# Group Scheduler Report

## I. Introduction

Many college students struggle to find a time in which all classmates can meet up to work on group projects. As engineering students, we may know this better than anyone. The idea of a group scheduling app came the same day the three of us formed a group. What if there was an app that automatically calculated a timeslot in which all individuals of a group could meet up? An idea was born.

While there were already a few apps on the market for creating an event for several people in a group, none of them automatically calculated an event timeslot for you. These apps were focused more on events that did not require everyone to attend; for example, allowing a Church Group to alert all members that an event was coming up and prompting those members to easily RSVP to the event.

We wanted our app to focus instead on the idea that everyone in the group had to attend the meeting. This app is not for extracurricular activities. This app is for mandatory class or work events that everyone in a group must attend. Specifically, this app imagines its target user as a student who aims to arrange a meeting within a relatively small window of time, such as a few days. With that target audience in mind, we decided to design an app that makes it easy to schedule a meeting with your group with just the click of a button.

## II. Design Process and Intended Goals

The first part of the design process entails capturing the narratives of the environment and the application. We quickly started to brainstorm ideas for our application. What views did we want? We knew we would need a place to see your own schedule, a friend's schedule, and a group's schedule. Our original plan consisted of three views for each of these categories: users would be able to see a list view of their events, a weekly view of their events, and then a continuous scroll view of all their events for each day in a two-week period.

We also needed a view for users to manually add events to their own schedules in our application, but also a way to calculate an event time automatically should they choose that option. The app dealt with personal data, so we knew we would need a way for a user to log in to our app. This lead us to decide that a login view and register view were also necessary.

Lastly, we needed a way to navigate through our application. Therefore, a quick side menu was formulated to navigate through the main parts of our app. We also decided we needed a list view of friends and a list view of the groups each user was in. With those brief ideas and sketches in mind we began thinking about how we were going to organize our code into classes and objects. To facilitate this task, we came up with a narrative of what our app aims to do.

> *When using this app, a user will first manually input their own schedule, adding events to represent the times when they are busy. These events can be recurring (that is, they will predictably happen on selected weekdays) or one-time events.*
>
> *A user can create a new group. Once a user is part of a group, they may add other users*

*to it. A group has a schedule, much like users do, except that a group schedule is comprised of the events of its members' schedules.*

*A user can ask permission to be another user's friend. Once the other user accepts the friend request, they may view each other's schedule, and see the other's contact information.*

*On any schedule, they have access to, a user may add new events. These events can be added manually, with the user specifying all date and time information, or "automatically," where the user will input a search window and a duration, which the system will generate such an event for, if one is possible.*

To capture the findings from the design process narrative, some text and screen flows were produced. First was a list of nouns and verbs.

**Nouns**: user, schedule, time, events, student, meeting, window, days, schedule, group, date, information, duration, app, system
**Verbs**: finding time, schedule, input [schedule], add event, add user [to group], create [group], add [event], generate [event]

From these nouns and verbs, the domain classes, responsibilities and collaborators were formed. A list of the main classes of the app is shown in Figure 1 below. The final app does divulge from these original classes a bit, but the core of the app is stemmed from these classes.

| **Domain Class** | **Responsibility** | **Collaborator** |
|---|---|---|
| Person | -schedule<br>-add event<br>-remove event | -event handler<br>-event viewer |
| Group | -add event<br>-remove event<br>-add user<br>-remove user<br>-create | -personal<br>-event handler<br>-event viewer |
| Friend | -add event<br>-remove event<br>-request permission<br>-give permission | -personal<br>-event handler<br>-event viewer |
| Event Handler | -add event<br>-generate event | -Group<br>-Personal |
| Event Viewer | -view events | -personal<br>-group |

**Figure 1: Original Domain Classes**

Next, we needed a way to store data for our users. Firebase uses a Non-SQL architecture and provides an "Authentication" data tree that is automatically structured for us [2]. The Authentication tree stores users by email and associates them with a UserID and password [2]. For our own data, we also made use of a data tree called "Mobile App Project" which used seven major child branches as seen in Figure 2 below.

| **Data Branch** | **Purpose** |
|---|---|
| Event | ● Used to generate and keep track of Event IDs.<br>● Stores event's attributes under Event ID.<br>● These attributes include: event name, begin time, end time, begin date, end date, and recurring days. |
| Group | ● Used to generate and keep track of Group IDs.<br>● Stores group's name under Group ID. |
| MembershipGroupToUser | ● Used for adding a group event to everyone's schedule in that group.<br>● Maps each group member's User ID to that Group's ID.<br>● Keeps track of which users are in each group. |
| MembershipUserToGroup | ● Used for displaying a list of groups the current user is in.<br>● Maps each Group ID the user is in to that User's ID.<br>● Keeps track of each group a user is in. |
| Schedules | ● Used to store a user's schedule to do necessary calculations.<br>● Maps every Event ID a user has in their schedule to a User ID. |
| UsersIDToName | ● Used to search users by User ID.<br>● Maps username to User ID. |
| UsersNameToID | ● Used to search users by Email.<br>● Maps User ID to username. |

**Figure 2: Firebase Data Structure**

## III. Translating Design to Implementation

After the design portion of our application was completed, our team decided to use Android Studio to develop our application meant for the Android operating system [1]. We only programmed using Java as it suited all our implementation needs [6]. First, we developed a calculator for generating an auto event timeslot. The calculator focuses on only the window provided by the user. Each day in the window is represented by an object that can hold event "blocks," which just contain the start and end time within that day. This way, the calculator will only look at blocks relevant to the current day it is examining instead of the full schedule of events. When searching for a time, the calculator will slide a block of the desired length through each day, comparing it against the existing event blocks to check for conflicts. When no conflicts are found, it generates an event object and adds it to the correct schedule(s).

From there we began implementing our views to retrieve user information. Our implementation started very rough, we had input text boxes for every piece of data we collected. However, when we had time to improve our design, we changed it so that users inputted times and dates using the Android Studio provided calendar and clock widgets [1]. The added improvements made it a lot more user friendly and easier to constrain to a layout.

Once we had ways to collect event information and group information, we began making views that showcased our data to the user's screen. We had an event list view, group list view, event detail page and group detail page. Our list views simply displayed a list of either groups or events to the user. Once a user clicked on a certain event or group, it would lead to the detail page for that specific item. If a user clicked on an event, it would display all that event's information to the user, as well as a button to delete the event from their schedule. The group detail view allows a user to add another user to the group as well as either leave the group or see the group's schedule which ultimately lead back to the event list view page.

Firebase requires a user to login to the database [2]. This required us to also create views for a login page as well as a registration page. As the authentication branch is structured separately from the main project branch, we had to learn how to save and store. as well as compare login input to, the data located in our authentication branch [2]. This allowed us to correctly implement our login and registration page.

Once we had all the views working, our next step was storing all our user's inputted data into Firebase, which is a third-party cloud storage database software [2]. Firebase proved very difficult for us to use. The two of us who mainly worked with firebase, both had extensive knowledge using SQL databases. So, we incorrectly assumed it would be quick work writing to the database. However translating over to Firebase, which does not use SQL, was challenging, as we started with an SQL-based normalized database schema and had to adapt to Firebase's branch system during implementation. While we did use Firebase's website to lookup how to save and retrieve data [2], it didn't always turn out structured exactly how we wanted it to in our database. This unfortunately, lead to a lot of finagling for certain lines of code until it appeared perfectly structured in our database. Though after the first two main branches, it became significantly easier for us to implement.

Once we could write to the database, pulling that information from it was the next step. After a quick lesson in how to make data snapshots and event listeners [2], reading information stored in our database was intuitive, and therefore easy to implement in the correct locations of our app.

While two of us were figuring out the database part of our app, our third member was working on a side menu to navigate between the pages of our app [3]. The menu was also straightforward after learning how to incorporate it into an existing project.

With the basics of our app working at this point, the only main use case of ours we still needed to finish was incorporating a sensor into our app.  We decided to use a shake listener [5] in our app to quickly bring up the auto event creation page. Again, this worked smoothly after learning how to implement it into our applicatio [5].

After all our use cases were up and running, we had a week to go back and improve our app. Some ways we improved our app can be easily seen as in the example of turning our text inputs to widget inputs. We also improved our app's performance by using constraint layouts, and improving how we did our calculations, which could be seen using Android Studio's profiler [4]. Lastly, we made the overall look of our app more finished by adding an About Us page, as well as a Home page that explained how to benefit the most from using our application.

While there were many small issues that came up when making our app as described above, we had a few major elements holding us back in our implementation. Our biggest issue was time. We knew the personal view and schedule view were necessary in our concept, so we decided to get rid of the friend view, as it was essentially just a group view minus the ability to see the friend's contact information.

Another thing we had to drop for time purposes and difficulty level of implementation, was the option to display your schedule in the three different ways we originally planned for. We knew we at least had to have a list view of a user's schedule, but we were unable to learn how implement a scroll calendar view and weekly calendar view in time for the official due date.

Lastly, we used JUnit testing when developing our app, especially in regards to checking to make sure our calculator was working correctly [4]. This helped speed up some of our debugging processes as it clearly laid out issues within our code to fix, this gave us a bit more time in developing other parts of our app.

## IV. Suggested Design Improvements

In hindsight, we would have made the in-app versions of user, event, etc. also be capable of being used as Firebase objects instead of having separate classes. This came about because we worked on the calculations and database separately before learning Firebase's system. This would have saved us a lot of redundancies in code, as well as gave us more time to work on other things instead of converting back and forth between almost the same object - just a local and remote version with the possibility of one or two different attributes.

We also believe that in the beginning of the design process - after identifying domain classes and responsibilities - each part should have been evaluated further. The individual pieces of the app, such as use cases, domain classes, and views could have been assigned an estimated difficulty and time commitment level. After determining those constraints, a priority level could have been given to each piece. This would have allowed for a better ordering of tasks, as well as a bit more thought process on the time it would take to finish each use case we had laid out.

We found that the design process as it is, purely focuses on the design itself, and not how the design would be implemented. While creating this app, there were several work hours that were spent on the more unnecessary elements of the app that were eventually cut, such as the weekly schedule display. Had we known we would not have had enough time to implement the different type of schedule displays in the first place, the time we spent writing partial code for it could have been spent making the elements of the app that were kept in more robust. Adding these other attributes to each domain class expectation would make it easier for developers to spread their effort more effectively, and finish one necessary piece of the app before moving onto other parts that were less important.

## V. References

[1] Android Open Source Project, Android, http://developers.android.com
[2] Firebase, https://firebase.google.com/
[3] Adding Nav Drawer to Existing Project, Team Treehouse, http://blog.teamtreehouse.com/add-navigation-drawer-android
[4] Android NF-requirements, testing, and profiling, http://web.cse.ohio-state.edu/~champion.17/5236/Lecture8_NFRTesting.pdf
[5] Android Tutorial: Implement A Shake Listener, http://jasonmcreynolds.com/?p=388
[6] Java API, https://docs.oracle.com/javase/7/docs/api/