[[1]](#footnote-1)

Building a Hybrid App and Website under a Unified Codebase to Maximize Business Results

Mike L. Lipman

*Abstract*—As consumers move their digital interaction with brands to internet-connected devices beyond traditional computers, businesses are increasingly challenged with providing a consistent experience across all touch points. Customers demand the ability to execute transactions, learn about products & services, shop, receive support and more from a brand’s digital presence regardless of the context of their interaction, and companies must follow through on these expectations to be successful. This paper will first define consumers’ changing expectations as it relates to accessing digital content. It will then discuss a Brute Force method for adapting to this new reality whereby businesses build separate pieces of software for each unique customer context. After explaining the challenges with this approach, the paper will introduce Responsive Web Design as a solution to supporting devices with divergent characteristics accessing the same content. Next, the influence of apps on web-based content will be explored, along with the resulting JavaScript tools, frameworks and best practices that allow for an “app-like” experience in the browser. Finally, technologies that marry web-based content and hybrid apps into a single codebase will be reviewed. This will culminate in an analysis for building a native vs. hybrid app within a broader digital strategy, focusing on the differences in performance and cost between the options. By employing advances in browser technology and the maturity of JavaScript frameworks, companies can leverage a unified codebase strategy to deliver the omni-digital brand experience demanded by customers while simultaneously keeping operational costs low.

*Index Terms*—Web sites, Web design, World Wide Web, Application Programming Interfaces, Browsers, Web 2.0, Programming, Internet, JavaScript, Responsive Design

# INTRODUCTION

T

HE rise of smartphones and tablets brought a well-documented change in how people consumed digital media. With comparable functionality to desktop and laptop computers but in a portable form factor, consumers gradually shifted their attention to these devices, despite the smaller screen and limited initial software. The introduction of app stores such as iTunes and Google Play in 2008 triggered a rapid increase in mobile media consumption which exceeded that of non-portable devices within 5 years[1]. This game changing concept quickly shaped the public’s expectation of what was possible on a mobile device. For example, a study from as far back as 2013 found that 85% of adults who completed a mobile transaction expected the experience to be better than on a laptop or desktop[2]. While companies were now afforded more digital channels to expose their brand to customers, this unique business opportunity also brought extreme challenges. Gone were the days of a limited set of platforms and browsers that would be used to access a company’s digital assets. Further, on-line experiences that had been meticulously crafted for consumption on a traditional computer screen either appeared awkward on smaller devices or ceased to be usable at all. Finally, mobile operating system manufacturers defined an ecosystem and experience that fundamentally diverged from the type of content being presented on a traditional web browser. Companies now have to adapt to this diversity in the digital ecosystem or find themselves at risk of becoming digitally irrelevant, which can quickly lead to overall irrelevance in the marketplace.

# BRUTE FORCE APPROACH

One mechanism to cope with this paradigm shift of digital consumption patterns and expectations involves a Brute Force approach. A completely separate experience is built for each major touch point, each with its own unique user interface and underlying codebase. Operationally, there are often separate teams that develop the visual display for each version along with varying operational playbooks for deployment and testing. The Brute Force approach for websites typically manifests itself in distinct sub-domains off of the company’s branded domain such as m.company.com, t.company.com and www.company.com, for mobile, tablet and desktop-optimized displays, respectively. Whereas the underlying business logic and back-end API integration layer is often shared, the functionality and information content may drastically differ, necessitating a separate codebase of HTML, CSS, and JavaScript assets (along with any media assets).

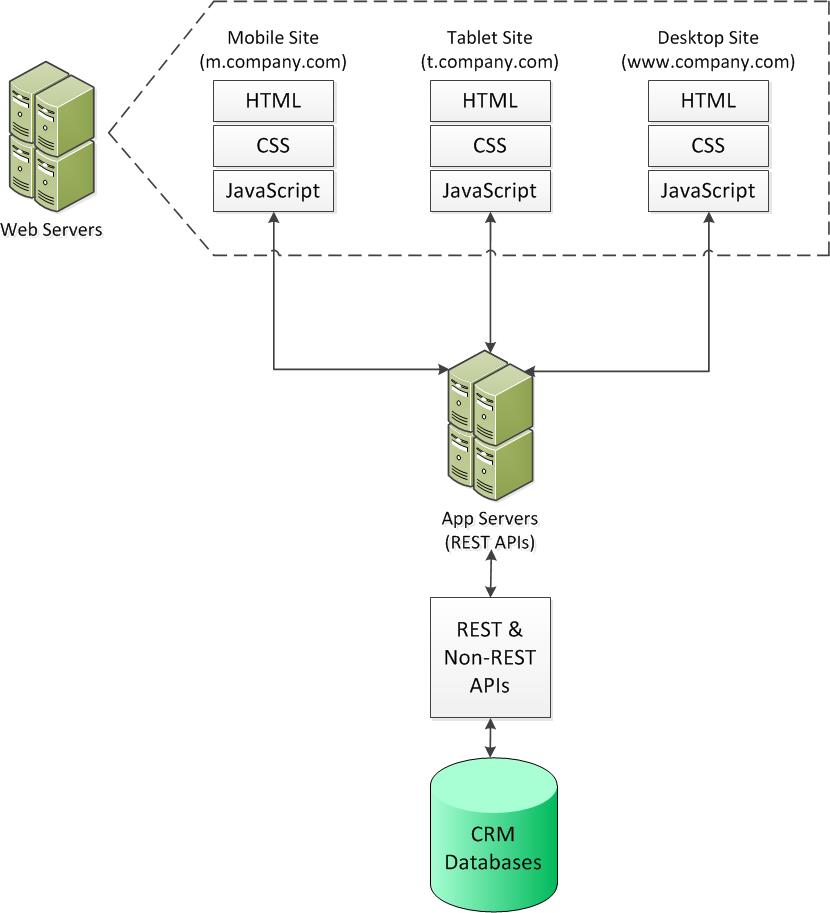


Fig. 1. High-level architectural diagram for Brute Force approach. A distinct JavaScript implementation for each codebase communicates with shared REST services and orchestrates the display uniquely per site.

Extending into the mobile app domain with this Brute Force approach requires building native applications for each individual platform. Unlike in the web domain where the programming language (JavaScript) and visual experience is governed by the common element of the browser, even these aspects are handled differently by each mobile O/S manufacturer’s ecosystem. For instance, building a native app requires Java development on the Android platform and Objective-C/Swift coding for devices that run iOS. Additionally, each platform dictates a unique end-to-end user experience pattern that needs to be adhered to in order to ensure acceptance on the associated app store[3].

An advantage of this approach is the opportunity to precisely tailor the experience for the customer’s context. For example, mobile apps permit access to the device’s camera whereas a traditional website running in a browser cannot. Companies can leverage this difference by offering a core set of functionality across all of their digital touch points while infusing a specialized utility that takes advantage of the camera when customers are using the app. Similarly, a feature that uses a browser-accessible component like the device’s location can be targeted for customers visiting the mobile/tablet optimized website whereas that same aspect can be dropped from the company’s desktop site. This is a concept known as Progressive Enhancement[4] and is crucial for molding the experience to the customer’s usage context. For developers implementing these experiential variations in the Brute Force approach, separate codebases allow the freedom to build these disparate functions independently, without concern for overlapping or conflicting programming logic. Designers are also able to craft unique experiences that precisely follow mobile O/S ecosystem guidelines for apps and take into account device viewport specs for websites.

Whereas Brute Force inherently solves the device variability problem, its methodology quickly becomes untenable for even the most resource-rich corporations. Device fragmentation, particularly in the Android ecosystem and even to a degree in iOS, exposes how poorly this approach scales. The following figures graphically illustrate the degree of variability for screen sizes of devices in both operating systems.

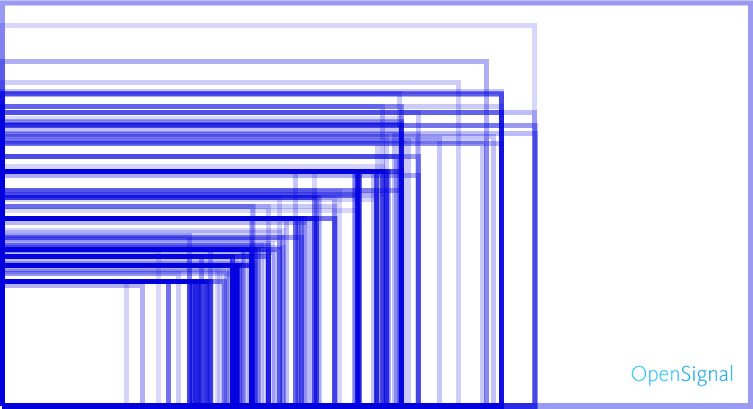


Fig. 2. Each box outline represents a unique screen size for devices in the Android ecosystem. Certain devices may have a different version of the Android OS despite sharing screen dimensions with other devices[5].

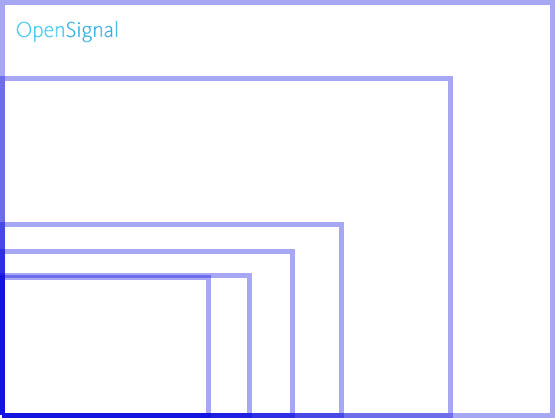


Fig. 3. Each box outline represents a unique screen size for devices in the iOS ecosystem. Certain devices may have a different version of iOS despite sharing screen dimensions with other devices[6].

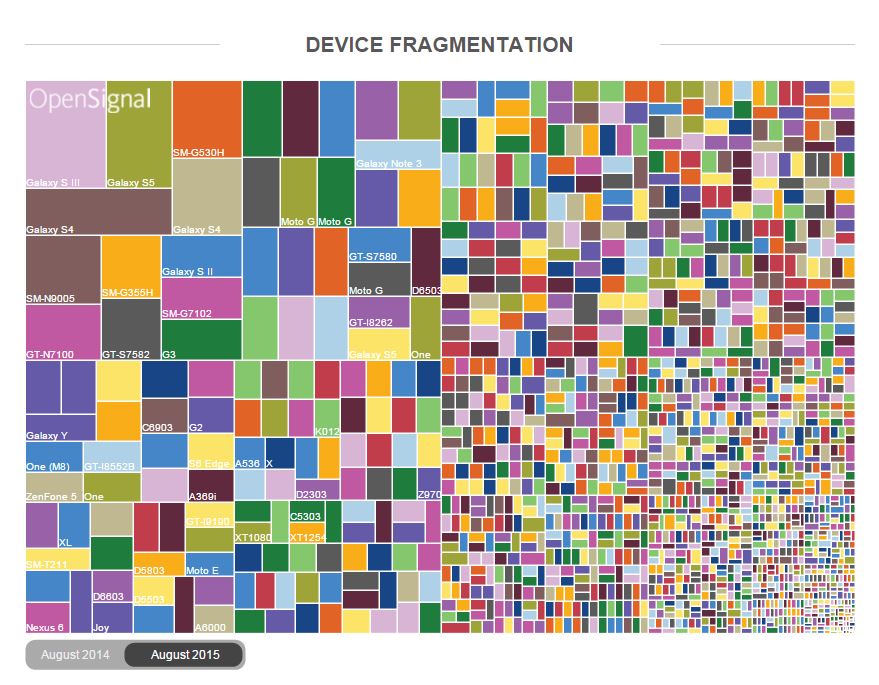


Fig. 4. Each box represents a unique device running a version of the Android O/S. Market share per device is represented by the relative size of each box[7].

Some device profiles are similar enough to group together, but the specificity embedded into each codebase of the Brute Force approach still requires a large number of variations to adequately cover the breadth. With each variant comes the cost of another codebase to maintain, alongside the associated operational playbook, infrastructure, and required technical skillsets in design, development and testing.

# Responsive Web Design

To overcome this problem, the open source community responded with several tools and frameworks. The concept of Responsive Web Design (RWD) is attributed to a blog post by Ethan Marcotte in 2010[8], and its primary concern is to attack the problem of viewport variability in the browser. It combines an existing practice of liquid page layouts with a new Cascading Style Sheet (CSS) spec called media queries, which finally gained widespread browser support in CSS3 after more than a decade in the making[9]. With just a few lines of CSS, developers can radically alter the layout of their HTML within the browser. This means that the same markup would be rendered quite differently depending on the viewport’s attributes, in particular its width. Further benefits include the concept of breakpoints (i.e. stacked media queries), which allow for a liquid display between targeted browser widths, and flexible images that permit graceful scaling of large width media down to a smaller screen. Opinionated visual frameworks such as Bootstrap and Foundation then emerged, which built on top of this generalized approach. Created by Twitter and the Zurb Foundation, respectively, they give developers a defined set of CSS classes, HTML snippets and JavaScript that allow one base of markup and styling to define the visual layout for any screen size. The precise implementations of these frameworks differ, but their open source nature permits designers to customize any aspect of each and therefore retain complete control over the final display.

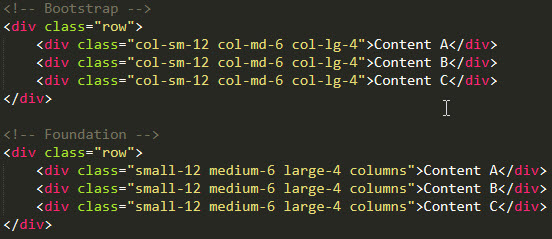


Fig. 5. Example using the responsive grid system in Bootstrap and Foundation. Each allows total customization of breakpoint widths and other styling.

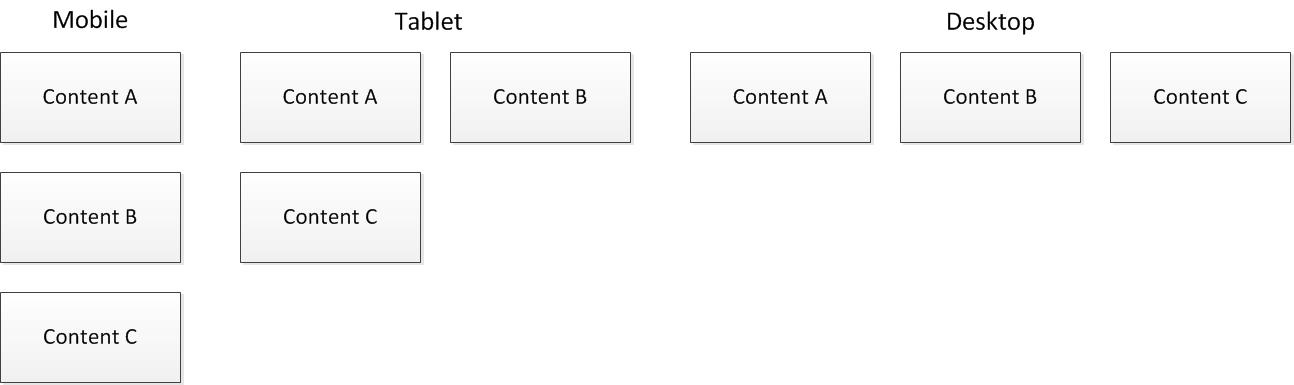


Fig. 6. Representation of how content will be laid out for each unique screen size using either snippet of HTML in Fig. 5. The exact breakpoint widths would be defined in CSS (or a preprocessor such as SASS/LESS)

RWD concepts paired with associated frameworks provide an effective mechanism to combat codebase proliferation for websites in the browser. However, the experience and “feel” that customers on mobile devices had come to expect from apps was still sorely lacking on mobile browsers. In a traditional web experience, users are presented with interactive elements on the screen which, when clicked or tapped, move to the next state following a complete re-paint of the display. In between, users are momentarily presented with a blank screen in which assets are loaded, HTML is rendered, and CSS styles are applied. This process is repeated for every subsequent state change. In a native mobile app, visual assets are embedded into the downloaded binary, restricting network downloads only to circumstances when dynamic data must be retrieved. This results in a far smoother experience when navigating between states since only aspects of the app impacted by the state change require downloading and rendering. Mobile apps also offered functionality that was simply impossible to replicate on the browser. By running on device platforms that directly expose APIs for hardware components such as the camera, accelerometer, and file system, developers could embed unique functionality in native apps while the browser experience was necessarily pared down.

# Single Page Applications & JavaScript

To address the disconnected feel of repainting on every browser state change, the Single Page Application (SPA) concept was introduced by the JavaScript open source community. There are several robust frameworks in active development at the time of this writing with AngularJS and EmberJS among the more popular choices. Though each framework has a unique way of implementing the SPA experience, most share a common theme of operating on a Model-View-ViewModel (MVVM) paradigm with a methodology for routing based on application state. Whereas a full discussion of MVVM is outside the scope of this paper, the key differentiator between it and a traditional software architecture like Model-View-Controller (MVC) is that the view layer is double-bound to its model (via the ViewModel) . This means that any change in the underlying JavaScript model can immediately impact the HTML view and vice versa. The engine in a framework like Angular orchestrates this interaction mechanism, which allows developers to maintain and manipulate application state across views. But whereas view-model double-binding is a critical cog in an SPA developer’s arsenal, dynamic routing is what truly imparts a native app-live feel to a browser experience. This process choreographs the user experience as a customer navigates through the site’s content. Taking advantage of the XMLHTTPRequest object that was standardized into the JavaScript implementation of all major browsers in the late 2000’s[10], AJAX-based network requests use this object to refresh only the necessary components of the screen. This entirely avoids the re-painting problem of a traditional client-server web-based architecture.

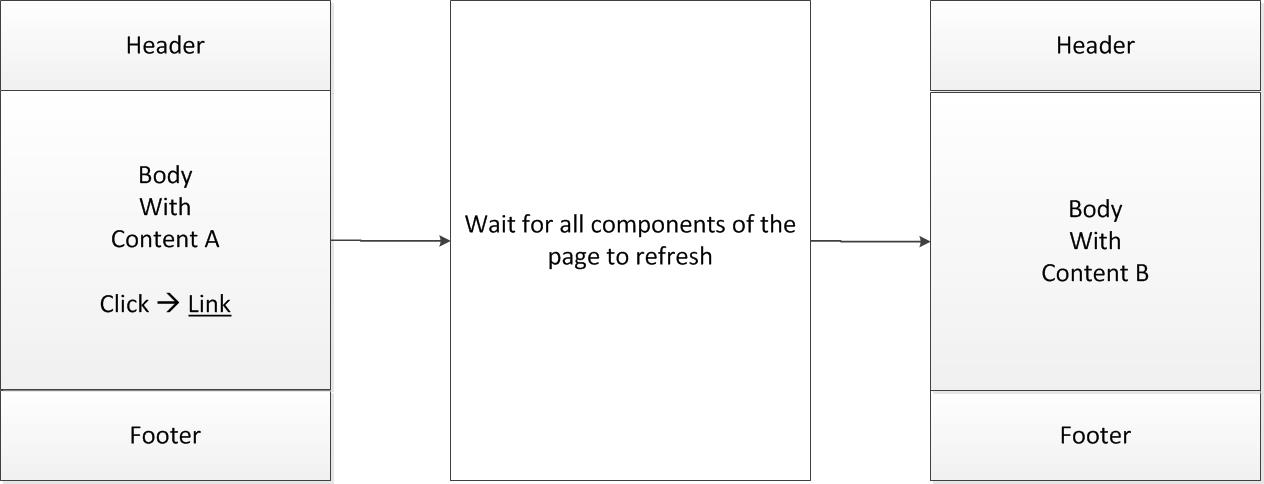


Fig. 7. Traditional page-to-page transition on the web. After clicking a link, the entire screen is re-painted, requiring assets to be re-downloaded from the server or retrieved from cache.

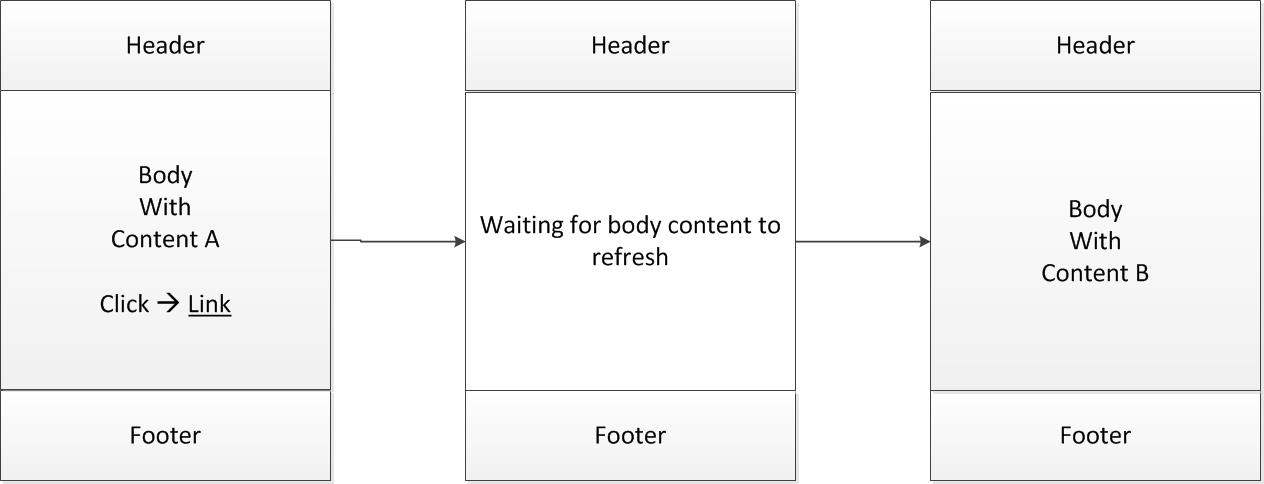


Fig. 8. SPA page-to-page interaction. After clicking a link, only the necessary components of the experience that need to change are downloaded and painted to the screen. Any components that don't require an update remain on the screen as is.

Browser manufacturers, in concert with standards groups such as the W3C, have taken on the task of closing the functionality gap between software written for native apps versus the web. As an example, the Battery Status API currently exists as a Candidate Recommendation[11] for standardization into a future version of ECMAScript, the coding foundation on which JavaScript is built. This interface defines the methods and properties available in the JavaScript language for detecting the status of the client machine’s battery. Previously, this information had been restricted to developers in the native app domain, but with this browser-based innovation, code can now be written that optimizes activities on a website for when the battery is below a certain threshold. Similar native app-inspired interfaces are in various stages of the W3C process including a device accelerometer API for detecting the physical orientation of the device, and a Service Worker API for allowing websites to work more effectively when not connected to the internet. While progress is being made rapidly in this area, the downside lies in the dependence on browser manufacturers to support the W3C standards. In general, Google (Chrome), Mozilla (Firefox) and Opera have been supporters of this move to expand browser functionality and improve web-based experiences, but Microsoft (Internet Explorer) and Apple (Safari) have been notable in their sluggish backing of the approach. Apple in particular has historically favored a clean separation between native apps and websites, as their user experience and developer guides attest[12].

# Hybrid App Development

Despite the aforementioned challenges with universal browser support, the SPA paradigm pared with at least some degree of advanced hardware access brought the web user experience ever closer to that of native apps. However the development ecosystems remain fundamentally different. There is a vast disparity in how a JavaScript-based web application is coded, built, maintained and run compared with a native Android implementation, which itself differs from a native iOS app. The net result is a single codebase for the web (now able to encompass all screen sizes and browsers), and separate codebases for each native O/S. Yet again, the open source community resolved this final chasm that prevented codebase unification with a framework appropriately named PhoneGap. Originally developed in 2009 by Nitobi, Inc., it was purchased by Adobe in 2011 and open sourced to the Apache foundation where it was re-named Cordova[13]. PhoneGap operates by a similar principle to jQuery, which was instrumental in advancing the use of JavaScript by abstracting differences in browser manufacturers’ ECMAscript standards support. However instead of abstracting JavaScript nuances, PhoneGap acts as a mobile O/S platform broker by exposing native APIs for areas such as the contacts list, compass, push notifications and more via JavaScript objects. Developers write standard JavaScript, just as they would for the web, and then compile their code using PhoneGap’s build process (now called Adobe Build). The outcome of which is a binary package (e.g. APK for Android, IPA for iOS, etc.) for each targeted mobile O/S, which can then be submitted to the respective app store.

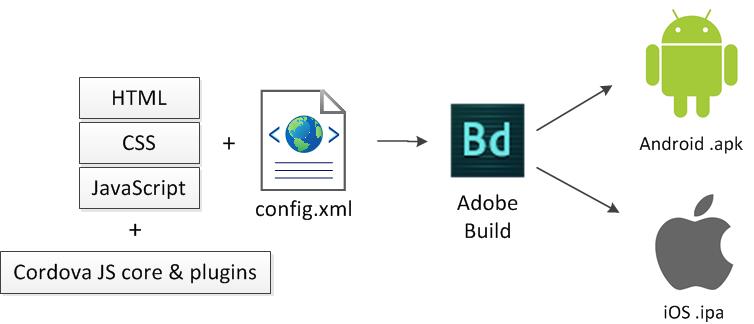


Fig. 9. Workflow for building a hybrid app using Adobe Build. The starting point consists of assets for a responsively designed SPA website combined with PhoneGap's Cordova libraries. A configuration file guides the build process to produce app binaries for the desired app stores.

The ecosystem has become more robust from its original core implementation, allowing third-party plugins to leverage its framework and expose ever more niches within the native realm. This has become increasingly important as O/S manufacturers continually add features to their product, thereby allowing JavaScript developers to deploy these so-called hybrid apps that nearly match the functionality of their native counterparts.

# level of effort analysis

With RWD principles, JavaScript SPA frameworks like Angular, and web-app bridges such as PhoneGap, developers are now fully equipped to leverage a single codebase to create and deploy experiences across the gamut of digital touch points. To understand the cost implications of using this approach, I will consider a project that creates a digital presence for a new brand where the mandate is to be universally accessible on a mobile/tablet/desktop via the web or app. For a cleaner comparison, I will consider a pure hybrid app (i.e. where no native components are injected into the app beyond the standard PhoneGap functionality), matched against purely native apps (one for Android, one for iOS).

The table that follows reflects the three different strategies that have been discussed thus far along with the relative amount of work involved for each step in building and maintaining these experiences. One unit of work represents a distinct effort needed for a given task (e.g. Front-end Development). 5.00 is the maximum possible since there are 5 experiences being built for the scenario in question. Anything less than 5.00 means efficiencies are gained for that task in a given strategy. Comparisons are only meaningful within a row of the table (i.e. within a task) – for example, it is not necessarily true that a 5.00 in Graphic Design and a 5.00 in Operations represent the same level of effort/cost, but if one strategy has a lower Operations value, it means it is more efficient relative to the others for that task. An explanation of each step and how the values were derived follow the table.



Fig. 10. Comparison of relative effort across tasks for three architectural strategies

In the User Experience (UX) step, wireframes are built from a set of user stories/scenarios and business requirements. The content area layouts and interaction design for each web breakpoint and app platform must be precisely specified, resulting in little to no difference in effort across the three strategies. The Graphic Design step is similarly implementation agnostic. Pixel-perfect design documents must be produced for each major web breakpoint and app platform regardless of the number of codebases. It should be noted that for the Single Codebase strategy, it is prudent to omit certain native experiences that may be overly difficult or time-consuming to replicate using HTML/CSS/JavaScript, but this would not lead to any significant cost savings in the UX or design steps. This aspect of visual design will be revisited when discussing the drawbacks of a Single Codebase approach.

The vast majority of cost benefit can be seen in the removal of code bases for Front-end Development. This row encompasses HTML/CSS/JavaScript as well as Objective-C/Swift/Java for purely native implementations. In building for the web, RWD permits one set of assets to be leveraged across all breakpoints with a terse mechanism for instructing the browser how to display them (see Fig. 5). Even in less optimal cases, specialized components for a particular screen size can simply be appended on to the code base, but this should rarely be needed given the robust nature of established RWD frameworks. Without using RWD on the web, as in the Brute Force approach, it becomes necessary to define the HTML/CSS/JavaScript for each specific implementation, which can even impact areas of the site that are componentized. It has been estimated that development costs when using RWD are roughly 25% higher than developing without these principles in a given codebase[14]. This extra effort is attributed to additional planning for styling and layout as well as unit testing that naturally results from multiple, unique displays rendering off of a single codebase. Critically, this 25% effort increase for RWD is only relative to a single codebase on the Brute Force approach. The scenario in question requires support for two additional, distinct web experiences, which are automatically covered by RWD but must be built from scratch using Brute Force. Therefore, in the units used in this example, Brute Force would require 3.00 worth of effort (1.00 for each of the three codebases) whereas RWD would only require 1.25 (125% of the effort for one of the codebases, then 0 for the remaining experiences). From personal experience on large websites, the stated RWD benefit may be overly optimistic, so for the purpose of this analysis, I will use a conservative savings of 50% for RWD *on each subsequent codebase*. This simplification translates to a work effort of 2.00 for RWD (1.00 + 0.50 + 0.50) compared with the 3.00 for Brute Force. It should be noted that the actual percentage savings can vary significantly, depending upon the strategy and use cases of the web experience being built. These variances result from exception cases where standard RWD methodologies fail to adhere to the design. Specialized logic for a particular screen size can be appended to the codebase to satisfy these situations, but this dilutes the benefit of RWD and clutters the code. If these special cases exist with high frequency, it is recommended to first revisit the design to eliminate as many inconsistencies as possible. Or, if they are in fact necessary, the RWD approach should be abandoned in favor of Brute Force, which will result in more maintainable code in this case.

The cost benefit of a single codebase becomes magnified when comparing against a native app strategy. For many applications that utilize a PhoneGap type of framework, appending an incremental piece of functionality translates to adding a simple enhancement on to the existing web codebase. Since both PhoneGap and the web code is written entirely with JavaScript, PhoneGap APIs are invoked conditionally when in an app context and will automatically execute in a cross-platform manner. For example, adding the ability to make use of the camera on a hybrid app built with PhoneGap involves only the following bit of code:

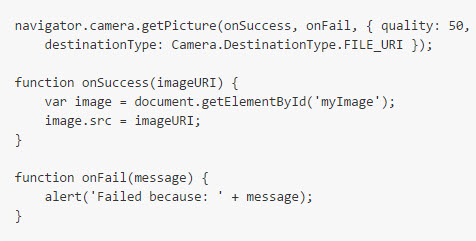


Fig. 11. Simple implementation of the PhoneGap Camera API that opens the camera app on the device and inserts the resulting image within a given target on the screen[15].

In a native development approach, there is no capability of code asset reuse with an associated web implementation because the languages are fundamentally different. This lack of reuse is exacerbated when additional app platforms enter the mix since each is not only incompatible with the language of the web, but also incompatible with one another. Referring back to the camera example, the way to mimic this functionality in a native app would be to stand up an entirely new codebase using Objective-C/Swift or Java – a very significant undertaking requiring additional development tools, testing methodologies, and human resources. The latter results from software engineers’ increasing specializing in a single discipline, reducing the probability of finding resources that can expertly craft two or more fundamentally different implementations[16]. While there is certainly a degree of effort in integrating PhoneGap into a web codebase, the choice of a web-app bridging framework offers a highly substantial cost reduction even in instances with a large volume of app-specific functionality. Though cost savings show considerable variation[17], a 90% reduction will be used as the estimated value per codebase. A more optimistic estimate is justified in this case since the existing web codebase will be leveraged and no native components will be layered on top of the hybrid app. This means that for each native app built using Brute Force or RWD/Native, only 10% of the effort would be needed for an equivalently functional hybrid app for a given platform. In the scenario in question, an app for Android and iOS is needed, which means the Brute Force and RWD/Native approach requires an additional 2.00 units of work (one for each platform) on top of the web effort whereas the Single Codebase approach only needs an additional 0.20 units.

API creation and integration with a CRM and/or other back-office systems is a necessary step regardless of the front-end approach. A shared set of REST or SOAP-based services are often built that can be consumed by any digital channel, whether it be the web, an app, IVR or other customer interaction channels. The approach to utilizing these shared services may be different depending on the context, but the choice of strategy will have virtually no impact to the cost of building and supporting those services.

The testing process can also expect similar costs across the spectrum of strategies. Though app simulators and browser user agent spoofers are readily available, truly effective and representational quality assurance must be done on physical devices. As a result, testing engineers should always run through the gamut of scenarios independently on each unique implementation irrespective of how the experience on a given device came to be developed. It should be noted that the same reasoning does not necessary apply for developer unit testing and certain levels of system integration testing. For these lower level tests, it is acceptable practice to run both visual and non-visual simulators to get a rough idea of quality. When using a single codebase, automation tools like Grunt can run a set of test scripts using frameworks such as Mocha or Protractor, and also push out saved or committed code changes to a variety of test devices on demand. This continuous deployment in lower environments can even extend to apps using a tool like Adobe Build’s Hydration, which is part of the PhoneGap suite. This cost savings attributed to developer testing has been built into the figures for the Front-end Development step.

The operational tasks involved in this example consist of deploying the initial version of the web and app experiences and then the subsequent updates. Predictably, a single codebase provides superior cost-benefit compared with other tactics. Only one code push is needed for websites designed using RWD whereas each codebase in a Brute Force approach must be pushed separately, since the unique sub-domains necessitate different virtual or physical web server instances. On the app side, there are no initial savings for even those constructed with web-app bridges since they too must be built into a binary. However, the benefits are reaped for subsequent updates. Any change related to the User Interface (UI) for a native app necessitates a re-build and re-issue to the associated app store as an update, where it must be downloaded and installed by the customer. No such requirement exists for hybrid apps since a tool like CodePush can automatically deploy new HTML/CSS/JavaScript assets when the customer loads the app.

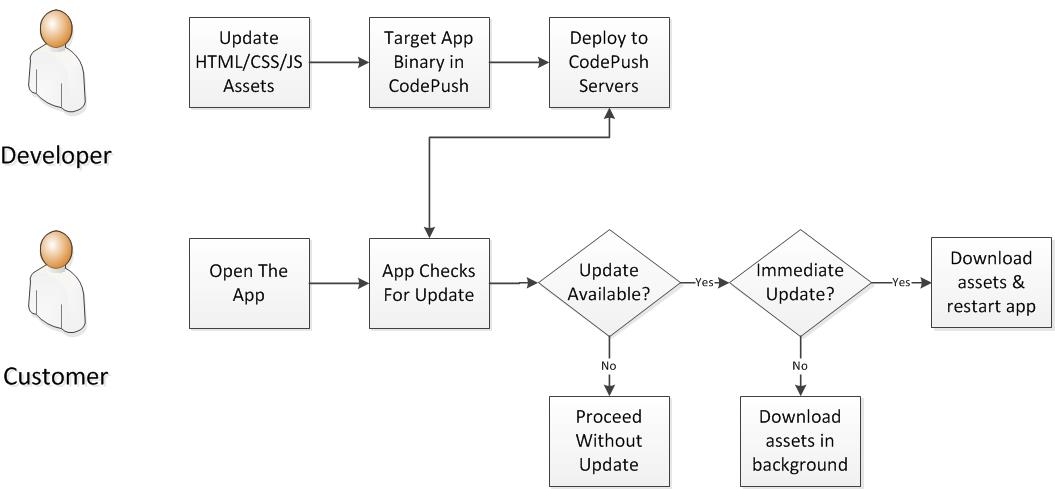


Fig. 12. CodePush process flow. A developer targets a version of the app with an update deployed on CodePush servers. Upon opening the app, a PhoneGap plugin checks for an update. Non-immediate updates download such that changes take effect the next time the app is run on a customer’s machine.

The degree of benefit contributed by these operational efficiencies is relatively small compared with those attributed to front-end development. Based on personal experience in ecosystems in each of the three approaches, I’ve estimated a modest 10% operational cost savings for each of the two web codebases consolidated by using RWD. The result is 2.80 units of work for the operational processes involved in the RWD + Native and Single Codebase approaches for the web versus 3.00 for Brute Force. For mobile apps, I’ve estimated a more robust 25% savings in the operations step for Single Codebase since there is no longer a need to compile, repackage and update the binary or app stores. This results in 1.50 units of work for the Single Codebase approach whereas Brute Force and RWD + Native remain at 2.00 units.

# cost analysis

It should come as little surprise that a single codebase is more cost-effective overall when standing up a brand new digital ecosystem. However, even in a situation where a choice exists to either maintain a legacy system built using one of the other strategies or replace the entire ecosystem with one codebase, the latter still proves superior. Consider a scenario where a full ecosystem replacement takes one year, and is maintained for an additional 5 years before another complete redesign is needed. For maintenance years, an estimate of 25% turnover of the site’s design, content and functionality will be used. This incorporates all ‘keep the lights on’ type of activity such as adding/updating content for products & services, implementing incremental functionality and modifying display logic/rules as the company’s business evolves. These are again highly conservative estimates, particularly for companies with a large, actively maintained digital presence[18]. For the analysis that follows, only front-end development costs will be tracked since this has been established as the main source of variance between the approaches. As was discussed previously, the Single Codebase strategy also provides cost efficiencies in operational deployments and updates but these will be omitted from the analysis for simplicity. The cost of capital will be assumed as 10%, which is meant to represent a well-established company still in a growth phase.



Fig. 13. Costs incurred at the end of each year for the three strategies. Single Codebase includes a full redesign in Year 1. Numbers are in millions.



Fig. 14. NPV analysis of the three development strategies. The values in the % Savings columns indicates the percentage of savings for Single Codebase over that approach. For example, Single Codebase offers a 32.18% savings over Brute Force in Year 1-5.

In Year 1, the single codebase approach engages in a full scale replacement for the existing ecosystem with a cost of $2MM, roughly the average for large sites[19]. Since no redesign takes place using the other approaches, the expense involves maintenance costs only, which, as mentioned, is 25% of a full redesign. However the cost of that 25% is not simply $0.50MM since the Single Codebase approach has been established as more efficient. To calculate the increased efficiency, the Front-End Development row from Fig. 10 is used. For example, the Single Codebase number (2.20) divided by the Brute Force value (5.00) yields .44, which can be understood as a dollar of value for Single Codebase being equivalent to 44 cents of value for Brute Force development. With this relative inefficiency, the $0.50MM Single Codebase cost becomes 0.50/0.44 = $1.136MM. A similar calculation using 55 cents on the dollar for the RWD + Native approach yields 0.50/0.55 = $.909MM. In Year 2, these costs remain constant whereas the Single Codebase strategy drops to $0.50MM as it too is now in maintenance mode. As can be seen in Fig. 14, the net present value (NPV) of Single Codebase is still less than either of the other two over the course of 5 years, despite the full redesign. The net impact is that companies can enjoy a cost savings plus the benefit of a refreshed ecosystem by going with Single Codebase. When the next redesign cycle is incorporated at year 6, this time for all approaches, the cost savings become even more prominent, with a nearly 70% NPV cost improvement for Single Codebase over Brute Force and almost 35% against RWD + Native.

# Performance

Cost is certainly a significant consideration for businesses when selecting a digital development approach, but there are circumstances where the cost-advantage of Single Codebase can be outweighed by other factors. Principal among these is performance vis a vis the user experience when customers interact with the mobile app. Native apps are compiled into bytecode which can therefore run with direct access to components of the hardware that accelerate performance[20]. Because web-app bridges utilize an embedded browser executing within an app container (specifically an instance of the WebView class on Android[21] and the WKWebView class on iOS[22]), they are unable to directly access those components. Fig. 15 demonstrates this difference visually for Android. Native apps are executed directly from the Dalvik Virtual Machine, which sits very close to the processing area in the Linux Kernel. Hybrid apps primarily make use of the Browser application, which sits at the top of the stack and therefore must go through a series of “hops” to eventually get to the same processing area at the bottom.

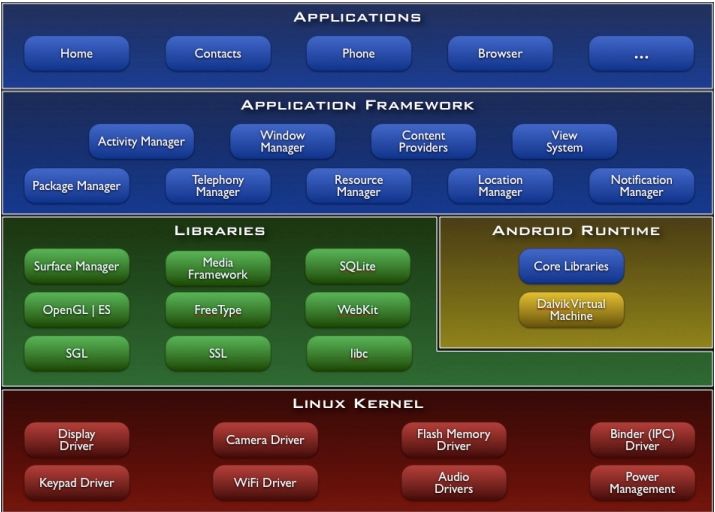


Fig. 15. Android component architecture diagram. Native apps run off the Dalvik Virtual Machine which is lower in the stack compared with the Browser, where hybrid apps run.

Adding further to performance concerns, the Webview had in years’ past been instilled with inferior JavaScript and rendering engines, even compared with the standalone web browser on each platform (Chrome on Android and Safari on iOS). These concerns led to hybrid apps being pushed into a niche role for only simple use cases, punctuated by a much-publicized move from Facebook into native app development from hybrid in 2012[23]. However, since this time the Webview on both mobile O/S’s has undergone significant refactoring and now offers comparable performance to its standalone equivalents[24].

Improvements to the web-app bridging frameworks themselves and novel coding techniques have also continued to shave hybrid app (and standard web browser) execution time. For instance, changing simpler animations from a JavaScript to a CSS-based approach allows the browser to utilize a degree of hardware acceleration (associated with the lower-level O/S component called WebKit), separate from the JavaScript parsing engine. Additionally, combining HTML5-based mechanisms for caching and Service Workers allows data and content to be pre-fetched and used in an offline mode in a separate execution thread from where user interactions are processed. Finally, adjusting for the browser’s slower responsiveness to taps is a key way to overcome a common user experience gap compared with native apps. To accommodate the possibility of double-tapping to zoom in on the screen, mobile browsers pause for 300ms to listen for a second tap prior to fulfilling the original tap. This delay reduces the perceived speed of the experience, but can be overridden such that the pause is eliminated and the experience behaves like a tap within a native app. These and a host of other similar subtle actions can be easily incorporated by web developers through open-source libraries or a few lines of hand-written code.

Despite considerable closure of the performance gap made by hybrid apps in recent years, the upper bound for even the best in class will always be slightly below what is possible with a native implementation. This is a fundamental principal since hybrid apps by definition cannot directly access the hardware – the Webview is always in the middle. For this reason, apps that demand the ultimate in performance, such as graphic-intensive multiplayer games, should opt for a native approach despite the opportunity for cost reduction. Another use case for native apps is if the company relies on technological leadership as a point of differentiation in the marketplace. Web-app bridges necessarily lag behind when it comes to offering apps an opportunity to utilize the latest features introduced by an O/S manufacturer. Either the core team responsible for the web-app bridge or the community that writes third-party plugins for that framework must build, test and release the translation mechanism from JavaScript to the native language. Along similar lines, O/S manufacturers will often alter the look and feel of native UI components such as the navigation tray or spinner icon, with a given release. Natively-built apps that utilize these objects will automatically render using the latest update whereas hybrid apps that attempt to mimic that look and feel with CSS will need to be manually updated to keep current.



Fig. 16. Example of the evolution of an Android UI component. A native app that makes use of a core Android list element will automatically incorporate any changes to the visual structure made in subsequent versions of the O/S. Hybrid app developers would need to manually make these updates using HTML, CSS and JavaScript.

# conclusion and future technologies

Considering the extraordinarily wide breadth of apps available for download (2MM+ for Android & iOS[25]), scenarios where hybrid apps would fail to meet consumer expectations are becoming increasingly rare. This is reminiscent of the gradual replacement of Adobe Flash-based experiences with HTML5 on the web. As browsers became more feature-rich and performant, the benefits of using a closed, proprietary system to build experiences were outweighed by the open nature of the web and its universal use of HTML, CSS and JavaScript. Once again, as the open source community continued to develop on top of the web technology stack, it has become possible to replace the closed architectures of the native app ecosystem without adversely impacting the customer experience.

Looking to the future, other alternatives are beginning to emerge that seek to retain the spirit of the single codebase while overcoming the aforementioned performance and UI component concerns of executing in a Webview. React Native is a framework developed by Facebook that allows developers to use JavaScript to create apps, just like PhoneGap. The crucial difference is that React Native apps do not run through the Webview but rather in the same runtime as truly native apps. In fact, these would not be considered hybrid apps at all. Developers must compile their code using standard native tools such as iOS’s XCode and Android Studio, but the React Native APIs allow for a translation of JavaScript into the native languages. The end result is an app that has access to the full breadth of hardware acceleration and to the core UI components of native apps. The aim of this approach is to do for native app development what RWD has done for web development. Developers write a single app codebase and are able to deploy truly native apps to iOS and Android. It is important to note that JavaScript code written with React Native cannot be used to render web pages on a browser, so this approach adds one additional codebase on top of that which is being used for the web. But the cost of this additional codebase can be considered less than if development was done in the default O/S language (Objective-C/Swift or Java) because the same JavaScript skillset can be leveraged. React Native has not yet reached version 1 at the time of writing so it should still be considered an immature framework, but it has shown tremendous promise, as evidenced by the large community that contributes to its development. There is even an effort, deemed react-native-web on GitHub[26], to extend this such that it can in fact be harnessed for the web as well. Were that to succeed, it would become possible to have one codebase that both serves up a responsively designed website and allows for compilation into truly native apps across platforms.

The landscape of front-end open-source frameworks is constantly changing but the cost-benefit of using JavaScript universally in the development stack of a single codebase for web and hybrid apps is a common theme. By leveraging these advances, companies can enjoy substantial cost savings over more traditional approaches while providing the experience that customers demand across all digital touch points.

References

1. Emarketer.com, ‘Mobile Continues to Steal Share of US Adults' Daily Time Spent with Media’, 2014. [Online]. Available: http://www.emarketer.com/Article/Mobile-Continues-Steal-Share-of-US-Adults-Daily-Time-Spent-with-Media/1010782 [Accessed: 06- Jun- 2016]
2. IBM Corporation, ‘Meeting the expectations of the mobile customer’, 2013. [Online]. Available: http://cdn2.hubspot.net/hub/190519/file-1075770486-pdf/docs/11\_-\_whitepaper\_-\_meeting\_the\_expectations\_of\_the\_mobile\_customer.pdf [Accessed: 27-Jun-2016]
3. Apple.com, ‘App Store Review Guidelines’, 2014. [Online]. Available: https://developer.apple.com/app-store/review/guidelines/#user-interface [Accessed: 06- Jun- 2016].
4. S. Champeon, N. Finck, ‘Inclusive Web Design For the Future with Progressive Enhancement’, South by Southwest®, 2003
5. Opensignal.com, ‘Android Fragmentation Report August 2015’, 2015. [Online]. Available: http://opensignal.com/reports/2015/08/android-fragmentation/ [Accessed: 09-Jun-2016]
6. Opensignal.com, ‘Android Fragmentation Report August 2015’, 2015. [Online]. Available: http://opensignal.com/reports/2015/08/android-fragmentation/ [Accessed: 09-Jun-2016]
7. Opensignal.com, ‘Android Fragmentation Report August 2015’, 2015. [Online]. Available: http://opensignal.com/reports/2015/08/android-fragmentation/ [Accessed: 09-Jun-2016]
8. Alistapart.com, ‘Responsive Web Design’, 2010, [Online]. Available: http://alistapart.com/article/responsive-web-design [Accessed: 09-Jun-2016]
9. W3.org, ‘Media Queries Publication History’, [Online]. Available: https://www.w3.org/standards/history/css3-mediaqueries [Accessed: 09-Jun-2016]
10. Wikipedia.org, ‘XMLHttpRequest’, 2016. [Online]. Available: https://en.wikipedia.org/wiki/XMLHttpRequest [Accessed: 12-Jun-2016]
11. W3C, ‘Battery Status API’, 2016. [Online]. Available: https://w3c.github.io/battery/ [Accessed: 13-Jun-2016]
12. Apple.com, ‘App Store Review Guidelines’, 2016. [Online]. Available: https://developer.apple.com/app-store/review/guidelines/#minimum-functionality [Accessed: 15-Jun-2016]
13. Wikipedia.org, ‘Apache Cordova’, 2016. [Online]. Available: https://en.wikipedia.org/wiki/Apache\_Cordova [Accessed: 15-Jun-2016]
14. Citytechinc.com, ‘Build Your Case For Responsive Web Design’, 2014. [Online] Available: http://www.citytechinc.com/content/dam/citytechinc/pdf/ICF%20Interactive\_Build%20Your%20Case%20For%20Responsive%20Web%20Design.pdf [Accessed: 24-Jun-2016]
15. Github.com, ‘Apache Cordova Plugin camera’, 2016. [Online] Available: https://github.com/apache/cordova-plugin-camera [Accessed: 24-Jun-2016]
16. T.W. Malone, R. Laubacher, T. Johns, ‘The Big Idea: The Age of Hyperspecialization’, Harvard Business Review, 2011.
17. N. Shah, ‘5 Things You Should Know Before Developing A Hybrid App’, 2015. [Online]. Available: http://arkenea.com/blog/develop-hybrid-app/ [Accessed: 24-Jun-2016]
18. Goldploy.com, ‘What is the right time for redesign’, 2013. [Online]. Available: http://goldploy.com/design/web-design-design/what-is-the-right-time-for-redesign/ [Accessed: 17-Jun-2016]
19. Cloudcraze.com, ‘2016 State of Digital Commerce Report’, 2016. [Online]. Available: http://www.cloudcraze.com/wp-content/uploads/2016/02/Cloudcraze\_State\_of\_Digital\_Commerce\_Report.pdf [Accessed: 17-Jun-2016]
20. Tutorialspoint.com, ‘Android Architecture’, 2016. [Online]. Available: http://www.tutorialspoint.com/android/android\_architecture.htm [Accessed: 26-Jun-2016]
21. Android.com, ‘WebView’, 2016. [Online]. Available: https://developer.android.com/reference/android/webkit/WebView.html [Accessed: 18-Jun-2016]
22. Apple.com, ‘WKWebView Class Reference’, 2016. [Online]. Available: https://developer.apple.com/library/ios/documentation/WebKit/Reference/WKWebView\_Ref/ [Accessed: 18-Jun-2016]
23. TechCrunch.com, ‘Mark Zuckerberg: Our Biggest Mistake Was Betting Too Much On HTML5’, 2016. [Online]. Available: https://techcrunch.com/2012/09/11/mark-zuckerberg-our-biggest-mistake-with-mobile-was-betting-too-much-on-html5/ [Accessed: 18-Jun-2016]
24. R. Khanna, ‘Evolution of Hybrid App’s WebView in iOS & Android & its Performance’, 2014. [Online]. Available: https://www.airpair.com/android-webview/posts/evolution-of-webview-ios-and-android [Accessed: 18-Jun-2016]
25. Statista.com, ‘Apple App Store: number of available apps 2016’, 2016. [Online]. Available: http://www.statista.com/statistics/263795/number-of-available-apps-in-the-apple-app-store/ [Accessed: 27-Jun-2016]
26. N. Gallagher, ‘React Native for Web: A framework for building Native Web Apps’, 2016. [Online]. Available: https://github.com/necolas/react-native-web [Accessed: 17-Jun-2016]
27. Zurb.com, ‘Foundation for Sites’, 2016. [Online]. Available: <http://foundation.zurb.com/sites/docs/> [Accessed: 10-Jun-2016]
28. Getbootstrap.com, ‘Bootstrap - The world's most popular mobile-first and responsive front-end framework.’, 2016. [Online]. Available: <http://getbootstrap.com> [Accessed: 10-Jun-2016]
29. Angularjs.org, ‘AngularJS — Superheroic JavaScript MVW Framework’, 2016. [Online]. Available: <https://angularjs.org/> [Accessed: 12-Jun-2016]
30. Emberjs.com, ‘Ember.js - A framework for creating ambitious web applications.’, 2016. [Online]. Available: <http://emberjs.com/> [Accessed: 12-Jun-2016]
31. Mozilla Development Network, ‘Battery Manager – Web APIs’, 2015. [Online]. Available: <https://developer.mozilla.org/en-US/docs/Web/API/BatteryManager> [Accessed: 13-Jun-2016]
32. W3c.org, ‘JavaScript Web APIs’, 2016. [Online]. Available: <https://www.w3.org/standards/webdesign/script> [Accessed: 13-Jun-2016]
33. Caniuse.com, ‘Can I use...Support tables for HTML5, CSS3, etc.’, 2016. [Online]. Available: <http://caniuse.com/> [Accessed: 14-Jun-2016]
34. Phonegap.com, ‘PhoneGap Build Documentation’, 2016. [Online]. Available: <http://docs.build.phonegap.com/en_US/introduction_what_is_build.md.html#What%20is%20Build> [Accessed: 14-Jun-2016]
35. Gruntjs.com, ‘Grunt: The JavaScript Task Runner’, 2016. [Online]. Available: <http://gruntjs.com/> [Accessed: 16-Jun-2016]
36. Mochajs.org. ‘Mocha – the fun, simple, flexible JavaScript test framework’, 2016. [Online]. Available: <https://mochajs.org/> [Accessed: 16-Jun-2016]
37. Protractortest.org, ‘Protractor – end to end testing for AngularJS’, 2016. [Online]. Available: <http://www.protractortest.org/#/> [Accessed: 16-Jun-2016]
38. Sass-lang.com, ‘Sass: Syntactically Awesome Style Sheets’, 2016. [Online]. Available: <http://sass-lang.com/> [Accessed: 16-Jun-2016]
39. Microsoft, ‘CodePush’, 2016. [Online]. Available: <https://microsoft.github.io/code-push/> [Accessed: 16-Jun-2016]
40. A. Schmitz, C. Thoburn, J. Tangelder, ‘Hammer.JS’, 2016. [Online]. Available: <http://hammerjs.github.io/> [Accessed: 27-Jun-2016]
41. Facebook, ‘React Native | A framework for building native apps using React’, 2016. [Online]. Available: <https://facebook.github.io/react-native/> [Accessed: 17-Jun-2016]

****

**Mike L. Lipman** was born in Skokie, Illinois, in 1980. He received a B.S. in Computer Science in the College of Engineering from the University of Illinois at Urbana-Champaign in 2002 and an M.B.A in Marketing and Finance at the Kellogg School of Management at Northwestern University in Evanston, Illinois in 2009.

From 2002 to 2007, he was an Application Developer for U.S. Cellular. Since 2007, he has been in the Marketing department, most recently as a Senior Manager of E-Commerce Customer Experience in Chicago, IL. He leads the team responsible for all User Experience, Graphic Design, and Front-end Development for uscellular.com mobile and desktop properties.

1. Submitted on 8/28/2016

   M. L. Lipman is with U.S. Cellular as a Senior Manager of E-Commerce Customer Experience 8410 W. Bryn Mawr Chicago, IL. 60631 (e-mail: michael.lipman@uscellular.com) [↑](#footnote-ref-1)