



The New Mobile Development Landscape

Table of Contents

Introduction / 3

Mobile Development The Early Days / 4

Mobile Development Today / 5

Native Development / 6

Browser Development Evolves / 8

Mobile Application Development Platforms (MADP) / 11

Hybrid Mobile Apps / 14

JavaScript-Driven Native Apps / 18

Other Types of Native Apps / 22

Making The Transition / 23

Testing Your Apps / 24

Building Your Mobile App Back-End / 25

Conclusion / 26

If you poke around on the Internet, you'll find hundreds or even thousands of frameworks and platforms designed to make platform mobile development easy. There's almost too many of them to keep track of, and nobody knows which ones are the best, and which ones are worthless. If you're part of a one to five-person development shop, you can afford to play around at the options to see which works best for you and your apps, potentially even switching approaches between apps.

Larger development shops have to think big; the decisions they make around development approach and platform or framework selection have a long term impact on the cost and effectiveness of their development effort. Changing course months or years later means considerable expense migrating off of a flawed or abandoned platform to the shiny new approach. Development organizations must be smart, making the right choice at the app's creation, knowing that the approach they've selected works for the long haul.

We've created this ebook to help you make smart choices about how you build modern mobile apps.

Note: To avoid any hint of favoritism, frameworks, platforms and tools are listed in alphabetical order throughout this publication. We've highlighted a sampling of options, not even trying to present a complete representation of the available choices.

Mobile Development, The Early Days

In the old days, which really wasn't that long ago, developers had limited options: build web apps or native apps.

For web apps, developers used:

- Whatever tools they used to code their desktop web apps.
- [Responsive Web Design](#) to build web apps that rendered well on desktops and mobile devices.
- Specialized frameworks like [Kendo UI®](#) and [jQuery Mobile](#) to make mobile web development easier or mobile apps prettier.

For native apps, developers had limited options; they could:

- Code mobile apps using the shallow list of languages supported by the target mobile platform (Java for Android, Objective C for iOS, and C#, Visual Basic, and JavaScript for Windows Mobile).
- Use the development and debugging tools provided by the mobile platform vendors.

That was pretty much it.

For devices, the mobile space was much more fragmented, offering users a wide variety of devices to choose from: Android, BlackBerry, Firefox OS, iOS, Symbian, WebOS, Windows, and probably a few others.

Testing mobile apps was expensive and time-consuming. To account for the variety of devices and form factors, Quality Assurance (QA) departments needed to test applications on each manufacturer-provided device emulator or simulator, plus any physical devices lying around the office. Development organizations bought at least one of each popular device, and used the automated testing frameworks provided by the device platform vendor. It wasn't long before third-party and open-source testing solutions appeared; cross-platform test frameworks and device lab solutions that made mobile app testing better.

Before long, organizations started looking for economical ways to deliver apps for multiple platforms simultaneously (from the same code-base). That led to the commercial development of Mobile Enterprise Application Platforms (MEAP), Mobile Consumer Application Platforms (MCAP), and finally, a consolidation of both into Mobile Application Development Platforms (MADP). The community responded as well, delivering cross-platform tools like the popular Apache Cordova framework, Appcelerator Titanium, and many others.

Mobile Development Today

Today's developer has many options for building mobile apps. Native and web are still viable options, but other variants exist as well. Users also have a lot less choices, meaning less platforms for developers to worry about as the market whittled down to only two choices: Android and iOS. In this section, we'll outline the options and explain how they stack up for developers.

Native Development

Native apps are still the best apps you can deliver to mobile devices and mobile device users. Unfortunately, native development is still pretty hard. For native apps, APIs exposed by the device's system software enable these apps to leverage all the capabilities the mobile device offers - with no compromises, so there's a lot of good reasons for building native apps. There are a lot of good arguments for options beyond native as well, so don't operate under the premise that all mobile apps must be native apps.

What you'll see in later sections of this publication is that many of the mobile development options available today enable you to use skills developers have with other languages (such as JavaScript and its variants, or C#), or allow developers to build an app that runs well on multiple platforms (which today essentially means Android and iOS plus possibly Windows). That's not possible using the native platform tools and languages. You could build a development team that has experience with both Android and iOS development (Java or Kotlin and Objective C or Swift), but that's just not likely. The native platforms are so complicated, the suite of APIs and UI components so different, that you're much better off building separate teams for each target platform. That means dealing with two different toolsets, languages, build processes, and more. Mobile app UI controls and themes vary as well across platforms, so you'll likely need to design two similar, but different, UIs for your app. Not only is this cumbersome, it's expensive as well.

From a tooling standpoint, native developers primarily use the free development tools provided by the platform vendors. This means Android Studio for Android developers and Xcode for iOS developers.

There are quite a few third-party Integrated Development Environments (IDE) for Android and iOS development. Many on the Android side only handle a subset of the requirements for Android developers. On the iOS side, JetBrains' [AppCode](#) is a popular, capable IDE for iOS development (macOS too).

Language options for native mobile development have expanded a bit. Where Android developers primarily used Java for their apps, they could also delve into C and C++ when they needed higher performance or needed to code closer to the hardware. Google recently embraced Kotlin as an alternate language option for native Android apps as well. The Kotlin team claims Kotlin is 100% interoperable with Java, so you can call Kotlin code from Java and Java code from Kotlin.

For iOS and macOS developers, Apple released the Swift programming language in 2014. Swift is a modern programming language (unlike Objective C which is much older than Java and even JavaScript) with some pretty cool features. For new developers, Swift offers an easier learning curve than Java or Objective C, making it easier for developers to come up to speed. The good news is that these language enhancements don't create much of an issue for existing developers, it's easy to switch between languages (in separate source files) within the same app.

As more third-party libraries became available for iOS apps, managing dependencies became a problem, so the market responded with the introduction of several dependency managers like [CocoaPods](#) and [Carthage](#).

Reasons to Build Native Apps

- Native apps, when coded correctly by an experienced developer, deliver the fastest apps.
- Native apps can do the most things on any mobile device (access to any native API).
- Native apps are the least risky path to users through native platform's app stores.

Reasons Not to Build Native Apps

- Native apps are expensive to build and maintain.
- Native apps require skills unique to the target platform.
- Experienced native developers more expensive than web developers.



Swift Beyond iOS and macOS When Apple released the Swift language, many saw the language as a way to enable development of more than iOS and macOS apps. Swift has some cool features and Apple's full backing, so many in the industry started looking for more places to use it. Teams quickly started thinking about writing cross-platform development, building [Android apps in Swift](#). IBM even enabled Swift support in their Bluemix cloud offering. Due to Swift's popularity, and the amount of skilled resources in the market, look to see the language popping up in unexpected places. Not much happened, Swift hasn't yet earned a good reputation outside of iOS and macOS apps.

Browser Development Evolves

In the early days of mobile, the mobile browser worked; it was able to render web pages and web apps on mobile devices, but the limited screen real estate, cumbersome touch interactions, and the reduced capabilities of the device (compared to desktops and laptops) affected performance and user experience. Browser users and app developers wanted the browser to be able to leverage the sensors and other capabilities of the mobile device. Apple expected that the browser would deliver app-like experiences for end users, but the market proved they were wrong.

Device manufacturers and the community responded, delivering better performance and libraries that extended web apps more easily into the mobile space. The browser got access to the camera, sensors, and other aspects of the device, and rendering engine performance increased. The community delivered mobile-friendly frameworks that enabled developers to more reliably recognize gestures, exposed finger-friendly UI elements on mobile devices, and reduced the work developers had to do to deliver web apps for desktop and mobile simultaneously.

Reactive Frameworks

In the web applications of old, developers wrote a lot of code to connect one or more data sources to the app's UI. This code had to deal with creating the data model, reading and writing to storage, and updating the UI whenever the data changed. Now we have reactive frameworks and web application frameworks (HTML, JavaScript, and CSS) that connect an app's UI with its data.

In 2013, Facebook open sourced the [React framework](#), a framework Facebook used to build its own web applications. Developers use React to create web applications coded primarily in JavaScript. The web app's index.html file bootstrap's the React framework, and from that point on, the app's JavaScript code handles creating and maintaining the app's UI.

React delivers a syntax extension to JavaScript called [JSX](#) that enables developers to craft an app's HTML UI elements in JavaScript (weird, right?). This approach gives developers a unique way to bundle their UI components' HTML, styling and interaction logic all in JavaScript. A React application's JavaScript code doesn't look like regular JavaScript code as JSX enabled HTML elements to be embedded in the JavaScript code without any special formatting or delimiters.

React and the other frameworks discussed in this section deliver JavaScript-driven web applications. This approach is becoming the norm for modern web development. If you're not learning them now, you're already behind the curve.

Facebook also produces React Native, the same React framework running on mobile devices in a native application shell. React Native applications use reactive development principles, but with native UI elements instead of HTML. You'll learn more about React Native in a later section of this publication.

Another popular reactive framework is [Vue.js](#). This community-driven framework describes itself as "An incrementally adoptable ecosystem that

scales between a library and a full-featured framework”, which means you can add Vue.js to a part of your app and expand as needed from there.

The framework is unique in that it was created by a single person (Evan You) and community financial sponsorship enables him to work full time on the framework; as long as the framework remains popular, Evan has a full-time job.

An alternate option for JavaScript-driven web development like React and Vue.js is SAP’s [openUI5 framework](#). openUI5 is the open-source version of SAP’s UI5 framework SAP uses to deliver the UI for many of the enterprise applications they deliver to customers.

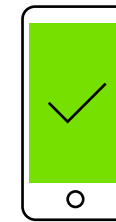
Each of these frameworks enables you to:

- Write your web applications using primarily JavaScript with little HTML.
- Create reusable UI components you’ll use to assemble your app’s UI.
- Easily bundle your app’s UI elements with its JavaScript code, and share them across applications.

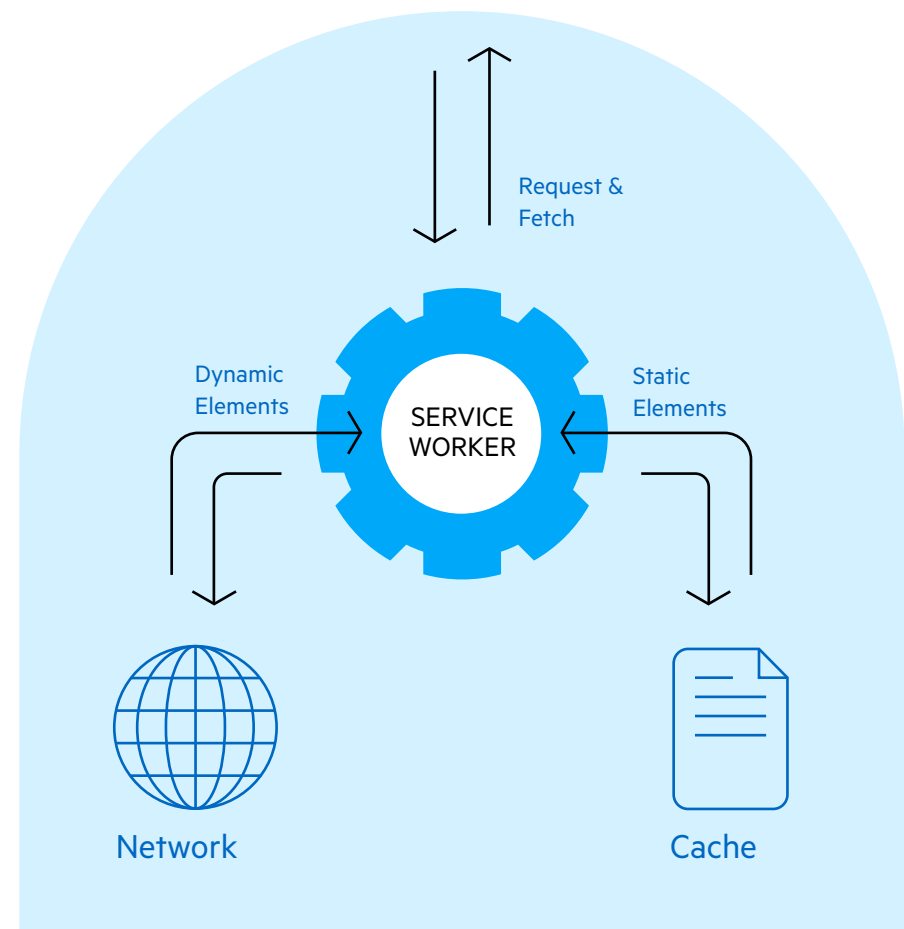
Progressive Web Applications (PWA)

We already know that with UI frameworks and reactive apps, mobile apps can look and feel like native mobile apps. One last piece is needed to make the mobile browser a first class citizen for mobile apps, developers need a way to make web applications act more like native apps. To be fully effective on mobile devices, web apps should:

- Install like a native application on the device’s home screen.
- Do things in the background (like synchronize data with back-end servers).
- Be capable of receiving and processing push notifications.



PWA App Shell



You could always add a web URL as a shortcut on your desktop, a feature that was available in early versions of Android and iOS. The [web manifest](#) standard made it easy for a developer to define parameters for how the home screen icon is added (for example, the image file and title to use for the home screen icon, and whether to render the browser chrome for the app).

The [Service Worker API](#) enables web applications to register a chunk of JavaScript code that runs in the background, enabling synchronization with back-end data sources, processing push notifications, and even enabling an offline mode for the application by caching resources so they'll be available when there's no network connectivity.

These capabilities bundle together to deliver what Google calls [Progressive Web Apps \(PWA\)](#). You can find a more thorough description in [Progressive Web Apps: Escaping Tabs Without Losing Our Soul](#). Google added support for PWA to the Chrome browser on Android, and Apple only recently added support for a subset of PWA features to the Safari browser on iOS. Microsoft offers full support for PWA in the Edge browser, and they're taking the extra step to scour the Internet looking for PWA and adding them to the Windows App Store (making it easy for Windows users to find and install these apps).

PWAs are a game changer for mobile users, making web apps work like native apps (although with limited access to native APIs). Most popular interactive or data-driven web applications are in the process of, or will be, enhanced to use PWA features very soon. You're still going to need mobile apps to interact most effectively with your users, but PWA gives you a way to deliver a much more robust experience for users who haven't, or won't,

install your mobile app. For some apps, PWA gives developers a way to skip native apps altogether, and deliver all of their app functionality via HTML or JavaScript-driven web development.

Mobile Application Development Platforms (MADP)

The mobile platform space exists because many development organizations (mostly enterprises) needed a way to minimize their costs of building mobile apps for Android and iOS. Most platforms consist of multiple components:

- On-premises or cloud-based server offering that manages authentication, connectivity to back-end data sources, data synchronization, and more.
- Browser-based, or locally installed proprietary development environment; typically a robust drag and drop app designer, often running in the browser.
- (Optional) Runtime container, an execution environment for the meta application created with the platform's proprietary development environment.

Where these platforms were the only way to build cross platform mobile applications, there are now many open-source and commercial cross-platform development offerings that eliminate the need for a formal platform. With all the mobile-optimized cloud service offerings available from Amazon, Google, Microsoft, and others, there's little need for proprietary development tools tightly coupled with proprietary back-end services.

With the wide array of options available to the professional mobile developer, mobile platforms have morphed into tools primarily used by enterprises to enable non-developers (what's commonly known as a Citizen Developer) to build mobile apps. The drag and drop nature of MADP development tools coupled with the data integration capabilities of the server process enables non-developers to easily 'draw' app UIs (using drag and drop components),

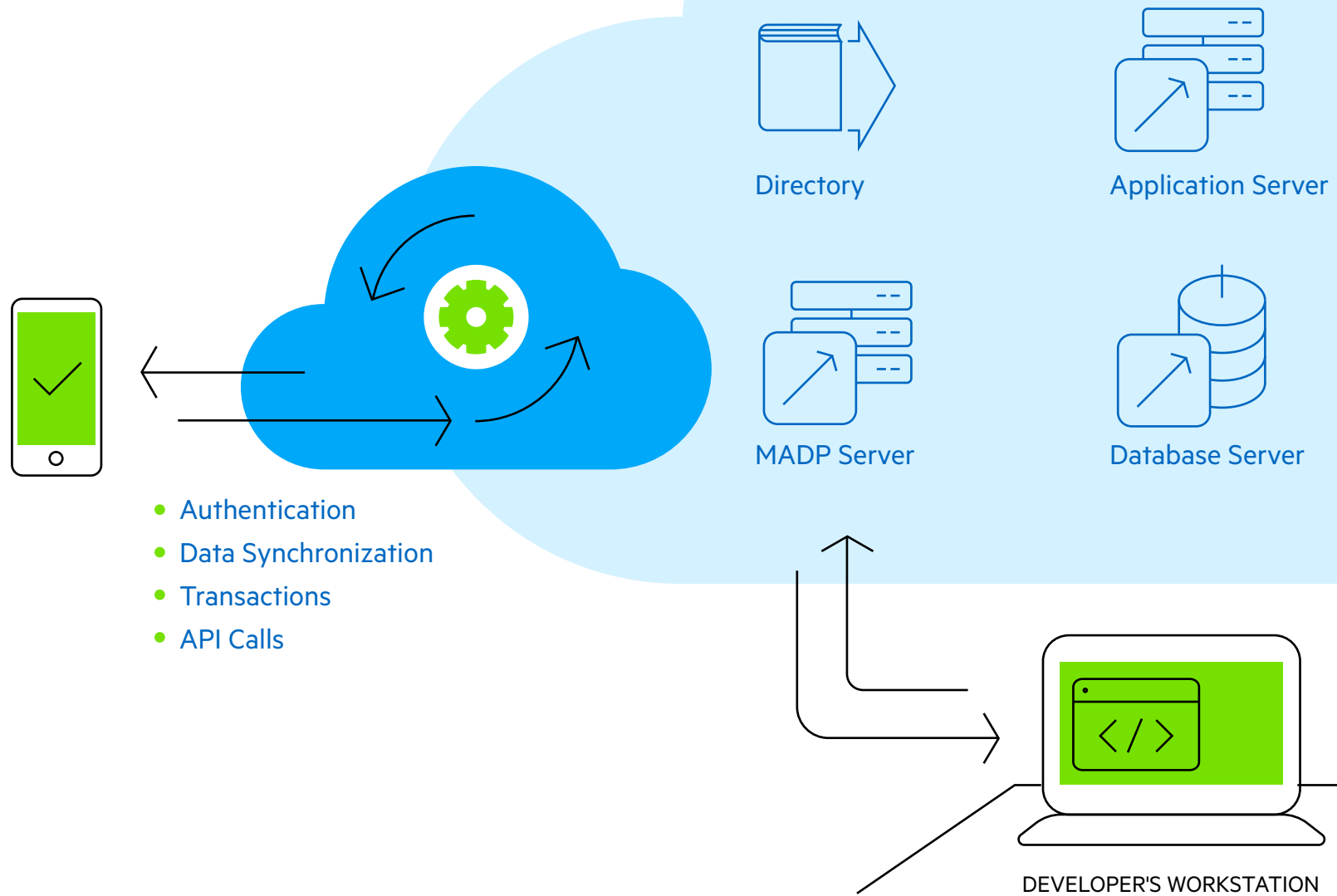
wire UI elements to data, connect screens together for navigation, control access to data based on user credentials, and more.

The typical MADP produces an app delivered in one or more formats:

- Most deliver web applications which run seamlessly on desktop and mobile browsers.
- Few platforms deliver native mobile applications by generating native code for the selected target platforms. These applications are rarely small, fast, tightly-coded native apps, instead they're bloated with all of the code needed to translate the designed app, connections to back-end data sources, authentication, and more, into a native app.
- Most platforms offer the ability to generate a web application that executes in WebView (basically a browser window) inside a native application. The most popular approach uses the Apache Cordova framework to deliver the native app running the generated web application.
- Many platforms generate a meta application (typically an XML or JSON¹ file 'describing' the application) which executes in a proprietary runtime container. The container parses the app's metadata, then generates the application UI at runtime using either proprietary or target platform-specific native UI elements. This approach essentially gives you native applications without writing any native code. Unfortunately, these applications can't deliver true native UIs as the app can only

¹ Pronounced just like the person's name (Jason), not "j-son" with a pause in the middle.

On-premises, or in the Cloud



support the UI capabilities exposed through the platform, and may omit native controls supported by the mobile device.

- A small number of platforms generate code in a single development language (like Java, for example), then deploy the compiled application along with the required language runtime environment as the mobile app. These bloated monstrosities do enable you to write one app and run it on more than one target platform, but should only be considered as a last resort.²

Reasons to Invest in a MADP

- You need to accommodate non-developer types who want to build mobile apps for their business unit or team.
- You want to use a consistent approach for all mobile apps.
- Your apps aren't that complicated, so MADP is enough to get the job done.

Reasons Not to Invest in a MADP

- Vendor lock-in, when the platform vendor disappears, you may no longer be able to use the app.
- Not able to accommodate all of your application requirements due to restrictive capabilities in the platform.
- When app performance matters.

² For example when you run your business on a commercial software package, and the vendor's mobile platform is the only way to get a mobile app that works with the software application.



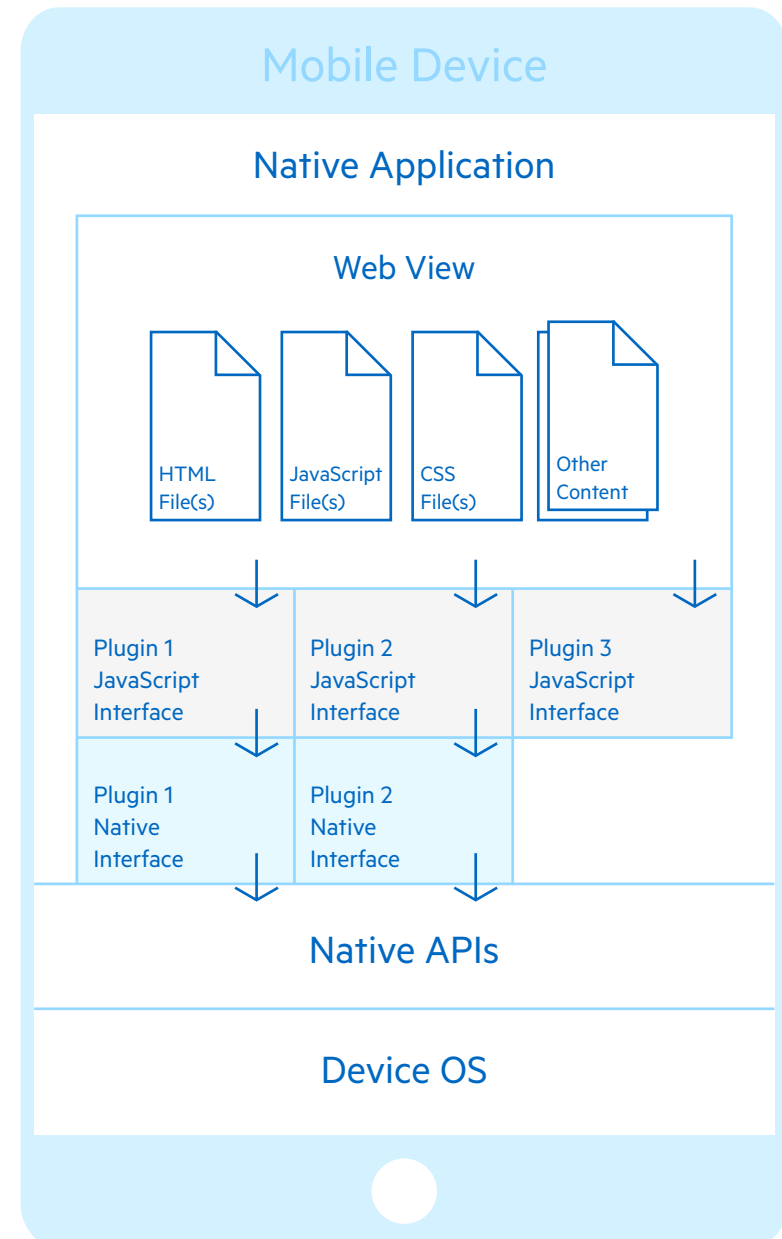
Relying on Apache Cordova As mentioned, the most common way MADP delivers mobile apps is as a web app running inside the Apache Cordova runtime container. These are web apps that provide access to native application capabilities (access to native APIs and other native apps running on the device) through plugins, and a Native/JavaScript bridge provided by the Cordova platform. There are risks here that your application may suffer from performance bottlenecks. This is because hybrid mobile applications are rendered in a web view, similar to the Android and iOS device's web browser. Every device, platform and operating system version handles the processing of web views differently, creating inconsistencies in performance and user experience.

Hybrid Mobile Apps

For a long time, the most common alternative to building native mobile apps was hybrid mobile applications. Hybrid apps are web applications packaged and deployed in a native application runtime container. The application UI renders in a WebView (a native component that essentially exposes a browser view in a native application) and developers code the application's business logic using JavaScript. The first approach to this was created in 2008 as [PhoneGap](#)³ (by a company later purchased by Adobe) and renamed to Cordova when Adobe donated the project to the Apache Foundation as [Apache Cordova](#).

The web applications running in the Cordova container aren't limited to the things web applications can do in the browser. One of the core components of Cordova is a JavaScript to native bridge that enables developers to access native capabilities (like native client-side APIs and hardware such as sensors and the camera) via a web application's JavaScript code. While this isn't a unique capability, Cordova was one of the first implementations of this in a popular framework for mobile apps. To simplify implementation of this, the Cordova team publishes a public plugin specification and a set of core plugins exposing a common set of capabilities. The open plugin specification enables developers to easily build custom plugins for internal use or public consumption, and drove the community to deliver almost 4,000 plugins.

³ PhoneGap still exists today as Adobe's distribution of Apache Cordova. Adobe added some additional capabilities to Apache Cordova for their version of the distribution and uses PhoneGap in several of their commercial products.



Hybrid apps at their core are web apps, so they won't have native application UIs, and native UI performance. This was a problem years ago when Google used a different rendering engine for the WebView component and Apple deliberately limited performance in the iOS WebView. Since then, Google and Apple eliminated those bottlenecks, so life's much better now.

Cordova simply manages the packaging and rendering of web content in a native app, so there are no Cordova UI components to use. To accommodate mobile developers who don't want to roll their own UI libraries, the community responded with several open-source and commercial UI frameworks for hybrid applications. These frameworks (like [Framework 7](#), [Ionic](#), [Kendo UI](#), [Onsen UI](#), and more) give web applications a native look and feel, tricking users into thinking they're using a native app. They'll notice a slight performance decrease, but the web app will act and look like a native app; for many apps, users won't be able to tell the difference.

You may be asking why technologies like [NativeScript®](#), [React Native](#), and [Xamarin](#) aren't mentioned here, that's because they're not hybrid frameworks; they're covered in other sections a little later. These frameworks create native apps with native UIs, there's nothing hybrid about them.

Ionic and Onsen UI are unique in that they both include tooling that enables them to deploy on top of Apache Cordova. They're both supporting Progressive Web Apps as an alternate deployment mechanism, which will become more important later in the publication.

Apache Cordova is the most common approach for building hybrid apps; you'll find many enterprise and consumer apps built with the framework. Most MADP offerings even run Apache Cordova under the covers. Cordova was so good at what it did that it was only recently that competing frameworks appeared⁴.

Predicting the Death of Apache Cordova

When the PhoneGap project launched, the project team stated "the ultimate purpose of PhoneGap is to cease to exist." The team saw themselves fixing a problem that eventually wouldn't exist anymore. The market demanded the ability to easily (and inexpensively) build cross-platform mobile apps and expected that the capabilities exposed through the mobile browser would eventually match the capabilities available to most native apps.

Remember, when Apple launched the iPhone, they expected users would never need anything more than browser apps (except, of course, for the apps produced by Apple). When Apple's customers demanded more, Apple unexpectedly found themselves needing to build an app store and expose the iOS APIs for external developer use. That (late) decision made a lot of money for app developers, but it's definitely not what Apple expected.

⁴ Ionic recently released the Capacitor [framework noted on the GitHub project page as "an eventual alternative to Cordova"](#)

As expected, the mobile browser, over time, began exposing native capabilities to web applications. Soon, developers were able to build web apps that determined the device's orientation using the internal accelerometer, determined location using the device's internal GPS radio, even interacted with the device camera. The (now) Apache Cordova team's prediction no longer seemed so far fetched.

Fast-forward to today, and the availability of PWA; with this capability, and other enhancements added to the mobile browser, one of the main use cases for Cordova apps no longer exists. In December 2017, the Cordova team deprecated many of the core Cordova plugins since the native WebView used by all Cordova apps now supported the capabilities exposed by the plugins. Additionally, Apple added support for (a subset of the capabilities of) PWA in the Safari browser in iOS 11.3.⁵

In the early days, the Apache Cordova project had the support of many well known companies (Adobe plus Google, IBM, Intel, Microsoft, Mozilla, Ubuntu, pretty much everybody in the mobile hardware space except for Apple). The big guys (Adobe, Google, IBM, and even Microsoft) had dedicated teams working on the project, anywhere from 6 to 10 people at a time. Google did a lot of work to ensure the Apache project's success on Android. IBM (plus SAP, Oracle, and many others) built mobile development platforms around Apache Cordova. Microsoft built a capable suite of tools for Cordova developers in the Tools for Apache Cordova (TACO) add-in for Visual Studio. Intel even built a robust IDE for Cordova development called XDK.

Over time, corporate involvement in the project waned. Google, IBM, and Microsoft reduced the number of resources dedicated to Apache Cordova. Intel disbanded their team and shutdown the XDK project. Apache Cordova continues to be supported primarily by Adobe, with help from the open source community.

Apache Cordova is still a popular, seasoned, reliable framework for enabling web developers to write mobile apps. The existing UI frameworks enable developers to build hybrid apps that look and act like native apps, so for many enterprise and consumer mobile apps, Cordova is a great option to use. As you'll see in subsequent sections, the mobile development world is migrating to other, more robust approaches. An existing investment in hybrid apps is still worthwhile, but know that you should already be retooling for the new mobile development world we live in. Read on to learn more. We do not recommend starting new projects or using mobile platforms or tooling with a Cordova framework dependency.

Ionic Saves Hybrid

With all that's been said so far about the Cordova project's impending doom, you're probably surprised to see the heading for this section. The Ionic framework is a very capable framework for building hybrid applications. Originally built on top of Angular, the framework offers robust command-line tools, flexible UI components (HTML-based), and is easy to learn.

5 [Progressive Web Apps on iOS are here - Medium.com Article by Maximiliano Firtman](#)

The Ionic team kept their eyes on the market and noticed that mobile development was trending toward web standards. To accommodate this, Ionic started moving away from Angular, migrating their existing UI elements to web components instead. This change enables developers to use the Ionic UI components in their web and hybrid applications, extending their reach. In early 2018, the Ionic team announced a "built from the ground up" hybrid mobile app container called Capacitor. Ionic has described the project as an "[eventual alternative to Cordova](#)" but as of the date of this publication, has not released the 1.0 version. Ionic is betting on web development standards, and hedging their bet by bringing to market a modern take on the hybrid approach.

Reasons to Build Hybrid Mobile Apps

- Hybrid offers a quick and easy way to deploy a web app as a native app.
- Hybrid offers the easiest way for web applications to access native APIs.
- Hybrid delivers app store distribution for web applications.

Reasons Not to Build Hybrid Mobile Apps

- PWA technology can do most of what Hybrid Mobile technology can do.
- JavaScript-driven native apps deliver more of what mobile developers want, building native mobile applications using web technologies.

JavaScript-Driven Native Apps

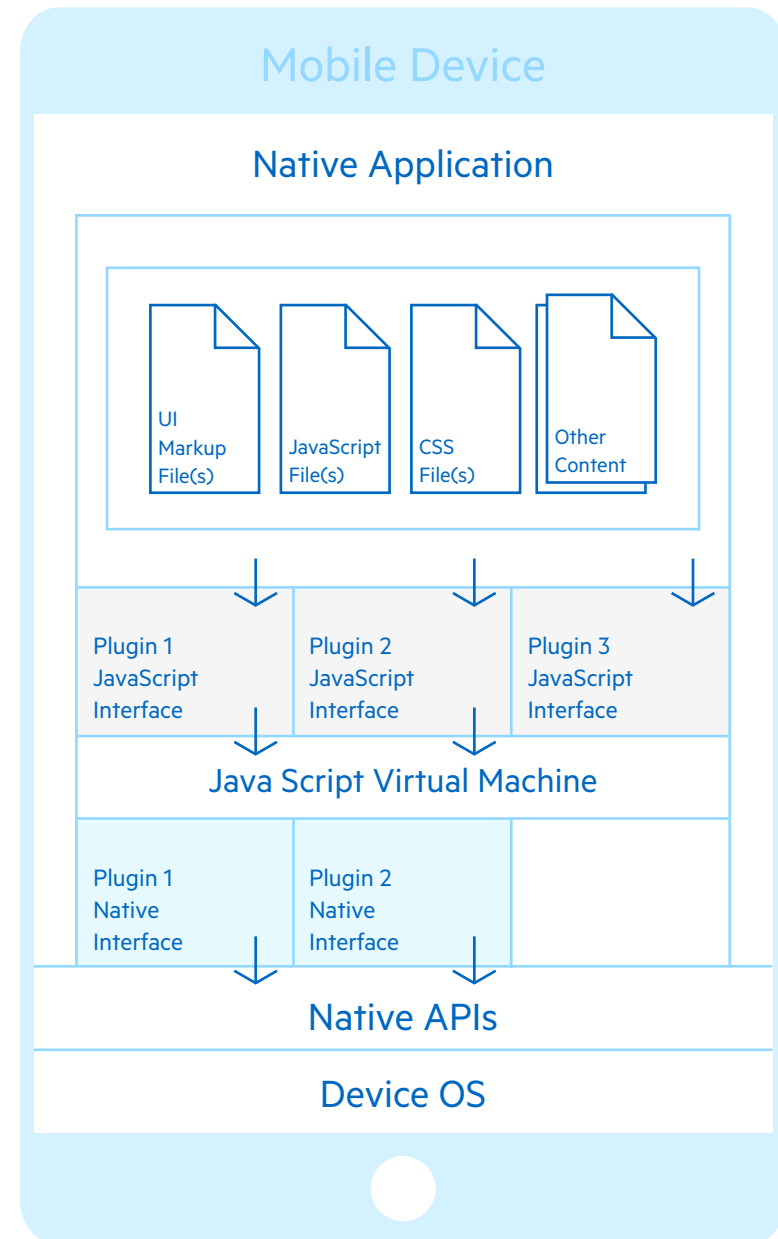
If you've gotten this far, you know that ultimately you should deliver native apps to your mobile app users. Native apps deliver the best user interfaces, and the best user experience (performance, the most capabilities, the most modern features of the target device) when written by experienced mobile developers⁶. With everything that's been said so far about the difficulty and cost of building native applications, how do you cost effectively build native applications for multiple target platforms? JavaScript-driven Native (JSN) apps!

JavaScript-driven native apps are native mobile applications coded using web technologies, primarily JavaScript (or its variants - more on that in a bit; we're just going to refer to those languages as JavaScript for now). With JSN apps, an app's UI and business logic are coded in JavaScript, but the app renders a native application UI. The UI may have some HTML in it, but that's not the primary source for its UI.

At runtime, here's what happens:

1. The native application launches and completes its normal startup operations.
2. The app passes control to the application's JavaScript code which executes in a JavaScript runtime process within the app.

⁶ Anyone can download the native development tools and build an app for Android or iOS; that doesn't mean the app will be any good. Mobile development is complicated, and it takes a while to get the skills you need to deliver a sophisticated app that meets modern user expectations.



3. The JavaScript code in the app uses the app's JavaScript/Native bridge (just like what you'd find in an Apache Cordova app) to create native UI components which are rendered by the native part of the application.
4. The app's JavaScript code continues to run, processing user input, manipulating screens and data, until the user closes the app.

Since JSN apps are native apps at their core, there's no issue deploying apps through public or enterprise app stores. These apps don't suffer from the performance issues that sometimes plague web or hybrid applications, so even savvy users won't likely know it's JavaScript code running under the covers.

There are really only two leading JSN options popular today:

- [NativeScript](#)
- [React Native](#)

We'll talk more about each in the following sections. Other interesting, but less popular, options include:

- [Appcelerator Titanium](#): the original JSN offering; a lot of apps were built using the platform over the years, but not much is heard about it today.
- [Tabris.js](#): has a small following, but is a true JSN option.



The JSN approach was popularized by the open-source (and commercial offering) Appcelerator Titanium way back in 2009. Appcelerator struggled to build a large user base and a successful profit model before finally selling themselves to Axway in 2016. Appcelerator was advanced for the time, eventually expanding to include a full back-end platform as well, but they struggled to get the cross-platform part of JSN simplified until it was too late.

JavaScript-driven native apps are different, so there's a learning curve for existing web developers, but that's not a reason not to write them. Building JSN apps requires that you have access to skilled JavaScript developers; but with the current state of web development you probably already have those. You'll still need a few native developers around, to help design app interfaces using native controls and dealing with any internal APIs the JSN app uses. The frameworks and tooling are all free, so there's no capital investment to make. This explains why JSN is such a popular approach to building native mobile applications.

Current JavaScript developers should be comfortable writing JSN apps, as they use the modern web development approach that's popular with many frameworks in use today. These apps are JavaScript-focused rather than HTML-focused and use ECMAScript modules to compartmentalize code.

From a tooling standpoint, JSN developers won't spend much time in the native platform's IDEs; instead, they'll use the regular, everyday web development tools they use today. The most popular editor for JSN apps is [Visual Studio Code](#); many developers use the [JetBrains WebStorm](#) IDE, and developers who just like to work in text editors use editors like [Sublime Text](#).

For compilation and other project management tasks, the popular approaches drive developers to command-line tools to interact with the framework and platform (native mobile platform SDK) tooling.

NativeScript

[NativeScript](#) is an open-source framework for building cross platform mobile applications for the Android and iOS platforms. The framework was originally

created by Telerik, and acquired by Progress in late 2014. NativeScript is an open-source project with the full support of the Telerik/Progress team.

NativeScript isn't a language, it's a way to write native mobile applications using JavaScript (and JavaScript-like languages like TypeScript). NativeScript applications are native applications with native UI elements. NativeScript is unique in that it allows developers to leverage [Angular](#) and [Vue.js](#) in their native applications, leveraging the reactive UI capabilities of those frameworks within native apps.

NativeScript apps don't leverage a JavaScript to Native bridge to access native capabilities in a mobile app; instead, the framework is structured in such a way as to enable direct execution of native APIs from JavaScript code.

The NativeScript framework ships with a large set of cross-platform abstractions like Button, Camera, ListView and others to allow developers to create both native iOS and Android versions from a single code base.

The framework found initial popularity with enterprise developers, partially because of its less restrictive license agreement than React Native, although the React project has since adopted a more friendly license. NativeScript is perceived by the community to be a more open project; Progress works for the betterment of the community and gets a return on its investment through NativeScript developers using other products and services from the company.

To make it easy for developers to code (and debug) NativeScript apps, the NativeScript community released the [NativeScript extension for Visual Studio Code](#). This free extension simplifies NativeScript development and delivers interactive debugging of NativeScript apps on devices, device emulators and simulators. Also, there are thousands of free and open-source plugins and app templates available for use by NativeScript developers.

React Native

[React Native](#) is an offshoot of Facebook's React project; both [React](#) and React Native are open-source projects that enable developers to code reactive applications. React Native is the native mobile app implementation of React where JSX, or plain JavaScript, is used to render native UI elements in an app.

The React team has stumbled a bit with React and React Native:

- While both are open-source projects, Facebook has a somewhat myopic view of the projects; they created them to help Facebook build better apps, not because they wanted to do something good for the community. They support the community, but the project exists simply to make it easier for Facebook to build their internal and external-facing apps and mobile apps.
- Facebook chose a restrictive license for both projects, limiting many company's interest in using the frameworks. They've since switched to a more community friendly license, so there's less legal risk in using the frameworks for your projects.
- It isn't easy to build a cross-platform native app using React Native. The framework and community offer completely different UI components for Android and iOS, meaning you must code the UI for your apps separately. There are some common UI elements across platforms, but app shell components (like toolbars and navbars) are specific to a platform.

From a tooling standpoint, React and React Native are popular enough that developers have multiple options to choose from:

- Deco IDE: A free, open-source React Native IDE for systems running

macOS from Airbnb (believe it or not - Airbnb acquired Deco, the company that wrote the IDE).

- Nuclide: A React Native development environment running in the Atom editor (from Github) for Linux and macOS. Windows is 'supported' but not the full experience.
- React Native Extension for Visual Studio Code: Visual Studio Code is the most popular text editor in the world. It's free, and open-source (it's a fork of Github's Atom editor) and is easy to use. The React Native Extension adds full support for React Native applications, enabling developers to code, test, and debug React Native applications.

Reasons to Build JavaScript-Driven Native Apps

- JSN apps provide an easier way to build native mobile apps than writing native code.
- JSN apps enable development organizations to leverage their existing web development skills.
- JSN apps are true native apps built using native UI components, so users likely won't know the difference.
- JSN platforms and tools are free, with a robust community behind them enhancing the platform.

Reasons Not to Build JavaScript-Driven Native Apps

- You don't trust JSN over pure native apps.
- You want the latest and greatest native capabilities, no matter what.
- Limited web development skills on-hand.

Other Types of Native Apps

There are even more ways to craft your mobile apps; this section highlights some popular ones.

Xamarin

[Xamarin](#) is a cross-platform development platform that enables developers to code native mobile apps using [Microsoft's C#](#) language with .NET as the underlying framework. The platform supports Android, iOS, and Windows. Xamarin is most popular in organizations with a big investment in Microsoft technologies, and a lot of experienced C# and .NET developers.

Originally created by the team that created [Mono](#) (an open source alternative to Microsoft's .NET framework), Xamarin (the company) opened in 2011 and was acquired by Microsoft in 2016. Microsoft immediately open sourced Xamarin and later released Xamarin Studio (Xamarin's IDE) as Visual Studio for Mac. The good news is that Xamarin, like NativeScript, has the full support of a corporation behind it, even though the platform is free.

Xamarin apps are native apps, the app's C# code is compiled into a format that executes in a native runtime environment on the device. Developers create an app's UI declaratively or through XML; at runtime the app UI renders using native UI components. Like with NativeScript and React Native, Xamarin apps look and feel like native apps (because they are native apps with native UIs).

Another variant of this approach is [Codename One](#), which takes a similar approach, although using Java as the underlying language.

Flutter

[Flutter](#) is a recent offering from Google; the framework has been in beta for quite a while, but may be released by the time you read this. Flutter is a bit like NativeScript and React Native in that it lets you build cross platform apps using a non-native language. For Flutter, Google chose to use the [Dart language](#) rather than JavaScript. Flutter app UIs are built using a catalog of widgets Google provides. Flutter apps are native apps, but don't use native UI components. This means that the apps will look like native apps, but may not deliver the performance of native apps.

Flutter is still in beta, so it is too early to tell what's going to happen with this one. There's a lot of interest in the market, and developers seem to be delivering cool apps using the framework, but using it requires that you learn an obscure language (Dart) and that may be a deal killer for many.

Reasons to Build Other Types of Native Mobile Apps

- Existing investment in the languages used by these platforms and frameworks (C# or Dart).
- Interest in trying another way to build mobile apps.

Reasons Not to Build Other Types of Native Mobile Apps

- Vendor lock-in; using these proprietary, although open-source, approaches means you can't re-use your code if you switch approaches later.
- Use of obscure languages. C# is very popular with Microsoft's customers and for those using Mono or Microsoft's newly open sourced .NET Core. Dart isn't even listed on Stack Overflow's 2017 Developer Survey's list of popular languages (although it is 23rd on the list of most loved languages).
- In Flutter's case, a new and unproven platform.

Making The Transition

You may be asking yourself, how do I pick a path and run with it?

For web apps, migrating to PWA is not a big deal, all you're doing is adding some additional functionality to your existing app, wrapping it in PWA capabilities by adding a couple of files and some JavaScript code to your app. The app runs as originally coded on older browsers, but on PWA-capable browsers, the extra code kicks in and makes the app, well, better.

Can you migrate from native apps to PWA? Yes, absolutely - that's what we expect to happen for many apps. You won't be able to use any of your existing native code though. For apps that don't need any special capabilities provided by native APIs or native UI elements, ditching multiple native apps (Android and iOS, for example) and replacing them with a single PWA makes perfect sense. It's much less expensive to deliver a web app experience that covers both desktop and mobile users than building web and native apps to cover the two audiences. You should assess your apps for this today.

Assuming you're already doing native development, hybrid is no longer a viable approach, so the argument is probably for migrating to PWA or JavaScript-driven Native.

If you're currently building hybrid apps using Apache Cordova (or one of its variants), the transition to JSN won't be that hard. You're already coding apps using web technologies (HTML, CSS, JavaScript), you've selected your favorite IDE for web development, and you're comfortable using command-

line tools. If you're using a modern web app development framework like React, you're already coding more JavaScript than HTML, so you're in good shape.

If you're hand-crafting your web apps, or using an older web development framework (like Bootstrap, Dojo Mobile, or jQuery Mobile), then you'll have some work to do. You and your development team will have to get comfortable using modern, JavaScript-focused frameworks. Your tools will stay the same, so there's no adjustment there.

If your team is knee deep in native development for Android and iOS, then the transition's going to be a bit harder as most native developers don't have a lot of experience with web development and JavaScript. You'll likely keep your native development team where it is, maintaining your existing native apps while you repurpose your web development team or bring on web developers specifically for building mobile apps.

In the JSN world, your native developers are still an indispensable resource. Even though you're writing your native apps in JavaScript, you'll still need native developers (or at least native app designers) to help design and implement your app UIs. An inherent understanding of how mobile apps work is still important no matter which approach you're using.

Testing Your Apps

Mobile device platforms provide their own proprietary test automation tools for executing functional tests, but the market demands tools that enable developers to code mobile app functional tests using a single set of tools across multiple target platforms. This led to many proprietary and open-source solutions, but ultimately open-source options won. Vendors offering proprietary solutions quickly learned that they also needed to support popular open-source solutions.

The leading open-source solution for web application testing is Selenium WebDriver. To accommodate testing mobile apps, the community implemented native mobile app testing using the WebDriver protocol through the open-source Appium framework. Most native mobile app testing today happens using Appium; if you're looking for experienced testing resources in the market, most will have experience with Appium. All of the major testing automation tool and device lab vendors support it as well.

Mobile app testing used to involve buying a whole bunch of mobile devices, and moving them around as developers and testers needed them. This led to many homegrown, but interesting, solutions to build device labs - connecting devices to USB hubs and writing special software to expose all of them to an automated test suite.

In today's mobile development world, device labs are available as capable open-source and commercial offerings from many vendors. You can deploy internal device labs, buying devices, connecting them to a server

and making them available to developers and testers all throughout your organization. Offerings in this space include the open-source [Software Test Farm](#), and commercial offerings such as [Micro Focus Mobile Center](#), [Mobile Labs](#), and [Telerik Test Studio® Mobile](#).

You'll also find a multitude of device labs in the cloud where the vendor buys the devices and leases them to you (either as devices dedicated to you, or shared across multiple customers). Remote devices simply become another execution target for your automated tests. Offerings in this space include [Amazon Device Farm](#), [Microsoft's App Center](#), and [Perfecto Mobile](#).

Developers will still keep some popular devices around for debugging as most device lab offerings don't include the ability to use a remote device as a debug target in platform IDEs. That will change, as there's really no need to keep two sets of devices: one for testing and another for debugging.

Building Your Mobile App Back-End

Long gone are the days of building proprietary back-end services for your mobile apps on-premises; it's too hard and too expensive. Over the years, the open-source community delivered a wide variety of robust, and very capable tools developers use to code and assemble back-end infrastructure for their apps. If you're looking for a scalable SQL or NoSQL database for your app, or an easy way to deliver microservices or authentication capabilities to your app users, there's multiple solutions available to you.

When it comes to scaling your infrastructure to accommodate your app's target audience, it no longer makes sense to deploy on-premises. Through very capable cloud offerings from Amazon, Google, Microsoft, and others, it's very easy to package your app services into one or more containers, and deploy them into an automatically scalable cloud environment. As demand increases, these cloud environments scale up to support it, scaling back down again as usage decreases - you only pay for the compute and storage you use.

It's no longer a million dollar venture to build and operate a successful mobile app.

Conclusion

As we've shown, mobile apps are built using a completely different set of technologies than they were just a few short years ago. Native apps were king, but that's no longer the case; they're no longer necessary, but they'll still be around. Progressive Web Apps (PWA) make web applications feel like native apps, so it's easier now to leverage your existing web development skills to deliver what mobile users want and need. If your app needs native UI capabilities, and native performance, JavaScript-driven Native (JSN) is a capable alternative and will be the way most native apps are written in the years to come.



About the Author

Dan Wilson

Dan Wilson is the Senior Product Marketing Manager for Mobility technology at Progress. Dan has extensive experience growing technology focused products and services. He got his first taste of fast-moving bleeding edge tech when he joined his first start-up in 1999. An avid participant in technology communities, he contributes to a variety of open-source projects, and presents at numerous developer conferences worldwide. Prior to joining Progress, Dan founded and directed a consulting practice for 10 years.

About Progress

[Progress](#) (NASDAQ: PRGS) offers the leading platform for developing and deploying strategic business applications. We enable customers and partners to deliver modern, high-impact digital experiences with a fraction of the effort, time and cost. Progress offers powerful tools for easily building adaptive user experiences across any type of device or touchpoint, award-winning machine learning that enables cognitive capabilities to be a part of any application, the flexibility of a serverless cloud to deploy modern apps, business rules, web content management, plus leading data connectivity technology. Over 1,700 independent software vendors, 100,000 enterprise customers, and two million developers rely on Progress to power their applications. Learn about Progress at www.progress.com or +1-800-477-6473.


Worldwide Headquarters

Progress, 14 Oak Park, Bedford, MA 01730 USA

Tel: +1 781 280-4000 Fax: +1 781 280-4095

On the Web at: www.progress.com

Find us on  facebook.com/progresssw  twitter.com/progresssw

 youtube.com/progresssw

For regional international office locations and contact information, please go to www.progress.com/worldwide

Progress, Kendo UI, NativeScript & Telerik Test Studio Mobile are trademarks or registered trademarks of Progress Software Corporation and/or one of its subsidiaries or affiliates in the U.S. and/or other countries. Any other trademarks contained herein are the property of their respective owners.