The Pros and Cons of Mobile App Cross Platform Development

By Yuri Brigance

Digital Experience & Mobile

In a world with two dominating mobile operating systems there is frequently an expectation to develop two versions of the same app. Very rarely do businesses chose to only develop for iOS, or only for Android. If they do, the app usually targets some rare and specific feature of one platform that the other lacks.

Naturally, product owners are looking to reduce cost and development time. One way to do this is by looking for ways to share as much of the codebase as possible between the two platforms. There are many cross-platform development tools that claim to solve this issue. Are they worth using? Do they actually save time and effort? What are their limitations and advantages over native development?

Native App Development

Before we look at several cross-platform tools, let's briefly talk about the benefits of native application development. Mobile applications for iOS are written in Objective-C, and more recently in Swift; while Android apps use Java or Kotlin. Maintaining two separate native codebases can seem costly for a number of reasons:

- Two separate codebases using different and incompatible languages
- The same features implemented twice, but not necessarily in the same way
- Different build tools, IDEs, CI/CD pipelines, UI builders, separate tests, etc.

But there are definite advantages, too:

- Fast code execution, smooth UI, familiar experience for app users
- Easier to comply with each platform's unique UI design guidelines
- Complete control over all platform features, hardware, networking, etc.
- Use feature-rich native platform-specific development tools
- Easier to hire expert developers versed in a specific platform & language(s)
- Fewer unknowns so that developers don't have to learn new technologies

Developers generally chose to specialize in a specific field, and their performance is reduced when they are tasked to write code using a language or platform that they are not familiar with. Native app development is the path of least resistance for finding a skilled workforce that can hit the ground running with little to no ramp-up time. Primarily because developers will write code in a language they love, for a platform with which they have significant experience, using tools they already know.

Cross-Platform Development

When most non-developers hear the words “cross-platform” they instinctively think it means “write code once, run anywhere”. Indeed, that would be the holy grail of development - but then why would separate platforms exist? If both platforms had the same features, there would be no difference between them; it would be the same platform.

The reality of cross-platform development is somewhat murkier. There are definitely some shared features between iOS and Android, and it would be nice to be able to only write code once to utilize them. But there are also major differences that are platform-specific--too many to list. Take Android’s hardware and soft buttons (like the “back” button) and compare them to Apple’s “home button” (or lack thereof in iPhone X) and you can see that the way users interact with each OS is not the same.

The advantages look great on paper:

- Save time and money by writing the code just once for both platforms
- Only maintain one codebase
- Consistent look and feel across both platforms

But look under the hood and you will discover some serious potential pitfalls:

- Developers have to re-learn tools, programming language, and abstractions (like networking, camera, etc.).
- Apps may not adhere to platform-specific UI/UX best practices. At best this will feel unnatural to users, at worst it risks rejection at submission time.
- Internal branching logic to deal with platform-specific things like networking, hardware, UI, storage (and more) will still be necessary.
- It will be more difficult to find developers experienced in a particular cross-platform tool who are also experts in both Android and iOS.

Developers spend years acclimating to the development tools of their choice, learning and keeping up with updates on specific platforms and understanding the intricacies of their chosen programming languages. **It cannot be stressed enough that switching software development platforms is an extremely costly and time consuming task.** There is a high setup cost--one that can be irreversible once any significant work has been done.

Additionally, many developers may not be happy with the change. They will have to learn a third-party tool that dilutes the power of their preferred platform and forces them to write code in an unfamiliar programming language. It can restrict their ability to effectively develop, test, and debug the code. This can lead to retention problems as developers may perceive becoming an expert in a potential "vaporware" tool that may or may not exist in a few years as a career-limiting choice.

Luckily, cross-platform development tools are exactly that—tools. In theory, a hammer can be used to drive a screw into piece of wood, but a screwdriver is a much better choice for that task, and a powered one is even better when you have a lot of screws. When applied correctly, cross-platform tools can produce real benefits. Depending on your goals, there are two main approaches to cross-platform development: hybrid and full cross-platform.

Hybrid Development

The hybrid approach can be a great middle ground between Native and Cross Platform development. With a hybrid approach, some components are shared between apps, but the apps themselves can be developed using native tools and languages.

One way this is frequently accomplished is with "webviews" - an embedded web browser that runs JavaScript to render the UI and communicates to native code via a JavaScript bridge. Another is using JavaScript Core to directly perform simple logic and calculations identically on both platforms. Usually these are custom written components partially backed by native code, since developers must provide a webview container and JS bridge to make the component render and be able to communicate with the rest of the app. This is not as time-consuming as it sounds since both platforms pretty much support this out of the box. A really nice side-effect of this approach is that you can write your UI or other logic components using JavaScript and use the same code on your website and in your mobile apps. Good candidates for this approach are custom UI elements like pie charts and graphs, as well as simple calculations like stream processing, sliding window averages, and counters.

When more serious performance is desired, however, webviews and JavaScript aren't the right tools to use. You wouldn't want to implement machine learning algorithms or image processing using an interpreted language. In this case you can also write portable C/C++ code that will compile and run on both iOS and Android platforms. You can even use Swift on both (we've covered that in a separate article). But there is still the issue of creating language bindings - your Java/Kotlin code needs to be able to exchange data with the C/C++ or Swift code. It is, admittedly, a bit easier on iOS since Objective-C is a superset of C++, and Swift is also a native language which has bindings into the C world.

Lastly, there are cross-platform tools which can be plugged into existing native code to provide components which utilize native UI elements. One of the most popular tools for this is React Native, created by Facebook. While it is true that you can write your entire app using React Native, they also provide a neat guide on how to integrate React Native into an existing application and reap the benefits of cross-platform code. We'll look into this tool, and others in the section below.

Cross-Platform Tools

Cross-platform tools like Xamarin, React Native, Ionic, and PhoneGap/Cordova claim they can completely replace the Android Studio and Xcode development environments. Most of them promise a "write once, run anywhere" approach, provide their own IDEs, and use their own programming language. Let's take a brief look at each one.

Xamarin

Xamarin, acquired by Microsoft, uses C# as the programming language and is probably the closest you will get to writing native code. It can be used in several different ways, with the goal of preserving as much native functionality as possible and just switching the language to C#.

You can continue using native tools to lay out the UI components - Interface Builder and XIBs/Storyboards on iOS, and XML layout files in Android Studio. Alternatively, you can go full cross-platform and use Xamarin Forms and XAML to write your layouts. A custom layout preview tool is available. However, your actual application code is written in Visual Studio, a non-native IDE.

Xamarin uses native components to render the UI and is compiled to byte-code so there is no runtime interpretation like with JavaScript and webviews.

Xamarin advantages include:

- Easy access to native code and native features.
- AOT (Ahead-Of-Time) compilation provides native-level performance.
- Use native UI or Forms as needed; both are rendered using native widgets.
- Easy to find C# developers. C# syntax is also similar to Java and C/C++.
- Compiled code, suitable for CPU and graphics intensive tasks.
- A lot internal app code can be shared (model objects, UI, etc.).

The disadvantages are:

- Using C# programming language to bind to native code.
- Some native method names are changed, requiring re-learning.
- Have to use Visual Studio instead of a dedicated native IDE.
- Still have to write separate code for Android/iOS, only just now with C#.
- Still have to deal with things like networking stacks and hardware separately.
- Limited binding to Swift programming language. Requirement to inherit from NSObject and use @objc annotation hinders ability to use pure-Swift third-party frameworks.

Despite the name "Xamarin" sounding vaguely similar to an anti-depressant medication, it won't necessarily cure all your headaches. If your app relies heavily on platform-specific features you will end up writing a lot of "if-else" statements for Android and iOS, but now you'll be using C#. Your developers may not be happy about having to use Visual Studio as their IDE if they have not been using it until now. The setup and learning curve is still fairly high because your C# devs also must have specific Android and iOS platform knowledge.

Xamarin may be good for fairly straight-forward apps that may do some CPU-intensive tasks, like an image filtering app or a video game. But it may not be the best tool to build a companion app for your next Bluetooth & Wi-Fi enabled fitness tracker. In either case, Xamarin is best used when you have a team of native C# developers that are looking to create apps for Android and iOS and are already familiar with Windows Studio and related tools.

React Native

Like Xamarin, React Native was created with the goal of replacing both Android Studio and Xcode entirely. Created and used by Facebook, it is very quickly gaining popularity. Aside from being backed by “Big Tech”, the barrier to entry for using it is fairly low. In fact, React Native may be one of the only cross-platform frameworks that can be used to create shared components within native code, or replace the native code entirely.

With React Native, developers will have to learn JavaScript and React. React advocates a “UI is a function of app state” approach. As your app state changes, it updates only the necessary elements. As with Xamarin, the templated UI is rendered using native components.

In terms of performance, what makes React Native different from Xamarin is its reliance on JavaScript, which is an interpreted language. While Xamarin compiles down to bytecode, React Native uses JavaScriptCore, which is present on both Android and iOS. Android’s JS Core supports JIT (Just-In-Time compilation) which speeds up JavaScript execution by compiling code at runtime after interpreting and executing it. But this is not supported by JavaScriptCore in iOS. Additionally, 64-bit execution mode is currently NOT supported on Android. The state of 64-bit support on iOS is unknown.

In terms of development tools, React Native allows you to either continue using native tools (when integrated into a native app), or use any text editor (like Sublime Text) and “Hot Reloading” to author your code and see the results in real-time. This is a nice feature, as you don’t have to recompile and reinstall the app each time to see the changes.

Advantages:

- Can be integrated with native code, or used completely cross-platform.
- Hot Reloading provides instant feedback without recompiling and reinstalling.
- Uses fast native widgets to render UI, but not all widgets on each platform are available.

Disadvantages:

- Interpreted at runtime, hurting performance of CPU intensive tasks.
- Not entirely native look and feel.
- Difficult to reach into native code when going fully cross-platform.
- High learning curve; complicated environment setup steps.

- No language bindings; must write an adapter to communicate with native code.
- No support for background code execution on iOS and limited support on Android (no services).
- Increased application bundle size due to a number of JavaScript dependencies.

By far the biggest advantage of React Native is the ability to provide shared UI widgets between both platforms by integrating into the native code. While React Native can provide full cross-platform functionality, the fact that it's interpreted at runtime and lacks 64-bit support makes it a poor candidate for apps that are more complicated than something like a storefront or a banking application.

Once you go full React Native cross-platform, it becomes very cumbersome to tap into the native code. With no real language bindings, developers must write "plugins" to access native features. This can completely negate the "single codebase" advantage since now the team is maintaining JavaScript, Objective-C/Swift, and Java/Kotlin codebases all at the same time. The learning curve can be costly since React is unfamiliar to most mobile developers and JavaScript is a weakly-typed language which will irk those used to a compiled strongly-typed programming environment. You really can't use pure web developers here either, since large platform knowledge is required for more advanced functionality. Additionally, with over a thousand JS dependencies, React Native is fragile, with official documentation struggling to keep up. As a result your team may spend additional time searching StackOverflow for answers about obscurely worded errors that seemingly came out of nowhere.

React Native abstracts some common functionality like networking and camera access, but here you run into an issue of reducing both platforms to their "least common denominator" and still end up with a lot of "if-else" code to truly take advantage of each platform's features. As platforms get updated and new frameworks are added, React Native's development team must write code to catch up. This means that you are almost always working with a partially outdated feature set.

When it comes to React Native, it seems best to use it as a cross-platform UI builder. Start small by integrating simple components and work your way up to applying it to more complicated challenges - so long as the cost of doing so doesn't outweigh the cost of just writing the same functionality natively. Another big advantage of React Native is, rather obviously, if you have a team of React web developers wishing to provide native mobile development services.

Ionic & PhoneGap

Both Ionic and PhoneGap use webviews to render components that look native, but are not actually true native widgets. There is not a lot to be said about using them to build applications. They are truly only (partially) suitable for situations where you need to render data with very little processing and storage - similar to a web application. These are all-or-nothing frameworks, so you can't augment existing native applications with shared components. All of your application code will have to be written using these frameworks.

Ionic does provide a UI editor, while PhoneGap seems to have a more limited GUI desktop application available. Interaction with native code is done via plugins and adapters, similar to React Native. This type of development is really the hybrid webview and JavaScript bridge approach, but taken too far and pumped with steroids. For anything more complex than a data viewer you will likely have to use a large number of plugins, or write your own, thus maintaining separate codebases in a variety of different languages. If you have ever used a really bad non-native looking banking application, it was likely written with either PhoneGap or Ionic.

Advantages:

- Common UI across platforms
- Live preview of changes

Disadvantages:

- Fully interpreted HTML running inside of a webview means slower performance
- UI may look native, but will feel strange and animate with choppiness
- No native code bindings
- JavaScript or TypeScript unfamiliar to most mobile developers
- Potentially maintain multiple codebases in different languages
- No background code execution
- Only partial, always outdated platform feature set available
- Does not use native development tools familiar to most developers

It is difficult to argue in favor of these webview-dependent frameworks, as it is very difficult to develop an application that feels truly native using them. There are a few good examples out there, but one has to wonder how easy it was to convince and keep developers willing to write mobile applications this way. Other questions spring to mind, such as how does this work with CI/CD environments, automated testing, and more.

Cross-Platform Tools

The table below shows the comparison between the various cross-platform options out there.

	Xamarin	React Native	Ionic & PhoneGap
Language	C#	JavaScript	TypeScript
Core Philosophy	Maximum compatibility with native code. Replace the language with C#	View reacts to application state; familiar to React developers	Uses web technologies for maximum portability across platforms
Performance	Good, on par with compiled	Native UI, but rest is mostly interpreted & no 64-bit support	Relatively poor performance, HTML UI, fully interpreted
Can Be Used In Native App Code	No	Yes	No
Uses Native UI Widgets	Yes directly, or via Forms	Yes, under the hood	No
Native Look And Feel	Yes	No	Kind of looks native, but does not act native (HTML)
Instant Updates Without Recompilation	No, requires recompilation like a traditional app	Yes	Yes, can also run in a web browser
Native Code Bindings	Yes, but not for pure non-@objc Swift	No, requires adapters	No, requires adapters
Background Code Execution	Yes	No	No
Platform Feature Set	Full	Partial	Partial

So what is our recommendation?

If your goal is to develop a flagship application which must be released by a certain deadline on a tight budget, but your team consists of only native mobile developers, then your best chance of reaching your goal is to opt for native development. This will avoid very costly ramp-up time associated with forcing developers to learn new platforms, programming language, and toolset. Pressuring developers to re-learn their platform of choice using different non-native tools and programming language will likely result in low morale, loss of quality, talent retention problems, lost time, and increased spending. But that does not mean that cross platform tools do not have legitimate applications that can provide significant benefits to some teams.

Xamarin may be effective if you have access to a number of C# developers who are interested in doing mobile development for iOS and Android--languages & tools already familiar to them. This is likely to result in fast, native code that does not have the usual limitations of not being able to utilize the platform's latest features. Just beware that on iOS you will likely lose the ability to use some purely Swift third party libraries. This is not a big deal right now, but Apple spent almost a billion dollars creating Swift so it can replace Objective-C completely on all of its platforms, and the inability of Xamarin to use pure Swift classes will eventually make it obsolete if this issue is not resolved.

React Native can be effectively used to create a number of shared UI components that plug into native applications, especially if you have access to web developers who use React. But beware of the learning curve associated with React and JavaScript if you only have native mobile developers, and keep in mind that web developers likely do not have the necessary platform knowledge to write complex mobile plugins. React Native can definitely save time by sharing some of the UI code between platforms, but may require initial time investment to ramp everyone up on the concept. It will not work for a project with a tight deadline unless the team happens to have a former Facebook React Native developer on it, but it can work in a team that is light on mobile developers but has significant React web expertise.

Using Ionic or PhoneGap for anything beyond a simple user manual type app cannot be readily recommended. And, as with all cross-platform tools, it is unlikely to save time or money on a project with a looming deadline, unless your development team is fully experienced using these frameworks.

There is a lot of value to be gained from using cross-platform tools. But just like any tools, they have to be applied to the correct problem to be truly useful. It is important to consider the ramp-up time, existing dev resources and their skills, as well as the maturity and support behind the cross-platform framework of choice. Our advice is to be wary of the "write once, run anywhere" line of thinking and take time to analyze whether or not cross-platform development is a benefit, not an impediment, for your team.

About the Author

Yuri Brigance is an award-winning developer with over eight years of experience within the consulting industry. He is particularly adept at developing mobile apps for both iOS and Android, with an additional background in Web Services development.

About AIM Consulting

AIM Consulting, an Addison Group company, is an award winning industry leader in technology consulting and solutions delivery. We solve critical business challenges for our clients, helping them delight customers, empower employees and drive innovation forward. Founded in 2006, we are ranked among the fastest growing private companies and the best companies to work for with a long track record of success with clients of all shapes, sizes and industries across the US.

Capability Areas

- Application Development
- Data and Analytics
- Delivery Leadership
- Digital Experience and Mobile
- Infrastructure, Cloud, and ESM

Learn more at aimconsulting.com.

Ready for a solution?

Contact us for a free consultation.

LET'S GET STARTED