Sandcrater Software White Paper

Native vs. HTML5 Mobile Applications

Ron DiNapoli
Sandcrater Software
July 1, 2013

This page intentionally left blank

Introduction

Mobile Applications come in many shapes and sizes. Additionally, they can be implemented in different ways. The "classic" mobile application is one written using a development environment and software developer kit (SDK) targeted for a particular mobile device. For example, iOS developers use a product from Apple called "Xcode" to write applications for iPhones and iPads using the iOS SDK. Such applications are referred to as *native* applications because they are built specifically for iOS devices. Similarly, Android developers use a product called "Eclipse" along with various plug-ins from Google to develop applications for Android using the Android SDK. In either case, the code written by the developer is only applicable to the platform they are developing for--that is--one cannot take code written for an Android device and expect to compile it in Xcode to run that program on an iPhone.

An alternative to writing native applications for a mobile device is to write an *HTML5* based application. An HTML5 application is a type of web-based application that takes advantage of the standards made available through HTML5. Assuming there is an HTML5 compliant web browser running on a given mobile device, any application written in HTML5 should run within that browser. This allows a programmer to write an application once that can be run on Android devices, iOS devices and any other device (mobile or not) that has an HTML5 compliant web browser.

The purpose of this document is to examine the differences between these approaches and make a recommendation on when to use each.

History of Cross-Platform Programming

HTML5 is an example of a "cross-platform" language technology—code written for one platform will work on other platforms unmodified. The early roots of cross-platform programming can be traced back to the early 70s and the C programming language where emphasis was put on writing "portable" code. The C language was designed to be usable anywhere a standard C compiler was present—be it a UNIX, DOS, VMS or other operating system. Prior to using C code many programmers of the time used what was called *assembly language* to write instructions for a computer in a language specific to the microprocessor inside that computer. C code "abstracted" this notion to provide a grammar and set of instructions that were *mostly* processor independent. And while this proved viable for the majority of code written there were still some areas of C (such as bit manipulation and the size of an integer) that were dependent on the processor and might function differently depending on the host system.

With the 1980s came the world's introduction to various graphical user interfaces. Apple introduced the Macintosh computer to the world which was later followed by other offerings such as Atari's GEM on their ST series of computers, Commodore's AmigaOS on their Amiga line of computers and finally, a little later, Microsoft's early versions of Windows. While many of the programmers who wrote programs for these graphical interfaces were using C (and later C++), it became more difficult to write "portable" code as each of these graphical environments came with their own unique SDKs. Cross-platform programming became an

exercise in separating out generic C code that did not deal with the user interface from platform-specific code that focused solely on presenting data and functionality to the user. The notion of writing code that could run on multiple systems had taken a bit of a step backwards.

In the early 1990s, a new paradigm became popular in some sections of the industry. By the 90s the Amigas and Atari STs had become pure hobbyist computers and were not considered part of a "cross-platform" solution. However being able to target Windows and the Macintosh with one code base was desirable. A number of desktop database vendors began incorporating the ability to write applications with a combination of scripting language and query language. Products such as "FoxBase/FoxPro" and "4D" that were available for both Macintosh and Windows allowed programmers to put together applications using one code base. You could write the program on a Macintosh in FoxBase and it would run under Windows using FoxPro. These applications could also be exported into double-clickable applications that looked just like the regular applications on these platforms. The only difference was they did always look quite right as they wouldn't always adhere to the user interface guidelines for a particular platform. This was part of the sacrifice one made for being able to consolidate to one code base. The cost savings of this approach, in many cases, outweighed the down side of not following a platform's specific UI guidelines. In most cases, the end users of such applications were usually not aware of the guidelines anyway and did not notice.

In the mid 1990s two new technologies came to the scene. The first was an increasing adoption of web browsers and the use of web pages as small "applications" that could provide functionality in some vertical markets that equaled or surpassed the functionality provided by a standalone application. The second was the introduction of the "Java White Paper" by Sun Computer. Java was a computing language that promised the ability to write an application that could be run on *any* computer including the ability to have a graphical user interface that would look "right" on whatever platform it was running on. The Java craze took over the industry for a number of years with the promise of Java-based operating systems that would alleviate the need for any other languages. While there were some applications that took full advantage of this technology, Java as a language to build cross-platform user interfaces never quite caught on due to problems with consistent implementations of the language on all target platforms. Java did settle in to be a very nice solution for server side computing and to this day is still the language used behind many of the web applications we use today.

In the first half of the 2000s, the notion of providing dedicated applications on desktop/laptop computers was losing popularity. Web technologies had advanced to the point were it was possible to provide more of an "application" type experience through a web browser. Many argued that the days of the traditional desktop computer were over and that we'd all be using thin-clients that only contained a feature-rich web browser for doing our computing work. However as we approached the mid-2000s, a new class of device began to gain in popularity that could not take advantage of the latest web technologies. We began to see more people using mobile devices. At first, this consisted of dedicated "personal digital assistants" (PDAs) such as the various PalmOS and WindowsCE/PocketPC devices. While these devices had rudimentary web browsers they could neither run web based application quickly or completely. The need for native applications became apparent again, although the mobile device market was *very* small compared to the desktop/laptop market.

The mobile landscape changed dramatically when Apple announced the first iPhone in 2007. There were more mobile devices in the hands of users with web browsers that were more capable than those that came on the early PalmOS and WindowsCE devices . A year later Apple announced the ability for developers to write their own applications for the iPhone and shortly thereafter the first Android devices began to appear on the market which also had a way for third party apps to be written for them. And these third party applications could present data to the end user in a manner consistent with the distinctive user interface design of these mobile devices. It looked as if native applications were making a comeback!

Finally , newer advances in web based technology such as HTML5, CSS3 and Javascript libraries (i.e., JQuery mobile)—all optimized for mobile devices—brought the ability to write code that could be run on multiple platforms to the mobile device arena. Just like with Java-based GUI applications and the old FoxBase/Pro applications from the early 90s, there are differences between the look and feel of a native application with that of an HTML5 based application. And so the discussion of which approach to take is renewed once again—only this time in the context of the mobile device ecosystem.

HTML5: History Repeats?

When viewed against the history of cross-platform computing, HTML5 seems like another chapter in the same book. However there has been much press on how HTML5 mobile apps can do everything a native application can do. Proponents of the technology argue that given the added cost of implementing native apps for every desired platform, the HTML5 solution is the right one every time. Additionally, tools such as Apache Cordova and Appcelerator allow HTML5 apps to appear as self-contained applications on a mobile device without the need to serve/consume the app from a web server. Is this finally the silver-bullet technology that conquers the cross-platform problem? Or will it suffer a fate similar to those technologies that attempted to solve the same problem before it?

I believe that, just like many of the cross-platform technologies that emerged before it, HTML5 comes closer than ever to solving the problem but does not conquer it. The areas where it falls short can be categorized as follows:

- Lack of application response time when served from a remote server
- Lack of true native GUI support/compliance
- Lack of support for the latest hardware/software changes on a given platform

The first area should be fairly self-explanatory. An HTML5 based application that is launched from a web-link and/or web page must be downloaded from the server, interpreted and presented to the user (just like any other web page). Even on a fast WiFi network, this will be somewhat slower compared to an application that is natively compiled for the device. In addition, if on a slower 3G/4G connection or in an

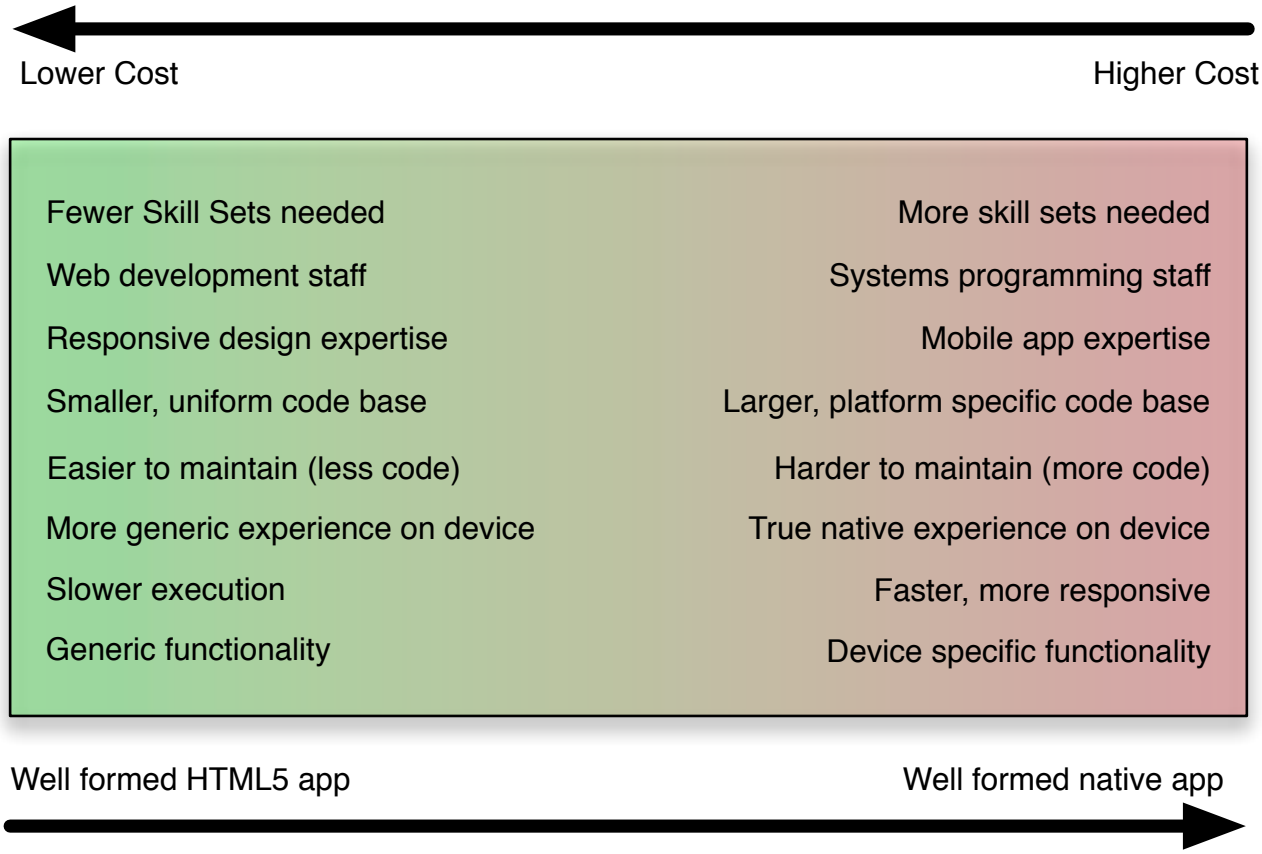
area where 3G/4G is not available such applications will be unacceptably sluggish and will not present a good user experience.

The second area is a little more subtle but possibly more important. While it is true that cross-platform technology has come a long way since the early 90s I also believe that user expectations have advanced as well. Consider the average iPhone user of today as compared to the average Macintosh II user of the early 90s. If the Macintosh II user ran a FoxBase application that user might get the feeling that something didn't look quite the same as an application such as MacDraw, but likely didn't realize that certain standard UI guidelines were not adhered to and did not realize it wasn't a native application. However with the iPhone, Apple has spent more time selling the entire "experience" to the user to the point where today's iPhone user is much more in tune with how an application is supposed to look and behave when compared to the Macintosh II user. So while the cross-platform technology used in HTML5 is *much* better than the cross-platform technology available to the FoxBase programmer, the target user is also *more* aware of what an iPhone application should look and feel like. I believe that a higher percentage of users can tell that HTML5 apps are not native apps and will favor native apps instead. Additionally, HTML5 based apps run the risk of not "keeping up" with interface changes on their target devices. Take the introduction of iOS 7 for example. An HTML5 app that was written in such a way to mimic use interface elements in earlier versions of iOS will look out of date when run on newer versions of iOS. The same could be true of Android devices should there ever be a major overhaul of the user interface elements and paradigm there.

Native vs. HTML5: What are the Implications?

Native applications (more specifically, up to date native applications) are the only way to guarantee that the native user experience is maintained in your application. After all, the end user bought a particular device because they wanted that device's unique user experience. An HTML5 based application will provide (from a historical standpoint) the best user interface experience of any existing or previous cross-platform technology but is still that—a cross-platform technology. It is still a "least common denominator" approach to user interface design and may not be able to take advantage of the latest technologies and guidelines available on a given platform.

Choosing between the two approaches is a function of multiple factors including resources available (cost), target audience and skill sets available as well as a good understanding of what you (the developer) "want" for the mobile application. Consider the following graph:



Similar to arguments supporting cross-platform efforts in the past, an HTML5 based application will provide the following advantages: You will need fewer staff and skill sets to build and maintain an application. You will be able to draw on "Responsive design" expertise you may already have in-house in connection with maintain mobile-friendly web pages. You will have a smaller, uniform code base across platforms (because there is only one "virtual" platform you are coding for—the HTML5 enabled browser). The code will be easier to maintain[1]. The only "disadvantages" to the HTML5 solution is that it will be a more "generic" user experience and has the potential of being slower/sluggish especially when served from a web server instead of being "wrapped/packaged" into a standalone application.

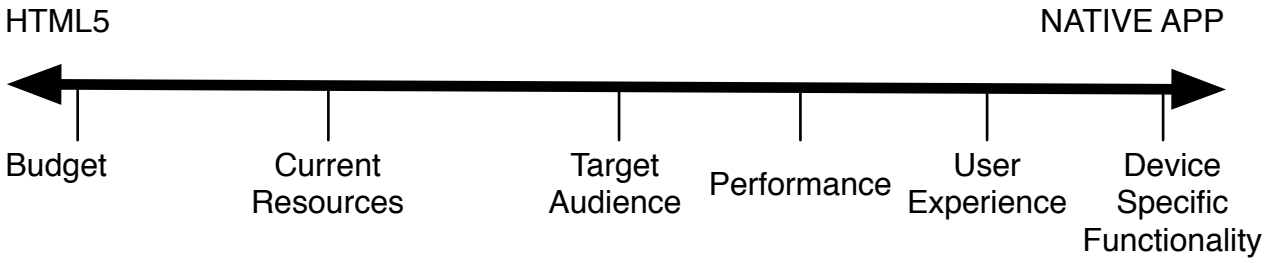
There's no denying that from a cost perspective native applications are much more expensive to develop and maintain. Native mobile applications require different skill sets across devices and unless you already have Java and/or MacOS developers "in-house" you will need to either hire or contract for those services. And since you will be (presumably) coding for multiple platforms your costs will be at least twice as much when compared to HTML5 development. The advantages you get for native applications are a true native experience for your app , the potential for your application to be faster and more responsive and the ability to take advantage of functionality that may be present on a particular device but not yet programmed into the HTML5 standard.

[1] This may be debatable. Since an HTML5/CSS3/JQuery Mobile type solution will rely on Javascript, the code base, though smaller, has the potential of being not as well organized.

Native vs. HTML5: Making a Choice

Deciding which approach to take is, as always, a business decision. What resources do you already have at your disposal that you can leverage? What is your budget for an application? What is your application's target audience and what kinds of usability do they expect? What kind of performance? Do you need to take advantage of device specific functionality?

Consider the following line-graph that relates the priority of certain business factors to the likelihood that you should develop natively or via HTML5:



Clearly, if budget is your biggest concern you will invest less money by developing an application in HTML5; especially if running across multiple mobile platforms. If the need to access device specific functionality is your biggest concern, then utilizing a native application is clearly the way to go as device specific functionality is not likely to be accessible from HTML5. Similarly, if you are very concerned about the user experience of your application matching the user experience of the device it is running on, native development is *likely* the way to proceed. If your organization and/or team already has a responsive design team coding in HTML5, you may want to consider utilizing their expertise to write a mobile application in HTML5 to alleviate the need to hire and/or contract out for a new skill set in native technologies. If performance is a key factor in designing your mobile application, you might want to think about native development first but know that there are steps you can take in an HTML5 world (such as using a application wrapping technology such as Apache Cordova or Appcelerator) to improve the performance of an HTML5 application.

Finally, you should evaluate the target audience for your application. What will they expect? If you feel that your application will have the greatest adoption rate if it looks and feels like a native application, then you should probably pursue native technologies first. However if you feel that the user experience does not matter dramatically (understanding that you will still design an application with good UI principles that may not share the exact same principles as the device it will run on), you would likely lean towards HTML5 development.

We hope that this white paper has helped you identify the concrete principles to consider first before making a decision on native vs. HTML5 based mobile applications solely on personal preference and/or ideology.

About Ron DiNapoli…

Ron DiNapoli is an independent mobile application and strategy contractor based in Ithaca, NY. He is a 26 year veteran of the IT industry with a specialty in systems-level programming. Ron has held various positions as a programmer, software engineer, team leader, line manager, IT director and collegiate lecturer with experience working for small companies and higher education. He is well versed in languages such as C, C++ and Objective-C with added experience in a variety of others. Ron's career has been full of cross-platform development efforts and he has experience with the successes and failures of such efforts for over 20 years.

Ron can be reached at ron.dinapoli@sandcrater.com