Refactoring our Swift SDK

Author: Noah Durell

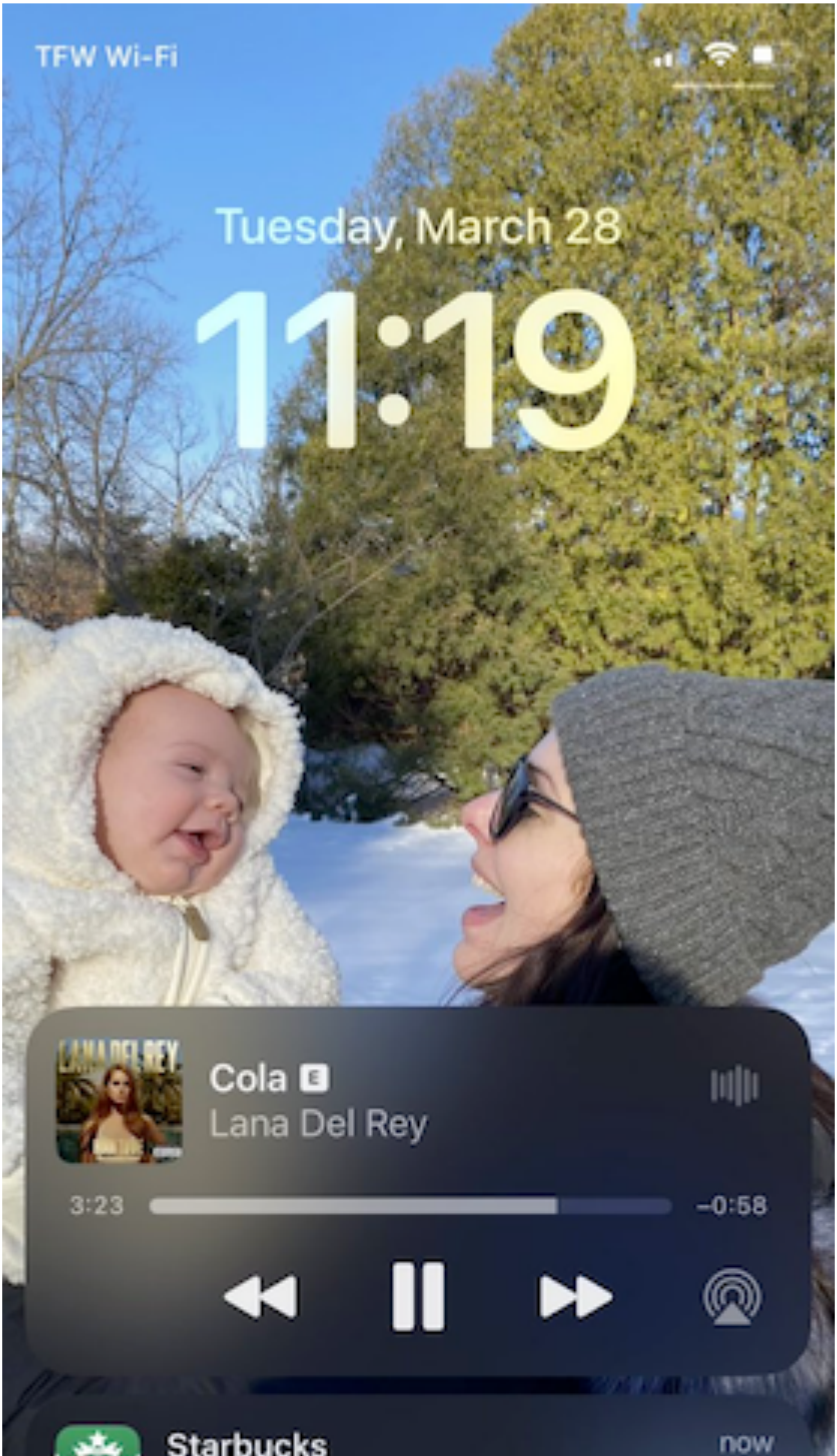
Claps: 28

Date: Jul 24

Iâ€™m an engineer working on our push notification platform. Klaviyo has had support for push notifications on iOS for many years. Recently, our team added support for Android, and this gave me and the team the opportunity to refresh our iOS offerings. So for the last several months, Iâ€™ve been leading the efforts to refactor our Swift SDK. In this post, Iâ€™ll outline the steps we took to accomplish this and explain many of the design choices we made to bring the new SDK to life.

Background

But first, letâ€™s explore push notifications and iOS development in general. Push notifications are short messages that can be sent to iOS devices via Apple Push Notification service (APNs). A developer configures an app to receive notifications by both registering for push notifications and asking the user for permission to send push notifications. You may have seen this before when you get an alert like this:



TFW Wi-Fi

Tuesday, March 28

11:19

Cola E

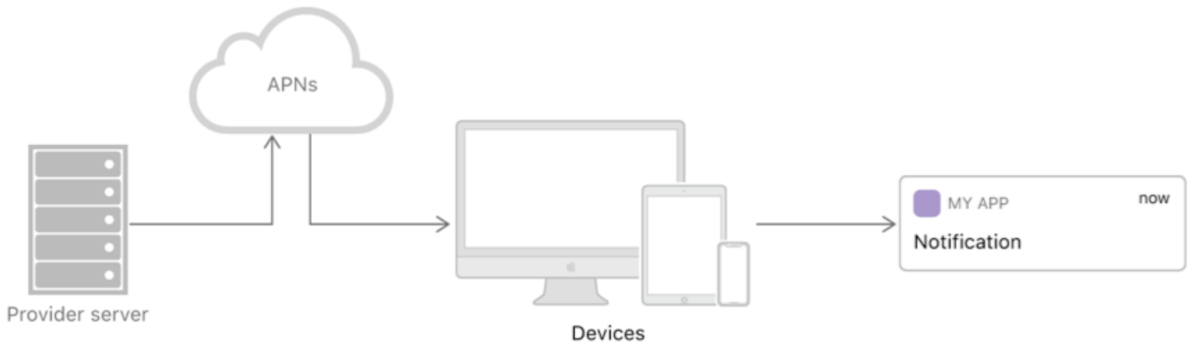
Lana Del Rey

3:23 –0:58

Starbucks now

Push notification sent to possibly millions of iOS users by now internet famous Sean K on March 28, 2023.

By registering the app, the operating system will provide your application with a push token. The token along with credentials provided by Apple can be used to send the notification to a phone (via Apple Push Notification service). Klaviyo provides services that allow you to store both the tokens for the users and the credentials to talk to APNs.



Our team builds and maintains SDKs that allow apps to communicate with Klaviyo. Our iOS SDK is written in Swift. Developers who work on iOS apps typically use Xcode to develop their apps. When a developer wants to support Klaviyo Push, they can use the Swift Package Manager (SPM) to download our iOS SDK and link it to their app. The developer then will make calls to the SDK, which in turn makes API calls to Klaviyo.

The SDK allows a developer to identify and store information about the user of their app, it allows them to track events back to Klaviyo (like when a user views a product), it allows them to associate a push token with said user, and it allows them to track when a user opens a Klaviyo push notification. Once a developer has completed the SDK integration, marketers who use the Klaviyo platform can now communicate with their users via push.

Getting Started with SDK development

Taking a step back, though, how does one get started with iOS development? For me, I got started when a cousin wanted to build an iOS app that curated Twitter content from comedians and entertainers. So in my spare time, I started watching Stanford courses online. This gave me a background in Apple APIs and Objective-C (this was the language used to develop apps before Swift). Between my full time job, learning how to build an app, and going back and forth with my cousin on design choices, it took me about a year to write the app and get it approved and published to the App Store.

I enjoyed the process so much that I decided I wanted to turn mobile development into my full-time occupation. I applied to a startup that was working on their own iOS SDK, which would allow developers to add a loyalty platform to their apps. While there, I was lucky enough to work with a developer who had written SDKs before. I learned a ton about how to structure SDK APIs, SDK best practices, and also got to work on their Android SDK.

Although the SDK work was fun and it had pretty wide adoption in top-tier apps, I was starting to miss working on code that users of the app interacted with directly. So I landed a job working on a more complicated app for an on-demand bus service. Later, I got the opportunity to learn Swift when I joined a company that wanted to refactor their app to follow modern mobile development practices, including converting the whole thing from Objective-C to Swift.

This brings us closer to the present, when I joined Klaviyo to work on our push offerings. My point in telling you all this is mainly to illustrate that if you have interest in mobile development, breaking in doesnâ€™t necessarily mean needing to completely retrain. Many online resources exist to get up to speed, and many of the toolchains are now much easier to work with than they used to be.

SDK Best Practices

Since an SDK ends up living inside another app, there are different design factors compared to building a mobile app:

- Get the API right
- Handle SDK state properly
- Rigorously test your code
- Make SDK setup easy
- Open source it if possible

In the sections that follow, Iâ€™ll detail how I applied these principles to our iOS Swift SDK and our new Android SDK.

SDK APIs

When you create an API for your SDK, remember that it will be used for a long time, so itâ€™s important to get it right. Even if you update it, you will rely on developers to update to your new SDK and then on users to update their apps to the new app version. This can often be several years until old devices go out of service. Thatâ€™s why you want to ensure that your APIs are good for the long haul.

How to use your API should be intuitive to developers. It should also prevent them from making common mistakes which could have downstream effects. For example, suppose you have an API that takes an event name as a parameter. You want to afford the developer a chance to pass whatever name they want to your API, so you might make this parameter a string. This is a (soon to be deprecated) API from our iOS SDK:

```
public func trackEvent(eventName : String?, properties : NSDictionary?)
```

While this is simple and straightforward, we can do better. As it turns out, many events pull from a common set of names (Viewed Product, Completed Checkout, etc.). If instead we use stronger types, we can leave the flexibility to specify custom event names but also prevent the developer from misspelling a common event. (Also the current API allows the name to be optional which in reality shouldnâ€™t be possible.) The API below proposes something like this:

```
public func create(event: Event)
```

Where Event (abbreviated) looks like this:

```
public struct Event: Equatable {
 public enum EventName: Equatable {
 case OpenedPush
 case ViewedProduct
 case SearchedProducts
 case StartedCheckout
 â€¦
 case SuccessfulPayment
 case FailedPayment
 case CustomEvent(String)
 }
 public let eventName: EventName
```

Why is this better? Well for one thing we donâ€™t allow for invalid parameters like nil, but also it gives developers suggestions for common event names. They now wonâ€™t have to worry about misspelling (unless we misspell them!), and IDEs like Xcode can give type hints for autocompletion. One potential downside is that logging custom events will take a little more code, but only very little. We also extended this type of thinking to the rest of the event object. For example, we added stronger types for dates and values. Again the idea is to offload the mental burden for the developer when adding Klaviyo to their app.

State Management

You might think that once the API is defined weâ€™re done and the implementation of each function can just directly make a network request to Klaviyo. In fact, our SDK needs to keep state to ensure continuity between requests, and to manage the flow of network requests going out to the server. We also expect this state to grow over time as we add more features. This is where state management comes into play.

In our SDK we keep one central state and ensure that any changes are handled in one place. This is important because code in the SDK can be called from everywhere within the app (and on any thread). By structuring it in this manner, the state changes are predictable, easy to track, and easy to test.

To accomplish all this, we use a pattern that originally became popular in the React world. **Actions** update **state** via a **reducer** function, whose job is to accept the current state and an action and generate a new state. If the state change requires outside work to be done, like a network request or interaction with a filesystem, that is achieved via an **effect**. When the effect completes, new actions are passed back into the state and it is updated like before.

For example, in our SDK we have state for the request queue as well as identifying user information. So an example action might be *enqueueRequest* or *setEmail*. The enqueueRequest action appends the request to an array in state. The setEmail action updates the email in state and also enqueues an identify request to later be sent to Klaviyo.

Although we could have implemented much of this ourselves by hand, we instead leaned on a project called the Composable Architecture. This made it easy, for example, to implement a simple FIFO request queue that syncs data back to Klaviyo and handles retries. In the future, we may implement a debug only feature to enable capture of actions within the SDK from a live app. This will save actions to a log file that could be uploaded with a bug ticket. With this data we could replay the actions in our dev environment (a.k.a. time-travel debugging) which will simplify debugging complex issues.

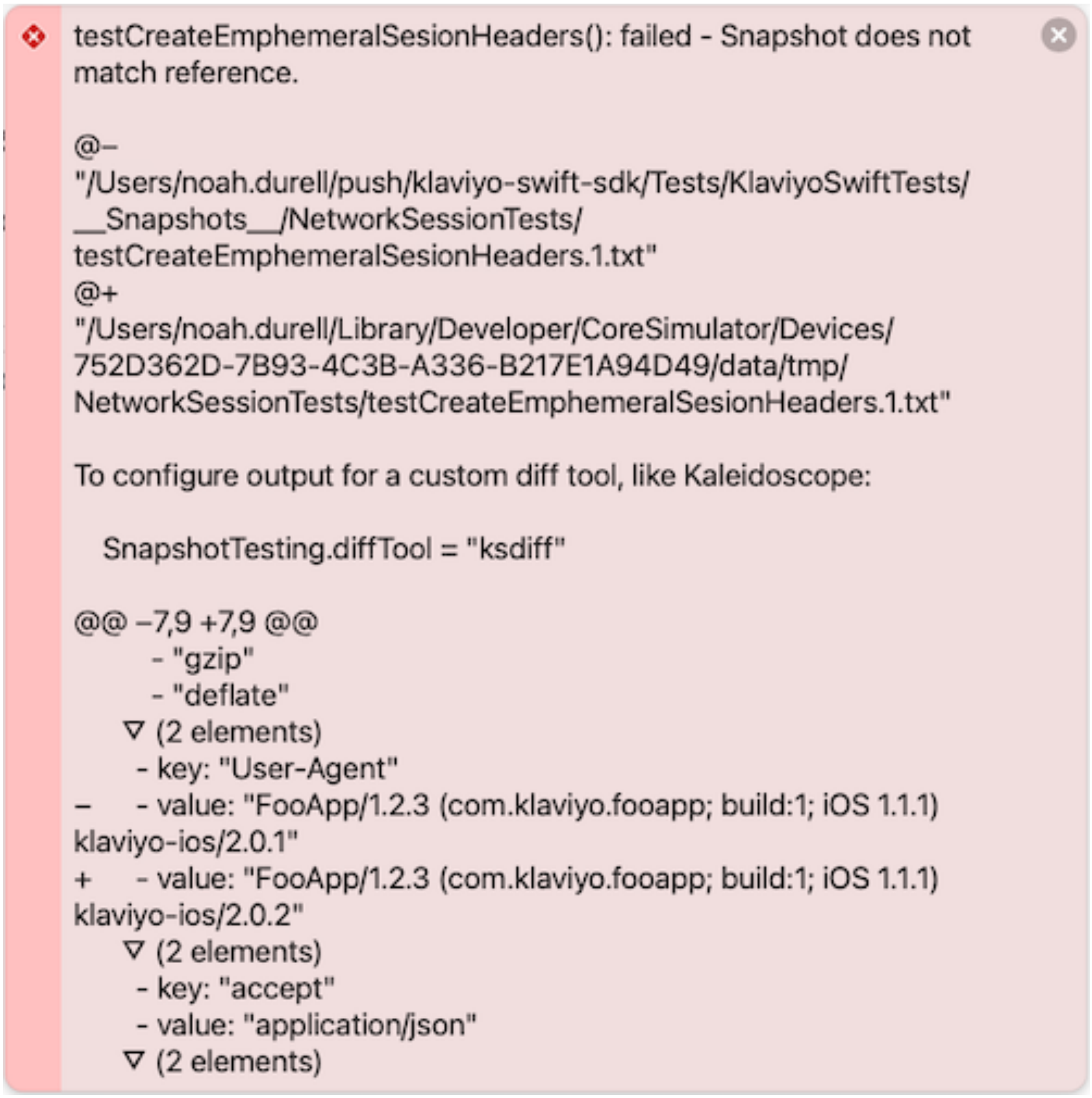
Testing

When youâ€™re building an SDK itâ€™s important to have a good testing strategy. A bug in your own app is bad enough, but introducing a bug into someone elseâ€™s app is on another level! Also, for the reasons explained above, you will often need to live with the bug for a very long time. Thatâ€™s why we spend a lot of time ensuring that our SDK is properly tested.

We start by writing testable code. For us this means making any SDK â€œside effectsâ€� swappable during unit tests or, to use a three-dollar term, using â€œdependency injection.â€� So when we need to do something like interact with the network or the filesystem, we can simply swap in a call that does something more predictable and simulate whatever conditions we need for the test (e.g. a down network).

Composable Architecture comes with TestStore. In tests, we use this to replay actions for common scenarios and assert that the state ends up being modified in a particular way. This type of testing is powerful and can catch subtle bugs. I would also say that it makes the test fairly easy to understand.

We also use a snapshot testing framework in some of our tests. These tests capture the state of various objects and save them to disk. For example, we may snapshot the format of a network request (as seen here). If another developer comes along and changes something about the request, they can now compare the new snapshot with the old and have a better understanding of the impact of their changes. This too can help to catch subtle errors like misspelling something or forgetting to add a request header. Hereâ€™s an example where the user agent has changed:

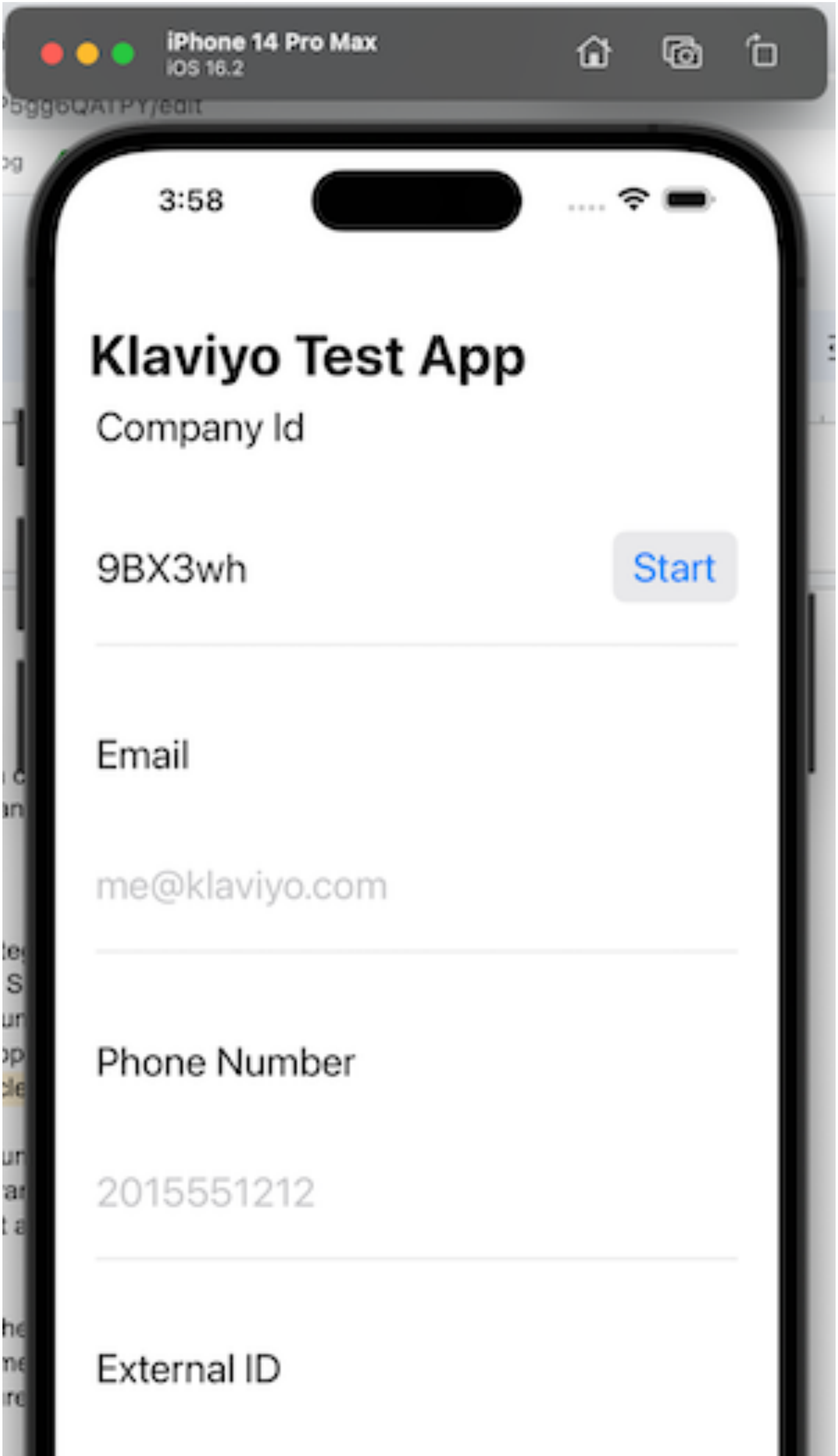


```
testCreateEmphemeralSesionHeaders(): failed - Snapshot does not
match reference.

@−
"/Users/noah.durell/push/klaviyo-swift-sdk/Tests/KlaviyoSwiftTests/
__Snapshots__/NetworkSessionTests/
testCreateEmphemeralSesionHeaders.1.txt"
@+
"/Users/noah.durell/Library/Developer/CoreSimulator/Devices/
752D362D-7B93-4C3B-A336-B217E1A94D49/data/tmp/
NetworkSessionTests/testCreateEmphemeralSesionHeaders.1.txt"

To configure output for a custom diff tool, like Kaleidoscope:

  SnapshotTesting.diffTool = "ksdiff"

@@ −7,9 +7,9 @@
        - "gzip"
        - "deflate"
     ▽ (2 elements)
       - key: "User-Agent"
−      - value: "FooApp/1.2.3 (com.klaviyo.fooapp; build:1; iOS 1.1.1)
klaviyo-ios/2.0.1"
+      - value: "FooApp/1.2.3 (com.klaviyo.fooapp; build:1; iOS 1.1.1)
klaviyo-ios/2.0.2"
     ▽ (2 elements)
       - key: "accept"
       - value: "application/json"
     ▽ (2 elements)
```

Output of a failed snapshot test. Looking closely we can see that a version changed tripping up this test. If we are ok with this change then all that needs to be done is delete the snapshot and re-run the test to regenerate it.

Test App

Once we had a good foundation of unit tests, we developed a test app. The test app offers a simple UI that exercises as much functionality of the SDK as possible.


iPhone 14 Pro Max
iOS 16.2

3:58

Klaviyo Test App

Company Id

9BX3wh

Start

Email

me@klaviyo.com

Phone Number

2015551212

External ID

Our test app allows the user to start our SDK, create profiles, log events, and of course receive push notifications. In addition, we expose special hooks into our SDK for the test app that allows us to see exactly which requests are being sent as well as the server response.

We distribute this app beyond the push team via TestFlight so that others can help us test the SDK. The app is also handy for our support and product teams to debug customer issues with push notifications since they can simulate the behavior of our customersâ€™ apps.

CI Test App (Future)

In the future, we plan to have an app that is run by CI. This will allow us to have a full end to end test of the SDK on every push to master. This will make us less dependent on manual testing. Weâ€™ll use a device farm, a service that gives access to a set of iOS and Android devices that can run our app. These tests have many moving parts and can be a bit flaky. However, given that mistakes in an SDK can be so costly, not only for us but also our customers, we believe it will be worth the added complexity.

SDK Setup

If an SDK is difficult to set up and integrate into an app, you may see low adoption rates. This is why itâ€™s important to have solid SDK documentation. We spend a long time developing and testing our documentation to ensure it is correct and easy to follow. We also keep the number of steps the developer has to complete to a minimum, and provide realistic code samples for each step so it is clear what the developer has to change in their app.

Another part of the setup is getting the SDK bits to the developerâ€™s app. Because developers may use different package management schemes, we currently support both SPM and Cocoapods deployments. In the future, we may also support Carthage.

We also put together a sample app to demonstrate the SDK. This is somewhat different from the test app described above in that it is more realistically structured. Our sample app is a mock store where you can add items to your cart and check out. We documented where API calls are made within the app so developers can copy the pattern.

Open Source

Another fun part of working on the Klaviyo SDK is that weâ€™ve been able to make it open source. This makes it easier for developers to understand how the SDK works, and to debug issues if they arise. For example, developers can drop breakpoints directly into the SDK to help identify the root cause of a problem. Iâ€™ve also personally found it rewarding to interact directly with developers to get feedback on the SDK and help them get unstuck if they run into issues. As a developer myself, I prefer to use open source software whenever possible, so I can be more confident about the code Iâ€™m adding to my app. My hope is that by making our SDK open source, it can make our developers more confident about using it in their own apps.

Conclusion

In this blog post Iâ€™ve discussed many of the factors that went into updating our iOS SDK. If you want to explore the new API, take a look at version 2.0.0 or later. (2.0.1, the most recent as of the time of this post, was released on May 2.) Our team would love to hear your feedback and suggestions! Thanks for reading!