native platforms.

Official Native Platforms

Official native platforms is the term I'm using to describe the current most popular mobile platforms and the official SDKs for developing apps on them. These are Apple's iOS, Google's Android, and Microsoft's Universal Windows Platform.

iOS

iOS needs no introduction. You can find everything you need to get set up at Apple's developer portal.[1] Getting set up to develop is free, but you'll need to develop on a Mac computer of some kind. We'll be using Xcode, Apple's official IDE for iOS development, so make sure you have that set up and running on your Mac before you dive into this chapter.

For iOS we'll build a weather app that shows current weather conditions at the mobile device's location. It will also allow you to search for a location

1. http://developer.apple.com

somewhere in the world and see the weather there. It will use our API's weather endpoint to provide current weather data.

Android

Android doesn't need an introduction either. To get set up to develop Android apps, install Android Studio.[2] You can develop Android apps on either Mac or Windows machines.

We'll build a currency-conversion app for Android. It will allow you to convert a certain amount of money from one world currency to another. We'll get that conversion data from our API.

Universal Windows Platform

The Universal Windows Platform, or UWP, may need some introduction because it's a pretty recent addition. Windows Mobile had its heyday and then was replaced by Windows Phone 7, with new tools and a modern platform. Windows Phone 8 moved that platform forward. With the release of Windows 10, though, Microsoft's aim is to provide one SDK for building apps on any Windows device, be it desktop, tablet, phone, or even Xbox. That's what the *universal* part means. The app we build will run as easily on a Windows 10 desktop as on a Windows 10 phone. However, Windows 10 is a requirement: developing for UWP requires a machine running Windows 10 and Microsoft Visual Studio.[3]

We'll develop a stock-quote app for UWP. It will display the latest quotes for a list of financial indexes, as well as a watch list of user-chosen stock symbols. It will get the stock-quote data through our API.

Cross-Platform Tools

In addition to mobile web and the official platforms, a well-rounded look at the mobile development ecosystem needs to offer a viewpoint on cross-platform tools. These are the tools that let developers build apps on the official platforms but with some purported advantage. The advantage seems to fall into two categories:

- Building apps with a certain language/toolset that some subset of developers prefers
- Building apps for multiple platforms while sharing code

2. http://developer.android.com/sdk/index.html
3. https://www.visualstudio.com

Usually a tool ends up offering both advantages. A ton of these tools are out there. I chose the three in this book based on two criteria: popularity and native app support. Popularity is easy enough to understand. By "native app support" I mean that these tools produce apps that are compiled and run natively, as opposed to running in a web view with a native wrapper. They use native components that users recognize, instead of custom components. I believe this approach is superior to any of the HTML/JavaScript-in-a-web-view solutions. Also, these three tools are well suited to building general-purpose apps, whereas other natively compiled tools are better suited to game development.

RubyMotion

RubyMotion has been around for a while and started out popular among Rubyists.[4] The original selling point of RubyMotion was that you could use Ruby to write iOS apps. But since 2014 RubyMotion has offered Android support as well, and that puts it in the category of cross-platform tools.

RubyMotion is a subscription service. A free "starter" version is available, but we'll be using some features that require a paid version of RubyMotion—for example, building an Android app for a specific minimum API level while still targeting the latest version.

To develop with RubyMotion, you need Ruby and the command-line tools from RubyMotion's site. (Refer to the instructions on the RubyMotion site for more information about setup.) To be able to build the Android app, you need to install the Android SDK. To be able to build for iOS, you need Xcode installed, and that requires a Mac.

For RubyMotion's app, we'll write a to-do-list manager. You can add to-dos to different lists, marking the to-dos done when you've completed the tasks. This will use the API's to-do endpoints.

Xamarin

Xamarin is a powerful cross-platform development product.[5] With a Xamarin.iOS project you can write code for iOS in C#, and with a Xamarin.Android project you can do the same for Android. Using a *Portable Class Library*, or PCL, you can write a library containing shared business logic, including networking code, and push as much as possible out of the platform-specific projects and into this shared code library. You can also share that logic with

4. http://www.rubymotion.com
5. https://xamarin.com/platform

Xamarin platform code written for Windows Phone, although we won't cover that in this book.

Further, you can use Xamarin.Forms, which is a cross-platform UI library. This plus a PCL allows you to write a completely cross-platform app in C#. We'll do both in the Xamarin chapter, starting out with Xamarin.iOS and Xamarin.Android and then converting the UI to Xamarin.Forms. Of course, to run the iOS version you need access to a Mac with Xcode, just as you'd need a Windows machine to build for Windows Phone. On Windows you can develop with Visual Studio, and on Mac you use Xamarin Studio, Xamarin's IDE. Xamarin was recently purchased by Microsoft, which made Xamarin free to use for indie developers and small teams.

We'll build a calculator app with Xamarin and a grid-based UI on both iOS and Android, code the calculator engine into a PCL, and then later convert the UI to Xamarin.Forms for a fully cross-platform app. We'll also integrate a currency-conversion feature using the API's currency-conversion endpoint.

React Native

Facebook's React Native is a very recent, very exciting cross-platform app-development tool.[6] You write apps in a modern flavor of JavaScript and use command-line tools to compile and run on both iOS and Android. React Native is still evolving quickly, but it has many things going for it: Facebook uses it for a number of its apps, and it's open source, with a growing community behind it. React Native is free, so there's no subscription to manage. To get set up, consult the React Native site. Again, you need a Mac with Xcode installed to run the iOS version of the app, and you need the Android SDK installed for Android.

The React Native app will be a note-taking app that uses the API's note endpoints. We'll allow the user to create and view notes, and then we'll geotag the notes with location data from the mobile device. We'll finish up by viewing those notes on a map.

System Requirements and Useful Tools

Before we dive in, let's talk about some of the supporting tools a bit. As you can see from the overview of the platforms, your operating system matters. You'll also want to get set up with the simulators and emulators that assist development and testing, and then get prepared to run the API by understanding how to call a locally running web server from these.

6. https://facebook.github.io/react-native

Development Operating System Requirements

I've repeated a few times that you need a Mac for iOS development, and that for the Windows Phone app you need a Windows 10 machine. So does that mean you need two computers, a Mac and a Windows machine, to get the most out of this book? Not necessarily, although it's an easy solution if you have one of each available. There are other solutions, though. One is to work only on a Mac and skip UWP for now. Or you could develop on a Windows 10 machine and skip any of the iOS parts, focusing on Android and UWP. That means you won't be able to build the iOS app, or the iOS apps from any of the cross-platform tools, unless you have access to a Mac.

Another solution, which I used, is to have a Mac with a virtual machine running Windows 10. I used a MacBook Pro, my main computer, and installed Windows 10 using Boot Camp. I then built the UWP app in Windows 10 and everything else in Mac OS. This seems like the perfect compromise, but I will mention a caveat. I was unable to install, let alone run, the Windows Phone emulator on either a virtual machine solution or Boot Camp on my Mac. There seems to have been some low-level incompatibility with the emulator's virtualization software and the Mac. Until this problem is solved, the only way to run the UWP app from a Mac to a phone is to buy an actual Windows 10 phone. The app will run fine on the Windows 10 desktop version, just not on the emulator. Luckily, you can buy a Windows 10 phone for a pretty low price.

If you run a Linux system you should be able to work with the Android examples, but you may not have full capabilities there. None of the code here has been tested on a Linux system.

Simulators, Emulators, and Mobile Devices

Developing mobile apps has an upside: you get to play with lots of toys. When you get set up for development on all of these platforms, you'll have access to a few choices for where to run the apps. Xcode comes with the iOS Simulator,[7] the Android SDK comes with the Android emulator,[8] and Windows Phone has an emulator too. You can also deploy any of the apps you build here to a physical mobile device. If you'd like, look into alternative Android emulators, such as Genymotion.[9]

7. https://developer.apple.com/library/ios/documentation/IDEs/Conceptual/iOS_Simulator_Guide/Introduction/Introduction.html
8. http://developer.android.com/tools/help/emulator.html
9. https://www.genymotion.com

So, where is it best to test? For day-to-day development, choose whatever works best for you. Work with all of the simulators and emulators, both third party and official, so you can get an idea of what works best for different development scenarios. The only advice I'll give here is that you should always test on a physical mobile device before you ship—probably on multiple devices, in fact.

For the purposes of this book, you should get to know the simulator and the emulators. Look up the documentation on each and learn how to:

- Simulate rotating the device.
- Interact with the menu button for the Android emulator.
- Run different device types and screen geometries for tablets and phones of different sizes.
- Clear local data and remove apps.
- Simulate physical device location.

Remember that the simulator and emulators work well for testing websites on mobile devices, too. Speaking of that, let's talk quickly about how to point to localhost.

Calling Localhost

When building the parts of the apps that require us to use the local API, we have to point to a local web host. From the iOS Simulator, it's no problem; just point to localhost. For the Android emulator, you need to point to `10.0.2.2`, and for Genymotion you need to point to `10.0.3.2`. You'll see this in the code when we get to it for each app, but it's worth mentioning now.

If you want to test with a physical mobile device, it's a bit harder, though. One solution I like is tunneling software called ngrok.[10] There are other, similar solutions. There is also the venerable Charles proxy.[11] With the tunneling solution, you'd point the app to a generated web host, which would then tunnel through to your locally running API. With Charles as a proxy, you'd instead point your code at some nonexistent host, such as `foo.bar`, proxy your mobile device through your development machine's address, and then set up a rewrite rule to translate `http://foo.bar` to your locally running API. Refer to the Charles documentation for more information if you'd like to try this.

10. https://ngrok.com
11. https://www.charlesproxy.com

As you can see, there are lots of little bits to think about when developing mobile software. These are the tips and tricks that you build up over the years to make development smoother for you and your team.

Don't Get Comfortable

Picking up this book and learning these platforms is a good move in your development career. It shows that you're not staying in your comfort zone. You're not going to let yourself get into a rut; you want to explore, know enough about the other platforms to make educated comparisons, and ultimately be ready when the next big change comes.

Online Resources

You can find the apps and examples shown at the Pragmatic Programmers web page for this book.[12] As we progress through each app, it's not always possible to show all of the changes, so the code in the book focuses on the important ones. The code for each project is separated roughly by day and significant change—in other words, the whole project is provided, including all changes for each section of each day. You can always refer to the code download for any questions you have about implementation.

On the web page for the book you'll also find the community forum and the errata-submission form, where you can report problems with the text or make suggestions for future versions.

12. http://pragprog.com/book/7apps