objectstyle

Developing white label mobile apps for a client

Our client is a technology startup that provides an employee training platform to corporate users. The platform is available in the form of two native white-label mobile applications.

Industry
Education

year founded
2018

location
USA

Challenge

Since our client is a startup, they wanted to keep the size of the development team small and agile. To cut down costs, they could have developed one cross-platform application, but then certain compromises would have to be made in terms of the UX (non-native apps may provide a subpar user experience in certain environments). Since our client wanted to lay a solid foundation for great UX from the start, they went with building two native applications whose look-and-feel would appear natural to Android and iOS users.

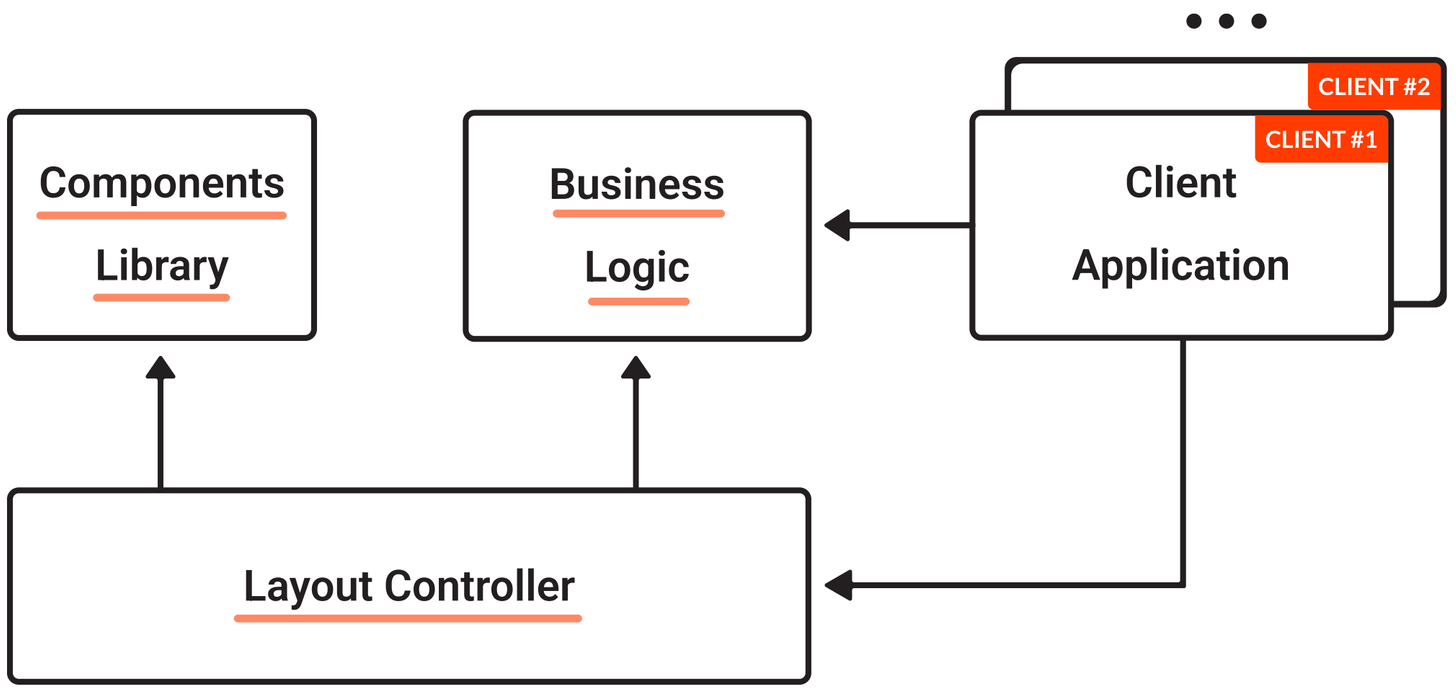
At the same time, our client's products are meant for B2B customers who are very demanding about the app's ability to match their brand identity. The majority of those corporate users have very specific branding requirements, including a highly-customized UI layout and functionality. Our client understood that the cost of onboarding new users, each with their customization needs, could eventually become a problem. Because the client had limited development and QA resources, they realized they had to be smart about satisfying their B2B users' needs, while also keeping a competitive production cost and time-to-market.

Solution

The solution was to avoid compromising on the UX and to develop two mobile apps that are native to the iOS and Android system. We had to ensure that, with each new corporate app user, the code base didn't grow out of proportion. We had to keep maintenance costs low.

ObjectStyle developers solved this by adopting a smart architecting approach. The underlying app needed to provide white-labeling capabilities – we needed a mechanism for customizing it for each new user with minimum effort. There had to be an optimal amount of code that was to be shared by all user applications. And we needed to single out components that varied depending on a client: for example, the layout of elements on the screen, brand-specific fonts and colors, etc.

Native iOS app architecture



Modules

1. Core.

This module has no dependencies. It serves as the source of business logic and use cases for the Screen Kit.

2. Screen Kit.

Screens can follow brand-specific business logic, which is stored in the Core module. This module uses components from the Component Library to build screens and UI layout for apps, as well as connect components to the Core. This module is also responsible for navigation and screen routing.

3. Component Library.

This module contains theme-agnostic, general-purpose UI components (such as buttons, icons, forms, etc.) that are used to build interfaces. They are not related to the Core repo in any way.

Native Android app architecture

Android app's architecture largely mimics that of its iOS counterpart. But it's a different platform, so sometimes similar goals are achieved by different means.

Modules

1. Core.

It is the only part that “talks” to the backend. The purpose of this module is the same as that of the Core part in the iOS app.

2. Common UI.

This module effectively combines the Screen Kit and Component Library of the iOS application on a conceptual level.

Results

Both apps are flexible and well-positioned for the future. The team has a plan for various "what if" situations that may arise. If a user wants to add non-standard screens that are not a part of the current modular approach, we can do it with minimum modifications. If a client wants to integrate a third-party API for their brand only, it is possible, too. In addition, the business logic code is ready to be developed using Kotlin Multiplatform. This would provide room for even more code sharing, while keeping the native application approach.

Technology stack

iOS:

- Swift and Swift Package Manager
- UIKit
- CoreData and GRDB (SQLite)
- Alamofire
- CI/CD: GitHub Actions
- Go
- TestFlight
- Sentry
- OneSignal
- GetStream

Android:

- Kotlin
- UIKit
- Dagger2 + Hilt
- RxJava3
- AndroidX
- Firebase
- Exoplayer
- Antlr4
- Sentry
- GetStream
- Lottie

Time Span and Resources

Duration: 3 years, and counting

Effort: 9000 man-hours

https://www.objectstyle.com/case-studies/developing-white-label-mobile-apps-for-a-client