How to improve iOS build times with modularization

APP DEVELOPMENT

iOS

Bruno Rocha

MARCH 28, 2024

Most mobile teams understand and appreciate the benefits of fast build times. Being able to quickly compile and test your code means quicker development and iteration, which in turn allows your team to ship more regularly and efficiently. But actually achieving fast build times, and

are relatively trivial — reducing the size of bundled resources, for example — others can be much more involved, or even kind of dangerous (think questionable compiler tricks)!

Luckily, the riskiest build time improvement techniques are rarely necessary. Instead, most teams will achieve build time success by implementing some of the more common approaches incrementally as their codebase grows and resources scale.

Modularization is one such approach that teams can gradually adopt to improve and stabilize build times. It establishes an architecture that allows scaling applications and codebases to grow while keeping build times in check. In this post, we'll look at how all this plays out in the iOS world (though there are certainly themes common to all platforms here). We'll first examine what contributes to slow build times and how modularization can help. Then, we'll zoom in on the more advanced **API/Impl modules** technique which can help you achieve even quicker compilation of interdependent modules and faster build times for apps containing hundreds of modules.

implementing a specific fix once builds get slow; it requires an understanding of what causes bottlenecks in the compiler and preemptive changes to the architecture of the project to address those causes. *Where* those bottlenecks exist and *how* they came about in the first place can vary dramatically from project to project. In order to anticipate future issues, it's important to first understand how iOS apps typically evolve. Let's start with the example of a very simple single-view app — we can then examine the kinds of problems it will face as it grows, and what we can do to solve them.

The evolution of an iOS app and impact on build times

The single-module app

Let's say we're creating a note-taking app, and after some weeks of development, we've built a simple app such that the whole app is contained within a single module in the main app target:

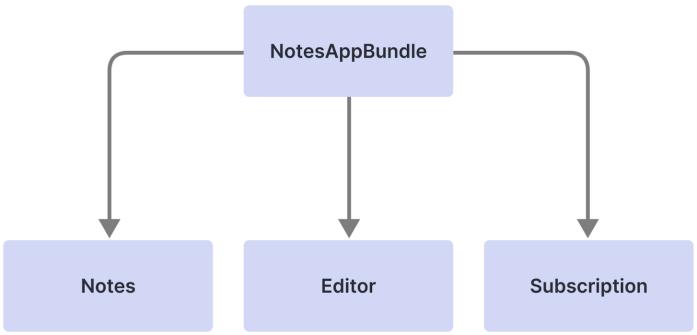
We know nothing about modularization, and we also haven't been particularly intentional about the architecture of the app. Because our app only has a handful of simple features, Apple's out-of-the-box MVC architecture fits the bill just fine. This is how most beginners usually develop their first few apps, and even how seasoned developers will still spin up apps that are relatively modest in scope.

While single-module apps are naturally some of the easiest to maintain, their architecture is one of the worst when it comes to build times. When everything is packed together into a single module, changing **anything** in the codebase results in **everything** being recompiled, even if the change is small or in an area of code that's otherwise decoupled from the rest of the codebase. While this doesn't pose a problem for small and simple apps, for larger, more complex apps, build times take massive performance hits with this approach.

The small multi-module app

Thankfully, a better approach presents itself immediately: modularization, or the breaking

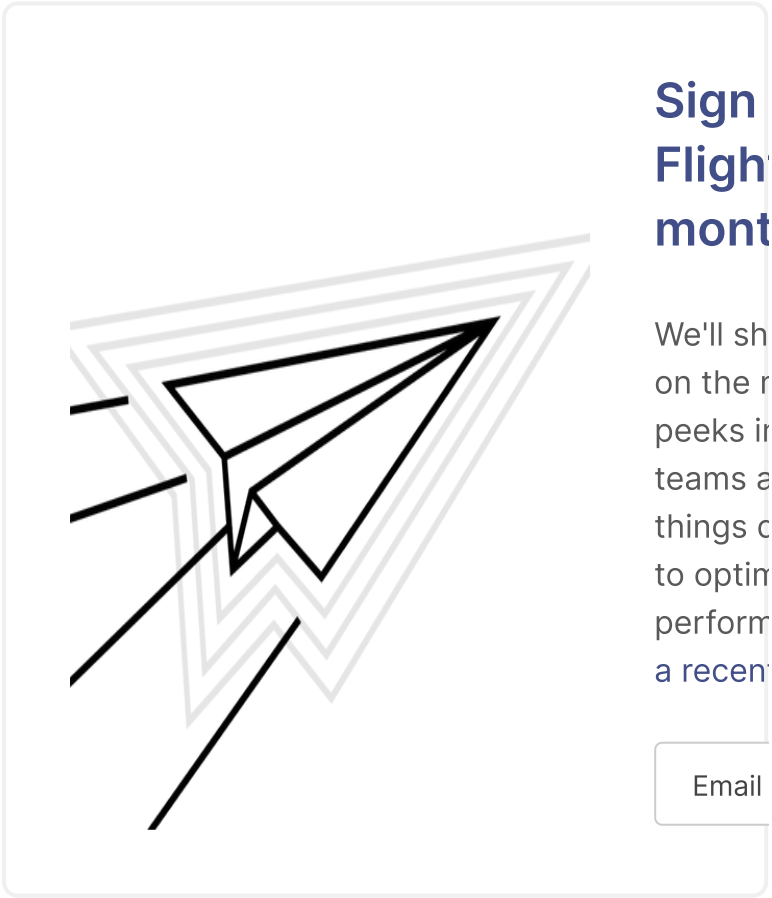
divide that up into several smaller, self-contained modules that are linked together in the app's bundle. Any time one module needs to reference code or functionality from another module, it imports it, creating inter-module dependencies. In our notes app example, imagine that each of the three main pieces of functionality for the app (*Notes, Editor, and Subscription*) is broken out into its own module:



The overall functionality of the app remains the same, but better modularization of the codebase allows the compiler to be smarter about what needs to be recompiled every time a change is made. In this example, a change to the *Notes* module won't require recompilation of the *Editor* and *Subscription* modules because there is no dependency between them. This allows Xcode to reuse a cached compilation of the unaffected modules, resulting in a significant reduction in overall build time.

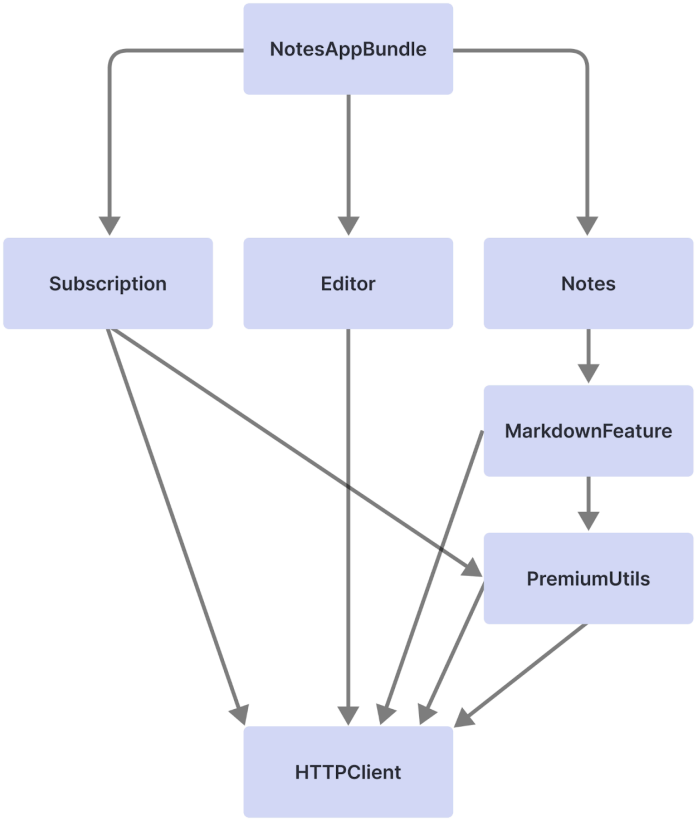
Xcode to that of the particular module you're working on means that it alone gets built, speeding up development significantly if you're making changes that don't require actually launching the app.

Nowadays, most medium-to-large iOS apps out there have this sort of multi-module architecture, and it's one of the most common and clearly-defined approaches for tackling slow build times as the size of the codebase increases.



The large multi-module app

implemented. Most real-world apps won't have simple dependency graphs like the one in our previous example, and especially as an app grows and more modules are added, the dependency graph becomes more and more complex:

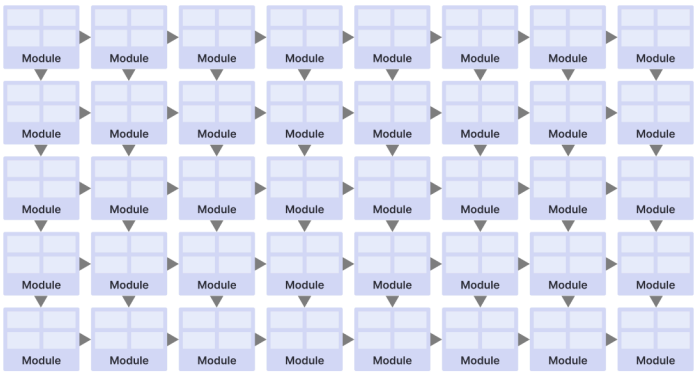


As an app is broken down into more and more modules, and as entirely new modules are added, there will inevitably be more interdependencies between all of them. And more of these transitive dependencies leads to less effective modularization overall. Incremental builds no longer only recompile the module that changed — they also

something that **depends** on that module, then that module will **also** be invalidated, along with any other modules that depend on it in turn. As a result, apps that are heavily modularized but have complex dependency graphs will soon run into build time issues once again.

Returning to our notes app example – three of its modules (*Notes*, *Editor*, and *Subscription*) have nothing depending on them, so they'll continue to compile quickly when changes are made to any of them. But changes to any other module in the notes app will entail invalidation and recompilation not just of that module itself, but of all other modules that depend on it as well. Take the new *HTTPClient* module for example: because *all* the other modules depend on it (either directly or indirectly), any changes made to it will cause the **entire app** to be recompiled, even if the changes are not at all relevant to or needed by those other modules!

In a small project with only a dozen or so modules, interdependencies between modules aren't a huge problem, but as your app grows to contain hundreds of modules, the effect all this recompilation has on build times becomes increasingly noticeable and unsustainable. Imagine the importance for



Both build times in general, and this modularization “gotcha” in particular, represent a very common and acute problem area in large iOS projects. Many large companies for whom developer happiness and productivity is an important concern devote significant resources to addressing it – even going as far as forming dedicated platform teams responsible for modularization and architecture improvements to keep build times in check.

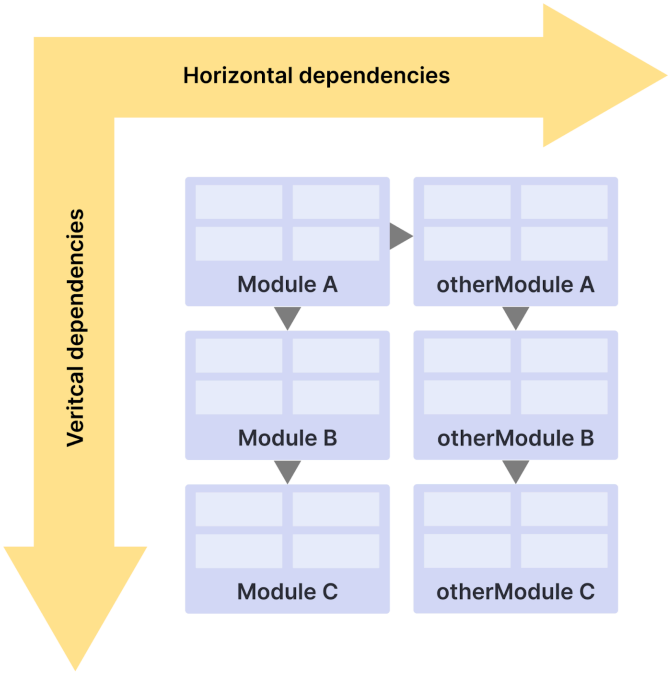
And at this scale, keeping build times in check means fundamentally changing how you think about modularization.

Modularization at scale: moving from vertical to horizontal dependencies

Let's assume that our fictional notes app kept growing and is now in this mega-modularized situation. We now have close to a hundred

the presence of so many **transitive dependencies**. Of particular concern are so-called **vertical dependencies:** any given module "stacks up" dependencies not just on modules it itself depends on, but also on any modules *those* modules depend on in turn– even if that first module on the bottom of the stack has no direct need for those others.

To improve things, we need to minimize this type of vertical dependency, in effect reducing the number of "stacked" dependencies each module has. The goal is to make our dependency graph as **horizontal** as possible.



By reducing the number of downstream dependencies of each module, we minimize

without affecting their functionality?

API/Impl Modules

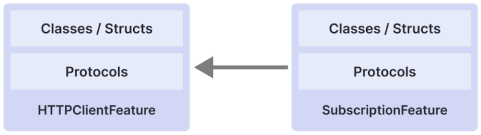
The key to solving the vertical dependency problem is realizing that **you don't need explicit dependencies** to access functionality across different modules. By building a way to reference the *functionality* of a module without explicitly depending on it, we can start to minimize the number of cross-module dependencies, simplifying our dependency graph and keeping build times under control.

But how is this accomplished? If there is no dependency between two modules, how can one access the functionality of the other?

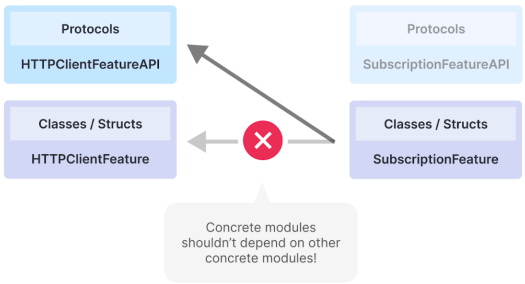
To answer that, we need to change how we think about dependencies. The inherent problem isn't about *depending* on other modules, but rather about *depending on other modules which change often.* Modules that contain **concrete implementations** of functionality are more prone to frequent changes and carry with them additional dependencies. By contrast, modules containing only APIs (defined as protocols or extensions, for example) change more rarely, and can be easily decoupled from their implementation – meaning they should be

APIs from their implementation and applies it to modularization, such that a module's API is broken out into a **separate** API module. If a different module needs to import some functionality, instead of referencing the functionality's concrete implementation module, it references the API module instead. **Importing concrete implementation modules should be forbidden, with no exceptions,** in order to prevent costly dependencies across implementation modules from forming.

Before



After

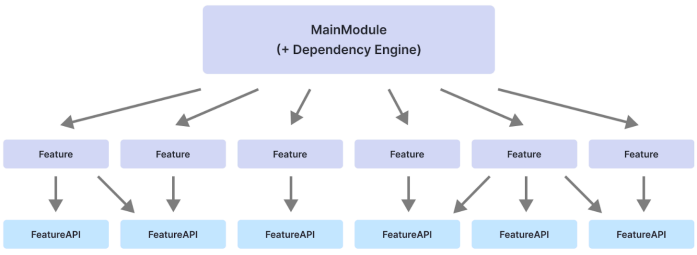


Architecting modules this way leads to considerably faster incremental build times because the majority of a codebase's changes will be happening in the concrete implementation modules – which have been decoupled from the rest of the codebase in

But one last mystery remains — someone, somewhere, does have to reference the otherwise-decoupled concrete implementation modules. Who, and how?

Dependency injection

A key prerequisite for this kind of architecture is the existence of a **dependency injection framework**. Where previously modules were referencing other modules directly, dependency injection can be used instead to coordinate module interdependencies via a central engine. The dependency injection engine knows of every single module that the app contains, and injects dependencies where needed across the app.



Although the concrete modules cannot depend on each other, there's always going to be a module that imports everything: the “lowest” module in the dependency graph, where the AppDelegate is most likely going to be. When the app first launches, the

At a high level, when a module requires access to a feature in another module, it can fetch it by grabbing the equivalent "referenceable" protocol in that feature's API module and sending a request back to the dependency injection engine. Since the engine holds a reference to all modules in the app, it can locate the correct feature module which provides a concrete implementation of that particular API, and inject it back into the module that requested it.

Most large companies build their own dependency injection frameworks, but there are a few open-source options out there, like Swinject, and some proof-of-concept frameworks with example projects, like RouterService (written by me!). These are a great place to start to explore different approaches for implementing dependency injection in your codebase.

Making changes to the dependency injection engine’s module will always result in the longest build times,as it will invalidate the entire application. But by keeping the dependency injection module as small as possible and isolating all of the app's logic into separate (horizontal) modules that reference other modules only through their feature APIs, you can achieve a great build

Keep build times in check as you grow, with an incremental approach to modularization

We've seen how modularization, when correctly implemented, can be a powerful tool for improving slow build times in your app as it grows from a simple project to a more complex one made up of lots of features that depend on each other. Although it might seem daunting, reaping the benefits of modularization doesn't necessarily mean spending months rearchitecting your entire codebase; modularization allows for a more incremental approach whereby pieces of functionality are broken off into their own modules slowly, over time. And eventually, once you reach the point of having too many interdependent modules, the API/Impl modules approach to modularization can provide significant build time improvements even at hundreds-of-modules scale. Many large companies with huge codebases (like Spotify!) have successfully used this technique to improve build times and boost productivity for the entire team as a result.

Runway integrates with all the tools you're already using to level-up your release coordination and automation, from kickoff to release to rollout. No more cat-herding, spreadsheets, or steady drip of manual busywork.

CONTACT SALES GET STARTED

RELATED POSTS

The 'what' and 'why' of iOS signing certificates and provisioning profiles — and how to manage them as your team grows

Bruno Rocha

MAY 26, 2022

Avoiding release anxiety, part II

Bruno Rocha

MARCH 14, 2022

Runway's next big step, and the funding to take us there

NOVEMBER 16, 2021

SIGN UP FOR THE FLIGHT DECK, OUR MONTHLY NEWSLETTER

Email SIGN UP

TRUSTED BY THE BEST MOBILE TEAMS

End-to-end automation
Rollouts
Mobile insights
Build Distro
Automations
Security
Pricing
What's new
Explore sandbox

Use cases

fastlane
Release trains
Mobile DevOps
Cross-platform
- React Native
- Flutter

- Linear
- Pivotal Tracker

Version control
CI/CD
App stores
- App Store Connect
- Google Play Console

Slack
Monitoring
All integrations

Documentation
Quickstart CI/CD
App review times
App Store Connect status page
App hotfix leaderboard

Company

About us
Contact
Terms of service
Privacy policy
Careers
Status