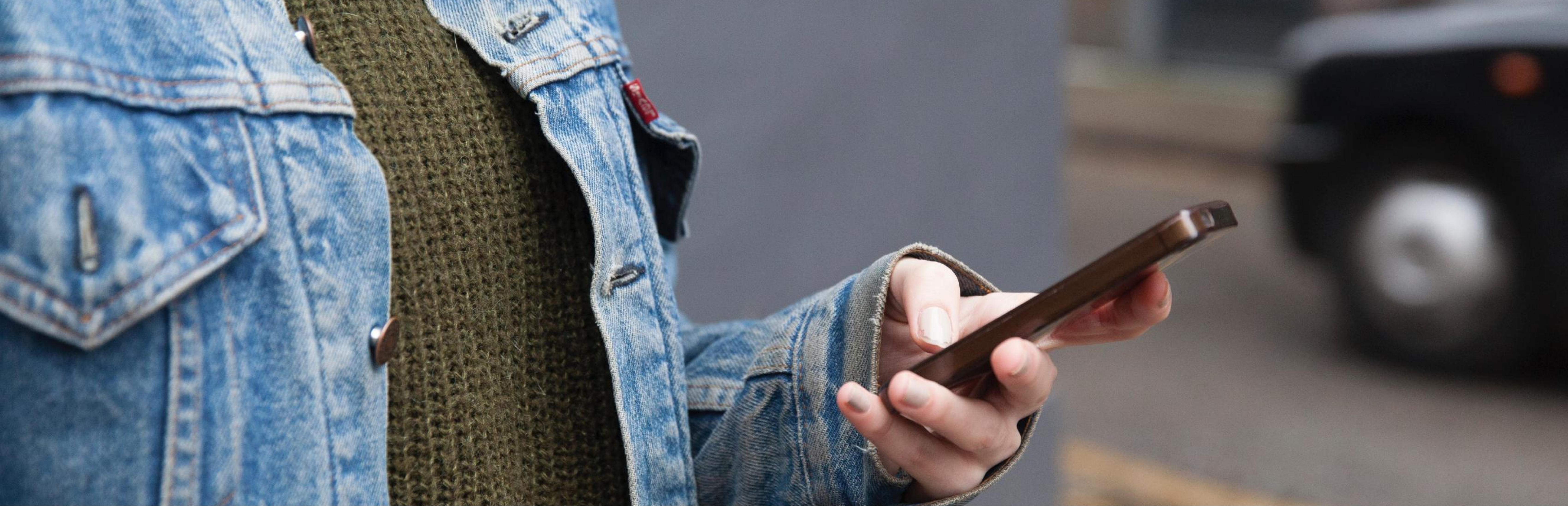
MOBILE APP PLAYBOOK: LESSONS LEARNED

Our mobile app playbook series features tips on the tools and processes we've learned in building mobile apps.

TABLE OF CONTENTS



Introduction

We know firsthand that it's hard to build successful iOS and Android apps. We've built quite a few: the Twitter app itself, and also lots of apps and samples written from the ground up with no Twitter internal systems or tooling. And of course we've learned a lot about how our partners build apps and tackle problems.

At this year's Flight conference, we showed off the power of some of the new features from Fabric, with two apps that we created and open-sourced on GitHub: Cannonball, a magnetic poetry game, and Furni, a mobile-first furniture store. Building these apps has been exciting, and educational. Moving from ideation to product on a (very) small team, we got a lot of assistance from others, and from tools available through Fabric.

Our experience building Cannonball made it so much easier to build Furni the following year. We thought we'd organize what we've learned into a chronological playbook that you can use as a guide when you build apps. Over the coming weeks, watch for our series of posts with tips on the tools and processes that we've learned.

We don't pretend to have all the answers, and you certainly don't have to use all the tools we mention – but we'd like to pay it forward, based on knowledge we've gathered from colleagues and learned by making mistakes.

PART ONE

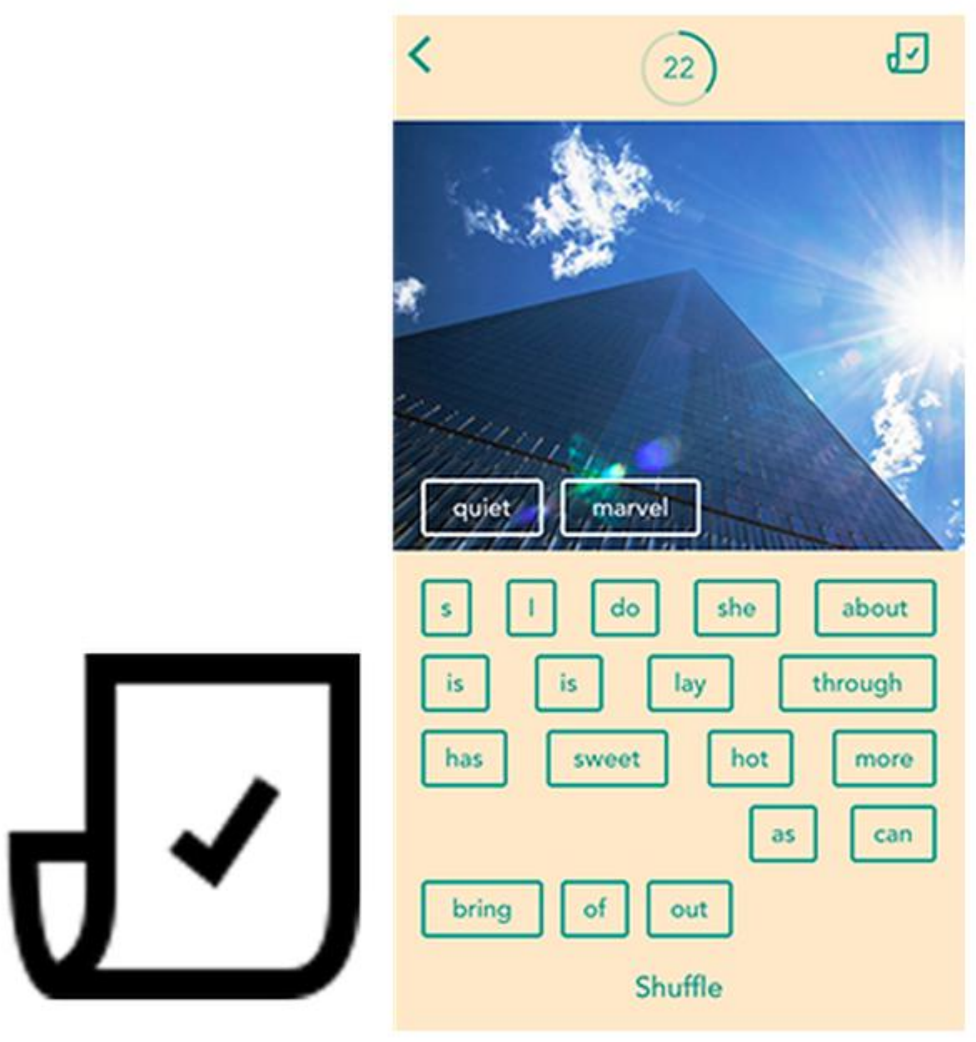
Prototyping and design

Every app starts as an idea. We've all built just-for-fun projects to play around with new frameworks or cool hardware – but the apps that go places solve real problems, enable new and interesting experiences, or entertain.

Even though good design seems obvious when you see it, designing for great user experiences isn't a simple process. Big companies devote entire teams to it – but if you don't have the luxury of having a big team, there are still ways to create a solid UX on your own.

Start with a whiteboard. It's old-school, but way faster for iteration (so easy to erase mistakes). Show other people the whiteboard. If they look confused, ask them why, then make changes with them.

Focus on the actions people will take most. Are they selling crafts? You'll definitely need a "create listing" view. Is it an events app? Collecting RSVPs should be easy. Make these screens easy to locate, and buttons for key actions easy to find- and understand. Designing your own icons may help establish an individual look and feel for your app, but don't reinvent the wheel. Obvious is what's familiar.



Our "finish poem" icon, alone and in context.

Wireframe it nicely – then get feedback. Tools like Sketch (@sketchapp), Paper (@FiftyThree), and Balsamiq (@balsamiq) can turn a scribbly hand design into one people can read and give good feedback on. These tools also make it much easier to communicate ideas between team members who aren't sitting next to each other. When we were building Cannonball with our team in London, San Francisco, and São Paulo, this was critical. A proper wireframe helped us all be sure we were building the same app.

Get a quick and (maybe) ugly version working. Once people agree on the overall key screens and interactions, build each screen and link all of them to your navigation structure. Storyboards (on iOS) make this easier. System fonts, standard padding and standard views (iOS and Android) are your friends. If you want to get fancy, you can use tools like Playgrounds in Xcode to prototype interactions quickly, before spending too much time building them into your app.

Then get a prettier version working. No amazing mobile app relies entirely on system-provided views, animations, and interaction design. Once you walk you can start to run– with animation libraries and physics engines that make your app more fun to use. Once you've made the app functional, it's important to make it fun.

In the future, we could certainly extend Cannonball to make the process of building a poem more visually dynamic, or update the horizontal scrolling behavior in Furni's main screen to feel more modern.

But we didn't in v1, which leads me to the next thing we tackled.

PART TWO
Start with a stable foundation

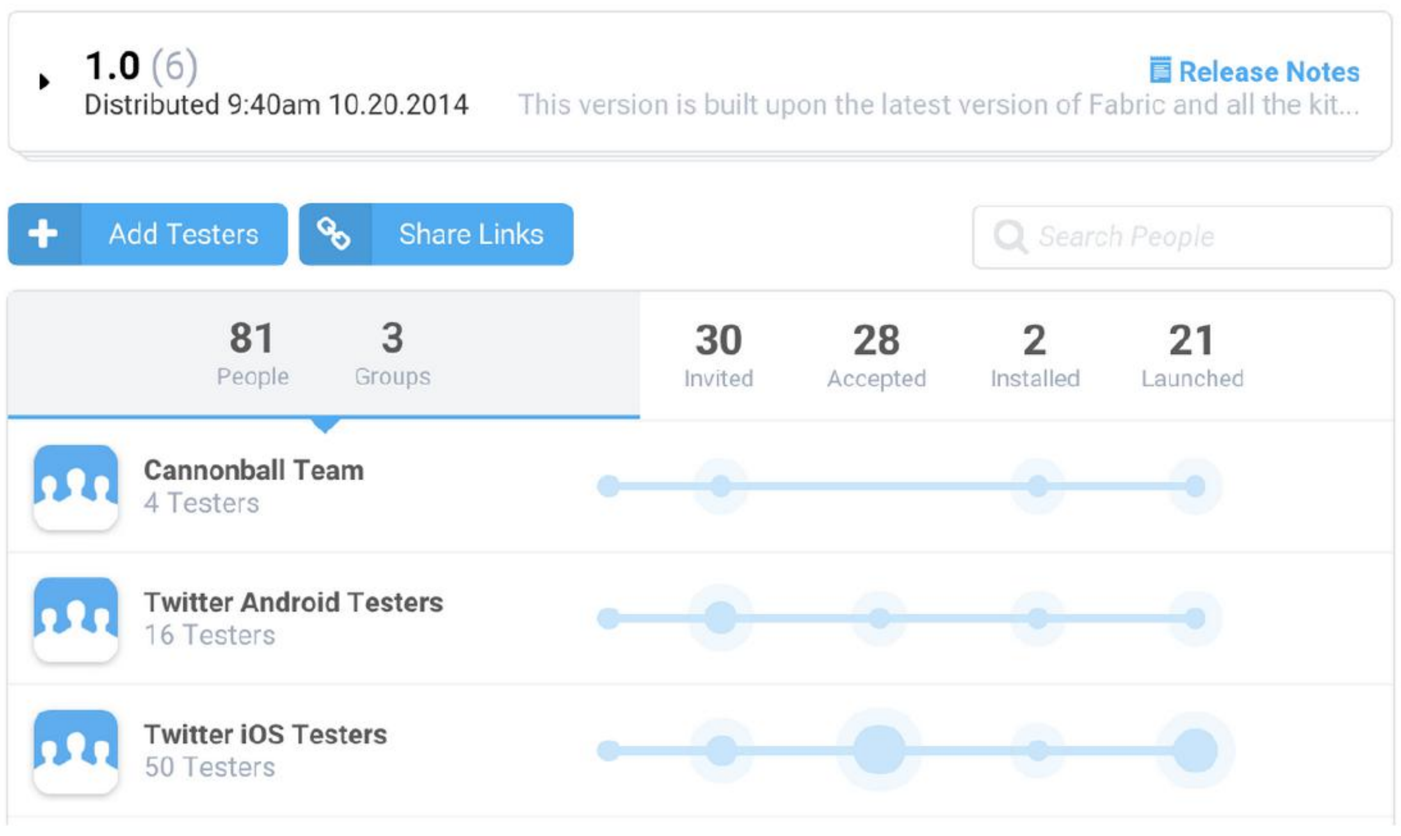
With Cannonball and Furni, building an open source-able sample helped us focus on a few operating principles. We knew the app wasn't going into wide release immediately, so we didn't waste time on premature optimizations — but knowing that the code would be published for anyone to see was the impetus we needed to write it right.

"Right" in this case meant nicely decomposed, easy to read – and also not crashy. Unless I'm hopelessly hooked, a new app loses me if it crashes all the time, particularly on basic or critical processes. And with mobile bounce rates at nearly 40%, I'm not the only one.

With Cannonball in particular, we were trying to release the open source code simultaneously on Android and iOS, while also shipping it to the App Store and Google Play in time for the Flight conference, while building on a platform that was (at the time) top secret. Our tester pool was pretty limited (employees only), and the number of things that could go wrong on any given build was very large. We also knew that the people downloading it on day one would be developers – and that a major part of our platform was crash reporting. No pressure.

Using Crashlytics Beta really saved us. Simulators and emulators can take you pretty far for some amount of testing, but there's no substitute for getting your app on real devices, being used by real people who use and abuse it in ways you couldn't predict. Having our colleagues test out the apps helped us find and solve dozens of bugs we had never seen before.

And we knew how to track down and pester the people who hadn't helped.


1.0 (6)
Distributed 9:40am 10.20.2014
Release Notes
This version is built upon the latest version of Fabric and all the kit...

81 People · 3 Groups · 30 Invited · 28 Accepted · 2 Installed · 21 Launched

Cannonball Team
4 Testers

Twitter Android Testers
16 Testers

Twitter iOS Testers
50 Testers

The two people who installed but didn't actually test our day-before-Flight build? Dead to us.

The combination of beta testing, crash reporting, and analytics was particularly helpful for getting a sense of how "bad" things were inside the app, and how effective our testing was. It doesn't matter if you send a build to 500 people if only 50 of them try it, and only 20 get past logging in: your app is essentially not tested.

We were able to get details about how many people tested, for how long, and the top screens where they spent their time in the app– and, on the flip side, which ones hadn't been as thoroughly tested. We were also able to see the percentage of people who actually experienced a crash — important information, when seeing any reports feels scary (don't worry, everyone has lots of them). Putting those crashes into context with numbers around the actual end user experience helped us figure out what was an edge case and what was a showstopper.

Beta testing the app and actively using Crashlytics for monitoring crash reports didn't just help us save face on the day of Flight; it's helped us retain hundreds of monthly active users for Cannonball – even a year after release.

And hey – if you've run either of the sample apps and are seeing lots of crashes in your own Fabric dashboard, we're always accepting pull requests.

PART THREE

Accounts and social login

With the exception of my BART trip planner, every single app on my phone features some kind of account system. Sure, the approach is different from app to app — some won't let you access any features or content until you've created an account, while others rely on local data storage until you share information or sync your account to a remote source. But at the end of the day, each app offers the option, the ability to track who you are, manage your preferences, and remember your habits.

But people's online identities generally aren't static. They pick and choose who they want to be given the context of what your app does. In my case, some services get my email address; which address they get (work, personal, college, my first AOL account from 1995) varies. Others get different social identities. Others get my phone number.

As a person who uses apps, I care about:

Control. Does creating an account enable the company to contact me? If so, how? What personal information am I sharing when I register or log in? Does this app have the ability to post as me, or contact my friends?

Expediency. Exactly how much typing do I have to do one-handed on this 5” screen just to log into your app? This can be a major barrier for first time users, and for people who have forgotten their passwords.

Sharing data. Is it convenient or a better experience for me if you have information about me from another app? Does it save me having to type more or fill out a profile (see “Expediency”)?

As an app developer, you should probably care about:

Conversion rates. This is the big one. If people need an account to use your service, ideally 100% of those who open your app would create an account. We all know the conversion rate is not even close to that. More on this topic soon.

Getting data you can use. What about user identity matters to you: their interests? Access to their social graph? A way to contact them? Distinguish between a need and what’s nice-to-have — and if you get their data, be sure that your customers get some value in exchange.

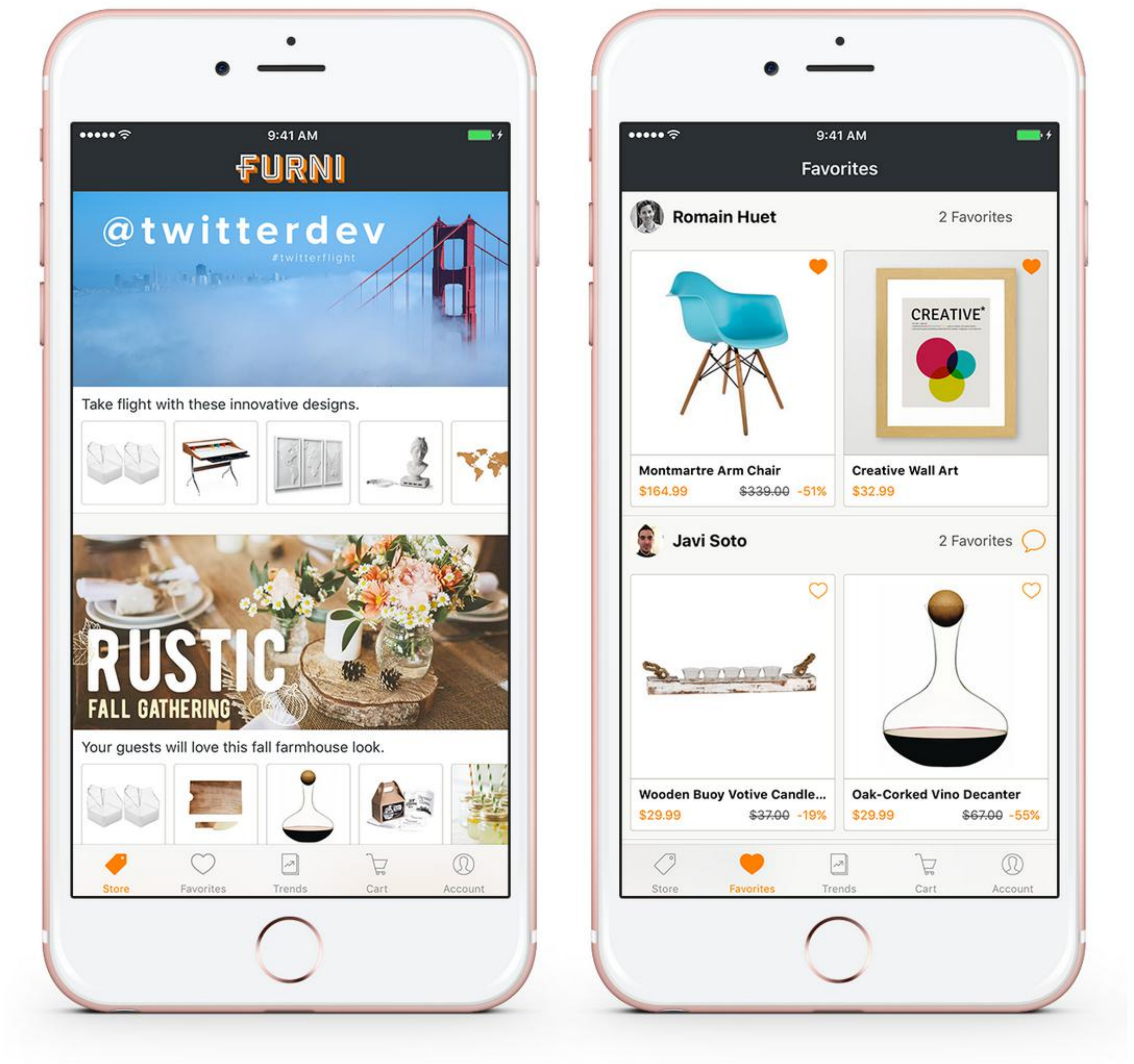
Encouraging growth of your app. If you ask people to share your app with their friends, they’ll likely share using the same identity they’ve chosen to identify themselves to you. You may want people to log in first with their email address, but later want them to sign in with a social service in order to share more broadly. It’s not uncommon for apps to ask for multiple logins for that very reason.

From our own experience, here are a few tips for how best to meet everyone’s needs:

Motivate the login. Let people know why you need them to log in. A simple sentence like, “Sign in to customize your FooBar experience!” goes a long way in persuading people it’s a good idea. If this is their first experience inside your app, you have to gain their trust. Justify any requests for data from other services and be crystal clear about why you’re doing with it.

Give people options when you can. Earlier, I suggested you identify what information you really need from other services, and what’s just nice to have. Maybe all you need is a consistent identifier. In that case, you can give people a choice of how they log in, which improves conversion rates all around. This also opens doors for a broader range of people to use your app. Not everyone has social media accounts; not everyone has an email address. Don’t prevent these people from being able to use your app. We’ve seen that login with Digits bumps conversion rates to 85%, because every mobile user has a phone number.

Allow a deferred secondary login. Maybe getting started in your app only requires a user identifier to save data, but sharing to a social network or inviting phone book contacts requires a specific type of login. In general, social conversions improve when you defer the prompt for people to login until the benefit is obvious (e.g., you clicked "Complete my profile using Twitter," so you need to login with Twitter).



In Furni, you don't need to log in to view furniture; you do if you want to save favorites or see items your friends liked.

- **Persist sessions sensibly.** In some cases, if a person is active daily, weekly, or even monthly, the activity is enough to keep a session alive. Other apps that require increased security, like banking apps, may require a login on every open, and end sessions if the app has been backgrounded too long.

Decide how frequently you need people to log in, and, where appropriate for user security, consider using touch-screen friendly options like touch ID on iOS, one-click social buttons, or a custom pattern.

- **Handle logout gracefully for all associated services and sessions.** Some people prefer to log out of services when they aren't using them; others may have multiple accounts in your app that they want to toggle between. Be sure that your app handles both scenarios.

That's enough preaching. What did we do in practice when we built our own apps? When we built Cannonball, we gave people the option to log in with Digits or Twitter, or skip login at the start so they could just start creating poems. We gave people options, check. Allowed deferred login, check. Persisted sessions sensibly, check. Made switching accounts straightforward, check.

It looks like we did OK, but if you've dug into the code for Cannonball, you may have noticed that even though the app features both Twitter and Digits login, the Twitter and Digits sessions are never used by any other piece of code outside the login controller.

In fact, Cannonball doesn't even have a backend to store identity or the poems people created – it's all just in a local DB and never synced anywhere.

We did this for the sake of expediency, and also sample code readability: when teaching people to use Fabric, we shouldn't make them learn 3-5 other tools just to get the sample running.

Furni was intended to demonstrate a more complete, "real world" app. People could browse content before using Digits to create an account; later they could include their Twitter handle to contact customer service, or their email address so the store can send them receipts. We didn't bother writing a separate email login option; asking the user to let us pull the email address directly from Twitter saved us the trouble of building out an entry form. The app makes use of phone book contacts through Digits, and also sharing through Twitter.

So we followed our own best practices, but that meant we had a new problem to deal with: a tangle of optional login identities, and the requirement that we sync people's data properly across devices.

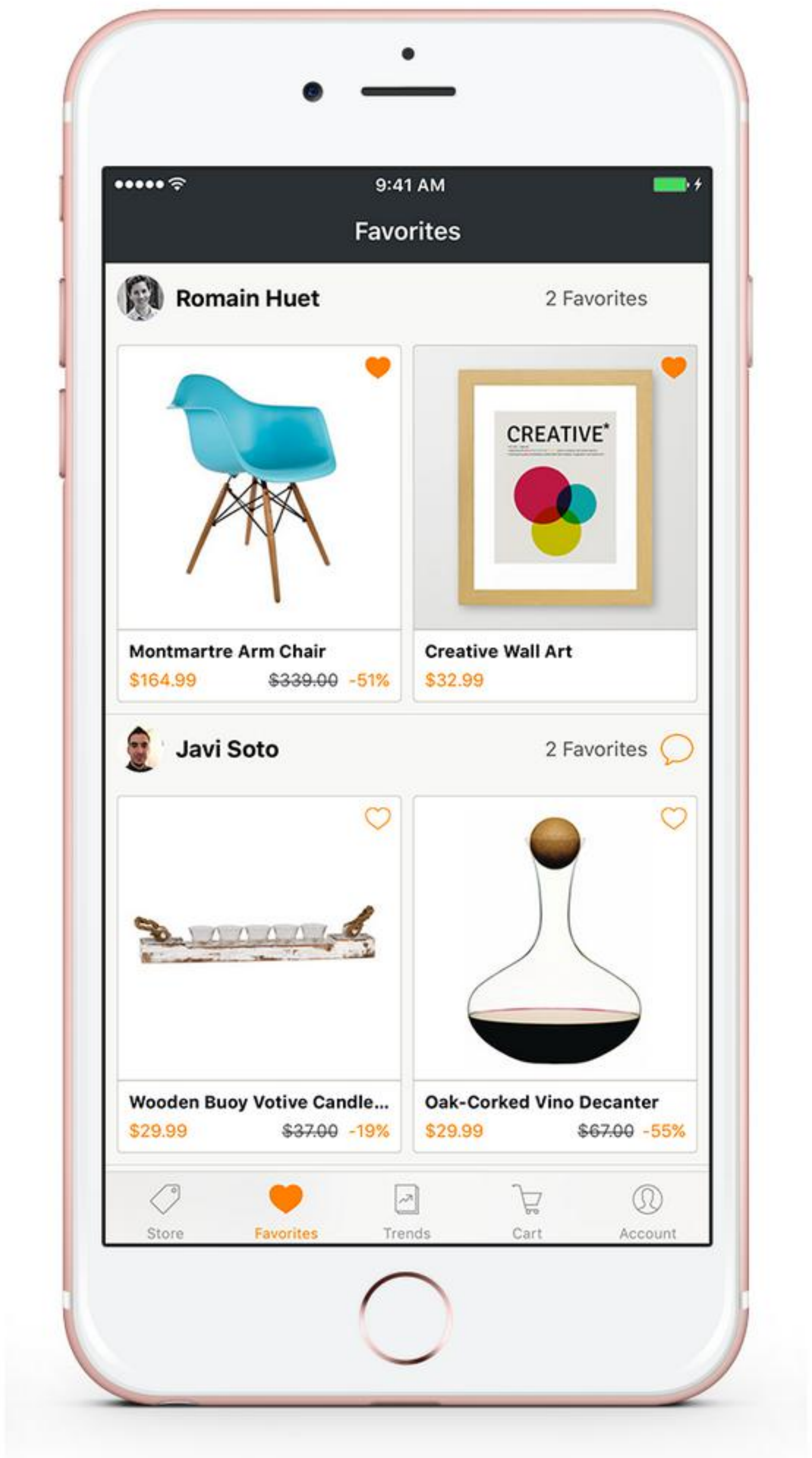
PART FOUR

Creating a backend to manage services, data, and identities

A common theme across the apps on my phone, apart from offering a way to log in, is that when I save data on them, it's persisted somewhere besides just on my device. I can tell this because when I log in from another device or from the web, my information appears. Just a few years ago, this felt magical. Remember the first time Google Maps automatically put directions on your phone to the place you just searched on your laptop? Now, it's table stakes. Apps need to store data, and people expect to manage and access that data wherever they are.

There are so many options for how to structure and build your backend that it would be absurd to try and make broad recommendations in a post like this. "Backend" comprises so many pieces — and at each level you have tons of choice: programming languages, frameworks, databases, and — unless you're building your own data centers, or expect to run everything off a machine in your garage — hosting providers.

Making all of these decisions is mostly a question of taste, skillset, and the balance between control and flexibility you want on the infrastructure. At one end you have full MBaaS (Mobile Backend as a Service) options like Parse (Facebook), Kinvey, Feedhenry (Red Hat), Azure (Microsoft), and Cognito (Amazon). These give you easy to use, native-feeling APIs for storing data to the cloud, and you don't need to manage any of the infrastructure directly. On the other end, you have IaaS (Infrastructure as a Service) like EC2 (Amazon) or Heroku (Salesforce), which give you tighter control over resource management. PaaS (Platform as a Service) choices like App Engine fall somewhere in between.



Data for our furniture items gets pulled from DynamoDB

There's no question that it's easier to use a platform than to build everything yourself. With Furni, we initially prototyped a backend with Go on App Engine, but we found so much value in saving time with AWS' Lambda, DynamoDB, and Cognito, we eventually decided to move entirely to Amazon. We wrote a simple backend in JavaScript using Lambda functions to expose the data from our furniture items stored in DynamoDB over a JSON API.

Cognito also helped solve our multiple-login problem, binding them all to one user with a unique Cognito ID. Cognito is also available through Fabric, so Digits and Twitter login are already login providers that are easy and quick to set up. We chose the structure we did based on ease of implementation and cost.

PART FIVE
Adding data from third-party APIs for fun and profit

It's truly astounding how much information is available to you, programmatically and for free, if you know where to look. As a developer advocate, I've been to a ton of hackathons — well over a hundred — so I've encountered a lot of different APIs, and seen thousands of different projects that people make with them.

Pulling in data from third-party sources can give your app depth and immediacy, but that data is rarely enough on its own. The best projects, the ones I want to see become real businesses, use third-party APIs alongside app-specific information or functionality — they synthesize multiple data streams into something new.

One of my all-time favorite hacks was a game where a set of friends each got a push notification at a random time of day, and competed to take the best photo of what they were each doing at that moment. Another great project helped people find the perfect vacation by letting them specify budget and temperature preference, then showing a map of affordable destinations together with a local Twitter stream, so you know what's happening in that city or country.

Cannonball took a lesson from the first example — start with a simple game, then make it feel more interesting and current by adding a way to interact with people, live. Cannonball is pretty fun to play once or twice, when the photos, word bank, and game mechanics all feel new. But they get stale fast, and the fun of games like magnetic poetry is sharing the silly things you wrote with your friends. Adding in a Twitter timeline with all the poems people shared kept new content in the app, and brought in some of the feel of an offline game.

Furni took it a step further, including a timeline of curated Tweets about design and furniture, but also combining the Dot & Bo data with a person's contacts list (if they chose to share it) to show a special feed of "Friends' Favorites" furniture. Even if the main catalog never changes, that curated list would probably grow, shrink, and update.

There are so many cool APIs you can use to make your app stand out. Check out Programmable Web's API Directory to get inspired, or Mashery's partner APIs and the Mashape Marketplace.

PART SIX

Making money from your mobile app

You can begin planning how you'll earn money from your app even if you're not ready to implement anything yet. Lots of successful apps have wildly different strategies — from freemium models to ad support, to direct sales of goods and services. To decide on the right strategy, you need to answer a few questions about your app:

- Can you just charge people for it outright?
- Are you selling anything, be it physical goods, in-app purchases, or subscriptions?
- Do you have a good place to put ads in your app?

There's also an important step 0: be sure you're in the clear to monetize. The licenses for some APIs may prohibit use of their content in paid apps or in conjunction with ads. I can't tell you how many hackathon projects I've seen that encounter this issue because people just don't know, so make sure that you have rights to monetize anything before getting started.

Can you just have a paid app?
When I was in college and the App Store was brand new, my dormmate built an app called Whale Songs. Whale Songs was a slideshow of photos of humpback whales paired with an audio track of them singing, about three minutes in a loop. He put it on the store for $0.99 late one night, and had made $200 by morning.

I don't think 200 people would pay a dollar for Whale Songs today — if 200 people could even find it among the million or so other apps out there. But I've certainly found plenty of apps worth purchasing: ad-free games like Threes!, utilities like 1Password, and a guidebook app chock full of maps and recommendations that works offline.

I bought each of these because their app descriptions highlighted features that solved real problems for me. If you can offer people a great solution to a problem that they have, you can charge for it.

Are you selling something?
When we wrote Furni, some people around the office joked that it looked so good we should spin it off into a full time business. We said sure — inventory management is the easy part once you've written a pretty front end.

Except, of course, it's not at all. There is a whole ecosystem of tools that have been created to help you track stock, manage payments and POS systems, and everything else that goes into order fulfillment. If you need an end-to-end solution, you have great tools like Shopify, Woo Commerce, Wix, Weebly or Magento.

If you'd prefer to handle certain parts yourself and use an API only for payments, Stripe is an awesome choice (and easy to integrate with Fabric); Braintree, PayPal, Venmo, Square, MasterCard and Visa all have APIs as well. Always check the terms of each of these providers, since some of them have specific use cases or restrictions.

The OSes themselves provide payment APIs for handling purchases — see the docs for iOS and Android. Just as with the sale of paid apps, 30% of the amount paid out will go to the stores.

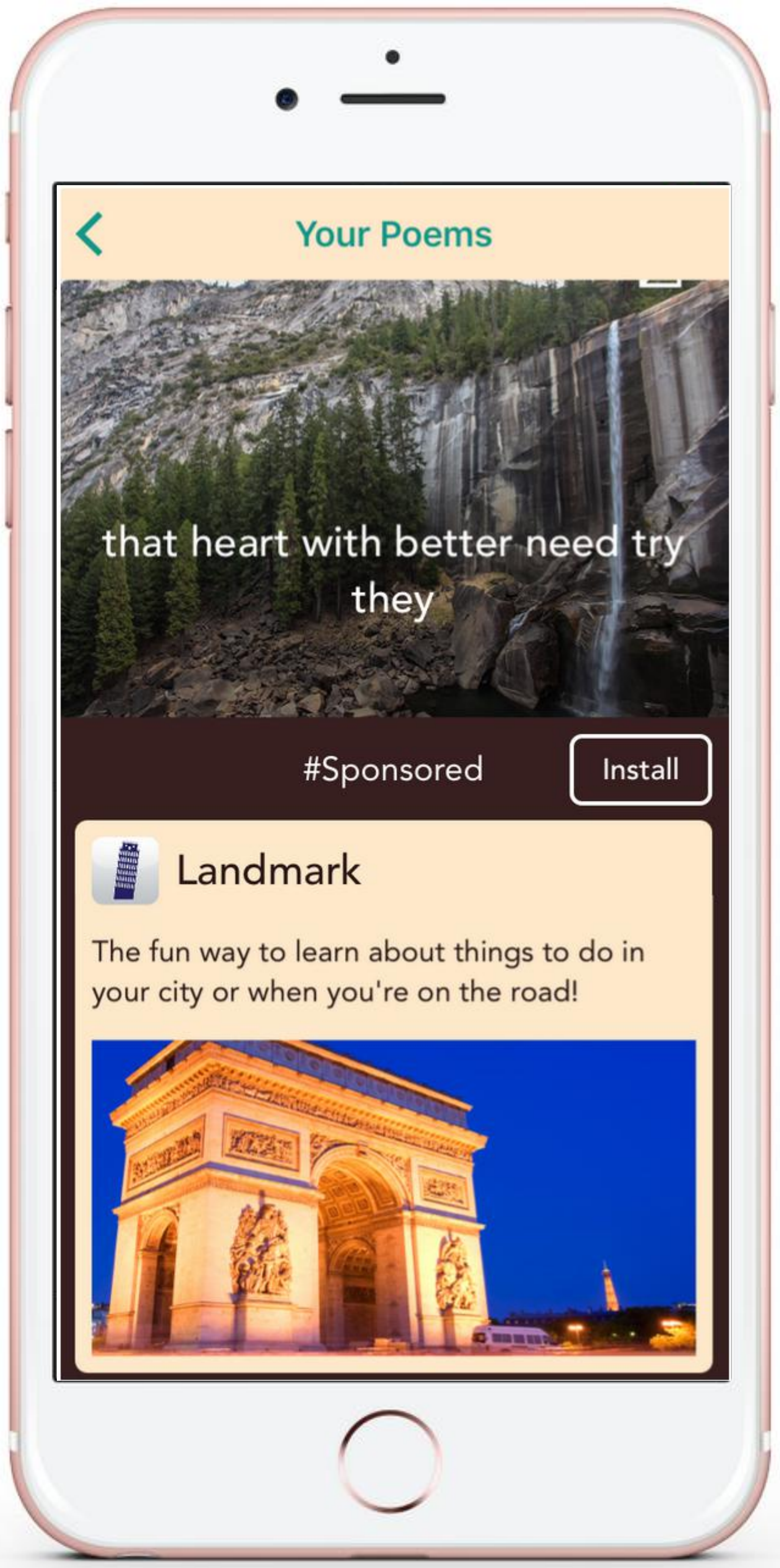
Do you have a good place to put ads in your app?
That 30% figure can be enough to turn a lot of people away from those routes of monetizing. Ads can be the easiest option, and you have a lot of flexibility to do it right. While ads can be frustrating, the good news is ad providers are just as incentivized as you are to provide a positive ads experience to the people who use your app. Annoyed people don't click on ads. Everyone makes more money when ads are well designed and well targeted. That's part of the reason behind the rise of native ads.

You have a lot of options when it comes to ads:

- Banner ads are great for their technical simplicity: slap an ad view at the bottom of one of your existing app views, turn on traffic from an ad network or exchange, and watch people click. Plenty of apps are able to make money this way, like the team from Big Duck Games, whose revenue from banners inside Flow Free has helped them start [living the dream]().

- Interstitial ads tend to command higher prices from advertisers because they get more real estate in front of customer eyeballs: the ad takes over the whole screen. They can also include video. Developers on MoPub Marketplace have actually seen a [31% increase]() in the price that they receive for full-screen interstitial videos. If you're nervous about the effect this can have on user retention, try A/B testing first. This is a good thing to do in general, since the best ads strategy for your app may be different from your initial assumptions. If it makes sense for your app, you might also want to try rewarded video, which give users the choice to engage with a video ad in return for an in-app reward, such as points or lives.

- Another option to consider is native ads. "Native" refers to the fact that the ad has been styled to more closely match the look and feel of your app's user interface, while still distinct from the app's content. It's a more visually integrated ad experience, with only a bit more implementation work to lay out creative components to match your existing views.



The fun way to learn about things to do in your city or when you're on the road!

A native ad in Cannonball

We chose to put native ads into [Cannonball]() in part because we were really excited about [MoPub's]() server side ad positioning, released around the same time as Cannonball. The MoPub SDK's helper classes made laying out and rendering native ads very easy, but up until that point, you had to ship your app with client side code determining how frequently ads would appear in your list view or table view. Server side positioning was a change that let you update those values from the web — so we could bake a future ads experience into our first release, then turn it on from the web when the time was right, without needing to ship a new version of the app to the store.

At the end of the day, don't be afraid to experiment a little bit. It may take you a little while to hit on the perfect balance of revenue sources for your app, but it gets you that much closer to making it a real business.

PART SEVEN

App marketing for developers

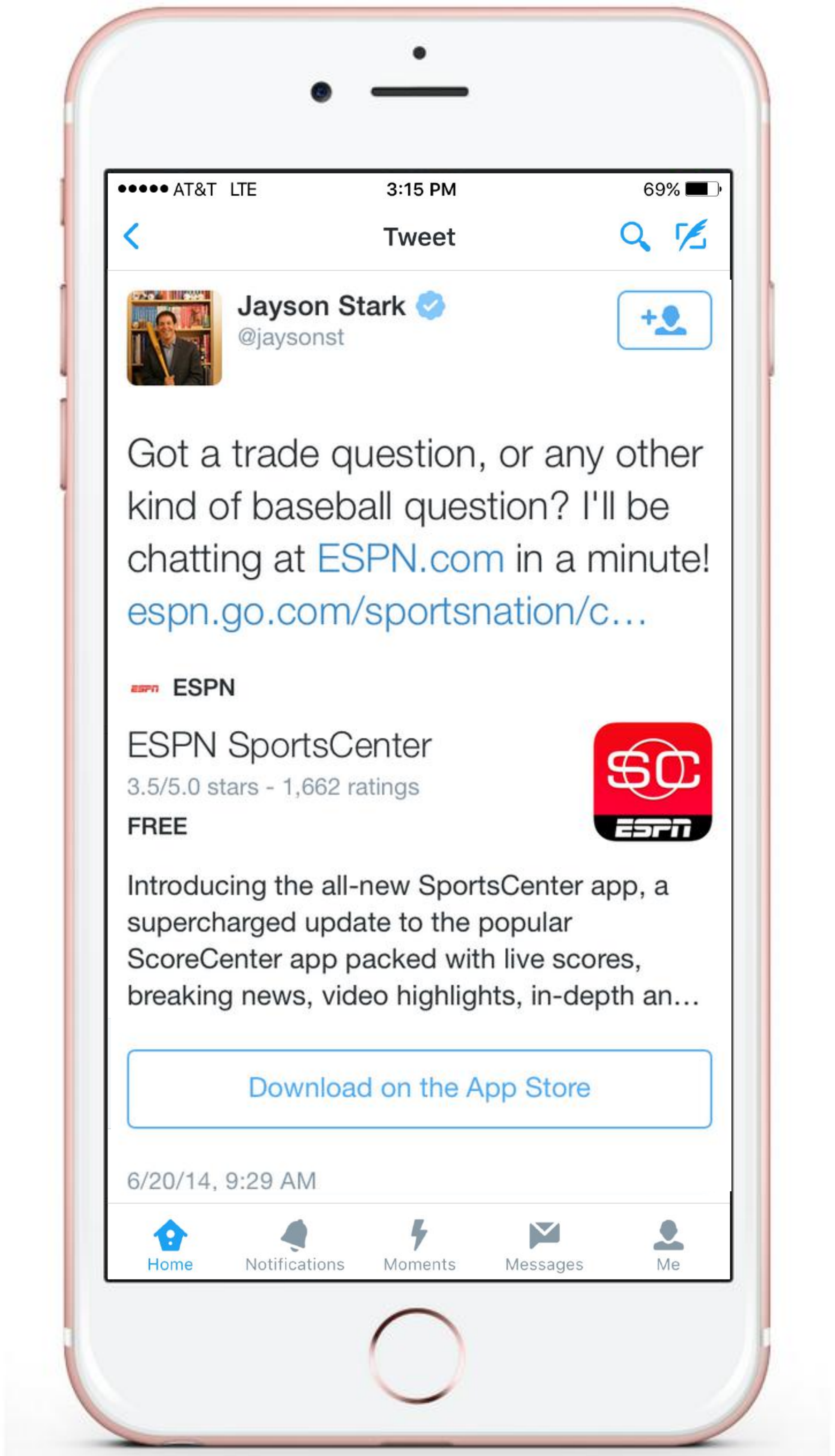
Marketing your app is incredibly important. You can work hard to build the best one possible, but if no one knows about it, can it really be the best? Marketing encompasses several pieces: overall product marketing, researching your customers' needs, branding and positioning, creating a communication strategy, and much more. All of these steps are critical to your app's success. When you can, hire someone, or a team of someones, to work on these tasks full time.
Assuming you haven't had any such help yet, here are some top marketing hacks you can pull off on your own when launching your app:

Double down on social integration
When people share content from your app on social sites, they become your advocates both for the app and for your brand. Social sharing is free, and can be highly effective. In a survey from ConversionXL, 82% of users said they'd consider trying a new product if someone they knew recommended it. The Share Sheet on iOS or a Share Intent on Android are so simple to implement that you really can't lose.

The secret sauce for creating a great sharing experience is to think about what you'd like to see in a social feed.

Which content in your app would you yourself click on, reply to, or reshare? Maybe a screenshot featuring your game's incredible graphics, or maybe a surprising headline from a news story. People share to social networks in the hope that their friends will like, comment, and interact. Design your sharing experience to encourage that interaction.

Use Twitter Cards and Open Graph to make your content shine
Posts with photos and links with extra details get more real estate in a feed, and are more eye- catching than a plain-text share. When you include extra information about what people are sharing from your app, you ensure that it looks its best inside social feeds.



Jayson Stark
@jaysonst

Got a trade question, or any other kind of baseball question? I'll be chatting at ESPN.com in a minute! espn.go.com/sportsnation/c…

ESPN

ESPN SportsCenter
3.5/5.0 stars - 1,662 ratings
FREE

Introducing the all-new SportsCenter app, a supercharged update to the popular ScoreCenter app packed with live scores, breaking news, video highlights, in-depth an…

6/20/14, 9:29 AM

Card markup makes the Tweet more visually interesting and adds a Download button for your app

Make sure you have deep linking setup

Once people begin posting content from your app to other sites, you need a way to capture traffic back to your app from those shares. A basic deep link setup could be as simple as adding markup to a Twitter Card to making sure clicks and taps from Twitter lead people to the store or open up your app; check out the instructions for iOS and Android. An even better deep link scheme will use specific tags for content from your app.

It's not a great experience if I click on a clip from an interview I want to listen to, but when I open your app, I am only sent to a main page. Take the time to make the clickthrough experience great.

Create some web presence, even if just a landing page or social account

You want your app to be as discoverable as possible. Since search engines can't easily index content tucked away inside a native app, create a place on the web where people can find you. When someone searches for your app, you want the top result to be something you wrote and created so that you have first-line control over messaging and perception of your offering.

At some point, you will need to buy ads

Unpaid sharing, word of mouth, and media mentions can take you pretty far when it comes to growing your app's audience — but there will come a time when buying ads will become a key part of your strategy to continue growing. Ads can serve a lot of purposes: drive people to download your app, remind people to keep using your app (re-engagement ads), and prompt people to do other things that will keep them interacting with you, like driving follows to your social accounts. Make sure your ad copy has a clear call to action, and that you are matching the right ads and right goals with different audiences.

Make re-engagement ads a part of your strategy
It bears repeating that ads can serve a lot of purposes. Downloads aren't the be-all and end-all of building a successful app. To sustain growth, you need people to keep using it. Show ads to people who are already using your app to remind them of the value they got from it originally, and to encourage them to come back.

Experiment with your creative
After a lengthy brainstorm session, you may feel like you have the best possible copy, imagery, and call to action to get people to try out your app. But your creative is not a set-and-forget process. It's important to A/B test your copy and creative. The results may surprise you. For example, app install ads often perform better when the ad image has real people and a mobile device in it. Even if you've got an ad that is doing really well and sending a lot of traffic your way, it's important to refresh your creative and messaging so that it remains fresh and compelling.

PART EIGHT

Progressive improvements and testing

There's always lots to work on in any app: fixing bugs, or starting to use new tools and OS-level APIs; improving load times and resilience under poor network conditions; making accessibility and internationalization updates; or experimenting with new layouts. Often, there's a business goal behind what you're doing, whether it's explicitly stated or not. Fixing bugs is a good thing to do for its own sake, but it's also key for retention, since people tend to delete or stop using slow, crashy apps. Proper internationalization is important for adoption because many people don't use apps that aren't available in their country or don't work in their language. Tweaks in user experience and design can have massive impact on engagement or purchases.

Whenever you make significant changes, it's important to keep track of how they're affecting the people using your app. Talking to your customers, in person or online, is always important. Feedback conversations provide nuance and details that you can't always capture with a few top-line metrics. You can and should use qualitative research to shape your approach to customers, but you can't always be in touch with each segment of people that's using your app. Logging metrics about the changes you make is critical, because analytics are user feedback at scale.

When you're ready to start moving past your MVP and shipping major updates, we have a few suggestions for how to do it right:

Establish a baseline
It's hard to measure the impact of changes you make if you don't know how you're doing to start with. Before going into any experiments, you should already know key numbers like your current DAUs, MAUs,

retention rates, and conversion rates on events like purchases or social shares (bonus: our free analytics tool Answers gives you these stats in real time).

Establish a baseline
It's hard to measure the impact of changes you make if you don't know how you're doing to start with. Before going into any experiments, you should already know key numbers like your current DAUs, MAUs, retention rates, and conversion rates on events like purchases or social shares (bonus: our free analytics tool Answers gives you these stats in real time).

Decide if your change actually needs to be user tested
You can safely assume that you should just fix bugs, or just ship a version of your app that works with a screen reader. You should certainly try and measure the effects that these types of change have on your business metrics — but unless it's a significant change to the way your app operates, you may not need to spend significant time testing it out first.

Take the time to actually construct a hypothesis to test
What do you think might happen after you make this change? How will you know that it has or hasn't happened? What do you consider success? Measure the effect you think it will have, but also keep an eye on your other key metrics — you may notice unintended effects alongside the ones you were testing.

Check the math
As in other parts of this series, we're assuming that you're operating lean and don't have a whole data science team behind you to help create and manage these tests. Here are a few important things to keep in mind when you do that:

- **Make sure you have substantial minimum sample sizes.**
  Whether you're running an A/B test or a multivariate test, you want your results to be statistically significant — that is, you want to be clear that the results you're observing are due to a real difference between the two samples and not just due to random variation. Think about the average traffic your app or site gets, and get a sense of how long it's going to take to collect enough responses to draw a conclusion (1,000 is a reasonable lower bound on responses, but you can use tools like this handy calculator to see what minimum sample you should collect).

- **Pay attention to how daily, weekly, or seasonal usage of your app can affect the sample of people you gather for the test.**
  Most testing methods assume that you have a random sample of your users. For example, if you run your test only during the weekdays, you may be skewing the sample set by excluding people who only use your app on weekends. Consider running tests for at least a full week.
- Be careful when running simultaneous tests.

Tests can affect one another, particularly if you're running the test with the same group of people, and it can be difficult to disentangle the results. If you want to run tests simultaneously, try to pick independent components of your app to test.

Running a statistically robust experiment isn't as simple as it looks on the surface. This blog post from ConversionXL gives some useful details on common mistakes people make, including ignoring validity threats, and increasing chances of false positives by testing too many variations at once. A/B testing frameworks like Optimizely help take a lot of the guesswork out of running tests and bake in some best practices around measurement and interpretation of the results, so you can make better decisions. There are also lots of great tips from around the web about running robust tests for your business.

PART NINE

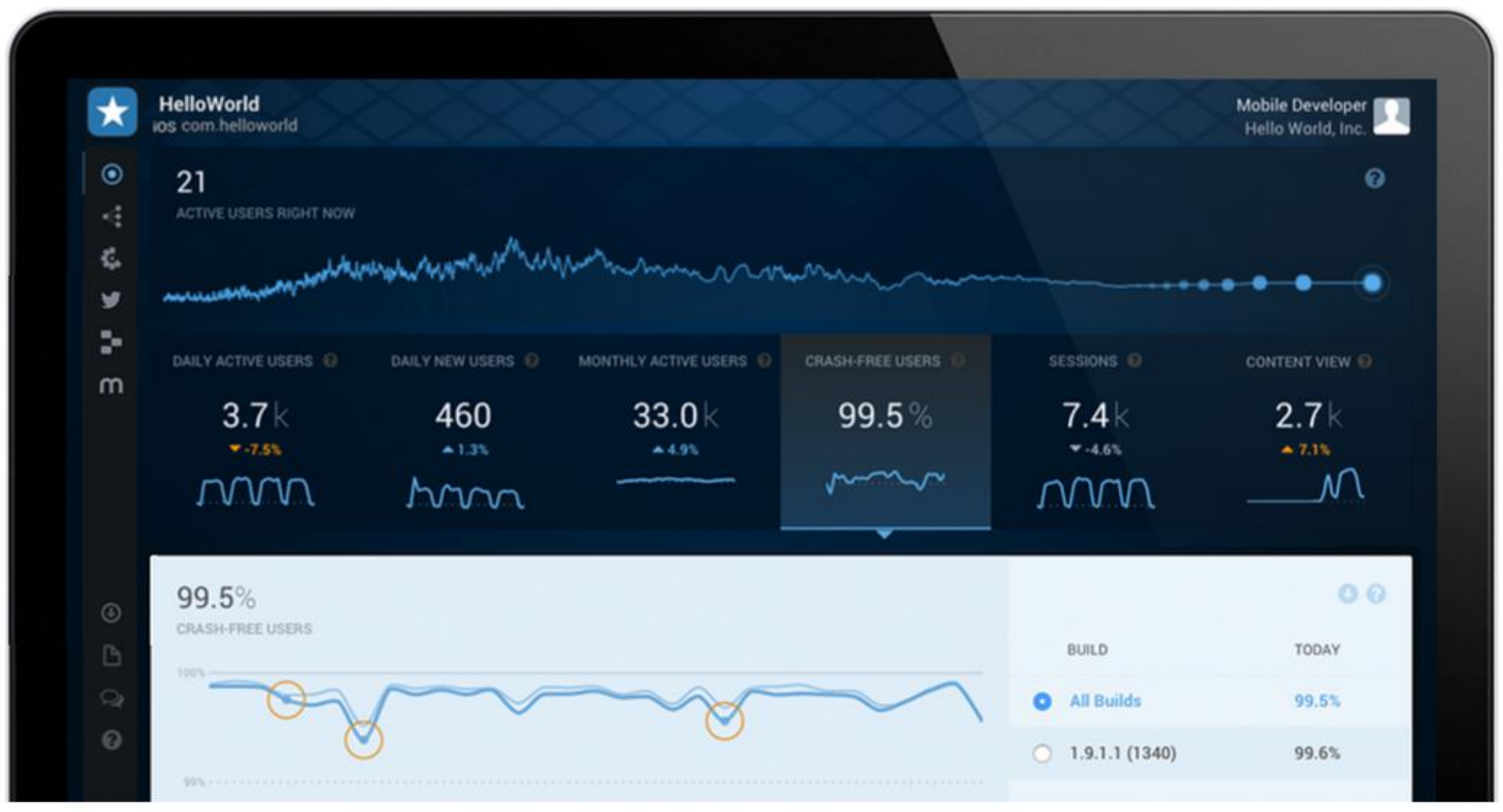
Analytics and what to track

We find that there are three common use cases for app data: daily checks, investigations, and ad-hoc analysis.

Daily checks

The goal of the daily check is to make sure things are trending as expected. You may be paying attention to numbers like user growth, retention, or a stability rating related to the total number of crashes you're observing. Looking at your top level metrics should give you an instant understanding of how your app is doing — so decide what matters and choose accordingly. For example, a task tracking app may be most concerned with getting people to visit every day, while a fitness app may care most about workouts completed, and having daily logins may not be a reasonable goal.

Daily checks should be quick and simple, taking no more than a few minutes of your time each day to confirm that everything looks as it should. When we built the Answers dashboard, we consulted app builders to see what they wanted in their daily check dashboard. You now see Monthly Active Users (MAU), Daily Active Users (DAU), crash-free users, and users in-app so you automatically have the "daily check" level information at a glance.



The Answers dashboard

Investigations

With any luck, those key metrics will always be trending in a positive direction — but that's not always the case. That's when you need to dig deeper to figure out if the problem is a temporary anomaly, or a long-term trend that requires attention.

For instance, let's say total workouts completed is down. This could be due to fewer people coming to the app overall, a smaller percentage of those people actually logging a workout, or people's activity naturally taking a dip (e.g., during a blizzard). Your team's intuition and customer knowledge will be key in coming up with any hypotheses.

When you're trying to assess the root cause, segmenting the data in different ways can help you mathematically isolate the source of the deviation. In this case, we'd start with the simple case: is DAU down overall? If it's not, we move on to the next hypothesis: is the ratio of workouts to users overall down? Once you find the probable source, dive into metadata: are there any commonalities (e.g., geolocation, network strength, user age group). Once you have a handle on the root cause, you can decide what needs to change, if anything.

Ad-hoc analyses

Ad-hoc analysis is the process you go through to learn more about how people interact with a specific part of this app. It would answer questions like: How can I improve this flow? Why aren't people purchasing as much as I'd like? If people complete a search, are they more likely to come back?

Doing this type of analysis usually requires having some interface to explore and segment the data. Top-level growth metrics (DAU, MAU, number of sessions) give you a surface-level idea of how things are going. But to make decisions about how to improve the app, you need to be able to segment your user base across multiple lines.

For example, you may assume that people who are in the app three times a week or more are getting more value out of it than people who only visit daily. If you want to find out what makes that group different, and how you can grow that group, you first need to be able to determine the unique characteristics of the group. Then you may want to add in-app events that will let you create different segments you think could be interesting. In our extended example, this may include:

- Whether or not people share workouts on social media
- Whether they do indoor or outdoor activities
- Whether they participate in team sports or individual sports
- Geolocation when the user provides it
- Things like profile completeness that may be a useful indicator of how invested they are in the app

So how do you pick the activities you should be tracking for later ad-hoc analysis? Typically, you want to consider user activities that define "success": likes, shares, content views, purchases, and so on. You may also want to consider events that are more UX-driven: clicked this button, or adjusted this setting.

UI events can be helpful information, but can also provide a lot of noise — be judicious about how many of these you track so you don't get lost in the data.

And if you do feel lost in the data, remember that analytics are customer feedback at scale. If you can't seem to get the initial hunch about what to track, or what might be affecting your top-line metrics, go and find some of your customers to talk to. Have an open-ended chat about how they use your app. Let them tell you — or show you — how they use it, what they like, what they don't, and what confuses them. The better you understand your customers, the easier it will be to develop hunches and ideas about their behavior that you can test with analytics.

PART TEN

Streamlining your workflow

There are a lot of decisions you need to make about managing your codebase — what kind of version control are you going to use? How do you manage contributions? Who is responsible for code reviews? Who decides which changes get merged and how do you manage the release? All of this is likely to evolve as your team and your code base grows. What works for a team of four may not for a team of 60 — and vice versa.

Repos

One size doesn't fit all when it comes to how to organize and manage your code. Multiple repos may work well; alternatively, there are also some major benefits to keeping everything in a single, monolithic repo, including simpler dependencies and organization. One size doesn't fit all, and there may come a time when you have to make changes; expect that to happen eventually.

Cadence

You'll need to establish a release cadence on your team. How often do you cut a new release branch, and how long does it get tested before release? Are you doing feature-based releases or are you sticking to a regular time interval? Different types of cadence may be appropriate with different team sizes and stages of the product. Be careful about releasing too frequently; every time you do, you're asking for action from all your users to stay updated. Automatic updates have made this more seamless, but you should still keep this in mind. Our Android engineering manager Jan Chong gave a great talk at Flight about how Twitter's thinking evolved on this as our team grew.

Deploy process

You're going to need to set up deploy processes, which may have many moving parts. Unlike on the web, where a deploy is more or less totally in your team's hands, on native mobile the store affects your build process because it has update protocols and timelines you need to conform to. A native mobile deploy involves a lot of actions in sequence:

- Building and packaging your app, which includes many substeps like committing version bumps and managing code signing
- Handling distribution to beta channels or the store
- If you're distributing to the store, updating your metadata and app information like screenshots, as well as uploading the new version of the packaged app

If you're doing this, you're probably already using multiple tools to get it all done — and that can get messy. Our goal is to simplify developers' lives and save people time, so we were excited to announce at Flight that fastlane was joining the Fabric family.

Fastlane is an open source tool that helps you define and run deployment pipelines, or "lanes," for different environments. You define each lane in a Ruby file called a "fastfile" that sets up a series of steps that every deploy should take. Think of it as a runbook for your deploys — it's documentation of how things work, so everyone on your team knows what happens in an app deploy, and the code to actually deploy. You can check the fastfile into source control so everyone on the team can use it — and you can produce more consistent results.

Fastlane integrates with other CI services and build tools you already use, like Gradle and CocoaPods, and is installed by default on most major mobile CI providers.
We've put lots of work into fastlane because we believe in making robust, flexible tools that save us all time. Fastlane is open source and accepting pull requests — and we'd love for you to contribute! Whether you'd like to see new third party integrations or you want to build new tools on top of Spaceship, the Ruby library for connecting to Apple's Developer Center, we'd love for you to share it with the community.