Final Report V. 1.0

May 4, 2018

Team Name:
Gnosis Solutions

Team Members:
Kalen Wood-Wardlow, Christopher Simcox, Thomas Back, Kristoffer Schindele

Project Sponsor:
Dr. Leverington

Faculty Mentor:
Dr. Leverington

Table of Contents:

1. Introduction

Grocery shopping chains in the past few decades have offered rewards programs that allow customers to follow daily deals and shop wisely, while the store collects data about customer shopping habits. Unfortunately, there are rarely any special programs that allow the customer to benefit from collected data such as: what certain items are in stock and helpful details about those items like seasonal availability. With the exception of curbside pickup or delivery programs, there are rarely any tools for the customer that will allow them to plan and shop more effectively. For this reason, customers develop an affinity for a “familiar” store that they visit exclusively to cut down on time spent looking for items or sales. This is done without any consideration that there may be stores available in closer proximity to their location or needs.

The purpose of this document is to explain in great detail the final product that was built in response to the client’s needs and problems needing to be solved and how this product solves them effectively. This document will have multiple sections going over items such as process overview, requirements, architecture and implementation and so on to go into specific detail about how this solution solves the problems given, what were the requirements or goals of this project and how our specific solution addresses these. Finally finishing off the document with how this application was tested and the schedule that was followed to give an idea of how this project progressed over the lifespan of development. So to start things off let’s take a look at the process overview.

2. Process Overview

The process we chose to create our project uses a dynamic approach where things changed in our project as needed based on requirements or changes based on necessity. Our choice for using version control was the software git used with github for remote storage of our software and its different versions, along with branches for working on different aspects of the project. In order to organize our work we used created a google drive for our team that each member had access to where they could create, edit, or download the various documents needed for our project, this serves as a remote storage for our documents.

The team roles were split as follows:

- Christopher Simcox: Databases and Connections Lead
- Kristoffer Schindele: Front End Developer
- Thomas Back: Front End Lead
- Kalen Wood-Wardlow: Team Leader, Backend Developer

3. Requirements

Our requirements for our project are as follows:

UR1: Must be able to use the system like a grocery list.
For this requirement this system needs to offer users a way to input shopping list items into a grocery list. This means adding, deleting and modifying information about the products in the list such as quantity, name, store name and price. With store name and price being optional until they get to the store and check off the item on the list.

UR2: Data must be secure and reliable in transfer.
This requirement offers security to all users so that no person can view any other person's cart or data. The data on items and store's should only be used for easy entry and price suggestions.

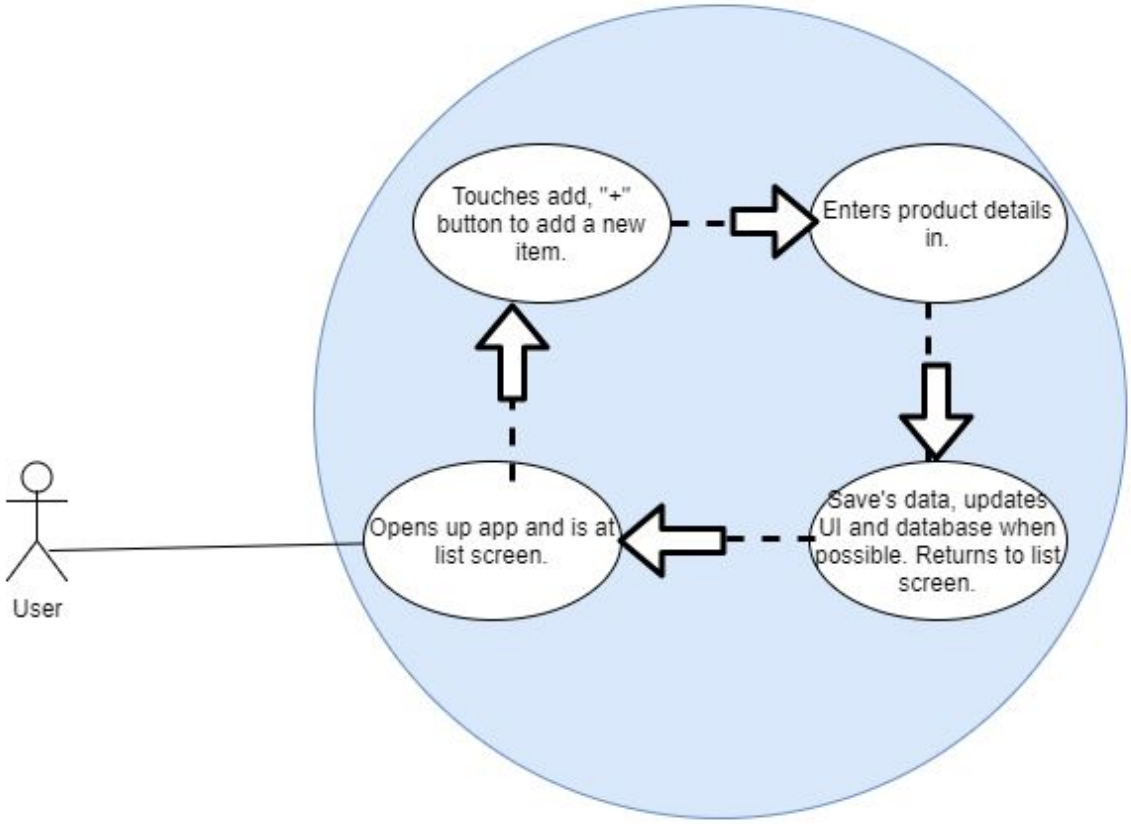
UR3: Notify users on lower price depending on store proximity for those who opt in.
This requirement gives users a notification on lowest prices depending on stores they are close to. This is to afford users the opportunity to get the lowest price on items in their cart and also the option to not have this done to not annoy any user.

UR4: Store user data in a cloud sourced environment.
This requirement covers the client's request to have data be updated in real time and crowdsourced. The data that the user's provide must be stored in an accessible place for other user's to take advantage of.

Our Use cases are as follows**:**

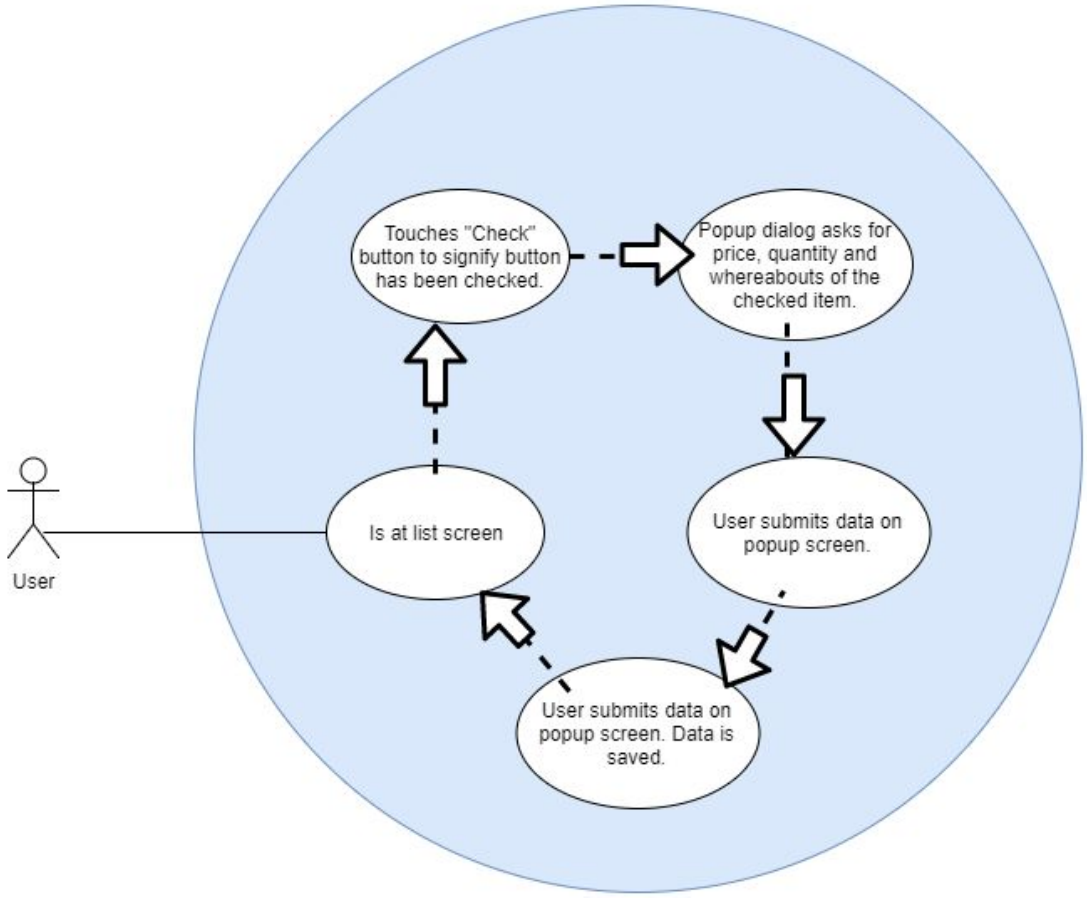
Considering the previous user requirements at this point it is important to highlight different use cases that users of the system will encounter before we get into the functional description of our system. The selected use cases are the most significant ones considering user interaction one being entering in items before going to the store and then while at the store checking off found items.

ENTERING ITEMS INTO SHOPPING LIST - Use Case Diagram



At the most basic function this is what the proposed application should do for the user. It should allow them to enter in as many items as they would like into their shopping list. This interaction happens while the user is making their shopping list.

CHECKING OFF FOUND ITEMS ON SHOPPING LIST - Use Case Diagram



This use case is when the user is at the store and will enter in more data about the item when they check it off their shopping list. This shows the circular effect that allows the user to easily provide substantial data about products and their location in very few steps.

Considering the use cases it is important to now describe how we are going to provide these use cases and all of the work going on behind the scene to make this all possible.

<u>Our functional requirements are as follows:</u>
The following functional requirements are attached to a specific User Level Requirement where “UR_” corresponds to the appropriately labeled User Level Requirement as defined previously. Using “FRQ_” as a label for the following functional requirements we will define as such. These requirements describe how the system will accomplish certain user requirements as stated above.

UR1-FRQ1: Must be able to enter product data into the application.

- Must be able to add, delete and edit the following information about products:
    - Price of the product, geolocation information such as store address, store name and number of items bought. These data inputs will be collect from easy to use text input fields, a map to which the user can point out a location to, auto collection of geolocation data from the user's phone upon approval of access, checkboxes and submit buttons to store and process the data.
- User must be able to add, delete and edit the location of a store and the name of that store in order to contribute to identifying stores.
    - This will be accomplished by allowing the user to interact with text field inputs and buttons to submit data to the system.

UR1-FRQ2: This system will provide easy to use check boxes so that the Users must be able to check and uncheck items to perform action on.
The checking and unchecking should be easy as a press of a button on the screen with touch. This will allow the user to edit, delete or signify they have picked up the items in their list.

UR2-FRQ1: This system will provide that grocery list items will not store information about whom gave information on them in database.
By not attaching metadata to items in the crowdsourced database even on a data breach information about what the contributors have bought will be given.

UR2-FRQ2: The system will provide a way to store data locally if no internet connection is available by the means of Firebase and the asynchronous API used to synchronize data between the server and all devices using the service.
If there isn't any internet connectivity on the phone then the application must recognize that and try to sync data at the next available time.

UR2-FRQ3: Data transfer will adhere to Firebase/React Native communication API.
By following the Firebase and React Native tutorials on data transfer and adhering to all notices and warnings it will be sure that the app is communicating in an encrypted and safe manner for all types of data to be passed.

UR2-FRQ4: Geolocation data from the user will not be attached in any way to any specific user instead will be stored as part of items themselves.

This requirement makes it so that we are not holding geolocation information about the user but rather about products within stores and the stores themselves. Geolocation data will be gathered in the following ways:

- Gathered when taking items off the grocery list.
- Gathered when near a common store and to check if the user is near a known store.

UR3-FRQ1: The system will notify users that our app would like to use notifications. Mention where this setting can be found if they deny to allow us access. Do not automatically assume they are ok with this because it could lead to frustrated users. Application should function otherwise independent of this feature even though this feature would depend on the other parts of the application.

UR3-FRQ2: Proximity calculation should be done on the user's phone and in the background.
Because data about store locations will be transferred but not user locations the calculations to determine nearby stores should be done on the user's phone in the background so it doesn't interfere with the operation of the device.

UR4-FRQ1: Setup a database in the Firebase platform to save data about products and stores.
This is done so that it can be cloud sourced and can be scaled if necessary. Per our client's request of attaining a crowdsourced grocery list application this is the easiest and safest route.

UR4-FRQ2: Use Firebase API for React-Native in order to establish database connection.
By using this we can ensure reliable transmission of data for future versions of React-Native with Firebase support.

UR4-FRQ3: Firebase API must allow users to login and maintain connection to the Firebase database for every user copy of the application by utilizing the API effectively. This requirement is so that every user will be able to contribute to the crowdsourced database and make sure that connection will be valid.

UR4-FRQ4: Universal database access must be available for all copies of software. This means that all copies of our application must use the save universal database access account to add to the database. This account must still work and be valid even if

a credential change is necessary in the future without any update to the application after publishing.

4. Architecture and Implementation

With the requirements in mind it's time to go over what the, requirements, goals of our project are it now here is the information regarding how the project was designed in the form of architecture and implementation to create a solution that solves all of the problems. The figure 1 will serve as a talking point that can be referenced throughout this section. Let's step into the first module which is where all functionality comes from which is the backend.

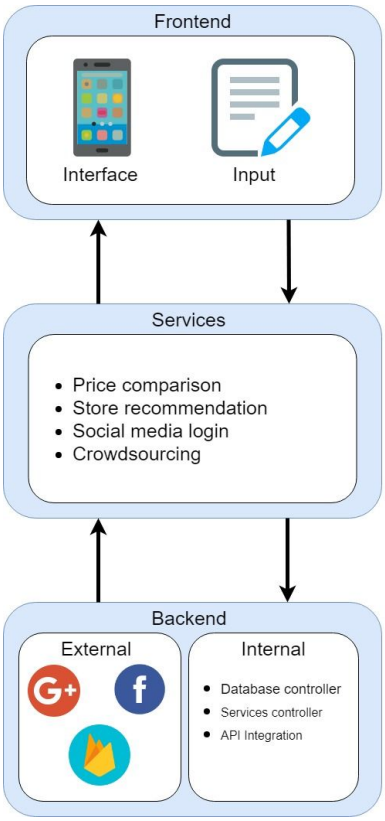


Figure 1: Architecture Diagram

4.1 Backend

There are three main modules on how this application is structured. The first is the Backend which is comprised of internal and external code and smaller modules. The external services are connected to by using the relevant API, Application Programming Interface, for each service. These are the following external services that were implemented in this project:

- Facebook login.
- Google plus login.
- Remote Realtime Firebase Database.
- React Native cross platform deployability.

With these the application is going to use Facebook and Google plus login to authenticate the user so this project doesn't have to keep sensitive data about the user. Using the Firebase Realtime database the app can store data remotely and on the device itself. Finally, the cross platform deployability is since this application uses crowdsourcing, which is a way of collecting data from all users to use it in a way to enhance the experience of everyone using the application, it is important to get the application in the hands of as many people as possible. Therefore, this application is developed with React Native that allows us to get as many users as possible no matter their platform, iOS or Android, choice. Next in the internal code logic we have the services controller which deals with the calculations and all necessary implementations for services offered to the user. Finally, in this module we have the database controller which allows the application to communicate between the remote Firebase database among these other features:

- Firebase data storage and communication with secure transmission.
- Offline datastore copy to operate offline.
- Offline and online syncing capabilities to ensure data is never lost.
- Ability to use application with saved data on any device as long as the same login is used.

With this logic it is possible to deliver not only a great working shopping list application but also one with services to make the shoppers experience better than ever before. Next it is important to go into the next module which is the services module that has the details on what services we provide to the user to ensure their shopping experience is easier.

4.2 Services

Given that the backend provides the raw logic for each of the supporting external libraries this project uses as well as the logic for providing services let's go into what services offered in this application:

- Price comparison
- Store recommendation
- Social media login
- Crowdsourcing

First, looking at the social media login, which is the first action users must take to login to our application, the application was designed to make logging in as easy as possible and since most people have either a Google plus or Facebook account a new account creation may not be required which will get the user into using the application faster than other options. Now since the easy service is out of the way let's go into the more integral services that give life to this application.

Secondly, after logging in, looking at price comparison this may sound simple at first but is far from it. In this the application will offer price comparison information about the users shopping list items at different stores to give the user an idea of which store they should shop at for the best deal. This is really important because there currently is no other service that will give information about competing prices in one easy to access location. This service is not standalone however because the application does rely on store information to categorize and to be able to calculate prices per store closest to the user. Which leads into the next service which is store recommendation.

With the Store recommendation service the application can get the users current location, upon permission, and be able to lookup known stores near the user and select the two closest stores to give information about the user's current shopping list before they go out shopping so they can get a good idea of where they should go. Which does tie back into the Price recommendation service as mentioned before. With all of these two services the user can get a good idea of where the user should try to shop to get the best price no matter if they have been to the store or not. Now since these services are covered there is one last service that powers all of these and really lends a hand to providing the data for use in the application to make it really useful.

Finally, crowdsourcing is the act of taking data that users enter, while not attributing data to individual users when using data for other services, and recording it for future reference. Such times are when the user is near another store previously entered and the application can give a recommendation because another user has found a cheaper price so the new user with the recommendation can utilize this information and use it to their advantage with the previous two services mentioned. This again is the driving force of our data pool that we use to offer all services and functionality in the application.

Now that a clearer picture has been painted about what services this application provides there is still some speculation about how the users will use this functionality. Because there is not way to have normal everyday busy people look at coding or complex user interfaces so there needs to be a easy to use and beautiful front end. Which leads into the final section of the architecture and implementation which is the front end design.

4.3 Frontend

The application is designed to extend and enhance the simple and accessible qualities of a written list through a mobile touch screen interface. This means that users should be able access the base functionalities of creating lists, entering items, and checking them off just as easily as they would with a written list in hand. As such, the application's frontend interface is designed to be clean and include intuitive visual cues to guide the user through the interface. This was accomplished by breaking each screen within the application into two major components: a header including navigation and option buttons relevant to the current screen and a scrollable list below the header that includes either list or item information.

The header is darkly colored to make it stand out from the rest of the features on the screen, and its functionality is accessed through well known icons, such as a 'plus' for adding a new list or 'gear' for user options. These buttons either produce dropdown lists of selectable options or modal windows that contain relevant data entry elements.

The lists of lists and items are colored lightly and clearly separated by borders; lists can be accessed through a touch (bringing the user to the next screen) and the list item details can be revealed by sliding the item title from right to left. The edit interface for each item detail, such as price and aisle number, can then be accessed with another touch. The application was designed this way to ensure that navigation was smooth and requires a minimal number of actions from the user. That way, they can quickly navigate between lists and update information while in the store.

The final design of the interface is significantly different from the original design and these changes were driven by extensive user testing. Feedback from beta testers in each of the testing stages revealed design decisions that prevented users from easily accessing the application's features as well as changes that increased the aesthetic appeal of the interface.

5.Testing

The testing for this project was always going to be focused on usability testing in an effort to create an application that was ‘so easy to use, they wouldn’t want to put it down’. To achieve this, most of the unit testing for functionality and integration was performed early in the development process to make way for the bulk of usability testing that was conducted during the second semester of development.

Unit testing was focused around major functionalities such as communication with the database, and updating the interface in response to new user input. In its early stages, the application was simply a single shopping list that could have new items added to it and subsequently checked off. Once testing proved this basic feature to work, new elements were integrated until the app could provide all the functionality agreed upon in the requirements document. The app was then linked to the database to allow saving list data to the cloud. This feature, as well as Facebook and Google sign-in options, were tested by logging into different devices to prove that cloud data was still accessible based on the same authentication information. Finally, the geolocational logic and dependent services were implemented and tested via live usage by team members. These functionalities proved to be the most tricky to get working consistently, and the team struggled with passing geolocational data from the module that collected it to the module that actually uses for calculations. Despite these difficulties, thorough testing revealed the source of the problem and the feature was in working order just before the acceptance demo.

Usability testing began much later in the process and was originally going to be divided into four stages to target specific aspects of the user interface: blind user interaction tests, collaboration testing, expectation tests, and user interface survey testing.

Blind user interaction tests involved sitting a beta tester down with the application and asking them to navigate its features without any prior instruction; this was done in an effort to reveal aspects of the interface that were difficult for users to learn. These tests were performed with ten beta testers both at the beginning and end of the testing process. These tests turned out to be the most telling, as many users in the first round of tests completely missed certain functionalities, such as accessing list item details. As a result, access to list item details was moved to an intuitive slide-out panel to provide easy cursory access to the user. These changes were received with positive feedback by the beta testers who had missed the feature during our first round of tests.

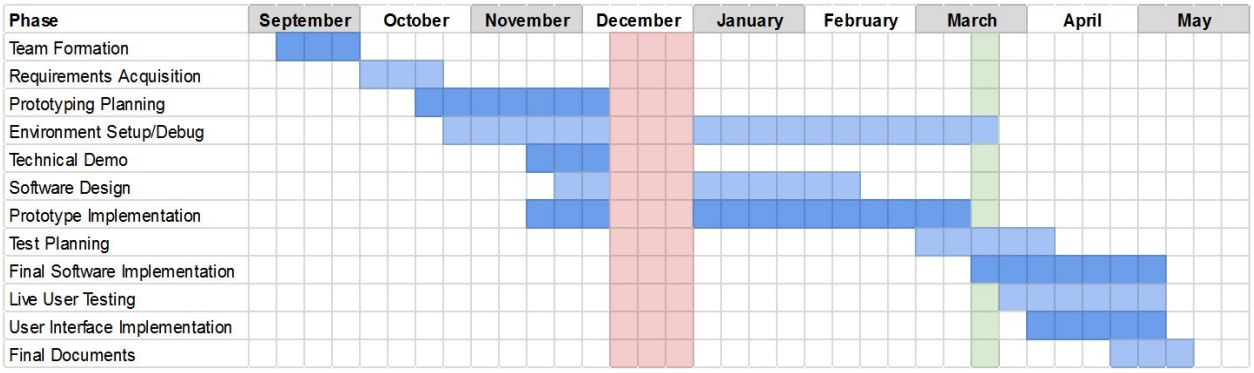
Collaboration testing was designed to field-test the crowdsourcing aspect of the application. This was to be done by following the usage of groups of beta testers that would share data and use it for no less than a week. Unfortunately, collaboration testing proved to be infeasible for two main reasons: first, most of the beta-test group were iPhone users, and second, that Apple maintains strict requirements on what software

can be added to their app store. As a result, it was impossible to get the application onto the devices of enough beta testers to make collaboration testing feasible, and the idea was dropped.

Expectation testing was performed in parallel with the blind user interaction tests and was conducted by providing a scripted demo of the app and then collecting opinions from beta testers about which features they thought to be slow or unresponsive. Surprisingly, these tests were not as informative as we expected and most users claimed that the app fell within a reasonable range of speed and responsiveness expected from mobile applications.

Finally, the user interface survey testing involved providing beta testers with questionnaires accompanied by similarly sized and styled sets of interface elements, such as button icons. Beta testers would indicate which elements they found most appealing and the aggregation of the results would drive our final design of the app's aesthetic design. Unfortunately, this type of testing also proved to be ineffective in practice. While React-Native allowed the app to be developed for both Apple and Android platforms, the minor aesthetic differences between the way they displayed the interface made it impossible to make the app look the same on both platforms. Rather than collect unhelpful data and waste the beta testers' time, Google's material design principles were used as a guide for the interface. The elements most heavily influenced by this decision were the button sizes, fonts, and the symbols used for each button.

6. Project Timeline



Most of the first semester of development was spent gathering requirements, setting up environments and making major design decisions. However, it's important to note that prototype implementation began in November, when the expectation was to begin in January of the subsequent semester. The backend team was the primary driving factor in this early development, and the database was passing data from the

app to the cloud before the end of the semester. Additionally a bare-bones frontend implementation was completed before the second semester as well.

The second semester was focused on implementing the features required by the client, and improving the look and feel of the front-end of the application. Each team member not already familiar with the existing code were also brought up to speed to ensure everyone had something to work on. Authentication services and a user interface with access to all relevant features were implemented during the beginning of the second semester. The rest of the semester was spent on usability testing and the logic behind the geolocation-based services, such as price comparisons.

As a whole, development went relatively smoothly throughout the process, and major milestones were met will minimal difficulty. This was in part caused by the allotment of additional buffer zones, such as spring break in the middle of march, that could be used to resolve any unexpected complications. In its current state, the project and last few pieces of documentation are ready to be delivered in early May right on schedule.

7. Future Work

While trying to accomplish all goals of this project there were some that were not feasible to be completed in the amount of time given. So there are some extra cool goals that, given enough time, would add more market value to this application. These ideas and goals include developing an algorithm that can create a shortest path through a store to more effectively lead you through the store in finding your items even if you have never been there before. Another future goal would be to have would be navigation capability of the application upon selecting a store to travel to given our store recommendation service as mentioned earlier. This would further ease the user's effort in their shopping experience. Now let's step into these goals with more detail. First up is the shortest path algorithm for store traversal.

7.1. Shortest Path Shopping

While having the isle numbers ordered in the user's list is handy and can save some time what would be really cool is having a way to visually draw a path through the store so the user can just follow a virtual path to obtaining the required items in the respective shopping list at a specific store. At this point it may be good to explain that store frequently change the location of items as well as if the user has never been to a store it is considerably harder to find the items needed quickly. Because of this creating an algorithm and ultimately a service that can provide this path could potentially save shoppers a great amount of time. Now while this is useful once at the store there is one other goal that would have been nice to have before even getting to the store and that is

how to get to a store the application might recommend or the user might want to visit. Which leads into the final future work goal.

7.2. Navigation to Store

Getting to a new store sometimes might be a task in itself because of construction or other obstacles. Since the application relies on user input heavily we could also crowdsource this information as well to provide the most up to date information possible about how to get to stores. This could eventually be expanded to show road closures and other hazards or even traffic flow during different times of the day to reduce travel time to and from any given store.

8. Conclusion

In conclusion current mobile shopping list applications fail to benefit the user's shopping experience because they do not take advantage of sharing information between users. Gnosis Solutions plans to solve this problem with a mobile application that makes use of crowdsourcing that feeds a shared database that is functional with and without an internet connection. This application will also include a streamlined and intuitive user interface that will propel it above competing software on the market.The lack of change in the project design since its inception is indicative of the accuracy with which Gnosis Solutions has pursued the best tools and technologies suited to create our application, which is a good proof of concept for this new type of mobile shopping application.

9. Glossary

API - a set of clearly defined methods of communication between various software components.

Crowdsourcing - the practice of obtaining information or input into a task or project by enlisting the services of a large number of people, either paid or unpaid, typically via the Internet.

Geolocation - the process or technique of identifying the geographical location of a person or device by means of digital information processed via the Internet.

10. Appendix A: Development Environment and Toolchain

The following sections give an overview of the hardware, software, and logistical requirements of developing the project.

10.1 Hardware

Among the developers, Mac, Linux, and Windows operating systems were used to develop the projects. Both Mac machines were Macbook Pro laptop no older than 2010 model. The Linux and Windows machines were a gaming laptop and gaming desktop, respectively. All development machines required at least 8GB of RAM memory and modern graphics hardware in order to effectively run emulation software required to work on the application. It is also important to note that these machines require an Intel processor to run Android Studio's emulation software.

10.2 Toolchain

Android Studio - allows access to required android software packages for android development, as well as an emulator for live testing. Also includes a code editor.

Atom IDE - used by some members as a relatively light code editor when Android Studio's editor was not used.

Node.js - lightweight cross-platform runtime engine that runs javascript-based applications.

Node Package Manager (NPM) - package manager for Javascript that is used to track and manage React-Native and its relevant base software. Also used to incorporate Facebook and Google authentication tools.

React-Native - primary framework for application development, includes base components for creating user interface elements as well as the ability to define custom elements. React-Native was used to build the front end of the application.

Facebook Authentication - a package provided by NPM that allows database authentication through an existing Facebook account.

Google Authentication - a package provided by NPM that allows database authentication through an existing Google account.

Git - All work was tracked and recorded in a private Github repository to ensure access to all team members and rollback in the case of non-functional builds.

Firebase Cloud Storage - free cloud storage platform that was used to backup all user data and store price and location data for registered grocery stores.

10.3 Setup

1. Install Node.js
2. Install Node Package Manager
3. Clone a copy of the project repository using Git
4. Run “npm install” to install all required software packages
5. Run emulator (differs for iOS and Android emulators, as well as the specific emulator)
6. Run “react-native run-android” (running android platform as example) inside the project directory. This will install the application the indicated emulator/device.
7. Open the application on the emulator/device
8. Project is now ready for development, and can be hot-refreshed within the emulator as changes are made.

10.4 Production Cycle

Example: Changing Button Text

1. Look in the source code for the relevant React-Native component.
2. Once the component is identified, change the text it uses as its label and save the file.
3. Rebuild the project using the command from step 6 of Setup *or* hot refresh the project during emulation (depends on emulation device, which provides the command)
4. Verify that the correct element of the interface was changed.
5. Add and commit the changes using git.
6. Push the changes to the repository to ensure each member has access to them.