Kennesaw State University

Department of Computer Science

CS 4850 Senior Project - Spring 2024

SP-5-Green Dynamic Grocery List

DynamicGroceryList: Grocery Item Organization and Location with Asynchronous Updates and Collaboration

Ayden Harris - Department of Computer Science, Kennesaw State University; aharr419@students.kennesaw.edu

Spencer Gerdes - Department of Computer Science, Kennesaw State University; sgerdes1@students.kennesaw.edu

Kevin Vu - Department of Computer Science, Kennesaw State University; kvu13@students.kennesaw.edu

Sharon Perry - Department of Computer Science, Kennesaw State University, Project Coordinator; sperry46@kennesaw.view.usg.edu

Project Statistics:

Lines of Code:1500
Number of components:7

1.0 Introduction

1.1 Overview

The mobile app project looks to develop a user-friendly grocery list application utilizing different technologies such as React Native, Expo Go, JavaScript, node.js, and Firebase. The project's goal is to create and simplify the needs for grocery shopping,

making it more organized, and efficient for consumers. Some of the objectives include creating an intuitive interface where users can easily add and modify their shopping lists and be able to share the shopping lists with their family/friends in real-time. Users will have the ability to view up-to-date prices, sale prices if applicable, and various other information about items by utilizing Kroger API. This will prove to be useful if users and families are wanting to budget and make a more sophisticated grocery list.

1.2 Project Goals

Create a dynamic web application for a grocery list. Users will be able to add and remove items from their list in real time. They can share the list with friends/family members so they can also see it in real time. Consumers are also able to look up items to see where the item is inside the specific store, by using Krogers Product API. This will retrieve the exact item location- its aisle location, shelf, and tag number.

1.3 Intended Audience

This document is aimed at the potential end-user of the application. Users who regularly grocery shop and enjoy planning out their trips to the store will be attracted to an application of this sort, as this application aims to provide the ability to organize their shopping list and even enhance their in-store visit.

1.4 Definitions and Acronyms

- Application Programming Interface - (API)
- JavaScript - (JS)
- Database – Collection of all information monitored by this application

1.5 Assumptions

- Users have a high-speed internet connection to share and synchronize grocery list in real-time
- Users have sufficient storage space and a compatible operating system
- Firebase services will be reliable and available for real-time update of grocery list and user login authentication

- Kroger Services are available and up to date with accurate item information

1.6 Project Scope

Phase 1 of this application will involve creating a mobile application that will allow users to create and share a dynamic grocery list. The application will be developed through React Native, making it compatible with Android and iOS. The application will connect to a server backend – Firebase. Key features of Phase 1 will include real-time updates, user profile support, and creating and editing a dynamic grocery list.

Phase 2 of this application will expand upon some of the initial features. Kroger API services will be utilized to retrieve item information: including price, image, location in store, and a description for the item. This information can be displayed within the application when a user searches for an item. Kroger's API also displays information about the product's price, including details about sales and special offers.

2.0 Overall Description

2.1 Product Perspective

This application is a standalone application that aims to create a convenient and efficient grocery list management system for users via mobile devices. By providing a user-friendly interface accessible on iOS and Android platforms, the application will be available and functional to a wide variety of users. The application will feature an interactive scrolling list that will make for an intuitive and seamless experience for the user. The application focuses on providing real-time updates to the user, where the user will be able to seamlessly collaborate with other users who are using the same grocery list, showing edits and updates instantaneously. The application will have a secure login protocol prioritizing user security using Firebase to ensure that login information and storage of grocery lists is protected against unauthorized access.

2.2 Product Functions

The application is not limited to, but will include the following features

- User Profile Support
- User Login
- Ability to create, edit, and delete grocery lists
- Ability to collaborate with other users and share the list
- Search Items and add them to the list
- Updates to the list will update in real-time

- Sort items in list based on their location in store, giving the user an efficient route to take within the store

2.3 User Characteristics

The application will have two types of users- a list creator role where a user will “own” the grocery list and will have the ability to share, update, and delete the list, regardless of permissions. The creator will also have the ability to give other users on the grocery list different permissions, such as being able to edit the items on the list, or viewability. Any user can create a list and become the owner, but other users on the list must be granted permission to view and edit lists.

2.4 System Design

Our application will run several different technologies. The front-end framework we chose was React Native. We chose react native because it does not use different codes for iOS and Android platforms. It uses JavaScript, which is compatible with both platforms. With React being our framework, JavaScript is going to be our language for the frontend, making it seamless to integrate with one another. Our backend is done through Firebase for the database management and authentication and the Kroger Product and Location APIs to retrieve product information.

3.0 Functional Requirements

3.1 User Login

3.1.1 Description

The user can log into the application using a valid email address and password. This data will be stored in a database, where all data pertaining to the user account will be stored. The application will also have support for resetting a password. The user will be authenticated upon login.

- 3.1.1.1 The system must allow users to input their information.
- 3.1.1.2 The system must allow users to sign in if the information is correct.
- 3.1.1.3 The system must allow users to reset their password if the user decides to update/reset their password.
- 3.1.1.4 The system must securely store a user's login information and authenticate upon login

3.2. Lists

3.2.1 Description

The user will be able to create and modify a grocery list and its attributes. The list will also be updated in real time if any changes are made to the list description or the items in it.

- 3.2.1.1 Users shall be able to create new grocery lists, creating a name and description.
- 3.2.1.2 The system must allow users to edit existing grocery lists, including updating names or descriptions, and even deletion.
- 3.2.1.3 Users shall be able to collaborate in real-time on shared grocery lists.

3.3 Profiles

3.3.1 Description

Users will be able to create a short profile that will consist of their name, email address, and Zip code.

- 3.3.3.1 Users shall be able to create and edit a profile name
- 3.3.3.2 Users shall be able to enter in their Zip Code, which will determine which store location to use for item information

3.4 Permissions

3.4.1. Description

Users will have different permissions- a user who creates a grocery list will have more permissions than a user who is added to the list. The owner of a list can modify other users permissions, such as viewability, ability to edit, and more.

- 3.4.1.1 Users shall be able to share a grocery list with other users, grant permissions such as editing or viewing.
- 3.4.1.1 Users shall be able to edit existing members permissions on a grocery list

3.5 Item Discovery

3.5.1 Description

Users will be able to search for items to add to the grocery list. There will be a search bar at the top of the screen where the user can input text and search for relevant items. Users can tap on an item that appears, and a screen will pop up with relevant information such as the item description, image, price, and location in store.

- 3.5.1.1 Users shall be able to search and add items to a grocery list by searching in a search bar, where relevant items will appear and be available to select.
- 3.5.2.2 Tapping on an item will pull up a screen that will show the item name, item description, item price, and item location inside the store.

4.0 Non-Functional Requirements

4.1 Security

- Application will authenticate and authorize users and protect user login information
- Transmission of user login data and grocery list data should be encrypted using HTTPS and OAuth2 from Firebase to prevent tampering or unauthorized access

4.2 Performance

- Application should load within a reasonable timeframe
- Response time for navigation and switching screens should be near instantaneous
- Application should perform well on a variety of mobile devices, including older model mobile devices with a compatible operating system

4.3 Usability

- User Interface should be intuitive and easy to navigate, requiring no training for users to use
- Text and graphical elements should have sufficient readability for easy user reading
- The user will be presented with uniform iOS and Android application behavior, promoting a familiar experience and predictability with interactions

4.4 Other

- Application should provide informative error messages in case of unexpected errors
- Error logs should be generated and monitored to identify issues

5.0 External Interface Requirements

5.1 User Interface Requirements

- Intuitive and user-friendly interface designed for smartphones
- Support for touch gestures including tapping, swiping, and pinch-to-zoom for interaction.

5.2 Hardware Interface Requirements

- Mobile devices running a compatible operating system and having a working touch screen
- Computers with monitor display

5.3 Software Interface Requirements

- Various Web Browsers are compatible, such as Edge, Chrome, Safari, and more.
- iOS and Android operating systems are supported

5.4 Communication Interface Requirements

Kroger API for item information, price, location and image
Firebase services for User Information storage and authentication
HTTPS Authentication for user login

6.0 Design

The Software Design for the grocery list mobile application outlines a detailed framework that defines the functional and non-functional requirements, system architecture, and design specifications critical for the development and successful deployment of the application. Our application utilizes JavaScript and React Native for both the frontend and backend development, with Node.js powering the server-side operations and Firebase as the database, as well as what handles user authentication.

6.1 System Architecture

6.1.1 Logical View

The logical view of the grocery list can be structured into three sections: Presentation, Business and data access. The presentation is created in JavaScript, focusing on

delivering a dynamic and user-friendly interface. The business logic powered by Node.js, handles core functionalities and integrates with external services, while Firebase constitutes the data access layer, offering robust data storage and real-time synchronization. This architecture supports efficient development, scalability, and a seamless user experience, aligning with the application's objectives.

6.1.2 Software Architecture

Users that are running operating systems: iOS and Android will be able to use our application. The application will run on a Reactive Native framework and JavaScript so that it is easy to integrate with both iOS and Android. Our data will be stored in a backend cloud computing service platform. The frontend, developed in JavaScript, communicates with the backend through RESTful APIs, ensuring a seamless data flow and real-time updates between the user interface and the server, hosted on Node.js. Firebase serves as the centralized database providing scalable solutions for data storage and real time synchronization across devices.

6.2 System Design

Our application will run several different technologies. The front-end framework we chose was React Native. We chose React Native because it does not use different code for iOS and Android platforms. It uses JavaScript, which is compatible with both platforms. With React Native being our framework, JavaScript is our language for the frontend, making it seamless to integrate with one another. For our backend environment, we chose Node.js because of how seamless it was to integrate with JavaScript and how compatible it is together. Our backend language is also going to be JavaScript. For our database we decided to go with the Firebase Realtime Database, because it is an ideal option to reduce the deployment time, enhance app development, and have seamless hosting. Firebase makes hosting information online very easy, and abstracts away much of the difficulties of having to create a database, create an api or SQL into the database, and host a database. The database will contain various tables to store information such as user account information, login information, grocery lists, and list members. Additionally firebase handles our authentication, featuring an API for registration and login. Lastly, we are incorporating Krogers Location and Product API’s within our application. These API’s are public access, once a developer account is created through them, and grants us access. Both the Location and Product API’s serve

the purpose of providing us Product information within Kroger stores. When a user wants to search for an item, the search will be queried through Krogers Product API and will return information back into the application that the user can see.

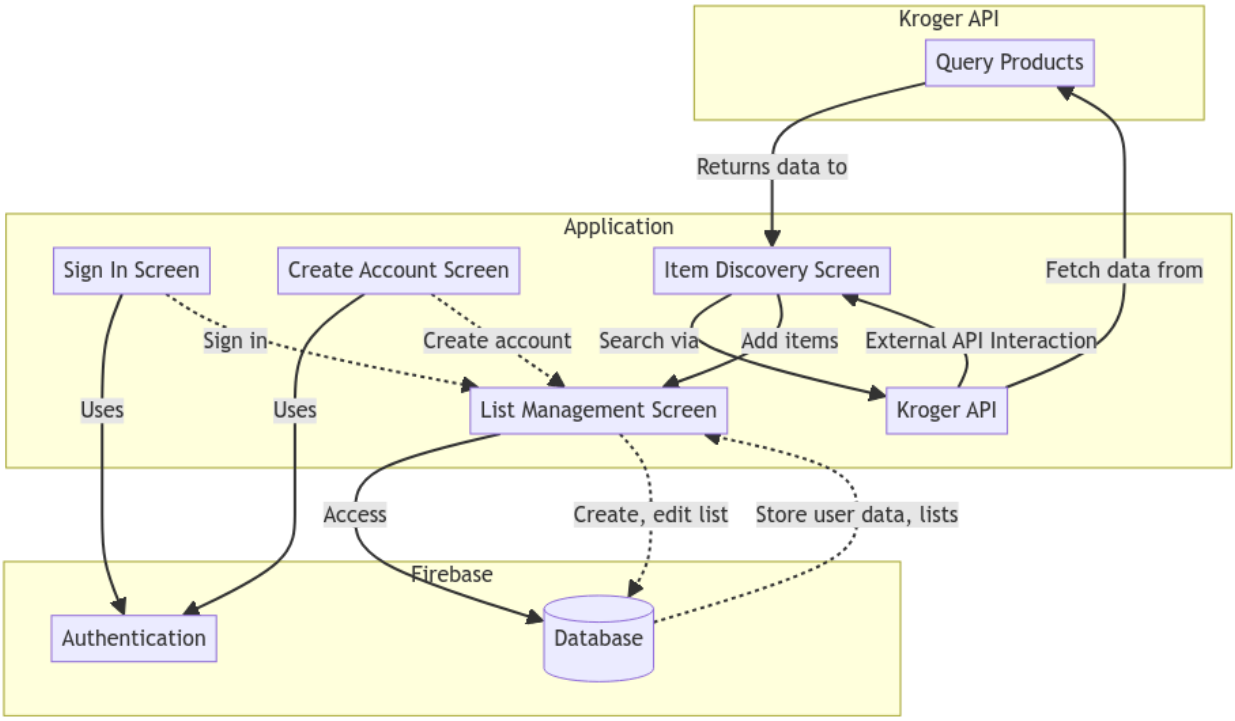
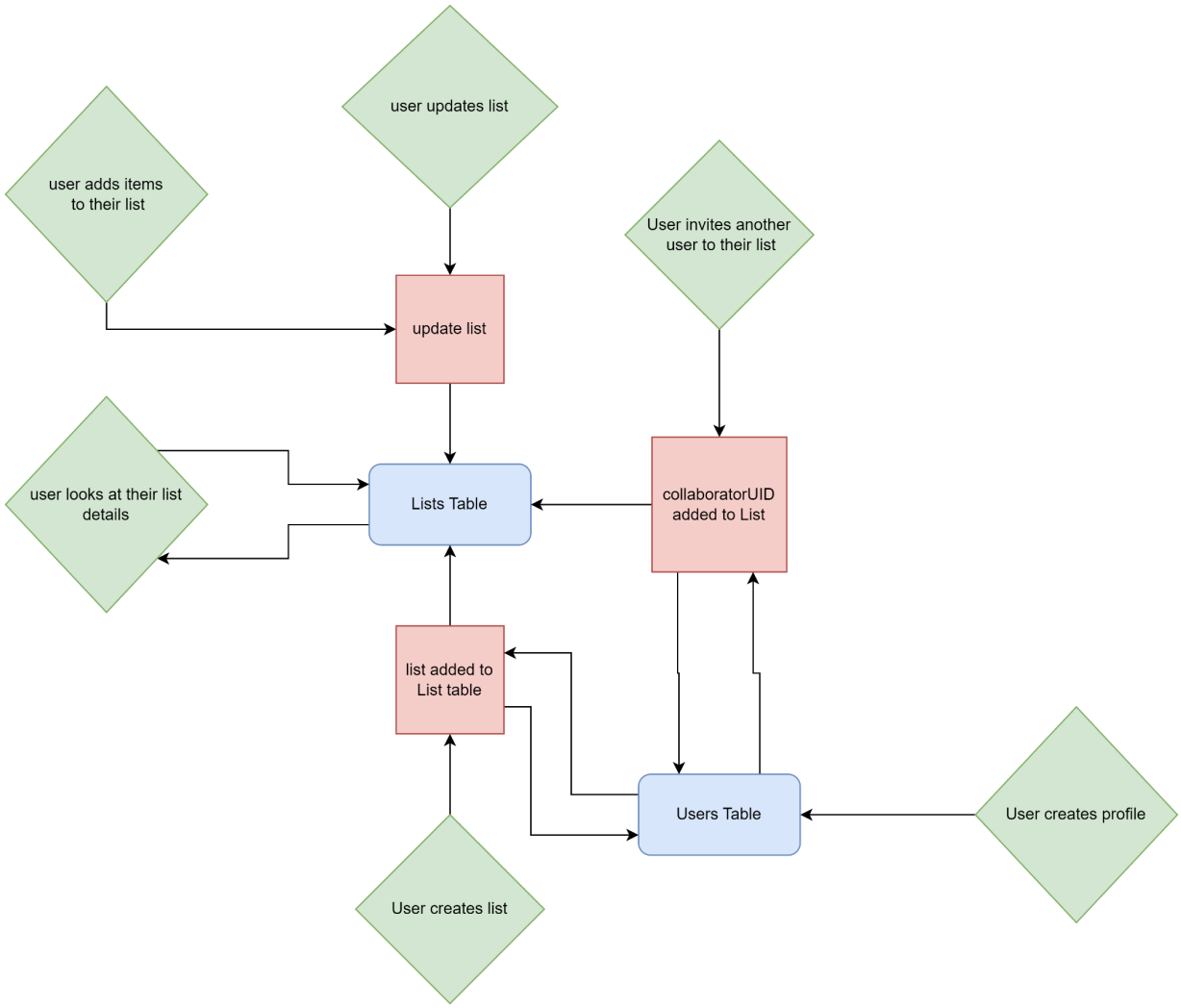


Figure 1.

6.3 Database Design

6.3.1 Realtime vs. Firestore

For our application we went with the firebase Realtime Database instead of the Firestore Database. The Realtime Database is easier to work with, storing and returning data as one big JSON tree that is able to accept rules that govern the database. The Firestore database is easier to scale large projects but ours requires very little data to be stored, as we really only need to store the list's information. Additionally, with our limited querying and the fact that we insert data into the database more than we retrieve it, the Realtime Database allows deep queries to happen easier than Firestore, which is better at shallow queries. In our application, storing into the lists table, into a list, then into the items list means that our app will almost always be writing to the deepest part of the database. Additionally the Realtime Database is time-tested and is known for its constant uptime and extremely low latency, while the Firestore database is a newer service and has a higher latency than the Realtime Database.



Users Table(not for login):

- Data Elements:
    - UserID: String. Length: 768 bytes
    - Email: String. Length: 768 bytes
- Source: User registration and profile management.
- Validation Rules:
    - UserID: Alphanumeric, auto-generated
    - Email: Valid email format, unique
- CRUD: Create, Read, Update, Delete.
- Data Stores: Users database table.
- Outputs: Data Elements (All the user profile information)
- Aliases: None.

- Description: This table stores the information related to users profiles. Does not store login information.

Lists Table:

- Data Elements:
    - ListID: Integer, Primary Key, Length: 10
        - List Name: String, Length: 768 bytes
        - creatorUID: String
        - collaboratorIDs: List
            - collaboratorID: String. Length: 768 bytes
        - Items: List
            - itemID: String. Length: 768 bytes
            - frontImage: String. Length: 768 bytes
            - Name: String. Length: 768 bytes
            - price : String. Length: 768 bytes
            - productId: String. Length: 768 bytes
- Source: User-created grocery lists, items from Kroger's product API.
- Validation Rules:
    - ListID: Numeric, auto-generated
        - List Name: Alphanumeric, not null
        - creatorID: Numeric, from user’s unique ID
        - collaboratorUIDs: Alphanumeric
            - collaboratorID: Alphanumeric, auto-generated.
        - Items: Alphanumeric, auto-generated
            - itemID: from krogerAPI
            - frontImage: url to image from krogerAPI
            - Name: from krogerAPI
            - Price: from krogerAPI
            - productId: from krogerAPI
- CRUD: Create, Read, Update, Delete.
- Data Stores: Lists Database table.
- Outputs: Grocery list details.
- Aliases: None.
- Description: This table stores the users’ created lists and their relation to other users. This also stores the items within the users’ list and the item’s associated data.

6.3.2 Data Retrieval

The backend infrastructure of our application relies on Firebase Realtime Database and Firebase Authentication for efficient data management and user authentication. The RealTime

Database serves as our primary storage and allows us to store and retrieve JSON objects in real time. We use Firebase's JavaScript SDK to allow our frontend to interact with the database and to allow real-time updates on the data displayed. We use Firebase's database references to access specific nodes with "ref" and perform operations such as reading with "get" , writing with "push" or "set" and listening for changes with "onValue". In our ListDetailsScreen we dynamically fetch and display the list items based on the selected list's ID. We set up listeners(the "onValue" function) to update the displayed items whenever changes occur in the database.

User authentication is done through Firebase Authentication, ensuring secure access control to our application. By integrating Firebase Authentication into our frontend we enable users to sign in securely using email and password through firebase. Firebase provides various methods such as: "signInWithEmailAndPassword" and "createUserWithEmailAndPassword" that we use in our CreateAccountScreen and LoginScreen to register and authenticate users. In our ListScreen we filter the lists based on the authenticated user's role. Lists are filtered to include those created by the user or where the user is listed as a collaborator, allowing the user to collaboratively edit lists they have not created.

6.4 User Input

6.4.1 User Information

The user will provide basic information, such as their first and last name, email address, and Zip Code. The Zip Code is crucial, as the Zip Code the users provide will determine what store locations the application will use when items are searched.

6.4.2 Item Search

The user, once navigated to the Item Discovery Screen, will have the ability to make an item search query through Kroger Product API. Using the users search query, the application will construct a valid API call to retrieve products and relevant information.

6.5 Application Output

- Users will be able to see what items they have added into their grocery list along with whomever they have shared the list with.
- Those who have a shared list will be able to add/remove the items in real time so everyone on the list can see the updated list.

6.6 User Interface / User Experience

6.6.1 UI Requirements

The application must follow some basic design principles. The interface must have clarity and efficiency, meaning that the interface clearly depicts the function to the user, and has minimal steps to perform a certain action. The application follows a consistent visual and interactive pattern, through the bottom-screen navbar and its icons to navigate pages. The main layout of the application consists of a light-gray background, with blue accents on the icons and buttons for easy visibility. Icons are used for navigation between screens, and every screen will also have a back arrow in the top left of the screen allowing the user to go back to the previous screen.

6.6.2 UI Prototyping

To visualize the UI requirements, we utilized Marvel to prototype our application. This website allows us to create quick screen mockups of how we want our application to look visually. This tool allowed us to test various color schemes and fonts, to determine what the best option would be when developing the application.

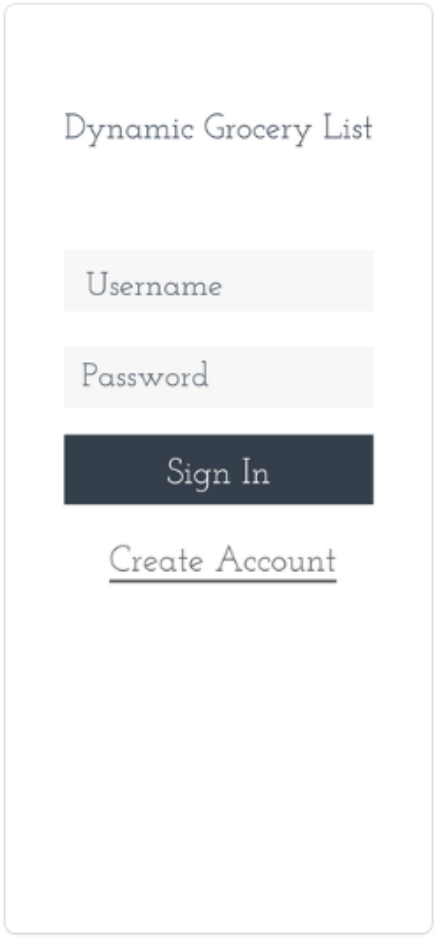


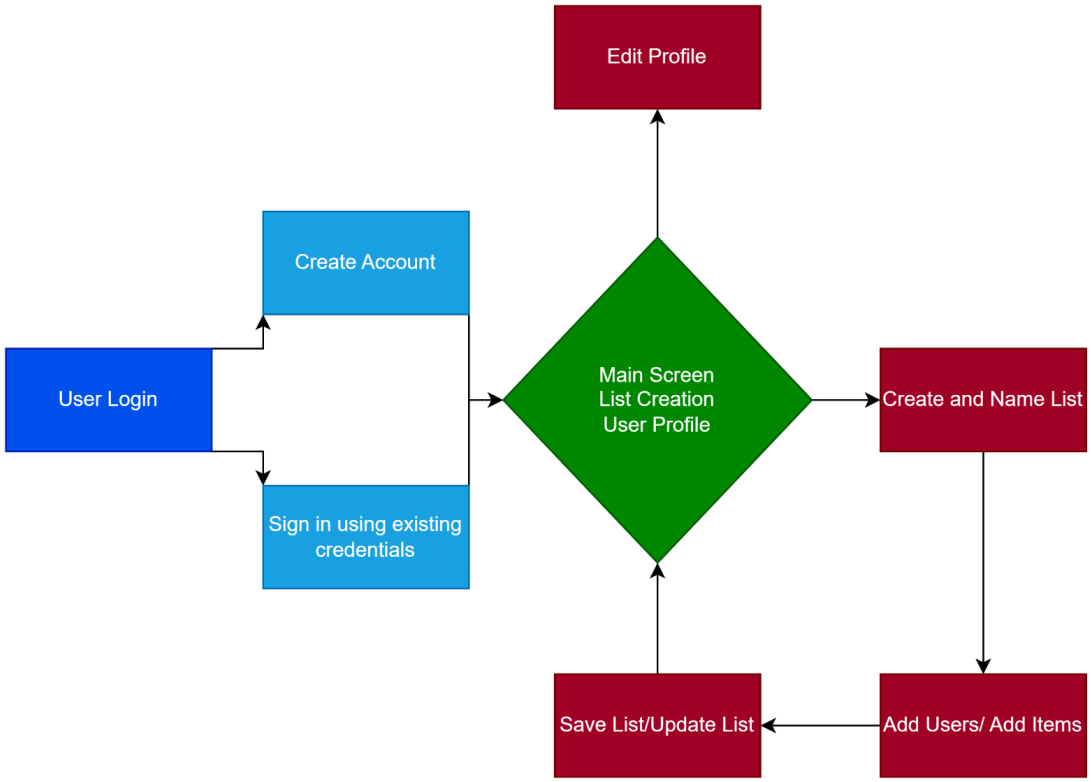
Figure 3.

6.6.3 Navigation

Navigation is handled in a specific manner within the application. There exists a Navigation.js file which imports every single screen in the application, and creates a stack of all the screens. We reference this stack of screens in the code on the individual screens, and with our icons that redirect users to specific screens.
Upon loading the application, the user will only be able to access the login and registration screens, but once logged in, the user will be presented with a header and bottom-bar navigation that presents a multitude of icons. Each of these icons in the bottom bar will redirect the user to a specific screen: the home icon will return the user home, the list icon will send the user to the list's screen, the magnifying glass will take the user to the Item Discovery screen, and the person silhouette will redirect the user to the profile screen. The header will include a bell icon to see alerts, and an open door icon that will allow the user to log out from any screen. When accessing sub-screens, such as the list management screen, the user will be able to backtrack screens through the back-arrow in the top left of the screen, and also navigate to any screen through the bottom bar.

6.7 Process Flow

The application process flow is simple. The application begins with the user login screen, where the user will be prompted to enter in their email and password or create an account if they have not already. Then, the user will be redirected to the main screen. On the main screen, there will be assorted options available to the user. From here, the user can edit their account information by tapping on the user icon, manage and open an existing list, or create a new grocery list by tapping the “+” button. The user will be able to create and name a list, then add users they want to share the list with and add items to the list as well. After, the list will update in real time with all users who are on the list, when modifications are added to the list.



6.8 Error Handling

The application has a variety of measures to handle errors in a less convoluted manner when the user encounters them, as well as measures to prevent the application from crashing.

6.8.1 Firebase Authentication

The application will recognize if the user has entered an invalid email address, or incorrect email and password combination. These errors are presented to the user in a small screen pop up, indicating that the entered login information is incorrect. Firebase will return an error code to the applications console log as well.

6.8.2 List Management

The application will recognize if there is an error when creating a list and adding it to the realtime Firebase storage. The error code is simply returned to the applications console log, and the user will see a brief pop-up indicating that there was an error adding the list.

6.8.3 Fetching Products

The application will throw an error to the user and to the applications console log if there is an issue fetching products through Kroger API. The application will simply have a pop-up with the error code returned by Kroger API on screen, and in the console log.

6.9 User Authentication

In our application we used Firebase Authentication to authenticate our users. We had the choice of selecting many different authentication methods, but we chose to go with the email and password login. Choosing this option, we manually created a list in the Realtime Database to store the user's email. In Firebase, all created users are handled through Firebase Authentication, not the database. When a new user is created, it is assigned a unique identifier specific to that user. We also store that in the user table because when we query lists we often filter them by the lists that the current user has. The Realtime Database has deep queries where it gets all the data in the current node, so when we are filtering by the user's unique ID, we also get the data for the user's email, which we can then display to other users.

On our CreateAccountScreen we import the Firebase authentication state (which validates and links our database to our app). We handle registration through Firebase by first getting a snapshot of the database and checking for the email they want to register with. If the email exists, we let the user know that the email already has an account in our app. If the email does not exist we call the Firebase function: "createUserWithEmailAndPassword(auth,email,password)". We take the user input from the text boxes and use them as the parameters for the createUser function. This is a call to firebase that will create a user and link it with our app. Once the creation is

successful, (we wait until it is or we get an error) we use the credentials the user entered along with the uniqueID and store it in our database for easy use. Any errors are caught along the way, and notify the user. For instance, if we get a Firebase error, we return firebase's message (such as "incorrect email format").

On our LoginScreen almost the same process happens, except we call the function: signInWithEmailAndPassword(auth, email, password). This is a call to firebase where it checks our authentication config to look at our application and see if the user has entered the correct credentials for an account. Just like in CreateAccountScreen, when an error occurs it is handled appropriately and the user is notified of the issue. On successful sign in we do not store and more user data, and instead set the user variable to the authentication state of the user that just logged in, and move the user to the HomeScreen.

In the HomeScreen we receive the user's authentication from the LoginScreen/CreateAccountScreen. This makes it easier to pass it along to the navbar at the bottom and use the user's unique ID and email for filtering lists and adding collaborators. On the top-right of each screen we also have a logout button that when the user presses it, calls the Firebase function: signOut(auth). If the logout is successful we move the user back to the WelcomeScreen.

7.0 Testing

7.1 API Testing

Kroger Product and Location API were tested using the Postman API testing application prior implementation in our application. The Postman application suite allows us to simplify work relating to using and testing APIs, with tons of features that are specific to making certain calls to API's, such as testing token retrieval and data retrieval using our token. We used Postman to test if we could successfully generate an OAuth2 token using our client ID and client secret provided to us from Kroger once we created an account.

7.2 Desk Checking

In our program, because we individually worked on atomic tasks, our testing was done mostly through trial/error desk checking. By using Expo and terminal outputs, we were able to see if our code was returning items in the ways we expected. In our Welcome screen it was very easy to test the functionality of our buttons and UI because Expo allowed us to download our app and run it ourselves. In the Login and Register screen

we were able to test the firebase connectivity by using the firebase console where we could create users manually as well as see all the users that were created through the app. Next we were able to use the firebase console to check the database and manually add lists under certain users. By doing this we were able to test the UI for the List screen to ensure that we would be able to test the future functionality for Lists and the firebase backend. We tested the Kroger APIs as stated above, but the way we tested the implementations in our app was by first creating a file that implemented the logic to reach out with the API. –index.js here— . Once we ensured that we could create an API request to kroger successfully, we then set about modularizing the file. By doing that, we could look at implementing different parts within our app. Once the ItemDiscovery screen was created, we verified the UI for the search bar worked through Expo and the terminal. Once that was completed we could implement the function to call the Kroger APIs, and again we verified this through the console by printing the entire return for the query in the terminal (including the parts of the API return we did not use). Once we verified that we could search with our app and have it generate a valid query to the Kroger APIs we implemented the flat list that showed the items to the user. This was again tested in Expo. –more here–

8.0 Version Control

8.1 Github

Github is the platform used for version control. There is an organization named after our application that holds the codebase for the application, and allows each member in the organization to make code changes, and pull/clone the code. In Visual Studio Code, each of the team members have linked their Github Accounts, allowing each member to make commits to the code repository from the text editor itself, and also pull newer code changes if a user is behind. Through Github, the team has the ability to publish builds on Github and establish a version number on it. There is also the ability to view every single commit made to the codebase, which allows us to backtrack or revert commits if a change causes issues afterwards.

9.0 Development

9.1 Introduction

Our basic approach to this application would be to complete certain functionalities each at a time- very similar to how a sprint would work in Agile. We also held weekly meetings that lasted anywhere from 30 minutes to 1 hour, depending on what we needed to discuss and also completing some work in real time.

9.2 Technology Selection

For creating a mobile grocery list application, there were various different technologies available for the team to implement. For the mobile development framework, there are two primary options available to use- Flutter, or React Native. They both are hybrid and cross-platform, meaning that the frameworks are compatible across iOS and Android which is important for goals. However, Flutter is written in Dart, and React Native is written in JavaScript- for us, JavaScript is a more familiar language so it made more sense to choose a framework that we would be more familiar with. This would ultimately save working hours in the long run. React Native, combined with a sandbox known as Expo Go, would allow us to easily test, make changes in real time, and bundle the application on a variety of platforms. Flutter lacks Expo Go support, so it would also be more difficult to test and bundle apps using that framework. For our backend, we chose to use Firebase Realtime Database. Firebase provides a multitude of features that prove useful for this application: A real-time database storage, user authentication using OAuth2, and various analytics and performance metrics. Instead of having a separate database and user authentication flow, Firebase provides all of that in one, free package. The realtime database allows us to store user information, create grocery lists, or just about anything we would want to store, as long as we create the table for it. The realtime database will also update in realtime using asynchronous calls, meaning that any changes to the data will instantly be updated on the users end. Firebase also has secure OAuth2 user authentication, which is implemented in our applications login screen. Lastly, while the application requirements simply included grocery list functionality, we wanted to take this a bit further. This is where we chose to implement Kroger's Product and Location API's into our application. Instead of simply creating a list, essentially just putting plain text into a file and saving it, we wanted a more intuitive and focused user experience. Kroger's Product API will allow the application to query items and relevant information and return it to the application in the form of a JSON object- where we can display information such as price, images of the product, and even the location of the item in the store using the Kroger Location API. This overall, will create a more realistic user grocery shopping experience, and reduces the need to potentially use other applications to supplement a person's grocery shopping needs. These API's have a very high data rate limit, and are free to use after creating a developer account with Kroger.

9.3 Challenges

Overall, with the technologies we chose, we ran into minimal challenges and hiccups in the development process. With React Native as our framework, written in JavaScript, we needed to complete the React Native tutorial to familiarize ourselves with how to use features such as components, screen navigation, and debugging. This required a short

refresher on JavaScript fundamentals, but not much time was needed here. Getting React Native setup within our IDE of choice, Visual Studio code, was a slight pain, as we needed to install various libraries and supporting dependencies through the terminal, and ensure the correct versions were being used. Once this was finished, we lastly needed to get the Github extension for Visual Studio Code. This would allow us to submit code pushes and pull latest changes, all within the IDE. There were some troubles getting this setup- as code would be sometimes pushed to the wrong branch, or not even seeing our organization in our Github. However, with enough trial and error, we were able to get this setup on each team member's machine, making code collaboration much more streamlined. Firebase and expo were both difficult to work with initially as well. There were many times where firebase would simply return an error with no message because expo cannot see firebase's errors unless you use React-Native-Firebase instead of React-Native + Firebase. This made debugging incredibly difficult and led to placing many console.log statements in our code to ensure we were getting the data needed before an error. Arguably the biggest challenge we faced as a team, was figuring out just how Firebase and Kroger API data was navigating or being transferred between screens. The errors the IDE generated would say something along the lines of "property e doesn't exist" We use user information from certain screens, and then through nested screens, and sometimes needed information would not transfer between screens. Ultimately we resorted to just calling the database each time we needed to retrieve data for Firebase, and then after redoing our overall app navigation, the error for Kroger Product Data seemed to have been resolved.

9.4 Performance

The application is in nature very lightweight, using small icons for bar and header navigation, as well as our home screen, and just simple colors for the UI in the app. Therefore, there are minimal assets bundled along with the application source code. The application performs as expected across all tested platforms: including iOS, Android, and Chrome web browser. There are no hiccups or slowdowns anywhere as far as application performance. There may be some latency between the application and Firebase database if the user internet connection is slow, otherwise sending and receiving times are near instant.

9.5 Roadmap

As mentioned earlier, we followed a similar approach to Agile sprints when creating the application. As a team, we focused on one aspect or functionality of the app before moving onto the next. At the start, we focused solely on implementing our UI mockup, creating all the necessary screens the app would need and the basic design and buttons. There would be no functionality yet, we would just create the skeleton for the

app. Once navigation was ensured to work between screens, we then worked on user authentication. This phase would involve implementing Firebase, and ensuring that user information entered from the Login Screen or Registration Screen would properly save to the database, creating a unique user ID, and saving email address. Once the application would properly register, authenticate users, and store the necessary information into the database, we would then work on implementing the Kroger API into the Item Discovery page. This is where we would ensure that the application would be able to reach Krogers Product API endpoint. Once the application had the ability to retrieve Kroger Product Objects, we would then add the functionality to store these objects within a user created list. We chose to order the development like this, so that we could thoroughly test each part of the application fully, and that later functionality would build off of past work that we know works. There would be minimal backtracking and working on past features with this approach. Since we did these in 2 week phases, and all members focused on working on the same features, we completed development in a timely manner, giving us sufficient time to discuss it and future ideas.

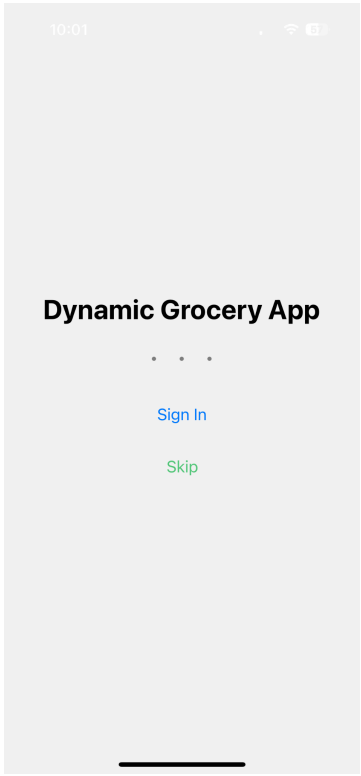
Appendix

React Native Tutorial

Completed by: Spencer Gerdes, Kevin Vu, Ayden Harris

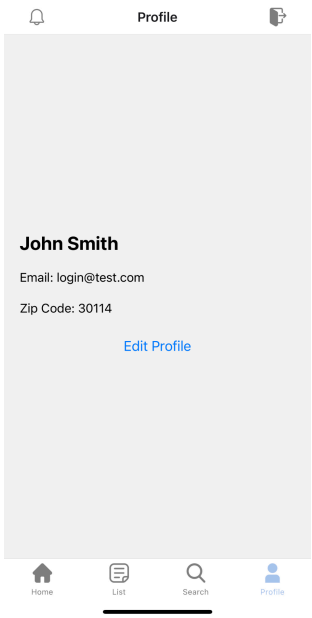
Screen Mockups

Welcome Screen

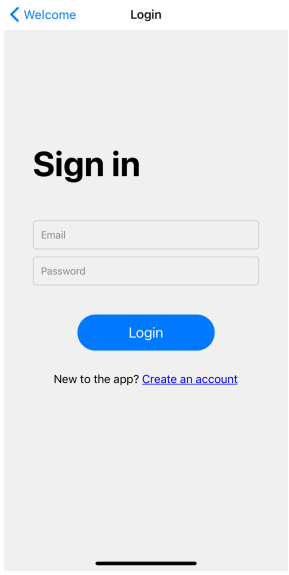


Home Screen

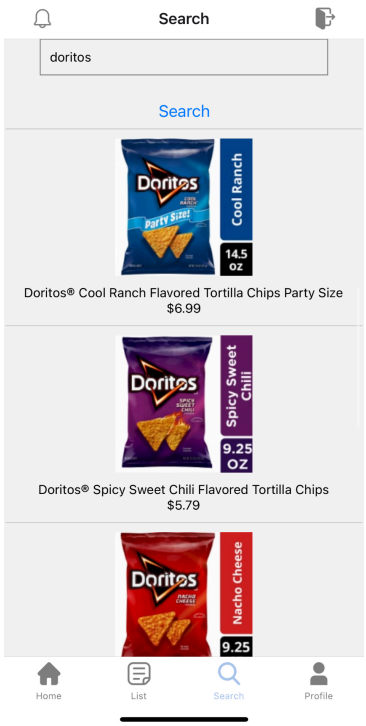
Profile Screen



Login Screen



Item Discovery



List Details