MealUp

Final Report

Andrew Zeng
Annie Zou
May Jiang
Chi Yu

Initial Planning

A. Timeline

We went into this project without any experience with the languages and systems we planned to use, so we made our timeline with a few concrete goals and left lots of time for testing. Overall we followed our timeline quite well. At the beginning we were a bit behind because we were still figuring out how to use the languages and systems, but after week 3, our productivity increased and we were able to finish most of our goals ahead of schedule and publish the app on app stores, which we had not planned to do. By having a minimal viable product done early, we had plenty of time for alpha and beta testing on users.

B. Design Planning

One step that our group took was to design a low fidelity prototype of our app using the application Sketch. We accomplished this before our meeting with Professor Kernighan. The prototype was just multiple images that were linked together, but Professor Kernighan thought we had programmed the whole app and was giving a demo of a very basic version of our app. By creating a design prototype, we were able to gain a clear overview of the scope and complexity of our app. Thus splitting up the work was easier as we could refer to specific components/features needed to be coded for a screen. If we had more time, we would have liked to create a high fidelity design prototype and also do more user testing with our designs. We made quite a few design changes later on during the coding process that resulted in some unnecessary coding.

While creating our design document we had a rough plan of our data model for our database, but we should have spent more time on this part. We were too eager to start coding, so throughout the development process we ran into many roadblocks because we hadn't figured out the specific details of our data storage model. We also did not consider performance and efficiency while planning our app, particularly with respect to querying and updating the database. This resulted in a particularly difficult challenge we had to solve later on. Overall a more detailed design document would have increased our efficiency and productivity during the development process.

Design Decisions

A. User Interface Design

Knowing that we were going to develop a mobile application, designing a clean, user-friendly interface has always been one of our top priorities. During the development phase, we often needed to make decisions on the format for displaying information to the users. Some of the questions we considered are as followed:

- Do we use horizontal scrolling or tabs for the different days on the freetime screen?
- How do we display the scheduled meals in an easily understandable way?
- Should we sort the requests by sent time or time of the meal?

Since user interface is ultimately designed for the users of the app, we decided to first collect our test users' thoughts on these questions. From their feedback, we arrived at the conclusion of minimizing the number of hand gestures needed for users. As a result, we ended up choosing horizontal scrolling over using tabs. We also decided to unify the sorting of pending requests with the sorting of scheduled meals (both sorted by meal time) to avoid confusion.

Regarding the actual implementation of the user interface, we mainly used the built-in components in React Native, such as flexbox and scrollview, along with components from the React Native Community, such as navigation bar and swipers. A shortcoming for our user interface implementation is a lack of consistency: some components were coded in separate files while others were coded within the logic of the corresponding screen, this made our code difficult to read and debug. If we were to start the project all over again, we would create a separate file for each UI component to increase modularity and clarity.

B. Database Design

We used Cloud Firestore from Firebase as our database for the app. The database itself is organized in an alternating sequence of collections and documents. We decided to have a single collection for all the users, and a

document for each user who has logged into the app. For each user, there are different subcollections containing relevant information about the user, such as free time, friends, groups, requests, and scheduled meals.

One key design decision was to store free time as an array, with each index denoting a 30-min time slot. We decided to use an array as opposed to storing only the free time slots as string variables because the array implementation made matching availability with a friend much easier. Originally we used a boolean array, where true indicated free and false indicated not free. However, soon we realized that we needed a different state to show that a time slot has a scheduled meal. Therefore, we ended up switching to an integer array with 0, 1, and 2 representing the three possible states.

Languages and Systems

A. React Native & Expo

We used React Native to develop our app, so all of our code was written in JavaScript and JSX, a syntax extension to JavaScript. We only had to learn one new programming language but the learning curve for JS is a bit steep. Make sure you understand how the “this” keyword works, asynchronous functions, as well as the scope of variables. We had many issues with these small things because we didn’t learn JS very well. Overall, using React Native made coding and styling the UI and components simple.

The Expo toolchain also made development and testing easier. The Expo Client app and Command Line Interface tools are described below in the Testing and Publishing section. Expo also provides a well documented SDK API reference for utilizing native cross-platform device APIs which made implementing Facebook Login, push notifications and export to calendar features of our app easy.

Another issue to keep in mind is Android and iOS platform differences. Although React Native allowed us to develop a cross-platform app using only JavaScript, some React Native APIs only worked on iOS, so we had to find other open source modules to implement some Android features. Both platforms also use different design systems (Flat vs Material), so we implemented platform specific UI wherever necessary.

B. Firebase Cloud Firestore

On the first week of the project, we found out about Firebase, a mobile development platform from Google that provides many services. We decided to use their Cloud Firestore service as our database. This saved us a lot of time because we didn't have to set up our own server. Querying and updating the database required simple API calls in client code.

While Cloud Firestore saved us a lot of time, we ran into an issue with it later in the development process because Firestore provides a NoSQL database. This was problematic because our app was intended to support sending meal requests in two different ways: request by time and request by friend. However, since a non-relational database has a hierarchical structure, the request by time method was inefficient because we had to query that database many times to get the free times for all friends of a user. We ended up creating an extra collection called FreeFriends for each user. This collection stores which friends are free at each time slot, so the request by time algorithm would only have to query the database once.

Another limitation with Cloud Firestore was that the free plan only allowed a limited number of reads and writes from the database per day. When a new user first logs into our app, it requires many writes to the database to create all the documents. Therefore, if many new users downloaded our app in one day, or if the number of active users of our app gets sufficiently large, the app would most likely crash. Given these limitations, next time we would consider using a SQL database such as MySQL.

C. Firebase Cloud Functions

As noted above, one design dilemma we faced was to store the user's free time on a NoSQL database such that both request by time and request by friend could be implemented efficiently. In the end, we decided to have two separate collections: one for storing all the user's free friends at each time, and one for storing whether or not the user is free at each time. However, a new problem arose: whenever a user selects or deselects a time, the app has to update both the Freetime collection for the user and the FreeFriends collection for all of the user's friends. Since multiple writes to the database are required for this, there was a significant lag in re-rendering the UI.

Our solution to this problem was Firebase Cloud Functions, a feature that automatically runs backend code on Google's servers whenever a specified

portion of the database is updated. Specifically, we wrote a function that updated the FreeFriends collection for all of a user's friends whenever a user's Freetime collection was modified. By moving a significant amount of database calls to a backend, our app achieved better performance and we implemented a backend without setting up our own server! In addition, we used Cloud Functions to detect requests that have a time conflict with a scheduled meal.

We would highly recommend considering using Cloud Functions for mobile app development because it allows the app to run more efficiently on the client side and requires minimum effort for developers to set up a backend.

Testing and Publishing

A. Expo Client App

Testing our app on physical mobile devices during development was very easy using the Expo Client App for iOS and Android. We simply served an instance of our project from our local computers which generated a development URL that we opened in the Expo Client on our phones to test and debug our app. React Native also offers many useful tools for development such as remote JavaScript debugging in Chrome, live reload, and hot reloading. Hot reloading was particularly useful in testing different UI designs.

However, even with these useful tools, testing was still a big task that we should have spent more time on. Our app has many screens and components that do a lot of database interaction and navigation. We did a lot of self-testing as well as user testing, and did some automated unit tests, which caught a few minor bugs, but there were still bugs that we didn't catch until we released the app to real users. We would have liked to put more effort into automated testing but we prioritized publishing the app to get real users to use it.

During alpha testing, we received a lot of user feedback on the UI design of the app. We received complaints about our original free time screen design. We iterated on the design of this page and surveyed users until we decided on a design that most people enjoyed using. Beta testing with more users helped us fix screen size issues on different phones as well as fix many bugs relating to scheduling group meals, which was much more complicated than meals with only one friend.

B. Distributing and Deploying App

Originally we weren't sure if we would be able to publish our app to the app stores; however, because we produced a MVP ahead of schedule we decided to publish. Unfortunately it cost us $99 for an Apple Developer Account and $25 for a Google Developer Account, but once we got accounts the actual process of publishing was easy. Expo makes it easy to create a standalone app binary for both Android and iOS using its command line interface. Once we downloaded the binaries, we submitted our app to both iTunes Connect and the Google Play Console. Google was much quicker in accepting APKs for testing and production. Apple's review guidelines are more stringent and the review process for our app took 2 days. Because we did most of our testing using the Expo Client app we did not need to do much alpha and beta testing using Google's and Apple's platforms although they do both offer very good testing services.

Updating our app was extremely easy using Expo's publishing feature. Whenever we needed to update our app for bug fixes we just called exp publish from Expo's command line interface instead of re-submitting new binaries to Google and Apple. When we publish our app through Expo, Expo's packager minifies our code and generates two versions of our code (one for iOS, one for Android) and then uploads those to a CDN. Then the next time a user opens our app on their phones they will automatically download the new version. These are commonly referred to as "Over the Air" (OTA) updates, and the users don't need to update the app through the app stores.

Future Features & Ideas

- Having to manually put in free times is a friction point in our app. However, not everyone uses calendar apps, so importing availability from users' calendars was not an original core feature we planned to implement. We would still like to support this feature though, and this is our next step in development.
- In order to make the meal scheduling process even easier for students, we would like to integrate dining hall menus into our app and possibly get funding from Campus Dining.
- Students often get repeated meals with friends at the same time every week. We would like to add the feature to mark a scheduled meal as a recurring meal.

- Some other small features include favoriting friends, displaying statistics about the meals you've had with friends, and sending reminders to the user when they haven't had a meal with a friend in a while.

Advice for Next Year's Class

- Split development early on so everyone knows what they should be working on and use some method for tracking what needs to be done and who should do it. We used an excel spreadsheet, but next time we would use a ticketing system like Trello.
- Use coding style guidelines. All of us wrote code in different styles, so it was very difficult to read each other's code. We also should have separated component style code from the render methods. Decide on naming conventions early on and be consistent.
- We would definitely use Redux, a predictable state container, for our app if we did it again. Eventually our app got more and more complex so managing the state of each screen became tedious and our files got very long.
- For groups developing a cross-platform mobile app next semester, consider using Google Flutter instead of React Native. It supposedly performs even better and doesn't use JavaScript, which was difficult to learn in a short amount of time. We didn't use Flutter because the beta version of it was only released a few weeks ago.