February 28, 2020

How Capacitor Works

by Max Lynch
Hello there! Welcome to the first email as part of my Ionic Insiders newsletter. As part of this newsletter, I want to share more details about some of the things we've built or decisions we've made at Ionic, in a way that is less formal and polished. As if we're just having a conversation instead of a presentation.

I also realized that I had a lot of knowledge about a lot of the projects and features here at Ionic that I've kept locked up. The topic of this email in particular is a great example of that: I built the first few versions of Capacitor and most of my code is still in the project (remarkably!). I also made the decision to jump in and build it. It would be a shame to not share my thought processes and design decisions in doing so.

With that, let's jump right into this post on Capacitor, its history, and how it works:

How Capacitor Works

One of the things that I find interesting is just how esoteric cross-platform tools like Cordova and Capacitor remain to users of those tools. There's a perception of great complexity and wizardry behind them that really isn't true, and I think it's that perception that kept people from questioning whether Cordova could even be improved.

With that, I'd like to explore how these tools work, what they attempt to accomplish, and then move into why we built Capacitor and how it works.

Introduction

What are Capacitor and Cordova?

Put very simply, Capacitor and Cordova are two projects that wrap a native Web View and

enhance it with the ability for JavaScript in the Web View to communicate with native code.

Frameworks like Ionic Framework then run their UI and native integrations on top of Capacitor or Cordova, with most of the app running in the Web View to preserve cross-platform functionality and enable the productivity and standardization benefits of web technology, but they are free to break out of the Web View to call native APIs or render native controls.

Cordova was created in 2009 initially as PhoneGap and then open-sourced to become part of the Apache foundation. Capacitor was created in ~2018 by Ionic (with yours truly coding the first implementation).

Another way to think about them

Another way to think about these two projects that I really like is they are just different web browsers with some "non standard" JavaScript APIs they add to the Web View. So if Cordova is Firefox, Capacitor is Chrome (p.s. I use both Firefox and Chrome regularly!)

For example, Cordova added a new JS API that you could access with navigator.camera. Capacitor added a new JS API that you could access with Capacitor.Plugins.Camera. Cordova tried to forward-implement draft web standards hoping to one day no longer need the native implementation. Capacitor is more focused on providing clean and consistent APIs that are predictable and familiar but not following any standards efforts.

All this is to say the APIs these tools add are no different than if the Chrome team added a new API like window.somethingCrazy to Chrome.

Those APIs are then backed by native code and access to native APIs so they can do interesting things.

Why Build Capacitor?

Cordova works well, and remains the dominant way that Ionic apps are made useful as native apps. The Ionic team continues to support it and has no plans of phasing it out.

Why, then, build yet another project in this space?

The answer to that is both simple and very complex. The simple answer is because I felt like it could be seriously improved and I had the time and motivation to do it. The complex answer is that we knew so many of our users struggled with and were frustrated by the way Cordova does certain things that we had to do something or risk the entire Ionic experience suffering. It felt like we had no choice.

Could we have forked it? Yes, quite likely. While a rewrite is an opportunity to start fresh

and challenge assumptions, a lot of the concepts from Cordova are the same no matter how you approach the problem. I don't know how easily it would have been adopted back but didn't want to have to make any compromises or shoehorn it to accomplish that. I also was excited about embracing Swift on the iOS side to make the project easier to build, maintain, and activate outside contributions. Another big reason we didn't try is because we also didn't want to mess up Cordova. Cordova did and continues to work well for us and many of our users. Like I said, the answer is a bit complex.

But the biggest thing that pushed me over the edge to get building Capacitor was recognizing the environment that we were in c. 2018 was *very* different from the one in which Cordova was born. We had Swift! We had Cocoapods! We had modern JS and TypeScript! We had npm! We had Web Views that actually worked well!

If Cordova was going to embrace those technologies it might very likely require a rewrite anyways, so it was worth trying. I built a prototype first on iOS with Swift and it worked so well that I knew we had to go all-in.

Capacitor Design Philosophy

Capacitor was intentionally built with a few key design goals that have become part of its core philosophy. Many of these goals were made to avoid pitfalls in other projects.

- Capacitor makes 95% of the work building an app easy, and the last 5% easier than it normally is. That means a developer should never get stuck implementing something as long as they are willing to write some native code to do it, but they should be able to build the bulk of their app in the web layer.
- Capacitor native projects should become part of a developer's source tree. This enables Capacitor to achieve the goal of making the last 5% not a burden because developers can quickly modify the project to add custom code and avoid getting stuck. They can also trust it won't be modified by Capacitor and, thus, have control over it.
- Capacitor does as little modification as possible to your native projects to achieve the goal above, and most of that is on project start.
- Capacitor apps can be enhanced and troubleshooted like any other native app, meaning the wealth of iOS and Android-specific StackOverflow posts and other sources of information can be used directly instead of having to translate through an abstraction (such as Cordova)
- Capacitor treats plugin development as a first-class activity and designed it in deliberately from the start.
- Capacitor uses standard or defacto standardized tools wherever possible, such as TypeScript and npm/yarn.
- Capacitor keeps custom tooling to an absolute minimum.

- Capacitor prioritizes stability over features. This is an interesting tradeoff. We recognize that this layer of the stack needs to work and not break as much as that is possible. Breaking changes and project upgrades should be minimal and rare.

Keep some of these goals in mind as we explore the inner workings of Capacitor

How Capacitor Works

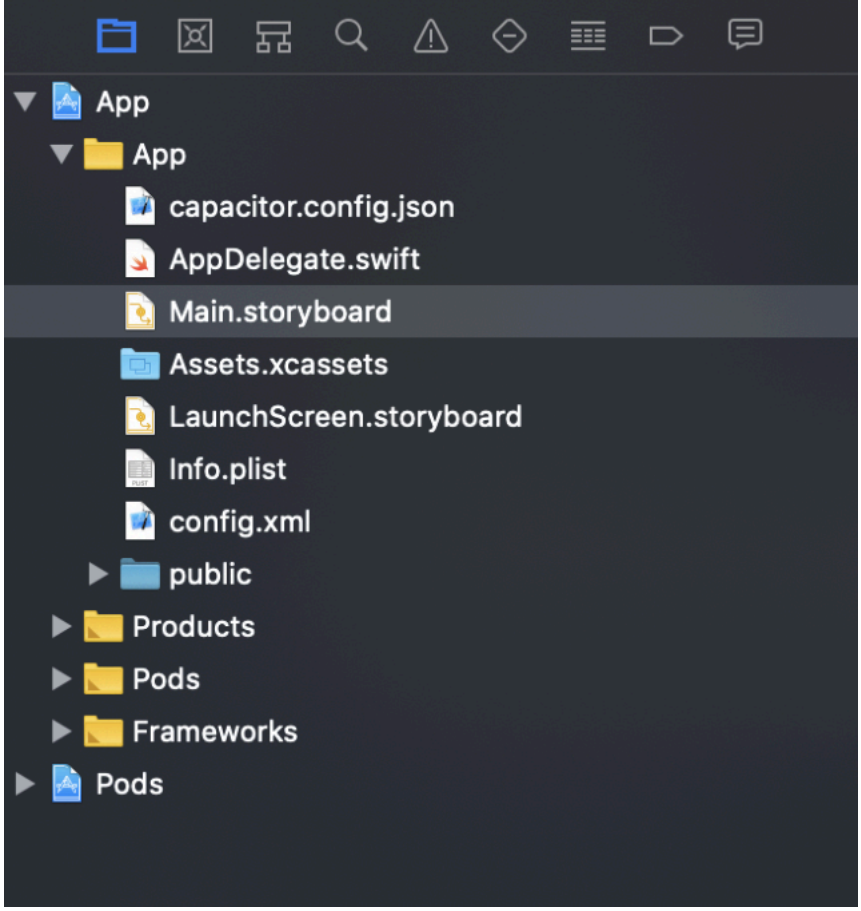
With the history and overview out of the way, let's jump into a technical exploration of how Capacitor works. I'm going to ignore Electron and Progressive Web Apps to start because they work a bit differently from iOS and Android and should be addressed separately.

The Structure of a Capacitor Native Project

Capacitor apps are and must be native apps in order to run natively, thus every Capacitor app must have a native iOS or Android project in order to actually run.

I'm going to focus on iOS because, while the technical details differ between iOS and Android slightly, they are actually remarkably similar, so just assume most of what I say here is similar on Android. I'll point out a few key ways they are different on the way.

When we start a new Capacitor app and add the iOS platform to it, the iOS workspace has this code structure:

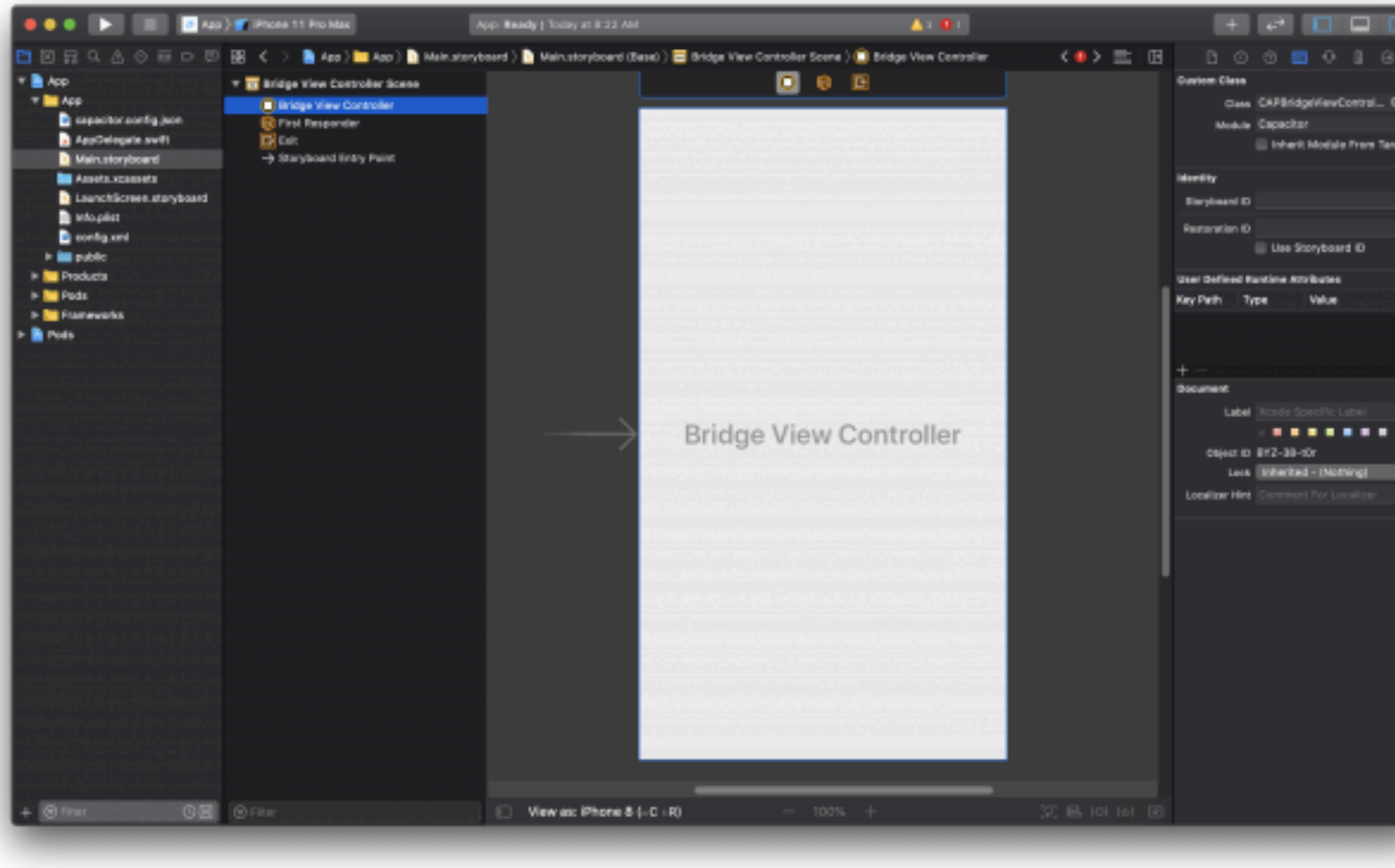


What you should notice above is that there are almost no Capacitor-specific files in your project! This is one huge difference between Capacitor and Cordova. Cordova was created before tools like Cocoapods so it works by copying all of its relevant files and any plugins you've installed directly to your project. Those files are then managed through Cordova's tooling. This approach is a big factor in tooling and plugin headaches developers face using Cordova.

In contrast, Capacitor's code lives in the Pods sub-project that Cocoapods manages for you.

This also means Capacitor's tooling is remarkably simple and small.

The magic happens with the Main.storyboard, where the only storyboard scene and view controller the app has (for this simple example) is the CAPBridgeViewController:



CAPBridgeViewController

The CAPBridgeViewController is the entry point for Capacitor. Every usage of CAPBridgeViewController (CBVC from here on out) can run a separate instance of Capacitor, so multiple instances can be used in an app and this controller can be easily used in an existing native app to embed mini hybrid apps (a common use case for big, old native apps that want to add agility through web-based screens).

CBVC manages a Web View instance and an instance of CAPBridge which is the Bridge that manages JS-to-native communication and invoking plugins.

When the CBVC, and thus your web app, initializes, it loads the URL specified in your configuration, and the Web View loads just like opening that URL in your browser. Only now it has access to a ton of new APIs through Capacitor.

Let's take a closer look at the Web View.

The Web View

CBVC manages a Web View, and that is its key responsibility. A Web View is a native control available on both iOS and Android that runs an embedded browser session. Basically, a Web View is just a browser without the chrome, but with some extra APIs that

make it useful for embedding.

In order to run a Web View, a Capacitor project needs to be a native iOS or Android project, and it needs to have a typical native UI hierarchy (hence why Capacitor apps are literal native apps). Thus, a Capacitor project is really a standard native iOS or Android project that has the same core UI controls as any other native app, but it puts a Web View front and center.

In terms of the actual Web View control, on iOS it's an instance of WKWebView which is iOS's modern "WebKit Web View". On Android it's an instance of android.webkit.WebView.

Capacitor then uses a few key APIs available natively on iOS and Android for Web Views to process JavaScript calls from the Web View and also inject code into the Web View.

To start, the Web View needs to be given additional JS calls to enable it to communicate with the native layer. On iOS, we get that for free as WKWebView provides an API called window.webkit.messageHandlers where we can add our own message handler using WKUserContentController (android is similar but a bit more open ended, we end up just defining a similar one to iOS). We add a new handler called "bridge" and then it's called using window.webkit.messageHandlers.bridge and it is given a method called postMessage which takes an object as an argument. That object can then be processed and unpacked at the native layer in order to call native code.

For example, it might be called as

window.webkit.messageHandlers.brige.postMessage({ "type": "message", "pluginId": "device", "methodName": "getInfo", "calbackId: "1000", "options": {})

Which then gets processed in the code below:

```
extension CAPBridgeDelegate {
  public func userContentController(_ userContentController:
    WKUserContentController, didReceive message: WKScriptMessage, bridge: CAPBri
    {
    let body = message.body
    if let dict = body as? [String:Any] {
      let type = dict["type"] as? String ?? ""

      if type == "js.error" {
        if let error = dict["error"] as! [String:Any]? {
          handleJSStartupError(error)
        }
      } else if type == "message" {
        let pluginId = dict["pluginId"] as? String ?? ""
        let method = dict["methodName"] as? String ?? ""
        let callbackId = dict["callbackId"] as? String ?? ""

        let options = dict["options"] as? [String:Any] ?? [:]

        if pluginId != "Console" {
          CAPLog.print("⚡  To Native -> ", pluginId, method, callbackId)
        }

        bridge.handleJSCall(call: JSCall(options: options, pluginId: pluginId,
          method: method, callbackId: callbackId))
```

Here we see the code that is responsible for unpacking the JS call and pulling out the data in the object. Once we have the data we pass it to the Bridge which will then route it to the proper plugin. Metadata about the plugin call is saved in order to route the response back to the JS caller (which is tracked using the callbackId) and this works even through opening intents or, in some cases, through app restarts (see the App API's appRestoredResult event for more info on that).

native-bridge.js

While we have the native-to-JS bridge configured, we aren't going to just call it directly as there is a lot of extra stuff Capacitor plugins need on the JS side, such as routing asynchronous calls to native back to the original JS caller, and the messageHandlers API isn't very clean or nice looking.

Thus, we need some sort of JS management layer on top of the low-level Web View JS APIs to orchestrate calls. That's where native-bridge.js comes in. This file is found in the @capacitor/core package and it is injected into the Web View as a user script at the start of the document. It contains the low-level Capacitor JS API. You'll never have to interface with this API directly, Capacitor plugins call it automatically.

This file keeps track of native calls and their caller, and helps route messages back and forth.

That brings us to how Capacitor plugins work:

Capacitor Plugins

To expand the functionality available to a web app there needs to be a way to add extra native code to a project and then call it from JS. Capacitor accomplishes this with Plugins.

Plugins in Capacitor, unlike Cordova, *only* contain native code for that platform. A plugin author can choose to provide additional JS alongside their plugin, but that would be just like any other JS library and would ultimately just wrap the plugin's API in Capacitor. This difference is subtle but results in a very different plugin usage and development experience.

Let's walk through an example with the Device plugin and the getInfo method that returns information about the device, OS, etc:

```
@objc func getInfo(_ call: CAPPluginCall) {
  var isSimulator = false
  #if arch(i386) || arch(x86_64)
    isSimulator = true
  #endif

  let memUsed = diagnostics.getMemoryUsage()
  let diskFree = diagnostics.getFreeDiskSize() ?? 0
  let diskTotal = diagnostics.getTotalDiskSize() ?? 0

  call.resolve([
    "memUsed": memUsed,
    "diskFree": diskFree,
    "diskTotal": diskTotal,
    "model": UIDevice.current.model,
    "operatingSystem": "ios",
    "osVersion": UIDevice.current.systemVersion,
    "appVersion": Bundle.main.infoDictionary?["CFBundleShortVersionString"] as? String ??
    "appBuild": Bundle.main.infoDictionary?["CFBundleVersion"] as? String ?? "",
    "platform": "ios",
    "manufacturer": "Apple",
    "uuid": UIDevice.current.identifierForVendor!.uuidString,
    "isVirtual": isSimulator
  ])
}
```

In the above, we have a simple Swift class that extends from CAPPlugin, then one method called getInfo. Every plugin method takes one argument: a CAPPluginCall. This object contains all the data passed from the JS call. Plugin classes and methods are exported to objc

to make sure their symbols can be read by the dynamic plugin invocation system.

Once the call is received, the plugin runs any native code it needs.

To return data back to the JS call, the call has two methods: call.resolve and call.reject. These methods are named as such to indicate that the call is actually following Promise semantics.

There is one last step: registering the plugin and its methods natively, as well as indicating return types expected:

```
CAP_PLUGIN(CAPDevicePlugin, "Device",
  CAP_PLUGIN_METHOD(getInfo, CAPPluginReturnPromise);
  CAP_PLUGIN_METHOD(getLanguageCode, CAPPluginReturnPromise
)
```

This makes the plugin and methods known to Capacitor. Android has a similar process but can better use reflection to avoid this. I hope one day this last step can be removed and maybe it can be today, it's been a while since I got that deep in the low-level plugin registration code.

One of the really wonderful things about this is that you can easily build yourself little plugins right in your project. Need to just call that one API that isn't available in the web? Easy, just create a Swift file and build a little plugin and register it with Capacitor. Done! More on this in the docs.

Capacitor Plugins: JavaScript

Finally we get to one of my favorite features of Capacitor: automatic JavaScript code generation!

Remember in Cordova having to wait for deviceready? Well, the complexity and frustration this step causes was something I knew I wanted to fix straight away with Capacitor. Why shouldn't plugins be available immediately as soon as the page starts loading? After all, we have them all registered and known about in native before the Web View loads!

Turns out doing this is as simple as enumerating the list of registered plugins and their methods, and then injecting JavaScript-mapped APIs to call those plugins through the native-bridge.js functionality discussed above. We can do this at the top of the page so all plugin calls are available *immediately*.

```
public static func exportJS(userContentController: WKUserContentController,
  pluginClassName: String, pluginType: CAPPlugin.Type) {
  var lines = [String]()

  lines.append("""
    (function(w) {
    w.Capacitor = w.Capacitor || {};
    w.Capacitor.Plugins = w.Capacitor.Plugins || {};
    var a = w.Capacitor; var p = a.Plugins;
    var t = p['\(pluginClassName)'] = {};
    t.addListener = function(eventName, callback) {
      return w.Capacitor.addListener('\(pluginClassName)', eventName, callback)
    }
    """)
  let bridgeType = pluginType as! CAPBridgedPlugin.Type
  let methods = bridgeType.pluginMethods() as! [CAPPluginMethod]
  for method in methods {
    lines.append(generateMethod(pluginClassName: pluginClassName, method: metho
  }

  lines.append("""
  })(window);
  """)

  let js = lines.joined(separator: "\n")

  let userScript = WKUserScript(source: js, injectionTime: .atDocumentStart,
    forMainFrameOnly: true)
  userContentController.addUserScript(userScript)
}
```

There's nothing special to this routine. It involves literally constructing JS strings and inserting them into the Web View.

The net result is we will have access to JS calls like Capacitor.Plugins.Device.getInfo which will call our low-level bridge functions in native-bridge.js and let us call this method using Promises.

For those familiar with Cordova plugin JS, you don't need to do any manual cordova exec calls or really have any JS at all unless you want to add some JS-specific functionality on top (and publish it in a separate library). This is a huge win in my books.

Capacitor Plugins: Web and Electron

Capacitor fully supports Progressive Web Apps and Electron within a single codebase that

also targets iOS and Android.

The way this works is Capacitor registers a list of known web-implementations for certain plugins (for example, Camera), then at runtime, figures out if Camera.getPicture should use the native implementation or the web one through the use of a JavaScript Proxy.

The net effect is that Capacitor plugin implementations can easily be overridden on certain platforms at runtime, and both Web and Electron use this technique. Native plugins can even be overridden with pure-JS implementations when running natively, if you wish to use a pure Web API on all platforms.

Writing Capacitor Plugins

Plugin authoring was a key focus of Capacitor. In Cordova, building plugins involves following a Cordova-specific structure and the only way to test them is to install them and run them in an app. I believe you could also build them directly into the native project but I'm not sure about that, even so the tooling approach to not make native projects a source artifact means managing that code would be challenging.

Capacitor plugins, in contrast, use a standard library project for iOS and Android, and come with runable unit tests should you choose to use them. To start a new plugin project, Capacitor comes with a convenient CLI generator called using npx @capacitor/cli plugin:generate that will scaffold out a new plugin project.

In addition, Capacitor plugins can be built right into your project code, and I imagine this is how most plugins will end up being built. To do this, just follow the same approach of the Device plugin above and follow the custom code docs to see how to create a plugin right in your project.

As someone that has built a number of Cordova plugins, I think the above approach is considerably easier for plugin authors and I hope it leads to many more plugins being built for Capacitor, as well as removing the need to build external plugins for app-specific native functionality you don't intend to ever share.

Installing Capacitor Plugins

For external plugins, Capacitor embraces npm/yarn to manage plugins. Capacitor scans your dependencies to find Capacitor plugins and then syncs them with your native projects.

For example, if we have a custom Capacitor plugin called capacitor-plugin-bluetooth, we will install and use it like this:

```
npm install capacitor-plugin-bluetooth
npx cap update
```

And that's it. Capacitor will detect the plugin from your dependencies and install it into the native project.

To remove it just npm/yarn uninstall it and run the update command again!

A note on URL schemes, CORS, and web servers

Capacitor does not use a local webserver to host your web app code in the Web View. Thus, in order to handle requests from inside the Web View that need to be handled natively (for example, loading files in an image tag through a URL), Capacitor must use certain APIs that enable it to intercept those requests.

In order to handle web requests natively, Capacitor has to define a custom URL scheme on iOS (Android can handle normal http/https calls to localhost so does not need that). The WKWebViewConfiguration class can be used to add new custom scheme handlers to handle requests. The default scheme is capacitor:// on iOS and http:// on Android. The default hostname on both is "localhost".

Unfortunately, this has implications for CORS as many people have noticed and struggled with. However, they are not necessarily any worse than if we didn't use this technique at all. For example, we used to use file:// which resulted in a (null) origin, and you still have to deal with the localhost origin on Android.

There are some solutions here if you absolutely just can't get CORS working correctly. First, you can proxy your API server which can then communicate with the API directly and allow you to configure CORS as you see fit. Second, you could make the HTTP request natively, perhaps from a Capacitor plugin.

The latter is something we will be addressing soon. Work is already under way on a native Http API for Capacitor to ease CORS issues.

Cordova Compatibility

Capacitor has backwards compatibility for many Cordova plugins, as long as those plugins don't depend on very advanced or esoteric Cordova features. But it also has some noticeable caveats due to design philosophy differences.

To enable this support, we've ported over meaningful portions of Cordova and made it interopt with Capacitor. The end goal being that Cordova functionality and plugins would feel like they are running in a normal Cordova environment but under the hood things like accessing the web view would be mapped to Capacitor. It works quite well and I credit Julio (jcesarmobile) for building this functionality and making it work so well.

Then, the caveats:

First, there is no variable support on installation or hooks. This is one of the first things people notice when trying to use advanced Cordova plugins in Capacitor, but it's by design. To enable you to manage the native source project, we avoid making major modifications to it, and that includes settings and configuration changes. To achieve our goal of keeping Capacitor's tooling to an absolute minimum, we don't have code that modifies the many different ways native projects manage configuration and other functionality. This then helps us achieve our goal of keeping Capacitor as stable as possible because we have less tooling that is bound to the way Native projects manage low-level settings and configuration, which can change between versions of Xcode and the Android SDK.

Instead, we think there's a simple solution that liberates everyone: just document it! Does your plugin need the user to add a key and value to their info.plist? Just put it in the README! Since capacitor doesn't destroy or recreate the native project source, developers can trust changes they make will be preserved. This is the exact way that native developers manage settings and configuration and it's quite simple on its own.

When we thought about how much simpler the documentation solution was, it became silly to even consider building out complex tooling to manage native source projects given all the ways that was guaranteed to break in the future.

Finally, the only other major factor that can limit whether a plugin is compatible is if it relies on certain source files being available in Cordova land that might not be available in Capacitor. We keep an updated list of known incompatible plugins that don't work and are proud that the list is not actually that big.

Roadmap

I'm sure many are curious where Capacitor is going from here. Given that our design philosophy is to keep Capacitor very stable, we don't have a lot of grandiose ideas about what to do with it. We just want it to be stable, be simple to upgrade, have 90% of the core APIs most developers need, and make it easy to fill in the rest with custom plugins.

With that said, the team is working on a number of tasks for 2.0 release. I believe one of the bigger tasks was to add support for Android X and that work has completed. Take a look at the current 2.0 milestone for way more detail.

We will also be adding a new Http API to Capacitor shortly to make it easier to call APIs without dealing with CORS.

Conclusion

I hope this dive into some of the design goals and technical implementation details of

Capacitor helped you better understand how the project works, why it was built, and where it's going. I know just writing this out really helped me recall certain decisions and remember how this whole thing works (I haven't worked on Capacitor directly for a while, Julio (jcesarmobile) leads the project now and is doing a wonderful job).

I would love to hear if anything wasn't clear or if I could answer any additional questions. Or call me out for getting something wrong! Please feel free to reply to this email and I will try to help!

And let me know what you think of this format. This is a very long email and likely will be one of the longer posts I write, but I do like the freedom to go deep and not have to edit everything down to a few soundbites.

Until next time,
Max

Thanks to Julio for reviewing drafts of this email!