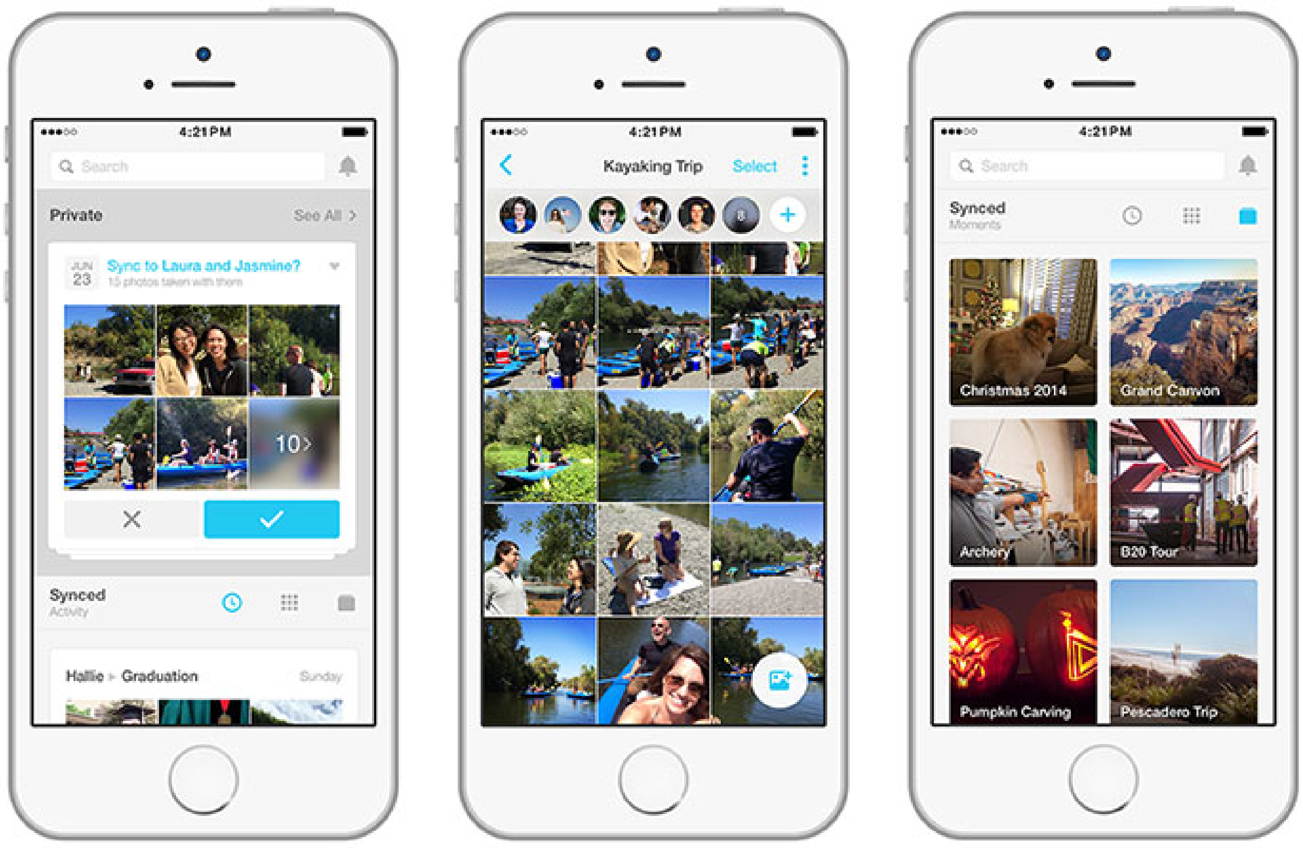
June 15 IOS · ANDROID

Under the hood: Building Moments

Ashwin Bharambe Zack Gomez Will Ruben

These days, with a phone in nearly everyone's pocket, it's easy to capture photos of the moments in our lives. But, somehow, you rarely end up getting all the photos your friends took of you. It's still cumbersome to share lots of photos privately with the friends who were there. We've all had to take that same group shot on multiple phones, because everyone wants to be sure they'll get a copy! And even if you do end up with some of the photos your friends took, keeping them organized and in one place on your phone can be a challenge.

We set out to make it easier for you to get the photos you didn't take, and we ended up developing an app we call Moments. Our objective throughout the development process was to create something that empowers people to exchange photos with their friends. We sought to eliminate as much of the friction as possible, while still ensuring that you stay in control of the photos you take.



Choosing an approach

At the beginning, we weren't tied to any particular solution, so we explored various ways that we could suggest which of the photos on your phone you may want to give to specific friends. We experimented with several different technologies — Bluetooth, location, and facial recognition among them. Bluetooth wasn't ideal for a few reasons. Some reasons were practical — for instance, friends who wanted to share photos would have to enable Bluetooth on their phones at the same time. Other reasons were more technical. Android and iPhones don't cooperate very well over Bluetooth without the presence of a beacon. Location presented its own issues. It worked great for pinpointing which friends are in the area, but in crowded or densely populated places, it was not precise enough to suggest the specific people with whom you'd want to share your photos.

In the prototypes we built, facial recognition produced highly accurate, actionable suggestions. If friends of yours are recognized in a photo you take, that's a signal that they probably want the photo. If you took other photos around that time or at the same event, they may want those photos, too. Your friends may want to give you the photos that they took around that time, as well.

We were fortunate to be able to leverage Facebook's existing facial recognition technology. Recognizing our friends is something that we can do easily as humans, but it's a complex problem for computers. Only breakthroughs in recent years, some developed by Facebook's AI Research team, have made this technology something that can be really useful to people.

For Moments, we built on top of the latest work by Facebook's AI Research Lab that was already powering tag suggestions on Facebook. This work includes one of the most accurate facial recognition systems in the world, matching human-level performance. You're in control, because you can always choose to turn off tag suggestions for photos of you in your **Settings**. We've also designed Moments so that your private photos and the friends who are recognized in them are not backed up on our servers until you choose to sync them to friends. Moments does not need to back up your private photos in order to work.

Rapid iteration

Building an easy way for groups of friends to share groups of photos was our team's North Star. We knew we had the right technology, but we ended up iterating a lot to find a UI that unlocked the possibilities in the technology. When building the app, one of our goals was to shorten our development workflow and test our latest features with real users almost every week. (You can read more about the process of designing Moments **here**.)

To produce and test a new feature set that often, we looked for development-process optimizations anywhere we could find them. Business logic for apps is typically split between server and client, and as the product evolves, it requires changing both the client and the server. However, managing two fast-moving systems is significantly harder than managing a single fast-moving one. Moving server logic to the client would enable us to focus solely on the client and also to make the development-run-test cycle as short as we could.

From the start, we wanted to make our application incredibly responsive. That meant avoiding as many “waiting-for-network” spinners in the UI as possible. This is achievable if updates made by the user are represented in the UI optimistically (in other words, before they are saved to the server). In our experience, building an optimistically updating UI as an afterthought is time-consuming and error-prone. We wanted it to work by design and by default.

Given the requirements of a client-driven development model that supports optimistic updates by default, we decided on a system with a few properties. The data model would be described by the client, and the server would store generic, untyped data. The client would hold a complete working set of typed objects, including optimistic objects that have not yet been written to the server.

To get a feel for the details and ramifications of these broad decisions, let's follow the data in a sample use case: one user, Alice, syncing a single photo to an existing moment with another user, Bob. Alice creates two data model objects — a photo and a notification. Here is a simplified definition of our notification data object:

```
"notification" : {
  "type": "string",
  "senderUUID": "string",
  "recipientUUID": "string",
  "momentUUID": "string",
  "photoUUIDs": "string", // JSON encoded array of photoUUIDs
  ...
},
```

Alice now sees this newly synced photo appear in the feed, as well as in all other (current and future) interfaces where this photo appears. We use a **Flux**-like architecture to flow this data from the client cache to UI. The notification and photo objects are optimistically persisted in the client cache as strongly typed objects and combined with other existing user and moment objects into appropriate view models. The logic to create view models works identically for optimistic and server persisted objects.

Once persisted in the client cache, these optimistic objects are converted (via code generated from the model definitions) into generic blobs containing an object type and payload, and then are sent to the server. The server reconciles and stores these untyped blobs, and returns them as the “server truth” versions. Once again, generated code converts these blobs into typed client objects and replaces the optimistic objects in the client cache. This allowed us to rapidly iterate while maintaining type safety for the data models.

When Bob opens the app, the client initiates a fetch of new objects. Many new objects are created every day, and by a variety of users. Bob is interested only in a small subset of those new objects. This interest is expressed by registering a subscription to *queues*. Users that are members of a queue can see all the objects associated with that queue. Objects are associated with a queue when they are created, but you cannot change this association later. For example, each moment in our app has a queue representing it. Users participating in the moment are its members. All photos synced to this moment are placed in this queue. The client uses a delta-syncing protocol, expressed as a GraphQL query, to download objects that have been created or modified since the last query. In this case, Bob downloads only the new notification and photo object created by Alice. This system allows the client to efficiently download the exact set of data needed, while keeping all UI code completely unaware of data fetching.

This design eliminated the majority of server changes as we added and modified features on the client. We were able to successfully shorten our development workflow, producing and user-testing a new milestone almost every week. Additionally, without any extra effort, all parts of our app continued to function in bad network conditions, as well as in airplane mode.

Building on two platforms

Our goal was to simultaneously ship iOS and Android apps. We started our iOS client before our Android one, and we needed to accelerate our Android efforts. To optimize our development velocity, we needed a way to share this code on both platforms. Without the server as the shared business logic, that logic needs to be written for each mobile platform.

There are many alternatives for sharing code between mobile platforms. We wanted to optimize for fast iteration, app performance, and native look and feel. After weighing the alternatives, we chose to write the UI in platform-specific code and the business logic in shared code using C++. Traditionally, C++ is known for providing high performance while lacking easy memory management and higher-level abstractions. However, using modern C++11 features such as `std::shared_ptr` reference counting, lambda functions, and auto variable declarations, we were able to quickly implement highly performant, memory-safe code.

To keep our C++ API boundary simple, we once again appealed to the Flux model and adopted one-way data flow. The API consists of methods to perform fire-and-forget mutations (e.g., `genSyncPhotosToMoment`) and methods to compute view models required by specific views (e.g., `genAllPhotosSyncedFromUser`). To keep the code understandable, we write functional style code converting raw data objects into immutable view models by default. As we identified performance bottlenecks through profiling, we added caches to avoid recomputing unchanged intermediate results. The resulting functional code is easy to maintain, without sacrificing performance.

When using a shared C++ library, you need to generate bindings between C++ and platform-specific code. On Android, we leveraged **Djinni**, developed and open-sourced by Dropbox, to convert view models from C++ to Java. We rewrote its code generator to create Java models that are Immutable and Parcelable. To avoid creating many objects that the Java garbage collector would have to deallocate, we cache weak references to converted Java objects and regenerate them only when they've changed. For our iOS view models, we take advantage of Objective-C's closeness to C++ to directly interact with the C++ view models via lightweight wrappers. These designs let us focus on the business logic and UI without worrying about boilerplate code to connect the two.

Today, the vast majority of our business logic is shared between our iOS and Android apps. Counting by lines of code, roughly one-third of the codebase on each platform is shared C++ code. As a result, we have been able to create new features with far less work and fewer bugs while retaining native look and feel and fast performance. This has also enabled us to more flexibly allocate engineering time between the two platforms, allowing us to ship on both platforms simultaneously.

We're really excited to launch Moments today in the U.S. on both iOS and Android. We hope you find the app as useful as we do, and we look forward to hearing your feedback. We will continue to move quickly, of course, to ship you improvements to the core Moments experience and some exciting new features. We're also excited about the potential for Facebook's continuing AI research to open up new, useful experiences. You can learn more about how we think about AI research in the video below.

Like Share 305 people like this. Sign Up to see what your friends like.

More to Read

Infer

Recommended

Under the Hood: Building Facebook Camera

Under the Hood: Building and open-sourcing the Rebound animation library for Android

Under the Hood: Building and open-sourcing fbthrift

Under the hood: Rebuilding Facebook for iOS

Want to work with us?

Join the team, we're hiring! Here are some of our current open positions:

Software Engineer, Data Infrastructure Engineering
Software Engineer, Payments
Software Engineer, iOS

More Engineering Positions

Connect

Like 1,304,614 people like this. Sign Up to see what your friends like.

Follow us on Twitter

Keep Updated

Stay up-to-date via RSS with the latest open source project releases from Facebook, news from our Engineering teams, and upcoming events.

Subscribe