CHAPTER 1

THE MOBILE LANDSCAPE

1.1 INTRODUCTION

When the idea of reaching people first struck home in the dark ages, we wanted to find ways for people to use and pay for our services. We had to find a way to let people know these services existed. In early days there were town criers, then during the industrial revolution when we could reproduce and distribute text to a largely literate audience, we had broadsheets. Then came the catalogue where mercantile companies would list their wares for sale. Once we heard the first radio waves, one of the first things they did was to sell advertising on the radio. This graduated to television advertising. Then along came the Internet. Everyone had to get themselves a website and would put their website address in their print, radio, and TV ads. Along came Facebook and everyone created a Facebook page for their companies.

Now, everyone wants to have an app for their customers to download. These apps go with customers wherever they are and provide instant interaction between consumer and supplier. We can push advertising into them, take orders through them, keep in touch with friends and relatives, and of course play games, listen to music and watch videos all in the palms of our hands. These experiences require devices, operating systems, and apps, all of which require software companies, architects, and developers to produce them. Unfortunately, these devices and operating systems often change.

In today's computing world, there is one thing you can be sure of; the leading operating system (OS) platform will change. As recently as 7 years ago, Microsoft Windows Mobile was the leading smartphone platform and tablet computers, while mobile, were large and clunky and ran full versions of the Windows XP and Windows 7 OS. Then came Blackberry which took a lot of market share from Windows Mobile. But that only lasted until the iPhone came along in 2007 and we went from a feature phone dominated world to a smartphone dominated one. This set a new benchmark and became the leading mobile computing platform. In the same year, the Open Handset Alliance re-released Linux-based Android-powered smartphones. Then in 2010, Google launched its Android-based Nexus series of devices. By 2011, Android-powered smartphones made up the majority of mobile OS-powered smartphones shooting past the iPhone.

While phones were taking off, in 2010 Apple released the iPad. Tablet computing was not new and in fact Microsoft and its Original Equipment Manufacturers

Designing Platform Independent Mobile Apps and Services, First Edition. Rocky Heckman.

OEM partners had been trying to sell tablet computers since 2003. However, the iPad's sleek design brought tablet computing to the masses despite the clumsy and restrictive iOS operating system. This opened up the tablet computing market which Android was well suited for. After the iPad's initial success, by 2013 Google's Android-powered tablets had overtaken iPads as the tablet of choice. Additionally, although lagging considerably behind, Microsoft has re-created itself to make a run in the mobile and tablet computer markets as well. With Microsoft's massive install base, and very large developer ecosystem, they are likely to challenge Apple and Google in the mobile and tablet space eventually. With Windows 10 released as a free upgrade for over 1.5 billion eligible devices [1], it is likely to be the most common cross platform OS. That is, over half of Gartner's predicted 2,419,864,000 devices shipped into the market in 2014 [2]. Overnight the app ecosystem market leader could change again.

What this means for software developers, independent software vendors (ISVs), hobbyist app developers, and online service providers is that every few years they will have to retarget their efforts for a new platform, new development languages, new development tools, new skills, and new ways of thinking. This is not an attractive proposition for anyone. However, due to the success of the iPhone and iPad, software developers were willing to re-skill and even purchase proprietary hardware just to be able to develop applications for the new platforms. Then when Android devices surpassed the Apple devices, these same developers painfully went through the whole process again. Developers were forced to maintain three or more separate and complete codebases. This is the problem that Platform-Independent Delivery of Mobile Apps and Services solves.

If you are not planning a platform-independent strategy, you will likely be an ex-company in 3–5 years. Due to the rapid change of the consumer and enterprise mobile computing landscape, software developers must be able to adapt to new platforms, devices, and services before their competition. While cross-platform goes a long way toward this goal, it is still cumbersome and tends to lag behind a more platform-independent strategy. While it is not practical to get completely away from device-specific app code, the more you can move off of the device and put into a reusable back-end service, the less code you have to write and maintain when a new OS version or a new platform comes along. In this book we will examine strategies to do this, and provide future proof foundations to support changes in the computing landscape down the road.

Disclaimer: This book was written in late 2014. Everything in it was accurate at that time. If you are reading this in 2025, expect that a few things have changed. Just keep this in mind as we go through this so I don't have to keep writing "At the time of this writing…."

1.2 PREVIOUS ATTEMPTS AT CROSS-PLATFORM

1.2.1 Java

"Write once, run everywhere" was a slogan developed by Sun Microsystems which promised cross-platform capability for Java applications supported by Duke, Java's Mascot shown in Figure 1.1. This gained significant traction in the mid to late 1990s.

Figure 1.1 Duke

In the beginning of this era, the promise seemed legitimate. You could write the Java code once, package up your Java byte code in .jar files and run them on any system that had the Java Virtual Machine (JVM) interpreter. It worked so well that there are even C to Java compilers so your C applications can run with the same cross-platform reach that Java had.

The promise was that you could write code like this:

```
class CrossPlatformApp {
    public static void main(String[] args) {
        System.out.println("I am omnipresent!");
          // Display the string.
    }
}
```

And it would run on every computer and device that ran Java without compiling multiple versions for each target device. All you had to do was make sure that the target device had the correct version of the JVM installed on it.

This worked fine until various vendors started creating their own versions of the JVM to run on their platforms. By 2014, more than 70 different JVMs [3] had been created that could run Java applications, for the most part. The catch was that they were each slightly different.

If we take the Sun JVM to be the standard, some of these other JVMs were better, and most were worse, at interpreting Java byte code. Some of them such as the IBM J9 (http://en.wikipedia.org/wiki/IBM_J9), the Azul Zing JVM (http://en.wikipedia.org/wiki/Azul_Systems), and the Microsoft JVM (http://en.wikipedia.org/wiki/Microsoft_Java_Virtual_Machine) were better

and faster than the original Sun JVM. They even went so far as to add extra features and some constructs that were more familiar to traditional C/C++ programmers in order to make the transition easier for them.

While this seemed fantastic at the time, because it meant every platform vendor had a JVM to run Java, they weren't all the same. So what may work on the Sun JVM may not work on the Microsoft or IBM implementation. Even though some of these implementations such as the 1998–1999 Microsoft JVM outperformed the Sun version, they weren't entirely compatible with the Java 1.1 standard. This lead to Sun suing Microsoft and other JVM vendors in an attempt to try to defragment the Java playing field. The result was these other vendors stopped supporting their proprietary versions of the JVM and true high-performance, cross-platform capability for Java applications started to deteriorate.

This is a trade-off that you see repeatedly in cross-platform development. There has always been a compromise between running on many different devices, and getting as close to the hardware as possible for fast execution. It's the nature of computers. Each device may have slightly different hardware running the code. This means that the operating system and CPU may understand different instructions on each device. Java tried to solve this with the JVM. Different JVMs are written for the different environments, and they provide an abstraction layer between your Java code, and the nuances of the underlying hardware. The problems arise when one JVM interpreted the incoming Java code slightly differently than the next one and the Java dream becomes fragmented.

While Java is still widely used for applications, there are many versions of it depending on what kind of applications you are writing. There are four primary versions of Java that are supported by Sun.

- Java Card for smartcards.
- Java Platform, Micro Edition (Java ME) for low powered devices
- Java Platform, Standard Edition (Java SE) for standard desktop environments
- Java Platform, Enterprise Edition (Java EE) for large-scale distributed internal enterprise applications

All of them require a very standards adherent JVM to be installed on the target machine for them to run. Often the JVM can be packaged up with the application deployment, but the dependence on the JVM and specific versions of the JVM have made cross-platform Java apps troublesome. This is largely because you can never be sure of the JVM on the target device.

This is a common issue with most interpreted languages such as Java, Python, Ruby, .NET and any other language that is Just-In-Time compiled and run in a virtual environment or through a code interpreter. These kinds of things also reduce the speed of the applications because everything is interpreted on the fly and then translated for the CPU rather than being compiled down into Assembly or CPU level instructions which are executed by the CPU natively. This is why C and C++ and similar languages are referred to as native languages.

So while Java was a very good attempt at write once run anywhere, it fell short due to its dependency on the JVM. It still has a large install base and works very

well in many web app scenarios. It is also the primary app development language for Android-based devices which at the time of this writing was the world's leading mobile device operating system. Java can also be used to create apps for Apple's iOS-based devices through systems such as the Oracle ADF Mobile Solution [4, 5]. However, the vast majority of iOS targeted apps are written in Objective-C using Apple's Xcode environment. Due to the difficulty of developing with Objective-C, Apple introduced a new language called Swift for the iOS platform to help combat the hard translation form Java or C# for iOS developers and to improve the performance over Objective-C apps. At the time of this writing, Java did not work on the Windows Modern apps or Windows Phone platforms. Java does still work in the Windows Pro full x86 environment.

Java is perhaps the closest the industry has come to write once run anywhere. But it has been plagued by spotty JVM support, and a push toward more proprietary development for iOS and Windows to get the speed and integration to a more seamless state.

1.2.2 Early Web Apps

On August 6, 1991 the first website was created by Tim Berners-Lee at CERN (http://info.cern.ch/hypertext/WWW/TheProject.html). Ever since then, we've been pushing web browsers beyond their intended limits. From their humble beginnings as static pages of information to the preeminent source of all information and social interaction, websites and web apps have become ubiquitous in our connected world. It was inevitable that web access from every computer would lead to web apps being seen as the next great cross-platform play.

Web apps have been a popular attempt to run anywhere. All you need is a web browser on whatever device you have and you can use web apps. Well, that's the idea anyway. In reality this proved much more difficult than anyone hoped. Prior to HTML 5 you naturally had HTML 4. HTML 4 was still largely just a markup language designed to handle content formatting. Web pages displayed in the browsers were largely static text and images. Then early browsers such as Netscape and Internet Explorer 3 incorporated a JVM to interpret the Java code in the web pages. Web server software such as Apache and Internet Information Services could also run server-side Java code and send the product of the code back to the browser as an HTML page.

This worked pretty well, up until Sun sued Microsoft and they stopped including the Microsoft JVM with Internet Explorer. Since at the time it had become the world's most popular browser, that was a problem for Java-based web apps. It forced users to manually install a JVM from Sun which was an extra step most people weren't overly fond of.

This resulted in some interesting changes. Netscape produced its own web programming language called LiveScript in 1995, which it then changed the name to JavaScript when it introduced Java support in Netscape 2 in 1996. Meanwhile in the same year Microsoft produced Active Server Pages (ASP) and in an attempt to get around the Java JVM problem, it also included VBScript for the coding portion in ASP. JavaScript pretty much won the client-side scripting battle when Microsoft included support for it in Internet Explorer 3 but had to call its version JScript.

JavaScript became adopted as a standard known as ECMAScript which is in its fifth edition (5.1) released in June 2011.

In order to do interesting things with web apps, we needed to do things that HTML 4 simply couldn't do on its own. So one of the first things that was built into web browsers was a JavaScript interpreter. Now you could run scripts in web pages that could do things like display today's date, manipulate text in text boxes, and rotate pictures. This was nice, but in the days of Mosaic/Mozilla, Netscape, and Internet Explorer 3, it was really pushing the envelope.

To get a bit more out of the web apps, people started developing plugins for web browsers for things like audio and video. Macromedia introduced Flash in 1996 and it opened up all kinds of new opportunities to do very advanced graphics in a web browser through the Adobe Flash Player. By 2000, Flash was everywhere and even used to produce some animated TV commercials and 2D programs [6]. Around the same time in 2007, Microsoft introduced Silverlight which was a competing technology to Flash and offered audio, video, and graphics for web apps.

At the time, HTML had been reduced down to something like the following:

```
<HTML>
<HEAD></HEAD>
<BODY>
…
</BODY>
</HTML>
```

Everything between the <BODY>…</BODY> tags were references to JavaScript and plugins of some sort that offered extended capabilities that were not part of the HTML4 specification. This included embedded audio, video, and pluggable content.

The core of the problem was that while HTML was an open standard that everyone understood and agreed on, things like Silverlight and Flash were not. This lead to controversy about its use and widespread adoption due to the dependency on proprietary technologies. In fact David Meyer quoted Tristan Nitot of Mozilla as saying:

> "*You're producing content for your users and there's someone in the middle deciding whether users should see your content," [7]*

This sentiment essentially created a mistrust of proprietary technologies that started developers looking for standards bodies to create web standards that could fill the voids that things like Silverlight and Flash handled.

Although these technologies are still prevalent today, they met with some resistance in the mobile computing era. Some of it was due to Apple initially not allowing Flash to operate on its iOS platform. While this was fixed by allowing Adobe AIR apps to run on an iPhone which wrapped Flash content, it was enough for Adobe to re-evaluate its position on Flash and to withdraw support for Flash on mobile devices in 2011 unless it is embedded in Adobe AIR applications. Instead, they plan to "aggressively contribute to HTML5." [8]

Android having become the most popular platform by 2014 allows users to develop their apps in Java. This has brought a resurgence of development into the Java camps and solidified the once flagging language as a pre-eminent language in the mobile app development space. Eclipse tends to be the Java development environment of choice for Android apps as well.

This was an excellent move on Google's part because there were a lot of Java developers out there already. They could bring their existing skill set to developing for Android. This made the transition easy and afforded Google a huge advantage in catching up and surpassing Apple in the numbers of apps available in the Google Play store. You can also write apps for Android in C and C++ if you chose. This freedom has led to Android being the first platform people tend to release apps for.

Microsoft tried to keep Silverlight alive by allowing developers to create Windows Phone 7 apps in Silverlight. But their preference was for HTML5-based apps for Windows Phone and in versions of Windows Phone after Windows phone 8, Silverlight is no longer the platform of choice. You might say Silverlight is not a forward looking option for Microsoft. Microsoft also has a strong support for HTML5 app development in its platforms and has recently adopted the Apache Cordova framework for developing cross-platform apps in its Visual Studio product.

This myriad of technologies, plugins, mixed standards, and a very high dependence on browser version caused the web apps explosion to become muddled and difficult to develop for. Many lines of web app code are dedicated to just figuring out which browser and what version you are running in so you can do things slightly differently. You would have to check to see if a particular plugin was installed like Flash or Silverlight. You had to check to which browser version to know what kinds of scripting the browser supported and then run the version of the script for that particular browser version. To put it mildly, it was a nightmare. Having to write several different versions of the same code in the same web page to accommodate all the possible browsers their versions and plugins drove most web developers to liver failure due to excessive drinking. There were simply too many competing technologies amidst the browser wars.

1.2.3 Multiple Codebases

When it came down to it, developers knew they needed to support multiple platforms one way or another. They had executives wanting the apps on the latest buzzword compliant device. They had IT staff wanting their monitoring apps on the myriad of different devices they used and then they had their customers wanting the apps on every PC, smartphone, and tablet ever made in the 1990's. They had to come up with something so they ordered pizza and got down to duplicating their apps on every platform that 90% of their users used. Typically this meant Windows for desktops and laptops, iPhones and iPads running iOS, and Android-based tablets and phones. At a minimum, it was three separate development exercises. Windows apps were written with the usual Visual Studio and .NET combination. iOS apps required Xcode running on Apple hardware to write Objective-C apps. Android's Java-based apps were normally written in Eclipse on Windows- or Linux-based machines.

If you want to cover these three platforms you need at least two development machines: an Apple Mac and a Windows- or Linux-based machine. This was a real problem because not only did you pay more for extra hardware, but you needed more desk space. Early on, you couldn't write apps in Visual Studio for an iPhone or Android. You couldn't write Windows apps with Xcode. So you were forced to duplicate your equipment, effort, time, and skillset to cover the cross-platform world. But when you were done, you had apps on all the platforms that worked really well on the platform. Of course, if you wanted to add or change any features, you had to do it three times.

This also meant that you had to keep track of three sets of source code, and more importantly track three separate lists of bugs. If a severe bug showed up on one platform and not on the other two, you had to update and release just that one app. This could become a real problem. For example, if your Android app was full of security holes and issues that were present on some versions of Android and not others, it was very difficult to keep up with it all and keep the Android app patched without getting too far out of sync with the Windows and iOS versions.

Then of course there is the problem of the three different environments having very different capabilities. For example, you can create complex large databases for your app on Windows running on a laptop, but you can't do that in an iOS or Android app because they don't run on devices with large hard drives and lots of memory. So the Windows version tended to get ahead of the iOS and Android versions when it came to features. This often lead to developers creating their main apps on Windows, and having companion apps on iOS and Android that provided a subset of the features of the Windows version, or that complemented it extending the app to mobile devices.

As time went on and mobile computing became the norm rather than the exception, the iOS/Mac combinations were developed first, and web apps developed for internal usage or other platforms. In most cases, line of business apps is being developed for iPads, as much as for desktop computers. Server-side services are almost exclusively in WS∗ or RESTful web services. This has put web technology development in the lead, followed closely by native Android and iOS apps.

This is a very expensive way to do app development. It's not just the cost of the extra hardware. You duplicate your source code storage, management requirements, and man hours and worst of all; it requires some very different skill sets, which means you essentially maintain three development teams for one product. Then inevitably when the next big thing comes along, and the PC or device lead changes hands again, you have to scramble to re-skill for the new platform and create yet another set of source code or development systems.

This tends to cripple most small shops and they are forced to focus on one platform. Large development shops and large organizations with their own IT departments and development staff tend to choke on budgetary issues and time to market issues due to the complexity of launching on multiple platforms. Up until now this has made it near impossible for all but the largest companies to do multi-codebase development.

That is what this book is all about. I will cover the improvements to cross-platform standards, improvements in cross-platform development tools, and most

importantly strategies you can implement to achieve cross-platform app development with minimal overlap, duplication, reskilling, and cost.

1.3 BREADTH VERSUS DEPTH

One of the things that every app developer wants is for millions of people to use and love their apps. This means they need to be able to reach as many potential users as possible. This also applies to enterprise line of business (LOB) app developers that have to develop corporate apps that will work on any device employees have in the 'bring your own device' world. The catch is that for their app to have a rich engaging experience, it has to work on whatever device the user has at the time. Unfortunately this means that if you want a rich experience on iPhone, for example, you have to develop the app strictly to run on an iPhone. This app won't be able to run on a Windows Phone or Android-based phone. If you want to reach Android users you have the same problem. You can create an amazing app on Android, but it won't run on iPhone or Windows Phone. This is the breadth versus depth problem that app developers face when deciding which platform to target as illustrated in Figure 1.2.

In the past, developers have had to decide on the trade-off between reaching as many users as possible and the rich feature set which requires very device specific code. This has led to them picking the platform that has the most local market share that day. There are effectively three relevant platforms in the mobile space at this time: Android, iOS, and Windows. In this book, I will consider these three major platforms as our baseline.

This leaves us with a problem. If you are developing apps for mobile devices, which one do you choose? Choosing one means that in a few months or years you could be cut out of 60% or more of the addressable market. Putting all of that work into something that will be usable by less than half of the market is not very attractive.

There are some alternatives. You could make a responsive website that works on all mobile devices and adapts to different screen sizes. This gives you the broad reach across the user base but web apps tend to be very restricted to the features you can use. With technologies such as Phone Gap using HTML5, CSS3, and JavaScript as their primary programming stack, you effectively develop web apps, wrap them in a native app frame, and upload them to the app store. It's just HTML though, is that feature set rich enough? HTML5 has tried to fix that.

HTML5 has given developers the ability to create fairly rich web apps that can do most basic things native apps can do. HTML5 apps can access the location information on the device, local isolated storage, stream audio and video, and even replace Flash with the canvas elements that are available. With HTML5 it is possible to create very functional business applications such as Excel Online shown in Figure 1.3.

Figure 1.2 Breadth versus Depth

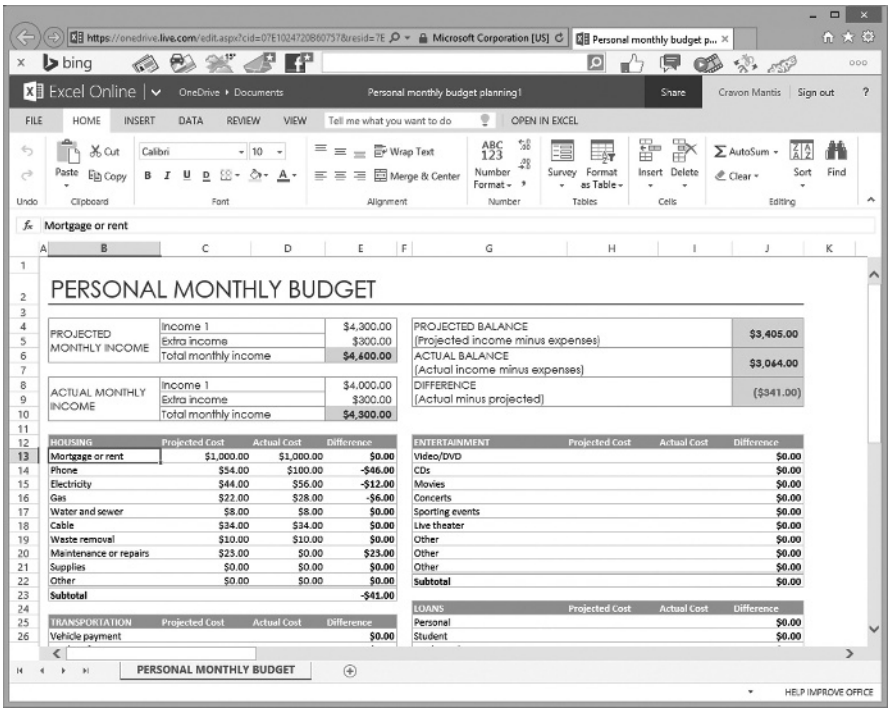

Figure 1.3 Excel Online https://office.live.com/start/excel.aspx

This is a very functional version of the Microsoft Excel that includes formatting, cell formulas, graphs, and data imports.

You can also create rich non-Flash games such as The World's Biggest Pac-Man. The World's Biggest Pac-Man (http://worldsbiggestpacman.com/) even allows users to create and upload mazes of their own for other players to play, all built with HTML5.

With HTML5 comes instant reach. Every major platform has an HTML5 capable browser that makes these web apps available to the users of those platforms. As good as this is, these web apps still lack important features like browser-independent off-line capability, access to other local documents and apps, integration with local contacts, e-mail, and calendars, etc. You still have to trade some features for reach. On individual platforms, native apps will be more attractive than web-based app in some circumstances.

1.4 THE MULTI-PLATFORM TARGETS

1.4.1 Traditional

By traditional platforms I am mainly referring to websites and clients in a client-server configuration. These can certainly be called multi-platform as different web

browsers have different capabilities. Additionally clients in client-server systems can be created on any platform providing that platform will allow communications with Internet-based services. This book really isn't concerned with thick clients as they tend to be self-contained and not normally mobile. That being said, you can certainly use the information in this book to create thick client apps that talk to a service back-end for whatever business, consumer, or game purposes you want.

This segment isn't really the one that developers think of when they think of new apps. However, it shouldn't be overlooked. The three major operating systems in play at the time of this writing, where apps are concerned, are Android, iOS, and Windows. iOS runs on iPhones and iPads, but not on MacBook or Mac desktops. Android and Windows however run on many more form factors. Android runs on phones, phablets, tablets, and notebooks. As of Windows 10, Windows runs on every form factor in common use today from embedded Internet of Things devices to phones, tablets, laptops, desktops, and large screen displays plus Xbox and HoloLens (www.hololens.com).

This is also why you need to make sure you don't walk into app development, thinking you are going to create an "iPhone" app or an "Android" app. If you do, you will certainly cut yourself out of a huge share of the potential market. If you don't consider the more traditional form factors such as laptops and desktops, you will miss the majority of the addressable market. There are still, at the time of this writing, more desktop PCs and laptops in use than there are iOS devices and Android devices combined. Go into this thinking cross-platform and cross form factor from the beginning. Don't cut yourself out of the majority of the market.

1.4.2 Mobile

This is a broad category but probably the one you are interested in developing for the most. After all, this is a book about mobile apps and services. It includes pretty much everything that is computerized, that is portable, but not that you wear. Phones, tablets, phablets, laptops, and other devices like these. Any device that a user can carry likely has device location capabilities, gyroscopic sensors, and cameras which open up a lot of new scenarios and potential for your mobile apps and services. This does apply to wearable and embedded devices as well but we'll cover those in a bit.

This is also the category that most people think of when they think of mobile apps. There are a lot of new developers coming on board, probably some of you reading this right now. This is the fastest growing segment of app developers. With millions of mobile apps in various app marketplaces, and some people making billions of dollars on them, it's not very surprising that this is such a hot topic. For your app to be successful, it has to offer a compelling user experience and features. It doesn't matter if it's a game, a family management app, or a business application; it will need to stand out amongst the crowd for you to be successful.

The app explosion, and accessibility of the tools, means that anyone with a good idea and some motivation can create good apps from the comfort of their home. This segment is making millionaires by the week with apps deployed to millions of people that only cost one dollar. Why not reach as many of those potential users as you can?

To reach this kind of success, an app that is integrated with a user's lifestyle, that connects them to the world, and that offers powerful features is required. You need to add value to their device, regardless of form factor, CPU power, RAM, or battery life. This means being able to extend beyond the device for connection to the world, and connection to more computing power than is available in the users' hands. Solid back-end services will get you there. These services will provide your app an edge over the myriad of other similar apps in the marketplace that are limited to the shell of the device.

1.4.3 Wearables

This is an emerging category in the mainstream. Even though the idea of a network-connected smart watch has been around since 2003 [9] and maybe before, they are starting to get a lot of public attention and most major manufacturers are coming out with watches or some other form of Internet-connected wearable computing device. If you are a Star Trek Next Generation fan, think of the communicators built in the Insignia on the uniforms. It provided not only communications, but also vital signs reporting, location reporting, and other network-connected services. From a more mundane perspective, think of smart watches, fitness bands, and glasses.

Now we even have fully self-contained holographic displays like Microsoft's HoloLens, an entire wearable PC that is less than the size of a bicycle helmet. HoloLens enables full persistent 3D holograms projected into your world. It comes complete with everything you'd expect in a mobile device such as location, cameras, and sensors, but includes the storage and CPU power of a desktop PC without the wires and goes where the user goes. This kind of thing introduces new gesture-based computing features and possibilities that we have only just begun to imagine.

Everyone will have some kind of wearable in the near future. In some cases they may not even know it. The ubiquitous nature of wearables such as new watches that have some kind of minimal GPS capability and health monitoring in them opens up very personal computing experiences for mobile app developers.

Many wearables are touting health and medical benefits. They can track everything from duration and route of walking, bicycling, and swimming to heart rates and even ultraviolet ray exposure and potentially glucose levels [10]. These kinds of lifestyle aspects make wearables attractive to certain segments of the population. In industry, think of patients, athletes, and military personnel. The military is already using wearable computing technology to track the heath of soldiers in addition to location, and transmitting live audio and video through satellite networks to command and control centers. Professional sports are using it to track things such as how far a player runs during a soccer match, to impacts they take in NFL games which may indicate they should undergo concussion screening.

For wearables to be useful, they have to be lightweight, with extended battery life. They tend to focus on data collection and upload. More advanced ones such as the Apple Watch, Android Wear, and Microsoft Band offer limited computing capability, but tend to integrate with the user's mobile phone. This means offloading processing power is critical. Being battery friendly is just as critical. A watch that you have

to charge twice a day isn't very useful. A fitness band that doesn't last a marathon is equally as useless. Glasses that make you gyrate your head and stare off into space will disrupt the normal in-public interactions. Wearables need to blend into our lifestyle and enhance it, not disrupt it. The services that power them can be powerful, and add a great deal to a user's lifestyle making them very attractive if they are built right.

1.4.4 Embedded

This category comprises all of the devices that contain an Internet-connected System on a Chip (SoC) or small computer such as Arduino, Raspberry PI or other proprietary systems. This devices are colloquially called the Internet of Things. The category isn't new, and we've had computers and applications buried in various things around the industry for many years. Cars have had embedded systems that mechanics would use to monitor and tune the car with. Racing has used this approach quite heavily for several years. Now that it is becoming easier to do with more SoC processors and operating systems, we even have drink vending machines with advanced processing capability in them [11]. These vending machines can look at you with built in cameras, and fairly accurately guess your sex, age, and height. They then feed this demographic data, along with your drink choice, up to cloud-based processing systems to aid in their marketing and machine placement choices. Not to mention the boring aspects of how much of which drink is left so the delivery person knows exactly how much of which drink to restock the machine with.

Gaining access to data that help with demographics and large-scale trend analysis is perhaps one of the largest areas of interest in the Internet of Things movement. Consider areas of home appliances. If you were a large grocery store chain owner, what could you do with a Smart Refrigerator? A fridge that knew what was in it, what the expiry dates of the various things were, and what kinds of things this particular family like to eat? With that kind of data you could automatically e-mail the family specials on items they need, recipes for food that they can make with things in the fridge that are about to expire, or allow them to integrate their food habits with their family doctor's records so he can make sure they aren't on the road to diabetes. The possibilities are boundless. It's a great time to be a mobile app service developer.

This is also the area that requires the most high performance data injection systems. When you start to consider all of the potential devices that will have Internet-connected computers in them, the numbers start to get very impressive. Consider smart utility meters. Every home will have one. How many devices are there, just in your city? How about New York, Hong Kong, or New Delhi? There are over 8.4 million people in New York City alone. If these meters and devices are all reporting daily, you will have to be able to scale to handle that traffic during reporting times, and scale back during non-reporting times or in the cases of personal devices, handle that kind of traffic on pretty much a 24/7 schedule.

So as you can see, we are in a situation in the industry where we need to think proactively about how we are going to future proof our designs. For a very long time we've been working on the back foot trying to create cross-platform technologies that haven't quite hit the mark. This is in large part due to the fact that one specific

technology may never be all things to all people because by the time it's done, there will be a new platform to contend with. With proper planning and a solid architectural approach, you can minimize the impact of cross-platform development for your apps and service. In this book, I will present some of the architectural patterns and features you can use to create and deploy rich cross-platform app services that will be comparatively easy to maintain across current and new mobile platforms.