



Final Report

Team: sdmay20-23 **Client:** General Public **Advisor:** Dr. Goce Trajcevski

Multi-Context Shopping Optimization

Team Members:

Max Garton - Sensor Setup Engineer
Jesrik Gomez - Deployment Engineer
Karla Montoya - User Interface Engineer
Arnoldo Montoya-Gamez - Frontend Engineer
Ethan Shoemaker - Sensor Engineer
Nate Wernimont - Backend Engineer

Team Website: <http://sdmay20-23.sd.ece.iastate.edu>

Revised: 4/25/2020

Table of Contents

1. Introduction	4
1.1 Acknowledgement	4
1.2 Problem Statement	4
1.3 Previous Work and Literature	4
1.4 Use Case Diagram	5
1.5 Intended Users and Intended Uses	5
1.6 Assumptions and Limitations	7
2. Specifications and Analysis	8
2.1 Functional Requirements	8
2.2 Non-Functional Requirements	9
2.3 User Interface Requirements & Specifications	10
2.4 Technologies Used and Rationale	10
2.4.1 Inventory Sensor: Weight Dependent Resistor	10
2.4.2 Sensor Device Microcontroller: Raspberry Pi	10
2.4.3 Mobile Development Framework: Android	11
2.4.4 Backend Technologies: MySQL Database & Go Server	11
2.4.5 Cloud Computing Platforms: Microsoft Azure	12
2.4.6 API and Data Interchange Format: gRPC and Protocol Buffers	12
2.5 Engineering Standards and Design Practices	12
3. Revised Design	12
3.1 Overall System Design	13
3.1.1 System Component Diagram	13
3.2 System Component Designs	13
3.2.1 Inventory Sensor Device Design	13
3.2.2 Backend Service Designs	15
3.2.3 Data Storage & Transmission	16
3.2.4 Routing Algorithm Design	17
3.2.5 Android Application Architecture	18
3.2.6 User Interface Design	18
4. Implementation	19
4.1 Sensor Device Implementation	19
4.2 Software Development Process	19
4.3 Features and Implementation Strategies	20
4.3.1 Inventory Sensor Setup (Max Garton)	20

4.3.2 Inventory Measurement (Ethan Shoemaker)	21
4.3.3 Shopping Recommendation (Nate Wernimont)	21
4.3.4 User Shopping List Feature (Karla Montoya)	21
4.3.5 Low Inventory Notifications and Actions (Arnoldo Montoya-Gamez)	21
4.3.6 Inventory Monitoring (Jesrik Gomez)	22
5. Testing	22
5.1 Functional Testing	22
5.2 Non-Functional Testing	24
6. Closing Material	25
7. References	25
Appendix I: Operation Manual	26

1. Introduction

1.1 Acknowledgement

The team thanks the Iowa State University department of Electrical and Computer Engineering for giving us resources, guidance and expert consultation. We appreciate the Electronic Technology Group for providing us with our team website server and hardware components for the project. Thank you to Goce Trajcevksi for meeting with us weekly to give us guidance and advice throughout the project.

1.2 Problem Statement

Consumers are presented with many different ways to shop - going to a favorite store, going to the closest store, or going to the store where an item's price is lowest, or taking the most efficient route to reach several stores. How can a consumer know that the choice they have made is the best choice? How can we provide an optimal shopping recommendation to a consumer using a combination of contextual information such as the user's current inventory, the user's current location, the user's shopping list, stores in proximity to the user, hours of the stores, as well as the stores' availability and prices for items?

Our solution is to develop a system that utilizes microcontrollers to automatically monitor the quantity of items that a user has in their household. The microcontrollers would communicate the data to a remote server that tracks the user's inventory of several items. This inventory data would be accessible and modifiable via a smartphone application. The user would also be able to utilize this application to maintain a shopping list which is also automatically curated based on detected low inventory values in their household. The app would then provide the user with recommendations based on the user's current location, inventory data, the supply of items at nearby stores, and other contextual information to suggest an optimal way for the user to shop. Figure 1 in section 1.4 illustrates these identified use cases.

1.3 Previous Work and Literature

A previous senior design team at Iowa State University worked on creating a "smart bin" device that measures the quantity of an item based on the measured weight, combined with software that helps businesses restock products more efficiently [1]. While this project aims to solve a similar problem, the user that we are targeting is a general consumer or shopper, rather than a business. The project had a lower cost than our anticipated cost - their project required \$80 minimum purchase of microcontrollers and sensor modules, while our anticipated cost is under \$250. In addition, the Multi-Context Shopping Optimization project includes the possibility of

extending the optimization to multiple shoppers, which differentiates it from past projects. A document detailing Sequence-Aware Recommender Systems states that it will discuss some relevant topics to any Recommender Systems; however, these are typically geared towards more general scenarios where the system scope is larger than what our system is designed to do. Such things include user-item pairing matrices to predict other user-item ratings and session-based recommendations. These do not apply to our project since we are not strictly developing recommendations for users based on their habits. We are providing recommendations based on factual information for which an objectively optimal selection can be chosen, such as the store that is closest and has the cheapest total cost for all of the items [2].

There are few widely known consumer services that offer similar combinations of automated inventory measurement and shopping recommendations to this project. One known service, Bottomless, is limited to only coffee bean measurement. A scale connected to the customer's wifi network tracks the weight of coffee beans that the customer has. When the amount of coffee beans reaches a certain level, Bottomless automatically orders more beans and ships them to the customer [3]. While Bottomless' automatic inventory tracking has overlap with our project, the scope of our inventory tracking is on any household item that is purchased regularly. Our project focuses on in-store shopping rather than online delivery of goods, and has no restriction on the brand or source of the items.

1.4 Use Case Diagram

Our use case diagram (*Figure 1*) below illustrates all actors, services and use cases implemented in our project. The backend services are separated into microservices, but are shown as a monolith here for simplicity.

1.5 Intended Users and Intended Uses

This solution is intended to be used by any consumer that shops regularly. These regular shoppers will typically be interested in saving money and time in their grocery-shopping trips. This could include:

- Families in an urban or suburban setting within 20 miles of a store
- Members of apartments who share kitchens and household supplies

We intend for this application to be used on an on demand basis whenever the user would like to shop or view their current inventory. *Figure 1* below illustrates the various on-demand uses of the application.

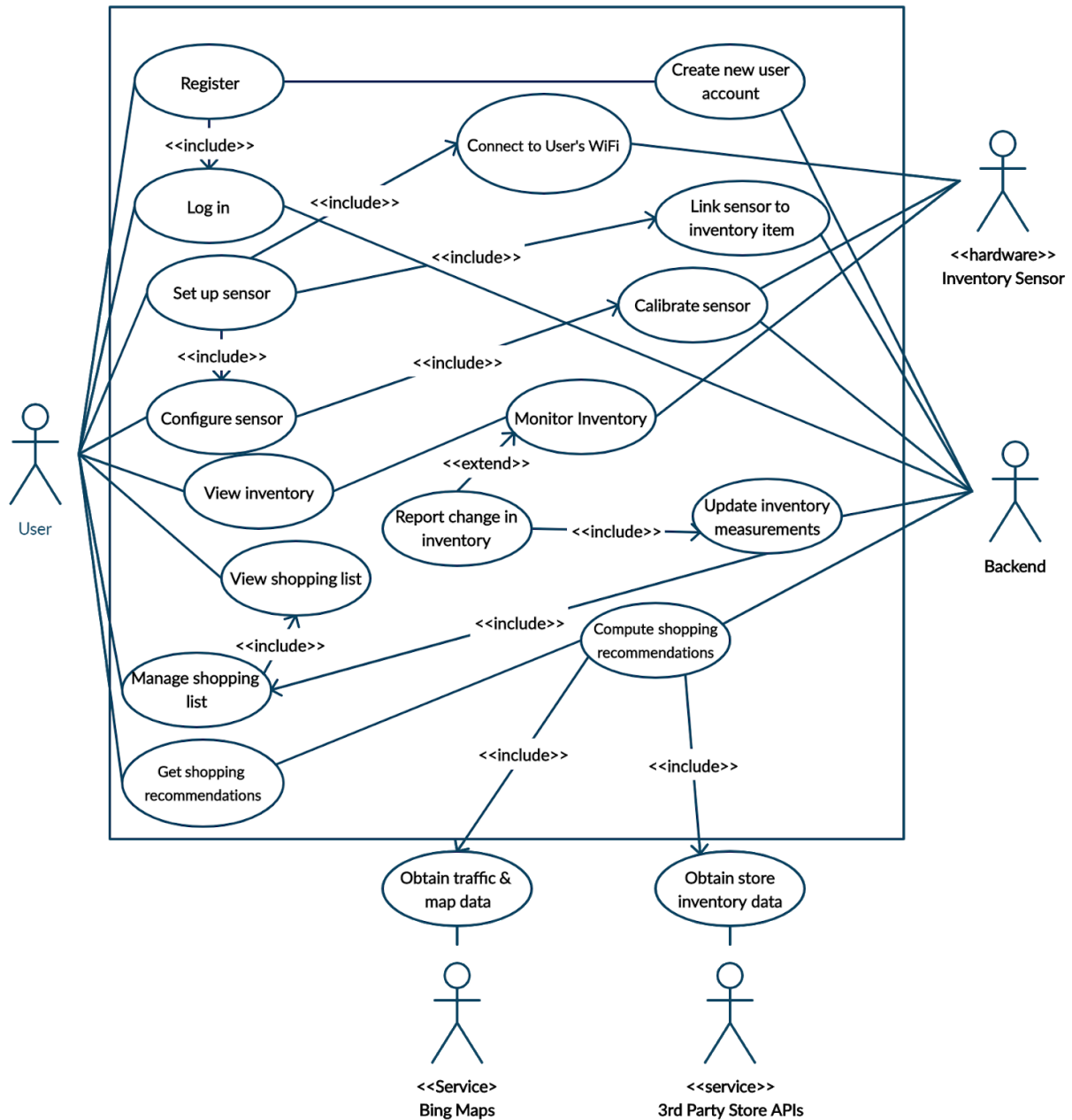


Figure 1: Use Case Diagram

The use cases are further described below from the perspective of a specific user, Mr. Smith, for continuity:

1. Mr. Smith creates an account and signs in
 - a. Mr. Smith opens the Android application and enters an email and password to create an account
2. Mr. Smith sets up a sensor module to add it to his account
 - a. Mr. Smith opens the Android application and opens the “add a sensor” menu

- b. Mr. Smith then finds the local WiFi of the sensor device and connects to it
 - c. The application prompts him to calibrate the sensor device using an item of known weight
 - d. The application prompts Mr. Smith to enter the Wifi SSID and passkey of his home WiFi in order for the sensor device to connect to his home network
 - e. After joining the network, Mr. Smith is prompted specify the details of the item that the new sensor device is measuring
3. Mr. Smith views the status of his inventory
 - a. He opens the Android application and logs into his account
 - b. He can see his current inventory amounts, such as:
 - i. Milk - full
 - ii. Eggs - 50% remaining
 - iii. Detergent - 10% remaining
 4. Mr. Smith views and modifies his shopping list
 - a. He opens the Android application and views his current shopping list
 - b. He adds another item to the shopping list
 - c. He updates the quantity needed of an item on the shopping list
 - d. He removes an item from the shopping list
 5. The inventory sensing module tracks Mr. Smith's inventory of items
 - a. A sensor module reading is sent to the remote inventory server
 - b. The inventory server identifies this as a low amount
 - c. The server updates the stored inventory value for that item
 - d. The server adds the item to Mr. Smith's shopping list if the low value is below Mr. Smith's set threshold
 6. Mr. Smith gets shopping recommendations
 - a. He opens the Android application and opens the "start shopping" menu
 - b. He establishes his shopping criteria within the app:
 - i. Store location - maximum distance away
 - ii. Prioritization of price and time - via a slider
 - c. He submits the request for shopping recommendations and is presented with recommendations on which stores to shop at, which items to buy at which stores, and the most optimal ordering to visit the stores

1.6 Assumptions and Limitations

In order to narrow the scope of the problem statement, we introduce the following assumptions and limitations:

Assumptions

- The user's home has wireless internet
- The user owns a GPS and data enabled Android device that they carry with them at all times
- External data including the inventory and price data from stores, is publicly available via one or more APIs

- The user has an available power outlet within reasonable distance of an item that is being measured

Limitations

- The application will only be available on Android devices
- Any devices used to measure users' inventory of items must be able to operate in within a refrigerator in lower temperatures
- The entire system must have a minimal impact on the user's internet traffic. Data should only be transmitted when needed.
- The application should not use an unreasonable amount of battery life on the user's Android device.
- The primary user interface will be within the Android application.
- Security and user privacy are valid concerns regarding this project, but they are not the major focus of this work.

2. Specifications and Analysis

Below we will discuss functional and non-functional requirements for the application. Also included is an explanation of selected technologies and our rationale in making these decisions.

2.1 Functional Requirements

- Access to interact the system
 - The system shall allow a user to interact with the system via a user interface in an Android application
 - The system shall allow a user to create an account, registering with the system using an email and a password of their choosing
 - The system shall allow a user to log into the system using an existing account
 - The system shall allow a user to log out of the system
- Inventory measurement
 - The system shall monitor the user's inventory of specific items
 - The system shall allow a user to begin measuring a new item by setting up a new sensor
 - The system shall allow a user to indicate to the system what a full and empty weight should be for a specific item
 - The system shall allow a user to use an existing sensor to measure a new item
- Shopping list curation
 - The system shall allow a user to manually add or remove items to a shopping list
 - The system shall automatically add items to the shopping list that were detected to be low in inventory

- The system shall automatically remove items from the user's shopping list after they have recently purchased them (after receiving shopping recommendations from the system)
- Shopping recommendations
 - The user shall specify a maximum distance within which they would be willing to travel to grocery stores. Then, they can also specify their preferences for whether to emphasize saving money or saving time by traveling to less grocery stores. The system will then route them between stores, specifying which items to buy at which stores.
- Notifications
 - The system shall notify the user when an item is detected to be low in inventory
 - The system shall notify the user when they are near stores where items on their shopping list can be purchased

2.2 Non-Functional Requirements

- Performance
 - The system should be capable of providing the user with up to date information about their inventory and shopping list in real time
 - The system should provide shopping recommendations on demand in real time
 - The user interface should not have noticeable buffer or lag, subject to the speed and availability of the internet connection
- Scalability
 - Capable of supporting 100,000 users
 - Including multiple sensors per user
- Longevity of sensor device
 - The sensor device should be capable of lasting at least two weeks when powered via a battery.
 - The sensor should either maintain its accuracy for an extended amount of time or allow the user to recalibrate on an as needed basis.
 - The sensor device should be able to withstand having weight of up to 3 pounds placed on it repeatedly for an extended period of time.

The following are identified as important design considerations that should especially be taken seriously when working with user data. We acknowledge that these are ideals, but they are not a major focus of this project as a demonstration of concept.

- User privacy
 - The system shall not collect any unnecessary information about users (necessary information includes the user's inventory measurements, shopping preferences and shopping list)
 - The system shall not send or expose any user data to third parties or applications
- Security
 - The backend services shall be invulnerable to SQL injection
 - Any requests involving user data shall be authenticated and secured

2.3 User Interface Requirements & Specifications

The main point of interaction with the system is the user interface implemented as an Android application. It is paramount that this application provides an attractive, user-friendly, responsive interface to the user. The following requirements ensure that the application meets these expectations.

- Visuals
 - The user interface should have a consistent look across all screens and menus
- User Experience
 - The user interface should have simplistic, straightforward navigation
 - The user should be presented with visual instructions for any features that are not self explanatory
- Performance & Responsiveness
 - The user interface should not have any perceivable delay or lag, subject to networking constraints
 - Any data or settings changes made by the user should be reflected in the interface without noticeable delay

2.4 Technologies Used and Rationale

Below we discuss our technology considerations for different components of the project. We also explain in this section how each technology is used in the project.

2.4.1 Inventory Sensor: Weight Dependent Resistor

Two main options were considered for the sensor portion of the inventory device. One was a simple force sensing resistor which would have a resistance value which varied depending on the amount of force being placed on it. This made it incredibly clear on how we would integrate it. We would use an analog input on our microcontroller and drive a voltage through this resistor. This device is also very thin and unobtrusive. Another option considered was a weight scale sensor which already included a platform for placing items on it. This option also includes an inline ADC and serializer.

Since the force sensing resistor could not deliver consistent outputs and the microcontroller option we chose did not have an analog input, we selected the latter option. An added bonus for this option was also the open source Python library for reading values from the sensor.

2.4.2 Sensor Device Microcontroller: Raspberry Pi

We considered two major microcontroller platforms: Arduino and Raspberry Pi. These were our primary considerations due to their wide use among developers, support network, and number

of libraries available. Both platforms have varying sized boards that allow rapid prototyping or more refined, compact designs. We primarily considered the Arduino Uno and the Raspberry Pi 3 model B. These are the most widely used microcontrollers, both having a large support network.

Due to the need for communication between the sensor device and remote, cloud-based backend services, we recognized that the microcontroller would need to have an internet connection. More specifically, the connection needed to be wireless to allow the sensor to be used in locations where there is not a physical ethernet jack available. Both the Raspberry Pi 3 model B and Zero “W” have wireless internet built in, while only the smaller Arduino Nano IOT board does [4][5]. Additionally, we wanted to have flexibility in terms of programming languages and libraries used on the microcontroller. Because the Raspberry Pi boards run a version of Linux, these were the most flexible option. We selected Raspberry Pi as the microcontroller platform due to its flexibility and wide offering of libraries. More specifically, we used the larger Raspberry Pi 3B due to its pin headers, which made rapid prototyping easier. If this project was marketed directly to consumers, we would use the smaller Raspberry Pi Zero “W” board so that we could make the inventory sensor device smaller and more efficient.

2.4.3 Mobile Development Framework: Android

The team considered the two most common mobile operating systems: Android and iOS. We ended up using Android development for several key reasons. Android is the most common mobile operating system, allowing more potential users to install the app. Android has a wide range of documentation such as developer guides, design guidelines and APIs that can easily be integrated into our app [6]. Developers can create apps in Java, which the entire team is familiar with. On the other hand, the majority of iOS applications need to be programmed in Swift, which the team is unfamiliar with. Lastly, at least half of the team is familiar with Android application development from previous experiences.

2.4.4 Backend Technologies: MySQL Database & Go Server

For our databases, we had two primary choices: NoSQL and SQL. Since our data is especially tabular, there was less of an advantage to storing blobs in a NoSQL database. As such, we went with a SQL database. Given that we were using a SQL database, we opted to use MySQL, rather than alternatives like SQL Server, MariaDB, or SQLite, due to its friendliness and ease of use [7]. It is incredibly simple to set up a MySQL server locally, and members on our team were familiar with the syntax of MySQL.

For our backend server, we considered several languages. Our primary considerations were NodeJS and Go. We did not want to use a JVM stack because it was less simplistic to scale horizontally. Between NodeJS and Go, we went with Go because it is designed from the ground up having simple concurrency. Go allows us to easily implement our microservices and all required backend functionality [8].

2.4.5 Cloud Computing Platforms: Microsoft Azure

We broke our system down into microservices in order to meet availability and scalability requirements. We considered three major cloud computing platforms: Amazon Web Services, Microsoft Azure, and IBM Cloud. All three of these options offered free tiers sufficient for development, and the ability to scale resources as needed.

We selected Microsoft Azure [9] to host our backend services because one third of the team already had experience with the platform. Moreover, it was also the platform that offered the best free tier services [10] for our storage and containerized microservices' needs.

2.4.6 API and Data Interchange Format: gRPC and Protocol Buffers

The two primary RPC frameworks are Apache Thrift and gRPC. After analyzing both solutions, we opted for gRPC due to its reliance on protocol buffers. Thrift message serialization is rather verbose. Since we want to be able to scale for hundreds of thousands of users, we wanted to use the lightest weight messaging format. On top of that, gRPC had a few features that Thrift did not have, such as streaming [11]. We wanted to prepare for eventualities where users could have hundreds of items in their shopping cart. These messages would get rather large, and we foresaw performance improvements by migrating to using streaming formats eventually.

2.5 Engineering Standards and Design Practices

The following engineering standards and design practices were applied in the design and implementation phase of this project.

Technology Standards and Protocols

- HTTPS
- gRPC
- IEEE 802.11 Wifi Protocols
- IP & TCP protocol

Design Practices

- Iterative prototyping
- Microservice architecture
- Modular design
- Object oriented programming

3. Revised Design

The system design has evolved slightly since our plan from last semester. The major functional components and flow of information have remained the same, but the details have changed in terms of technologies and implementation. In this section we will discuss the significant design

patterns, data models, backend services, external services, and hardware components that we have used in the project.

3.1 Overall System Design

The system is divided into four primary components: the inventory sensors, the Android application, the backend services, and the external APIs that we depend on for store and map data. We discuss these components as a whole and individually in the following two sections.

3.1.1 System Component Diagram

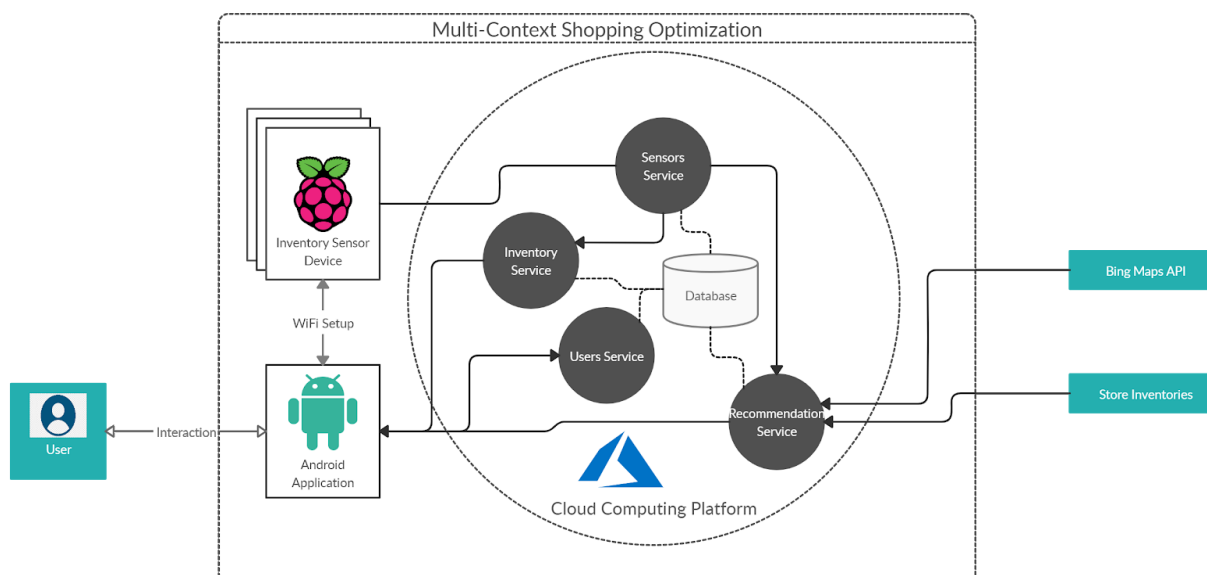


Figure 2: System Component Diagram

The diagram above depicts the four primary system components. Each of these components is discussed in detail in section 3.2, along with the frameworks and protocols that support the exchange of data between the components.

3.2 System Component Designs

Each of the major system components involved in this project are discussed in detail in the following subsections.

3.2.1 Inventory Sensor Device Design

The inventory sensor device consists of the Raspberry Pi model 3 B and a weight scale which includes an analog to digital converter (ADC) and data serializer. The Raspberry Pi will be running a version of Linux (Raspbian) which has been preloaded with our data sensing and command processing programs. The microcontroller has been pre-configured to run the necessary programs at startup.

The sensor device is first initialized in a wireless access point mode in which it broadcasts its SSID for the user to connect to via WiFi on their Android device. After the devices are connected together, the user interacts with the Android app in order to pass the sensor device the SSID and password for the user's home WiFi network. This way, the sensor device may now access the internet and communicate with the remote server.

Once it dialogues with the remote server, the user then sets an item for this sensor to be tracking along with a maximum and minimum weight for this item. From here, the sensor device will now automatically record the weight of the item on the scale every 15 seconds and store this entry locally. Every hour (or otherwise requested by the remote server), the sensor will send the most recently observed value to the remote server.

If the sensor device ever loses power or must reboot for any reason, the device will still automatically reconnect to the user's WiFi and will not need to be reconfigured.

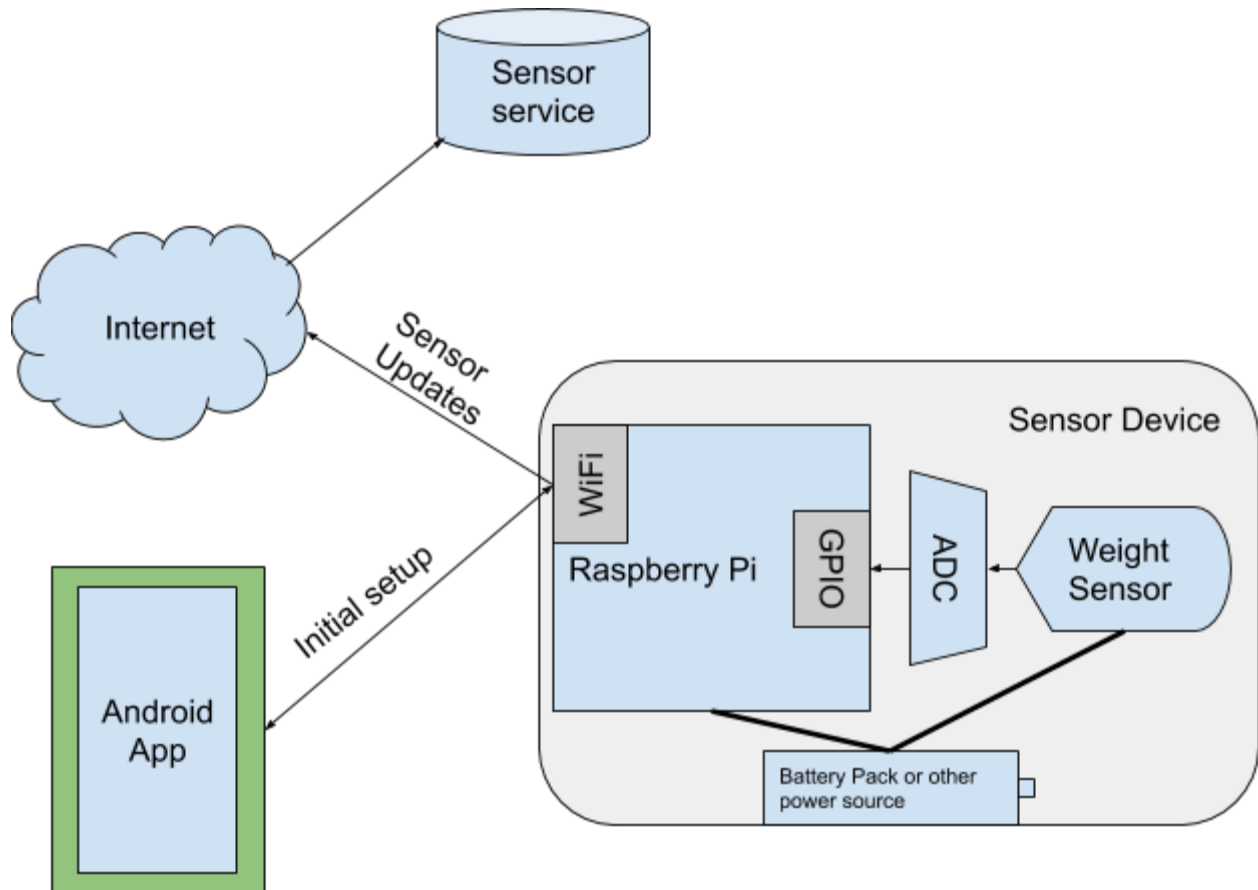


Figure 3: Inventory Sensor Device Architecture

3.2.2 Backend Service Designs

Models: Next, we will go through the various models included in the backend services. There are many different objects passed around throughout the backend service.

Item: At its core, an Item represents a grocery item for a consumer. An Item tends to include the ID associated with an item and the item's description. Depending on the Item's context, it sometimes includes its pricing or its quantity. For example, in the stores microservice and the routing microservice, Items include pricing. In the inventory microservice and the routing microservice, it includes quantities. In the sensor service, Items include neither.

Cookies: When the frontend submits a successful login request, they receive a response that includes a Cookie. A Cookie is simply a UUID that is used to associate a user to their successful login request. The frontend can then remember that Cookie and include the Cookie on all other requests to the backend in order to authenticate themselves.

Instruction: When the frontend needs to get routing recommendations, they need to be able to understand how to travel to the various stores and which items to buy at which stores. To do this, the backend gives them a sequence of Instructions. These Instructions are an ordered list containing information about the Store and the list of Items. A Store is simply a latitude, longitude, and an ID that uniquely identifies that store. This information allows the frontend to regenerate the directions using Bing Maps.

Sensors: When the frontend first registers a Sensor, it must include information such as the minimum value of the sensor, the threshold at which the Sensor is considered low, and the maximum value of the Sensor. The frontend then receives the ID of the newly created sensor, which is a UUID used to uniquely identify that sensor. The physical sensor device can then use that ID to report its values. After initial sensor registration, the frontend must also specify an ItemID for that sensor. Sensors then also contain information about its most recently reported value.

User: The final important entity in the backend is a User. A User consists of a unique username, a password, the first name of the user, and the last name of the user. The first and last name of the user is simply stored as metadata of the user.

Services: All of these entities are used throughout our microservices. We will now detail the core functionality of each of the microservices.

User service: The user service is designed to be able to register, login, and logout users. That is the only functionality that the user service provides. The user service functions by storing information about the user in MySQL. Such information includes the username, first name, last name, and a hashed version of the password. All data is stored in a single table within MySQL.

When a user logs in, they are returned a cookie that is a UUID. A key-value pair is then stored in a Redis cache with the key being the cookie and the value being the username. Then, when a user makes other requests, the other microservices can access the cache to map the given cookie to the associating username.

Sensor service: The sensor service is used to register sensors, configure sensors, report values for a sensor, fetch sensors for a user, and fetch low sensors for a user. All data for the sensor service is stored in MySQL. There are two tables for the sensor service: a metadata table and a values table. When registering and configuring sensors, the metadata table is used. When a sensor reports values, the service tracks the most recently reported value for each sensor in the values table. When authenticating requests, it accesses the Redis cache populated by the users service.

Inventory service: The inventory service is responsible for tracking the items manually added by users via the Android application. It can only modify the quantity of items needed by the user and fetch all currently needed items for a user. It stores all data in a single MySQL table that maps a (user, item_id) pair to quantities needed. This service also has access to the Redis cache needed to authenticate users and transform cookies to usernames.

Routing service: The most complicated microservice is the routing service. It has a single purpose, and that is to fetch routing information for a given user. When the frontend requests routing for a user, the routing service will fetch all currently needed items from the inventory service and all currently low sensors from the sensors service. It will then fetch all stores near the user according to the users preferences. Then, it will find the cheapest places to purchase every item and return that information to the frontend. It has no database backing. It also has access to the Redis cache.

Store service: Because we could not get access to public store APIs, we were forced to mock store data. In order to do that, we added a new store microservice. The store microservice stores information about stores and their locations in a single table. It utilizes spatial data types within MySQL in order to find stores near the customer. Then, there is an additional table that maps (store id, user id) pairs to pricing information. There is no authentication needed to reach the store service, but it is not exposed to the frontend. Only other backend services can access it.

3.2.3 Data Storage & Transmission

Data is stored in mySQL relational databases, and data is transmitted by directly calling the methods of each service with gRPC. We chose mySQL because of its ACID architecture, and because most of the team already had experience with relational databases and querying. Using mySQL constrained us to using relational data models; however, this was not an issue.

We limited the use of foreign keys in the database schema because foreign keys limit performance. Additionally, many schema migration tools cannot migrate tables with foreign keys. Instead, we rely on our application using correct logic.

We used Redis to store cookies of logged in users to facilitate their access.

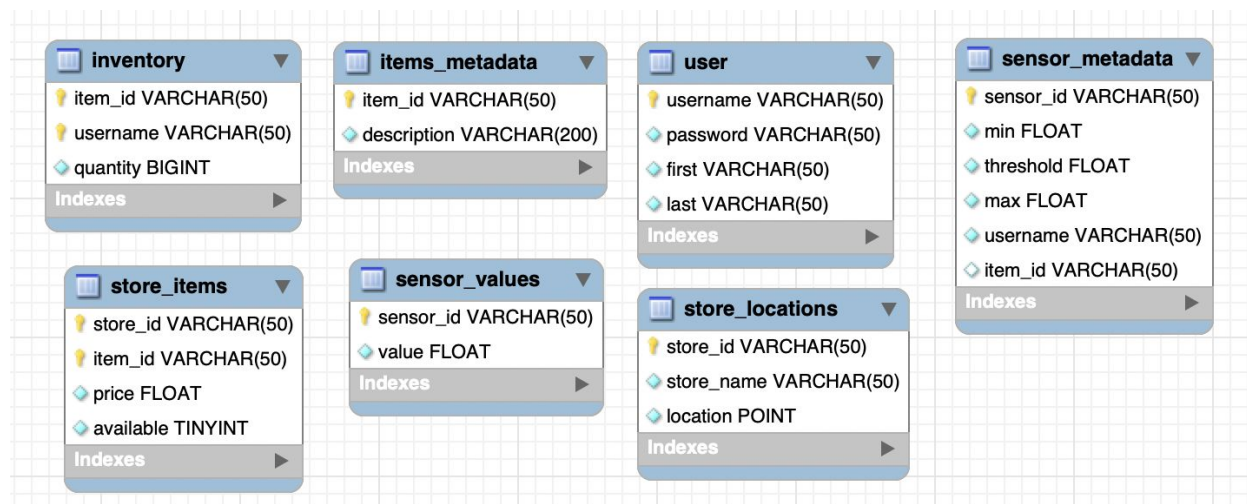


Figure 4: Database tables

3.2.4 Routing Algorithm Design

When implementing the algorithm, we started with a simplistic algorithm that accomplished our goals. We were unable to provide further functionality to the algorithm due to time constraints. So, in its current state, the algorithm has three primary phases. The first phase is an information gathering phase.

In the information gathering phases, the algorithm first converts the given cookie into a username. Then, it fetches the manually added items for the user from the inventory service. Then, it asks for any low sensor values from the sensor service. It combines these two lists of items to get a superset of all items needed. Then, it queries the store service for a list of stores within a configured value of the user. The user sets this radius on their mobile device. Finally, it fetches the pricing for every item from every store. It can then use this information to find which store has the cheapest items in the selection phase.

In the selection phase, every item is iterated over in order to find which store has the cheapest price for that item. This is the store where that item will be purchased within. This is the phase where we had planned most further improvements. We wanted to give the user a slider that would let them choose whether they wanted to prioritize time or money and by how much. In the event that the user wanted to prioritize time, the algorithm would weigh the cost benefits of going to additional stores with the added time cost of traveling to extra stores. However, the algorithm is currently implemented to always prioritize cost at the expense of time.

The final phase is the routing phase. In this phase, the backend orders the selected stores from the selection phase into a navigable list. Instead of implementing a traveling salesman algorithm, which is NP-hard, we chose to navigate to stores in order of their proximity to the user. While this is less than ideal, we chose this approach due to time constraints. If we had further development time, we would iterate on this in order to choose an approximate version of the TSP problem, giving the user a good route, but not compromising execution time.

3.2.5 Android Application Architecture

The Android application is divided into packages that distinguish the models, views and controllers. The design is open for extension but closed for modification to create modular, flexible software and meet the non-functional requirements. Object-oriented design principles incorporated throughout the design, for instance by implementing interfaces to allow us to mock the data controller. Figure 5 below illustrates the software architecture of the Android application. An example of how it is divided is, in the figure below, we have different views, each pertaining to different data, being the view in an MVC design model. The data controller is the controller in the MVC model, and the DB is our model.

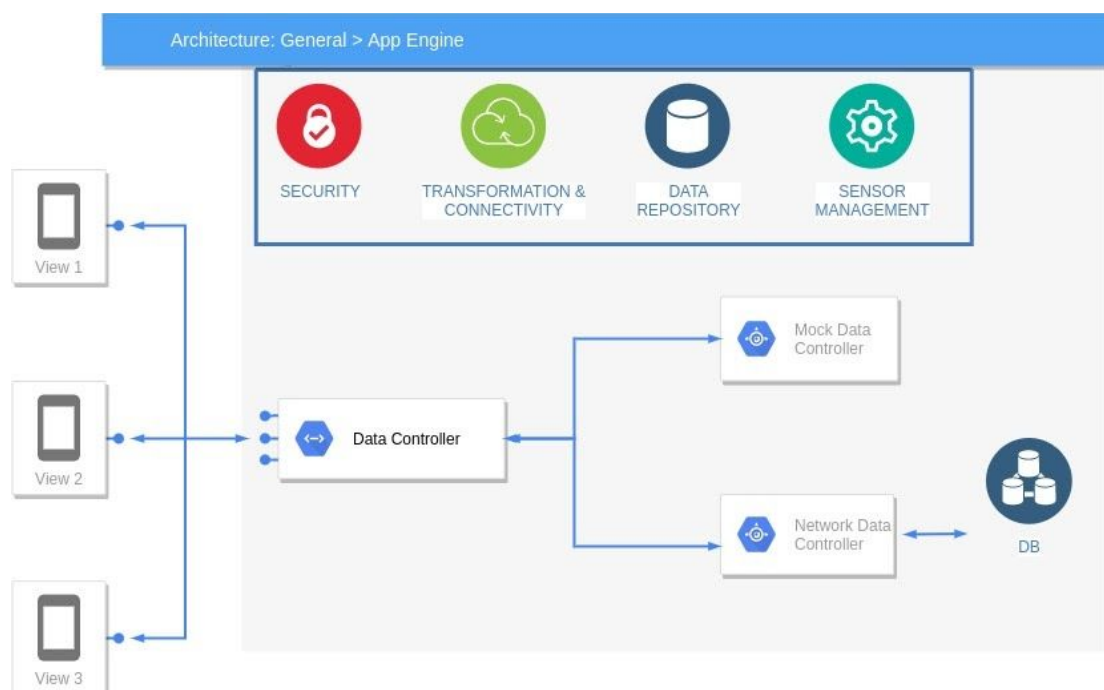


Figure 5: Android Application Architecture

3.2.6 User Interface Design

The user interfaces within the Android app were designed to support the primary use cases defined in the project. First, interfaces were sketched and mocked, envisioning how functionality and information would be displayed to the user. After determining what the ideal layouts and

screens would look like, the team researched existing Android user interface components that can be used. Our initial mockups were modified to take advantage of Android's existing UI components to allow us to focus on implementing the functionality instead. Throughout this process, we strived to design interfaces that provided a positive user experience. We did this by adhering to the following design practices:

- Clearly label icons and buttons
- No more than three actions should take place on one screen
- Provide feedback to the user's responses via visual changes and notifications
- Maintain consistent visual design throughout the app

4. Implementation

4.1 Sensor Device Implementation

We had previously experimented with different sensor options in the fall semester. We had arrived at a plan in terms of the microcontroller and sensors to use to create the sensor device. This semester, we determined more specific functional requirements for the sensor device, such as the need to perform necessary tasks on boot without any user input. This was implemented using daemons written in Python. We also added the requirement that the user must be able to reset the sensor to its original, unconfigured state if needed. The device would then need to enter its setup mode again. This requirement was satisfied by adding a "reset" button that the user may press should they need to reset the sensor device. All sensor device implementation and testing was done using the Raspberry Pi model 3 B. We have identified that the smaller, more efficient Raspberry Pi Zero "W" may be used to assemble a compact, finalized sensor device if this were a consumer product. However, in the interest of rapid prototyping, we stuck with the original model 3 B.



Figure 6: Inventory Sensor Device

4.2 Software Development Process

During the spring semester, the team used an agile development process to focus on a subset of the development items during each sprint. We had already established functional requirements, features that satisfy these requirements, and rough prototypes during the previous semester. This semester, we implemented them by first specifying the integration details for features that require communication between devices. Data interchange formats and requests were defined in shared documents to the entire team. By predefining the integration details, our team was able to easily mock the integration aspect of each feature until the other components involved were also ready to integrate. After mocking the integration details, we implemented the features. After implementation, functional testing was used to ensure that all functional requirements were met. If any bugs or functional flaws were detected, the

development item moved back into the “implement, fix and refine” phase. After finally passing functional testing and after other components were ready to integrate, the features were then integrated so that the entire system supported them. After that, the functional testing was repeated, using the newly integrated connections, to ensure that the functionality was still correct when using data from other system components. Once again, if any defects were found, the development item moved back to the “implement, fix and refine” phase. After passing integration testing, the code changes moved to a code review process where peers review the code and comment on any problems or inconsistencies with the rest of the project codebase. After passing the code review, the code changes were finally merged into the main project repository.

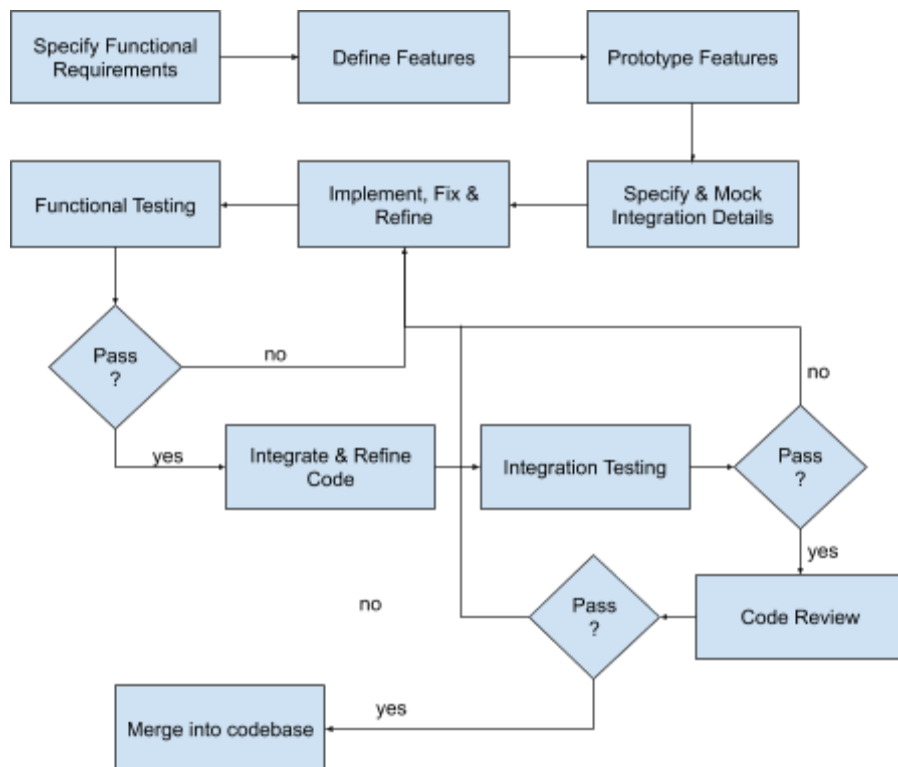


Figure 7: Iterative Development Process Diagram

4.3 Features and Implementation Strategies

We now describe in detail the implementation aspects of every major feature, and for each of them we indicate the key role assignment. We note that as part of the collaborative activities, the implementations of certain features were conducted jointly by multiple team members.

4.3.1 Inventory Sensor Setup (Max Garton)

The sensor setup is designed to be as simple for the user as possible. During initial setup, the inventory sensor acts as a wireless access point, broadcasting its own WiFi network that the

user connects to from their Android device. Then, the user opens the application on their phone and enters the sensor setup menu. From there, the application uses an IP socket for communication between the Android device and the microcontroller. The user enters their wireless network info, which includes the network SSID and passkey. This information is sent to the sensor device, which then disables its own wireless access point and connects to the user's home wireless network. The Android application walks the user through every step of the setup process with specific instructions. On the microcontroller, the network configuration is switched between two states (serving as a wifi access point or connecting to the user's wireless network). This is implemented by using two sets of configuration files that are copied into the board's Linux operating system directories as needed to change configurations during setup or reset of the sensor device.

4.3.2 Inventory Measurement (Ethan Shoemaker)

The inventory was recorded by starting daemons once the system had booted and using files as well as signals to communicate between these daemons. A simple polling structure was utilized in order to: 1) take a measurement every 15 seconds; 2) send the most recent measurement every hour; and 3) listen for activities from the server and report an on demand measurement. This was all done using Python as the language with the Pickle library for efficient file storage and systemctl for daemonizing the programs.

4.3.3 Shopping Recommendation (Nate Wernimont)

The shopping recommendation feature was written on the backend, which was implemented in Go. It lives within the routing microservice. It pulls data from the inventory microservice and sensor microservice in order to get the items needed by each user. It also pulls data from the stores microservice, which consists of mocked data for a few stores in the Ames area. It then aggregates this data and returns it to the mobile application, which displays the data to the user.

4.3.4 User Shopping List Feature (Karla Montoya)

The user shopping list is displayed in a RecyclerView, which recycles a list of views to create the illusion of a scrolling list. This list was created using Android's LiveData framework, which updates data on the user interface when it changes within the model. The Android ViewModel framework was utilized to save, load and display data models in the user interface.

4.3.5 Low Inventory Notifications and Actions (Arnoldo Montoya-Gamez)

The Android application regularly polls the remote inventory service to determine if any of the user's inventory items are low (indicated by a low sensor reading). If this is the case, then the user receives a notification that prompts them to add the item to their shopping list.

4.3.6 Inventory Monitoring (Jesrik Gomez)

The users' inventory sensor devices send an inventory measurement to the remote inventory service every 15 minutes, or when a change is detected. By regularly monitoring the data, we can easily detect if a sensor has gone offline for any reason (when no update has been received within 15 minutes). With each new measurement update, the remote inventory service stores the new value so that it can be fetched and displayed by the Android application.

5. Testing

Due to the complexity of the project and the interconnections between components, we primarily relied on functional testing to verify that the functional requirements were satisfied. Below we will discuss our functional test procedures and results, as well as non-functional testing.

5.1 Functional Testing

Our team followed an agile approach to software development, where software changes were typically focused on adding functionality to a use case. Because development on many of the use cases took place in parallel between different team members, we established pre-defined testing procedures that clearly laid out the initial conditions, the event trigger or user actions, and then the expected outcome. The table below describes our testing procedures and acceptance criteria for each of the use cases in section [1.5](#). The initials by each use case column indicate the team member(s) responsible for conducting the respective tests. However, as with our development items, each use case was developed and tested by several (and oftentimes all) members of our team.

Table 1: Functional Testing Plan & Acceptance Criteria

Use Case & Description		Testing Plan	Acceptance Criteria
1	Create an account and sign in (NW)	The user opens the Android application on their device and enters account information (email and a password of their choice). They click "sign in or register" to either create an account or sign in.	The application signs the user in (via an existing or newly created account) so that they are presented with the main application screen. If a new user account was created, the remote users service is now aware of this.
2	Sensor device setup (MG)	The sensor device is powered on for the first time by the user - it has previously been programmed by us. The user will be given instructions via the Android	The sensor device is connected to the user's wifi network and registered to the user's account. The inventory service, a remote

		application. The application will walk them through the setup process (connecting the device to their wifi network, calibrating the weight thresholds, and naming the item that the sensor is measuring).	backend service, is now aware of the sensor device and relates it to the user.
3	Viewing inventory status (AMG)	The user opens the Android application on their device and goes to the inventory menu. They view the status - relative inventory amount - of the items that are being tracked by sensor devices associated with their account.	The user can view the inventory status of all sensor devices associated with their account. The sensor reading is up to date and reflects the most recently measured weight.
4	Shopping list (KM)	The user opens the shopping list menu in the Android application. They modify their shopping list by adding, removing, or updating the quantity of items on the list.	The user is able to modify their shopping list by adding, removing or updating the quantity of items. These changes are reflected in the application the next time they open the application - in other words, the remote server also saves the changes.
5	Automatic inventory tracking (ES)	The user decreases the quantity of an item that is being tracked by a sensor device such that the weight of the item decreases. The device is already set up and associated with the user's account. Both small decreases and full depletions of inventory are simulated.	After a change in the weight of an item that a sensor is measuring, the inventory server updates the stored inventory value in the database. If the value is below the user's set "low" value that was established during the sensor setup process, then the item is added to the user's shopping list.
6	Shopping recommendations (JG)	A user with an established shopping list containing multiple items that can be found at multiple various stores opens the Android application. They open the "start shopping" menu to receive shopping recommendations based on contextual information. The user adjusts their shopping preference between saving money and saving time accordingly.	The user receives shopping recommendations based on their current location, their shopping list, and their input optimization criteria of saving the most time, saving the most money, or somewhere in the middle. The recommendations are also based on store proximity and availability including operating hours, available items in stock, and sufficient quantity in stock at the

			specific store.
--	--	--	-----------------

Integration Testing

The functional testing procedures in the table above were utilized both during initial development and during integration. The primary difference was that the initial development testing ensured that the use case was supported in a closed, simulated environment. Integration testing expanded the test procedures to use real data that is communicated to and from the different components of the system.

5.2 Non-Functional Testing

After the functional requirements of the project are satisfied and validated by the testing procedures defined above in section [5.1](#), we planned to conduct non-functional tests in order to ensure the quality of the product beyond the functionality. Unfortunately, the team did not have time to fully test all of the non-functional requirements specified in section [2.2](#). However, we specify our plans to verify each of the non-functional requirements below.

Performance Testing

- Monitor the network use while the inventory sensors are active with the user's home
- Monitor the network, battery, RAM, CPU, and storage usage on the smartphone when the app is running and when the app is in the background

Security Testing

- Use a packet sniffer to try to gain access to user information
- Use SQL injection and buffer overflow attacks to try and break both client and server code

Usability Testing

- User testing on outsiders who are unfamiliar with our project can indicate whether the project and the user interfaces are usable enough or if they need additional explanation.

Scalability Testing

- We are initially focusing on smaller data sets to simplify the scope of the recommendation problem - otherwise we are dealing with an NP-complete algorithm problem. Later on, we can introduce more data and possibilities for recommendations and test the algorithms that we have designed. We can also simulate a large number of sensors or users to see how our system scales.

6. Closing Material

During the past two semesters, our team has identified a series of problems that can be solved using technology and creativity. Our project aims to optimize several aspects of the shopping experience, ultimately allowing users to save time, money and effort. Although we did not have time to develop all of the use cases that we had in mind for this solution, we are satisfied with it as a polished proof of concept. Future expansion potential for this project includes support for collaborative shopping between family members or peers, more advanced inventory detection using smartphone cameras, as well as the combination of online ordering for both delivery and pickup at stores. The features that we have developed and the potential improvements identified above can make a significant improvement in the consumer shopping experience.

7. References

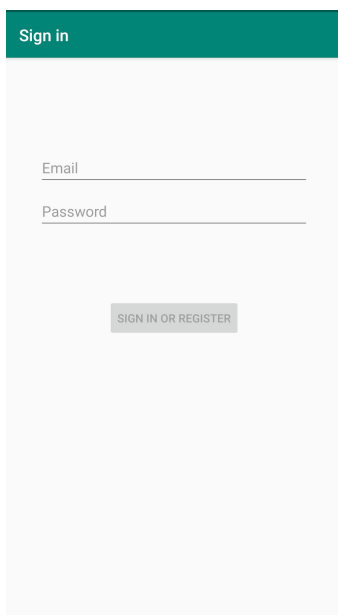
- [1] A. Hauge, D. Bis, S. Guenette, H. Moser, N. Bix and B. Gruman. *Automating Inventory Management: Routing through Sensor Networks*. Iowa State University: 2019.
<http://sdmay19-29.sd.ece.iastate.edu/>
- [2] M. Quadrana, P. Cremonesi, and D. Jannach, "Sequence-Aware Recommender Systems," Sequence-Aware Recommender Systems, Feb-2018.
<https://arxiv.org/pdf/1802.08452.pdf>.
- [3] Bottomless. <https://www.bottomless.com/faq>
- [4] Raspberry Pi Foundation. <https://www.raspberrypi.org/products/>
- [5] Arduino. <https://www.arduino.cc/en/products/compare>
- [6] Google Android Documentation. <https://developers.google.com/android>
- [7] Oracle. MySQL Reference Manual. <https://dev.mysql.com/doc/refman/8.0/en/>
- [8] Writing Web Applications. Golang. <https://golang.org/doc/articles/wiki/>
- [9] Microsoft Azure. <https://docs.microsoft.com/en-us/azure/?product=featured>
- [10] Microsoft Azure Free Tier. <https://azure.microsoft.com/en-us/free/>
- [11] gRPC Documentation: <https://grpc.io/docs/>

Appendix I: Operation Manual

This manual contains step-by-step instructions to help you optimize your shopping experience.

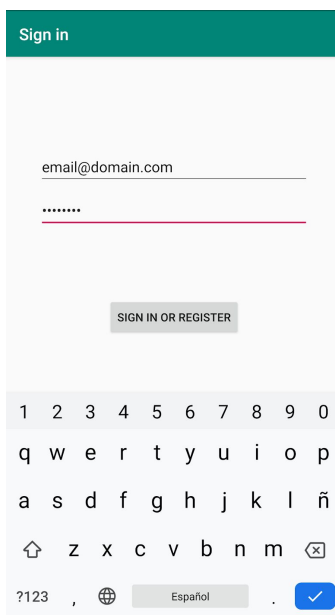
Getting Started

Step 1: Open the App



Step 2: Enter Credentials

Enter a username and password of your choosing, then press the button to register a new account.



Step 3: You will be logged into your account



Sensor Setup & Installation

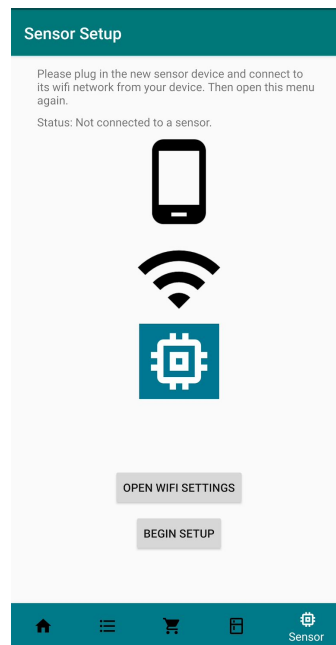
Step 1: Open the Sensor Setup Menu

From the home screen, navigate to the sensor setup screen by touching the sensor icon on the navigation bar.



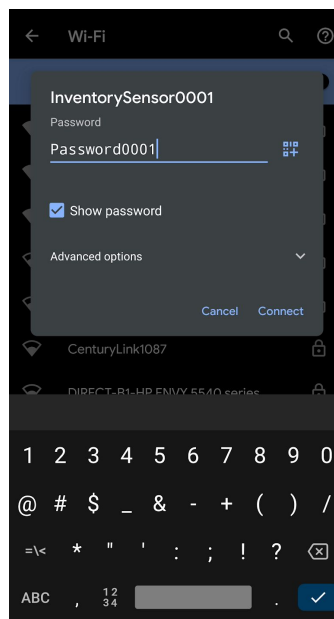
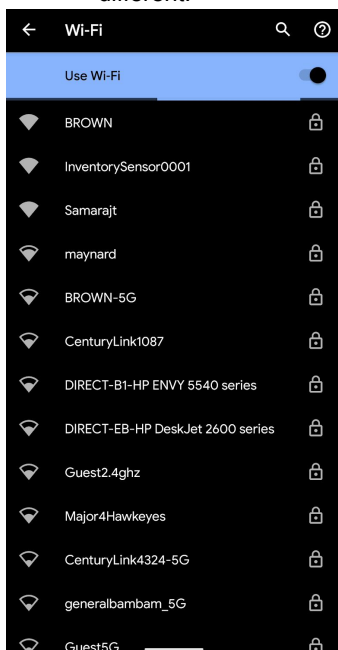
Step 2: Open Wifi Settings

Follow the on-screen directions. Click the button to open your device's wifi settings.

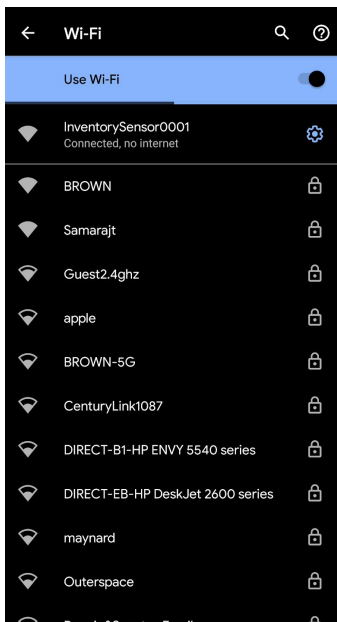


Step 3: Find and Connect to the Sensor

- Locate the inventory sensor's wifi network and tap on it to connect to it. Note: depending on your device, this settings menu may look different.
- Enter the password to connect to the inventory sensor. The default password is "PasswordXXXX!" where "XXXX" is the four digit number found in the network name. Then touch "connect" to be connected to the network.



- c. Verify that your device has connected to the sensor's wifi network. It may say "limited connection" or "no internet".

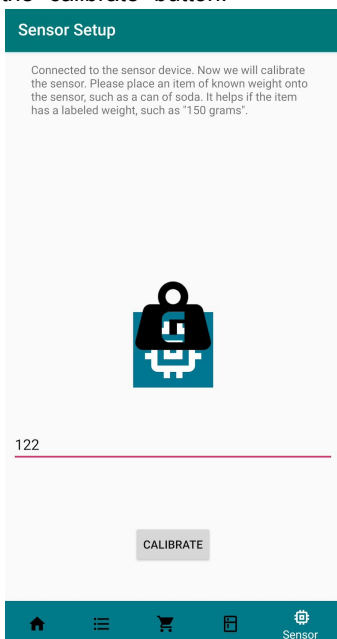


- d. Navigate back to the multi-context shopping optimization app by pressing the back button on your device. This will return to the sensor setup screen. The status should now indicate that your device is connected to the sensor.



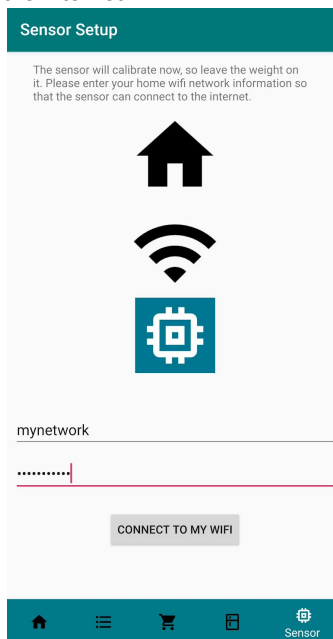
Step 7: Calibrate the Sensor

Follow the on-screen instructions to calibrate the sensor. You should use an item that has a known or labeled weight in grams. If the item has a decimal weight, round it to the nearest whole gram. Then press the "calibrate" button.



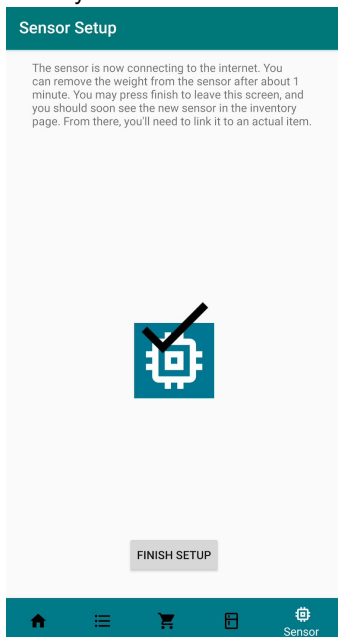
Step 8: Wifi Setup

Leaving the calibration item on the sensor, enter your home wifi network SSID and passkey into the text entries. Then press the button to connect the sensor to the internet.



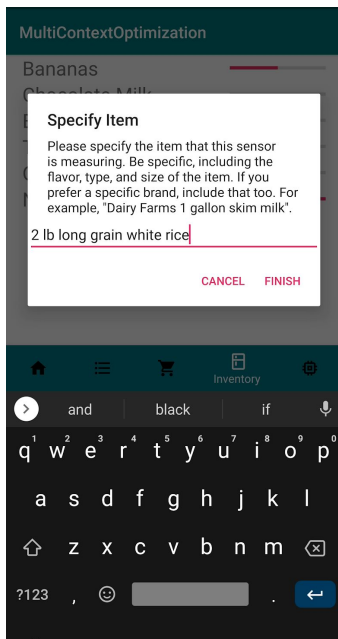
Step 9: Finish Setup

Continue following the on-screen instructions. You may now press the button to finish the network setup. The app will automatically navigate to the inventory screen, where you should see the new inventory sensor.



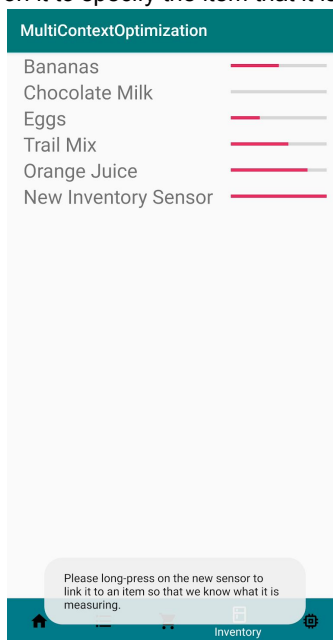
Step 11: Specify the item

Specify the item that the new sensor is measuring. If the item is from a specific store or brand, include that information. If the item is generic, include the size or other details.



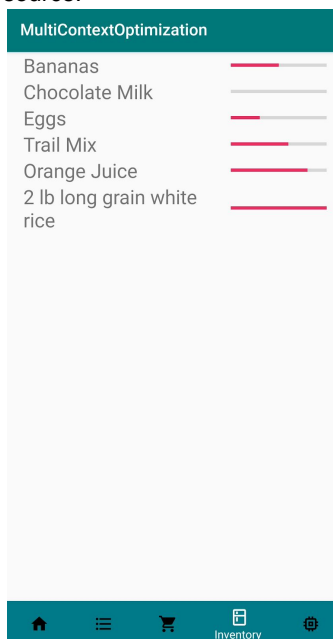
Step 10: Viewing the new sensor

You should see the new inventory sensor on the inventory screen. The app will prompt you to long-press on it to specify the item that it is measuring.



Step 12: Verify & Finalize the Setup

You should now see the new inventory sensor tracking the item that you specified. You may now place the sensor in the location of the item (such as in the fridge or in a cupboard). Ensure that the sensor is always within range of a wifi network and plugged into a power source.



Modifying the Shopping List

Navigating to the Shopping List

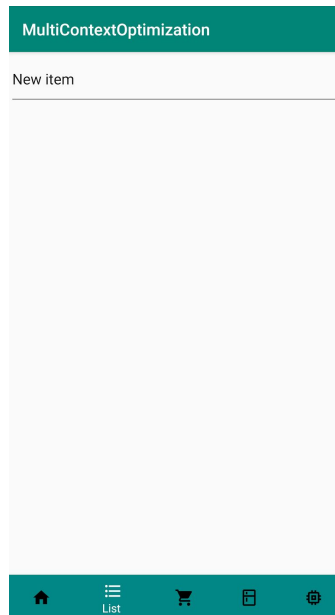
Step 1

From the home screen, tap the list icon on the navigation bar.



Step 2

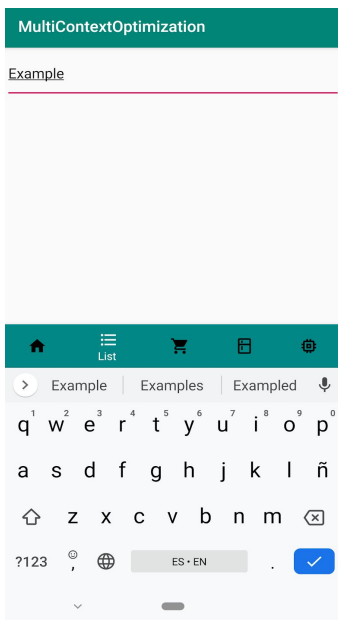
The shopping list screen should be displayed.



Adding a New Item to the Shopping List

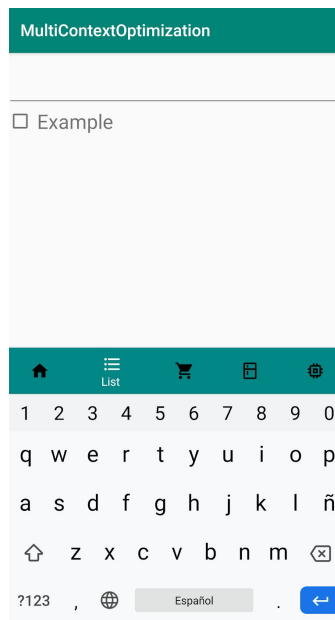
Step 1

Type a new item name in the text box and press enter.



Step 2

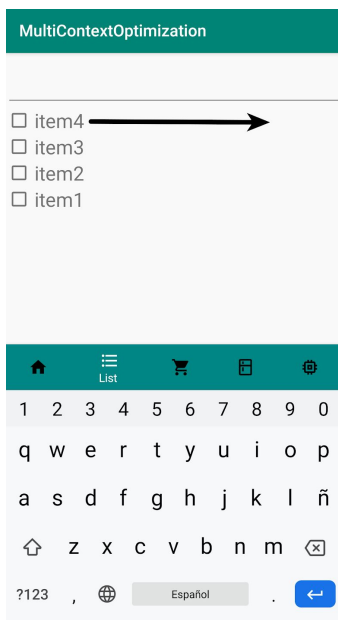
The new item should be displayed in your shopping list.



Removing an Item From the Shopping List

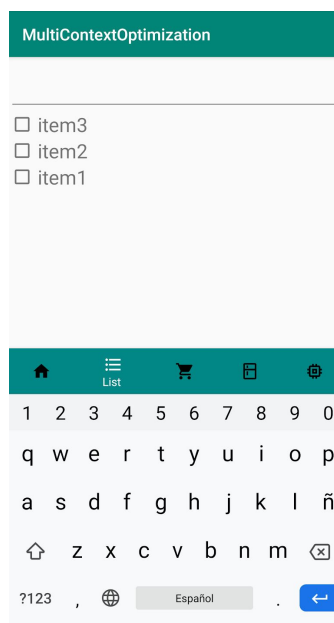
Step 1

Open the shopping list screen. Touch and drag to the right on an item that you'd like to remove, as shown in the screenshot below.



Step 2

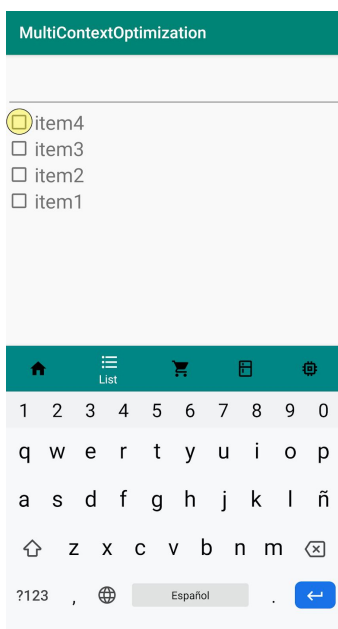
The item should now be removed from the list.



Checking & Unchecking Shopping List Items

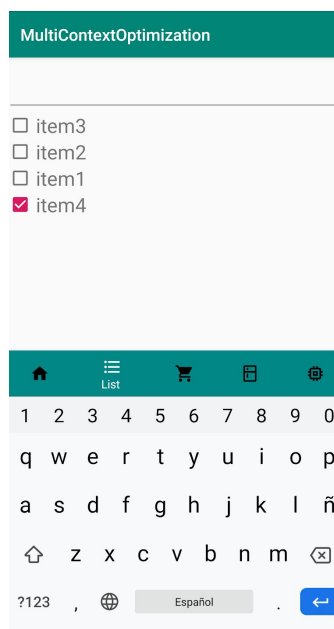
Step 1

Tap on the checkbox next to the item you want to check off.



Step 2

The item should now be checked off and moved to the bottom of the list.



Shopping Recommendations

Navigating to the Recommendations Screen

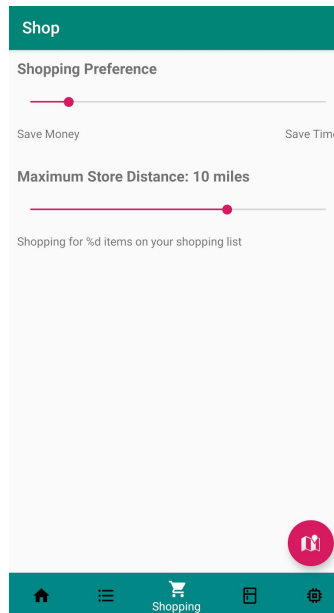
Step 1

From the homescreen, tap on the shopping icon in the navigation bar.



Step 2

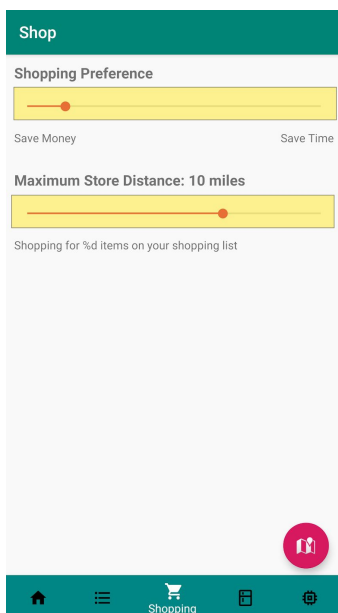
The shopping screen will be displayed.



Entering Shopping Preferences

Step 1

Slide the shopping preference and maximum store distance bars as desired.



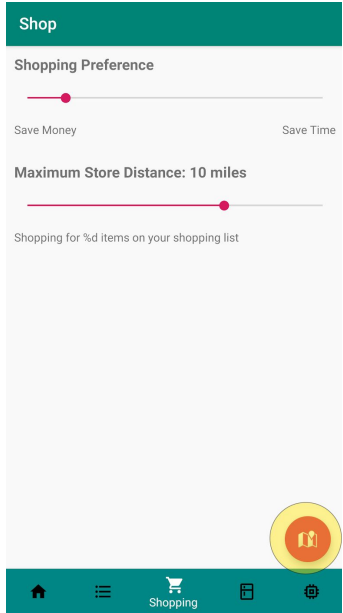
Slider Descriptions:

- **Shopping Preference** is the priority of saving time by including stores closer to you versus saving money by including stores with lower prices. Slide this to the right if you are mainly interested in saving time, or to the left if you would prefer to save more money at the expense of time.
- **Maximum Store Distance** indicates how far away you're willing to drive to reach nearby stores.

Getting Shopping Recommendations

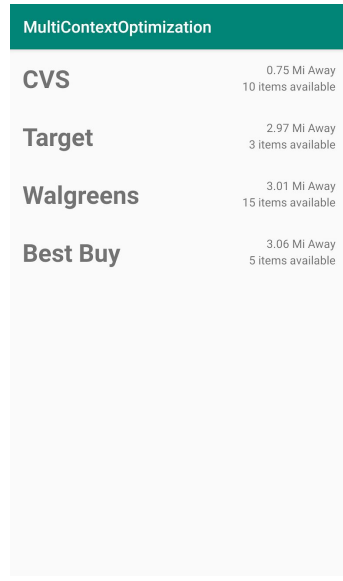
Step 1

After entering your shopping preferences, press the pink map button to request recommendations.



Step 2

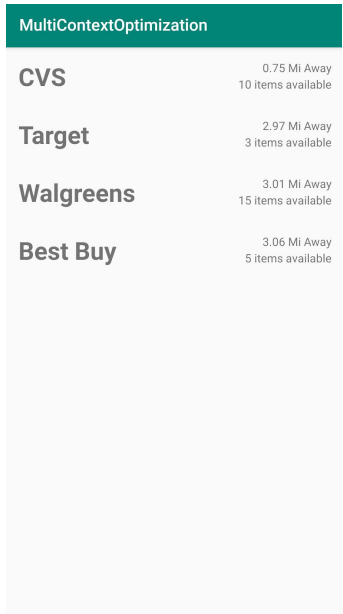
Depending on the number of stores that have items you're shopping for, you will see results sorted by distance away from your current location.



Navigating to a Store

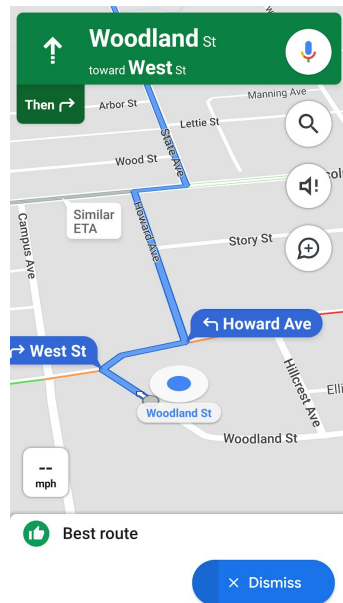
Step 1

Tap on the store you'd like to shop at.



Step 2

Google maps will start navigating to the store from your current location.



Monitoring Inventory

Navigating to the Inventory Screen

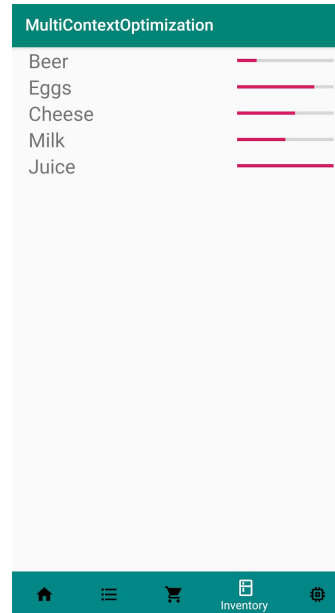
Step 1

From the home screen, tap the inventory icon in the navigation bar.



Step 2

If you have any sensors setup and measuring items, they will be displayed in your inventory here. To set up new sensors, see "Sensor Setup & Installation".



Resetting an Inventory Sensor

Step 1: Press the Reset Button

Press the reset button located on the back side of the sensor device.

Step 2: Wait 2 Minutes

The sensor device will reset its configuration, including wifi settings, to the default state. It will need to be set up again.

Step 3: Set Up the Device Again

See "Sensor Setup & Installation" for instructions on how to set up the device again.