

SP-5-Green Dynamic Grocery List

Final Report

Accepted to C-Day, Group UC-261

<https://hungr.dev/>

<https://github.com/quinwoods/hungr.dev>

<https://github.com/quinwoods/api.hungr.dev>

Senior Project – CS 4850 Section 03

Professor Sharon Perry

Fall Semester 2022

|  |  |  |  |
| --- | --- | --- | --- |
| Team leader | Quin'darius Lyles-Woods | Full Stack Dev iOS native | [quindarius@hey.com](mailto:quindarius@hey.com) / 704-470-7036 |
| Team members | Aidan Le-Beard | Full Stack Dev | [aidan30004@gmail.com](mailto:aidan30004@gmail.com) / 678-643-2863 |
| Adam Maksymczuk | Front End Dev | [amaksymc@students.kennesaw.edu](mailto:amaksymc@students.kennesaw.edu) / 678-860-6408 |
| Jeetu Sharma | Back End Dev | [contact@jeetusharma.com](mailto:contact@jeetusharma.com) / 678-648-9697 |
| Danny Sor | Full Stack Dev | [dannysor987@gmail.com](mailto:dannysor987@gmail.com) / 404-512-1354 |

## Correspondence

This project was managed by Professor Sharon Perry and is being submitted directly to her through D2L.

## Overview

The goal of our project was to deliver a dynamic grocery list app for both Android and iOS that was secure and had full back-end database support. We designed and developed this mobile app using Google’s Flutter framework for the front-end. Flutter is an open-source software development kit that makes creation of cross-platform apps much easier, between Android, iOS, desktop, and web. To create the app, coding inside of the Flutter framework was done with the Dart programming language. Dart is a programming language designed for web and mobile apps, designed by Google, with a C-style syntax.

The back-end of the app was developed using an SQLite database created in Python. SQLite is a database creation program that allows simple creation and querying of a database using SQL commands. The database is accessible through API calls through the internet. API calls can both write to the database, using the HTTP POST or PATCH command, and return information to the user using the HTTP GET command. The POST, PATCH, and GET functions are defined in Python, and execute a SQL command on the SQLite database giving the desired function. The defined GET commands return a JSON file that includes the requested information retrieved from the database. These API calls are written in Python, and this file supports API calls running on a server. The Python file is integrated into the web using Flask. Flask is a web framework that allows Python files to run directly on the internet.

The app additionally offers account creation, and to maintain security, accounts are managed using Google’s Firebase service. Firebase is a multi-purpose tool developed by Google that offers a variety of features, one of which is secure creation and authentication of user accounts. Firebase was selected to be used alongside our SQLite database to make sure that user accounts have maximum security. Firebase is additionally utilized in the app for its service in sending out push notifications to users who have joined a group together. Overall, the app is designed to be secure, cross-platform, shareable among users who choose to form a group, and to eventually have monetization through advertisements.

## 

## Table of Contents

[**Correspondence**](#_3m5qo99vkgwn) **2**

[**Overview**](#_cr2qr6x6d9y) **3**

[**Table of Contents**](#_8lih1rrdq8en) **4**

[**Background Information**](#_i1fpzkxkfhjc) **5**

[**Requirements**](#_fvt4bknwyvij) **5**

[**Analysis**](#_ulfelzwmb6s5) **7**

[**Results**](#_k70p3h5dux1s) **8**

[**Development**](#_141mioo9vccp) **12**

[**Project Planning and Management**](#_vigb5i5655jj) **14**

[**Version Control**](#_houx1tj71798) **17**

[**Test Plan and Test Report**](#_zax9y4hv0wn3) **17**

[**Summary / Conclusion**](#_mf81vx1x5o81) **18**

[**APPENDIX**](#_sevzm0fi77he) **19**

## 

## Background Information

Hungr was developed as a multi-platform dynamic grocery list app with the goal of being easy to use and convenient for practical use. The idea of the app was to take a traditional grocery list and add multiple features that make this app more convenient, unique, and simple to use. One of the more unique design features is the idea of being a dynamic grocery list. This means that users in a group will have their grocery lists automatically synced so that everyone is up to date on changes in the list. With this, shopping with family and friends becomes easier to use and manage. Users can manage what items are on the list, add their own, and track changes to see what has been added or removed in real time.

With many other grocery list apps already existing, it was important to define a few features that would make the project unique and more convenient than other grocery list apps. For example, one of the main selling points was the idea of being a dynamic grocery list app across multiple platforms. With this feature, friends and family can join a group together where they can update their list which will then be updated for all users in the group. Alternatively, Hungr also allows users to individually produce a grocery list and access the app offline with the usage of local files.

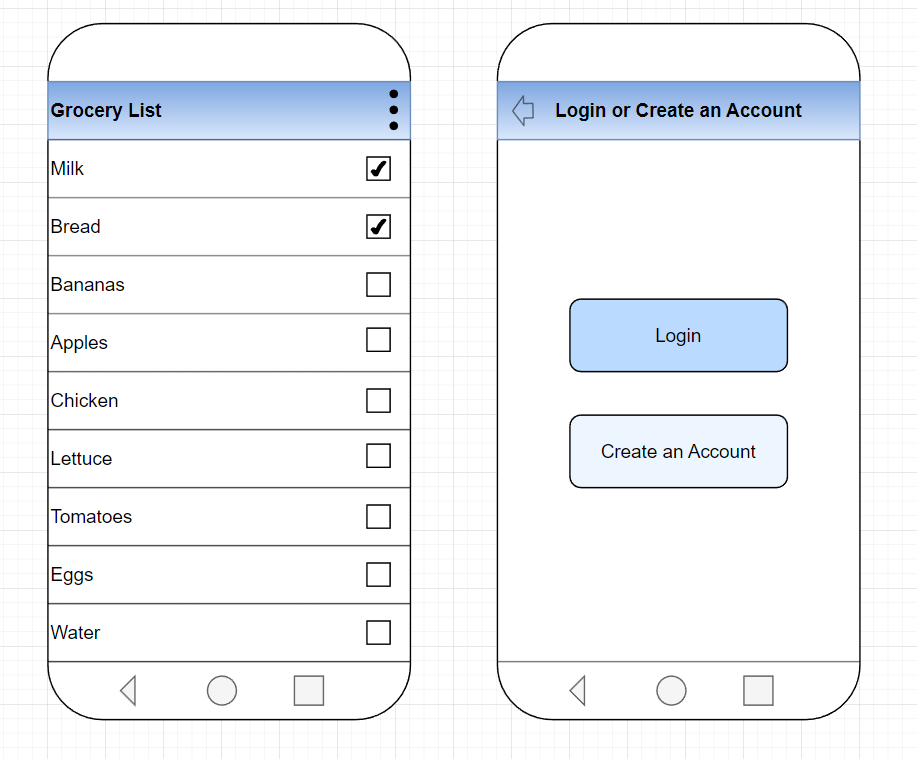
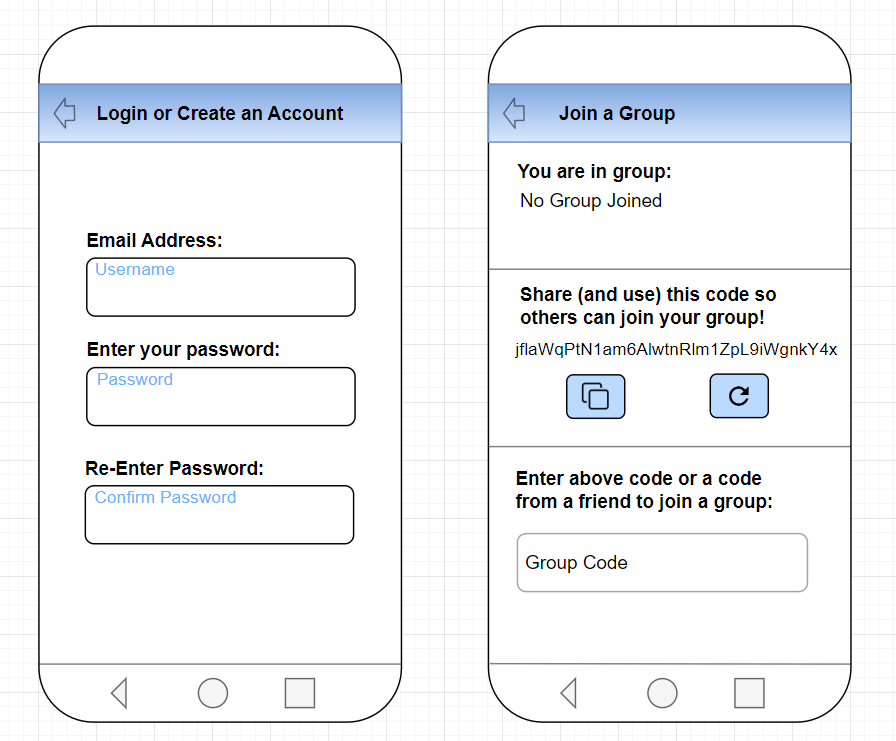
## Requirements

This project’s scope was split into two phases, with phase 1 being absolute requirements, and phase 2 being stretch goals. Phase 1 consisted of creation of mockups, creation of initial individually created prototypes, selection of one prototype to continue with for the final project, creation of the website, creation of the backend SQLite database to hold group information, food items, and various other data, and creation of the API and server to be able to query and write to the database from within the app. In phase 1, the app had the requirements to show a pre-populated grocery list, and to give the user the ability to add and delete food items from the list. The user had to be able to securely create an account with a unique email, with their password adequately protected, and then have the ability to join a group. The user’s list of items that they have added and deleted from was required to be stored locally, and read from a file upon opening the app. The communal shared list is read from the database when viewed, with list items being created from the items returned from the database, assuming a user has joined a group. The app finally had to generate a shopping list from the communal list, where the user who does the shopping can use a check box to indicate items are in their cart, and submit the shopping list to the app.

In phase 2, the first goal was to complete the requirement that the app must send push notifications to the group when the list is updated by any other group member. In this phase, the second goal was various sorts of the grocery list. The list is able to be sorted alphabetically by its nature, and support was added using local files to be able to sort the list by the user’s individual purchase frequency that is stored on a JSON file on the user’s device. A final sort that was implemented was a group frequency sort, using the purchase frequencies of the entire group that is stored in the SQLite database. The third major focus of phase 2 is the interactivity between iOS and Android. The final major focus of phase 2 was monetization, with the main focus being on adding advertisements. Another opportunity for monetization is using purchase frequencies, as this information could be used for more targeted advertisements, for shopping statistics, or to prompt a user for items that they often purchase that they might have forgotten, allowing potential partnering with grocery stores by proxy of these extra items adding to the store’s profits. A further security goal in this phase was two-factor authentication. The final stretch goal of phase 2 was utilizing various APIs, to accomplish tasks such as measuring the distance to the closest grocery store, finding the pricing of the food items in the shopping cart at that store, and finding the aisle that those food items are in at the store.

The following figures are the mockups that were used to facilitate the design of the user interface for the app. Mockups of the user interface were developed using <https://app.diagrams.net/>. The app.diagrams.net website is an online diagram software where users are able to create flowcharts, mesh together diagrams, and draw any types of shapes that might fit their needs for their mockup. The mockup tool was used to create the visuals of user interface design for the app and it influenced the overall look of the app as it is. The mockup diagrams helped to visualize and contextualize a general idea of the flow of the app. The mockups were designed using mostly drawings with the basic shapes that were provided on app.diagrams.net. The drawings on the mockup are a mixture of shapes to facilitate the buttons such as checkboxes or textbox fields to represent the user input. The drawing tool capabilities allows for multiple types of diagrams to be displayed, so a simple phone design was chosen. The app was also designed to be simple to use and easy to learn.

The mockups were designed with simplicity and minimalism in mind. The user interface design was heavily influenced by easy-to-navigate and simple to use menus. The initial idea for the mockup was to highlight important features but also keep the app relatively straightforward. As seen in the mockup screens, each screen was designed to keep a relatively clean user interface design. It was important for the mockups to show a balance between useful information as well as not overloading with too much information on one screen.

**Figure 1: App Mockups**

## Analysis

We began the project by examining coding the app natively, in both Swift for iOS, and Kotlin for Android. However, after experimenting with both Kotlin and Swift for the initial app development, we decided to experiment with Flutter, as it offered cross-platform ability and further features in app creation, such as widget creation. In examining Flutter we also realized it gave greater consistency between the two app versions, rather than attempting to code two separate apps to have the same functionality and look. We found out that we preferred using Flutter over the previous environment setups, so we experimented with the functionalities that Flutter provided and ultimately we ended up choosing to develop the final app with Flutter. Flutter is a software development kit created by Google that we are utilizing for its cross-platform ability between iOS and Android.

GitHub was utilized to maintain version control, as well as to update the functionality of the app between everyone in the group. The entire app and back-end was initially contained in one GitHub repository, but after beginning to develop the app, the API and database were separated into their own repository, to allow splitting up of tasks between the API/database and Flutter. Our GitHub pages are managed by our team leader, and can be found at <https://github.com/quinwoods/hungr.dev> and <https://github.com/quinwoods/api.hungr.dev>. GitHub is a hosting service for code in all languages that highlights changes made to the code, allowing collaboration on coding projects between groups.

We initially started using SQLite, a database creation program, for all back-end operations. However, after analyzing the difficulties associated with secure user authentication, and with security being of great importance in creation of this app, we decided to switch this portion of the database over to Google’s Firebase database and hosting service. Firebase is directly designed to create accounts and provide authentication from apps. Firebase offers methods for account creation, for logging in, to send emails to users for password resets, for persistent verification of the device through authorization tokens that can be revoked and checked for validity, and finally for logging out.

The app was written in Flutter, using Android Studio. Android Studio offers error detection and automatic dependency checking and downloading through the terminal. It additionally updates any associated files when dependencies are changed. Additional features it has are emulated devices to test the app on, as well as options such as hot reloading on live devices that are connected, to be able to instantly see and test changes to the code. Android Studio offered all features necessary to develop a complete Flutter app, allowing complete creation and testing of the frontend functions.

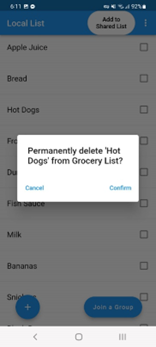
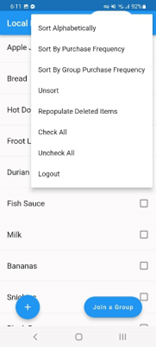
The API and database were created in Python, in VS Code. VS Code is an IDE compatible with most, if not all, programming languages. Users can create extensions that offer a wide-range of capabilities. VS Code was utilized as it offered SSH capabilities, where every member of the group was able to access the Python Flask API code hosted on the team leader’s server, and able to make changes that could instantly be seen in regard to the database and API calls. In addition to the SSH extension, VS Code was utilized as it offers an extension to view SQLite databases, allowing changes such as adding tables or columns, or incrementing frequency variables, to be instantly viewed, without having to use any further programs or web apps. VS Code offered all features necessary to access files hosted on a server, to refresh the server, to make changes to the API and database, and to view the database, allowing complete creation and testing of the backend functions.

## Results

* The following is a link to a short demonstration of the app in action: <https://www.youtube.com/watch?v=YLBflr1681M>
* The following is a link to an in-depth discussion of the inner workings of the app, SQLite and Firebase databases, and API: <https://www.youtube.com/watch?v=vZcPpey8oNk>

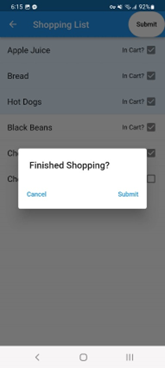
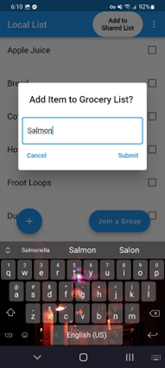
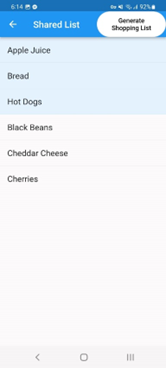
Upon opening The App, a user is not logged in and not in a group, and as a result any API calls or internet connectivity is not needed, but The App’s rest state needs to be maintained. That is accomplished through locally saved files, which is accessed using the appDocumentsPath via several JSONs. GroceryItems.json is shows the user a list of populated grocery items. GroceryItemsCopy.json is associated with another array of frequencies, which keeps the individual purchase frequency in order as those indices do not change. GroceryItemsCopy is also used, for example, if an item is swiped right to delete it, those same items can be repopulated using a dropdown tab. The App does this by cross-referencing groceryItemsCopy with groceryItems and checks for anything that is not in view and adds it back to groceryItems and resets The App state so that those items are repopulated. Another feature is permanent deletion, which deletes an item from all arrays, this occurs locally.

**Figure 2: App Screenshots**



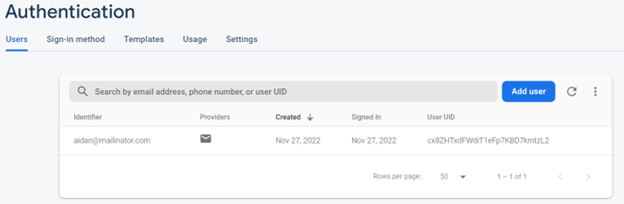
When the user presses the Add to Share List button, The App adds it to an array called “checked”, which corresponds to the checked.json file. Upon app restart, the program reads from groceryItems.json to load our list and then it reads from check.json to see which items we just added to the shared list. The obtained items are highlighted with a blue background. To remove them we can swipe them to the right on the shared list. At the end screen, we can purchase our items to finish shopping. The shared list is updated in real time, live to all users, while people in the group are adding items to the list. The local saved list does not keep updating, as it is primary purpose is for shopping and live updates are not required, so this one does not change. We check items as we put them in our cart and then we can submit it to our app and that increments the frequency index that's associated with that item being in groceryItemsCopy. If we sort by purchase frequency, for example, peanut butter might be at the bottom, once it is bought again and it automatically reapplies the frequency sort so we see peanut butter purchases are greater than corn. Buy it again and it will jump in frequency because we've bought it more than all of those items below.

**Figure 3: App Screenshots**



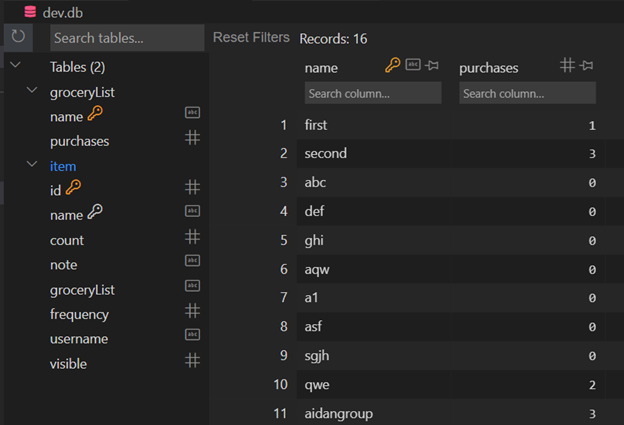
The app is completely usable as a single person without logging in or joining a group without any internet connection. It does that through locally saved files. The local save files are also used when we are in a group because we still want to maintain this local list. Our accounts are maintained through authentication security which is done through Google's Firebase. According to Google’s own documentation, “Firebase Authentication provides backend services, easy-to-use SDKs, and ready-made UI libraries to authenticate users to your app.”

**Figure 4: Firebase Database for Users**



Account creation is a standard affair. Any valid email can be used and password created. The Firebase database securely stores all account(s) on Google’s cloud. Firebase doesn’t store user information with insecure plain text keeping of passwords. New accounts are stored using a random user ID string. Groups are formed using an alphanumeric string to send to all members of the group. Firebase chooses a random number from 20 to 100 and then fills each uh index of the string with a random uppercase or lowercase letter or number from zero to nine. The code is refreshable to prevent any malicious users from infiltrated groups unwantedly.

**Figure 5: SQLite Database groceryList Table**



We also have a SQLite database that is running to keep track of group names. Any new list gets added to this table, one entry in the table “purchases” is how many purchases that a group has submitted. That table listing is used as a clock, in that the action that is performed gets a timestamp with it and the computer knows that whichever time is the furthest in the future is going to be the most accurate. Upon grocery items being added to the group list, a timer that refreshes every 10 seconds is enacted. It calls the syncCheckedToServer method that is setting the state looking for items that were added, for example, if the user decides they don't want those items anymore and the other user(s) update(s). That's how the users are keeping in sync through constantly reading through the database.

Grocery lists in the table count and note items that are not currently in use, frequency is how many times we have bought an item, username exists so there's that the user ID Firebase automatically signs as automatically generated hashes, that's how we're keeping our users separate and seeing who added an item and then the final thing is our visible quantity which is in sharedListView and the subsequent shopping list that's created.

## Development

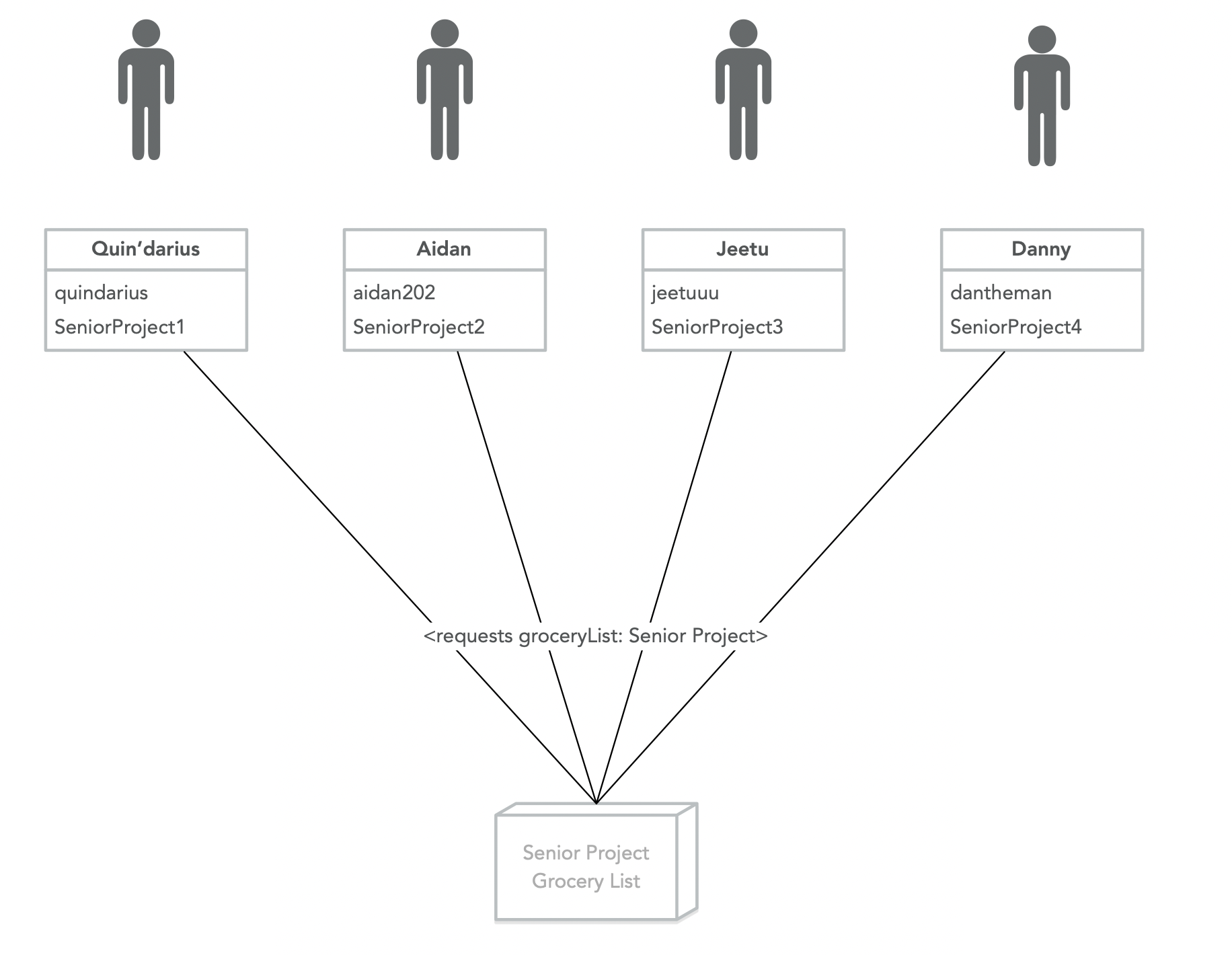
As discussed in our Analysis, we initially started development using Swift for iOS and Kotlin for Android. However, after we experimented with Flutter we found that we preferred the functionalities and overall environment of Flutter over the previous environment setups. We continued to use Android Studio as our IDE for app creation, but continued using it with Flutter and Dart, rather than the Kotlin and Swift we had started with. It was determined that most of what could be done using Dart could be done using the Flutter framework and environment setup, so the decision was made to continue with the rest of the project based on the initial prototype done in Flutter.

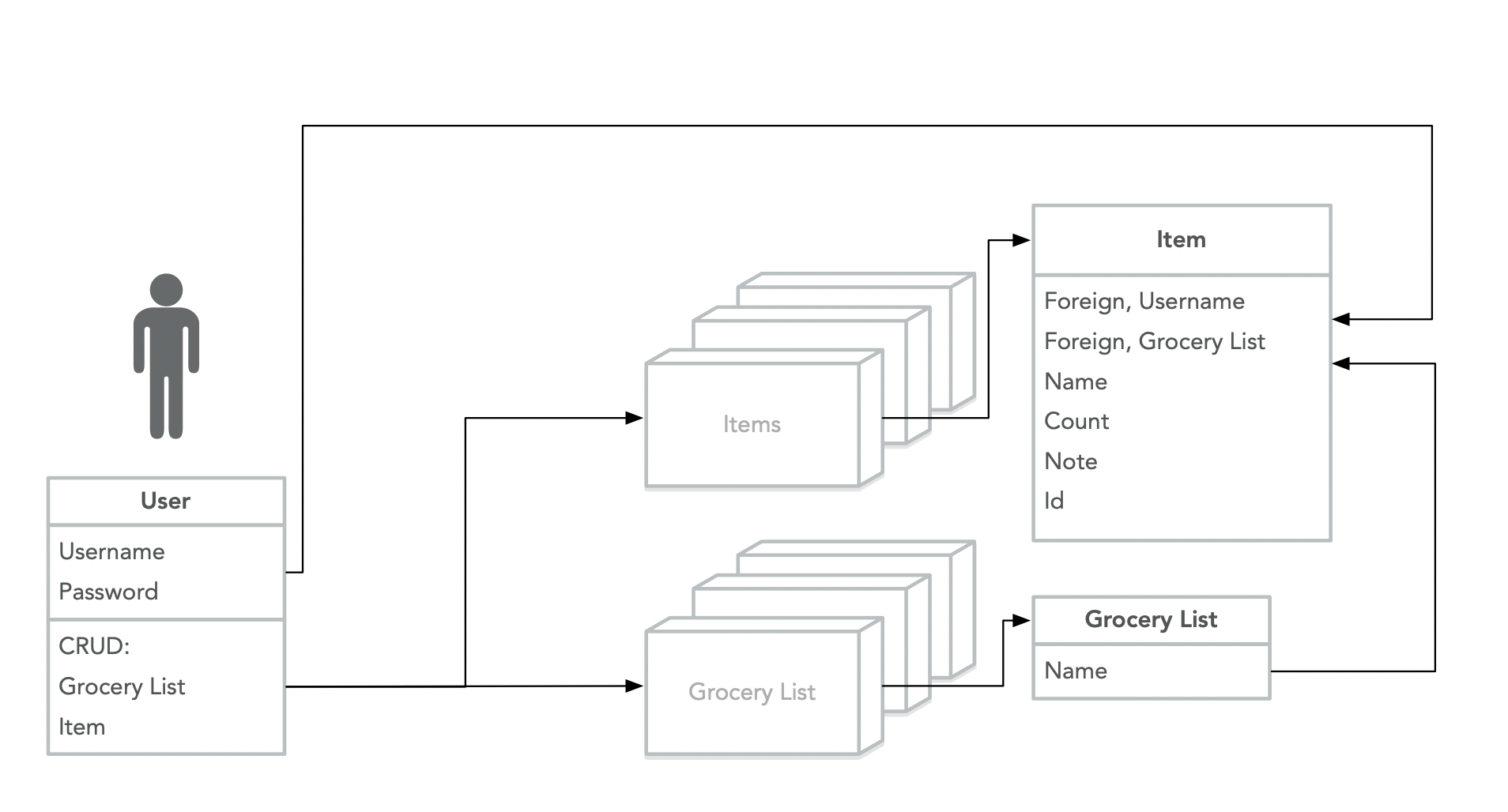
Roadblocks include the initial development of the first prototype that was developed in Android Studio. The environment setup using Dart was worked on for a considerable time before switching over to Flutter. Each group member developed their own prototype for the initial phase 1 requirements, with a basic functioning app using either Dart, Flutter, or Swift. After a review of the pros and cons of using Flutter, it was decided that the prototype using Flutter would be the main project that would be used going forward. The prototype that was developed using Flutter worked well and provided a strong work around for the previous prototypes that were developed using the other environment setups.

The general flow of the program was designed with simplicity in mind. The flowchart diagram seen in the figure below is what was followed and what worked well for the app. Users have the choice of either using the app freely without a group or joining a group to sync their list dynamically with others. The first option to use the grocery list alone allows the user to freely access and manage their grocery list offline. The second option is to create or join a group with others to dynamically update the grocery list. Joining a group allows the users in the group to sync their list together so that it updates in real time and changes can be made by any member of the group. The code for a group would need to be retrieved in order to have a unique and secure code for the group.

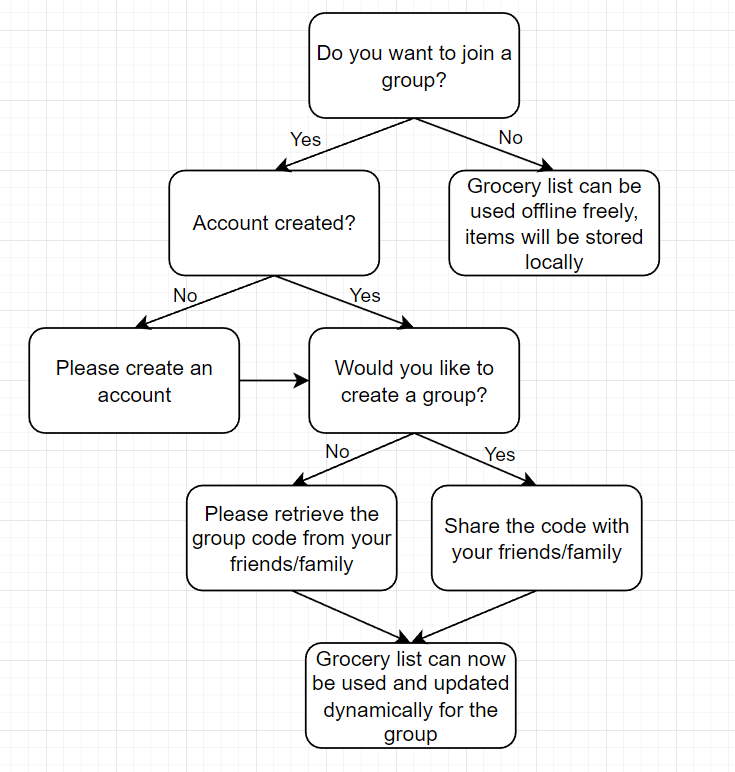
A final roadblock was adding advertisements to the app. Effort was made to integrate both Google ads and Huawei ads. Google would not allow creation of an Ad Mob account from the project’s Firebase account, making it impossible to integrate Google’s ads. Huawei ads were attempted once Google ads did not work out. Huawei required information such as an ID and bank statement to integrate ads, presumably to deposit revenue, however further research revealed that Huawei ads might not function at all in the US. After this discovery, the Huawei advertisement attempt was discarded, before providing personal information.

The following figure is a high-level architecture of how each user interacts with each other on the back-end status. Each user has created an account which would be successfully verified and authenticated through Google Firebase. Then, since each user possesses a unique username, it is used to connect and sync up the group’s grocery lists. When the user enters a unique group code, the group list would be updated accordingly upon joining the group. Requests to access the grocery list are sent and the group is then synchronized to dynamically update each of the users’ lists from the SQL database servers.  
  
**Figure 6: Detailed Architecture Drawings of Backend**



****

**Figure 7: Flow Chart of App Flow**



## Project Planning and Management

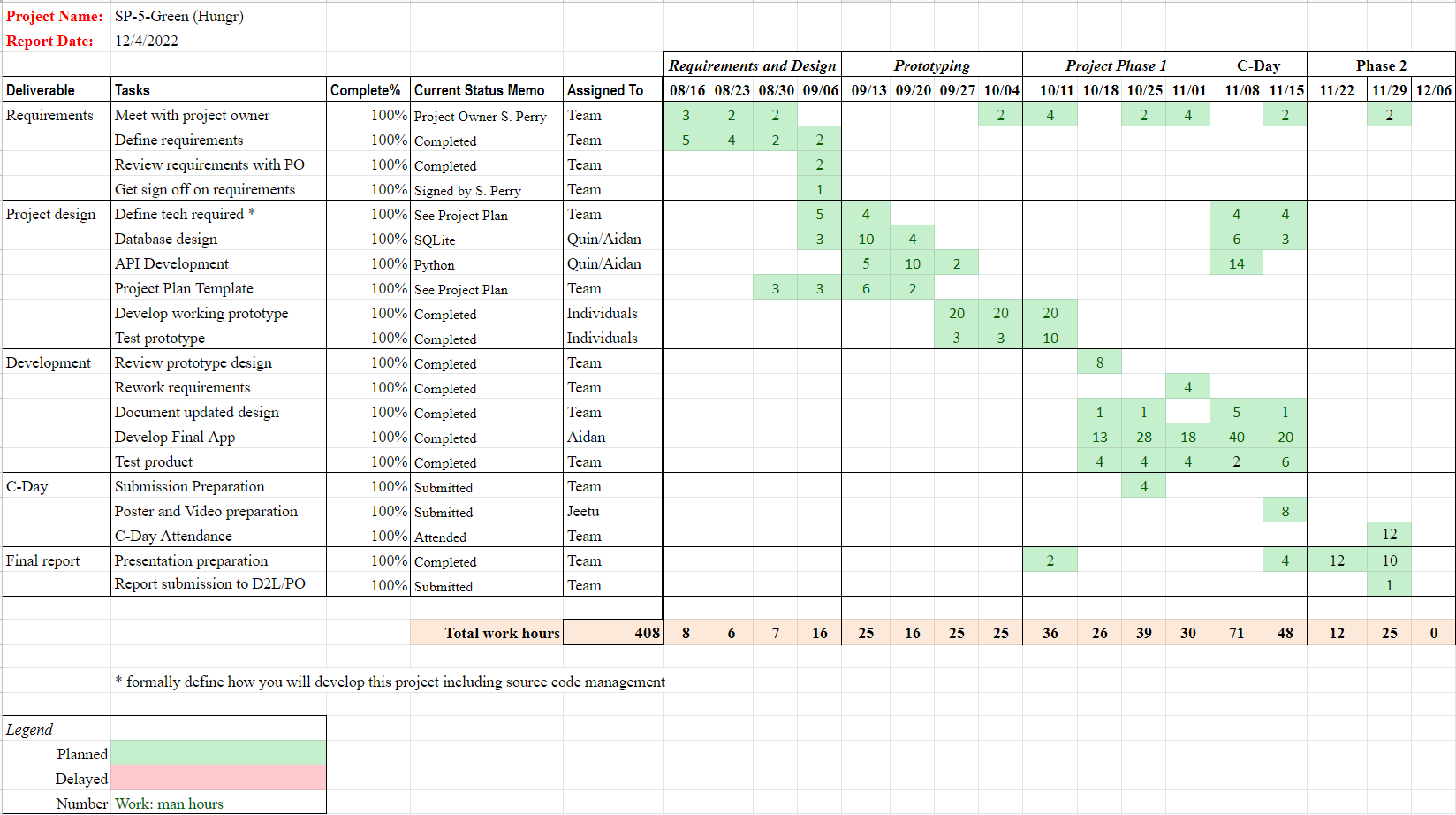
The app was developed in a semester-wide collaboration of 5 team members. Collaboration methods included both virtual and in-class meetings, ranging from weekly to biweekly incorporating the use of programs such as Discord, Gmail, and GitHub. Documentation and reports were mostly handled using Google Docs and Microsoft Word. Microsoft Excel was also utilized to maintain the status of the Gantt chart schedule. Virtual meetings would be held through Discord through the usage of both text and voice chats, with the addition of permissions such as screen sharing enabled through Discord. Project file shares and transfers were mostly facilitated with the use of GitHub as well as Discord messaging. Video editing and management was completed and uploaded using YouTube. For example, the C-Day video introduction as well as the full presentation video, or the in-depth analysis of the workings of the app functionalities and features.

Vital deadlines and due dates were communicated frequently and managed with the usage of the Gantt chart to help visualize and update the current progress at all times. Other deliverables were mostly communicated through text chat via Discord. Members also provided email and phone numbers in case communication was urgently necessary. The Gantt chart facilitated the process of handling tasks by updating the process of each group members’ responsibilities as well as their progress at all times. Meetings that needed to be scheduled were scheduled using Discord with a built-in feature that allows the user to schedule an event and updates each member on the details of the meeting.

Whenever file transfers were necessary, GitHub was utilized to facilitate version control as well as organize the app’s software bundles, documents, and a few assignments. GitHub managed all of the source code version control as well as helped with version history and tracked all of the changes that were made. GitHub’s features informs exactly which user contributed to which specific part of a file and shows all of the code changes that were made to files. Team members also made sure to communicate through text messages to update on what was done and the status of their respective task process.

The Gantt chart was managed by all members and revisited multiple times to review the status and update multiple items that were managed and shown in the Gantt chart below. Items were added throughout the project period and new features needed to be updated and reflected on the schedule. The Gantt chart was updated multiple times dynamically to include newer development updates as well as features that were worked on or completed. The project plan requirements were referenced and a comprehensive but tentative schedule was created initially to form the Gantt chart. Team meetings were held accordingly to address the project plan elements that were being worked on or needed to be examined.

**Figure 8: Gantt Chart of Project Development**



## Version Control

Version control is an important aspect of software engineering. Irreparable mistakes can be made when writing code, and depending on how much work has been done, it might not be obvious where the exact issue is coming from. Version control solves this issue, by allowing the software development team to roll back the software to a prior version that is known to be working, if an issue comes up and the team is unable to track down what the exact change is that is causing the issue, or if the issue is not fixable. Version control can also be useful if a feature is decided to no longer be necessary, or if a feature is to be completely remade. The software can then be changed to an earlier version, to continue using this as the main version, or to continue redeveloping from the version roll back.

Version control of this project was handled through Github. Github includes the option to “commit” a new version of the code, which updates the project as hosted to have the newest version of the code, highlighting differences and maintaining the old version so that any changes can be rolled back if necessary. Highlighting changes is a very useful aspect of Github’s version control that it offers. In this project, one app breaking issue came up. This was uploaded to Github, and through the highlighted changes that Github provided, it was much simpler to track down where the bug had occurred, as only the highlighted sections were possibilities, and to roll back the portions of the code to an earlier version where necessary.

Along with being able to highlight important changes across source files, github allows the ability to make branches. This allows the project to be meaningfully divided and conquered among many different feature branches. With the ability to work on different features simultaneously and gracefully merge the changes back into the main source tree, we were able to rapidly and independently work across many different features without making many breaking changes throughout the code tree.

With the ability to make feature branches we also have the ability to protect our main branches on Github. Branch protecting is a crucial feature when you have a team size of two or more. With branch protection you are able to withhold the ability to merge the code into a certain branch if it hasn't been properly reviewed by the appropriate members. With this enabled it enforces an environment where we thoroughly review our teammates’ code before we merge it within the main branch. Without such checks the branches would have surely withered away.

## Test Plan and Test Report

For our Testing and Quality Assurance Testing we relied on Unit Test and Self User Interfacing Test to verify the functionality of the application. We did not achieve 100% Test Coverage due to the velocity that the application was programmed in. But with more time that would definitely be a longer term goal for the team.

For the Test Report, we had all of the tests passing on the API and the application side within both the Flutter and iOS application. The database and API were confirmed to write back the correct results from GETs. The API was also tested with PATCHes and POSTs to confirm proper writing to the database and all the call results were correct.

For the test plan of our Grocery App, we planned to integrate a test driven development approach that checks the proper function of a feature. Test driven development helps focus the correct implementation of requirements.

The test driven development cycle and strategy:

1. Make a unit test for a feature asking about use cases and informal specifications.
2. Run all tests except the new unit test. This should show the new unit test 100% failing. This ensures the new feature is actually needed and no flaws are present in the new unit test.
3. Write new simple code that will pass the new unit test. This is the coding of the new feature that will 100% pass the unit test (ex: using an assert). No further code should be added for that feature.
4. Run all tests and make sure everything 100% passes all the tests. If any tests fail then the new code/feature must be revised to ensure a passing of all tests. This ensures the code meets requirements and doesn’t break any previous features.
5. Refactor as needed, and test after each refactor. Refactoring is done to give readability and maintainability. Examples of refactoring: moving code, removing duplicate code, adding documentation in the code, splitting methods/units into smaller ones, rearranging class inheritance hierarchies.
6. Repeat. (For each feature)

The unit tests should be small and incremental thus ensuring easy debugging. For the app we plan to use flutter’s built in test package to help run tests.

## Summary / Conclusion

Building a grocery list mobile application involves a number of steps. Here is a brief summary of the process:

1. Identify a need or problem that the app will solve. For example, the app could help users manage and organize their grocery lists, or it could provide recipe ideas and meal plans based on the ingredients that users have on hand.
2. Develop a concept for the app and create a design plan. This can involve defining the key features and functions of the app, such as the ability to create and edit grocery lists, search for recipes, and share lists with others. The design plan can include wireframes and mockups to visualize the user interface and user experience of the app.
3. Write the code for the app, using a suitable programming language and software development kit (SDK). This can involve creating the app's user interface and defining its functionality, as well as integrating any necessary third-party services or APIs.
4. Test the app to ensure that it functions properly and is free of bugs and other issues. This can involve conducting both manual and automated testing to ensure that the app performs well and meets the needs of its intended audience.
5. Publish the app on app stores and market it to potential users. This can involve creating a marketing plan to promote the app and attract users, as well as gathering feedback and reviews to improve the app over time.

Overall, building a grocery list mobile app can be a complex process that requires a combination of technical skills and creative thinking. By following these steps, it is possible to create an app that helps users manage and organize their grocery lists, and provides valuable features and functionality.

## APPENDIX

**Risk Assessment**

This app has accounts, and so one risk that was assessed was the possibility of passwords leaking. For this reason, various encryption schemes were assessed, to see which had the least risk, and the greatest payoff for protecting users’ personal information. It was determined that using Google’s Firebase framework for authentication would provide the best security, rather than using an in-house option. Password protection was a must for phase 1, while a phase 2 goal is implementing two-factor authentication.

Risk assessment for Phase 2: How might monetization be implemented through the app? Ads might be intrusive? How to plan frequency or sizes of advertisement? Will advertisements target the consumer’s interests? Also another risk assessment issue includes account security and password management via two-factor authentication as mentioned in Risk Assessment above.

**C-Day Videos**

<https://www.youtube.com/watch?v=PvmRCP0rMzI>

<https://www.youtube.com/watch?v=Ga0r36XDLyk>

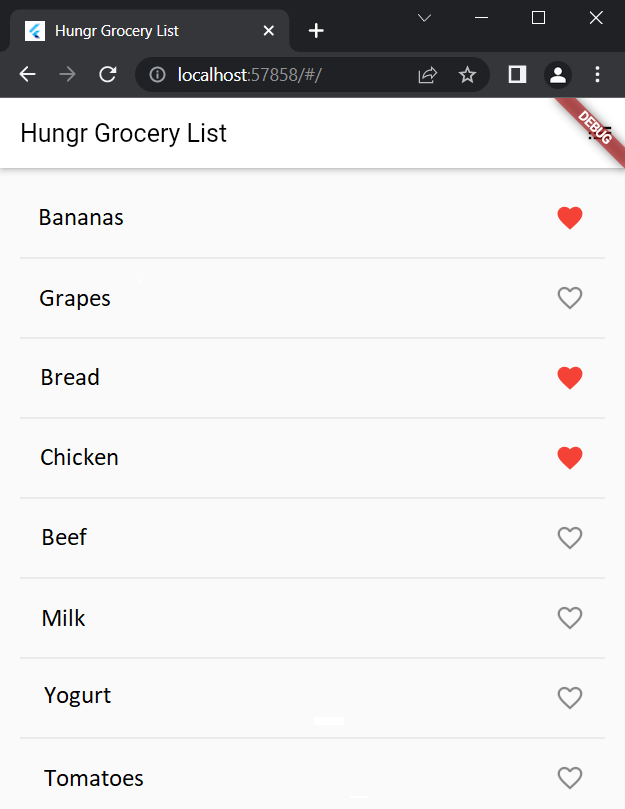
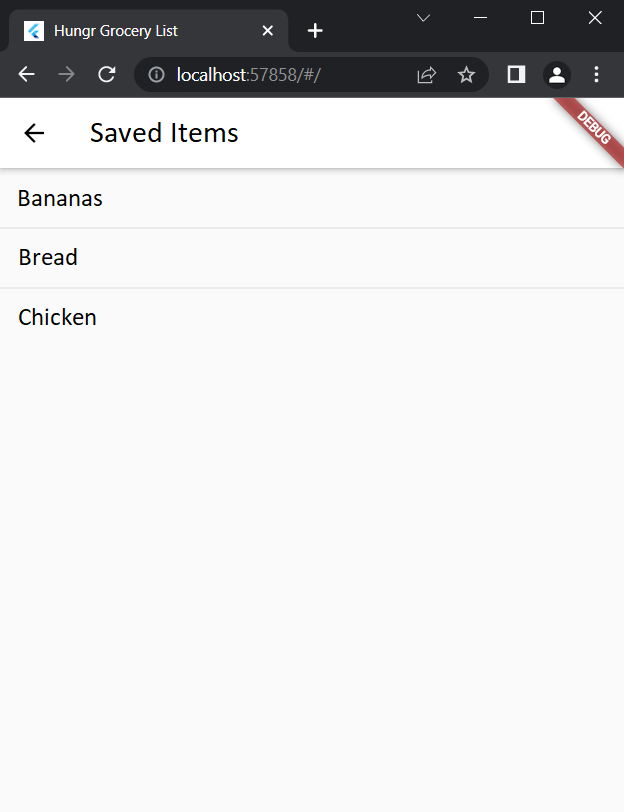
**Demo Videos**

Android Demo: <https://www.youtube.com/watch?v=YLBflr1681M>

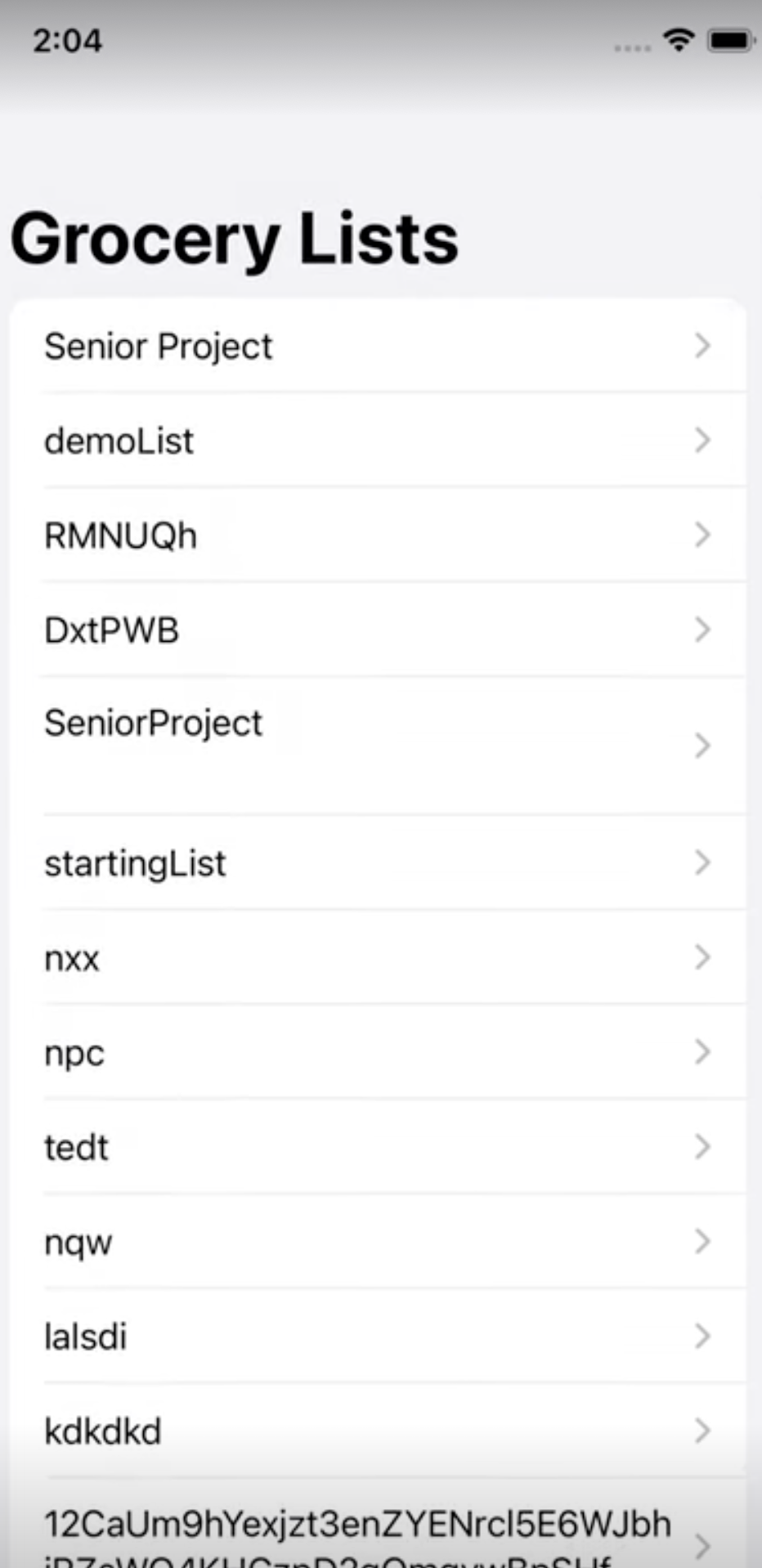
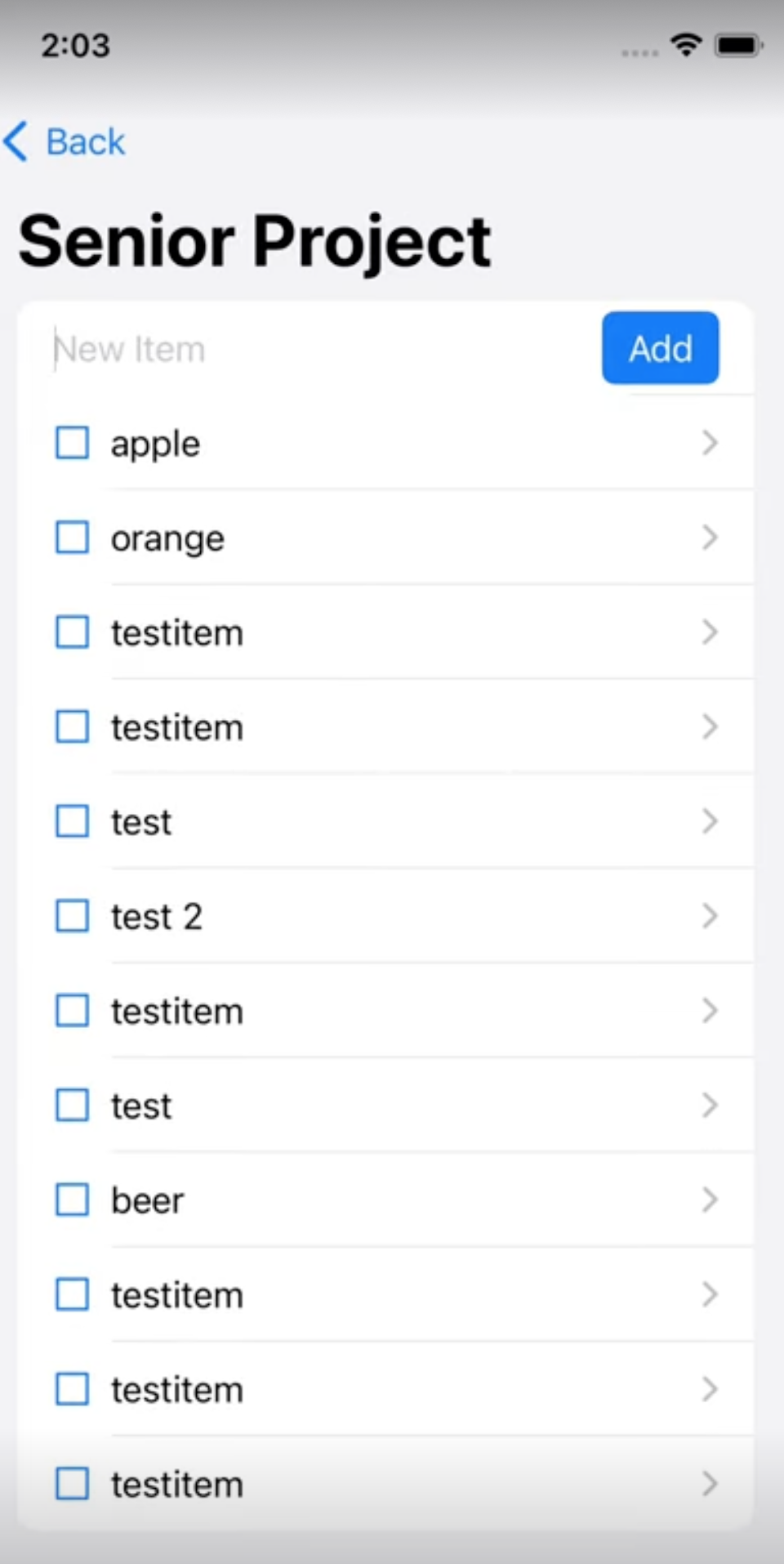
iOS Demo: <https://youtube.com/shorts/KvcdTJ1onco>

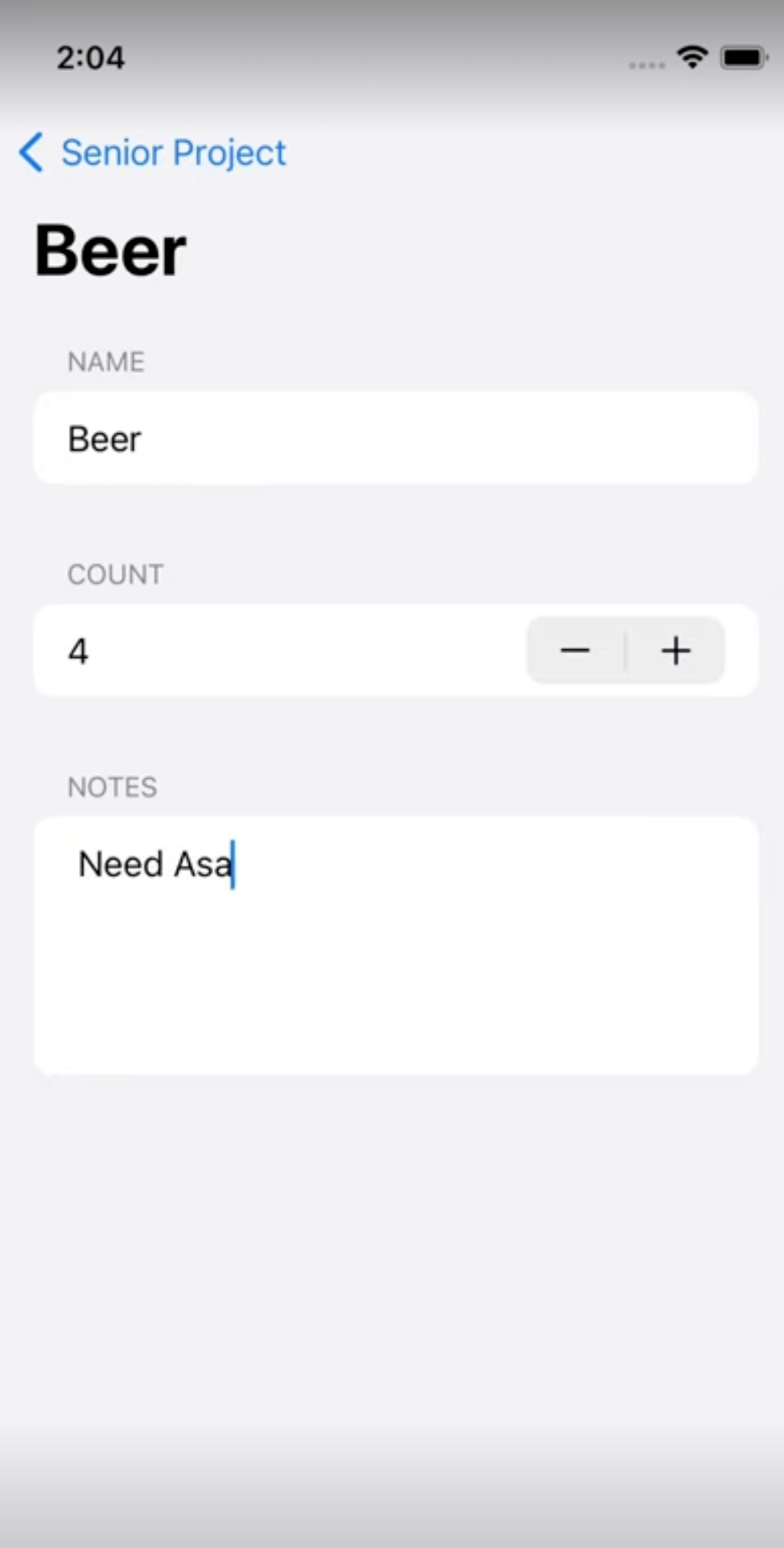
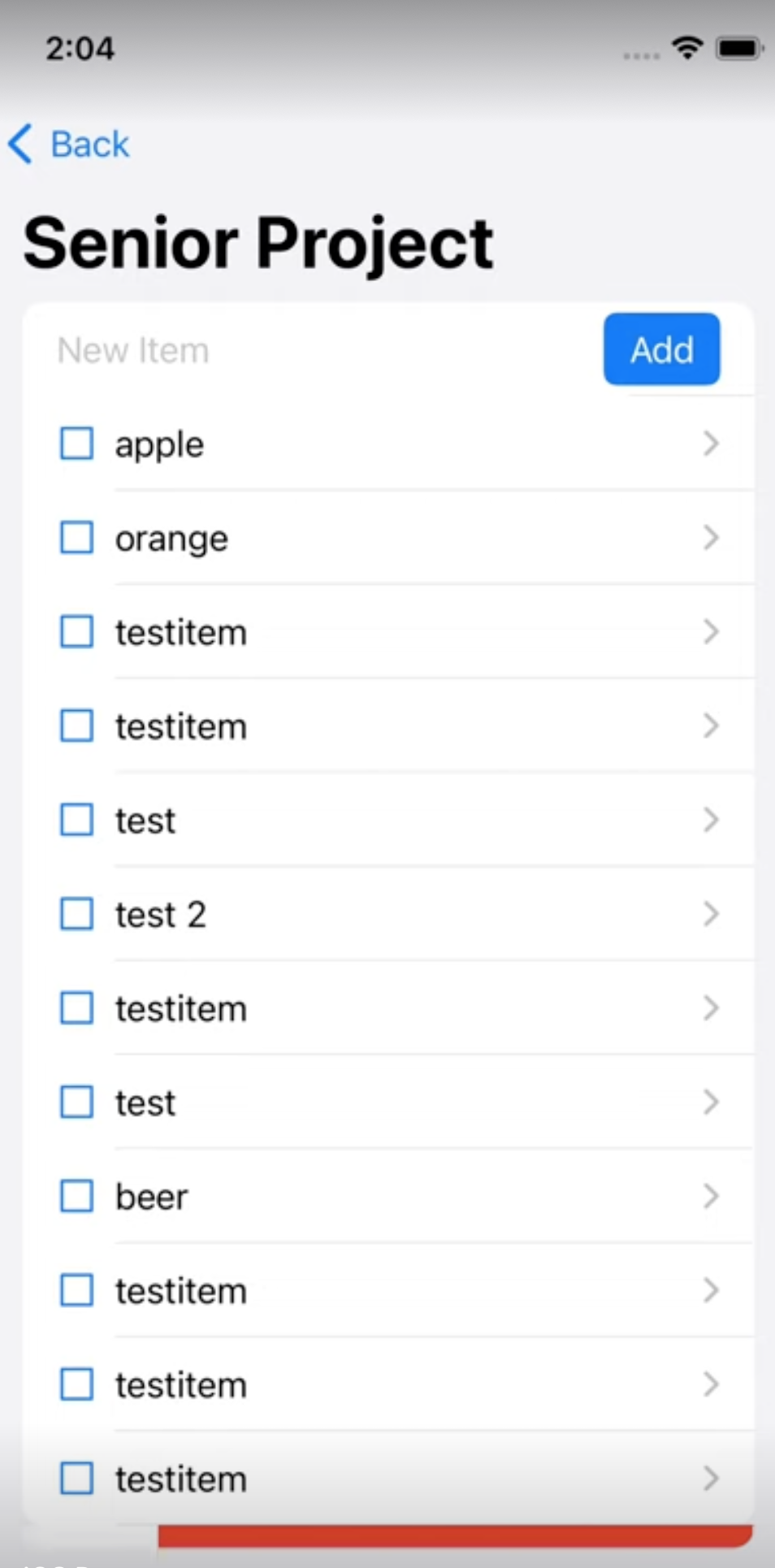
**Prototypes**

Secondary Flutter Prototype:

** **

iOS Native Prototype:

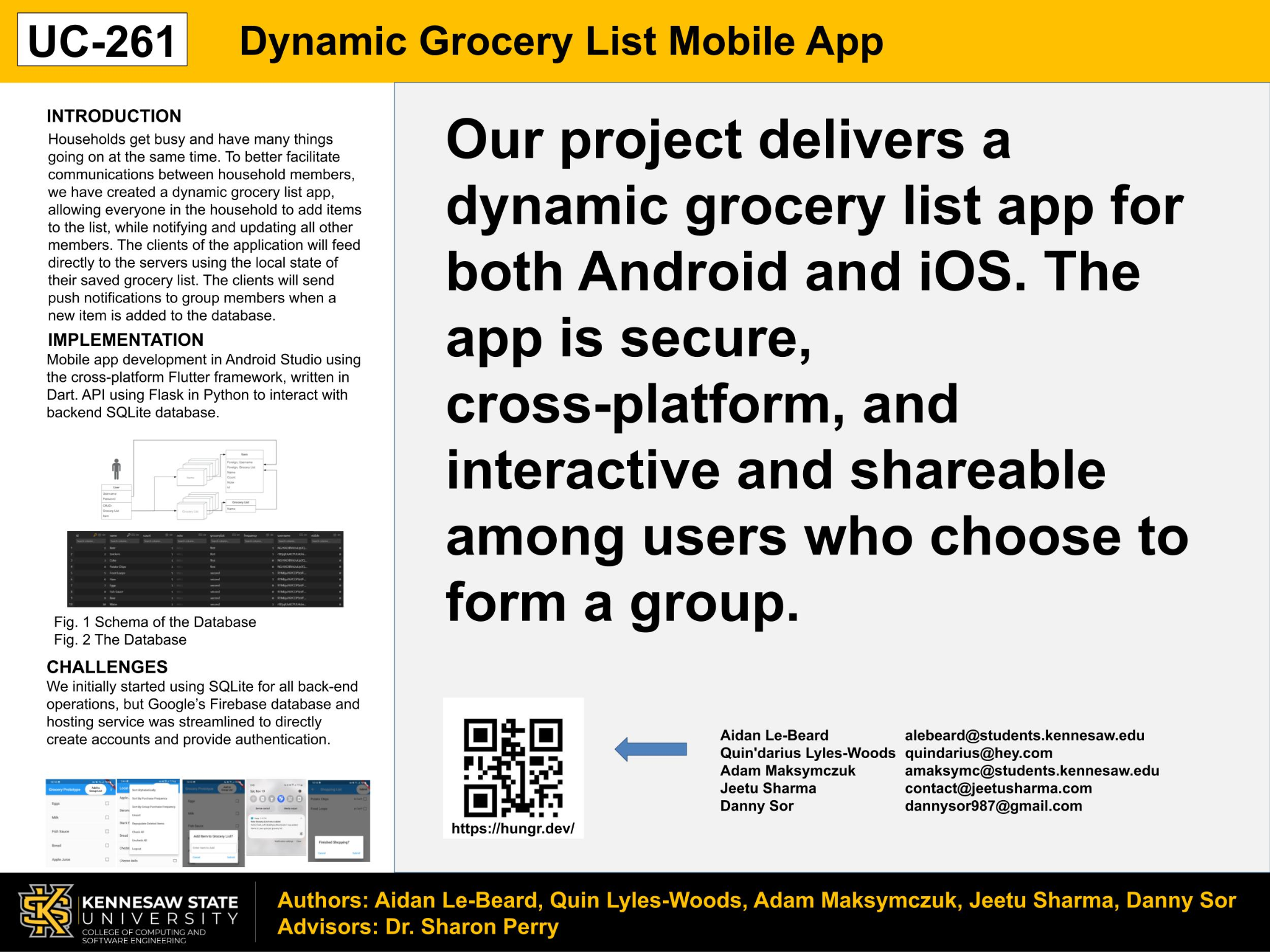
****

****

**Hungr Logos:**

****

**C-Day Poster:**

****