# Kotlin native multiplatform

Photo Agnes Vasarhelius Ben Usher & Agnes VasarheliiMy began using kotlin/Native and Kotlin multiplatform on PlanGrid as our first cross-platform initiative in early 2019. We have learned a lot, and now we are ready to share our experiences and conclusions using the Kotlin multiplatform! JetBrains' KotlinKotlin is now the preferred language for writing Android apps, and we use it for our Android apps in Autodesk. It has familiar modern language features such as zero security and extension features, and there are first-party libraries to support high-quality coding, such as corutins for asynchronous programming. Our Android team has been writing the PlanGrid Android app in Kotlin for years. Cotlin is also somewhat like Swift. The PlanGrid iOS team wrote Swift for a while, so we could all understand Cotlin reading it for the first time. And after some Cotlin Coan, the iOS team wrote to Cotlin, a ✅ of our veteran Cotlin Android peers. Sharing the CodeIn PlanGrid software ecosystem, we have Android, iOS, Windows and web clients talking to the same servers. In particular, their own mobile and desktop customers support almost the same set of features and standalone features. They all implement the same infrastructure for loading, analyzing, storing and downloading building data that all need to navigate through a standalone synchronization system. All this logic should be the same for all native customers. To make sure we want to share this core sync infrastructure and logic across platforms with a shared library. Kotlin/Native is now one of the many cross-platform solutions out there. PSPDFKit uses, for example, an approach that is not Kotlin. But, there were a few important details that prompted us to Kotlin:Kotlin/Native and Kotlin MultiplatformKotlin/Native is the LLVM server for the Kotlin compiler, implementing and creating our own codes using the LLVM toolkit. Kotlin/Native is a toolkit that allows us to staple kotlin code to a binary file that runs on native (i.e. not JVM/JS) platforms. Kotlin Multiplatform (called MPP for multiplatform project) is a language-level model for building Kotlin for multiple platforms. We write our Kotlin code into the Kotlin multiplatform project shared between our mobile teams. Using a hail build system in tandem with the multiplatform gradle Kotlin plugin, we've set up gradation tasks in our project that compile, test and publish for multiple purposes - one for each architecture/platform we support (Android, iOS and Windows). We create the following artifacts when creating our project for all platforms: An .aar for androidA .framework with Objective-C headers for iOSA .dll with windowsonce headers that they are published inside the PlanGrid network, we use them in their respective clients like any other 3rd Party library in our respective projects. Moving FastKotlin/Native is a new project. Things break down and quickly snap that familiar if you were an early adopter of Swift: Sometimes there are compiler errors (meaning that you use all good features, right?) Every time a new version is published, all 3rd Party libraries that support multiplatform form should be re-published (expect improvement here in Kotlin 1.4)Sometimes you bump into the edges of a new compilerDor, although that's what Kotlin/Native people are super responsive to problems, especially if you're giving small reproducible sample cases! 😎 there is also a kotlinlang slack, which has been very useful for getting help from both the community and JetBrains. GradleOne of the biggest struggles working with the multiplatform Kotlin right now is getting to know the Gradle infrastructure that makes it work. Gradle is a great tool, but if you are not new to it (ie you have a background in developing iOS or Windows or otherwise avoided writing grading tasks), it can be very painful to jump without a tutorial. There are many great resources out there though when faced with difficulties:What to share? At PlanGrid, our goal is to share code that allows us to share synchronization logic across all platforms. To get there, we decided to keep our library super focused and avoid solving more specific platform issues like shared user interface and networking. NetworkingTypically, to get one model (i.e. an instance of a user object) to exist on a client, an engineer will implement the same code on platform A in X, and another customer engineer will do the same on platform B Y. Except sometimes, they will implement it differently, which will cause differences in product specification, possible differences in customer capabilities, and ultimately, confused customers. To avoid all this, as well as make the implementation of new models/features faster, we would like to share all the code needed to download, analyze, save and download these types of models to all native customers. We do not share the actual network code as when making requests to servers. It remains the responsibility of client applications to transfer a request compiled in a shared code to any network library they use on different platforms. We decided to avoid jointly implementing the network to help us quickly begin implementing the logic of joint synchronization. The network can be very specific to the platform and comes with many sub-problems in the context of the mobile application:Knowledge of the current API environment (e.g. staging against production)Recorded in user credentials (storage and processing)Platform network layers Are problems that our applications have already solved, and we were happy to postpone the work on joint implementation at a later date. User interface codeY don't want to share user interface code. BrowsingModels? Perhaps. Rendering is often platform-specific, and each has its own UI frameworks. Sharing UI code will require a level of integration that we're not ready to take on early on in our multiplatform Kotlin adventure. Database diagram and and PlanGrid, our applications are focused on synchronizing large amounts of construction data. The data/model sharing schemes and business logic around this data provided the most natural fitness for the initial attempt to share code between our native mobile customers. All this logic and scheme is replicated in native languages for each mobile platform, but most of it is platform-agnostic. We use the square SQLDelight library to share the sqlite database layer. Difficulties, as always, have some problems, especially starting. Breaking the changes disrupting GitHub PR as a break-in change With each native customer engineer working on a single shared code, we started breaking things for each other all the time. We support our shared library in a separate repository, so you may not realize that you broke an API that uses a different platform until an engineer from that team went to update the library later. We expect this to get better as our process stabilizes and the library matures. At the same time, we've implemented several processes to help improve this: PRs are marked as broken, backward compatible or only internal, so you can easily look back at recent rip-off PRs to see what's changed. We'll have to do the checks that build each platform's app with the latest version of the library so we can get ahead of the library update before unsuspecting colleagues try to update the library to integrate their own changes. Mark the Kotlin API as internal as often as possible so that APIs can start from a position where they can be changed without breaking their customers.😱 failed to verify the compilation! Now Agnes knows she has to go upgrade and fix the iOS client. Kotlin/ Native, Concurrency, and CoroutinesEarly on, we knew we wanted to use Cotlin's corutin function to do ashink work. We found that this is a great way to manage simultaneously. And thanks to the structured simultaneousness you get with Cotlin's corutins, our asic tests rarely flak compared to similar tests in the Swift/iOS ecosystem. However, the kotlinx.coroutines library does not have full multi-threaded corutin support for Kotlin/Native, so we wrote and open source coroutineworker to allow us to use corutins in a shared library. In addition, the Kotlin/Native simultaneous model is completely different. Tl;dr is that objects are either volatile and belong to a single branch, or they are frozen (unchanged) and can be shared through streams. We could write a whole blog just about this topic, but we'll set aside for many who are already there, as this one kevin on Touchlab.ConclusionMa started by moving data of one function and business logic to our shared Kotlin library. Despite some bumps, the experience has been positive enough that we plan to move all of our schemes, sync logic and much of our business logic to our shared Kotlin library. We are also pleased JetBrains already has a preview available for multi-threaded Corutins support for Kotlin/Native, and Kotlin Kotlin promises to bring a lot of Polish to Kotlin multiplatform. We can't wait to see how the ecosystem develops in 2020! 📈 📈