acmqueue

Mobile Application Development: Web vs. Native

Web apps are cheaper to develop and deploy than native apps, but can they match the native user experience?

Andre Charland and Brian LeRoux, Nitobi

A few short years ago, most mobile devices were, for want of a better word, “dumb.” Sure, there were some early smartphones, but they were either entirely e-mail focused or lacked sophisticated touch screens that could be used without a stylus. Even fewer shipped with a decent mobile browser capable of displaying anything more than simple text, links, and maybe an image. This meant if you had one of these devices, you were either a businessperson addicted to e-mail or an alpha geek hoping that this would be the year of the smartphone. Then Apple changed everything with the release of the iPhone, and our expectations for mobile experiences were completely reset.

The original plan for third-party iPhone apps was to use open Web technology. Apple even released tooling for this in its Dashcode project.[4] Fast-forward three years and native apps are all the rage, and—usually for performance reasons—the mobile Web is being unfavorably compared.

There are two problems with this line of thinking. First, building a different app for each platform is very expensive if written in each native language. An indie game developer or startup may be able to support just one device, likely the iPhone, but an IT department will have to support the devices that its users have that may not always be the latest and greatest. Second, the performance argument that native apps are faster may apply to 3D games or image-processing apps, but there is a negligible or unnoticeable performance penalty in a well-built business application using Web technology.

For its part, Google is betting on Web technology to solve the platform fragmentation problem. Vic Gundotra, VP of engineering at Google, claimed that “even Google was not rich enough to support all of the different mobile platforms from Apple’s App Store to those of the BlackBerry, Windows Mobile, Android, and the many variations of the Nokia platform,”[6] and this was before HP webOS, MeeGo, and other platforms emerged.

In this article we discuss some of the strengths and weaknesses of both Web and native approaches, with special attention to areas where the gap is closing between Web technologies and their native counterparts.

NATIVE CODE VS. WEB CODE

Implementing a software app begins with code. In the case of native code, most often these days the developer typically writes in a C dialect, as in the case of the iPhone. In our work at Nitobi (http://nitobi.com/) and on PhoneGap (http://www.phonegap.com/), we have had plenty of experience wrestling with the various mobile platforms from a native development perspective.

Of course, for various market or organizational reasons most developers or teams have to support apps on multiple smart platforms. Want to write an app in native code and hit every single mobile operating system? No problem if your team has the skill sets shown in table 1:

Table 1. Required Skill Sets for Nine Mobile OSes

Mobile OS Type	Skill Set Require
Apple iOS	C, Objective C
Google Android	Java (Harmony flavored, Dalvik VM)
RIM BlackBerry	Java (J2ME flavored)
Symbian	C, C++, Python, HTML/CSS/JS
Windows Mobile	.NET
Window 7 Phone	.NET
HP Palm webOS	HTML/CSS/JS
MeeGo	C, C++, HTML/CSS/JS
Samsung bada	C++

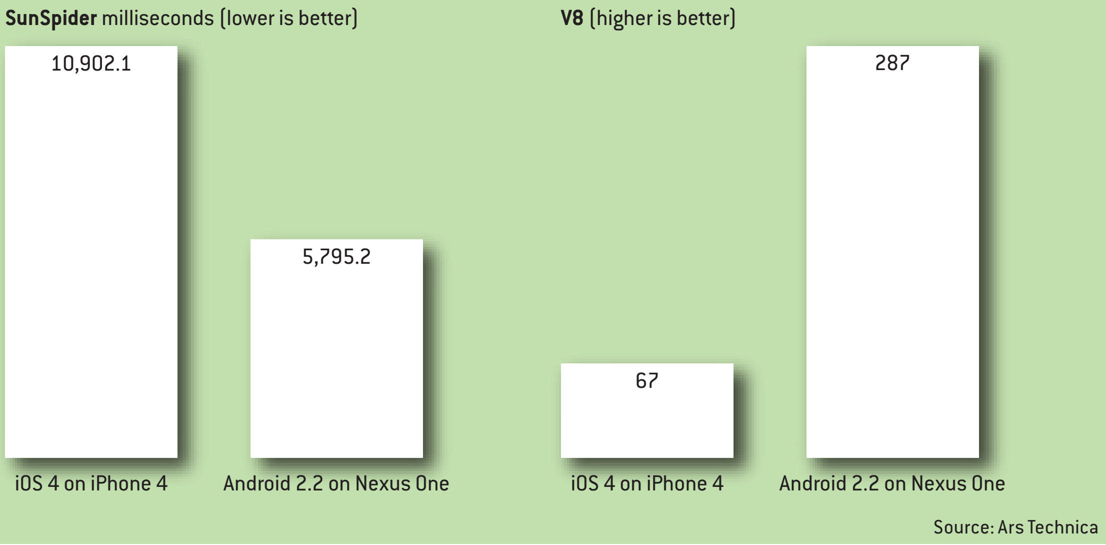
What makes things even more complicated are the differences among the actual platform SDKs (software development kits). There are different tools, build systems, APIs, and devices with different capabilities for each platform. In fact, the only thing these operating systems have in common is that they all ship with a mobile browser that is accessible programmatically from the native code.

Each platform allows us to instantiate a browser instance, chromeless, and interact with its JavaScript interface from native code. From within that Webview we can call native code from JavaScript. This is the hack that became known as the PhoneGap technique pioneered by Eric Oesterle, Rob Ellis, and Brock Whitten for the first iPhone OS SDK at iPhoneDevCamp in 2008. This approach was later ported to Android, BlackBerry, and then to the rest of the platforms PhoneGap supports. PhoneGap is an open source framework that provides developers with an environment where they can create apps in HTML, CSS, and JavaScript and still call native device features and sensors via a common JS API. The PhoneGap framework contains the native-code pieces to interact with the underlying operating system and pass information back to the JavaScript app running in the Webview container. Today there is support for geolocation, accelerometer, and more.

What is native code exactly? Usually it's compiled, which is faster than interpreted languages such as JavaScript. Webviews and browsers use HTML and CSS to create user interfaces with varying degrees of capability and success. With native code, we paint pixels directly on a screen through proprietary APIs and abstractions for common user-interface elements and controls.

In short, we're pitting JavaScript against compiled languages. These days, JavaScript is holding its own. This isn't surprising—JavaScript virtual machine technology is the new front line for the browser wars. Microsoft, Google, Apple, Opera, and Mozilla are all iterating furiously to outperform competing implementations.[5] Right now, by some benchmarks (http://arewefastyet.com/), Mozilla's SpiderMonkey is closing in on Google's V8 engine. JavaScriptCore by Apple, found in most WebKit browsers (which is on most mobile devices), is somewhere in between. The bottom line is that heavy spending by all the major players is fueling this JavaScript arms race. The benchmark by Ars Technica shown in figure 1 is an example of how these companies are marketing themselves.

FIGURE 1 **JavaScript Performance: iOS 4 vs. Android 2.2**



JavaScript is rapidly getting faster—so fast, in fact, that HP Palm webOS 2.0 rewrote its services layer from Java to the extremely popular node.js platform (http://nodejs.org/), which is built on Google's V8 engine to obtain better performance at a lower CPU cost (and therefore longer battery life). The trend we're seeing is the Web technology stack running at a low level, and it's in production today on millions of devices.

USER INTERFACE CODE

Things aren't as pretty when it comes to the user interface. Most native platforms have wonderful abstractions for common user-interface controls and experiences. No two platforms have the same, or even similar, user-interface paradigms, let alone APIs to instantiate and access them. The Web platform is consistent, for the most part, but the number of built-in or SDK-included controls is limited. You have to roll your own. Sometimes the differences among browsers can cause pain, but—at least in the modern smartphone world—most devices sport the very capable WebKit rendering engine, and only small differences prevail.

Unfortunately for the Web, those small differences are becoming a big deal. For example, on iOS, the CSS position property does not properly support a value of "fixed" (this issue has been corrected in the latest Android 2.2 code). BlackBerry operating systems earlier than version 6.0 sport a completely arcane browser for which there has been much suffering and toil at unfathomable cost to Web developer sanity. Fortunately, RIM has addressed a lot of this in 6.0, and in general, things are getting better.

Some operating systems include something called hardware acceleration. The iOS stack famously supports this concept in CSS transforms, which is how some Web frameworks achieve silky smooth transitions between view states. It's a technique first uncovered in Dashcode. It was painstakingly reverse engineered by David Kaneda, pioneered in jQTouch (http://jqtouch.com/), and released later in Sencha Touch (http://www.sencha.com/). Both are incredible Web projects and examples of what can be done when developers push the boundaries.

When we first started tapping into these next-generation mobile browsers, no framework worked properly across devices. Today there are more than 20 mobile frameworks, and support is being rapidly added to existing DOM (Document Object Model) libraries—not the least of which is John Resig's jQuery (http://jquery.com/) and jQuery Mobile (http://jquerymobile.com/); that code is improving and adding support for more devices every day. With tools like these, it's getting easier and easier to support multiple targets from a single Web-oriented code base.

Rapid execution and beautiful user interfaces aren't the whole story when contrasting the Web technology stack to native code. Web technology lives in a sandbox, which is also a jail from lower-level APIs that native code can access—APIs that can access device storage, sensors, and data. But this gap is being bridged, too. Most mobile browsers support geolocation today, for example, and iOS recently added Accelerometer and a slew of other HTML5 APIs. Given that the W3C has a Device API Working Group (http://www.w3.org/2009/dap/), it's likely we will be seeing many more APIs reach the browser in the near future. If the near future isn't soon enough, you can use PhoneGap (http://docs.phonegap.com/) to access these APIs today.

Of course, the Web technology stack (HTML/CSS/JS) is itself implemented in native code. The distance between the native layer and the browser is just one compile away. In other words, if you want to add a native capability to a browser, then you can either bridge it or recompile the browser to achieve that capability. *If a browser does not support a native capability, it's not because it can't or that it won't; it just means it hasn't been done yet.*

USER EXPERIENCE: CONTEXT VS. IMPLEMENTATION

Another area that has a big effect on both native and Web mobile application development is *user experience*, the term we use for the overall experience a user has with a software application. User experience can even extend outside the app. For example, we can use push notifications to wake up an application under certain conditions, such as a location change, or to spawn a new purpose-built application to handle different application aspects. Obviously, a successful user experience is crucial for successful application adoption.

Generally speaking, a mobile software project user experience can be divided into two primary categories:

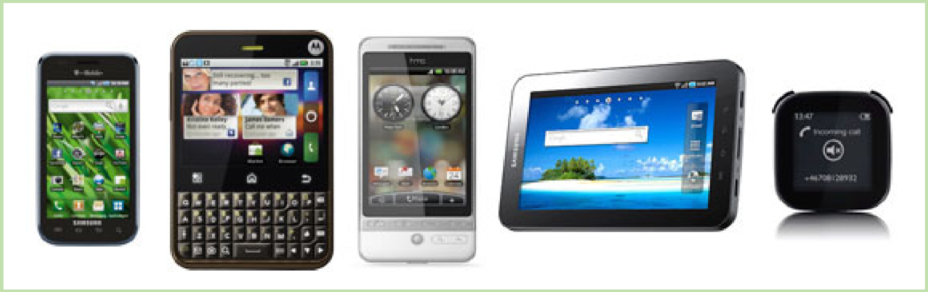
- The *context*—elements that must be understood but cannot be changed or controlled. These include hardware affordances, platform capabilities and UI conventions, and the environment in which your application is used.
- The *implementation*—elements that can be controlled in an application, such as performance, design, and integration with platform features such as accelerometer data or notifications.

THE CONTEXT

The context in which your application will be used affects users' expectations. The context for a single application may be radically different from one user to the next, even on a single platform. We're not really talking about *a* context; we're actually talking about *multiple* contexts. Let's look at the things that define the contexts to which a successful mobile application must adapt.

FIGURE 2

Variety of Contexts across Android Devoices



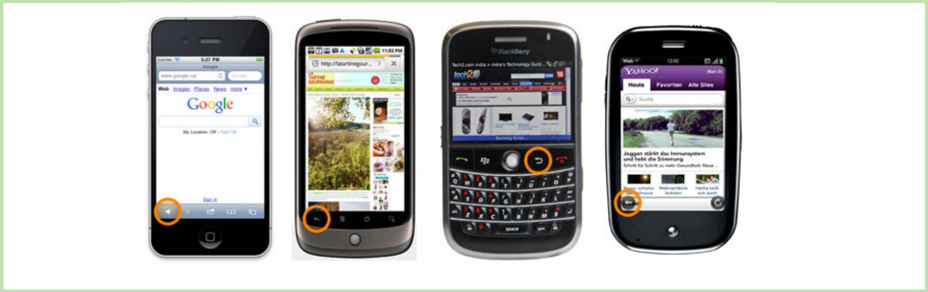
Hardware. The Android device ecosystem (figure 2) is a fantastic example of this variety of contexts, with devices varying significantly in terms of display (physical size, color depth, screen resolution, pixel density, aspect ratio); input (trackball, touchscreen, physical keyboard, microphone, camera); and capability (processing power, storage, antennae, etc.).

The combination of these properties greatly impacts how your application will appear, and the range of possible ways the user might choose to interact with it. If a particular combination doesn't exist today, it very well could tomorrow. A successful application must account for the habits associated with all of these hardware devices.

Platform Conventions. Each platform has its own user-interface conventions, typically described in a human interface guideline doc and evidenced in the operating-system interface. The variety of mobile Web browsers provides a prime example of how different these conventions can be:

FIGURE 3

The Variety of Mobile Web Browsers



A common user expectation is the ability to “go back” in the browser. iOS satisfies this with a virtual button; Android and BlackBerry devices rely on a physical hardware back button; webOS uses a back button and a back gesture. Whatever the method, users expect that they will be able to “go back” in your application.

Users also expect context menus. In the default Android and BlackBerry browser, the context menu is accessed through a physical button found at the bottom of the screen, close to the natural position of the thumbs. On iOS and webOS the context menu is accessed through a persistent virtual tab bar positioned close to the thumb at the bottom of the screen. The persistent tab bar at the bottom of the screen on devices other than iOS and webOs often produces a poor experience because users can easily accidentally hit their context menu or back buttons, causing an app to close unexpectedly. These are limitations with which both native and Web apps must contend.

Developers must consider approaches that make good sense for both data and users. HTML5 does support the concept of a menu element so a common abstraction is possible here, but the work has yet to be done.

Environment. Environment is the greatest wild card of all! Is it daytime or nighttime? Is the user standing or sitting? Standing still or in motion? One or two hands free? In a busy place? The variables are endless.

Where does that leave us? Expectations borne out of the context are not inherently cross-platform. Both native and Web implementations must provide designs and code that support these expectations. The good news for Web developers is that they can fall back on a familiar paradigm in the Web technology stack to satisfy user expectations.

THE IMPLEMENTATION

To produce the best possible user experience, implementations must deliver designs and code that support expectations set out by a user’s particular context.

Performance: The Hobgoblin of Software Development. Without a doubt, performance is a cornerstone of a great user experience. Like security, it is the most misunderstood and oft-used scapegoat of the software developer. It’s not uncommon to hear developers reject ideas with a flippant, “We can’t do that, it will negatively impact performance.” Rarely quantified and frequently cited, performance is the hobgoblin of software development. How do we quantify performance? Latency is a form of performance. Execution, the time an operation takes to perform, is another. We’ll address these separately.

Latency is a huge consideration in the mobile world. Be it a native or a Web application, there’s a performance penalty to downloading an app and the data it consumes or publishes through the network. Obviously, the smaller the payload, the faster the app.

Using JSON (JavaScript Object Notation)-formatted data is a good idea as it tends to result in a smaller data payload compared with an equivalent XML payload, depending on how the XML is formatted. On the other hand, XML data can make sense when returning HTML fragments that are to be inserted into a Web page rather than returning JSON-formatted data that, while smaller over the wire, needs to be converted to an HTML fragment using JavaScript. Your mileage will vary. Benchmarking is the only way to know for sure.

Another latency issue can be the initialization of code. Once we actually get the code into memory, it still needs to be parsed. There can be a noticeable performance penalty in this process. We can fake it and enhance the perception of performance with determinate or indeterminate progress indicators.

Execution time is, of course, a key facet of performance. When interpreting code (as we do for the Web with JavaScript), the more there is to interpret, the longer the execution time. Here the Web technology stack has some catching up to do. JavaScript, for all its leaps in performance, is still slower than native counterparts. On the other hand, the time it takes a programmer to write comparable logic in a native compiled language on multiple mobile devices may be worth the time penalty for execution; however, this will certainly require more maintenance than one code base written in JavaScript that can run on multiple devices, maybe with some tweaks per platform. Less code often leads to less and easier maintenance.

That said, the benefit of less code doesn't matter to the end user, who expects a responsive interface. The developer tradeoff is a larger code base—often vastly larger, considering support for multiple native platforms. In the world of native code, the main challenge is reimplementing to multiple targets. In the world of the Web, the main challenge is limiting your footprint as much as possible to produce a responsive user experience. That's not to say that one user interface can suffice in all environments, rather that the majority of the application logic is in one code base and then specific device-specific UI idioms can be implemented with conditional code. You therefore might want to implement slightly different functionality and user experiences appropriate to the expectations of the users of a particular device. For example, Android and BlackBerry devices have physical back and menu buttons, whereas an iOS device does not.

Another key point to remember is that even though the mobile industry is quickly converging on WebKit as the de facto standard for HTML rendering engines, every device and operating system has a slightly different flavor of WebKit. This means you should expect development to be similar to cross-browser Web development today. Thankfully, there are many libraries such as jQuery Mobile, Sencha Touch, and SproutCore that seek to address this.

All of this discussion of latency and execution of code means taking a tough look at the business goals of your application development initiative. Favoring data over decor is the most pragmatic approach. Gradients, drop shadows, bevels, embossing, highlights, rounded corners, and Perlin noise do not make an application useful or usable—they don't fulfill a business requirement—but they do impact performance. CSS gradients, in particular, are real devils for performance in the mobile world. You need to decide what your objective is: looking neat or providing a useful interface for data publishing and acquisition. You win some of these capabilities on some platforms with optimized (often hardware-accelerated) pixel painting with native code. It's not that these effects are impossible to achieve, but they should be used judiciously and only when they enhance and do not distract from the user experience. It is possible to deliver a great user experience that succeeds in the market; it just requires proper mobile Web development techniques and good user-experience skills that take into account the constraints of the environment.

Lovely Bounces and Beautiful Design. Of course, beautiful design matters. From aesthetics to intangibles such as the structure of a good program, software designers must commit to great design and to building on solid practices already in place. Scrolling via kinetic physics, lovely bounces, easing, and so forth create reactive interfaces that feel real and are a delight to use. This is an area where native controls are particularly good.

We have yet to solve the problem of native scrolling satisfactorily with Web technology.[1] There have been many attempts: iScroll (http://cubiq.org/iscroll), TouchScroll (http://uxebu.com/blog/2010/04/27/touchscroll-a-scrolling-layer-for-webkit-mobile/), GloveBox (https://github.com/purplecabbage/GloveBox), Sencha (http://www.sencha.com/), and jQuery Mobile (http://jquerymobile.com/). All of these address the scrolling issue but do not solve it as well as a native device. Even the Google mobile team is working on releasing a solution for this problem.[3] Without a doubt, this is the most common complaint the PhoneGap team hears, but we're one bug fix in WebKit away from it being a nonissue. The Google Mobile team has recently released its solution and code for WebKit-based browsers and platforms.[2]

Here's the rundown. The Web technology stack hasn't achieved the level of performance we can attain with native code, but it's getting close. We're confident that Web technologies will become indistinguishable from native experiences. In the meantime, Web developers must focus on delivering data while working diligently on improving the decor.

LOOKING TO THE FUTURE

As much as native and Web are pitted against one another in this debate, the likely outcome is a hybrid solution. Perhaps we'll see computing as inherently networked and (this is my sincere hope) free for anyone to access. We already see signs of a native Web: WebGL recently proved that in-browser 3D gaming is possible, even running Quake III (http://media.tojicode.com/q3bsp/)!

In the meantime, software makers must balance the Web-vs.-native debate based on an application's primary objectives, development and business realities, and the opportunities the Web will provide in the not-so-distant future. The good news is that until all of this technology makes it into the browser, hacks such as PhoneGap can help bridge the divide. I encourage developers not simply to identify software development trends but to implement them! If the Web doesn't fulfill a capability your particular application requires, you're presented with an exciting opportunity to contribute and close the Web/native chasm in the process. Q

REFERENCES

1. Ecker, C. 2010. Ars iPad application redux: where we're going; http://arstechnica.com/apple/news/2010/11/ars-application-redux-where-were-going.ars.
2. Fioravanti, R. 2010. Implementing a fixed-position iOS Web application; http://code.google.com/mobile/articles/webapp_fixed_ui.html.
3. Google Mail Blog. 2010. Gmail in Mobile Safari; now even more like a native app; http://googlemobile.blogspot.com/2010/10/gmail-in-mobile-safari-now-even-more.html.
4. Lee, W-M. 2009. Build Web apps for iPhone using Dashcode; http://mobiforge.com/developing/story/build-web-apps-iphone-using-dashcode.
5. MSDN, IEBlog. 2010. HTML5, and real-world site performance: seventh IE9 platform preview available for developers; http://blogs.msdn.com/b/ie/archive/2010/11/17/html5-and-real-world-site-performance-seventh-ie9-platform-preview-available-for-developers.aspx.
6. Nuttall, C. 2009. App stores are not the future, says Google. FT Tech Hub; http://blogs.ft.com/fttechhub/2009/07/app-stores-are-not-the-future-says-google.

LOVE IT, HATE IT? LET US KNOW
feedback@queue.acm.org

ANDRE CHARLAND is the co-founder and CEO at Nitobi Inc. He's been at the forefront of Web 2.0 software development for almost a decade and is an expert on the next generation Web. He is an advocate for usability and user experience and speaks regularly about how to keep users engaged and active on websites or web-based application. Most recently Charland presented on the Adobe AIR Tour throughout Europe. He's also been a speaker at the Voices That Matter web design conference, Adobe MAX, JavaOne and AjaxWorld. He is the co-author of "Enterprise Ajax", published by Prentice Hall last summer, and is the lead blogger for O'Reilly's InsideRIA.com.

BRIAN LEROUX is the lead architect at Nitobi Software with the prestigious title SPACELORD. He also has the dubious distinction of being the creator of wtfjs.com and crockfordfacts.com. To make matters worse he actually has a non-breaking space tattoo. He is also responsible for leading the direction on the wildly popular PhoneGap free software project that has the ambitious goal to provide a Web platform complete with Device APIs for nearly all smartphone operating systems.