

Crowd-Source Grocery Store Layouts for Dynamically Sorted Shopping Lists

By:

Haseeb Khan (SE) (101009713)

Jobin Mathew (SE) (101006153)

Abraham Srna (SE) (100997482)

Supervisor: Dr. James Green

A report submitted in partial fulfillment of the requirements

of SYSC-4907 Engineering Project

Department of Systems and Computer Engineering

Faculty of Engineering

Carleton University

Date: 1 April 2020

Abstract

Grocery shopping is generally a very straight forward task. The shopper creates their list throughout the week, goes to the store, locates their items, and purchases them. The most time-consuming aspect of this process is locating items. Grocery lists are often out-of-order, requiring shoppers to double back to retrieve forgotten items. This, alongside the fact that grocery stores purposefully layout items to increase shopping time and encourage spending, makes the shopping process more difficult [1]. This process grows easier the more times a user visits a store, but any knowledge can be reset if the layout changes. Similarly, going to a new store for the first time requires the user to relearn the entire layout.

To combat these issues, a smartphone application was designed and developed that would combine crowd-sourcing with machine learning to sort a user’s grocery list based on the layout of the store the user is visiting. The problems being addressed can be solved by learning the layouts of stores and the locations of items. This is knowledge gained by frequenting a store. By applying crowd-sourcing techniques, the app can build a database using the information offered by users. Then, when a user visits a new store, the app can rearrange their list using information gathered by other users. Machine learning is used to fill in gaps left by any missing information.

The final application developed for this project was a cross-platform application built using React Native. The app has a simple user interface that allows users to gather and save item information, item locations, and store information. Users can create and share lists with each other. Users can sort their lists using a variety of criteria and can select the store to base the sorting on using an input box with predictive text or a map API developed using Here Maps. Similarly, users can add items to their lists using predictive text input boxes or through tapping on recommended items. The app also offers recipes through an API called Spoonacular, allowing users to quickly add recipes to their lists. A NoSQL database was created to store all needed information. Finally, a machine learning suite was developed to handle item location prediction, store layout estimation, and recommendation of items.

Acknowledgements

The idea for this application originated from a discussion between Dr. James Green and Dr. Alistair Boyle.

We would like to thank Dr. James Green for his supervision, feedback, and guidance throughout the project.

We would like to thank Dr. Alistair Boyle for his suggestions throughout the project.

Table of Contents

[1 Statement of Objectives 10](#_Toc36740272)

[2 The Engineering Project 12](#_Toc36740273)

[2.1 Health and Safety 12](#_Toc36740274)

[2.2 Engineering Professionalism 13](#_Toc36740275)

[2.3 Relation to Degree Program 13](#_Toc36740276)

[2.4 Project Management and 14](#_Toc36740277)

[2.5 Individual Contributions 19](#_Toc36740278)

[2.5.1 Project Contributions 19](#_Toc36740279)

[2.5.2 Report Contributions 20](#_Toc36740280)

[3 Project Background 23](#_Toc36740281)

[3.1 Existing Apps In The Market 23](#_Toc36740282)

[3.1.1 Out of Milk 24](#_Toc36740283)

[3.1.2 Grocery Shopping List – Listonic 26](#_Toc36740284)

[4 Project Description 30](#_Toc36740285)

[4.1 Project Overview 30](#_Toc36740286)

[4.2 Front End Features 32](#_Toc36740287)

[4.2.1 User Registration and Login 32](#_Toc36740288)

[4.2.2 Crowd-Sourcing Information for Database Growth 33](#_Toc36740289)

[4.2.3 Adding Item Locations 35](#_Toc36740290)

[4.2.4 Registering Store Layouts 38](#_Toc36740291)

[4.2.5 Creating and Managing Lists 42](#_Toc36740292)

[4.2.6 Adding Items To Lists and Recommending Items 45](#_Toc36740293)

[4.2.7 Sorting Lists and Selecting Stores 47](#_Toc36740294)

[4.2.8 Maps Page 52](#_Toc36740295)

[4.2.9 Contacts 56](#_Toc36740296)

[4.2.10 Sharing Lists and Notifications 57](#_Toc36740297)

[4.2.11 Recipes 58](#_Toc36740298)

[4.3 Back-End Features 60](#_Toc36740299)

[4.3.1 Managing the Database Using Google Firebase 61](#_Toc36740300)

[4.3.2 Python and JavaScript Module Used for Machine Learning Capabilities 64](#_Toc36740301)

[4.4 Functional and Non-Functional Requirements 65](#_Toc36740302)

[4.4.1 Functional Requirements 65](#_Toc36740303)

[4.4.2 Non-Functional Requirements 70](#_Toc36740304)

[4.4.2.1 Look and Feel Requirements 70](#_Toc36740305)

[4.4.2.2 Usability and Humanity Requirements 70](#_Toc36740306)

[4.4.2.3 Performance Requirements 70](#_Toc36740307)

[4.4.2.4 Operational and Environmental Requirements 70](#_Toc36740308)

[4.4.2.5 Maintainability and Support Requirements 71](#_Toc36740309)

[4.4.2.6 Security Requirements 71](#_Toc36740310)

[4.4.2.7 Legal Requirements 71](#_Toc36740311)

[5 Development Methods 72](#_Toc36740312)

[5.1 React Native for Application Development 73](#_Toc36740313)

[5.1.1 UI Kitten for User Interface 74](#_Toc36740314)

[5.2 Google Firebase for Database Management 75](#_Toc36740315)

[5.3 Python for Machine Learning Module 76](#_Toc36740316)

[5.4 Validation, Verification and Testing 77](#_Toc36740317)

[6 Crowd Sourcing and Machine Learning 80](#_Toc36740318)

[6.1 Data Collection 80](#_Toc36740319)

[6.1.1 Initial Data Collection 80](#_Toc36740320)

[6.1.2 Maintaining the data in the database 83](#_Toc36740321)

[6.2 ML Algorithms 84](#_Toc36740322)

[6.2.1 Collaborative Filtering (Estimate location of items) 85](#_Toc36740323)

[6.2.2 Weighted Ranking (Sort list of items to follow the shortest path) 92](#_Toc36740324)

[6.2.3 Map Clustering (Estimate layout of the unknown store) 94](#_Toc36740325)

[6.2.4 Association Rule Mining (Recommend items to the user) 100](#_Toc36740326)

[6.2.5 Use of Cloud Functions 106](#_Toc36740327)

[6.2.6 Use of Existing Libraries 108](#_Toc36740328)

[7 User Data Security and App Maintenance 110](#_Toc36740329)

[7.1 Privacy 110](#_Toc36740330)

[7.2 Business Inquiries and Monetization Plan 112](#_Toc36740331)

[7.3 Open-Sourcing the Application 114](#_Toc36740332)

[7.4 App Distribution 117](#_Toc36740333)

[7.4.1 Google Play Store 117](#_Toc36740334)

[7.4.2 Apple App Store 118](#_Toc36740335)

[8 Project Timetables 119](#_Toc36740336)

[8.1 Project Iterations 119](#_Toc36740337)

[8.2 Project Gantt Chart 121](#_Toc36740338)

[9 Conclusions and Recommendations 125](#_Toc36740339)

[10 References 127](#_Toc36740340)

[11 Appendix I – List of Item for Initial Crowd Sourcing 133](#_Toc36740341)

[12 Appendix II – Initial Version of ML Algorithms 136](#_Toc36740342)

[12.1.1 Collaborative Filtering (Estimate location of items) 136](#_Toc36740343)

[12.1.2 Travelling Salesman Problem (Shortest Path and Layout Estimation) 144](#_Toc36740344)

[13 Appendix III - Work Completed At Time Of Progress Report 148](#_Toc36740345)

[13.1 Machine Learning 148](#_Toc36740346)

[13.2 Backend 151](#_Toc36740347)

[13.2.1 Database 151](#_Toc36740348)

[13.2.2 Crowd Sourcing 152](#_Toc36740349)

[13.2.3 Cloud Functions 156](#_Toc36740350)

[13.3 Frontend 157](#_Toc36740351)

[13.3.1 UI Framework 157](#_Toc36740352)

[13.3.1.1 Styling and Theming 158](#_Toc36740353)

[13.3.2 Functionalities 160](#_Toc36740354)

List Of Tables

[Table 1: Project Contributions Of Each Group Member 18](#_Toc35257030)

[Table 2: Report Contributions Of Each Group Member 20](#_Toc35257031)

[Table 3: Example of Store Database 87](#_Toc35257032)

[Table 4: Sample Distance Calculations Using Gathered Data 98](#_Toc35257033)

[Table 5: The Store Names Corresponding To Each Letter 99](#_Toc35257034)

[Table 6: Project Iterations 119](#_Toc35257035)

[Table 7: Project Gantt Chart 121](#_Toc35257036)

List Of Figures

[Figure 1: Sample Code Review 16](#_Toc35257037)

[Figure 2: The Project's Kanban Board 17](#_Toc35257038)

[Figure 3: Sample Issue 18](#_Toc35257039)

[Figure 4: Out of Milk - List Screen 23](#_Toc35257040)

[Figure 5: Out of Milk - Edit Item Screen 24](#_Toc35257041)

[Figure 6: Listonic - List Screen 26](#_Toc35257042)

[Figure 7: Project Architecture 30](#_Toc35257043)

[Figure 8: Sample Login/Set Up Page 31](#_Toc35257044)

[Figure 9: Sample Register Item Page 34](#_Toc35257045)

[Figure 10: Add Item Location Page 35](#_Toc35257046)

[Figure 11: Sample Map Creator Page 37](#_Toc35257047)

[Figure 12: Mockup Of the Map Creator Page Using A 2D Layout 39](#_Toc35257048)

[Figure 13: Sample List Page 41](#_Toc35257049)

[Figure 14: Sample Of Created List 42](#_Toc35257050)

[Figure 15: Add Item To List Page 44](#_Toc35257051)

[Figure 16: The Sorting Dropdown Menu 46](#_Toc35257052)

[Figure 17: Select Store Page 48](#_Toc35257053)

[Figure 18: List Sorted Based On Fastest Path 49](#_Toc35257054)

[Figure 19: Maps Page On Andriod With Current User Location And Nearby Stores 53](#_Toc35257055)

[Figure 20: Maps Page On Android With Search Results Provided By Here Maps API 54](#_Toc35257056)

[Figure 21: Summary Of Notification Use [9] 56](#_Toc35257057)

[Figure 22: Summary Of Architecture 59](#_Toc35257058)

[Figure 23: Sample Capture Of Aisle Information 153](#_Toc35257059)

[Figure 24: Sample Of Generated Floor Layout Using Collected Data 155](#_Toc35257060)

[Figure 25: Light Theme (Left) And Dark Theme (Right) Applied To Home Page 158](#_Toc35257061)

[Figure 26: Your List Page (Left) And Current List Page (Right) 159](#_Toc35257062)

# Statement of Objectives

The objective of this project was to develop an application that can be used to store and sequentially sort grocery lists based on the layouts of various stores. Most consumers iteratively generate grocery lists and add food items in random order throughout the week. One of the problems with this strategy is that if the consumer were to shop based on their list sequentially, they would end up going back and forth between aisles. A preventative measure of this issue includes carefully analyzing the entire list after obtaining each item in order to identify the next closest item to select. This countermeasure is inefficient, error-prone, and can make grocery shopping more troublesome than need be. Another issue that a consumer may encounter is the intimidation of trying new stores because of their lack of familiarity with said store’s items and their respective locations. The question of whether a store sells specific items, or thought of wandering around a store, walking back and forth between aisles, is enough to deter consumers from new stores. The objective of this application is to ease the task of grocery shopping by automatically rearranging the list to the order that shoppers would encounter the items based on any store the user visits.

A simple and user-friendly interface was created, allowing for the gathering and validation of store layouts and food item locations using a crowd-sourced methodology. Crowdsourcing is used to obtain relevant information such as the layout of a store and the locations of items within that store. Furthermore, machine learning is used to provide a variety of features including the rearrangement of food items within lists to the estimated order that one would expect to see the food items as they sequentially move throughout the aisles, the recommendation of food items based on items frequently bought together by users, and estimated item locations based on the location of that item in other stores saved in the database. The application is primarily targeted at those who grocery shop periodically. The application can be used as a tool during the user’s weekly shopping trips, to plan parties/events, or for any other instance that one may need to go grocery shopping. As this group is the primary target for the application, an important feature is the capability to allow for collaborative editing. The final deliverable is an application available to be downloaded onto Android and iOS mobile devices.

# The Engineering Project

## Health and Safety

This project was a completely software-based application with no impact on anyone’s health or safety. The only part that contained a physical aspect was the data collection phase of the project where group members travelled to grocery stores to gather data. While performing data collection, group members employed the recommendations of Carleton’s Working Alone Guidelines [66]. This included notifying the other group members of when and where one was going to ensure they could be checked on regularly.

Another significant portion of this project involved collecting data from users and mining user data. Though this will not impact the user’s physical health and safety, it is important to maintain the safety of their data. As this is the case, an extensive Privacy Policy was developed. This Privacy Policy is outlined in depth in section *7.1 Privacy* and a related section is section *7 User Data Security and App Maintenance.* A quick overview of the Privacy Policy is that data will be kept anonymized. Additionally, all machine learning algorithms will learn from general user data, but will not learn specific user patterns. Should this Policy change in the future, users will be notified of the changes along with a summary of changes, and they will have the option to opt-out. Finally, in accordance with Canada’s Personal Information Protection and Electronic Documents Act, users will be notified of any data breaches of the app [15].

## Engineering Professionalism

Engineers have a variety of duties, but above all else, they have a duty to public welfare, should act as faithful agents to clients and employers, and they should perform due diligence to mitigate risks [16]. While it is difficult to perform all these duties to a significant degree during this project, they were all exercised to a certain degree.

The paramount duty of an engineer is the duty to public welfare. While this app will not have a significant impact on public welfare, this duty was exercised in the form of protecting user data. As stated previously, a Privacy Policy was developed which can be seen in section *7.1 Privacy.* This Privacy Policy reflects the importance of protecting user data so that it is not abused or mishandled. Similarly, due diligence has been performed to mitigate the risk of data breaches. To access their account, users are required to enter an email and password. This ensures that only authorized users will be able to access their corresponding data. These email and password combinations are kept encrypted by Google Firebase’s encryption strategy. Furthermore, rules have also been designed for the database to prevent data from being accessed through external HTTP requests. To access data, one must have the database key and be verified, which means accesses can only come from the app at the request of logged-in users.

## Relation to Degree Program

Creating an application that can be used on mobile or web platforms requires multiple skills developed throughout the software engineering degree program. Firstly, a plan of action was devised, and a rough timetable with deliverables was created to outline the project. Use of Gantt charts, Use Case Diagrams, and Requirements Elicitation are a handful of skills developed in courses such as Software Requirements Engineering (SYSC 3120) that helped in the initial planning phase of the project. During the development process, the skills learned throughout various programming courses are applicable and aid when learning and utilizing fewer familiar technologies. Next, the writing and communication skills learned throughout Communication Skills for Engineering Students (CCDP 2100) have proven beneficial during the creation of the other deliverables of the projects (poster fair, report, etc.). Lastly, as the course Verification and Validation (SYSC 4101) is completed, the different testing techniques learned can be applied to ensure the specified requirements outlined during planning are met.

Software Engineering teaches how to develop large software projects in a team, proper techniques to use to ensure that code is scalable, and strategies to use to maintain code after release. This project required a culmination of all of the aspects covered during course work.

## Project Management

One of the ways to define the term ‘Software Engineering’ is the process of analyzing consumer wants and needs, and then designing, constructing, and testing an application that satisfies those needs [13]. The average American spends sixty hours a year doing grocery shopping [14]. This does not include purchasing an item twice because miscommunication has led to multiple trips to the grocery store, driving back to the store because you forgot to buy an item or any other communication-related hindrances that come with the task of grocery shopping. Through observation of daily activities, user habits, and personal experience, it was decided that creating a featured grocery list app would allow users to grocery shop more efficiently and effectively.

There are multiple strategies to use when developing a software project. These generally determine what cycle of planning, development, and testing developers will follow when creating a project. To develop this project a Waterfall strategy was used where most of the planning is done as the first step before starting development. This project required the group members to apply the project planning strategies learned through course work.

When developing a large software project, we are taught to use version control, testing, and code reviews. This is done to ensure that all aspects of the project are functional, that every member of the team has at least a rough understanding of all components in the codebase, and that bugs can be quickly fixed. This process was greatly exercised in this project. To perform version control, GitHub was used. When a group member was working on a new feature they would create a new branch, write the code, create a pull request, wait for reviews from the other members, and then merge the pull request. This process helped to ensure that all changes were tested on both iOS and Android devices and that at least two developers were aware of a change. A sample of a code review from this process can be seen below in Figure 1:

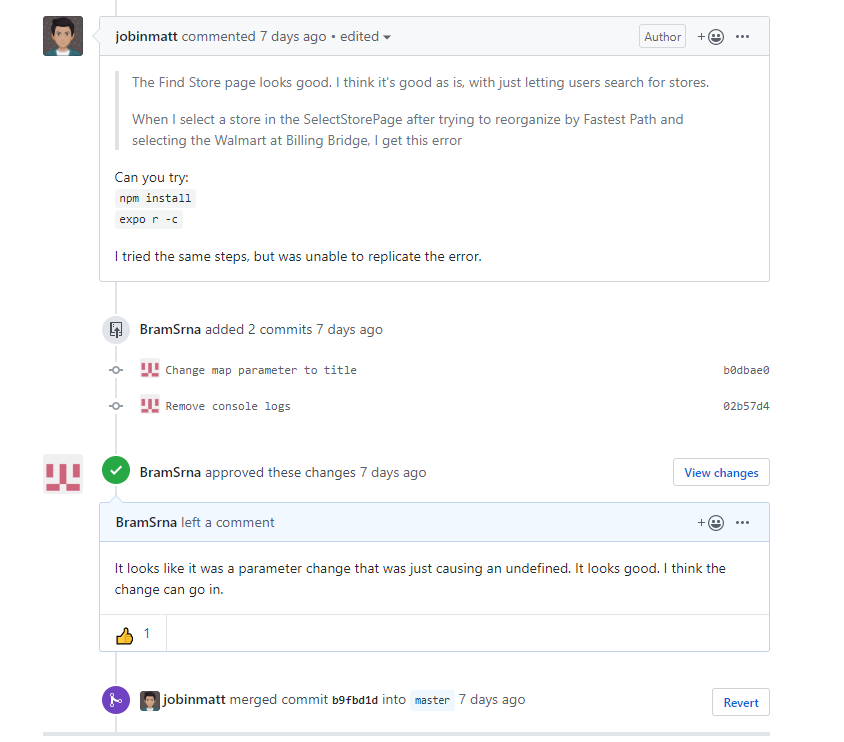


Figure 1: Sample Code Review

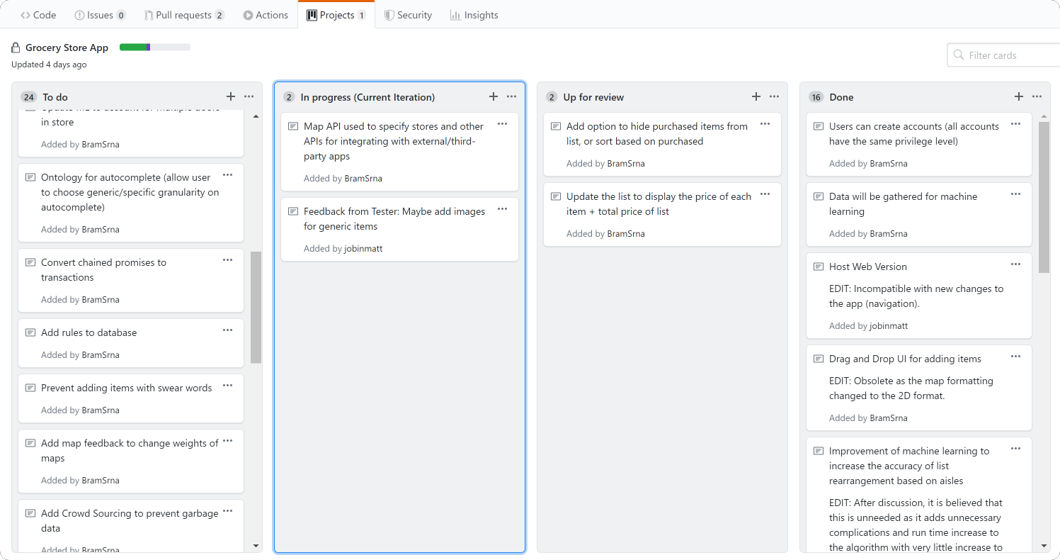
As for the development process, an Agile method was followed. The group members would have frequent meetings to discuss the next stages of the project. This included weekly meetings with the project supervisor, Professor James Green. The code was developed using a modular approach to ensure it could continue to scale to meet all project requirements. This process also helped to fix issues as they would appear in the codebase. To keep track of the project, a Kanban board was used as can be seen below in Figure 2:

Figure 2: The Project's Kanban Board

The features to implement in the project were maintained on this board. As can be seen, the Kanban board contained four buckets: To-do, In Progress (Current Iteration), Up For Review, and Done. Elements planned for the future are first put in the To-do bucket. The project followed an iterative design process, so each iteration, the features for that iteration would move from To-do to In Progress (Current Iteration). Once a developer finished a feature and created a pull request, the item would move to Up For Review. After the pull request was merged, the item was moved to the Done Bucket. In addition to a Kanban board, issues were also used. Issues allow developers to mark bugs that need to be fixed and suggest enhancements that could be made. An example issue can be seen below in Figure 3:

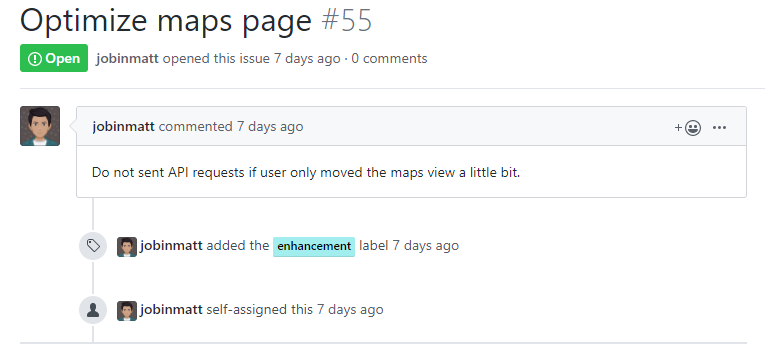


Figure 3: Sample Issue

## Individual Contributions

### Project Contributions

Table 1: Project Contributions Of Each Group Member

|  |  |
| --- | --- |
| **Developer Name, Program, and Student Number** | **Contribution** |
| Haseeb Khan (SE) (101009713) | * Pages for creating and modifying lists * Application notification system * Capability for users to share lists with each other (including communicating with those within a list) and add contacts * Development of recipes and sharing recipes functionality * Manual data collection |
| Jobin Mathew (SE) (101006153) | * Maintenance of user interface and enhancing app UI using UI Kitten module * User registration, login pages and several other pages such as user accounts, etc. * Development of store selection functionality using Here Maps * Manual data collection |
| Abraham Srna (SE) (100997482) | * Pages to handle crowd-sourcing * Machine Learning suite * Maintenance of database * Manual data collection |

### Report Contributions

Table 2: Report Contributions Of Each Group Member

|  |  |
| --- | --- |
| **Developer Name, Program, and Student Number** | **Contribution** |
| Haseeb Khan (SE) (101009713) | * 1 Statement of Objectives * 2 The Engineering Project (All Subsections) * 4 Project Description (4.2.5, 4.2.6, 4.2.9, 4.2.10, 4.2.11) * 5 Development Methods (5.2) * 7 User Data Security and App Maintenance (7.4) * 8 Project Timetables * 9 Conclusions and Recommendations * 10 References * Appendix III |
| Jobin Mathew (SE) (101006153) | * Abstract * 2 The Engineering Project (2.5) * 3 Project Description (All Subsections) * 4 Project Description (4.1, 4.2.1, 4.2.2, 4.2.3, 4.2.4, 4.2.7, 4.2.8) * 5 Development Methods (5.1) * 7 User Data Security and App Maintenance (7.1, 7.2) * 8 Project Timetables * 10 References * Appendix III |
| Abraham Srna (SE) (100997482) | * 2 The Engineering Project (2.5) * 4 Project Description (4.3, 4.4) * 5 Development Methods (5.3, 5.4) * 6 Crowd Sourcing and Machine Learning (All Subsections) * 7 User Data Security and App Maintenance (7.3) * 8 Project Timetables * 10 References * Appendix I * Appendix II * Appendix III |

# Project Background

Many people use grocery lists to keep track of what they need to buy while grocery shopping. In today's technological era, users often use device-specific text interfaces such as ’Samsung Notes’ [2] and ’Notes’ [3], or third-party list applications to maintain grocery lists. The existing alternatives all seem to be lacking features to enhance and simplify the user’s shopping experience. These third-party list implementations lack features that enable the user to complete their grocery shopping as efficiently and effectively as possible. The purpose of this project is to provide users with a digital list that they can share amongst one another, and that makes the routine task of grocery shopping simpler and more efficient.

## Existing Apps In The Market

There are several other applications that are available on the marketplace that try to incorporate some of the functionality that is being proposed in this project. Two of the most popular apps for creating grocery lists are briefly described below alongside their user interface and some of the functionality that each of them provides. These applications were found by searching for “Grocery Shopping List” on the Google Play Store. The most popular apps in terms of downloads were then selected. The first application, “*Out of Milk*” has over 5 million downloads [4], and the second application, “*Grocery Shopping List – Listonic*” has over 1 million downloads [5].

### Out of Milk

The app named ‘Out of Milk’ is a cross-platform grocery list application with over 5 million downloads on the Google Play Store. It allows users to create and share shopping lists, pantry lists, and to-do lists [4].

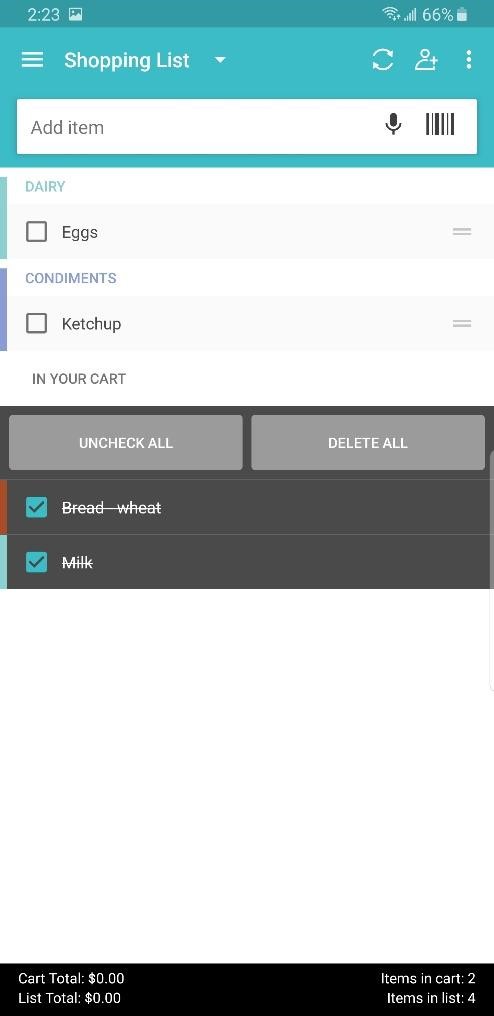


Figure 4: Out of Milk - List Screen

This app has a relatively simple design to manage a user’s grocery list. Figure 4 shows the user interface used to maintain lists in Out of Milk. Users can quickly uncheck items as they are collected. Additionally, the list also displays other helpful information to the user, including the number of items that are in the list and the list’s total cost. It allows users to add items to a list manually or by scanning the barcode of a product. General grocery items added to the list automatically get categorized into sections based on the department.

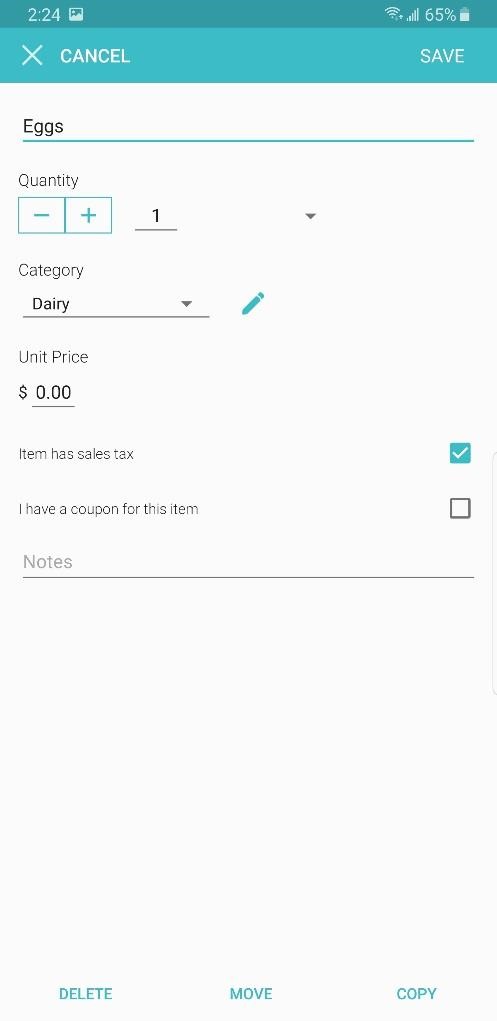


Figure 5: Out of Milk - Edit Item Screen

Users are also allowed to manually enter quantity, modify the category, enter price information, and add any additional notes to an item in the list, as can be seen in Figure 5 above.

This app is very simple to use. Creating and maintaining lists is intuitive, using a clean and straightforward interface. Items are automatically sorted into departments as they are added to the list. Additionally, it offers a wide range of methods that can be used to add items, including manually typing the name alongside predictive search, scanning a barcode, or vocally stating the name of the item. Lists can also be shared between users. This app also has several limitations. Although lists can be reordered manually by moving the individual items, the application does not allow the department tabs to be rearranged to match the store. Another downside is that the cost of the items needs to be manually inputted, meaning prior knowledge of item prices is mandatory to make use of the functionality. Overall, the app offers a solid base of features with a simple interface but still contains a few flaws.

### Grocery Shopping List – Listonic

The app named ‘Listonic’ is a cross-platform grocery list application with over one million downloads on the Google Play Store. It claims to make grocery shopping easier, faster and smarter [5]. The app allows users to create and share lists with other users.

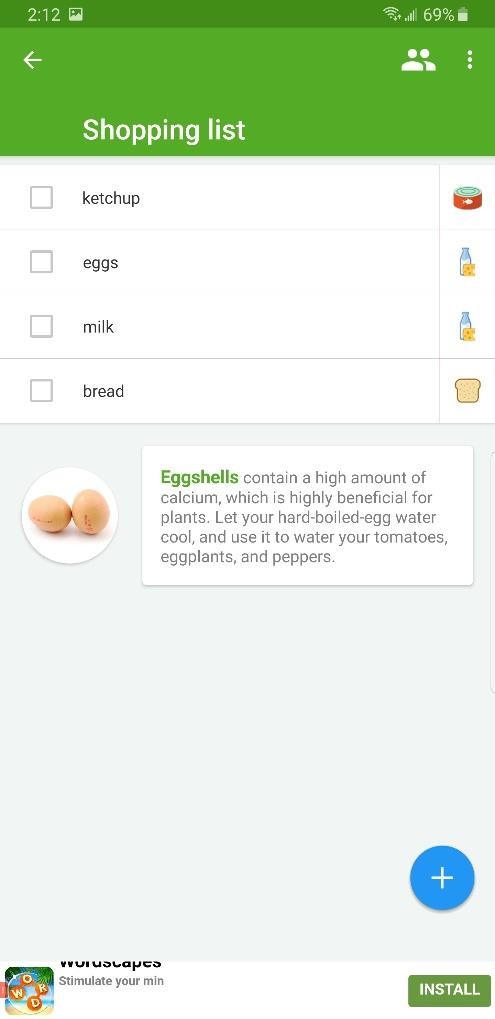


Figure 6: Listonic - List Screen

Similar to the other apps on the app store, this app also lets the user add items to their grocery list, which automatically gets categorized based on the departments that each item belongs to, as seen in Figure 6 above. The app provides suggestions of items to add to the grocery list as the user starts typing; this is similar to an auto-complete feature. The app also offers hints and tips about some of the items that are in the users’ grocery list.

One of the most significant strengths of Listonic is its security policy. When the user first opens Listonic, the app displays the full security agreement and allows the user to control how their information will be used. For example, the user can disable personal information being used to cater to advertisements that are based on user behaviour and information. In addition to ad selection, the user can also disable measurements, content selection, personalization, and information storage. Another strength is the full customization of list orders. Lists can be sorted alphabetically, by the department, or using a custom sort order. Furthermore, lists can be shared for collaboration with others. Finally, the app also displays cooking tips for certain items in the list, which sets it apart when compared to other list apps.

As seen above, two of the most popular grocery-list apps have several useful features, but they both sacrifice features and customization for simplicity. For example, although both *Out of Milk* and *Listonic* sort items based on departments, the reorganized list may not be accurate because different stores may have different departments and items within each department. Also, since departments may appear in different orders depending on the store, the user must identify the closest department before they begin shopping for items on their list. This project proposes an application that offers the basic features found in many grocery apps, but supplements the capabilities of the app with a feature suite that simplifies the shopping experience. The proposed application will help users create and manage lists by providing the user with more capabilities in terms of rearranging items.

# Project Description

The main deliverable from this project is a mobile application that can store and sort users’ grocery lists based on the layout of the selected grocery store. At its core, the primary purpose of the application is to maintain a user’s grocery lists. When a user visits a store and opens their list to begin shopping, they can reorganize the list based on the layout of the store using a simple button press. This allows the user to follow the optimal path in the store for finding items. The application uses crowdsourcing to collect the item location information and store layouts used for sorting lists. There are several large components involved in this project.

## Project Overview

There are three major components to this project. The main component is an application that users can interact with to utilize the app's features. This app is built using React Native and is currently hosted on Expo. The app allows users to create profiles, create lists, modify lists, share lists with other users, and browse recipes. The app also allows users to specify which store they are shopping in and rearrange their lists using a variety of options

including several that are implemented using machine learning. The app interface is also how users can enter data to crowd-source information.

The second component is a Firebase database. This is a NoSQL database that is used to maintain all the information that the app needs to function including user credentials, crowd-sourced information, list information, and contact information. The app interacts with the database when using most of its features.

The final component is a set of Google cloud functions running alongside the database. Cloud functions are functions running as Functions as a Service, meaning they are accessed through HTTP requests. They have the benefit of running on hardware separate from the user’s device, so they can run quickly. These cloud functions are used to update variables in the

database as information is gathered and to run the machine learning functions such as generating recommendations and organizing lists.

A visual representation of these three components can be seen below in Figure 7 and are expanded upon throughout this document:

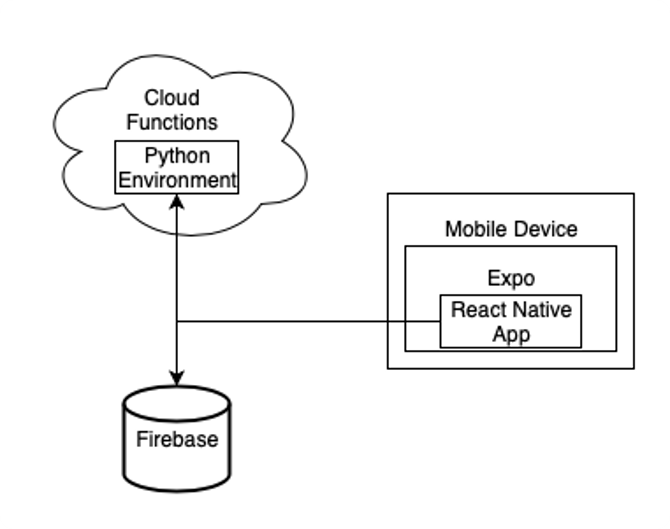


Figure 7: Project Architecture

## Front End Features

### User Registration and Login

From the view of the front-end, there are several features available to the user. The first of these features is the user account. Users can sign-up with the app using their email. Once a user has created an account, they are able to login to the app using a simple login page, as seen in Figure 8. Once logged in, users can create and store their lists in the cloud. Additionally, they are able to share their lists with other users, allowing both users to access and modify the list in real-time. User information is stored using Google Firebase, the justification of which will be discussed in section *5.2 Google Firebase for Database Management*.

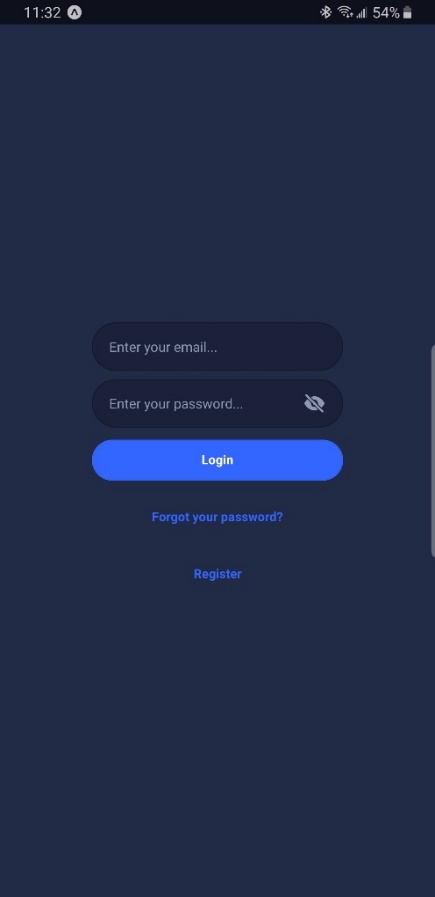


Figure 8: Sample Login/Set Up Page

### Crowd-Sourcing Information for Database Growth

Crowdsourcing is used to gather a wide range of information and to validate certain machine learning algorithms. The information being gathered using crowdsourcing includes item locations, item information such as costs and sizes, and the layouts of stores. One issue with this approach is how to handle garbage data in the form of incorrect data. To address this, each piece of inputted data is assigned a weight that signifies the accuracy of the piece of data. As more users corroborate the data, the weight is increased. This helps to ensure that accurate data is used by the application.

Another issue with this approach is how to handle changes in the data. To address this issue, the weighted majority could be updated to account for how long ago the data was sourced. For example, data added the previous day would be weighted a lot higher than data added the previous year. In addition to this, the functionality could be added to recognize patterns. For example, if the price of an item changes depending on the season, then data gathered during that season could be weighed heavier than data from other seasons. Crowdsourcing is also be used to validate the results of the machine learning modules.

Feedback functionality has also been added to the application to validate the machine learning modules. When the user reorganizes their list based on a method that requires store information, they have the option to view the map used by the machine learning module during resorting. The user can then update and modify this map as needed. This updated map is then used to update the weights saved in the database so that future sorting is more accurate. This strategy is discussed in more depth in Section *6.2.2 Weighted Ranking (Sort list of items to follow the shortest path)*.

### Adding Item Locations

The next feature available for users is the ability to register items to the central database and add locations for those items. Items are registered by inputting the item name and important information, as seen in Figure 9. Item locations are added by inputting an item name, store name, department name, and aisle number of that item using the Add Item Location page, as seen in Figure 10. The app uses the location of items in the store to sort the list based on the best path. The re-sorting is only based on the department location of items, but this could be expanded to be based on aisles within the department.

To gather the locations of each item in each store, crowdsourcing is used. When the location of an item is unknown, or the location is incorrect, users can use the app to enter the location information of the item. The new location is then added to the database, or in the case of updating incorrect information, the information in the database is updated. If a user is attempting to find an item and the location is unknown, then the application estimates the location using a collaborative filter. The application compares stores to find the instances most similar to the store the user is shopping in. This similarity metric is based on the common departments in each store and the items found in those departments. The most similar stores are then be used to estimate the location of the item. This discussion is furthered in section *6.2.2 Weighted Ranking (Sort list of items to follow the shortest path and Estimate layout of the unknown store).*

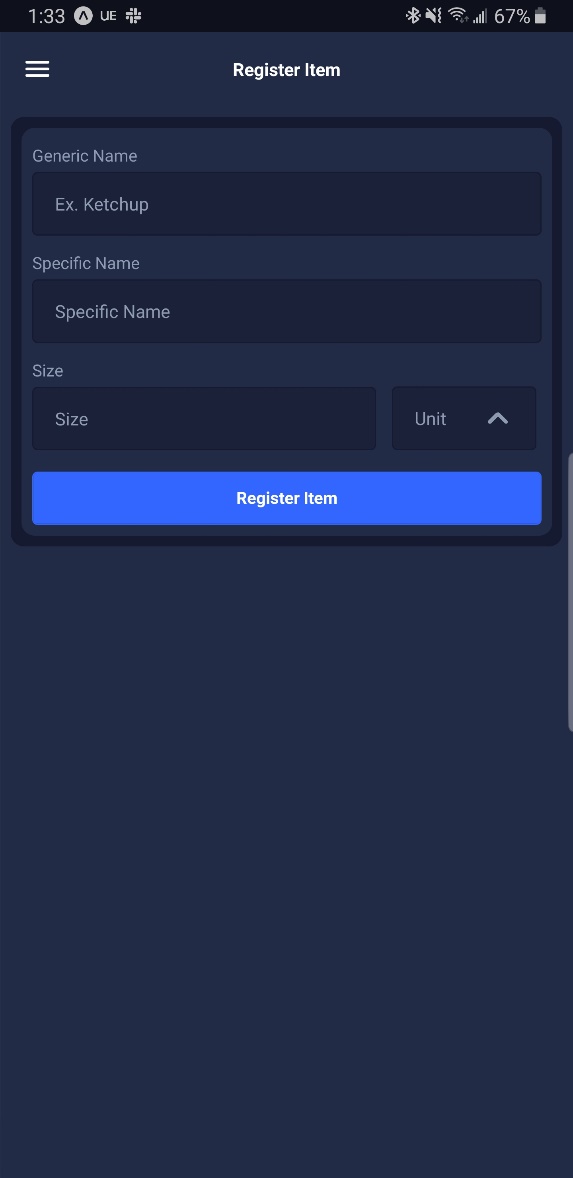


Figure 9: Sample Register Item Page

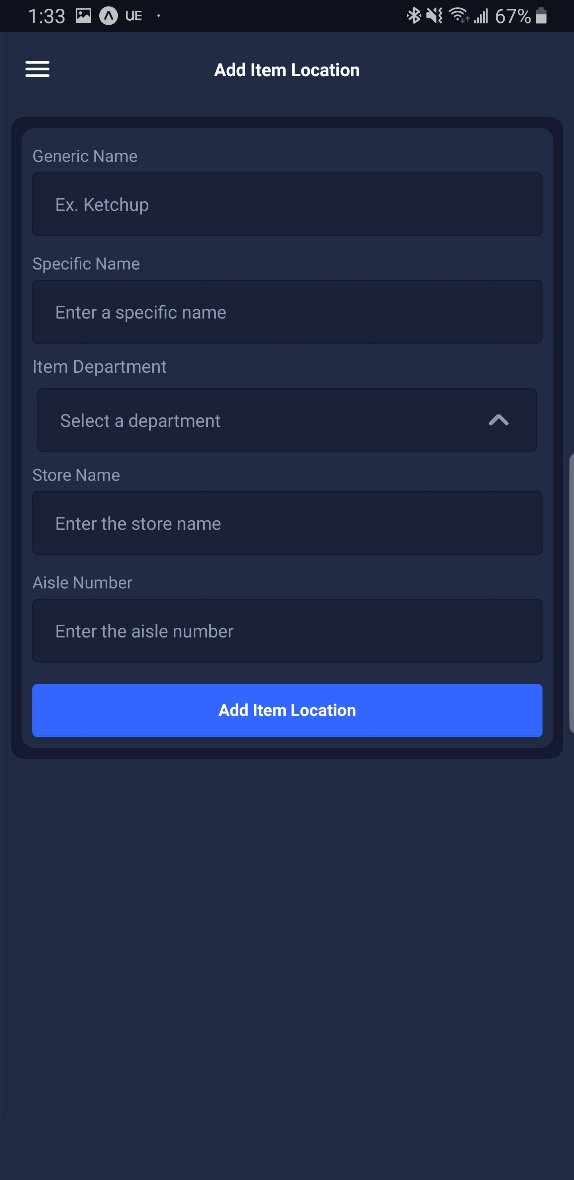


Figure 10: Add Item Location Page

The item information is primarily used for reorganizing the grocery lists and for the machine learning capabilities. For the sorting, the name of the item and its location in the store is needed. The store location information includes the name of the store, the department the item is in, and the aisle the item is in. This necessary information can be used for the main features of the application. Additional information such as the brand of the item, the type of the item, the cost of the item, and other significant tags could potentially be used to improve the accuracy of the machine learning, and to improve the user experience.

To simplify the process of adding items, a drag-and-drop style interface could be developed. Using this interface, the user will enter the name and other important information about the item. The user will then select the location of this item in the store using the map. This would simplify the process of adding items for users. In addition to the above, there are several more potential areas of improvement, including adding items to the database by taking pictures of the item, or by scanning barcodes.

### Registering Store Layouts

The third feature for users is the ability to map out stores. Similar to adding items, users can map the layout of stores by sorting the departments in the store in a list format based on the order that the departments appear in the store. A screenshot of this user-interface can be seen in Figure 11. For example, when the user enters a store, they may find that the Bakery is the closest department, so this would be the first element in the list. Then, if the Produce department is the next closest, this would be the second element in the list. This process would repeat for each department in the store. This layout is used to sort the lists, as it provides a macro view to be used alongside the item locations. Departments can be added, removed, and reordered as the user requires.

This feature could be expanded to incorporate aisles. Grocery stores often include signs above each aisle that supplies the number of the aisle and the type of items in that aisle. This feature could be changed to allow the user to input the aisles and tags in each department as they are mapping out the store. This information could be incorporated in the automatic reordering of lists to allow the calculation to be more informed.

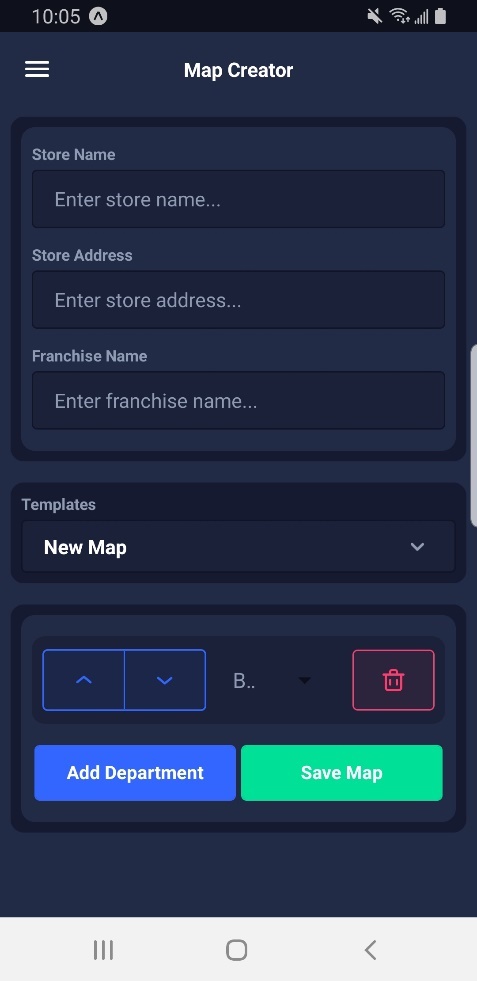


Figure 11: Sample Map Creator Page

To simplify the map creating process and make it easier for the user, the page includes a selection of templates that can be used to quickly populate a map. These templates are New Map, One Of Each, Most Popular, and Used By Optimizer. The New Map template is the default option and creates an empty map. This can be used by the user to reset maps and start from scratch. The One Of Each template adds one of each type of department to the map in alphabetical order by department name. This template helps to map out larger stores with more departments. The departments can then be moved and deleted as needed. The third template, Most Popular, is the map that has been inputted by the most amount of users. The database saves each map and keeps track of the validity of each map based on user feedback. This template populates the map creator with the map that has been verified by the most amount of users. The final template, Used By Optimizer, is the map used by the backend machine learning suite to rearrange a list. This is the map calculated for a store by the suite based on all of the inputted maps. The algorithm used to determine this map is discussed in section *6.2.2 Weighted Ranking (Sort list of items to follow the shortest path)*.

An alternate strategy for mapping a store that was investigated was the ability to create a 2D map of the store layout by colouring in cells in a grid to match departments. A mockup of this alternative can be seen below in Figure 12:

A screenshot of a cell phone

Description automatically generated

Figure 12: Mockup Of the Map Creator Page Using A 2D Layout

This was the original strategy that was planned for the application. To determine the fastest path in a store, a solution to the travelling salesman problem was going to be used as discussed in *Appendix II – Initial Version Of ML Algorithms*. As a quick overview, two solutions to the travelling salesman problem were investigated: the Christofides algorithm and the Greedy Algorithm along with the k-opt improvement strategy. The computational overhead of these solutions was going to be addressed by having the algorithm run as a cloud function which would greatly improve the runtime of the algorithm. Cloud Functions are discussed more in-depth in section *5.2 Google Firebase for Database Management.* The reason that this method was ultimately scrapped was due to user feedback on the frontend implementation. It was found that the grid implementation is difficult to use on a phone screen. This could be addressed by reducing the amount of cells in the grid, but to get the grid to a size where it is usable, a lot of the granularity in the map would be lost. An additional reason that the method was changed from the 2D map to 1D list was because the 2D map is not very scalable. If later on it was decided that more granular detail such as aisle numbers and tags should be gathered, the 2D map would be unable to support the change.

### Creating and Managing Lists

Another feature available to users is the ability to create and share lists. Users can create lists once they register for an account. A simple mockup screen for creating new lists can be seen in Figure 9. The lists are filled by adding items.

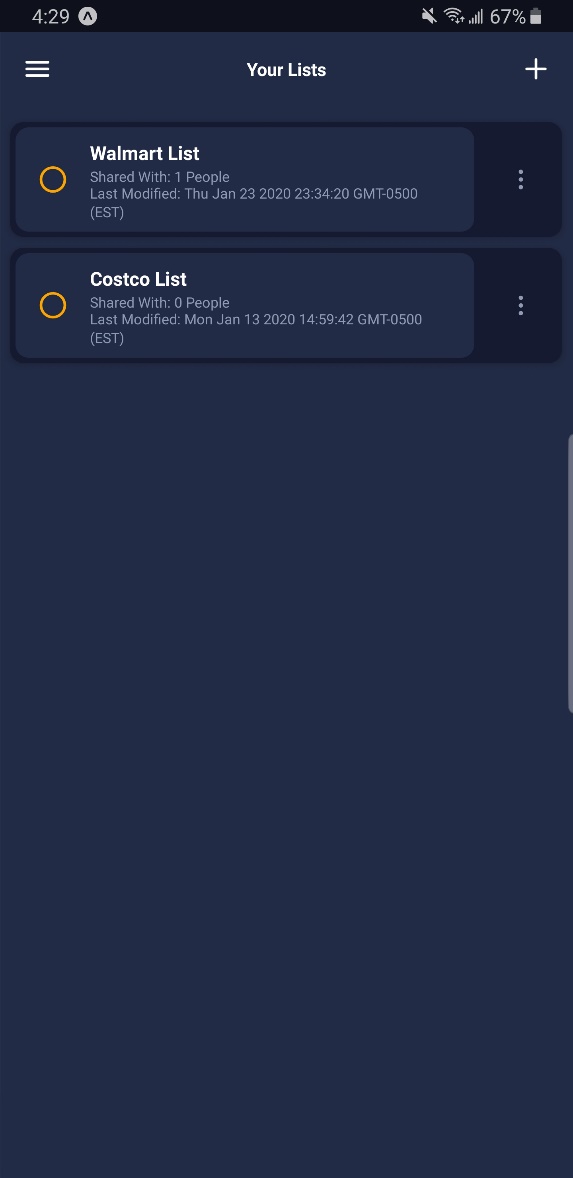


Figure 13: Sample List Page

Once the user has created a list, they can populate it with items, share the list with others, and maintain the list while shopping. A sample of a created list can be seen below in Figure 10.

A close up of a device

Description automatically generated

Figure 14: Sample Of Created List

Users can mark items as purchased while shopping by double tapping on the item. Additionally, items can be removed from the list using the drop down menu attached to each item. The list also contains a dashboard that provides a quick overview of the contents of the list. This overview includes the number of items in the list, number of users the list is shared with, and a price range of the list based on the information in the database. Furthermore, each individual item has a price range based on the database information. If the pricing information of an item is unknown, then no price is displayed for that item. The price range in the dashboard also contains a counter of the number of items with unknown pricing information.

### Adding Items To Lists and Recommending Items

A related feature to creating lists is the capability to maintain lists through adding items. Users can add items through two options: an input box that requires manual input or check boxes to quickly add recommended items as can be seen in Figure 11:

A screenshot of a video game

Description automatically generated

Figure 15: Add Item To List Page

The input box includes predictive searches based on the contents of the database. As the user types, a drop down menu will appear that the user can use to select an item they want to add to their list. The recommended items are based on the current contents of the user’s list and the history of all previous users. The methodology used for deciding which items to recommend is discussed in depth in section *6.2.4 Association Rule Mining (Recommend items to the user).* An overview of the methodology is that all grocery lists in the app are parsed to search for frequent subsets of items in lists. These subsets create rules that are used to recommend items. The rules are of the form “if subset X is in the list, the recommend item Y”. When it comes time to actually recommend items, the list is parsed to see if any rules are triggered, and if they are, the result of the rule is offered to the user. If there are no rules being followed, then the app will simply recommend the most popular items.

### Sorting Lists and Selecting Stores

One of the main features of the application is the ability to sort lists based on a variety of criteria. Using a dropdown menu when looking at a list, the user can select between six criteria: Order Added, Alphabetically, By Location, Fastest Path, Fastest Path (Auto Update), and Purchased. The dropdown menu can be seen in figure 12 below:

A screenshot of a cell phone

Description automatically generated

Figure 16: The Sorting Dropdown Menu

Each of the sorting methods is based on a different criteria. The Order Added criteria is based on the order that the individual items were added to the list. The first item added is placed at the top of the list and the last item is placed at the end of the list. The Order Added criteria is the default sort method. The Alphabetically criteria is based on the names of the items, following the normal alphanumerical order. The Purchased criteria is based on whether an item has been marked as purchased yet. This sorting methodology places unpurchased items at the top of the list and purchased items at the bottom. Similarly, there is also the option to hide purchased items from a list. In the upper-right corner of Figure 12 above, there is a box. When this box is tapped, all items marked as purchased are hidden from the list.

The remaining sorting methods, By Location, Fastest Path, and Fastest Path (Auto Update), are all based on the locations of the items in the list in the store. The By Location sorting method groups items based on the departments and aisles where they are found. The groups are then sorted based on the alphabetical order of the department names and numerical order of the aisles. The Fastest Path sorting method also groups by department and aisle, but instead of reordering the groups based on the department names, the groups are reordered based on the path that would get the user through the store the fastest. The Fastest Path (Auto Update) sorting method is the same as Fastest Path, but it will continue to update the path as the user marks items as purchased. It will place purchased items at the bottom of the list, then reorder the unpurchased items to put them in the fastest path the user can follow from their current location. As these sorting methods require item locations and store maps, a machine learning suite is used to predict item locations and store maps as needed, which is discussed in section *6.2 ML Algorithms*.

A related feature to the sorting of lists, is the ability for a user to select the store they want to use for the sorting. The location based sorting methods, By Location, Fastest Path, and Fastest Path (Auto Update), require the user to enter the store name when they select one of these sorting methods. The user can either use an input box with predictive text based on stores saved in the database or a map API done through Here Maps to select the store. The page for selecting the store can be seen below in Figure 13:



Figure 17: Select Store Page

Once the user enters the store, the list is rearranged based on the soring methodology. In addition to a resort, the items are also coloured based on their department, as seen below in Figure 14:

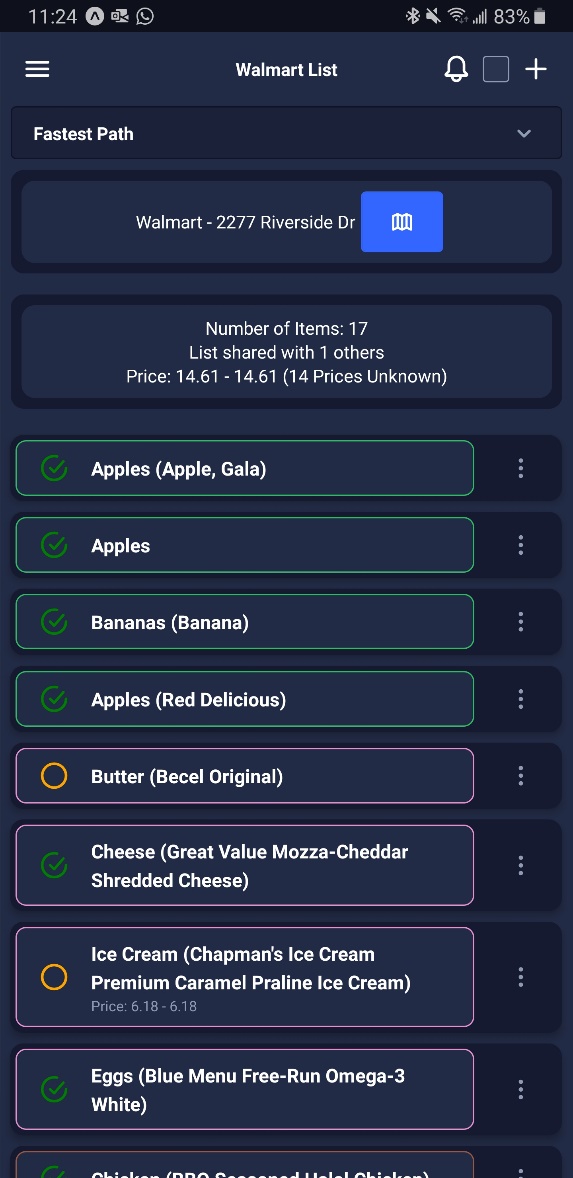


Figure 18: List Sorted Based On Fastest Path

One final aspect to note is the dashboard added below the sort method picker after By Location, Fastest Path, or Fastest Path (Auto Update) is selected. This dashboard contains the store name and address given by the user as well as a button that can be used to view the map used for sorting the list. This button brings the user to the map creator page, with the page prepopulated with the map used for sorting. The user can then edit and save the map.

### Maps Page

The maps page of the application is designed to improve the user experience when selecting a store or searching for a store. The maps page has two main functions. The main function allows the user to quickly select a grocery store from a list of available stores near their current location from a map. The second functionality allows the user to search for a store relative to their current location using a search bar. In order to implement these functions, several popular map API providers were investigated, including Google, Microsoft, and Here Technologies.

Google Maps Platform was the option since it was the most popular one used across several popular applications. Some of the features and advantages that the Google Maps Platform had were that it had almost 99% coverage of the world, 25 million daily updates and one billion monthly active users [6]. The biggest drawback of the Google Maps Platform with regards was that it requires a billings account before it can be used. This may lead to extraneous charges during development.

Microsoft’s Bing Maps was the second service that was investigated as it is the second most popular maps API provider across North America. Bing maps had several features that could be useful for this project including a ‘find location by address’ feature that could have been useful, but during testing, some of the basic API queries would return incomplete results hence, this was abandoned.

Here Technologies was the final provider investigated since it had a growing developer community, a useful feature set available to developers in the free tier, and some of its functionalities were even used by other map API providers, such as Bing Maps [7]. Here Maps provided robust documentation and even a map editor for developers to update and clarify issues in the global map. Ultimately Here maps was selected as the maps API provider as it had a free tier with a generous API request limit of 250K requests per month.

The map view in the maps page was designed and generated using a react native library called ‘react-native maps’ [8]. The react-native-maps library is the most popular and most functional library available for developing a maps view on both Android and iOS in react native. It has an active community working to fix bugs and improve its functionality.

In this project, the maps view on Android is displayed using Google maps and the maps view on iOS is displayed using Apple maps. This was done to ensure maximum compatibility of features on both platforms. As seen in figure 15, when the user opens the maps page, the user will be shown a map view of their current location, with their current location represented by a red pin. If the user has not enabled location permissions for the application they will be requested to allow location permissions before using this feature.

As seen in Figure 15, the map view also shows the user up to a maximum of 20 of the closest grocery stores near their current location using a green pins. These store locations are fetched by using the Here maps API. If the user wants to find stores near another location, the user can simply drag the map to the new location–center of the screen, and the Here maps API will update the store results to represent the new location.

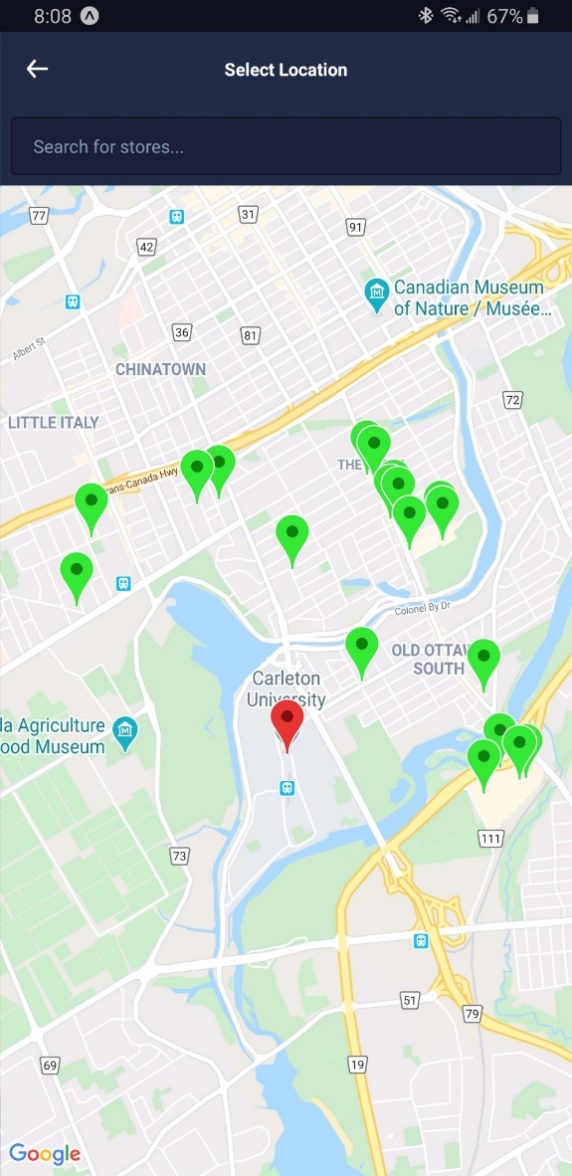


Figure 19: Maps Page On Android With Current User Location And Nearby Stores

Users can also search for stores using the search bar that is present above the map view (see Figure 16). This field provides results to the user in a dropdown menu that shows the closest matching stores to the entered query. Selecting an option from the dropdown will center the map view to the selected store location. The user can simply select the location from the map view to confirm store selection.

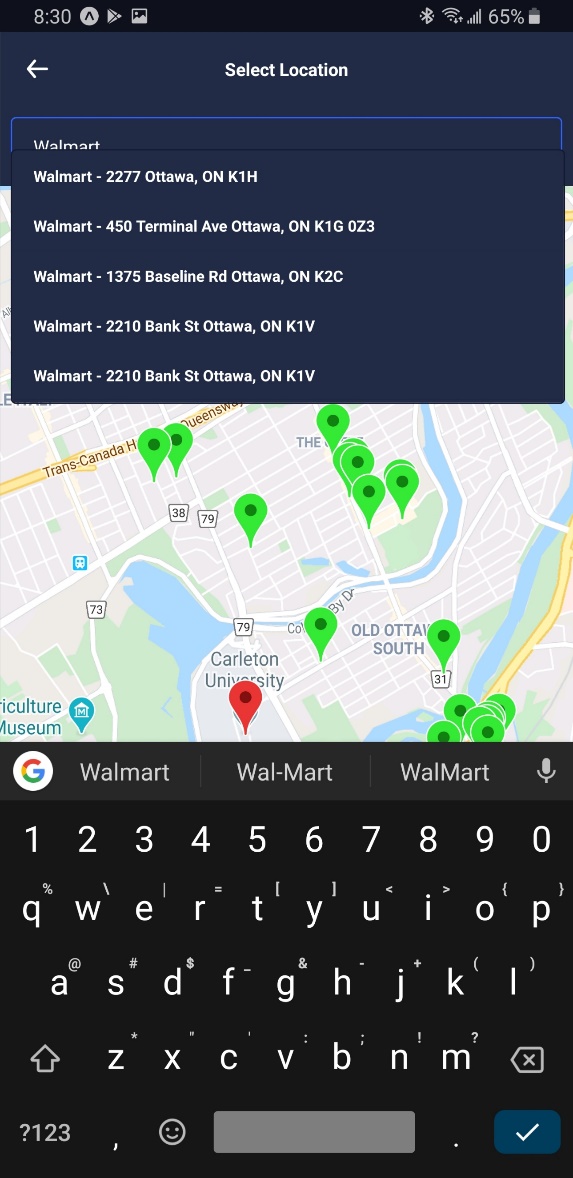


Figure 20: Maps Page On Android With Search Results Provided By Here Maps API

### Contacts

Contacts can be considered a secondary feature of the application, where its value lies in unlocking the potential of the other core features. Users can add others to their contacts by using the email that they had registered on the application with. Users can also add said contacts to one or multiple groups. Groups are used to help organize contacts and select multiple contacts at once. By selecting on the group, rather than the contact, the user can share with multiple users in a press rather than having to select each contact individually. These contacts allow the user to share lists and recipes that others may be interested in.

### Sharing Lists and Notifications

One of the core values of the application is cooperation amongst users in order to enhance shopping experiences and save time. Sharing lists encourages this by leveraging the capabilities of the real-time database. The ability to share lists allows multiple users to modify the list and its details. Currently, lists can only be shared with direct contacts. This was done to discourage spam and the accidental sharing of lists with the wrong person. A user does not need to be a direct contact with each member of the list to use the sharing feature. If one of the members of the list is a direct contact, the sharing features work for every member of the list.

Notifications was a functionality added to help increase the ability of users to communicate with one another. We allow users to share messages with everyone who the list is shared with via Expo Push Notifications. The way Expo Push notifications work is when a user sends an Expo push token, it goes directly to the Expo backend. If the receiving user has an Android, the backend uses Firebase Cloud Messaging to send the user a notification. On the other hand, if the receiving user uses an iOS/iPadOS device, it uses APNS (Apple Push Notification Service) to send the notification.

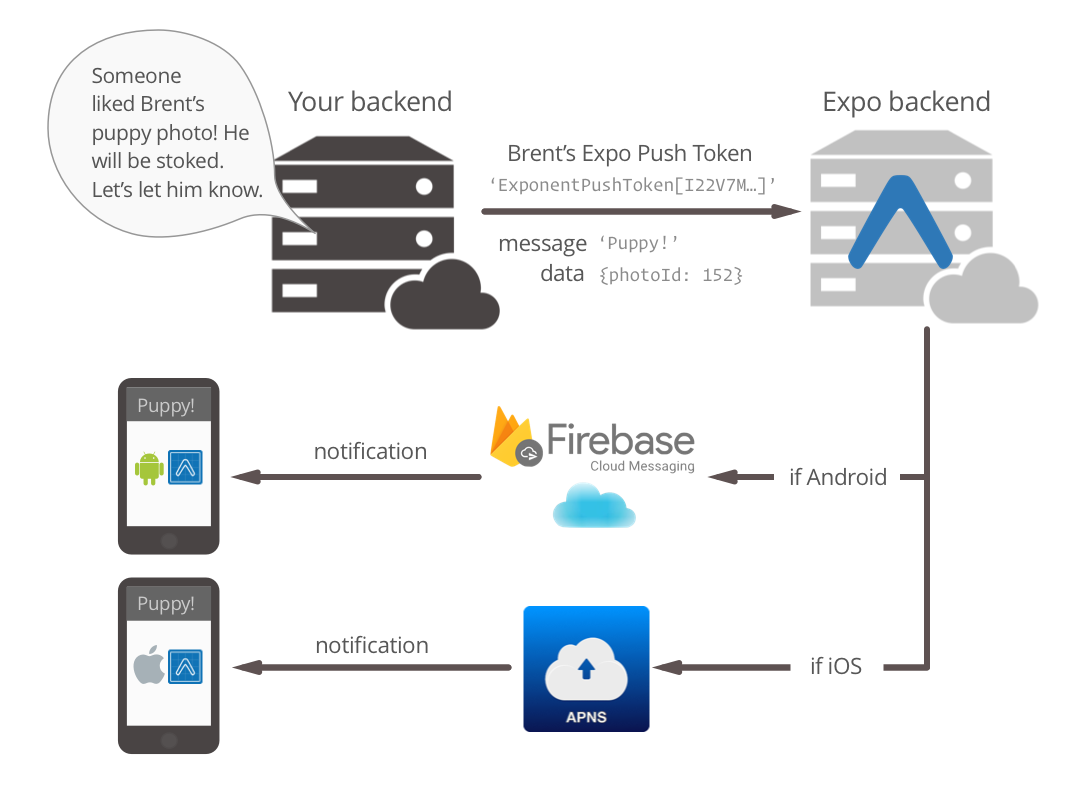


Figure 21: Summary Of Notification Use [9]

The addition of notifications allows the user to relay a message to the entire group of people who share the list (whether they are in your contacts or not). Sending reminders or having discussions across multi-platforms helps users communicate no matter the device or relationship between the users, without forcing the user to leave the application.

### Recipes

Recipes functionality was added as a luxurious feature that would help separate our application from its competition. The ability to favourite recipes, share them with those in your contact list, and import ingredients from said recipes into a list of your choice.

There were multiples factors that would go into our decision of what source we wanted to use for the different recipes. The first decision to make was whether we wanted to scrape data from reliable websites such as Food Network [10] or use an external API to obtain recipe data. We decided to use an API, since the use of a web scraper would be difficult to maintain given the constant changes of websites. Another problem with using the web scraper is that it would be a lot more resource dependent given that it must obtain the HTML of the webpage and then parse it looking for specific items. As for our choice of API, there were many options, such as Recipe Puppy and Big Oven. The factors that would influence our decision were: number of API calls allowed (for free), number of available recipes, and option to obtain the specific ingredients for the recipe. The reason we chose Spoonacular [11] was because it gave us the most API calls for free, and also provided a link to their website which hosted the recipe which provided details such as nutritional info, expected price, and tips and tricks to cook the meal.

Now that we knew what source of data we would use, we needed to determine how we would distribute these recipes to the users. We knew we had to save the recipes in our database given that we were only allowed 40 calls/day [12] to the API. Therefore, we would update the database with 30 new random recipes once a day. We would then store a list of 30 random recipes from our database updated daily within the cloud functions. The reason we did this rather than randomly selecting 30 from our entire table of recipes was because as table grew, the time it would take to select 30 random recipes would grow, decreasing the performance of the application. Since a fast user interface and smooth experience was one of our highest priorities, we decided to store 30 random daily recipes via a cloud function in a different table where it would be easy to obtain.

## Back-End Features

There are two primary services running in the backend to support the app. The first backend component is a database that is used to store all needed information, including user information, store maps, and item details. The second component is a Machine Learning Toolkit used in order to fill in gaps of missing data and improve the accuracy and functionality of the application features, such as list rearranging, as mentioned in *3.2.2 Python Module Used for Machine Learning Capabilities.*

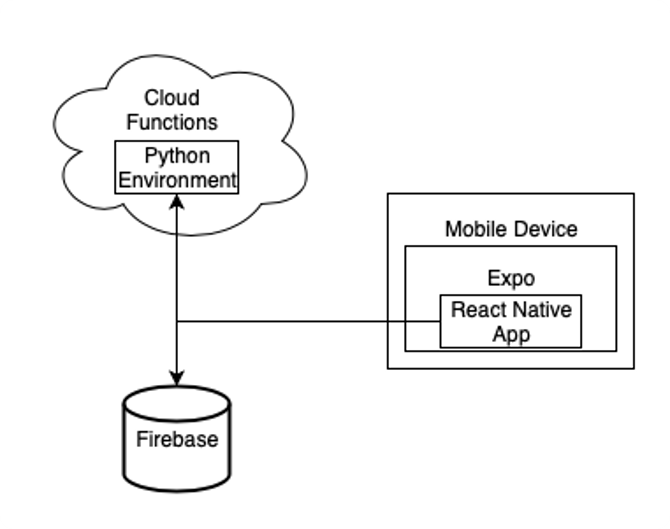


Figure 22: Summary Of Architecture

### Managing the Database Using Google Firebase

The first backend component of the application is a database that is used to store user information, the crowd-sourced item information, and grocery store layouts. This database uses a NoSQL implementation. NoSQL implementations require flatter database implementations when compared to SQL databases. This means there needs to be some redundant information to allow for efficient querying of the database. NoSQL meets the requirements of the project because its use of unstructured data allows a dynamic schema that can be horizontally scaled to meet the complex and constantly changing data needs of the application. Also, storing and processing of data in real-time is more efficient in NoSQL databases when compared to SQL databases.

There are eleven tables in the database: dailyRecipes, favRecipes, recipes, globals, items, stores, mapClusters, recommendations, contacts, userInfo, users, lists. The first three tables are used to store information related to the recipes API, the globals table is used to store information used to monitor the state of the database, the items and stores tables hold crowd-sourced information, the mapClusters and recommendations tables hold the information generated by the machine learning suite, and the final four tables are used for user information.

The first user table is a User Info Table. This table contains each user’s login information and user id in encrypted format using Google’s default encryption method. The second table is a User table. This table contains the rest of the user information, including user preferences and the sets of grocery lists that the user has access to edit and delete. The third user table is the Contacts table. This table details user’s contacts, which are other users that the user frequently shares list with. The final user table is the Grocery Lists Table. This table is a list of all created grocery lists, including the information about each item, such as the quantity of the items and whether it has been retrieved yet.

The first crowd-sourced information table is the Stores Table. This table maintains information about each store, including the identification used for each store, the layout of the store if known, and the locations of each known item in the store. The other information table is the Items Table. This table contains the information about each item, including the name of the item, features of the item, and any known locations of the item. This format will allow the database to be effectively used by the application.

There are two tables to hold the information generated by the machine learning suite. The first is the mapClusters table. This table stores all of the saved map clusters generated through the clustering algorithm. The other table is the recommendations table. This table saves all of the rules used for recommending items. It also saves a list of the most popular items. The method used to populate these tables is discussed in section *6.2 ML Algorithms*.

There are three tables used to handle interaction with the Recipe API. The first is the recipes table. This table stores all of the saved recipes along with their ingredients and other related information returned by the Spoonacular API. The second table is the dailyRecipes table. This table stores the recipes to recommend to users on a given day. This table is updated daily to contain that days recipes. The final table is the favRecipes table. This table keeps track of which recipes each user has favourited.

The final table in the database is the globals table. This includes general information about the database and is used by the app to maintain the database. It contains counters, flags, and keys needed by the application.

### Python and JavaScript Module Used for Machine Learning Capabilities

The final back end feature is a machine learning module to support the user. This module is implemented using a combination of JavaScript and Python and its pre-existing libraries. The machine learning modules support three major features: predicting item locations, sorting grocery lists based on the store layout, and the suggestion of items based on existing user purchasing patterns.

The first feature, predicting item locations for items with unknown locations, uses a method called collaborative filtering. If the location of an item is unknown, the application compares the store the user is currently shopping in to similar stores based on the contents of the stores. Then, by determining the majority location of the item in each of the similar stores, the location can be predicted.

The second feature offered by the machine learning module, sorting the grocery lists, uses the crowd-sourced information to generate the optimum list order based on the crowd-sourced store layout. The algorithm takes the weighted average of all known maps to compute the fastest path. The weights for each map are maintained by prompting the user to verify whether or not the generated map is correct.

The final feature, suggesting items to users based on purchasing patterns, uses a method called association rule mining. As the database is updated with new lists, the machine learning module scrapes the database looking for patterns. These patterns are used to form rules dictating items that are frequently purchased alongside one other. As users add items to the database and coincidentally begin following a rule, the application will suggest other items using the rules. The rules are built using purchasing patterns from all users. The algorithm can be expanded to be based off of the frequency with which the user purchases items to tailor the recommendation to each specific user.

For an expanded discussion on the machine learning capabilities, please see section *6.2 ML Algorithms.*

## Functional and Non-Functional Requirements

### Functional Requirements

1. Users wants to create an account.
2. User Viewpoint
3. The system shall allow the user to create an account using their name, email, and a password.
4. The user shall verify their email by following a link sent to the entered email account.
5. Google Firebase Viewpoint
6. Firebase shall send an email to the user to prompt them to verify their email.
7. Firebase shall save the login information and shall assign the user a user ID.
8. App Viewpoint
9. The app shall navigate the user to the login page so they can login to the application.
10. User wants to add a new contact.
11. User Viewpoint
12. The User shall be prompted to enter the email of the other User they want to send the contact request too.
13. The App shall allow the user to optionally enter a name for the contact and specify a group to add the contact too.
14. The User that receives the request will receive a notification stating they have a pending contact request.
15. The User that receives the request shall be given the option to accept or deny the request.
16. Google Firebase Viewpoint
17. Firebase shall retrieve the token used to link a profile to a phone and return it to the app so that the notification can be sent.
18. App Viewpoint
19. The app shall query firebase for the device token of the target user.
20. The app shall send the target user a contact request notification.
21. User wants to rearrange their list.
22. User Viewpoint
23. The User shall specify which criteria they want to use to rearrange their list using a dropdown menu on the list screen.
24. Should the user select a criterion which requires the app to know the location of the user (By Location, Fastest Path, Fastest Path (Auto Update)), the user shall enter their location using the autocomplete input box or Here Maps API.
25. Google Firebase Viewpoint

NOTE: The following shall only occur when a cloud function (machine learning and database data) is required to perform the reorganization, so this only occurs for By Location, Fastest Path, and Fastest Path (Auto Update)

1. Firebase shall call the appropriate cloud function.
2. The cloud function shall determine the layout and item locations of the store the user has specified, estimating information as needed.
3. The cloud function shall reorganize the given list using the appropriate criterion.
4. App Viewpoint
5. The app shall trigger the reorganization function corresponding to the selection the user made.
6. The app shall display a loading icon to the user while it waits for the reorganization to finish and then shall update the list once the reorganization is finished.
7. Should the user select a criterion which requires the app to know the location of the user (By Location, Fastest Path, Fastest Path (Auto Update)), the app shall navigate the user to the store selection screen and prompt the user to enter their store.
8. User wants to add items from a recipe.
9. User Viewpoint
10. The User shall navigate to the recipe they want to save and tap the button to save the items.
11. The User shall specify the list they want to add the items to or shall create a new list.
12. Google Firebase Viewpoint
13. Firebase shall retrieve the items in the given recipe and return them to the App.
14. App Viewpoint
15. The App shall create the new list if necessary.
16. The App shall retrieve the items for the recipe from the database.
17. The App shall save all of the items to the selected list.
18. User wants to save crowd-source information.
19. User Viewpoint
20. The User shall navigate to the screen corresponding to the type of information they want to save (item, item location, or store layout).
21. The App shall prompt the user to enter all of the needed information.
22. Google Firebase Viewpoint
23. Firebase shall save the given information to the appropriate table.
24. App Viewpoint
25. The App shall ensure that all needed information is entered by the user.
26. The App shall send the entered information to the database.

### Non-Functional Requirements

#### Look and Feel Requirements

1. The app shall have an interface that is consistent across all android and iOS devices.
2. The app shall have a dark theme and light theme and allow the user to select which theme they wish to use.

#### Usability and Humanity Requirements

1. The app shall have a simple interface that is easy to use.

#### Performance Requirements

1. The app shall maintain separate views of a list in real-time.
2. The app shall reorganize lists accurately to the department level at minimum and shall reorganize to the aisle level when the information is available.
3. The app shall support 100 concurrent users.

#### Operational and Environmental Requirements

1. The app shall run as an application on the user’s mobile device.
2. The app shall run on Android and iOS devices.
3. The app shall use the application programming interface (API) provided by Google firebase for interacting with the database and running cloud functions.
4. The app shall be hosted on the Expo app.
5. The app shall be distributed on the Google Play Store and Apple App Store. (Currently, the app is only available through Expo, but if it is to be widely distributed, it will be available on both store)

#### Maintainability and Support Requirements

1. The app shall be open-sourced so that it can be maintained over time. (Currently not open-sourced, but it can be made open-sourced)

#### Security Requirements

1. The features in the app shall only be accessible to users with email-verified accounts.
2. The app shall contain an extensive Privacy Policy.
3. Users shall be notified at least 90 days in advance should the Privacy Policy be changed.
4. User information shall not be made available to any third-party sources.
5. Users shall be allowed to delete their accounts and change their passwords.

#### Legal Requirements

1. The app shall comply with Canada’s Personal Information Protection and Electronic Documents Act when dealing with data breaches and Privacy Policy changes.

# Development Methods

The development methodology that was chosen for the project was Agile development. This development process provides the greatest opportunity for meeting the goals and deadlines in an organized manner. Agile development involves collaborating with a group of individuals in order to meet similar goals, following the twelve principles behind the agile manifesto. Some of the core principles proving to be the most important towards the success of the project include [17]:

* Setting customer satisfaction as the highest priority
* Welcoming and adapting to changing requirements
* Delivering working software frequently
* Paying attention to technical and design excellence
* Focusing on simplicity

Customer satisfaction through simplicity as well as technical and design excellence is vital to the success of this project because consumers prefer to use applications which are simple and easy-to-use, yet functionally powerful. To measure customer satisfaction, users trusted by the developers will test the app to ensure it is intuitive to use and that the features are helpful. Adaptation to changing requirements and delivering working software frequently is important because it ensures that tasks are broken down into smaller chunks, which can iteratively be built upon. This is important because it encourages simplicity and allows the team to effectively test features without being overwhelmed by large and complex pieces of code.

To keep track of the project, a Kanban board is being used. Kanban boards are sectioned project boards used to monitor the project elements to be developed. Kanban boards generally have several sections used to track the project including in progress elements, completed elements, and backlog elements. Elements are moved throughout the sections as they are completed to keep track of the project. Kanban boards are a popular tracking method for Agile Development and are helpful for monitoring the state of the project. In order to simplify each iteration of the development process, the project was broken into three main categories: User Interface, Cloud Application Program Interface (API), and Machine Learning.

## React Native for Application Development

To develop the application, React Native was chosen due to its cross-platform availability, and its specialty in mobile application features that are important to the project [18]. React Native uses JavaScript, ready-to-apply elements, and has a large range of libraries to shorten the development time considerably. On the other hand, use of some of these ready-to-apply elements slightly reduces the speed of the applications because they are not fine-tuned for a specific operating system.

Other options, such as Ionic and Flutter, were also considered as viable options but were decided against due to certain limitations in each. Ionic also uses JavaScript and ready-to-apply elements, but it lacks many libraries that can help create a more enjoyable user experience. Additionally, use of Angular makes the learning curve steeper in order to perform more complicated operations [18]. Flutter is one of the newer frameworks developed by Google, meaning its lack of libraries and community support would bottleneck the development process [18].

The reason React Native best suits the needs outlined above is because of its large community support. A beautiful and interpretable user interface with intuitive functionality is of the utmost importance for the application. Community support for common issues, along with many third-party modules which provide useful functionality, can be found for React Native. These resources provide the greatest opportunity to meet the user quality goals as efficiently and effectively as possible.

### UI Kitten for User Interface

During the initial phase of development, it was identified that there were several inconsistencies among the user interface design across platforms. In order to have consistency in user interface elements and properties such as buttons, dropdown options, backgrounds colours, and input fields, it was decided that a user interface package should be used. Several alternatives were investigated to find a user interface framework that was available for react native with a beautiful design, active community, and most important of all, a free and fully open-sourced library. The chosen library was ‘react-native-ui-kitten’, as this was the library that satisfies all of the above requirements for this project [19]. This framework also provided excellent documentation for all of its components and it also had examples on how to use each feature [20].

## Google Firebase for Database Management

In order to maintain the database, the team decided to use Google Firebase. Firebase offers a free tier that gives developers access to a robust suite of tools, including information storing, user authentication, and server-side machine learning and information processing with the help of Tensor Flow. Firebase also provides detailed API documentation along with generous limits in the free tier, followed by a subscription model for higher limits [21]. Firebase uses a NoSQL database implementation. The downside to using a NoSQL database is the fact that it is a non-relational database, so it lacks the concept of primary and secondary keys. This means that redundant information needs to be stored in order to keep the list flat. The benefit of this implementation is that it scales horizontally when adding data, and it is designed to handle lots of reads of the data, which the application will often be doing.

Alternative cloud platforms such as Amazon Web Services (AWS) brought forth many features that would have been useful for the project. These included a free tier functionality, but it required payment for the use of its Machine Learning module, and its free tier was not as generous as Firebase’s free tier. For these reasons, along with the fact that Firebase had everything the project will require, Google Firebase was the database of choice.

## Python for Machine Learning Module

Finally, for the machine learning aspect, Python was chosen due to its robust existing machine learning libraries and optimized components. Python is widely considered one of the best programming languages for machine learning due to two reasons. The first is its large community invested in creating libraries specifically for machine learning [22]. The existing machine learning library base can be expanded on for the purposes of the application, both in the approximation of item locations and the formulation of association rules. The second reasons is because of its highly optimized components. Python allows for methods to be implemented in C. Since C is a low level language, the methods can be highly optimized. These methods are then compiled and placed in Python languages for reuse. This functionality makes Python a very strong language for machine learning. For example, Python’s dictionaries and maps are very useful. Furthermore, Python is regularly used to implement the backend of mobile applications, meaning community support and other resources can be used throughout the project. Finally, all members of the group already have some experience using Python through schoolwork and external projects, so no time was needed to learn Python syntax.

## Validation, Verification and Testing

Developing features at a fast pace while maintaining minimal bugs was a high priority during this project. There were a series of steps and practices that were being followed to limit the number of bugs.

Firstly, features were broken down into small, incremental development steps. Not only does this allow the developer to focus on a small section of code, but it prevents any potential distraction or fatigue from working on a complex feature spanning multiple weeks. The constant cycle of picking up a new task, quickly and efficiently completing it, and then picking up a new task helps keep the developer’s minds sharp and provides each team member with a fresh perspective on the section of code they are working on.

Next, the use of small, incremental development made the process of testing and code reviews a lot faster and simpler. Before a developer can merge into the ‘master’ branch of the application repository, their code changes needed be approved by another developer. Typically, this process included looking through the changes in code and then testing the changes on multiple platforms (iOS, Android, etc.). A smaller code review makes it easier to detect potential bugs when reading through the code and trying to find different scenarios which may cause bugs or issues down the line. Another benefit of a smaller code review is that they were done more frequently, allowing each developer to reflect upon and improve their coding practices and standards, leading to cleaner code, which is more likely bug-free. Incremental development also made testing more manageable because it allowed developers to focus on a specific portion of the application, making it easier to detect different inputs, which can cause the app to misbehave.

One method of making sure the application was working correctly and as intended throughout each iteration was by having unit-tests. Several testing frameworks were examined for this project. The first was unit testing with Jest. Jest is a simple testing framework for JavaScript [49], specializing in simplicity, excellent API documentation, and its compatibility with React Native applications. Jest provides snapshot testing, which is a testing method that ensures that the user interface does not change unexpectedly [50]. Snapshot testing is done by taking a snapshot of a screen and comparing it to an expected snapshot. Jest can also be used to test Firebase cloud functions in order to make sure the functions are behaving as expected. The issue with using Jest was that it is incompatible with UI Kitten. This meant that errors would occur when initializing tests, so they would not run. Furthermore, most of the functionality being tested was related to interacting with a database. These functions require a database key and due to how this key must be initialized, it made these functions difficult to test with Jest.

Another testing framework that was investigated was Detox. Detox is a framework for creating cross platform End-to-End tests [51]. The issue with using Detox is that it requires the app to be deployed separately to iOS and Android before the tests could be configured and run. This increases the number of dependencies that needs to be maintained during development and greatly increases development time, which makes it unideal for Agile development. The final testing framework investigated was Appium. Appium is an open-source testing tool that allows for cross-platform testing that allows for automated testing [52]. The issue with Appium is that it requires the code to be deployed to a simulator to test every time. This meant that it needed to be compiled into an APK or run in an iOS simulator to use. As the developers were using multiple platforms to develop, this made Appium incompatible with the development strategy. Additionally, it had the same deployment issues as Detox. In the end, it was decided that small unit testing of the code along with manual testing was the most appropriate way to test the application.

Lastly, since user satisfaction is a high priority of the project, constant feedback from various users allowed the developers to reflect upon and improve the application, especially the user interface. Feedback from the Project Supervisor, friends, and family, amongst others, provided different perspectives and suggestions to consider during the development of the app to not only make it accessible, but also easy to use and understand for a wide range of demographics.

# Crowd Sourcing and Machine Learning

## Data Collection

### Initial Data Collection

The first step in designing the machine learning modules was to gather a large amount of data to use for training, verification, and testing. The final goal for the project was to be able to crowd-source the needed information. Crowdsourcing requires a robust UI that users can use for adding data to the central database. As shown in the Gantt Chart in *Section 8 Project Timetables* below, the app followed an iterative development pattern, so the UI needed for gathering the data was not completed until later in the development timeline. Due to this delay, it was necessary to collect the needed data manually. Then, once the working version of the app was completed, the data collection strategy was swapped to the crowd-sourcing method.

There are several common crowd-sourcing issues that needed to be addressed [23]. The first issue was the complex architecture required to handle large amounts of crowd-sourced data. To address this issue, a modular design strategy was used to scale the architecture. The architecture was built around a small amount of data initially, allowing for it to be verifiable that the architecture was consistent and properly scalable. Another common issue with crowdsourcing was making the functionality too complex on the user end. This makes it difficult for the user to input data. To counteract this problem, user feedback was used to develop the crowd-sourcing pages. A small set of beta testers were used to ensure that the pages were easy to use, and the goal of the pages was obvious to the user.

To manually gather the data, the developers visited stores and added information to a shared information file using a Microsoft Excel Sheet. To meet the requirements for machine learning, several sets of information were needed:

* Store Information:
  + Name
  + Address
  + Franchise Name
  + Layout
    - Department layout
    - Aisle layout
* Item Information:
  + Name
    - General name (Ex. Ketchup)
    - Brand name (Ex. Heinz)
  + Location
    - Store name
    - Department name
    - Aisle number
  + Price
  + Size and/or quantity if applicable (Ex. 2L of Coke, 24 cans of Coke)
  + Any other data that may be needed to differentiate it from similar items such as colour

The above list included all of the information elements that were needed to run the complete suite of machine learning functions. Each component of the suite required different subsets of the data. For example, the reorganization of items required the store layout and item locations, whereas the recommendation of items only required the item names. According to the Gantt Chart in *Section 8 Project Timetables* and planned iterations, it can be seen that not all of this information was needed for iteration one. For example, the price of the item was not be used until iteration three. The reason that this data was gathered earlier than it was to be used was so that the items do not have missing data in the future. The alternative would be to regather the data for these items or to use data imputation techniques to fill in the missing data, though this may be impossible with certain features, such as the address. With this data collected in an Excel file, the Pandas Python library [24] was then used to read the data and create the needed dataset.

The bulk of the difficulty with the manual data collection step was in gathering enough data. The first feature of the machine learning to be implemented is the estimation of item location using collaborative filtering. There is no specific minimum amount of data needed for machine learning. For example, some data scientists recommend at least 50 samples [25], whereas others suggest using heuristics, such as ten items per class [26]. The amount of data that can be gathered was limited by both the time available for data gathering and the variety of grocery stores available. As such, the plan for gathering essential data was as follows. Each developer was to visit three different grocery stores to gather item data and map the store. This gave a total of nine stores worth of information that was used as a base. The items being identified were made up of fifty items. The fifty items included a set list of thirty of the most popular items purchased by shoppers and an additional twenty flex items that were used to gather data for less popular departments. To view the full list of items that were recorded in each store, please view Appendix I.

The majority of the information gathered was either in the form of strings or numbers. The only exception was the maps of the stores. The maps of the stores were saved as an ordered list where the ordering was based on the order that the departments appeared in the store. Departments closest to the entrance appeared near the top of the list, while departments farthest from the entrance were last in the list.

### Maintaining the data in the database

The final goal of the project is to be able to crowd-source the information needed for machine learning. Individual screens were created to allow users to enter the relevant information: one for registering items in the database, one for adding item locations to the database, and one for mapping out stores. There were two significant hurdles to crowd-sourcing the information. The first was to have a UI that is easy and comfortable to use. To address this issue, external testers were used to ensure that the UI was intuitive. The second issue with the crowdsourcing was how to address fraudulent, incorrect, and/or unreliable information. This issue will be addressed in the machine learning module itself. The module will prompt users to verify existing information or generated information so that the data stored in the database can be verified.

In addition to crowdsourcing through screens to add information, the algorithm will also be trained via user support. When the algorithm reorganizes a list to place items in the optimum order, the user may be prompted with confirmations of store information as mentioned above and a simple user interface to reorder their shopping list based on the optimal path to items in that store. Then, by applying the feedback from the user, the algorithm can be improved to address issues. This discussion is expanded below in section *6.2.2 Weighted Ranking (Sort list of items to follow the shortest path and Estimate layout of the unknown stores).*

## ML Algorithms

The machine learning module is used for several features:

* Estimate the location of items
* Estimate the layout of unknown stores
* Sort list of items to follow the shortest path
* Recommend items to a user
  + Based off of similar purchasing patterns to other users
  + Based off of purchase history (Ex. Item has not been purchased in a while; frequency of re-ordering)

### Collaborative Filtering (Estimate location of items)

To estimate the location of items, a machine learning algorithm called collaborative filtering is being used. Collaborative filtering is a method often used in recommender systems to recommend items to users based on shared prior opinions [27]. Collaborative filtering generally follows two steps. The first step is to find the users who are most similar to a given user by comparing rating patterns. Then, use the history of those users to estimate how the given user will rate a new object.

In the example of predicting item locations based on the locations in other stores, the following steps would be used. In this example, a user is in the store ACTIVE\_STORE, and they want to find item UNKNOWN\_ITEM, but the location of UNKNOWN\_ITEM in ACTIVE\_STORE is unknown. The only information known about ACTIVE\_STORE is the location of some items, NUM\_ITEMS. Additionally, there is a database containing X stores, each with some known item locations. Refer to Table 3:

Table 3: Example of Store Database

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Store Name** | **Item 1 Location** | **UNKNOWN\_ITEM** | **Item 3 Location** | **Item 4 Location** |
| **ACTIVE\_STORE** | DEPARTMENT\_1 | -- | DEPARTMENT\_3 | DEPARTMENT\_5 |
| **STORE\_1** | -- | DEPARTMENT\_1 | DEPARTMENT\_4 | -- |
| **STORE\_2** | DEPARTMENT\_1 | DEPARTMENT\_6 | DEPARTMENT\_3 | DEPARTMENT\_4 |
| **STORE\_3** | DEPARTMENT\_2 | -- | -- | DEPARTMENT\_7 |

To estimate the location of UNKNOWN\_ITEM, the algorithm would start by checking every other store in the database to find the stores that are most similar to ACTIVE\_STORE. One way of determining similarity would be to check how many items in the store match the location of the item in ACTIVE\_STORE for each item in NUM\_ITEMS. For instance, if each store is assigned a score starting at 0, and then the score was increased by 1 if the locations match, decreased by 1 if the locations are different, and unchanged if the location is unknown, then the scores for each store in the table above would be as follows: STORE\_1 has a score of 0, STORE\_2 has a score of 1, STORE\_3 has a score of -2. After the similarities have been determined, each store is assigned a weight based on the similarity metric. These weights are then used to calculate a weighted average based on the known information to determine an estimate. In this case, the location would most likely be DEPARTMENT\_6 as STORE\_2 has the highest weight.

One problem in using collaborative filtering, in this case, is that collaborative filtering is generally used to calculate a weighted average based on a continuous value. For example, a score between 1.0 and 10.0. Whereas, in this case, it is being used to classify locations based on discrete, non-continuous values. To address the issue, the collaborative filtering will instead be used to determine the most similar stores, and then the most similar NUM\_STORES will be used to perform a majority calculation to get the most likely location of items.

The main issue to address is how to similarity between stores is calculated. As stated previously, it was desired to compare stores based on the locations of shared items, i.e. is item A in the same location in both stores being compared. To perform this calculation, the two stores are compared based on their layouts. The similarity algorithm follows the following steps, where S1 is the store the user is in and S2 is the secondary store:

1. Determine the set of common departments between S1 and S2
2. Determine the sets of unique departments for S1 and S2
3. Calculate the distance between the first common department for both stores using the modified Levenshtein Distance as described below
4. Repeat step 3 for each common department
5. Calculate the distance between the first unique department for S1 and an empty department
6. Repeat step 5 for each department unique to S1
7. Calculate the distance between the first unique department for S2 and an empty department
8. Repeat step 5 for each department unique to S2
9. Sum up all of the distances to get the total distance

The above process is repeated for each candidate store, where a store becomes a candidate store if it contains the desired item. The above algorithm can go into further granularity by performing the same calculation on an aisle level to compute the total distance for each department.

Currently, the distance metric is only considered with a department level granularity. This could be expanded in the future to go to an aisle level granularity. The reason that the algorithm does not currently go to an aisle level is because it was found that going to that level greatly increases the computational overhead without giving much increase in the reordered list. When comparing two stores, each individual department needs to be compared. When going to the aisle level, the computation time and memory use greatly increases. This is acceptable when there is a small amount of data for each store, but as the amount of data increases, this could quickly become unacceptable.

For the distance calculation, a modified version of the Levenshtein Distance is used. The Levenshtein Distance is generally used to calculate the distance between two words or sentences [28]. This distance metric calculates the distance by counting how many letters need to be removed, added, or changed to convert one word/sentence to another. For example, the words “tomato” and “potato” would have a distance of 2 because the first “t” would need to be changed to a “p” and the “m” would need to be changed to a “t”. These two words are very similar, so they have a small distance, but if “lettuce” was used instead of “potato”, the distance would be much higher at a value of 8, as many more changes need to be made (add “le”, remove “oma”, change “o” to “u”, and add “ce”). To modify this metric for use in comparing departments, the words were substituted with departments, and the letters substituted with items.

A few extra changes were made to the metric to better fit the crowd-sourcing nature of the application. As crowdsourcing is the medium used to gather data, it can be assumed that the calculation will regularly deal with missing or extra data. For example, in Store S1, it may be known that the Produce department has Apples, while for Store S2, it may be known that the same department has Apples, Bananas, and Oranges. The Levenshtein distance punishes extra and missing data, as it assumes it should not be there. If this were used, the metric would punish stores where a lot of data has been collected, and reward stores with sparse data. To fix this issue, the penalty metric will be changed. To handle extra items, the penalty will be added only if it is known that the extra information is incorrect. To continue the above example, if it is known that Bananas are in the Bakery in Store S1, then Store S2 would be penalized for having it in the Produce department, but if it is unknown, no penalty is added. To handle the case of missing data, the penalty will be based on the percentage of missing data. Normally, the penalty is 1 for each missing value. This will be changed to the number of missing items divided by the total number of items in the department of the store the user is in, unless it is known to be incorrect, in which case the maximum penalty will be used. Continuing the above example again, if the data was swapped between Store S1 and S2 and the location of bananas and oranges in S2 was unknown, then the penalty would be 2/3 for each missing item. This changed metric allows for stores with missing data to be penalized, without rewarding stores with sparse data.

Collaborative Filtering has several additional problems that need to be addressed [27]. One issue is data sparsity. This problem is caused by the fact that collaborative filtering cannot work until information about the store is gathered. If there are no known item locations in the store, then the similarity metric cannot be calculated. This problem can be addressed by updating the machine learning module to simply return the mode location of that item if nothing else is known. Additionally, this problem can be minimized by making it easy to add items to the database. The easier it is to add items, the more likely that information will be known, and the more likely collaborative filtering is going to be able to estimate the location.

Another issue is scalability. As stated above, collaborative filtering compares each item in each store. This can be a massive computing undertaking as the number of stores and items increases. This issue can be addressed by minimizing the amount of data being compared. One way of minimizing the data is to compare fewer stores. Stores of the same brand are more likely to have the same layouts, so by only comparing to stores of the same brand when possible, the algorithm can be sped up. For example, if a user is in a Wal-Mart, then the algorithm would only compare to other Wal-Marts. Another method would be to prioritize stores geographically closer together.

The third issue with collaborative filtering is synonyms. This refers to items that may have multiple different names. For example, French fries have a number of names such as fries, chips, and wedges. This issue will be difficult to address, but one way of solving it would be to save the synonyms for each item and then when that item is added to the database under one of its synonyms, it is instead saved to a single uniform name. Another method that can be used to address this problem would be to use a thesaurus API to convert item names to a common synonym.

There is one more problem that the application is very unlikely to encounter, the issue of Gray Sheep/Black Sheep. This issue refers to users who do not follow regular patterns or who regularly go against the grain. This issue is unlikely to affect the application as grocery stores tend to follow similar patterns, putting items in the same location to make it easier for customers to find items.

### Weighted Ranking (Sort list of items to follow the shortest path)

The goal of sorting the list of items to follow the shortest path is accomplished using a combination of crowdsourcing and weights calculated through backpropagation. The maps for stores are gathered through crowdsourcing using a list format, as discussed in section *4.1.4 Registering Store Layouts*. The list that users input is the order that the departments appear in the store, so these can be taken as the fastest path to visit all departments.

To sort the items in the fastest order, the list of items needs to be reorganized to match the fastest past as determined by the crowd-sourced data. To do this, the location of all the items in the list needs to be determined, either by retrieving the location from the database if the location is known or by estimating the location using the algorithm outlined in section *6.2.1 Collaborative Filtering (Estimate location of items)*. The items are then mapped to their corresponding departments and aisles. The departments and aisles are then reorganized to match the order determined by the crowd-sourced fastest path.

As the maps for the stores are crowd-sourced, there will most likely be multiple maps for each store, so machine learning is used to determine the average map used to retrieve the fastest path. The following algorithm is used to calculate the optimum path [29]:

1. Assign each department a score based on its position in the map. The first department will receive a score of N, where N is the number of departments, while the last department will receive a score of 1.
2. Multiply each score by that map’s weight
3. Repeat for each map in the store
4. Sum the weight-multiplied scores for each department
5. Sort the departments based on their summed values from highest to lowest

The weights will be calculated and updated based on user feedback. When a user uses the fastest-path organization feature, they will be prompted after shopping to ask if the given path was the fastest. If the path was not the fastest, the user can then reorganize the calculated path to the one they believe was the fastest. This information will then be backpropagated to update the weights of each list. Lists that were similar to the actual fastest path will have their weights increased, while lists that were dissimilar from the actual fastest path will have their weights decreased. The amount that the weights are increased or decreased will be based on how similar the two lists were. The similarity will be measured based on the average distance that departments were from their actual rankings. For example, if the actual order had the Produce in second place and the map in question had it in fourth place, this would be a distance of 2.

The weights for each map are updated based on user feedback. When a user uses any of the location-based sorting methods that require a store map to be used (Fastest Path or By Location), they have the option to view the map used to perform the sorting. They can then change this map by adding, removing, and reordering departments in the list. Saving the map then triggers a cloud function that updates the weights. The weights are then updated based on how similar the corresponding map is to the map the user entered. If the are similar, then the weight will increase, and if they are dissimilar, the weight will decrease. To calculate the similarity, the Euclidean distance metric introduced in section *6.2.3 Map Clustering (Estimate layout of the unknown store)* is used. The weight is then updated by keeping a running average of the weights based on user’s input.

### Map Clustering (Estimate layout of the unknown store)

The main use case for the application is the capability for a user to enter a store they have never been in before and rearrange their grocery list to quickly find the items they need. To handle items with unknown locations, the collaborative filtering can be used to predict locations. For the collobartive filtering to work, a map must be known, but in some cases, the map may be unknown.

There are several methods that can be used to address this issue. The first is to have the user map out the store before the app will rearrange the list. This strategy has the highest likelihood of determining the correct layout of the store, but is inconvenient for the user and defeats the purpose of the application. The second strategy is to take the average of known lists. This can either be a complete average of all maps in the database or a weighted average based on some known information about the store. This strategy requires very little intervention from the user, but is very unlikely to produce proper results. The final strategy was to use map clustering. Clustering is a simple strategy where similar items are grouped together into clusters that define regular expected behavior. This strategy takes the pros and cons of the previously introduced strategies. It requires some intervention from the user to determine which cluster is the correct choice, but it is more likely to produce a correct map than a complete average. Clustering was the first choice for this implementation as it is the most appropriate.

The first step in clustering is determining if it can actually be used. Clustering can only be used if clusters can be formed from the collected data, meaning that datapoints must be similar to each other. To form clusters a distance metric and maximum threshold must be chosen. The distance metric determines the similarity between two items. The maximum threshold determines how close to items need to be to each other to be clustered together. As stated previously, the map layouts for the stores are kept in an ordered list where the order is based on the appearance order of the departments in the store. There are several popular distance metrics for ordered lists. The distance metric chosen was Euclidean distance, as it is the simplest to use. The Euclidean distance metric is commonly used for calculating the distance between points by taking the square root of the summation of the difference between both points in all dimensions using the following equation:

Equation 1: Euclidean Distance [30]

Where,

p = The first point

q = The second point

i = The dimension

n = The number of dimensions

qi/pi = The value of q/p in the ith dimension

To adapt this equation to the application, point p and q were the indices of the departments in their corresponding maps. Since indices are one dimensional, n was set to 1. To calculate the distance between two stores, the distance between each common department in the two stores is calculated and summed. One assumption that was made when using this strategy was that unique departments in a store, departments in one map and not in the other, should not have an impact on the calculation. For example, if store 1 has a map of [A, B], while store 2 has a map of [A, C, B], they would have a distance of 0, as the common departments A and B are effectively next to each other. The reason that this assumption was made was due to the nature of the maps. The maps are the order that a customer would want to travel through the departments to visit all departments in a store in the fastest path. Returning to the example, this means that if a customer does not need to visit department C, they could simply go straight through department C, making the map essentially the same as [A, B]. Once the distance metric was calculated, the distance between each store was calculated as follows:

Table 4: Sample Distance Calculations Using Gathered Data

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | A | B | C | D | E | F | G | H |
| A |  | 28 | 9 | 20 | 11 | 17 | 14 | 45 |
| B | 28 |  | 35 | 24 | 49 | 38 | 31 | 36 |
| C | 9 | 35 |  | 27 | 11 | 28 | 18 | 42 |
| D | 20 | 24 | 27 |  | 42 | 23 | 14 | 32 |
| E | 11 | 49 | 11 | 42 |  | 38 | 31 | 68 |
| F | 17 | 38 | 28 | 23 | 38 |  | 12 | 54 |
| G | 14 | 31 | 18 | 14 | 31 | 12 |  | 45 |
| H | 45 | 36 | 42 | 32 | 68 | 54 | 45 |  |

In the above table, the most similar store(s) to a given store are highlighted in green, while the least similar store is highlighted in red. The stores in the above table are as follows:

Table 5: The Store Names Corresponding To Each Letter

|  |  |
| --- | --- |
| **Letter Value** | **Store Name – Address** |
| A | Independent - 2277 Riverside Dr |
| B | Walmart - 2277 Riverside Dr |
| C | Independent - 2681 Alta Vista Dr |
| D | Food Basic - 1670 Heron Rd |
| E | Loblaws - 2210 Bank St |
| F | Walmart - 2210 Bank St |
| G | Metro - 2515 Bank St |
| H | Whole Foods - 951 Bank St |

The next step after determining the distance metric and calculating the distances was to see if clusters could form. To do this, a script was developed that would slowly iterate a distance threshold until a desired amount of clusters was reached. If no distance threshold existed for the desired cluster count, it would get as close as possible. To started, the target cluster count was set to 3 as this is the square root of the number of stores rounded to the nearest integer. After running the script, it was found that no threshold existed that allowed for 3 clusters, but 4 clusters could be obtained. This revealed the following four clusters:

* Cluster 1:
  + Independent – 2277 Riverside Dr.
  + Independent – 2681 Alta Vista Dr.
  + Loblaws – 2210 Bank St.
* Cluster 2:
  + Food Basic – 1670 Heron Rd.
  + Walmart – 2210 Bank St.
  + Metro – 2515 Bank St.
* Cluster 3:
  + Walmart – 2277 Riverside Dr.
* Cluster 4:
  + Whole Foods – 951 Bank St.

These clusters showed that certain franchises, such as Independent/Loblaws, have very similar layouts across their stores, while some franchises, such as Walmart, have more variation in each store. Once the clusters were formed a map was calculated for each cluster using the weighted ranking calculation described in *Section 6.2.2 Weighted Ranking (Sort list of items to follow the shortest path).*

When a user visits a store with an unknown map, the app will prompt them to determine which cluster is the most appropriate to use for the given store.

The final step was to set trigger rules for when to reform the clusters. As the amount of stores increases and the known information about each store changes, these clusters will need to be reformed to ensure that the clusters are appropriate. A counter was added to the database that will iterate whenever a map is added, changed, or deleted in the database. When this counter passes a certain threshold, the clusters will be reformed and the counter will reset. This threshold is set to be an increase of 10% of the previous threshold. For example, if the previous threshold was 100, the next will be 110, then 121 and so on. This was done because as the amount of information increases, the clusters will need to be recalculated less often.

### Association Rule Mining (Recommend items to the user)

One of the advanced features in the application is the ability to recommend items to users based off of similar purchasing patterns to other users. This could be expanded in the future to be based off of the users purchasing history so that the recommendations are tailored to the user.

To implement this feature, a machine learning technique called Association Rule Mining was used. Association Rule Mining develops sets of rules that can be used to predict behavior [31]. For example, if Item C is purchased whenever both Item A and Item B are purchased together, then the rule {Item A, Item B} => Item C can be developed.

Association Rule Mining uses several metrics to determine rules. The two most important rules are called Support and Confidence [31]. The Support metric describes how often an item set appears in a database. It is calculated using the following:

Equation 2: Support Metric [31]

Where,

X is the desired itemset

T is the complete set of transactions

t is the set of the transaction containing X

The Confidence metric describes how often a given rule has been found to be true. The metric is calculated using the following:

Equation 3: Confidence Metric [31]

Where,

X and Y are the subsets dictated by the rule

supp() is the support function

Generally, to calculate the rules using Association Rule Mining, minimum values are determined for both a support threshold and the confidence threshold. These two values are hyperparameters that can be tuned to increase the likelihood of rules being accurate, though thresholds that are too high may lead to no rules being found.

There are multiple algorithms that can be used to implement association rule mining. One such algorithm is called the Apriori algorithm [32]. This algorithm follows a simple set of rules to identify frequency subsets in a list of transaction. First, a value is determined for the minimum support threshold. Next, the support for each separate item is calculated using the list of transactions. Then, for each item that meets the determined minimum threshold, repeat the process for pairs that include those elements. If an item does not meet the minimum support threshold, then any larger subsets, including this item, can be pruned. This process is repeated until the maximum possible set size is reached. These subsets can then be used to construct possible rules. This algorithm has a number of issues which make it unsuitable for the desired purpose. One issue is that it spawns too many subsets. The database in the case of this application will potentially be extremely large, meaning the algorithm may run into memory issues. Another problem with this algorithm is that it scans the database too many times. The database is going to be crowd-sourced, so it may be constantly changing, meaning the amount of rescans needs to be minimized.

The algorithm that was used in the application is called The FP-Growth Algorithm [33]. This algorithm works by first constructing a FP-Tree (Frequent-Pattern Tree) using elements that meet the minimum threshold and then uses this tree to determine the subsets that meet the minimum threshold. This algorithm is split into two parts. The first part is used to construct the FP-Tree using the following steps:

1. Assign a value to the minimum support threshold
2. Determine the support for each separate item in the transaction table
3. Sort the items by their support value in descending order
4. Sort the first transaction in the order determined in step 3
5. Insert the sorted transaction into the tree, incrementing values that are shared between transactions
6. Repeat step 4 and 5 for every transaction

Once the FP-Tree has been created, the second part of the algorithm, called FP-Growth, is used to determine the subsets that meet the minimum threshold. This part of the algorithm uses the following steps:

1. Select the first item in the order determined by step 3 above
2. Create a sub-tree of the FP-tree from part 1 by pruning any branch that contains that item, starting at the node that contains that item. Additionally, prune any branch that does not lead up to a node containing the item.
3. Set the new value of each node to the number of transactions in the sub-tree that include that item
4. If the sub-tree from step 2 has more than one branch, repeat step 2 and 3 using the created sub-tree and iterating each item to become a pair made of the values from the nodes that have reached the minimum support threshold
5. Repeat step 4 until each sub-tree only has one branch
6. Save each subset that has reached the minimum threshold value
7. Repeat steps 2 - 6 for each item

The FP-Growth algorithm offers several improvements over the Apriori algorithm. One improvement is that it only iterates over the database twice as opposed to the unpredictable amount from the Apriori algorithm. Additionally, this algorithm compresses the input database, so it requires less memory. To further decrease the needed amount of memory, the database can also be partitioned into several smaller databases before applying the algorithm. The FP-Growth algorithm was used for association learning. This algorithm could be modified in the future to use an improvement of the FPGrowth algorithm called the DynFP-Growth Algorithm [34]. This algorithm offers the large benefit of only having to scan the database once when the algorithm is run. When DynFP-Growth is used, it saves the created FP-Tree in the database and then updates this tree as needed. Then, when the algorithm is run, the database only needs to be scanned one time. This would be very helpful in the case of this application, as the database will be frequently updated.

There are several parameters that need to be addressed for applying association mining to the grocery list application. One parameter is when to run the algorithm. If the algorithm is run after every grocery list is created, then it will take a lot of processing time on the server, which may cause issues. On the other hand, if it is run very rarely, then rules may not be detected until a long time after they appear. To address this, a parameter was assigned to specify that the algorithm should be run after the number of lists in the database increases by 10%. This value was chosen so that the rate of developing new rules would be high in the beginning and then would slow down as the number of lists increased. Two other parameters are the support and confidence thresholds. The support threshold was set to 30%, meaning that a subset must appear in 30% of lists to become a rule. The confidence threshold was set to 80%, meaning that when a rule appears to be being followed, it must be valid 80% of the time. These values were chosen after experimenting with different thresholds during development. These values could be improved in the future. For the support metric, it would be more appropriate to be based off of the magnitude of the value as opposed to percent based (i.e. subset must be in at least 10 list, as opposed to 10% of lists). This would be more appropriate as a percentage based would cause items to be frequently ignored. Additionally, the confidence metric should be further experimented upon to determine the optimum value.

A final source of information that may help create useful rules would be to use snapshots of lists as they are being modified. Users often exhibited patterns when modifying lists that can be used to simplify the process of modification. For example, if a user decides they want to cook a specific meal, they would open their weekly grocery list and add all of the ingredients to their list. If the algorithm compares the new list and the original list, it will show items that may be more likely to be purchased together. This may also be the case when removing items from the list.

### Use of Cloud Functions

The Machine Learning algorithms chosen are relatively computationally heavy. This makes running the algorithms locally on the user’s phone unviable as it would be slow and would drain the phones battery. To address this issue, Cloud Functions can be used. Cloud Functions are a service offered by Google to be used alongside Firebase that allow developers to create functions that are implemented as Functions as a Service. To use this service, the developer writes the function locally, defines the triggers for that function, and then deploys the function to Firebase. The database then monitors all transactions to see if a cloud function has been triggered, and if so, it runs the function. This has the benefit that the function is run on Google servers, so they are running on better CPUs, or in some cases GPUs, making the function run faster. Additionally, since the functions are run alongside the database, the app does not need to make as many requests to the database to retrieve information, greatly reducing transaction count.

For the application, two trigger types were used: on write and on call. The ‘*on write’* trigger is used to run automatic functions. This trigger is set on a specific table or section of a table in the database using a path. When the portion of the database being monitored is created, updated, or deleted, then the function runs [35]. This trigger is useful for updating counters and running functions upon database updates. For example, in the database for this project the path “users/{uid}/lists/shared” is being monitored. This table contains the list IDs of all lists a used has shared with them. When this list is modified, a counter is updated to keep track of how many lists a user has shared with them. In this path, it can be seen that the layer called “uid” is in braces. This defines the “uid” as a wildcard so that it triggers for every valid used id. The on write trigger is also used to run the machine learning algorithms for determining recommendation rules and map clusters.

The second trigger type used for this application is the ‘*on call*’ trigger. This is used to handle functions that are run upon request by the user or application. To call these functions, an http or https request is made to the server. The server then runs the appropriate function and returns the result. Method parameters and return information is handled using JSON objects. The return values are returned in JavaScript Promise objects. These are objects that do not currently have a value, but will be populated with a value at some point in the future. They are commonly used to handle asynchronous http calls, as the developer is unsure of when the server will respond. This trigger is used for most of the remaining machine learning functionality. Resorting lists, retrieving needed map templates, and modifying map weights are all handled using http triggers.

The downside to using Cloud Functions is that they require internet connections to perform the needed calls. This will result in service time impacts when the user has slow connection or not internet connection. This impact is unavoidable as the app would require an internet connection to make database requests for local functions. To mitigate this impact, the functionality that requires an internet connection could be clearly separated from the functionality that does not require an internet connection. This would make it easier for the user to be aware of what functionality is available to them. Additionally, warnings could be used when the user tries to make a function call with a slow connection.

### Use of Existing Libraries

Python and the Python community offer a number of existing libraries that were used to help implement the machine learning module:

* The Pandas library was very helpful for dealing with the Excel information used during manual data collection. [22]
* NumPy can be used to help perform complex calculations [36]

Most of the machine learning algorithms were implemented from scratch, but they could modified in the future to use pre-existing algorithms that may offer a variety of improvements and optimizations:

* Scikit-learn offers a robust suite of existing Python machine learning and data manipulation functions [37]
* The TensorFlow library can be used to build Tensor models using Python [38]. This offers the extra benefit that Google Firebase provides server-side execution of

TensorFlow models, so this can be used to decrease the execution time of the module.

* To help control the use of synonyms when managing the database, the Natural Language ToolKit (NLTK) can be used [39].

# User Data Security and App Maintenance

## Privacy

Due to the applications' dependence on crowd-sourced data, user data privacy is one of the most critical areas in the application. A number of security techniques were used to keep data secure and anonymous.

The first step when a user starts using the app is to create an account. Users need to create a profile using their email and password. This helps to ensure that only authorized users can access their corresponding information. These login combinations are secured and encrypted using Google Firebase’s FirebaseUI Auth strategy. This handles a large amount of the login process including encryption, account recovery, and account linking [40]. The only step the developers need to implement is the interface for allowing users to enter their login information. This helps to ensure that data us secure. Firebase also automatically assigns a user ID to each user account that can be used to check what user is currently logged in.

The next security technique used was to create database rules. Firebase rules allow developers to create rules to define what data users can access in the database [41]. Firebase allows developers to create rules that limit who can read from the database and what they can read. This can be similarly applied for writing to the database. These rules help to prevent malicious users from accessing the data saved in the database. Without these rules, a malicious user could access the database through http requests should they gain access to the database key. One rule in the database for this application is a rule to check that the user accessing the database has a valid user ID, that they are signed in through an authorized login provider, and that the email they have verified their account using the email the account is linked to. This rule is meant to ensure that people will only be able to access the database by logging in with the app.

Another aspect in the security policy was to ensure that the machine learning does not learn specific shopping habits. All machine learning techniques learn from data in the form of past history. To prevent the machine learning from tracking specific user history, all machine learning has been kept general and anonymized. When the algorithms are running they take entire tables of data or subsets of data to use. They do not look at specific users or subsets of users. This provides a level of anonymity between the user, the machine learning, and its output.

Finally, the app will also act in accordance with Canada’s Personal Information Protection and Electronic Documents Act, so users will be notified of any data breaches of the app [15]. The Act requires that users be notified of any data breaches to the app or its database and that users be notified of what data has been accessed. Additionally, users shall be notified if this Privacy Policy ever changes. Should the Policy change, users will be provided with a summary of the changes at least 90 days in advance of the Policy change. For example, previously in the document it was stated that the machine learning could be expanded to account for individual user history. This change would be in violation of the current Privacy Policy, so should a change like this occur, users will be notified of the change and the new Privacy Policy. Users will then be given the option to opt-out of these changes.

## Business Inquiries and Monetization Plan

The application has a few potential sources of income. The application in development has the potential to possess data that may be invaluable to both grocery stores and food item companies. Given the user’s permission, the application has the ability to extract and analyze information that grocery stores and food companies may use to increase sales and prosper amongst their competitors in their respective markets. Some of the information that can be generated includes, but is not limited to:

* List of items people are buying, specifically items they generally buy together
* How often consumers are purchasing these specific items
* What stores consumers choose to go to
* Popular items at certain stores

Grocery stores can use this type of information to increase their sales. For example, a grocery store can interpret this data and conclude that one of the reasons that consumers go to stores other than theirs is because they do not sell a specific item. Therefore, it would be in their best interest to begin selling said item and placing it near the entrance of the store to showcase that item. On the other hand, a grocery store may determine that a specific item is a hot commodity and decide to alter the price or change the location of that item in order to try and increase sales.

Food item companies could also benefit from this information. For example, it can be brought to a company’s attention that users are often buying one of their products (such as bread, etc.), with another company's product (such as peanut butter etc.). Perhaps this leads to a discussion of the said company expanding into the peanut butter market in order to expand profits.

This monetization strategy has two substantial flaws. The first is that grocery stores and food companies already collect a low of this information by themselves. Their data is also a lot more accurate as they collect data based directly on what items users are buying, where as this application can be incorrect if a user purchases something not on their list or they do not buy something on their list. The other flaw with this strategy is that the purpose of the app is in direct contention with the goals of the store. The app aims to minimize the time one spends shopping, while stores aim to maximize this time. Stores may not want to support the app due to these conflicting goals.

Another potential monetization plan is to display advertisements in the app. These could be in a variety of forms such as coupons, item advertisements, brand advertisements, and store advertisements. For example, when recommending items, advertised items could be placed at the top of the list. This may be an appropriate monetization plan for the app as companies as other companies may see benefits in offering ads directly to users. This strategy could be used so long as advertisements are unobtrusive. If the ads get in the way of using the app, then users may switch to an alternative app and, due to crowd-sourced nature of the app, the app would be directly affected.

For this specific application, it was decided that the best strategy would be to not monetize the app. For the app, it is extremely important that it maintain a high userbase. If the app sells user data, people will not want to use the app. Furthermore, it ads are too obtrusive, they may also not want to use the app. The best state for the app may be to leave it free. To help offset database costs and other API costs, the app could have a website that allows users to donate. Another strategy would be to place certain features behind paywalls. These could be features that have high API costs. This would prevent user loss, while also allowing for the app to be cost neutral and potentially cost positive.

## Open-Sourcing the Application

One future goal for the application is to open-source the app, allowing future developers to maintain and improve the app. This fits in with the crowd-sourcing nature of the application as it allows the app to continue to grow. There are a few changes that would need to be made to the app to allow it to be open-sourced.

The first change that would need to be made would be to add a Continuous Integration tool to the application, such as Travis CI [67]. Continuous Integration tools offer a wide range of benefits, but the main one is that it automatically runs tests on branches and commits to ensure that changes do not harm the app. The reason that a Continuous Integration tool is not being used currently is because all of the testing is done through unit tests and manual testing. To migrate to an open-sourced project it would be necessary to create a full end-to-end testbench and accept the negative affects outlined in section *5.4 Validation, Verification, and Testing.* This would make the dependencies more difficult to manage and would change the development strategy employed, but it would be needed for an open-sourced application.

The second change that would be necessary is to remove the database key from the repository. Currently, the key used to access the database is stored in the repository. This was acceptable for the duration of the project because the repository is private on GitHub, so only the group members could access the database. Moving the app to an open-sourced project would mean making the repository public, so having the key available to all is no longer acceptable. It is generally recommended by Google, that this key be removed from source control and then accessed through Environment Variables [68]. Then, if a developer wants to make changes that would affect the database, they need to create their own database and point the application at their own implemented backend. Once the changes are merged, the user’s change can then be implemented on the application’s main database. This helps to ensure that data is not accidently deleted, that developers will not conflict with each other when working, and prevents malicious users from overflowing the database’s limits. It is recommended to have bootstrap information that users will need in their database, including initial database values, security rules, and authentication settings [69]. This could be done through JSON files, as this is what firebase uses to manage its settings and data.

The final change that would need to be made is defining what process developers need to follow if they want to contribute to the project. These rules exist for most open-sourced project. Generally, they define how pull requests need to be formatted, coding conventions that need to be followed, and rules for defining when a pull request can be merged. For this project, the rules could just be an extension of those used by the group members. For example, developers would need to state what their pull request changes and link any related issues or Kanban cards. Additionally, the pull request must pass all tests and be verified by another developer before it can be merged. These rules are used to ensure the long-term health of the app and to ensure it can be maintained.

To summarize, the changes that would be needed are expanding the existing testbench, adding a continuous integration tool to the project, hiding the database config/access key, making the bootstrap information files, writing the rules that developers must follow, and making the GitHub repository public. The process of shifting the project to an open-source project is relatively straight forward. The only downsides are the extra dependencies in the project, a limitation of the packages that can be used, and an ejection of the app to individual android/iOS packages. See section *5.4 Validation, Verification, and Testing* for more details on the downsides.

## App Distribution

There are two main ways to distribute the application out to users. One method will be to allow the user to download and install a file with extension “apk" on their Android device and allow the user to use the application using a web-interface on any platform (iOS, Windows, Mac, etc.). This is not ideal as specific platforms will have a better user experience over the alternatives. The second method of distributing the app will be to deploy the application out to the Google Play Store and the Apple App Store. There are several cost and development considerations to be made if the second method of deployment is chosen, including financial costs, to obtain developer accounts for the respective app stores. Furthermore, the application development process will need to ensure that the developed application also meets the respective app store's policies.

### Google Play Store

In order to distribute an app on the Google Play Store, the developer needs to obtain a developer account [42]. The cost associated with achieving a developer account is a one-time payment of $25 required by Google.

App deployment time on the Google Play Store takes between 1 to 7 working days. In order to get approved, all the Google Play Policy Requirements must be met [43]. Occasionally, it could take more than seven working days if the developer account has violated any of the policies outlined by the Play Store [44].

### Apple App Store

Similar to the Google Play Store, in order to distribute an app on the Apple App Store, a developer needs to obtain a developer account and enroll in the Apple Development Program [45]. Apple requires a one-time enrollment fee of $99 to enroll in the Apple Development Program.

App deployment time on the Apple App Store takes roughly 3 to 7 working days to get approved based on developer feedback found using several online sources [46]. In order to get approved, all Apple App Store Review guidelines [47] need to be satisfied. If the guidelines are satisfied, then according to Apple, it only takes 24 to 48 hours for the review process to complete [48].

# Project Timetables

## Project Iterations

Table 6: Project Iterations

|  |  |  |
| --- | --- | --- |
| **Iteration Number** | **Iteration Deliverables** | |
| **Iteration 1** | * Simple list app * Users can create accounts (all accounts have the same privilege level) * Users can create lists * Lists can be manually rearranged * Items and store layouts can be added to the database * Data will be gathered for machine learning | |
| **Iteration 2** | • Predict the item location for unknown locations (collaborative filtering)  • Automatic reorganization of lists  • Prompt user to reorganize the list when they enter a store | |
|  | • | The store will be manually inputted |
|  | • | Lists can be shared between users |
|  | • | Introduce user privilege levels |
|  | • | Machine learning updated to handle reorganization of the list (rearrange based on departments and layouts of departments) |
| **Iteration 3** | • | Map API used to specify stores and other APIs for integrating with external/third-party apps |
|  | • | Drag and Drop UI for adding items |
|  | • | Update the list to display the price of each item + total price |
|  | • | Improvement of machine learning to increase the accuracy of list rearrangement based on aisles |
| **Iteration 4** | • | Users can add recommended items directly to their list |
|  | • | Users can find recipes and import its items into a list |
|  | • | Recipes can be favourited and shared between users |
|  | • | Update machine learning to recommend items to users based on association learning |
| **Non-Programming Deliverables** | • | Sketch of the privacy policy |
|  | • | Monetization plan |

## Project Gantt Chart

(INSERT GANTT CHART HERE JOBIN)

Table 7: Project Gantt Chart

# Conclusions and Recommendations

Grocery shopping can be a time-consuming process and is complicated when shopping at a store with an unfamiliar layout. This, along with marketing strategies which dictate confusing store layouts to influence customers to spend more time at stores can make grocery shopping a tedious task.

Our solution to this problem is to provide an application that will help guide users through the store. The application takes the list generated by a user and those the list is shared with, and then reorganizes the list in order from which the user is expected to see the items when they enter the store. This is done using crowdsourced data from users who are more familiar with the layout of a store.

To improve the applications functionality and fill in gaps left in the crowdsourced data, a machine learning suite is used. The machine learning enables the core features in the application, specifically the ability to reorganize lists into the order you are expected to see them when shopping. If a specific item is not crowdsourced in the database, the machine learning also uses algorithms and factors such as surrounding items to estimate where the location of a specific item should be. It is also used in order to recommend items that are popular amongst other users lists in order to promptly add trending items to your list.

The application offers many other features such the ability to find grocery stores near a given location, and the ability to find recipes and import their ingredients into any list. There are other features that can be added to help improve the usability of the application. The login can be expanded to allow for other login partners such as Google, Facebook, and Apple login. This would make it so that the user does not need to create a new account to use the app. Another feature would be the ability to resort lists by price or other metrics that a user may want to view the items in a list. The recommendation of items could also be improved to be tailored to each specific user, rather than general users, in order to increase the chance that they are offered an item they would want to purchase. The addition of items to lists could also be improved by offering items the user frequently purchases. Lastly, the display could be updated to include images or icons for items to make the app more visually appealing.

In conclusion, machine learning modules were applied to grocery lists and crowd-sourced data in order to help users save time at the grocery store. Eliminating the factor of unknown and unclear layouts of grocery stores while shopping can help users save time while performing the essential task of grocery shopping.

# References

1. -- --, “The Secrets Behind Your Grocery Store's Layout,” *Real Simple*, 14-Feb-2018. [Online]. Available: https://www.realsimple.com/food-recipes/shopping-storing/more-shopping-storing/grocery-store-layout. [Accessed: 09-Mar-2020].
2. --, --. (2019, November 19). Samsung Notes – Apps on Google Play. Retrieved December 5, 2019, from <https://play.google.com/store/apps/details?id=com.samsung.android.app.notes&hl=en_CA>.
3. --, --. (2016, May 22). ‎Notes on the App Store. Retrieved December 5, 2019, from <https://apps.apple.com/ca/app/notes/id1110145109>.
4. Play.google.com, 2019. [Online]. Available: <https://play.google.com/store/apps/details?id=com.capigami.outofmilk>. [Accessed: 27- Sep- 2019].
5. Play.google.com, 2019. [Online]. Available: <https://play.google.com/store/apps/details?id=com.l>. [Accessed: 27- Sep- 2019].
6. -- --, “Geo-location APIs  |  Google Maps Platform  |  Google Cloud,” *Google*. [Online]. Available: https://cloud.google.com/maps-platform/. [Accessed: 09-Mar-2020].
7. R. Amir, “BING Maps - HERE Developer,” *HERE Developer Blog*, 06-Jan-2016. [Online]. Available: https://developer.here.com/blog/bing-maps. [Accessed: 09-Mar-2020].
8. -- --, “react-native-community/react-native-maps,” *GitHub*, 06-Mar-2020. [Online]. Available: https://github.com/react-native-community/react-native-maps. [Accessed: 09-Mar-2020].
9. -- --, “Documentation,” *Push Notifications - Expo Documentation*. [Online]. Available: https://docs.expo.io/versions/latest/guides/push-notifications/. [Accessed: 09-Mar-2020].
10. -- --, “Recipes,” *Food Network*, 2020. [Online]. Available: https://www.foodnetwork.com/recipes. [Accessed: 09-Mar-2020].
11. -- --, “Food API and Recipe API,” *spoonacular recipe and food API*. [Online]. Available: https://spoonacular.com/food-api. [Accessed: 09-Mar-2020].
12. -- --, “API Pricing,” *spoonacular recipe and food API*. [Online]. Available: https://spoonacular.com/food-api/pricing. [Accessed: 09-Mar-2020].
13. "What is Software Engineering? - Definition from Techopedia", Techopedia.com, 2019. [Online]. Available: <https://www.techopedia.com/definition/13296/software-engineering>. [Accessed: 27- Sep- 2019].
14. "7 Facts about Grocery Shopping that Might Shock You", Fivestarhomefoods.com, 2019. [Online]. Available: <https://www.fivestarhomefoods.com/blog/grocery-shopping-facts>. [Accessed: 27- Sep- 2019].
15. vol. STATUTES OF CANADA 2015 Chapter 32. 2015, p. 2.
16. B. Ennis. Professional Engineers Ontario. Class Lecture, Topic: “Engineering – A Regulated Profession” School of Engineering, Carleton University, Ottawa, Ontario, Jan. 28, 2020.
17. "What is Agile Software Development?", Agile Alliance, 2019. [Online]. Available: <https://www.agilealliance.org/agile101/>. [Accessed: 27- Sep- 2019].
18. J. Code, "React Native vs Ionic vs NativeScript: A Practical Guide | Jscrambler Blog", Jscrambler, 2019. [Online]. Available: [https://blog.jscrambler.com/react-native-vs-ionic-vs-nativescript-apractical-guide/](https://blog.jscrambler.com/react-native-vs-ionic-vs-nativescript-a-practical-guide/) . [Accessed: 27- Sep- 2019].
19. -- --, “akveo/react-native-ui-kitten,” *GitHub*, 24-Feb-2020. [Online]. Available: https://github.com/akveo/react-native-ui-kitten. [Accessed: 09-Mar-2020].
20. -- --, “What is UI Kitten?,” *React Native UI Kitten*, 2019. [Online]. Available: https://akveo.github.io/react-native-ui-kitten/docs/getting-started/what-is-ui-kitten#have-questions. [Accessed: 09-Mar-2020].
21. "Why Python is the most popular language used for Machine Learning", Medium, 2019. [Online]. Available: [https://medium.com/@UdacityINDIA/why-use-python-for-machine-learninge4b0b4457a77](https://medium.com/@UdacityINDIA/why-use-python-for-machine-learning-e4b0b4457a77) . [Accessed: 27- Sep- 2019].
22. "Python Data Analysis Library — pandas: Python Data Analysis Library", Pandas.pydata.org, 2019. [Online]. Available: <https://pandas.pydata.org/>. [Accessed: 27- Sep- 2019].
23. Bhandari, R., Chatterjee, S., Gupta, K., & Panda, B. (2019). How to avoid the pitfalls of IT crowdsourcing to boost speed, find talent, and reduce costs. Retrieved December 5, 2019, from <https://www.mckinsey.com/business-functions/mckinsey-digital/our-insights/how-to-avoid-the-pitfalls-of-it-crowdsourcing>.
24. A. Burkov, The hundred-page machine learning book. [S.l.]: Andriy Burkov, 2019, p. Chapter 5.
25. J. Brownlee, "How Much Training Data is Required for Machine Learning?", Machine Learning Mastery, 2019. [Online]. Available: [https://machinelearningmastery.com/much-training-datarequired-machine-learning/](https://machinelearningmastery.com/much-training-data-required-machine-learning/) . [Accessed: 27- Sep- 2019].
26. "Collaborative filtering", En.wikipedia.org, 2019. [Online]. Available: <https://en.wikipedia.org/wiki/Collaborative_filtering>. [Accessed: 27- Sep- 2019].
27. "Cosine Similarity and Handling Categorical Variables", Medium, 2019. [Online]. Available: <https://medium.com/@rahulkuntala9/cosine>[-similarity-and-handling-categorical-variables29f907951b5](https://medium.com/@rahulkuntala9/cosine-similarity-and-handling-categorical-variables-29f907951b5) . [Accessed: 27- Sep- 2019].
28. --, --. (2019, December 5). Levenshtein distance. Retrieved December 5, 2019, from <https://en.wikipedia.org/wiki/Levenshtein_distance>.
29. --, --. (2019). Ranking Question. Retrieved December 5, 2019, from <https://help.surveymonkey.com/articles/en_US/kb/How-do-I-create-a-Ranking-type-question>.
30. -- --, “Euclidean distance,” *Wikipedia*, 12-Feb-2020. [Online]. Available: https://en.wikipedia.org/wiki/Euclidean\_distance. [Accessed: 09-Mar-2020].
31. "Association rule learning", En.wikipedia.org, 2019. [Online]. Available: <https://en.wikipedia.org/wiki/Association_rule_learning>. [Accessed: 27- Sep- 2019].
32. "Apriori algorithm", En.wikipedia.org, 2019. [Online]. Available: <https://en.wikipedia.org/wiki/Apriori_algorithm>. [Accessed: 27- Sep- 2019].
33. "Data Mining Algorithms In R/Frequent Pattern Mining/The FP-Growth Algorithm - Wikibooks, open books for an open world", En.wikibooks.org, 2019. [Online]. Available: [https://en.wikibooks.org/wiki/Data\_Mining\_Algorithms\_In\_R/Frequent\_Pattern\_Mining/The\_F](https://en.wikibooks.org/wiki/Data_Mining_Algorithms_In_R/Frequent_Pattern_Mining/The_F%20) [P-Growth\_Algorithm#FP-Growth\_Algorithm](https://en.wikibooks.org/wiki/Data_Mining_Algorithms_In_R/Frequent_Pattern_Mining/The_FP-Growth_Algorithm#FP-Growth_Algorithm) . [Accessed: 27- Sep- 2019].
34. "A Comparative Study of Association Rules Mining Algorithms", 2019. [Online]. Available: [https://www.researchgate.net/publication/267866191\_A\_Comparative\_Study\_of\_Association\_](https://www.researchgate.net/publication/267866191_A_Comparative_Study_of_Association_%20) [Rules\_Mining\_Algorithms](https://www.researchgate.net/publication/267866191_A_Comparative_Study_of_Association_Rules_Mining_Algorithms) . [Accessed: 27- Sep- 2019].
35. -- --, “Database Triggers |  Firebase,” *Google*. [Online]. Available: https://firebase.google.com/docs/functions/database-events. [Accessed: 09-Mar-2020].
36. "NumPy — NumPy", Numpy.org, 2019. [Online]. Available: <https://numpy.org/>. [Accessed: 27- Sep- 2019].
37. "scikit-learn machine learning in Python — scikit-learn 0.21.3 documentation", Scikit-learn.org, 2019. [Online]. Available: <https://scikit-learn.org/stable/>. [Accessed: 27- Sep- 2019].
38. "TensorFlow", TensorFlow, 2019. [Online]. Available: <https://www.tensorflow.org/>. [Accessed: 27- Sep- 2019].
39. -- --, "Natural Language Toolkit — NLTK 3.4.5 documentation", *Nltk.org*, 2019. [Online]. Available: http://www.nltk.org/. [Accessed: 29- Sep- 2019].
40. -- --, “Firebase Authentication,” *Google*. [Online]. Available: https://firebase.google.com/docs/auth. [Accessed: 09-Mar-2020].
41. -- --, “Firebase Security Rules,” *Google*. [Online]. Available: https://firebase.google.com/docs/rules. [Accessed: 09-Mar-2020].
42. "Google Play Console | Android Developers", Android Developers, 2019. [Online]. Available: <https://developer.android.com/distribute/console>. [Accessed: 27- Sep- 2019].
43. "Google Play Policies | Android Developers", Android Developers, 2019. [Online]. Available: <https://developer.android.com/distribute/play>[-policies](https://developer.android.com/distribute/play-policies) . [Accessed: 27- Sep- 2019].
44. "Developer Policy Center", Play.google.com, 2019. [Online]. Available: <https://play.google.com/about/developer>[-content-policy/](https://play.google.com/about/developer-content-policy/) . [Accessed: 27- Sep- 2019].
45. "Before You Enroll - Apple Developer Program - Apple Developer", Developer.apple.com, 2019. [Online]. Available: <https://developer.apple.com/programs/enroll/>. [Accessed: 27- Sep- 2019].
46. 2019. [Online]. Available: [https://www.quora.com/How-long-does-it-take-for-an-app-to-beapproved-on-major-platforms-Google-Apple-etc/](https://www.quora.com/How-long-does-it-take-for-an-app-to-be-approved-on-major-platforms-Google-Apple-etc/) . [Accessed: 27- Sep- 2019].
47. "App Store Review Guidelines - Apple Developer", Developer.apple.com, 2019. [Online]. Available: <https://developer.apple.com/app-store/review/guidelines/>. [Accessed: 27- Sep- 2019].
48. "App Review - App Store - Apple Developer", Apple Developer, 2019. [Online]. Available: <https://developer.apple.com/app>[-store/review/](https://developer.apple.com/app-store/review/) . [Accessed: 27- Sep- 2019].
49. "Jest · Delightful JavaScript Testing", Jestjs.io, 2019. [Online]. Available: [https://jestjs.io.](https://jestjs.io/) [Accessed: 27- Sep- 2019].
50. "Snapshot Testing · Jest", Jestjs.io, 2019. [Online]. Available: <https://jestjs.io/docs/en/snapshot>testing. [Accessed: 27- Sep- 2019].
51. -- --, “wix/Detox,” *GitHub*, 09-Mar-2020. [Online]. Available: https://github.com/wix/Detox. [Accessed: 09-Mar-2020].
52. -- --, “Introduction,” *Appium*. [Online]. Available: http://appium.io/docs/en/about-appium/intro/?lang=en. [Accessed: 09-Mar-2020].
53. D. Editors, "America's Most Delish", *Delish*, 2019. [Online]. Available: <https://www.delish.com/food/a23117553/america>[-most-delish-grocery-items-2018/](https://www.delish.com/food/a23117553/america-most-delish-grocery-items-2018/) . [Accessed: 29- Sep- 2019].
54. "Introduction to Recommender System", Medium, 2019. [Online]. Available: <https://towardsdatascience.com/intro>[-to-recommender-system-collaborative-filtering64a238194a26](https://towardsdatascience.com/intro-to-recommender-system-collaborative-filtering-64a238194a26) . [Accessed: 27- Sep- 2019].
55. "Label Encoder vs. One Hot Encoder in Machine Learning", Medium, 2019. [Online]. Available: <https://medium.com/@contactsunny/label>[-encoder-vs-one-hot-encoder-in-machine-learning3fc273365621](https://medium.com/@contactsunny/label-encoder-vs-one-hot-encoder-in-machine-learning-3fc273365621) . [Accessed: 27- Sep- 2019].
56. "Encoding Categorical Features", Medium, 2019. [Online]. Available: <https://towardsdatascience.com/encoding>[-categorical-features-21a2651a065c](https://towardsdatascience.com/encoding-categorical-features-21a2651a065c) . [Accessed: 27- Sep- 2019].
57. A. Burkov, The hundred-page machine learning book. [S.l.]: Andriy Burkov, 2019.
58. "Grocery Shopping Statistics: 23 Fun Size Facts to Know", CreditDonkey, 2019. [Online]. Available: <https://www.creditdonkey.com/grocery-shopping-statistics.html>. [Accessed: 27- Sep- 2019].
59. W. DictVectorizor? and P. Joshi, "When to use One Hot Encoding vs LabelEncoder vs DictVectorizor?", Data Science Stack Exchange, 2019. [Online]. Available: <https://datascience.stackexchange.com/questions/9443/when>[-to-use-one-hot-encoding-vslabelencoder-vs-dictvectorizor](https://datascience.stackexchange.com/questions/9443/when-to-use-one-hot-encoding-vs-labelencoder-vs-dictvectorizor) . [Accessed: 27- Sep- 2019].
60. "Travel Time Optimization with Machine Learning And Genetic Algorithm", Medium, 2019. [Online]. Available: [https://towardsdatascience.com/travel-time-optimization-with-machinelearning-and-genetic-algorithm-71b40a3a4c2](https://towardsdatascience.com/travel-time-optimization-with-machine-learning-and-genetic-algorithm-71b40a3a4c2) . [Accessed: 27- Sep- 2019].
61. "Travelling salesman problem", En.wikipedia.org, 2019. [Online]. Available: <https://en.wikipedia.org/wiki/Travelling_salesman_problem>. [Accessed: 27- Sep- 2019].
62. Orion.math.iastate.edu, 2019. [Online]. Available: <https://orion.math.iastate.edu/dept/links/formulas/form2.pdf>. [Accessed: 27- Sep- 2019].
63. "3-opt", En.wikipedia.org, 2019. [Online]. Available: [https://en.wikipedia.org/wiki/3-opt.](https://en.wikipedia.org/wiki/3-opt) [Accessed: 27- Sep- 2019].
64. --, --. (2019). UI Kitten 4.0. Retrieved December 5, 2019, from <https://akveo.github.io/react-native-ui-kitten/>.
65. "Documentation | Firebase", Firebase, 2019. [Online]. Available: <https://firebase.google.com/docs>. [Accessed: 27- Sep- 2019].
66. -- --, “Working Alone Guidelines,” *Working Alone Guidelines*. Environmental Health and Safety, Ottawa, ON. Available: <https://carleton.ca/ehs/wp-content/uploads/Working-Alone-Guidelines_Final.pdf> . [Accessed: 15- Mar- 2020].
67. -- --, “Hey there!” *Travis CI - Test and Deploy Your Code with Confidence*. [Online]. Available: https://travis-ci.org/. [Accessed: 15-Mar-2020].
68. -- --, “Understand Firebase projects,” *Google*. [Online]. Available: https://firebase.google.com/docs/projects/learn-more. [Accessed: 15-Mar-2020].
69. D. Stevenson, “Can I open source my firebase Android project?” *Quora*, 04-Oct-2016. [Online]. Available: https://www.quora.com/Can-I-open-source-my-firebase-Android-project. [Accessed: 15-Mar-2020].

# Appendix I – List of Item for Initial Crowd Sourcing

The list of items to be manually collected are made up of 50 items. This includes the 30 most popular items in grocery stores and an additional 20 items to fill up the data on departments. The list was compiled using a list of the top 100 items provided by Delish [53]:

**Dairy Products**

1. Milk
2. Eggs
3. Cheese
4. Butter
5. Yogurt

**Bakery**

1. Sliced Bread
2. Bagels
3. Cinnamon Buns
4. Muffins
5. Pie

**Whole Body**

1. Toilet Paper
2. Shampoo
3. Toothpaste
4. Shower Gel
5. Deodorant
6. Laundry Detergent

**Meat and Poultry**

1. Chicken
2. Ham
3. Sausage
4. Chicken Nuggets
5. Turkey

**Produce**

1. Potatoes
2. Lettuce
3. Rice
4. Apples
5. Bananas

**Condiments**

1. Ketchup
2. Mustard
3. Mayonnaise
4. Olive Oil 31. Pesto

**Wine and Spirits**

1. Beer
2. Wine

**Frozen Foods**

1. Pizza
2. Ice Cream
3. Frozen Burgers
4. Frozen Vegetables

**Junk Food**

1. Cookies
2. Chocolate
3. Jell-O
4. Chips
5. Soda

**Drinks**

1. Tea
2. Juice
3. Water
4. Coffee

**Seafood**

1. Tuna
2. Sardines
3. Shrimp
4. Salmon

# Appendix II – Initial Version of ML Algorithms

The machine learning module will be used for several features:

* Estimate the location of items
* Estimate the layout of unknown stores
* Sort list of items to follow the shortest path
* Recommend items to a user
  + Based off of similar purchasing patterns to other users
  + Based off of purchase history (Ex. Item has not been purchased in a while, frequency of re-ordering)

### Collaborative Filtering (Estimate location of items)

To estimate the location of items, a machine learning algorithm called collaborative filtering is going to be used. Collaborative filtering is a method often used in recommender systems to recommend items to users based on shared prior opinions [27]. Collaborative filtering generally follows two steps. The first step is to find the users who are most similar to a given user by comparing rating patterns. Then, use the history of those users to estimate how the given user will rate a new object.

In the example of predicting item locations based on the locations in other stores, the following steps would be used. In this example, a user is in the store ACTIVE\_STORE, and they want to find item UNKNOWN\_ITEM, but the location of UNKNOWN\_ITEM in ACTIVE\_STORE is unknown. The only information known about ACTIVE\_STORE is the location of some items, NUM\_ITEMS. Additionally, there is a database containing X stores, each with some known item locations. Refer to Table 7:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Store Name** | **Item 1 Location** | **UNKNOWN\_ITEM** | **Item 3 Location** | **Item 4 Location** |
| **ACTIVE\_STORE** | DEPARTMENT\_1 | -- | DEPARTMENT\_3 | DEPARTMENT\_5 |
| **STORE\_1** | -- | DEPARTMENT\_1 | DEPARTMENT\_4 | -- |
| **STORE\_2** | DEPARTMENT\_1 | DEPARTMENT\_6 | DEPARTMENT\_3 | DEPARTMENT\_4 |
| **STORE\_3** | DEPARTMENT\_2 | -- | -- | DEPARTMENT\_7 |

Table 7: Example of Store Database

To estimate the location of UNKNOWN\_ITEM, the algorithm would start by checking every other store in the database to find the stores that are most similar to ACTIVE\_STORE. One way of determining similarity would be to check how many items in the store match the location of the item in ACTIVE\_STORE for each item in NUM\_ITEMS. For instance, if each store is assigned a score starting at 0, and then the score was increased by 1 if the locations match, decreased by 1 if the locations are different, and unchanged if the location is unknown, then the scores for each store in the table above would be as follows: STORE\_1 has a score of 0, STORE\_2 has a score of 1, STORE\_3 has a score of -2. After the similarities have been determined, each store is assigned a weight based on the similarity metric. These weights are then used to calculate a weighted average based on the known information to determine an estimate. In this case, the location would most likely be DEPARTMENT\_6 as STORE\_2 has the highest weight.

One problem in using collaborative filtering in this case, is that collaborative filtering is generally used to calculate a weighted average based on a continuous value. For example, a score between 1.0 and 10.0. Whereas, in this case, it is being used to classify locations based on discrete, non-continuous values. There are two ways to handle this issue [54][55][56][57]. One method, called Label Encoding, is to assign integer values to each discrete value from 0 to n – 1 based on the number of possible values. For example, DEPARTMENT\_1 would have value 0, DEPARTMENT\_2 has value 1, and so on. The second method is called One-Hot Encoding. In One-Hot Encoding, each categorical feature is transformed into several binary ones. For example, if there are 5 departments, they would be transformed as follows:

DEPARTMENT\_1 = [1, 0, 0, 0, 0]

DEPARTMENT\_2 = [0, 1, 0, 0, 0]

DEPARTMENT\_3 = [0, 0, 1, 0, 0]

DEPARTMENT\_4 = [0, 0, 0, 1, 0] DEPARTMENT\_5 = [0, 0, 0, 0, 1]

For the case of converting the department categories, Label Encoding is going to be used. Generally, Label Encoding is considered the lesser of the two methods because it implies that there is an order to the classification labels and that this order is significant to the algorithm [58]. This may lead the algorithm to attempt to find a regularity when there is not one which can cause overfitting where the model is heavily dependent on the training data. One-Hot Encoding fixes this issue by using a binary approach. The reason that Label Encoding is going to be used is due to the scenario of the task. The average grocery store contains 42400 items on average [59] across dozens of departments. If One-Hot Encoding is used, the amount of memory required to map out one store would be extremely large, and most of the space would be empty data [60]. Additionally, departments that are specific to only certain stores and departments that carry few items would require entire columns filled with mostly empty space. Furthermore, One-Hot Encoding would be problematic to implement in the database. The database uses a NoSQL implementation. In NoSQL database implementations, it is recommended to keep the table shallow. To properly implement the desired table format with One-Hot Encoding, it would most likely be the case that two separate tables would need to be used: one for all the items in a store and another for the location of each item in the store. This would lead to a large number of required redundant reads and writes.

With the numerical representation decided, the next issue to address is how to calculate the similarity between stores. As stated previously, it is desired to compare stores based on the locations of shared items, i.e. is item A in the same location in both stores being compared. To perform this calculation, it will be necessary to compare each item in each store to build up a total similarity score for that store. The most common approaches for calculating similarities between two users (stores in this case) are Pearson correlations and cosine-based approaches:

Equation 4: Pearson correlation [27]

Equation 5: Cosine-based correlation [27]

Due to the format of the data in the database, it will be necessary to use a modification of the cosine formula. In the case of just the department, the input and output data are purely classification based. The cosine formula is generally used in the case of One-Hot Encoding, so it will need to be modified for this case. The similarities will be calculated using a simple percentage measure. When just departments are used, the similarities can be calculated using the following:

Equation 6: Similarity Metric Based Off of Department

Where,

1. is the active store
2. is the store being compared to the active store

Ix,y is the set of items in both store x and y

Lx,i is the location of item i in store x

Ly,i is the location of item i in store y

N is the number of known items in store x

Then, to update the location similarity to work with aisle numbers in addition to the departments, the equation will need to be updated as follows:

Equation 7: Distance Metric Based Off of Department and Aisle Where,

1. is the active store
2. is the store being compared to the active store

Ix,y is the set of items in both store x and y

Dx,i is the department of item i in store x

Ax,i is the aisle number of item i in store x

Dy,i is the department of item i in store y

Ay,i is the aisle number of item i in store y

N is the number of known items in store x

H is a hyperparameter that determines the weight between the department and aisle matching

Then, using the similarity metrics, the location will be determined using a weighted mode calculation.

Collaborative Filtering has several additional problems that need to be addressed [27]. One issue is data sparsity. This problem is caused by the fact that collaborative filtering cannot work until information about the store is gathered. If there are no known item locations in the store, then the similarity metric cannot be calculated. This problem can be addressed by updating the machine learning module to simply return the mode location of that item if nothing else is known. Additionally, this problem can be minimized by making it easy to add items to the database. The easier it is to add items, the more likely that information will be known, and the more likely collaborative filtering is going to be able to estimate the location.

Another issue is scalability. As stated above, collaborative filtering compares each item in each store. This can be a massive computing undertaking as the number of stores and items increases. This issue can be addressed by minimizing the amount of data being compared. One way of minimizing the data is to compare fewer stores. Stores of the same brand are more likely to have the same layouts, so by only comparing to stores of the same brand when possible, the algorithm can be sped up. For example, if a user is in a Wal-Mart, then the algorithm would only compare to other Wal-Marts. Another method would be to prioritize stores geographically closer together.

The third issue with collaborative filtering is synonyms. This refers to items that may have multiple different names. For example, French fries have a number of names such as fries, chips, and wedges. This issue will be difficult to address, but one way of solving it would be to save the synonyms for each item and then when that item is added to the database under one of its synonyms, it is instead saved to a single uniform name. Another method that can be used to address this problem would be to use a thesaurus API to convert item names to a common synonym.

There is one more problem that the application is very unlikely to encounter, the issue of Gray Sheep/Black Sheep. This issue refers to users who do not follow regular patterns or who regularly go against the grain. This issue is unlikely to affect the application as grocery stores tend to follow similar patterns, putting items in the same location, to make it easier for customers to find items.

### Travelling Salesman Problem (Shortest Path and Layout Estimation)

The goal of sorting the list of items to follow the shortest path is an example of a popular problem called The Travelling Salesman Problem, or TSP for short [61]. In this problem, there is a map with N nodes, and the goal is to find the shortest path that visits each node. This is a very old problem, being first formulated in 1930 [61], so there are a wide range of possible heuristics and algorithms that can be employed by the machine learning module to determine the path. As the algorithm is going to be used frequently by users, an approach that finds a balance between accuracy and speed should be used. For example, a greedy algorithm where the next closest node is always chosen is very fast, but its solution is generally 25% higher than the minimum value [61]. On the other hand, an algorithm that always finds the minimum solution will take a lot longer to compute, which will frustrate users.

It is difficult to predict which algorithm will be the most efficient for the desired application, so several different ones will need to be tested to determine which to use. The first solution to try is called the greedy algorithm. This algorithm is as follows [61]:

1. Pick a random starting point
2. Travel to closest unvisited point using a distance metric

Equation 8: Distance Metric [62]

1. Repeat step 2 until all points have been visited

The benefits of this algorithm are the simplicity to implement and the speed of calculation. The downside to this method is that its results are, on average, 25% higher than the shortest route, but there are layouts where the greedy algorithm may return the worst solution. Another heuristic to try is called Christofides algorithm. The steps for this heuristic are [61]:

1. Find a minimum spanning tree
2. Create duplicates for every node in the tree
3. Find a Eulerian tour for this graph
4. Convert to TSP and fix revisited nodes by creating shortcuts.

The benefit of this solution is that at worst, its solution is 50% longer than the minimum. The downsides are the difficulty to implement and the length of execution time. The above two heuristics are referred to as constructive heuristics and are often used alongside improvement heuristics to improve the solution. The improvement heuristic that will be used has the general name of k-opt, with its most popular implementation of 3-opt. This improvement works as follows [63]:

1. Start with a basic solution from one of the construction heuristics
2. Choose K edges and remove them
3. Reconnect the now unconnected nodes in a way that minimizes the length of the tour
4. If a solution is found, use the new solution, otherwise, use the original solution
5. Repeat steps 2 – 4 for all combinations

The k-opt improvement generally results in a solution 5% better than Christofides algorithm. Due to this relatively small improvement, it may not be beneficial to use depending on the increased runtime.

The issue that needs to be addressed with this component is the number of points. For example, if a user has 50 items on their list, the algorithm would need to use all 50 items when solving the TSP. Regardless of the method used for calculation, this will would take a while to compute, so users would not want to use the feature for large lists. There are several methods that can be used for solving this issue. One solution would be to group adjacent items into one node. If a user has several items in one aisle, these items could be treated as a single point. Similarly, instead of doing the TSP for individual items, it could instead be applied to departments. Lists could first be organized based on the department and aisle each item is located in. Then, the midpoint of each department to visit would be used in the TSP. As can be seen, both solutions require sacrificing specificity. This may be acceptable in this implementation, as it would not have too large of an impact on the result.

A related issue is how to deal with stores when the layout is unknown. This can be solved by first applying the collaborative filtering outlined above to estimate the map based on similarity. If no locations are known inside of the store, then the collaborative filter will be a complete average. The result of the collaborative filter will be the average midpoint of each department type. This can be used to build a skeleton of the map. Next, the location of each item will be estimated using the “Estimate Item Location” feature as described above. Once all this data has been gathered, the shortest order in the list can be calculated.

# Appendix III - Work Completed At Time Of Progress Report

The team has completed Iteration 1, and most of the proposed requirements from Iteration 2. Overall, the completed work can be broken down into the following sections, which include machine learning, backend and frontend.

## Machine Learning

There are three major machine learning components to be built in the suite to support the application. The three are estimating item location, sorting lists based on fastest paths, and recommending items to users. Of these three components, two have been implemented in the app in basic forms: estimating item locations and sorting lists. Both of these components have been implemented using a combination of frontend and backend support.

The estimation of item locations is performed entirely in the backend, as this functionality is not user-facing. The estimation functionality is only called by the list sorting functions when the location of an item is unknown. When the location of an item is unknown, the sorting function calls the item prediction method. The first step in the method is to check what stores contain the item trying to be found. Once the stores are determined, the similarity of each of these stores to the desired store is calculated as described in *5.5.1 Collaborative Filtering (Estimate location of items).* The most similar stores are then used to estimate the location using a majority calculation. The current implementation of this functionality only supports location estimation on a Department level, so it still needs to be extended to support an aisle level estimation. Currently, this expansion of the functionality is planned for Iteration 3.

There are several improvements that could be made to the current implementation of the function. One of the biggest issues is the need to recalculate the similarity every time an item location is being predicted. This can be problematic as multiple locations may need to be predicted for each list reorganization. As the amount of data increases, the calculation will also take longer and longer. To address this issue, a lookup table could be maintained that keeps track of the similarities between all stores. Then, when the known data on a store increases by a certain threshold, that store’s similarity row in the table could be recalculated and updated. This would remove the need for constant recalculation. Another similar improvement would be to change how the stores to compare with are determined. Currently, all stores are checked to see if they include the desired item. This could be reversed to speed up the check. If each item kept track of its known locations, then the algorithm could just check the item to determine the stores that should be checked.

As for the sorting of the lists, four sorting methods have been implemented. The four methods are: order added, alphabetical, by location, and fastest path. The default sorting order is the order added, which is the order that the items were added to the list. The reason that this is the default is because it is how the lists are saved in the database. All of the sorting functionality is handled using cloud functions, except for alphabetical, which is done locally as it does not require any information from the database other than the names and ids of the items.

The sorting work as follows. When the user loads the page, the app loads the list and saves a local copy. A listener is then created at this stage to handle updates such as items being added or removed. Using a dropdown menu, the user can then select the sort order. If the user selects location-based or fastest-path-based, the app will use a modal to prompt the user to enter the store they are visiting. The corresponding cloud function is then run to retrieve the new list order. Using the returned information, the local copy of the list is then updated. The time it takes to update the list is related to the length of the list and how much is known about the store. A list containing items where each location needs to be predicted would take the longest.

The location-based and fastest-path-based sorting are handled similarly. First, the location of each item is retrieved or predicted if unknown. The items are then sorted based on their departments and aisles. If the location-based sorting is being used, then the list is returned at this point. If the fastest-path based sorting is being used, then the map of the store is retrieved. The items are then rearranged to match the department layout, as determined by the known store map. Currently, the sorting still has the same limitation as to the location prediction in that it does not go to an aisle-based granularity. This issue is planned to be addressed in Iteration 3 of the project.

## Backend

The backend of the application was set up using Google Firebase as originally planned. The backed is composed of two major components, which are the database and cloud functions.

### Database

During Iteration 1 of the project, the team was able to set up a Firebase real-time database as planned. With the creation of the database, security rules needed to be applied to the database in order to protect it from unauthorized users and improper use. Firebase security rules can be applied to every path/node in the database. Firebase security rules follow a rules cascade policy where if a broad rule has allowed read or write to a node, the deeper nodes will also follow that broad rule. The following rules have been tested and deployed for the firebase database:

* A security rule has been applied to all nodes to only allow read or write access to users that have a ‘Valid Authorization ID’. For the case of this database, it will be considered that a ‘Valid Authorization ID’ to be one that is not ‘null’, has an authorization provider that is not ‘anonymous’ and the authorization token has an ‘email verified’ property with the value of true. This rule is valid for all nodes unless it is specified that it is not valid.
* A specific rule has been applied to the ‘users/$user\_id’ path. This rule ensures that users can only read or write to that path if and only if that path belongs to that user. That is, the ‘$user\_id’ has to be the same as their ‘auth.uid’
* A specific rule has been applied to the ‘lists/$list\_id’ path. This rule ensures that a user can only read a list with ‘$list\_id’ only if that same ‘$list\_id’ is found under the same user’s created lists which is within the ‘users/auth.uid/created/$list\_id’ path or shared list which is within the ‘users/auth.uid/shared/$list\_id’ path.
* Another specific rule has been applied to the ‘lists/$list\_id’ path. This rule ensures that a user can only write to a list with ‘$list\_id’ only if that same ‘$list\_id’ is found under the same user’s created lists. This is found within the ‘users/auth.uid/created/$list\_id’ path or shared list which is within the ‘users/auth.uid/shared/$list\_id’ path. This rule also ensures that a user is not allowed to delete a list path that was shared with them by another user.
* A security rule has been applied to the ‘items’ path to ensure that only users who have a node with a key called ‘privileged’ within the ‘users/auth.uid’ path with a value of ‘true’ is allowed to write the ‘items’ path.

### Crowd Sourcing

To populate the database, each team member visited three stores to gather sample information as described in section *6.1 Data Collection*. The data collection process included scenarios in which the team members had to take pictures of aisle tags, as seen in Figure 24 and convert that information into a layout drawing, as seen in Figure 25. Additionally, the crowdsourcing user interface tools were created. These include the Map Creator page for outlining store layout, the Register Item page for adding items to the database, and the Add Item Location page for adding the location of items to the database. The pages for registering items and adding item locations are simple pages with a series of input boxes to retrieve the needed information. The Register Item page currently prompts users for an item name and a size descriptor. The Add Item Location page has input boxes for an item’s name, the store name for the location, the department, and the aisle number. Both of these pages can be expanded if more information is required in the future. Finally, the map creator page is a simple reorderable list containing the departments in a store. The user can move departments to place them in the order they appear in the store. Departments can also be removed and added as needed. These pages can be seen throughout section *4.1.2 Crowd-Sourcing Information for Database Growth*.



Figure 23: Sample Capture Of Aisle Information

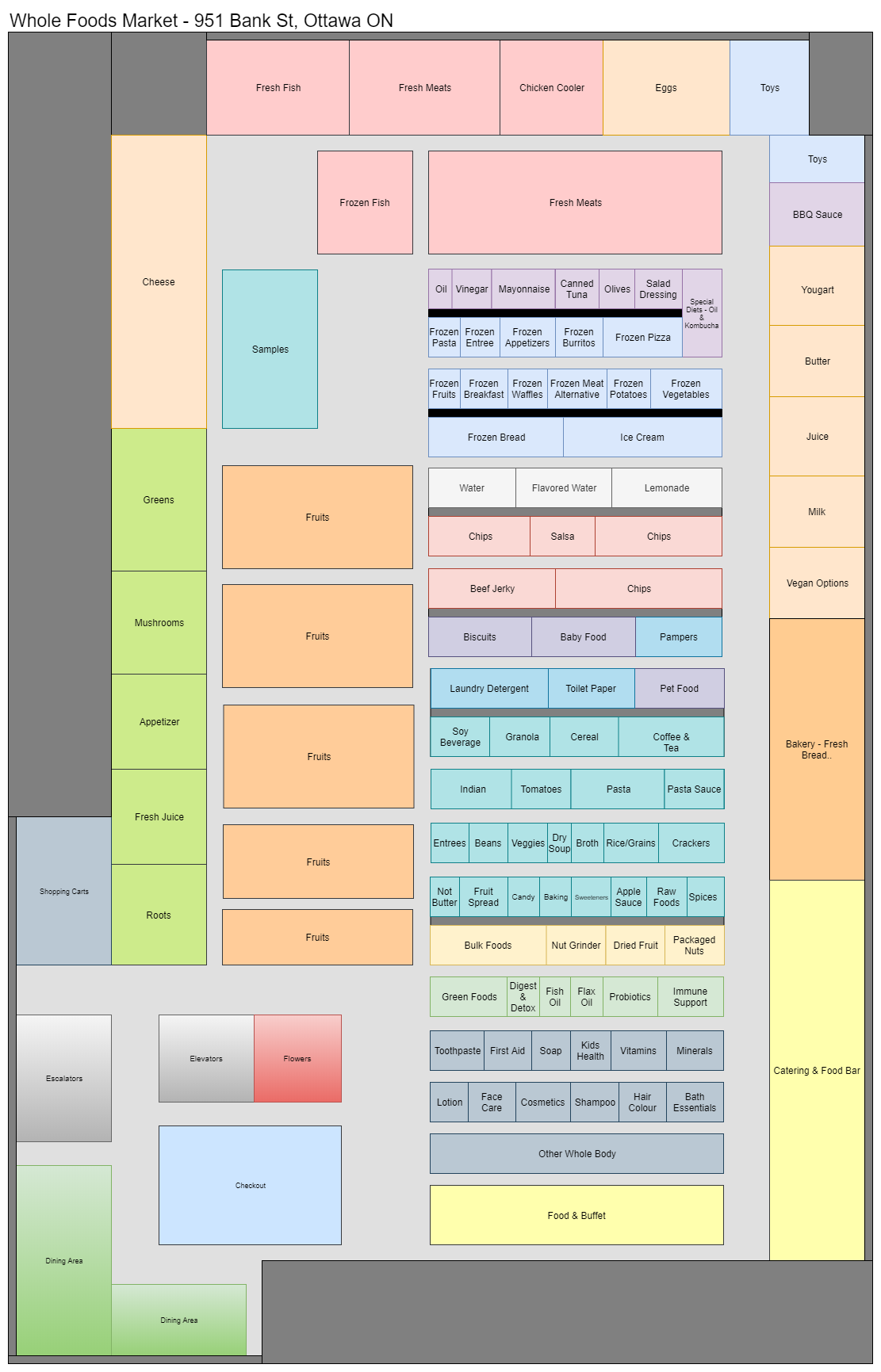


Figure 24: Sample Of Generated Floor Layout Using Collected Data

### Cloud Functions

To support the backend functionality of the application, Cloud Functions are used. Cloud Functions are functions in Google Firebase that automatically run backend code in response to some activity such as a value being changed, an item being removed, or an item being added. These functions are currently being used to help manage the backend database and to support machine learning.

To help manage the backend, cloud functions are used to update user information as they use the application. These functions use the “onWrite” method, so they are called when a specific table is updated. In this sense, the functions essentially act as listeners. These functions listen to the user’s table of created and shared lists to update that users created and shared list count as the table is updated. To use Cloud Functions for machine learning, the “onCall” method is being used. This makes the trigger for the function it being called, so for the most part, it can be treated like a normal function. To call these functions, a wrapper async function that uses await to asynchronously call the function and then wait for it to finish is used.

The benefit of using Cloud Functions is that they run in the backend on the Firebase server, which means they are faster, and the app does not need to make repeated calls when looping through a lot of data. The downside to using Cloud Functions is that they require an internet connection, which means that the app will slow down when the user has a slow connection, and certain aspects may be unusable when the user has no connection. This downside needs to be accepted to a certain degree as the application needs a lot of the database data for its core functionality, but there are some steps that can be taken to mitigate this issue. For example, the functionality that does not require a lot of the database information could be separated, so as to clearly define what functionality requires an internet connection and what can be done without a stable connection.

## Frontend

As initially planned, the team was able to build the application using React Native and deploy the app for testing using the Expo client. In order to build a working application that users can interact with, there were two major regions that the team focused on: the User-Interface Framework and the Functional components interacting with the UI.

### UI Framework

Initially, during Iteration 1, the team designed a simple UI with the basic tools and packages that were provided by React Native, but it was soon realized during testing that both the Android and iOS displayed UI components in their native fashion rather than in a standardized manner across both platforms. This inconsistency across different platforms caused several inconsistencies and issues to prop up during testing. In order to simplify the UI design process, the team decided to use a free and open source React Native framework called UI Kitten [64]. This UI framework brought forth several advantages over the base UI framework of React Native.

#### Styling and Theming

UI Kitten provided a global theming capability that allowed the team to set up multiple theme constants in JavaScript, containing a list of values representing different colours for a specific theme. Using these theme constants, the team could just set the theme property value of the root node to a specific theme constant during runtime, and all of the child components would adjust their colours to match the theme property value of the root node. Figure 26 shows two different themes applied to the same page.

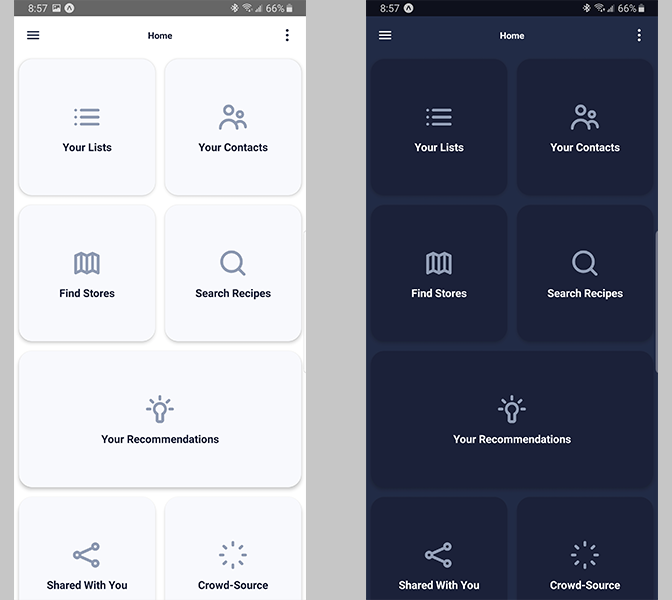


Figure 25: Light Theme (Left) And Dark Theme (Right) Applied To Home Page

The framework also allows each component, such as buttons and cards, to be styled individually. Figure 27 shows an example of pages containing multiple styled components that match the applied theme.

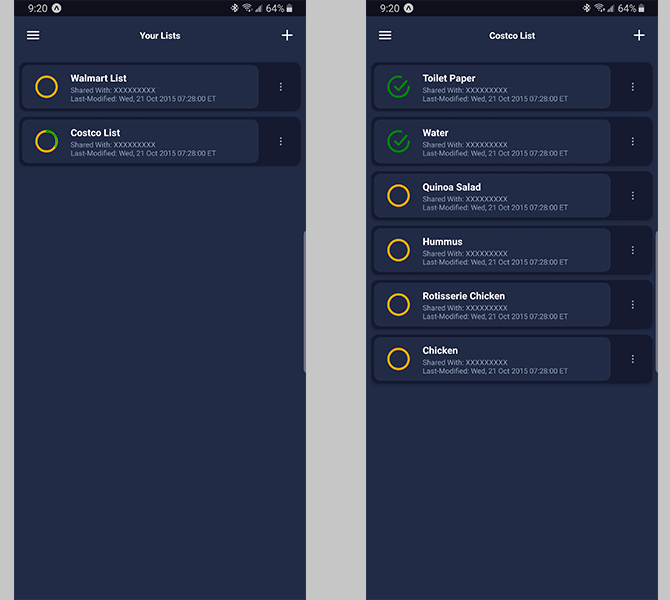
**

Figure 26: Your List Page (Left) And Current List Page (Right)

### Functionalities

In order to make the UI functional, multiple functions had to been written in JavaScript that connects the UI components with the backend of the application. Some of the major functionalities that the team has implemented so far are listed below:

* 1. Create List – Allows the user to create a new list entry in the database with a specific name.
  2. Add Item - Allows the user to add a new item entry in the database with a specific name within an existing list.
  3. Add Contact - Allows the user to add a new contact to their Grocery List App contacts using the email address that the new contact used during their account creation. Additionally, it allows the user to assign a nickname to the new user.
  4. Accept Contact Request – Allows the user to accept a request that they received from another user. The user is also allowed to deny the request.
  5. Share a List – Allows a user to share one of their existing lists to multiple selected users from their contacts.
  6. Real-time Update – The current version of the Current List Page updates the list in real-time for all shared users, allowing them to view and modify what items have been purchased.