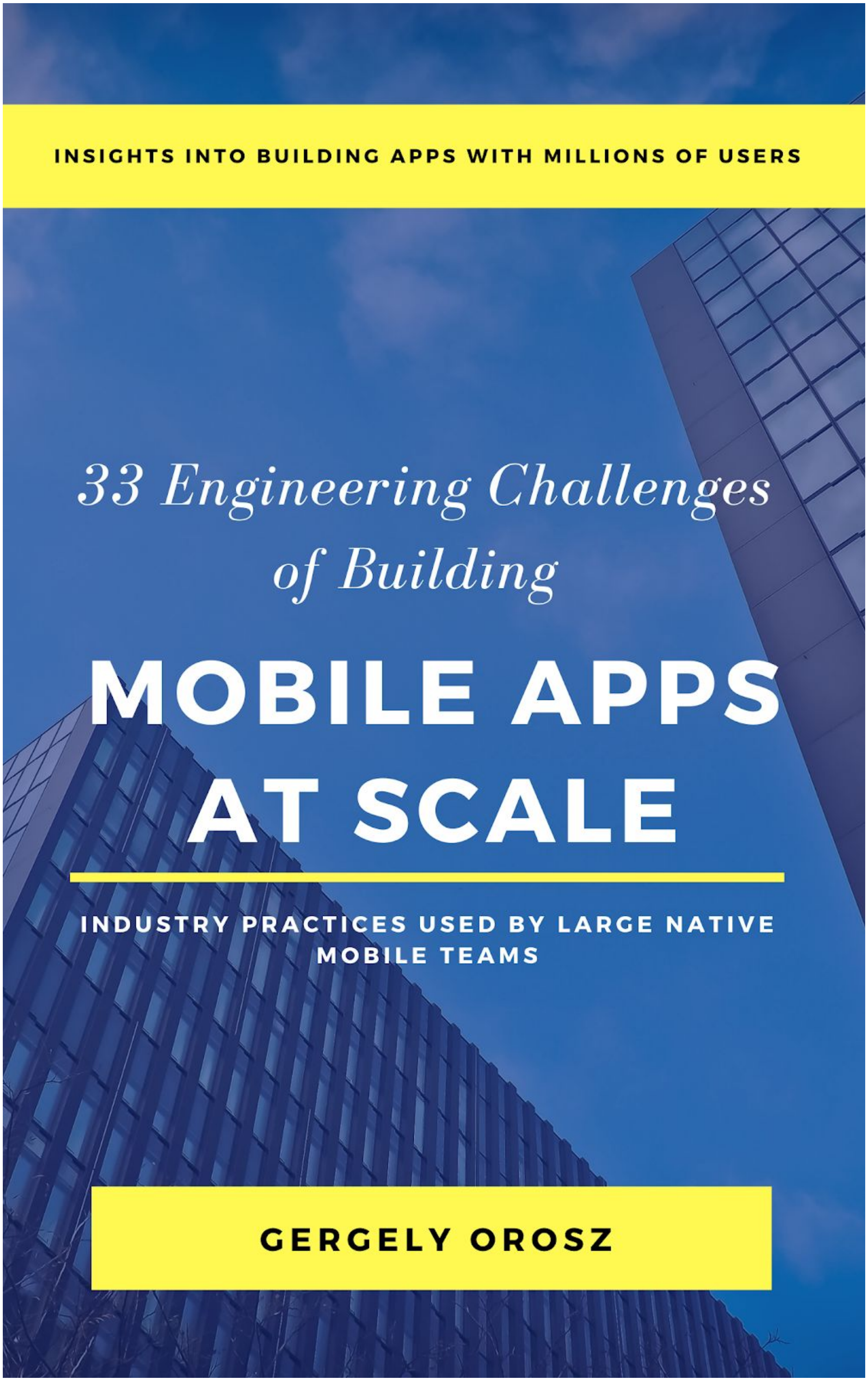
INSIGHTS INTO BUILDING APPS WITH MILLIONS OF USERS

33 Engineering Challenges of Building MOBILE APPS AT SCALE

INDUSTRY PRACTICES USED BY LARGE NATIVE MOBILE TEAMS

GERGELY OROSZ



33 Engineering Challenges of Building Mobile Apps at Scale

Table of Contents

⚠ Under Construction ⚠

This book is still being written. I am releasing this version earlier both to get feedback, and to get it in your hands earlier. This way, we both win!

This is the first release from the book release schedule. If you purchased bonus content, I'll ship that content at later dates.

Please, send me feedback with corrections, suggestions, typos and missing parts to scale@pragmaticengineer.com. Thank you, and enjoy the contents!

Foreword

I've been noticing that while there's a lot of appreciation for backend and distributed systems challenges, there's a lot less empathy for why mobile development is hard when done at scale. Building a backend system that serves millions of parallel customers means building highly available and scalable systems and operating these reliably. But what about the mobile clients for the same systems?

Most engineers - who have not built mobile apps - assume the mobile app is a simple facade that requires less engineering effort to build and operate. Having built both types of systems: this is not the case. There is plenty of depth in building large, native, mobile applications - but often little curiosity from people not in this space. Product managers, business stakeholders, and even non-native mobile engineers rarely understand why it "takes so long" to ship something on mobile.

This book collects challenges engineers face when building native iOS and Android apps at scale: scale meaning they have a large number of users, are built by large teams, launch features continuously, and need to operate reliably. It's a summary of the current industry practices used by large native mobile teams.

I hope this book helps non-mobile engineers build empathy for the type of challenges and tradeoffs mobile engineers face and be a conversation starter between backend, web and mobile teams.

About the Author

I've been building native mobile apps since 2010: starting on Windows Phone, later on iOS, and Android. Starting from one person apps, I worked with small teams at Skyscanner, to hundreds of engineers working on the same codebase at Uber.

At Uber, I've been part of the Rider app rewrite, Driver app rewrite, both projects involving hundreds of mobile engineers. The apps my team worked on had 100M monthly users in 60+ countries, with several features built for a single country or region.

I've been working on this book - originally meant to be a blog article - on-and-off, for over a year. A Twitter post gave the final push to finish the contents off. The contents were too long for a single post. This is how this book was born.

You can read more of my writing on The Pragmatic Engineer, connect on LinkedIn or on Twitter. Reach me at scale@pragmaticengineer.com.

Acknowledgements

The book has been written with significant input and reviews of over 20 mobile engineers and managers - many of them deep experts in their respective fields. Thanks very much to all of them. If you are on Twitter, I suggest you follow them:

- Abhijith Krishnappa (Halodoc)
- Andrea Antonioni (Just Eat)
- Artem Chubaryan (Square)
- Barisere Jonathan
- Corentin Kerisit (Zenly)
- Franz Busch (Sixt)
- Guillermo Orellana (Monzo, Skyscanner, Badoo)
- Injy Zarif (Convoy, Microsoft)
- Jake Lee
- Javi Pulido (Plain Concepts)
- Jared Sheehan (Capital One)
- Julian Harty
- Matija Grcic
- Michael Sena (Amazon)
- Patrick Zearfoss
- Robin van Dijke (Uber, Apple)
- Rui Peres (Sphere, Babylon Health)
- Tuğkan Kibar
- Will Larson (Calm, Stripe, Uber)

When Things are Simple

Let's address the elephant in the room: the frequent assumption that client-side development is simple. The assumption that the biggest complexity lies in making sure things look good on various mobile devices.

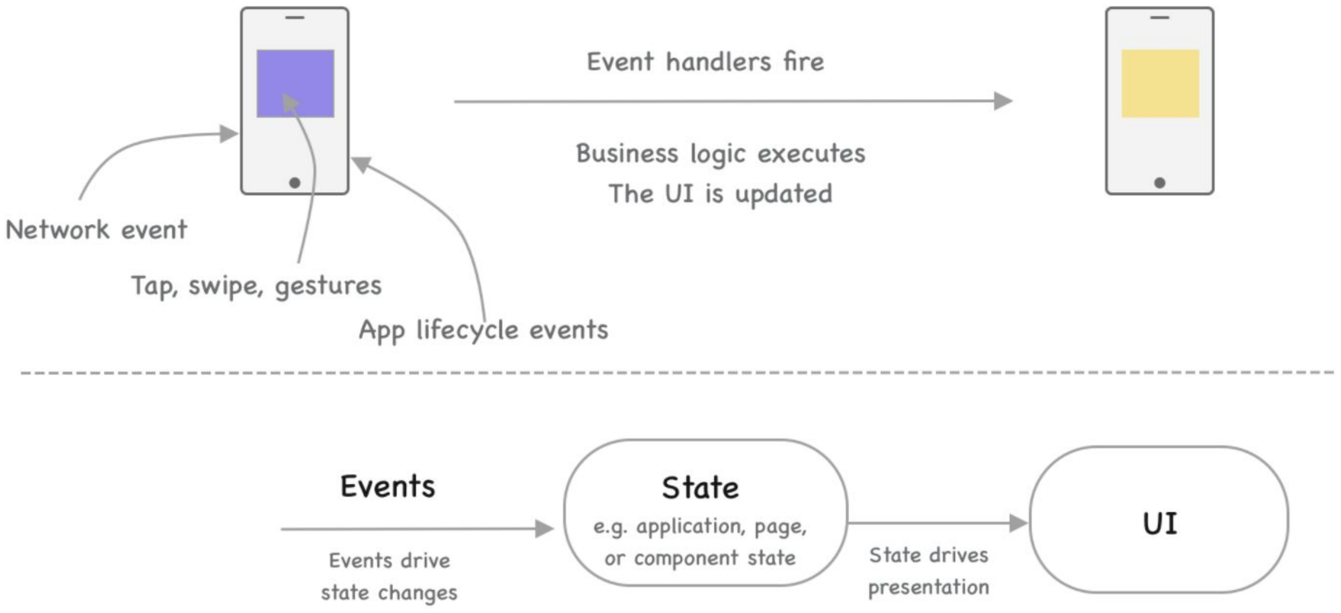
When the problem you are solving is simple, and the scope is small, it's easier to come up with simple solutions. When you're building an app with limited functionality with a small team and very few users, your mobile app shouldn't be complicated. Your backend will likely be easy to understand. Your website will be a no-brainer. You can use existing libraries, templates, and all sorts of shortcuts to put working solutions in place.

Once you grow in size - customers, engineers, codebase, features - everything becomes more complex, more bloated, and harder to understand and modify: including the mobile codebase. This is the part we'll focus on in this article: when things have become complex. Once your app has grown, there are no silver bullets that will magically solve all of your pain points, only tough tradeoffs to make.

PART 1: Challenges Due to the Nature of Mobile Applications

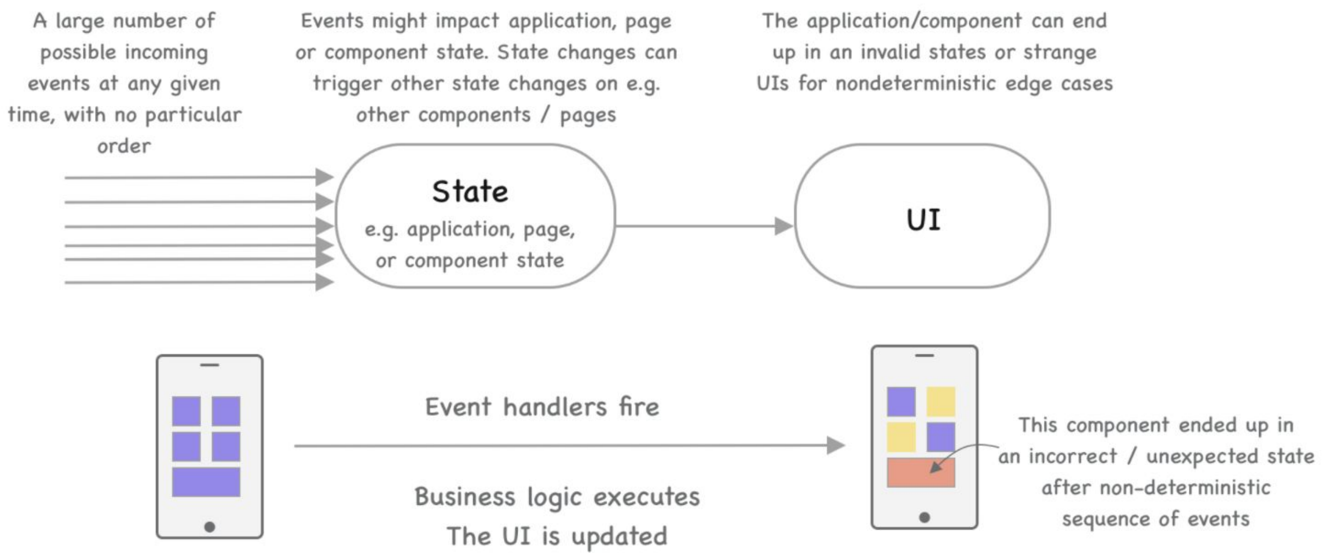
1. State Management

State management is the root of most headaches for native mobile development - similar to modern web and backend development. The difference with mobile apps is how app life cycle events and transitions are not a cause for concern in the web and backend world. Examples of the app-level lifecycle transitions are the app pause and going to the background, coming back to the foreground or being suspended. The states are similar, but not identical for iOS and Android.



Events driving state changes in mobile apps

Events drive state changes in most mobile apps. These events trigger in an asynchronous way - application state changes, network requests, user input. Most bugs and unexpected crashes are usually caused by an unexpected or untested combination of events and the application's state getting corrupted. State becoming corrupted is a common problem area with apps where global or local states are manipulated by multiple components unbeknown to each other. Teams that run into this issue start to isolate component and application state as much as possible and tend to start using reactive state management sooner or later.


A large number of possible incoming events at any given time, with no particular order
Events might impact application, page or component state. State changes can trigger other state changes on e.g. other components / pages
The application/component can end up in an invalid states or strange UIs for nondeterministic edge cases
State
e.g. application, page, or component state
UI
Event handlers fire
Business logic executes
The UI is updated
This component ended up in an incorrect / unexpected state after non-deterministic sequence of events

A common root reason for exotic bugs in complex apps: non-deterministic events put parts of the app in invalid states

Reactive programming is a preferred method to deal with a large and stateful app to isolate state changes. You keep state as immutable as possible, storing models as immutable objects that emit state changes. This is the practice used at Uber, the approach Airbnb takes, or how N26 have built their app. Though the approach can be tedious in propagating state changes down a tree of components, the same tediousness makes it difficult to make unintended state changes in unrelated components.

Applications sharing the same resources with all other apps and the OS killing apps on short notice is one of the biggest differences between developing for mobile versus developing for other platforms - like backend and the web. The OS monitors CPU, memory, and energy consumption. If the OS determines that your app is taking up too many resources - may this be in the foreground or the background - then it can be killed with little warning. It is the app developer's responsibility to react to application state changes, save state, and restore the app to where it was running. On iOS, this means handling app states and transitions between them. On Android, you need to react to changes in the Activity lifecycle.

Global application state - permissions, Bluetooth and connectivity state, and others - brings an interesting set of challenges. Whenever one of these global states changes - for example, the network connectivity drops - different parts of the app might need to react differently.

With global state, the challenge becomes deciding what component owns listening to these state changes. On one end of the spectrum, application pages or components could listen to global state changes they care about - resulting in lots of code duplication, but components handling all global state concerns. On the other end, a component could listen to certain global state changes and forward these on to specific parts of the application. This might result in less complex code, but now there's a tight coupling between the global state handler and the components that it knows of.

App launch points like deeplinks or internal "shortcut" navigation within the app also add complexity to state management. With deeplinks, additional state might need to be set up after the deeplink was activated. We'll go into more detail in the Deeplinks section.

2. Mistakes Are Hard to Revert

Mobile apps are distributed as binaries. Once a user updates to a version with a client-side bug, they are stuck with that bug until a new version is released and this user updates.

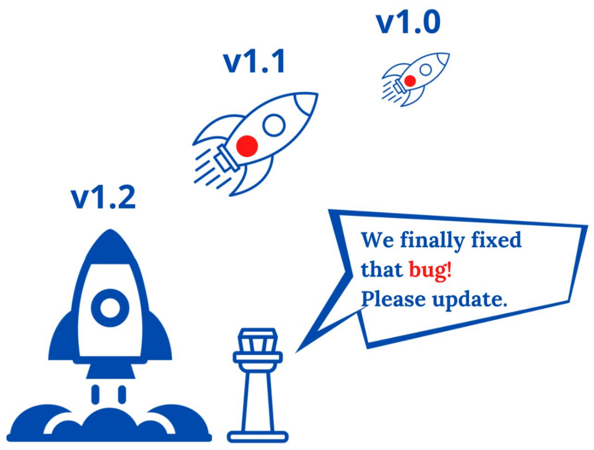
Multiple challenges come from this approach:

- **Apple does not allow updating native code on the fly.** Interestingly enough, they do allow this with non-native code like JavaScript: and this is why solutions where business logic is written using JavaScript, bug fixes are pushed to the client are gaining popularity. Solutions include React Native or Cordova with services like Codepush are gaining strong traction. At Uber, we've built a homegrown solution among the same lines, as several other companies have done so.
- **It takes hours to days to release** a new app version on the store. This is more pronounced for iOS, where a manual app review needs to take place. Historically, every review had the possibility of rejection. As of June 2020, Apple has changed guidelines, so bug fixes are no longer be delayed over guideline violations, except for legal issues.
- **Users take days to update to the latest version** after a new version is published to the app store. This lag is true even for users with automated updates turned on.
- **You can not assume that all users will get this updated version, ever**. Some users might have automated updates disabled. Even when they update, they might skip several versions.

Chuck Rossi, part of release engineering at Facebook, summarizes what it's like to release for mobile on a Software Engineering Daily podcast episode like this:

> *"It was the most terrifying thing to take 10,000 diffs, package it into effectively a bullet, fire that bullet at the horizon and that bullet, once it leaves the barrel, it's gone. I cannot get it back, and it flies flat and true with no friction and no gravity till the heat death of the universe. It's gone. I can't fix it."*

This means that all previous versions of your app need to be supported indefinitely - at least in theory, you should do this. The only exception is if you put homegrown controls in place and build a force update mechanism to limit the past versions to support. Android supports in-app updates through the Play Core library. iOS doesn't have similar native support. We'll cover force updates in the "Force upgrading" section in Part 3 of the series.



Fixing a bug on mobile

Assuming that you have an app with millions of users, what steps can you take to minimize bugs shipped to users or regressions happening in old versions?

- **Do thorough testing** at all levels. Automated testing, manual testing, and consider beta testing with easy feedback loops. A common approach at many companies is releasing the beta app to company employees and beta users and having it "bake" for a week, collecting feedback on any issues.
- **Have a feature flagging system** in place, so you can revert bugs on the fly. Still, feature flags add further pain points - we'll discuss these points in the "Feature flag hell" section in Part 3 of the series.
- **Consider gradual rollouts**, with monitoring to ensure things work as expected. We'll cover monitoring in more detail in Part 3 of the series.
- **Force upgrading** is a robust solution - but you'll need to put one in place, and some customers might churn as a result.

3. The Long Tail of Old App Versions

Old versions of the app will stay around for a long time - up to a number of years. This timeframe is only shorter if you're one of the few teams that put strict force app upgrade policies in place. Apps that have a rolling window of force upgrades include Whatsapp and Messenger. Several others use force upgrades frequently, like banking apps Monzo or American Express.

While most users will update to new app versions in a matter of days, there will be a long tail of users being on several versions behind. Some users disable automatic updates on purpose, but many who don't update are blocked because of old phones or OSes. At the same time, old app versions are unlikely to be regularly tested by the mobile team: it's a lot of effort, with little payoff.

Even a non-breaking backend change can break an older version of the app - such as changing the content of a specific response. A few practices you can do to avoid this breakage:

- **Build sturdy network response handling and parsing, using dedicated tooling** that solves for these problems. Prefer strongly typed, generated contracts between client and backend like Thrift, GraphQL, or other solutions with code generation over REST interfaces that you need to validate manually - which is bound to break when someone forgets to update the parsing logic on mobile.
- **Plan well in advance for breaking backend changes.** Have an open communications channel with the backend team. Have a way to test old app versions. Consider building new endpoints and not retiring old ones until a forced upgrade moves all current app users off the old endpoint.
- **Version your backend endpoints** and create new versions to accommodate breaking changes. Note that in case of using GraphQL, GraphQL has a strong stance against versioning. When making breaking changes, you'd usually create a new endpoint and mark the existing one as deprecated.
- **Take care when deprecating endpoints** on the backend: monitor the traffic, and have a migration plan on how to channel requests, if needed.
- **Track usage stats on an app version level.** What percentage of users is lagging three or more versions behind? Once you have this data, it's easier to decide how much effort to dedicate towards ensuring the experience works well on older versions.
- **Put client-side monitoring and alerting in place.** These alerts might be channeled to a dedicated mobile oncall, or just the normal oncall. We'll dive into monitoring and alerting in more detail in Part 3 of the series.
- **Consider doing upgrade testing**, at least for major updates. Upgrade testing is expensive, hard to automate, and there might be several permutations to try. Teams rarely do it because of this overhead.

4. Deeplinks

Deeplinking - providing a web or device link that opens a part of the app - becomes a surprisingly tricky problem on both mobile platforms. Both iOS and Android offer APIs to deal with this, but without any opinionated native frameworks or recommended approaches. As Alberto De Bortoli puts it in his excellent article deeplinking at scale, on iOS:

> *"Deep linking is one of the most underestimated problems to solve on mobile."*

There are a few things that make deeplinking challenging:

- **Backward compatibility:** ensuring that existing deeplinks keep working in older versions of the app - even when significant navigation or logic changes happened.
- **State problems when deeplinking to a running app with existing state**. Say you have an app open and are on a detail page. You tap on a deeplink in your mail app that points to another detail page. What should happen? Would the new detail page be added to the navigation stack, preserving your current state? Or should the state be reset? The solution that results in the least amount of non-deterministic behavior is to reset the app's state fully when receiving a deeplink. However, there might be flows that you don't want to break: plan carefully.
- **iOS and Android deeplink implementation differences**. Deeplink implementations are different for iOS (Universal links and URL schemes) and for Android (based on intents). There are third-party deeplink providers that provide abstractions to work with a single interface: Branch and Firebase Dynamic Links are such providers, among others.
- **Lack of upfront planning.** Deeplinks are often an afterthought after having shipped multiple versions of the app. However, unlike on the web where adding links/deeplinks are more straightforward, retrofitting a deeplinking strategy can be a real engineering challenge. Deeplinks are connected to state management and the navigation architecture (we'll discuss this area in Part 2 of the series).

The biggest problem with deeplinks is how neither iOS nor Android provides a much-needed opinionated approach on how to architect - and test - deeplinks. As the number of deeplinks grows, the complexity of keeping these deeplinks working as intended snowballs. You'll have to plan well ahead in building a sensible - and scalable - deeplink implementation.

5. Push and Background Notifications

App push notifications are a frequently used notification, communication, and marketing tool. The business loves to use push notifications, and as a developer, you'll be asked to support this communications method, sooner or later. However, push notifications bring a set of new challenges you'll need to tackle.

Setting up and operating push notifications is complex. Both for Android and iOS, your app needs to obtain a token from a server (FCM on Android, APNS on iOS), then store this token on the backend. There are many steps to take to get push notifications working: see this comprehensive tutorial for iOS and one for Android.

Sending push notifications has to happen from the backend: you'll need to work with the backend team on the type of notifications they want to send and their triggers. Your backend counterparts will have to become familiar with the mobile push notification infrastructure and capabilities to make the most out of this channel.

Users can opt out of push notifications or not opt in to start with. On iOS and Android, you have different ways - and limitations - in detecting when this is the case. Push notifications are usually a "nice to have" for many applications, exactly because you cannot guarantee that each user will opt into them - or that their device will be online to receive them.

Using push notifications together with emails and text messages is a popular strategy for marketing activities. If your app is used for marketing purposes, you'll almost certainly not implement push notifications from scratch. You'll use a third-party customer engagement service like Twillio, Airship, Braze, OneSignal, or similar.

Push notifications come with the same challenges as deeplinks for the action the notification performs. A push notification is a glorified deeplink: a message with an action that links into the app. Thinking about backward compatibility, state problems, and planning ahead all apply for push notifications as well.

Testing push notifications is another pain point. You can, of course, test this manually. However, for automated testing, you need to write end-to-end UI tests: expensive tests to create and to maintain. See this tutorial on how to do this for iOS.

Background notifications are a special type of push message that is not visible for the user, but goes directly to your app. These kinds of notifications are useful to sync backend updates to the client. These notifications are called data messages on Android and background notifications on iOS - see an example for iOS usage.

The concept of background notifications is handy for realtime and multi-device scenarios. If your app is in this area, you might decide to implement a cross-platform solution across iOS and Android, and instead of the mobile app polling the server, the server sends data through background push notifications to the client. When rewriting Uber's Rider app in 2016, a major shift in our approach was exactly this: moving from poll to push, with an in-house push messaging service.

Background notifications can simplify the architecture and the business logic, but they introduce message deliverability issues, message order problems, and you'll need to combine this approach with local data caching for offline scenarios.

6. App Crashes

An app crashing is one of the most noticeable bug in any mobile app - and often ones with high business impact. Users might not complete a key flow, and they might grow frustrated and churn or leave poor reviews.

Crashes are not a mobile-only concern: they are a major focus area on the backend, where monitoring uncaught exceptions or 5XX status codes is common practice. On the web, due to its nature - single-threaded execution within a sandbox - crashes are rarer than with mobile apps.

The first rule of crashes is you need to track when they happen and have sufficient debug information. Once you track crashes, you'll want to report on what percentage of sessions end up crashing: and reduce this number as much as you can. At Uber, we tracked the crash rates from the early days, working continuously to reduce the rate of crashed sessions.

You can choose to build your own implementation of crash reporting or use an off-the-shelf solution. Coming up to 2021, most teams choose one of the many crash reporting solutions such as Crashlytics, Bugsnag, Sentry, and others.

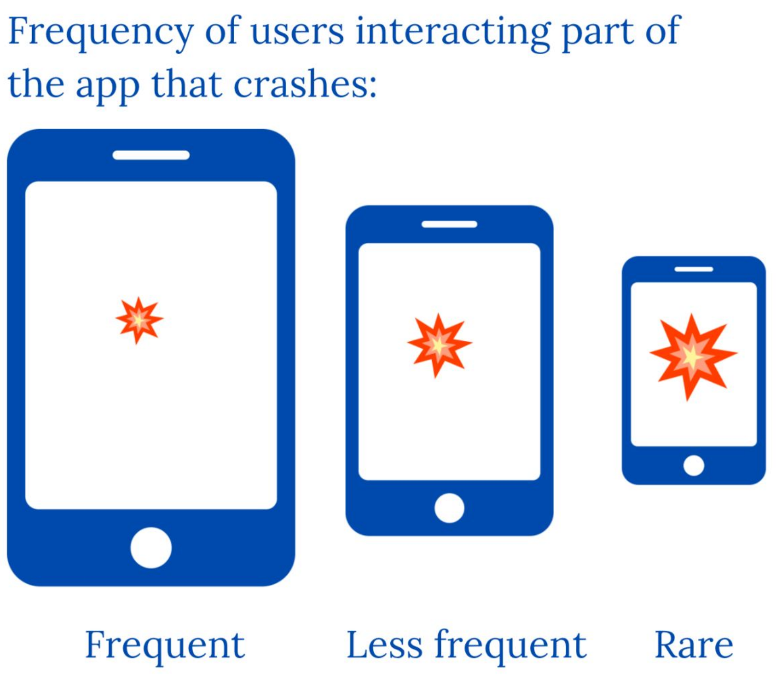
On iOS, crash reports are generated on the device with every crash that you can use to map these logs to your code. Apple provides ways for developers to collect crash logs from users who opted to share this information via TestFlight or the App Store. This approach works well enough for smaller apps. On Android, Google Play also lets developers view crash stack traces through Android Vitals in the Play Console. As with Apple, only users who have opted in to send bug reports to developers will have these crashes logged in this portal.

Third-party or custom-built crash reporting solutions offer a few advantages on top of what the App Store and Google Play have to offer. The advantages are plenty, and most mid-sized and above apps go with either a third party or build a solution with the below benefits:

- **More diagnostic information.** You'll often want to log additional information in your app on events that might lead up to a crash.
- **Rich reporting**. Third-party solutions usually offer grouping of reports and comparing iOS and Android crash rates.
- **Monitoring and alerting** capabilities. You can set up to get alerts when a new type of crash appears or when certain crashes spike.
- **Integrations** with the rest of the development stack. You'll often want to connect new crashes with your ticketing system or reference them in pull requests.

At Uber, we used third-party crash reporting from the early days. However, an in-house solution was built later. A shortcoming of many third-party crash reporting solutions is how they only collect health information on crashes and non-fatal errors, but not on app-not-responding (ANR) and memory problems. Organizations with many apps might also find the reporting not rich enough and might want to build their own reporting to compare health statuses across many apps. Integrating better with in-house project management and coding tools could also be a reason to go custom.

Reproducibility and debuggability of crashes are another pain point that impacts mobile more than backend or web teams. Especially in the Android world, users have a variety of devices that run a wide range of OS versions with a variety of app versions. If a crash can be reproduced on a simulator or on any device: you have no excuse not to fix the problem. But what if the crash only happens on specific devices?


Frequency of users interacting part of the app that crashes:
Frequent
Less frequent
Rare

How do you prioritize fixing a crash? Is a less "smaller" crash in a more frequently used part of the app more important to fix than a "larger" crash in a less frequently used part?

Put a prioritization framework in place *to define thresholds, above which you'll spend time investigating and fixing crashes. This threshold will be different based on the nature of the crash, the customer lifetime value, and other business considerations. You need to compare the cost of investigation and fixing compared to the upside of the fix, and the opportunity cost lost in an engineer spending time on something else, like building revenue-generating functionality.*

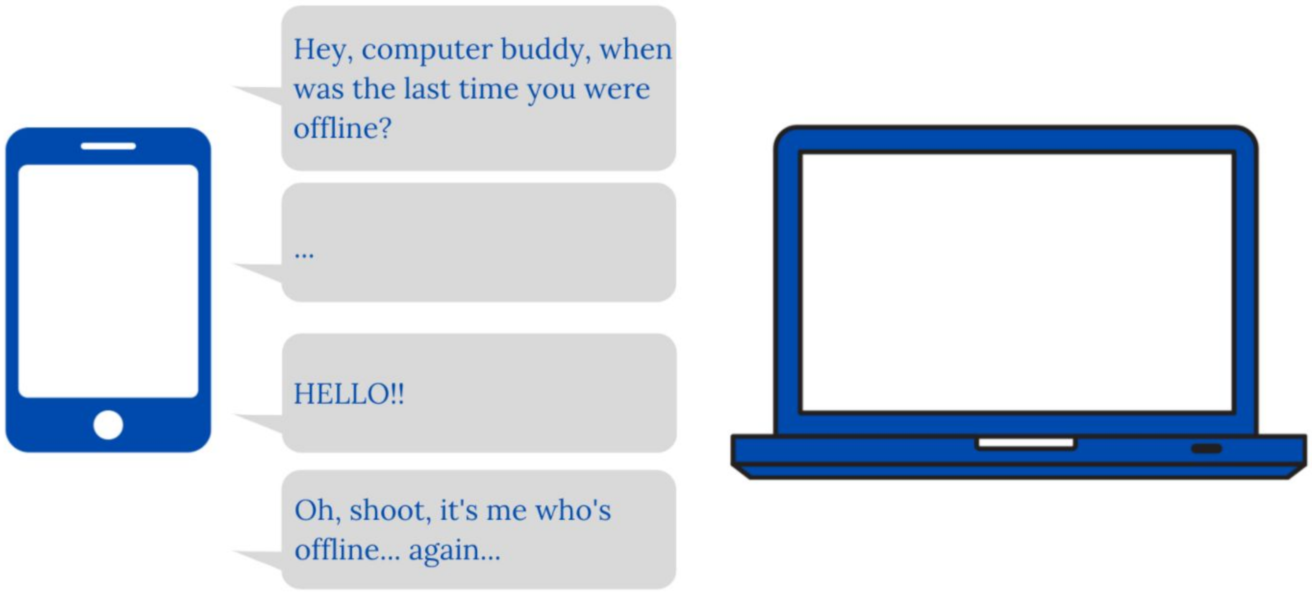
7. Offline Support

Though offline support is becoming more of a feature with rich web applications, it has always been a core use case with native mobile apps. People expect apps to stay usable, even connectivity drops. They certainly expect state not to get lost when the signal drops or gets weaker.

Proper offline mode support adds a lot of complexity and interesting edge cases to an app. State needs to be persisted, locally and when connection recovers, it needs to be synchronized back. You need to account for race conditions when a user uses the app on multiple devices - some online, one offline. You should take additional care with app updates that modify the locally stored data, migrating the "old" data to the "new" format - we'll cover this challenge in Part 3 of the series.

Decide what features should work offline and which ones should not. Many teams miss this simple step that makes the planning of the offline functionality easier and avoids scope creep. I suggest starting with the key parts of the application and expand this scope slowly. Get real-world feedback that the "main" offline mode works as expected. Can you leverage your approach in other parts of the app?

Decide how to handle offline edge cases. What do you want to do with extremely slow connections: where the phone is still online, but the data connection is overly slow? A robust solution is to treat this as offline and perhaps notify the user of this fact. What about timeouts? Will you retry?


Hey, computer buddy, when was the last time you were offline?
...
HELLO!!
Oh, shoot, it's me who's offline... again...

The app going offline is an everyday scenario. How will you handle this?

Retries can be a tricky edge case. Say you have a connection that has not responded for some time - a soft timeout - and you retry another request. You might see race conditions or data issues if the first request returns, then the second request does so as well.

Synchronization of device and backend data is another common yet surprisingly challenging problem. This problem multiplied with multiple devices. You need to choose a conflict resolution protocol that works well enough for multiple parallel offline edits and is robust enough to handle connectivity dropping midway.

With poor connectivity, the network request can sometimes time out. Sensible retry strategies or moving over to offline mode could be helpful. Both solutions come with plenty of tradeoffs to think about.

Retry strategies come with edge cases you need to think about. Before retrying, how can you be sure that the network is not down? How do you handle users frantically retrying - and possibly creating multiple parallel requests? Will the app allow the same request to be made while the previous one has not completed? With a switch to offline mode, how can the app tell when the network has reliably recovered? How can the app differentiate between the backend service not responding versus the network being slow? What about resource efficiency - should you look into using HTTP conditional requests with retries utilizing ETags or if-match headers?

Much of the above situations can be solved relatively simply when using reactive libraries to handle network connections - the likes of RxSwift, Apple's Combine, or RxJava. An edge case goes beyond the client side, which does get tricky: retries that should not be blindly retried.

Requests that should not be retried come with a separate set of problems. For example, you might not want to retry a payment request while it's in progress. But what if it comes back as failed? You might think it's safe to do so. However, what if the request timed out, but the server made the payment? You'll double charge the user.

As a consumer of backend endpoints, you should push all retries on API endpoints to be safe by having these endpoints be idempotent. With idempotent endpoints, you'll have to obtain and send over idempotency keys and keep track of an additional state. You'll also have to worry about edge cases like the app crashing and restarting and the idempotency key not being persisted. Implementing retries safely adds a lot of mental overhead for teams. You'll have to work closely with the backend team to map the use cases to design for.

As with state management, the key to a maintainable offline mode and weak connection support is simplicity. Use immutable states, straightforward sync strategies, and simple strategies to handle slow connections. Do plenty of testing with the right tools such as the Network Link Conditioner for iOS or the networkSpeed capability on Android emulators.

8. Accessibility

Accessibility is a big deal for popular applications, a few reasons:

1. If you have a large number of users, many of them will have various accessibility needs, finding it difficult - or impossible - to interact with your app without adequate support for these.
2. If the app is not accessible, there is an inherent legal risk for the app's publisher: several accessibility lawsuits targeting native mobile apps are already happening in the US.

Accessibility is not only a "nice" thing to do: your app quality increases as you make it more accessible. This thought comes from Victoria Gonda, who has collected excellent iOS and Android accessibility resources.

Before you start, you'll need to confirm the level of depth you'll go into implementing WCAG 2.1 mobile definitions. Ensuring the app is workable for sighted people over VoiceOver (iOS) / TalkBack (Android) and making sure colors/key elements are contrastful enough are typical baseline expectations. Depending on your application type, you might need to consider hearing people or users with other accessibility needs.

Accessibility goes deeper than ensuring sighted people can use the app. Allowing people's accessibility preferences to work with the app, such as supporting the user's font size of choice - Dynamic Type support on iOS and using scale-independent pixels as measurement on Android are both practices you should follow. You'll also need to take device fragmentation into account. For example, in the Android world, the OnePlus model is known to have a different font size to the rest of the ecosystem.

Implementing accessibility from the start is a surprisingly low effort on iOS and a sensible one for Android. Both platforms have thought deeply about accessibility needs and make it relatively painless to add accessibility features.

Retrofitting accessibility is where this problem can be time-consuming. Making accessibility part of the design process is a better way to go about things - this is why it's a good idea to make accessibility part of your planning/RFC process. Thinking in VoiceOver frames at a page level (iOS) and following accessibility best practices from the start are a good investment.

Testing accessibility is something that needs planning. There are a few levels of accessibility testing you can - and should add:

- **Automate** the parts of accessibility checks that can be automated - like checking for accessibility labels on screen elements. On iOS, you can also have VoiceOver content displayed as text and potentially automate these checks as well.
- **Manually test** accessibility features: do this at least semi-regularly, as part of the release process.
- **Recruit accessible users** in your beta program to get feedback directly from them. This is more feasible for larger companies - however, the payoff of having these users interact with the engineering team can be a major win.
- **Turn on accessibility features during development** where it's sensible to do so. This way, you can inspect these working and get more empathy on how people who rely on these would use them.

9. CI/CD & The Build Train

CI/CD for simple backend services and small web applications is straightforward. Yet, even for simple mobile applications, it is less so: mostly because of the app store's manual submission step. On Android, you can automate this process, as you can with enterprise iOS apps: just not for App Store releases.

No fully automated continuous deployment on iOS.

iOS and Android platforms are different: each requires their own build systems and separate pipelines. *Companies who end up going with a third-party CI will also struggle to find a simple-to-use solution, and in the end, will probably choose Bitrise. Bitrise is the only mature service on the market that started with iOS and Android CI as their core offering. All other CI services try to "lump in mobile" on top of the backend CI offerings, and it's just more painful.*

When owning your own infrastructure and having some dedicated staffing for builds, solutions like Buildkite can give more control and a better experience than third-parties. A few mobile leads at medium and large teams shared how they are happier with keeping builds in-house, despite the higher cost.

You'll find yourself using popular build tools to automate various build steps, such as uploading to the app store. For iOS, this will likely be Fastlane, and for Android builds running on Jenkins, it could be a Jenkinsfile or similar.

Be vary of maintaining your homegrown CI system if you won't have dedicated people bandwidth *to support this. I've seen startups repeatedly set up a Jenkins CI, get it running, just to realize months later that someone needed to keep dealing with infrastructure issues and the growing pile of Mac Minis. I suggest to either offload the build infra to a vendor or have a dedicated team owning mobile build infrastructure. At Uber, we had a dedicated mobile infra team who owned things like the iOS and Android monorepo or keeping master green at scale.*

The build train *is the next step after you have a CI in place. A build train is a way to track the status of each of your weekly or bi-weekly releases. Once a release cut is made for a "release candidate" for the app store, a series of validation steps need to happen: some of these automatic, some of them being manual. These steps can include running all automated and manual tests, localizing new resources, dogfooding, and others.*

Once the release candidate is validated, it is uploaded to the app store and waits on approval. After approval, you might roll out with a staged release - a phased rollout on iOS and staged rollouts on Android.

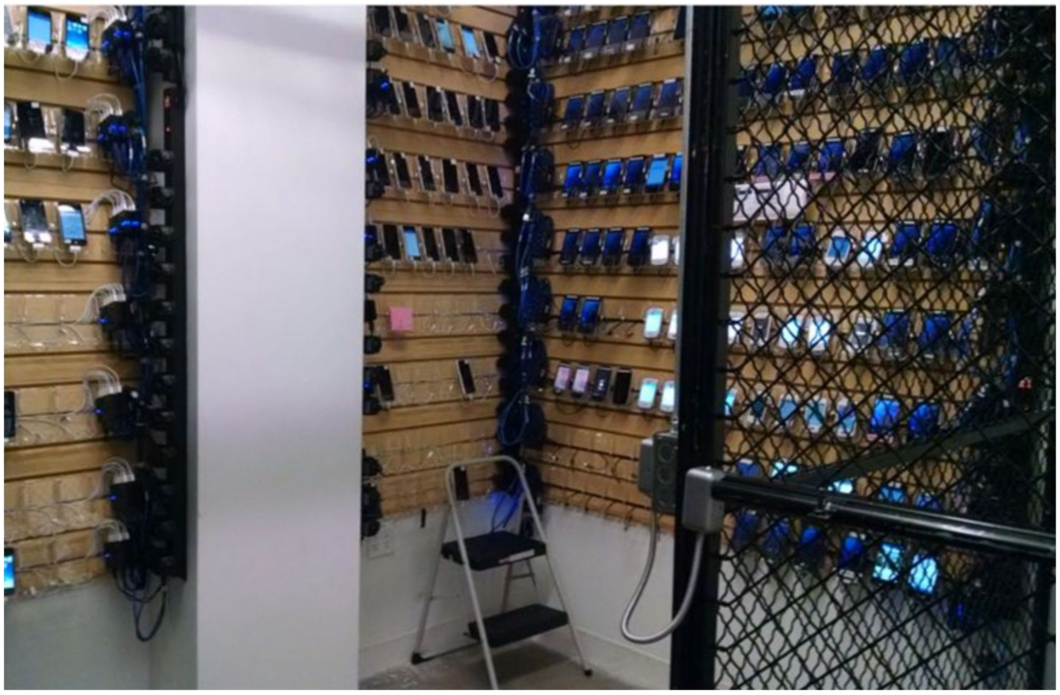
Your build train would visualize the status of all of the above: which commit was the build candidate cut, where the validation process is, and what the staged rollout status is. The release manager might manually track the build trains. Companies with complicated release steps and mobile infra teams tend to build their custom solution - we did this at Uber.

10. Device and OS Fragmentation

Device model and OS fragmentation is an everyday problem on both platforms. Device fragmentation and weird, hardware-related bugs have always been familiar pain points on Android. OS fragmentation is less of an issue on iOS, while it keeps getting worse on Android.

***Keeping on top of new OS releases and the accompanying API changes require a focus from mobile engineers.** Both iOS and Android keep innovating: features and APIs keep being added, changed, and deprecated. It's not just big changes SwiftUI or Dark Mode on iOS13, biometric authentication APIs on iOS 8 (2014), and on Android 10 (2019). There are several smaller APIs, like credit card autofill on Android Oreo - that exist on one platform, with no equivalent on the other. In all honesty, learning about the new APIs on WWDC or Google I/O, then adding them to the app is the fun part.*

***Making sure the app keeps working without issues on older OS and devices** is more of a challenge. You'll typically need to either set up an in-house device lab or use a third-party testing service to ensure that the app works correctly on all major models.*



A peek into Facebook's device testing lab in 2016: the slatwall. Credit: Facebook engineering blog.

Android has far more quirks when it comes to edge cases and crashes that are specific to certain devices. For example, Samsung devices are well-known for strange crashes related to the Samsung Android customization - not to mention special layout considerations for the Galaxy Fold. Amazon's Fire OS is another problematic device to support, thanks to the forked Android version these devices run on. Crash reports, user bug reports, and large-scale manual testing are ways to stay on top of new issues and regressions. All of these will be far more time consuming and expensive than most people expect.

Android has one more fragmentation issue: Android forks that do not run on Google's ecosystem. Apps built for Fire OS or future Huawei devices won't have access to Google Play Services. This means functionality like Firebase notifications won't work. For businesses that want to support these devices, using alternative approaches will mean additional time spent building, testing, and maintaining.

***Deciding how and when to stop supporting old OS versions** is a process your mobile team should put in early on. The cost of supporting old iOS and Android versions are high: and the payoff can be low. The business will naturally push to support as many devices as possible. The team needs to quantify what this support adds up to. When revenue or profits from the old version is less than the cost to maintain, the pragmatic solution is to drop support for old OSes.*

While there might be legal requirements in certain industries to support old OSes, the smaller windows you support, the faster you'll be able to move. At the end of 2020, it's common for Android teams to support from version 24 and up (Nougat) - but rarely going back to before v21 (Lollipop). On iOS, thanks to more rapid OS adoption, many businesses for versions beyond the last two or three ones, soon after a new OS release.

Part 1 of the series covered areas that differentiate building mobile apps from web and backend development. However, things start to get more interesting as the app grows. You keep adding new features to the app and tweaking the existing ones. Soon, the app that used to be a few screens gets so complex that if you were to print out screens on a navigation flow chart, it would take up the whole wall.

When working with a large and complex app, with a larger native mobile team, you'll find yourself with new challenges. We'll cover the most common 10 of these in this article - carrying on the numbering from the previous part of the series.

PART 2: Challenges Due to App Complexity and Large Dev Teams

Part 1 of the series covered areas that differentiate building mobile apps from web and backend development. However, things start to get more interesting as the app grows. You keep adding new features to the app and tweaking the existing ones. Soon, the app that used to be a few screens gets so complex that if you were to print out screens on a navigation flow chart, it would take up the whole wall.

When working with a large and complex app, with a larger native mobile team, you'll find yourself with further challenges. We'll cover the most common 10 of these in this article - carrying on the numbering from the previous part of the series.

11. Navigation Architecture Within Large Apps

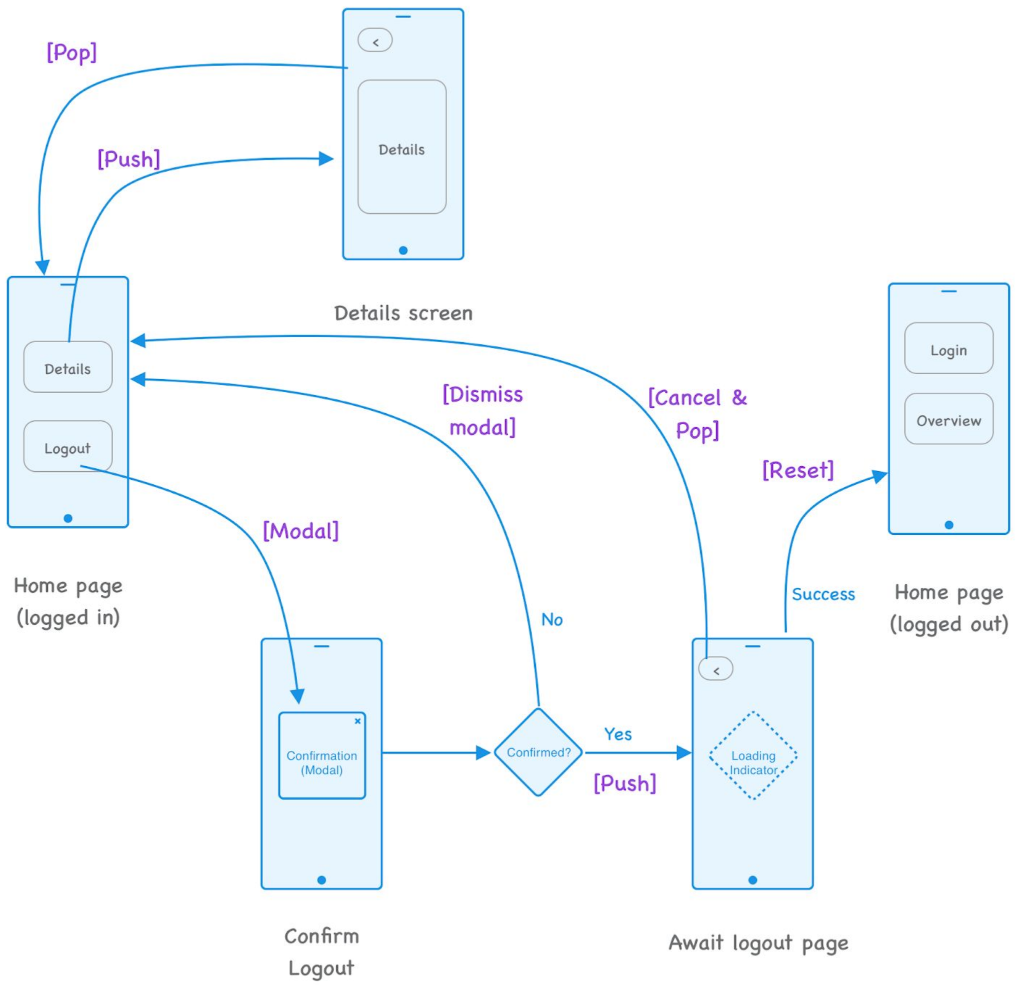
Navigation within mobile apps is just as much of an underrated problem area as deeplinks are. While the app is small, we tend not to pay much attention to it. As the app grows, we realize the navigation architecture has become a beast that needs to be tamed as the number of screens and transitions grows.

While both iOS and Android provide basic navigation concepts, they leave defining of the navigation architecture up to engineers. In turn, we tend to reinvent the wheel on navigation. This is mostly out of necessity, as neither iOS, nor Android ship with navigation approaches that scale well beyond simple apps.

Having a well-defined app navigation strategy with good separation of app state is key for any decent-sized app. What navigation happens between screens and components? What triggers this animation between taps and gestures? Is navigation independent of app state?

Many teams only build this map once they code themselves in a corner, discovering they have built inconsistent navigation solutions that lead to bugs when the user gets in unexpected states.

Inconsistent navigation can be as visible as the app using popups, toasts, full-screen modals, or pages/screens inconsistently. It can mean that different animations are used between different screens. It usually also means lots of code duplication in instructing the app to navigate.



An example of an app workflow with navigation overview. Does your app have a navigation overview? If so - is it the same across iOS and Android?

Asynchronous navigation is a common use case that few engineers consider handing ahead of time. By asynchronous navigation, I mean when something needs to finish before navigation can continue. Logging into the application, submitting a form are all examples of this. What happens when the user attempts to navigate away during this phase? If you don't plan and test for this scenario, the app can get into strange states. When using RIBs, Workflows provides an elegant solution to this problem.

A navigation framework or consistent navigation approach is something you'll find yourself either building, enforcing, or utilizing an existing component for with more complex apps.

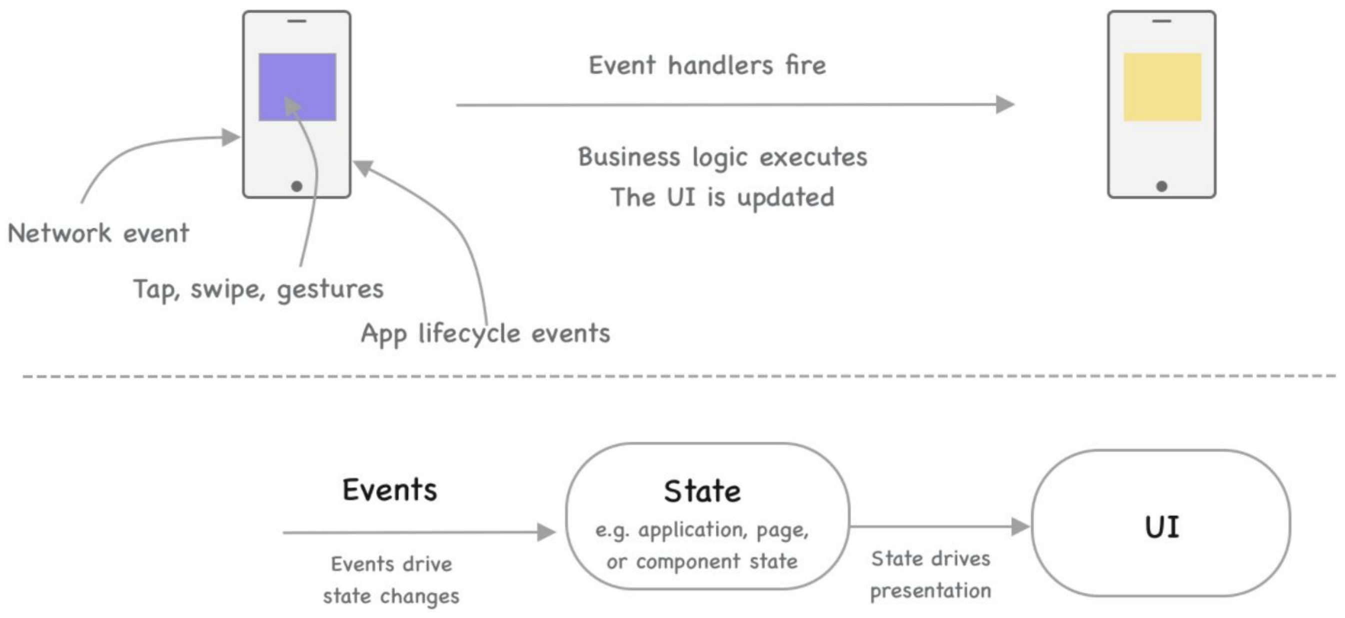
On iOS, there's no one native navigation component you can use. While several open source projects provide helpers, navigation is far from a solved problem on this platform. As John Sundell concludes in the article Navigation in Swift: "having a nice way to perform navigation, in a way that doesn't paint you into corners, can be really tricky."

Android is perhaps a step ahead. The Jetpack Navigation Architecture Component is becoming the preferred out-of-the-box library for navigation. It works well enough except for lack of support for some edge cases. Still, it's only been released in 2018, meaning apps the previous 8 years have built their own solutions on top of Activities and Fragments and now need to decide if they maintain those or migrate over. Even without Jetpack, Android is more opinionated in its navigation approach, like guidelines on the Up and Back keys and a back stack in place.

Mobile and tablet navigation differences is an interesting edge case. If your app has larger screens and forms, mobile devices might have multiple steps or screens, while the tablet version uses a single one. This scenario is more likely with iOS, where both the phone and the tablet sizes are well-defined. Supporting this scenario is not too difficult: but only if you plan for it.

12. Application State & Event-Driven Changes

What drives changes in the UI of your mobile app? Users tapping on certain parts of it. Data arriving from the backend. Perhaps a timing function. Most of the changes are driven by events, as we've discussed in the State Management section.



Events driving state changes in mobile apps

As the app grows in complexity, so does the number of possible states. Some state changes might trigger other state changes: for example, a component changing its state after a user tap might trigger the state of the page, or application to change.

Mobile apps typically have more states than web or thick client apps. This is due to the variety of lifecycle events they need to support - like app locking, app switching, background mode, and others - and offline mode support adding several other states. Web apps have fewer lifecycle events and seldom support offline mode extensively, and thick client apps have fewer lifecycle events, and connectivity drops are much more of a rarity than with mobile.

The larger and more complex the app, the more likely bugs are caused by a combination events that are out of the ordinary. The problem with out of the ordinary events is they are difficult to plan or test for. For example, a background push notification that causes a state change in a component might arrive right after the user locks the screen of the app - which notification causes a different state change. One of the more rare inbound events might be consumed by independent components/teams and this combination resulting in exotic bugs.

Follow state management best practices to keep the number of bugs low that happen because of state change issues. Keep state as immutable as possible and store models as immutable objects that emit state changes.

Record invalid states with information to replay or debug, so you have details on what went wrong and how to reproduce the issue. Using a bug reporting tool is the easiest way to start. One of the most popular tools is Instabug, with a notable mention for Bugsee. Several crash reporting tools also come with a bug reporting component.

Typically, the bug reporting tool would be shown to beta users to submit issues. When submitting the bug report, you'll also need to include the series of events that happened leading up to the issue and enough information so you can debug what happened. Most bug reporting tools will allow you to attach the logs emitted during the session: your challenge will be to emit enough logs to make

Automatically replay, pause, rewind, and debug the user's session is what an ideal tool would look like to debug application state issues. You'd want to have a log of not just user events but incoming network notifications. Being able to do this on a per-thread basis would be important to debug cross-thread issues. While there are such session replay tools on the web, unfortunately, on iOS and Android we're left with either having to build custom tools to do the above or work with debug logs and memory dumps for the foreseeable future.

13. Localization

Both iOS and Android offer opinionated ways to implement localization. iOS supports exporting localizations for translation, while Android builds on top of resource strings. The tooling is slightly different, but the concept is similar: to localize your app, define the strings you want to localize and provide these as a separate resource in the binary. Still, with large apps and many locales, you quickly run into challenges with this workflow.

Deciding what to localize in the app versus doing so on the backend is one of the first challenges any growing app will face. Any resource you localize within the app will be "stuck" with the binary: you won't be able to change phrases or update mistakes. If you serve localized string from the backend, you have more flexibility in this regard: and you only need to do one localization pass for both iOS and Android.

The topic of how "smart" the mobile app should be and what strings should live here, versus just getting them from the backend, run deeper. We'll go into more detail in Part 3: Backend-driven mobile apps. The more localization the backend does, the better. Backend-heavy localization keeps client-side logic low and reduces the number of resources to localize resources on mobile. While delegating more localization to the backend usually needs work both on the iOS and Android app workflows: it leads to easier maintainability in the long run.

When supporting a large number of locales, you need to ensure that all localization translations are complete before shipping to the app store. You might be using third-party localization services or an in-house team to do so. Assuming you have a build train, you'll want to allow the train to proceed only after all new resources have been localized.

Ensuring iOS and Android use the same language and localization is yet another challenge. Especially for larger brands, it's important that both the iOS and Android apps use the same - or at least similar - languages. It's hard to ensure this consistency without some shared localization tooling or the iOS and Android teams working closely with each other. Having the same designer and PM overseeing both apps also helps. Using the same localization IDs or keys is key to reduce duplication - and this needs to be done through the iOS and Android teams agreeing on conventions.

At Uber, we used an in-house localization tool across iOS, Android, backend, and web. This tool was integrated to generate iOS and Android resources and to track the completeness of localization. Was it an overkill to build a dedicated tool just for localization? I don't think so. Unfortunately, there is no good localization tool that you can use to unify translations across several stacks unless you build it yourself.

Localized custom fonts is a frequently overlooked area. It's common to have larger companies and those with strong brand identity to use custom fonts. However, this custom font doesn't always support every glyph for the app's supported set of languages.

It's tricky enough to confirm what languages and characters your custom font is missing support for. This is even more of an issue when using these fonts to display user-generated content, as you don't have control over this input.

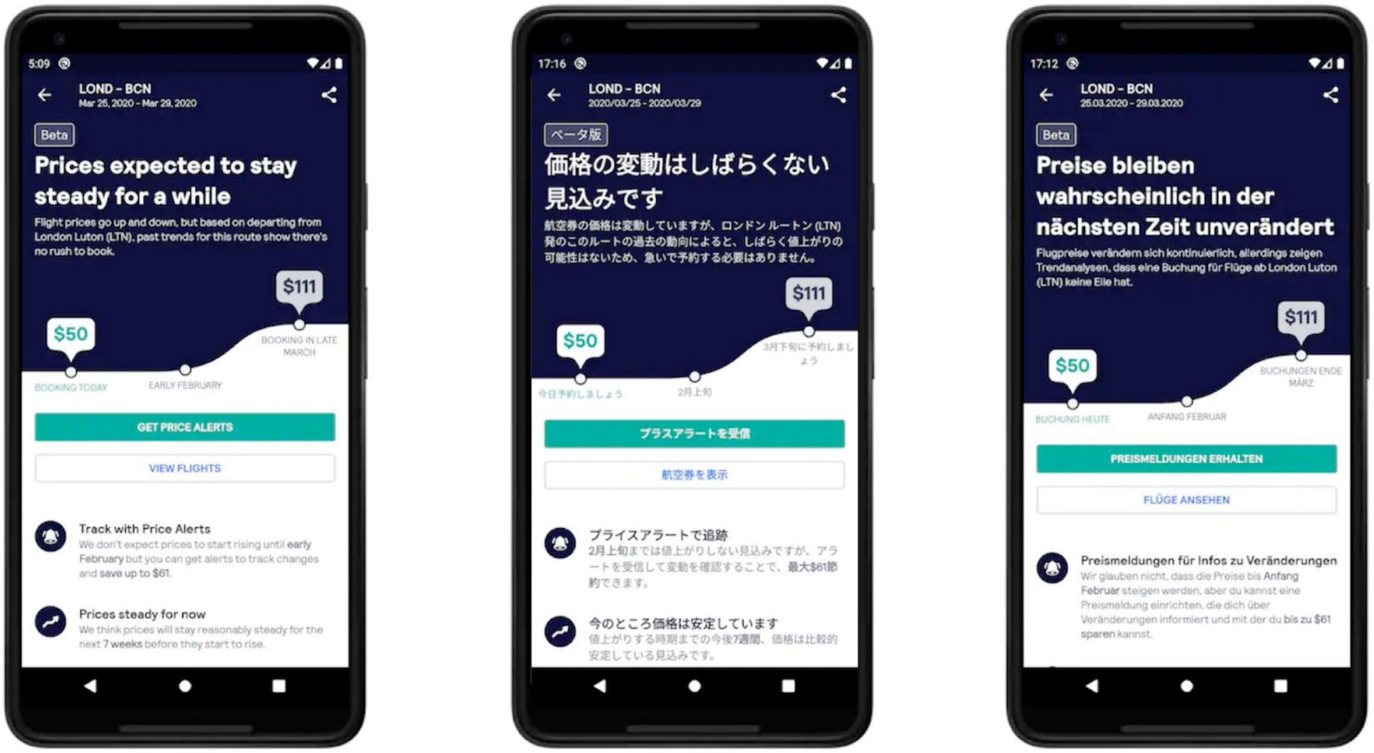
Once you've identified, you'll either need to have this support added or - more frequently - use a different font for the locales not supported. It gets even more difficult to manage a consistent, cross-platform mapping of fonts and languages, based on which languages each font supports.

When using custom fonts, you need to include these in the binary, adding to the app's size. We'll discuss app size in more detail in Part 3 of the series.

Currencies formatted differently on iOS and Android on certain locales is another pain point multi-language, multi*platform apps displaying monetary values will come across. Web has a similar problem with the inconsistency: Indian rupee values are a good example of this. The problem is especially pronounced when using currency types to add or subtract values. A common workaround is to have the backend format all currency data to the locale and not have the client do math operations or not with the currency representation.

Supporting RTL languages can often go beyond just translations. Most people reading this article will either be used to LTR and Latin-based languages or speak it fluently enough. However, when designing UIs for RTL languages like Arabic, Hebrew, and others, it's not just strings that need to be mirrored, but the layout of the app might need to be changed. "Mirroring" the layout is a common approach: but it might result in strange UIs for native speakers and users. It's best to have locals involved in this process: both for development and for testing.

Unique locales are ones always worth paying additional attention to. In my experience, **Japanese and German** are locales worth paying additional attention to. They can both be more verbose than English, and you'll want to make sure the layouts, paddings, and line breaks still work for these locales.


5:09

LOND - BCN
Mar 25, 2020 - Mar 29, 2020

Beta

Prices expected to stay steady for a while

Flight prices go up and down, but based on departing from London Luton (LTN), past trends for this route show there's no rush to book.

$50
$111
BOOKING TODAY
EARLY FEBRUARY
BOOKING IN LATE MARCH

GET PRICE ALERTS

VIEW FLIGHTS

Track with Price Alerts
We don't expect prices to start rising until **early February** but you can get alerts to track changes and **save up to $61**.

Prices steady for now
We think prices will stay reasonably steady for the next **7 weeks** before they start to rise.

17:16

LOND - BCN
2020/03/25 - 2020/03/29

ベータ版

価格の変動はしばらくない見込みです

航空券の価格は変動していますが、ロンドン ルートン (LTN) 発のこのルートの過去の動向によると、しばらく値上がりの可能性はないため、急いで予約する必要はありません。

$50
$111
今日予約しましょう
2月上旬
3月下旬に予約しましょう

プラスアラートを受信

航空券を表示

プライスアラートで追跡
2月上旬までは値上がりしない見込みですが、アラートを受信して変動を確認することで、**最大$61節約**できます。

今のところ価格は安定しています
値上がりする時期までの今後7週間、価格は比較的安定している見込みです。

17:12

LOND - BCN
25.03.2020 - 29.03.2020

Beta

Preise bleiben wahrscheinlich in der nächsten Zeit unverändert

Flugpreise verändern sich kontinuierlich, allerdings zeigen Trendanalysen, dass eine Buchung für Flüge ab London Luton (LTN) keine Eile hat.

$50
$111
BUCHUNG HEUTE
ANFANG FEBRUAR
BUCHUNGEN ENDE MÄRZ

PREISMELDUNGEN ERHALTEN

FLÜGE ANSEHEN

Preismeldungen für Infos zu Veränderungen
Wir glauben nicht, dass die Preise bis **Anfang Februar** steigen werden, aber du kannst eine Preismeldung einrichten, die dich über Veränderungen informiert und mit der du **bis zu $61 sparen** kannst.

The Skyscanner app with English, Japanese and German languages. German is often a good choice to "stress test" the app, as the language is more verbose.

Testing localization is no small effort. In an ideal world, a native speaker would go through every flow of your app after each localization change. In reality, this rarely happens: it's too expensive to do so. Most large companies rely on beta testers using the app with different locales to report glaring inconsistencies. For example, at Uber, we could rely on this method in spotting localization issues early: thousands of Uber employees dogfooded the app every week, alongside beta testers. Localization issues almost always got caught before the app got pushed to the app store.

You might need to define a workflow to validate localization changes. What should happen when a localized string changes in the app? Should this trigger an action to manually inspect the change? Should developers for the screen be notified? Should the release manager ship a new version of the app, even if there are no other changes? These might seem like minor questions. They are not. They are especially not when people translating strings work independently to engineers writing the code. Someone needs to define a workflow of not just adding but updating and testing localization.

Phrases that should not be localized are one final edge case. At Uber, we decided not to localize certain brand terms like Uber Cash or Uber Wallet. Some teams were not aware of this request and went ahead and translated the strings in certain screens: the owning teams had to test on all locales to find these issues. Several testers also reported the lack of these translations on a regular basis: this was a bit of noise we had to manage.

14. Modular Architecture & Dependency Injection

As apps become large, it often makes sense to build parts of the application as reusable components or modules. For large companies, either with several apps or several mobile teams, reusing the code owned by another team becomes a no-brainer. For example, a mobile platform team might own networking and shared architecture components; a Money team build and own payments components - this was my team at Uber! - and a Maps team would own all things mapping. Components and modules in the app would often map to the structure of the company's teams, mirroring the observation known as Conway's law.

With multiple modules, modules need to have a way to define their dependencies, both at a module and a class level. This concept is dependency injection - a form of inversion of control. It's a simple yet often underrated concept in mobile development - one that is far more commonplace with backend and web projects.

A major challenge with dependency injection is the amount of work it takes to modify or update dependencies if you don't use a framework to solve for this. Even without using a framework, the time-consuming nature of this is a tradeoff for clearer abstractions and good testability.

Dependency injection is a powerful tool for maintaining testable code consistently across the codebase. With dependency injection, classes that have multiple dependencies can be unit-tested by passing mock dependency classes when instantiating them. Larger, modular apps tend to introduce this concept one way or the other.

Manual dependency injection - creating all interfaces, then hardcoding all dependencies - works alright when there are few such dependencies. However, as the number of components and the number of dependencies grow, maintaining and updating dependencies becomes more difficult. Spotting things like circular dependencies also becomes tricky. Starting to use dependency injection frameworks starts to make more sense. Android has a mature dependency injection framework that analyzes dependencies compile-time: Dagger2. Google recently introduced Hilt on Dagger: a dependency framework built on top of Dagger. On iOS, there have historically not been similar frameworks: we built, used, and open sourced Needle at Uber, based on similar concepts.

At Uber, we would have had trouble scaling the code with over a hundred engineers working on the same codebase without dependency injection. We used this tool to be explicit about all class dependencies, make unit testing easy - and non-negotiable for most of the code - and reuse components across teams.

15. Automated Testing

If you're not doing a decent level of automated testing on a large app, you're digging yourself in a hole. By large app, I mean either a complex codebase or a large number of people contributing to it.

Here is what we understand under different types of tests in this article:

- **Unit test**: the simplest of all automated tests, testing an isolated component, also called the "unit". For iOS and Android, this would usually mean testing the behavior of a method on a class or a specific behavior of a class. These tests are simple to write and understand and are fast to run.
- **Integration tests** are a step up in complexity from unit tests. They test the behavior of multiple "units" interacting. These tests are more complex and take longer to run than unit tests. They might or might not use mocks. Here's a good overview of integration testing.
- **Snapshot test:** one comparing the layout of a UI element or page to a reference image. It's a cheap and fast way to ensure code changes did not result in unexpected UI changes. Popular snapshot testing frameworks include iOSSnapshotTestCase, SwiftSnapshotTesting, and Screenshot Tests for Android.
- **UI test**: a test that exercises the UI, and tests for the UI behaving in a specific way. Inputs come through UI automation, and the UI is inspected to validate assumptions. UI tests are usually the most complex ones to write and take the longest time to run. UI tests might have their data layer - the backend endpoints - mocked.

Unit testing is a baseline tool for sustainable engineering, assuming you have a team of more than a few engineers and a mobile app that is considered complex. Unit testing business logic, paired with code reviews, reduces bug rate, keeps the code cleaner, and increases knowledge sharing within the team. It also is the safety net to be able to make meaningful refactors and to be able to keep cleaning up tech debt. Unit testing comes with a pyramid of benefits - I wrote about these compounding benefits in more detail in this article.

When you inherit an application that has few tests in-place, and the parts of the app were not built in a testable manner, you often have little choice but to add integration tests and slowly add unit tests - for the new pieces of code. Retrofitting the existing parts of the application might make little sense.

Integration testing is more complex than unit tests: you test how two or more classes, modules, or other units work together. The most common case for integration tests is ensuring that library integrations work as expected. Integration testing is especially useful for libraries and modules you write that other app parts will reuse. The integration test would exercise the library/module's public API and confirm the component works as expected.

A special case of integration tests is UI testing parts of the app - but not doing this end-to-end. For example, testing that clicking a button on the app navigates to a specific screen (iOS) / activity (Android) would be an integration test.

Integration tests are more expensive to write and maintain than unit tests: though they can also be more valuable, given they test a larger surface area. There's no golden rule on whether you should lean on these or unit tests more: it will all depend on your environment and project.

Snapshot tests are the next line of automated testing. They're a special form of integration or UI/end-to-end test. A page is spun up, a screenshot taken and compared to a stored image. The data used is usually mocked. If it's different, the test fails, attaching the new image as the reason for failure.

I'm personally undecided on how useful these tests are. They're cheap to build, and they can catch visual regressions. However, they only test on specific resolutions: so you won't cover a variety of device sizes. I know of engineers who swear by them and those who don't think they add much value. Much of the usefulness of these tests will depend on your app. Covering the "core" flows that you don't want to change without noticing this is a good start to these tests.

At Uber, we took over iOSSnapshotTestCase from Facebook and used this tool heavily - but only on iOS. Back then, the Android team decided not to have snapshot tests as they felt the effort to do "proper" snapshot testing would have been too high on this platform, thanks to the variety of device sizes. This is also why I'm uncertain about recommending this testing approach across both platforms. Another downside to mention for snapshot tests at scale is how the reference images are stored in the repo: with thousands of test cases, these images take up lots of space and slows repo checkouts and updates.

Automated UI testing is where native mobile tooling gets quite limiting - when done at scale. Apple provides UI testing out of the box, which is a manual process to record, then replay. This tool works fine for a few test cases. For larger apps, you'll have to engineer a more robust solution and use approaches like robots or page objects. Capital One showcases a nice reference implementation for robot pattern testing and Wantedly shared a good reference for the page object pattern approach. Building your own system using robots or page objects isn't too difficult and scales pretty well.

Android natively supports UI testing with Espresso and with the UI Automator test frameworks. These are both powerful solutions that scale well for larger apps.

Once you hit dozens or more test cases, the existing tooling will feel somewhat limiting. For one, none of these test frameworks support mocking of network responses - you'll have to use other tools like Mocker on iOS or using OkHttp's MockWebServer functionality on Android. You'll have to choose a mocking approach and decide how to manage the mock test data.

Any team that writes more than 20-30 UI tests will run into the problem of how long these tests take to run. And the dilemma of whether to execute these tests before merging to master, or just occasionally. This issue highlights two things:

- **Consider where you focus your testing efforts** in the context of the testing pyramid. Yes, UI tests are important: but they are the most expensive ones to write and maintain and the slowest to run. Do you have the right level of investment with the other tests?
- **Solving for running UI tests at scale** has many benefits: engineers won't have to overthink adding new UI tests and can do so confidently. This talk outlines a few possible solutions for doing so. At Uber, we've spent considerable time and effort to parallelize UI test execution, detect flakey tests, and track/report the cost and benefits of UI tests.

Mocked live data for testing is another complexity of automated testing. Using mocked data has several advantages:

- **Speed**. Tests run faster when mocking data upfront, as opposed to fetching it live. Live data needs to go through the network.
- **Edge cases**. It's much easier to set up mock data that represents edge case scenarios than doing this with live data.
- **Reliability**. When using live data, the test can fail due to network or backend problems. While this might be a good thing in some cases - especially if the backend should be up 100% of the time - it can also generate noise.
- **Frequency**. When tests are fast, it makes sense to run them on every change: run tests locally that exercise code that was changed. This means close to immediate feedback to engineers. When tests are slower, they'll typically only be run pre-merge. The feedback loop gets longer, making development loops also longer.

Using mocked data comes with some problems you'll need to address - especially as the number of test cases grows:

- **Live data and mock data being out of sync**. Testing isn't worth much if the tests pass, but the app fails. You'll want to somehow detect when the live data changes, and you'll need to update your mocks. This is easier said than done.
- **Test data management.** As soon as you'll have many tests using mocked data, you'll want to come up with a structured way to both store this data, and to label the use cases they support. There's no definite solution: some teams keep the data in the code, some use human-readable formats like JSON, and others keep this data in a file hierarchy. Choose a solution that will make both understanding and modifying the test data easy while not making it too difficult to add new mock data and test cases.
- **Dynamic data experiences**. With more complex UI tests, you'll often want to set up a scenario where the data changes dynamically. For example, you might want to simulate a user with a set of attributes, then reset the app and simulate with another set of attributes. This case is an extension of how you manage test data. As your number of scenarios increases, keeping both the data and the tests easy to maintain becomes more tricky.

Automated testing could be its own article - or book! - by itself. Here is more food for thought from a variety of people:

- Two Years of UI testing by Tomas Camin and Francesco Bigagnoli (Swift Heroes, 2018)
- Testing for success in the real world by Donn Felker (Android Summit 2019)
- A Framework For Speedy and Scalable Development Of Android UI Tests at Doordash

16. Manual Testing

When starting with a small app, manual testing each release is a viable path. As the app grows, this effort gets more tedious. At scale - an app that's released weekly, with many engineers working on it - this approach will break down. The goal at scale is to always have a shippable master with zero - or as close to zero - dependency on manual testing.

There are a few challenges even when manual testing is done by choice: either because it's not a large overhead (yet) or because there aren't enough automated tests in place to allow to drop this testing.

- **Who does manual testing?** When I started at Uber, the engineering team owned all the manual testing for their features. We kept a simple Google Form in place, with basic instructions, recording "pass" or "fail". We repeated the checklist every week as part of the build train. Of course, as we grew, this approach did not scale well, and we started to rely on a release/test team.
- **How do you keep testing instructions up to date?** Regardless of who does the testing - and engineer or a dedicated person - you'll want clear and simple instructions. You'll also need test accounts, their login information, and the data to input in each test step. Where will you store this? Who will keep it up to date? At Uber, the platform team built a manual test administration system where engineering teams would record test instructions, and testers would mark the test as executed or failed every week.
- **Keep manual testing in-house, outsource, or mix?** At Uber, I calculated how much it cost us engineers to execute manual tests every week. This number was high enough to justify staffing this effort with dedicated people. During my time there, we used both third-parties like Applause, and a dedicated in-house quality team. In-house teams are less effort to start with and can access in-house systems. On the other hand, third-parties can be more reliable and scale up or down based on how much testing you need.
- **How do you integrate manual testing in your build train and release process?** How do you handle issues found? What type of issue should block the release, and what are things that won't? You'll need to weave the manual testing step into your release workflow.

A frequent headache with manual tests is how regressions are found at the eleventh hour - right before release. This is still better than the alternative - releasing with new regressions. Still, it can put engineers under pressure to fix a bug quickly, to avoid the release from being delayed. The root cause of the bug can often be hard to locate, as it might have taken a week or more since the offending code was merged.

If this "last-minute bugreport" issue is hitting you or your team frequently, consider either reducing the time to get feedback. For example, could manual testing start earlier? Could it run parallel to development? Can it happen continuously for key scenarios? Should engineers execute some basic tests? Can automation help more?

When you have a manual testing process in place, make sure to leave enough time not just for testing but also to fix any high-impact bugs. Do this either by doing manual testing early and leaving buffer time for fixing or by being flexible, pushing back the app release schedule if you find regressions.

Manual tests stay essential for mobile apps in a few cases - even companies who invest in best-in-class automation and spare no time and effort agree.

- **Interfacing with the physical world**. When relying on camera input for recognizing patterns like QR codes, doing document scanning, or AR, you can automate much of the tests but will still need to do manual verification to stay safe. When you build NFC applications
- **End-to-end testing of payments systems**. I've spent four years at Uber working on payments. Automating payments tests has a catch-22: payments fraud systems are sophisticated enough to detect and ban suspicious patterns - such as things that look seemingly automated. This means they quickly ban automated tests. You could test against test harnesses for payments providers, but then you're not testing against production. Deciding if you invest in working around payments fraud systems or investing more in monitoring, alerting, and staged rollouts of payments changes will depend on your situation. At Uber, we moved towards the former, and we'll cover monitoring, alerting, and staged rollouts in more detail in Part 3.
- **Exploratory testing.** Great QA teams and vendors excel at this scenario. They'll attempt to "break" the app through creative steps that engineers' won't' think of, but end users will do it, regardless. Except unlike users, who often won't report anything, but quietly churn, these testers will provide detailed steps for reproducing all issues. For apps with millions of users, exploratory testing is an area to invest in: the only question is how frequently to do it and how much budget or time to spend on it.

There's far more to talk about with testing, and I recommend the book Hands-On Mobile App Testing by Daniel Knott to go deeper for both automated and manual testing. This book covers test strategies, tooling, rapid mobile release cycles, testing throughout the app lifecycle, and more.

17. Scaling Build & Merge Times

Native mobile app build times quickly become problem areas with most projects. iOS engineers are - unfortunately - familiar with how seemingly slow Xcode builds projects, and Android engineers with a project beyond "Hello, World" will also start to count the seconds - then the minutes on a build.

Neither Apple nor Google have prioritized build time improvements for large projects. This is especially true for projects with hundreds of thousands or millions of lines of code and dozens of dependencies.

Luckily, there are tools to make build times faster, but they take time to integrate, tweak, and fit into the "one-click-build" workflow. Bazel is the tool that is becoming the most popular among companies building mobile at scale. Many companies are in the process of moving to Bazel from Buck or Gradle: Uber, Pinterest, and the Android AOSP platform are examples of companies and projects moving over. Even with Bazel, there is plenty of work to do to speed up the build. The more mobile engineers in a company, the more it can make sense to have engineers work part-time or full-time on improving the app build experience.

Focusing on build improvements becomes a must, on a per-platform level, as the mobile developer team grows. There are several low-hanging fruits teams like this can tackle and other, more complex ones. An example of a lower hanging fruit is how on iOS, dynamic frameworks can be costly both for build times and for app size. When shorter build times and the smaller app size is a priority, it's worth switching to static libraries. A more expensive to implement change is moving to a monorepo.

Whether to keep using distributed source code vs move to a monorepo is a question that teams at the size of 20-40 engineers start to ask. Most teams start with pulling in dependencies and libraries from different repositories, doing this before compile time. The time to download dependencies through the network is time-consuming, so caching is used to optimize this approach.

As the codebase and the number of engineers grows, the time spent fetching dependencies keeps expanding. Suddenly, the idea of having all dependencies live in the same monorepo seems less crazy. There are still no great monorepo tools coming up to 2021 for mobile, so you'd have to build much of this for yourself.

At Uber, we moved to a monorepo for Android and one for iOS, when we hit around 100 native engineers per platform. Should you do the same? Unfortunately, there is no definite answer, only tradeoffs to consider. What is more important for your team? Short build times, standardized versioning, or keeping the investment low on custom tooling or mobile platform teams?

Keeping master green - at scale. How difficult is it to keep the time to merge changes short, and the main branch always green? If you have builds that execute fast - say, 15 minutes to run all tests - and few merges per day - say 10 or 20 - this is no challenge.

What if your build takes 30 minutes to complete with unit, integration, and UI tests? And what if you have 10 pull requests coming in per hour? Is there a way you can both keep the main branch green and also keep the time to merge low - as close as possible to the 30 minutes minimum?

To make this happen, you need to parallelize both the builds. But is it safe to run two PRs parallel when one could impact the other one? This problem leads to a complex but equally exciting problem space. At Uber, we ended up building a Submit Queue that breaks up builds into parallel parts. It then does probabilistic modeling on the build queue, determining changes likely to pass and prioritizing these. If this sounds like something you'd like to learn more about, you can read the details of this approach in this whitepaper.

While there are probably few companies with the amount of mobile PRs happening daily as Uber had: if you're one of them, you'll have to invest in bringing down the average time to merge changes and keep the engineering feedback loops low.

18. Planning and Decision Making

When you work on a small mobile team with a handful of engineers or less, you just go and build the new features. You'll probably discuss decisions with each other and comment on code reviews, but that's about it. With a larger team, this process will have to change - both to avoid stepping on each others' toes and to keep the code and architecture choices consistent.

Formalizing the planning process is a practice worth introducing above a certain team size. I've seen companies do this as early as a few mobile engineers, and ones delaying it until they had 20-30 native engineers. Uber started an RFC planning process early on when there were less than a handful of mobile engineers.

You'll probably want to formalize the specification phase before or at the same time as the (mobile) engineering planning phase. A large part of project delays come from unclear requirements and scope creep - and being clear on what the business wants to build. This is usually done by product managers starting up a PRD (Product Requirements Document) process, this document being a formal, "you can now start work" step with engineering. See this list as examples of how various companies approach PRDs.

iOS and Android teams working together on planning is such a big win efficiency-wise, and yet in many companies, these teams work in silos. By planning features together, you ensure you'll both build the same functionality. iOS and Android engineers learn about the differences compared to the "other" platform. You can standardize things like feature flags, analytic events naming, and, perhaps, even class structure. Engineers can review each others' code, and an iOS engineer might be able to point out an incorrectly implemented business case in the Android code. A win for everyone!

At Uber, one of the major advantages of using RIBs as our architecture is how the structure of the app and components were very similar across iOS and Android. This made shared planning a given: something we just did, without even thinking about it. Why would you not do so?

Decision paralysis is a situation that comes up with teams that start to get good - and thorough - with planning. I've observed teams doing a thorough analysis of the problem space and alternatives, then delaying making a decision.

I suggest timeboxing planning, and in the end, going with the most sensible approach, given the information you gathered. I've noticed diminishing returns in spending more time over planning instead of building in short iterations and getting feedback if your plans actually work. Especially on mobile, where you can run into tooling and framework issues, shifting to spend time on prototyping can often be more useful than whiteboarding further approaches.

Getting the "signal to noise ratio" right is a problem many teams and companies worry about planning. Teams are often hesitant to work on a new feature, as they don't want to create too much noise. Executives sometimes worry whether their teams can focus on shipping features after those teams start to broadcasts plans more openly.

I’ve personally only seen downsides of working in silos - at Microsoft, this was the case back in 2014. At Uber, I was surprised to see the “broadcast to all” model work exceptionally well. Whenever a mobile team was about to build a new feature, they sent an “Intent for RFC” summary to a mobile-only RFC mailing list. Almost all of the 300+ mobile engineers at Uber were on this list. Then, once the RFC was ready to review, they emailed this list as well. This process helped catch problems early. It also spread knowledge fast - and even helped people decide on moving teams to areas they were more passionate about!

I’ve since advised startups and mid-sized companies on starting a design doc or RFC process. My advice in brief runs like this:

- Define a design doc template for a given stack (e.g. for mobile, backend, or web).
- Get most engineers on board with the idea.
- Start writing short docs and sending them out to everyone, as well as making it clear who the required approvers are.
- Over-communicate rather than under-communicate. Prefer lightweight process and tooling to heavyweight ones.
- Observe what happens, and iterate based on the reaction.

If your mobile team is not planning with lightweight design docs or RFCs: consider giving it a go.

19. Architecting Ways to Avoid Stepping on Each Other's Toes

We've not mentioned mobile architecture until now. Is it not an important consideration, even when apps are small? In my observation, while a team is small, *architecture really doesn't matter that much*. As long as you choose a structure that is testable enough, you'll be fine with MVP, MVVM, MVC, VIPER, or a similar approach.

Back in the day, the old Uber app followed the "Apple MVC" approach on iOS and MVP on Android. While pain points kept growing with the number of engineers, the app grew to almost 100 native engineers working with this architecture - before bigger architecture changes were made, migrating to RIBs.

The trouble starts when too many engineers end up modifying the same files. In the case of the early Uber app, this file was the presentation/view logic of the on-trip page. This was a screen where up to 50 engineers made changes - and accidentally broke each other's functionality. We tried to break the page down into smaller components, but component communication and navigation conflicts kept coming up.

Building an architecture that supports hundreds of mobile engineers collaborating on the same codebase is not trivial. The team at Uber spent half a year prototyping and experimenting with different approaches, from the usual suspects, all the way to (B)-VIPER. In the end, we built our own and open sourced the architecture known as RIBs. We optimized the architecture to work for a large number of engineers and many nested states. The architecture uses immutable models across that re-emit values across the app whenever possible. The architecture wasn't just diagrams: we built tooling to deal with the increased complexity from code generation tooling to onboarding tutorials.

With a large team, architecture is a means to control the level of isolation between teams and components - limiting overlaps and accidental conflicts while increasing complexity and lines of code. Maybe MVP, MVVP, (B)VIPER, RIBs, or another better-known approach works well enough for your team. If it doesn't: start to formalize what solution would work, why, and what the tradeoffs would be.

You can formalize your approach in various ways:

- **Verbal agreement**: the quickest way. It's also the one that leaves room for most interpretation, and new joiners might not be aware of past discussions.
- **Documenting the approach**: a common approach and works well for smaller teams. It's down to each engineer to follow recommendations and for code reviewers to notice when engineers are diverging from this agreement.
- **Tooling to help with formalizing**: you can build frameworks that help with consistency. You could use templates or code generation tools so your components are "wired up correctly" at the click of a button. You could also put lint rules in place to automatically catch "architecture violations". The larger the team, the more it can pay off to invest beyond just documenting your approach. At Uber, we've built tooling around components generation and had the linter "enforce" certain architecture rules, like Presenters not being allowed to know of or invoke Routers.

Principles you'll most likely need to follow to support up to hundreds of mobile engineers will include things like:

- Feature isolation
- Monorepo-like code structure - here's how we did it at Uber on iOS and on Android
- Strong code and feature ownership
- Automated tests guarding the app's features

For a mobile architecture to support working at scale - with several dozens of engineers - mistakes need to be clear before merging into the main branch. Automated tests should catch functionality issues as part of the pre-merge steps. You'll need to invest in an architecture that supports this kind of testability and keeps a high level of test coverage. Engineers should be confident in merging changes, knowing they did not beak other parts of the app.

21. Tooling Maturity for Large Apps

Companies whose apps are compiled from millions of lines of code usually also set up a mobile platform team. At this scale, the company will have dozens of native engineers - or more - working on apps. They'll run into the problem of native development tools starting to have performance or process issues at this scale, with no good off-the-shelf solutions. This was the case at Uber, and I hear companies like Facebook, Spotify, and many other companies face the same issue.

Build time at this scale is one of the biggest problems. By at scale, I mean building a project with a few million lines, and doing this a dozen times per engineer, times a hundred or more engineers. Throw in running a full automation test suite, and it's not hard to justify hiring a few people to build something that's better than what most services have to offer.

Should you use Bazel, Buck, Gradle, or xcodebuild? What are optimizations you can make on the build? This problem space will keep engineers busy: especially as the build tools area is an ever-evolving one.

The workflow of shipping more than a few apps is also challenging. Each app needs to have checkpoints from the build being cut, through test phases - localization tested, manual and smoke tests, performance tests, beta users, and others. Orchestrating this process for one app is not that difficult. However, keeping track of dozens of build trains across iOS and Android is something that internal tools can do better than anything you can readily buy. You'll find yourself needing to build a build or release train for your own needs.

The maturity of tools keeps improving across both iOS and Android - I personally expect more off-the-shelf solutions to become available in the coming years. Still, compare the mobile tooling ecosystem to that on the backend: tooling for large mobile apps seems to be less of a solved problem than for large backend systems.

PART 3: Challenges Due to Stepping Up Your Game

This section is being written, and will be released in an upcoming update.