Design Document: Cactus

Specification

The Cactus application stack consists of two main parts: Cereus, which is an Android application, and Saguaro, which is a web server written using Spring Boot. At a high level, the application allows users to login, create and edit grocery lists, save them to the server.

Cereus:

Cereus is an Android app written natively in Java using the Android Platform APIs. Below will show the flow of the program.

- Toolbar (at the top of each page)

The toolbar is the rectangle present above each activity. It contains the name of the app along with two buttons. The home button will bring the user to the display list page, which is the home page. The user button will bring the user to the user profile page where they can edit their credentials and add friends. Both these buttons only work when logged in.

    - Go to home page
    - Go to user profile

- Main Activity (login page)

This page allows the user to login, provided they have an account. They can provide their login credentials which will redirect them to a page with a list of all their grocery lists. There is also a signup button to allow new users to be brought to the signup page where they can create a new account.

    - Type username
    - Type password
    - Login (will bring you to display list activity)
    - Go to signup page (will bring you to signup activity)

- Signup Activity (signup page)

This page allows a new user to sign up with our app. They can provide their credentials to create a user for them. This will redirect them to the login page where they can login with their newly created account.

    - Type name
    - Type username
    - Type password
    - Signup (will being you to main activity)

- Display List Activity

This page displays all the grocery lists and grocery list templates that your user has created. There are two titles, lists and templates, and under both you can find all the lists and templates respectively. You can delete lists and templates you no longer need by clicking the delete

button. When you select one of the lists or templates, you are brought to a page with a list of all the grocery items that it contains. The button next to the titles will allow you to create a new list or template from their respective creation pages.

    - Go to create list page
    - Scroll through your lists
    - Delete a list
    - Select a list (will bring you to display item activity)
    - Go to create template page
    - Scroll through your templates
    - Delete a template
    - Select a template (will bring you to display item activity)

- Display Item Activity

This page displays all the grocery items that your user has created for the selected list / template. At the top is again a text field and a button to allow new items to be created. Duplicate or blank list names are not permitted and will prompt a pop up warning. You can delete items you no longer need by clicking the delete button. There is also a share button at the bottom of the screen that will allow the user to share the given list with their friends.

    - Type name of new item
    - Add new item (from name of new item)
    - Scroll through your items
    - Delete a item
    - Share list (will bring popup window to see who it is shared with)

- Create Grocery List Activity

This page lets the user create a new list. There is a text box to enter the name of the list and then you can select a template to create the list from. The template is basically a list of default items you wish to populate the list with. By default, a template with no items is selected. Once you click create, a list with the given name and template is created and you are brought to the display list page.

    - Type name of list
    - Select template
    - Create list (will bring you to display list activity)

- Create Template Activity

This page lets the user create a new list. There is a text box to enter the name of the template and then you can add items to the template. These are the items that you wish to populate a new list with. Once you click create, a template with the given name and items is created and you are brought to the display list page.

    - Type name of list
    - Add items
    - Delete items
    - Create template (will bring you to display list activity)

- User Profile Activity

This page lets you edit the user credentials and add friends. Once you click the edit button, you can click on a field, the name or password but not username, to edit it and then you click save to

save your changes. The add friends button will bring you to the add friends page. The logout button will let you logout and go back to the login page.

    - Edit name
    - Edit password
    - Press add friends button (will bring you to add friends page)
    - Logout (will bring you to login page)

- Add Friends Activity

This page lets you add friends. You enter the name of a friend in the textbox and press the add friends button to add them. They will only be added if the name exists. Once some friends are added you can delete them from the list if you wish to not be friends anymore.

    - Type name of friend
    - Add friend
    - Delete friend

Saguaro:

Saguaro is a web server created using the Spring Boot framework. It exposes two main categories of endpoints: user related and grocery list management related. The user related endpoints include ones for login, registration, logout, and adding friends. The grocery list management related endpoints support getting all list names for a user, getting the contents of a list, creating a list, saving a list, deleting a list, and sharing/unsharing a list. A user can also choose to create list templates, pre-filled with grocery items.

Authentication with Saguaro is token based. The client exchanges a username and password for a non-expiring token, which must be included with subsequent requests to protected resources in an Authorization header. Logging out invalidates this token. A client can register a user with a username, password, and name. Adding friends can be done by providing the username of the user a client wishes to add as a friend.

All grocery lists and items are protected resources, since they belong to specific users. Saguaro supports the fetching of just grocery list names belonging to a user, meaning more specific information for each list can be loaded lazily by Cereus. An endpoint for returning list names that a user owns as a dictionary mapping a unique list ID to the list name still exists for the purposes of supporting legacy code. In addition, Saguaro provides a “v2” list names fetching endpoint that returns more fine-grain information about whether a list is owned/shared and whether it is a template. The endpoint for fetching a single list returns a JSON object with the following properties: ID, name, and items. The “items” property of a grocery list is a list of string item names. Additionally, it is possible to create a new list by providing a list name. A unique ID for this new list is automatically generated. Saving a list is done by providing a JSON object with the following properties: ID, name, and items (identical to fetching a list). This operation is not additive; the provided list object will overwrite the existing list rather than appending to it. A client is able to delete a list by specifying a list ID. Lists can be shared by providing its ID and a username of a friend. On success, the new state of the grocery list is returned as a JSON object. Finally, Saguaro supports creating “templates”. These are essentially grocery lists,

however they can be used to quickly pre-fill other grocery lists on creation. Templates support all operations that can be done on regular grocery lists.

Data persistence in Saguaro is done using Hibernate. For convenience, a file-based relational database is used, rather than running a complete database instance.

Major Design Decisions

By far the largest design decision our group has made was to implement a separate server application to handle user management and data persistence. One of the reasons why we decided to undertake this project was because a core feature of our app is the ability for different users to share grocery lists. The best way to implement concurrent access to some shared resource is through a web server. Java also conveniently has the extremely powerful Spring framework, allowing our project to focus on implementing server logic rather than the specifics of running a server application.

Cereus:

The implementation of the adapters involved an important design decision about which library we use to send and receive our requests. Initially, we largely used the Java HttpClient, with some use of the Apache library to send and receive our HttpRequests. However, the Apache library was incompatible with the Android App upon integration of the app and the adapters, and required a migration to a new library. For this purpose, we chose the OkHttp3 library. For mapping the Json string returned by the server to the Entity object we are using, we made use of the Jackson library. The library helped implement most of the mapping, as we can simply enter in a class with the same fields as the Json string, and the ObjectMapper class from the Jackson library executes the rest.

A design decision made in conjunction with the Android app team was surrounding the coupling of the create and login functions. Initially, the create function created a new User in the server, and logged the User in. However, with conversations with the Android app team, the decision was made to deliberately separate the two functions because the app was built so that the creation of the user brings you back into the login page as opposed to the create user page directly logging you in. The separation ensures that the pages in the app are accessed in one sequence, from login to the page that displays your list, instead of both the create and login page directing to the page that displays your list.

Another design decision was to ensure that whenever there is a failure in the interaction with the server, we return null. When the adapters were first implemented, we noticed that an interaction with the server that results in a bad request does not explicitly show an error, aside from the status code returned. At first, the adapters ignored the status code, and the resulting object mapped from the response body was simply filled with null attributes. It was decided to instead return null and to look at the status codes and exceptions thrown to decide when to return null,

instead of just returning an object filled with null attributes. For Phase 2, we realized that even better than returning null, is to actually throw informative exceptions that can be handled in the UI and used to provide information of what exactly failed in the code. Thus, we implemented exception handling for the Adapter classes with our custom Exception classes, and when they are caught in the UI, they provide messages to the user about what exactly failed.

Saguaro:

A major design consideration for Saguaro was how to structure the database relationships between entities. This decision would be impactful to both performance and functionality. In particular, the relationship between grocery lists and grocery items was a point of contention. One of the features that Cereus hopes to have in the future is the ability to provide recommendations for items to add to grocery lists, based on what other users are adding. Then the application must have the ability to recognize that the same item has been added to multiple lists. In the end, preemptively supporting this functionality was the reason behind having a many-to-many relationship between grocery lists and grocery items, even though intuitively a grocery item should not contain grocery lists.

CLEAN Architecture

Cereus:

Cereus largely adheres to CLEAN architecture by dividing the project into the four layers, entities, use cases, controllers, and UI. For entities we have users, grocery lists, and grocery items which have no dependencies. The next layer up we have use cases, which consist of the WebAuthAdapter and WebGroceryAdapter, which depend on the entities, as they handle them to manage users and grocery lists. However, they interact with our database, which is a violation of Clean Architecture, as that is an outer layer. We felt that the interaction between the database and our adapters was unavoidable and we followed a similar example in class where the professor brought attention to and allowed this same violation. We created the interfaces AuthAdapter and GroceryAdapter to separate the interactions, and allowed us to swap the database in this phase from an EntityRepository Class to a server. The next layer up in the controllers (or systems) which depend on each other, through a facade, and on the use cases. Finally we have the UI, or the Android portion of the app, which depends solely on the facade which would be the controller layer. For the most part each class is only dependent on the layer below and for some rare classes, they depend on each other. This has allowed our development to be very efficient as any bugs are for the most part isolated to that class and do not require a refactor by other classes.

Saguaro:

Being easily modifiable and extensible was one of the main design goals of Saguaro, and it does this by implementing a CLEAN architecture. Functionality is divided into three layers:

controllers, services, and entities. An additional repository component is defined to perform data access.

The actual interfacing with the web is functionality provided by Spring, meaning that the outermost layer of specifically the Saguaro application is the Interface Adapter layer. Saguaro makes use of two controllers: UserController and GroceryController. The function of these two classes is to take in deserialized raw HTTP input, and adapt it into a form appropriate for some particular operation. In particular, the controller may perform some type of input validation, and then it compiles relevant pieces of information and passes it to the services layer. Importantly, the controller never directly manipulates data or performs any sort of logical processing. The only job it has is to look at all the input data it's given, and filter out inputs irrelevant to the application. The controllers are dependent on entities, however this is not a problem from a CLEAN architecture design perspective, since the dependency only exists to facilitate easy serialization and deserialization; controllers do not depend on entity implementations, only the entity type itself.

As mentioned, the controllers then delegate logic to services in the application, which would be part of the Application Business Rules layer as defined in the CLEAN architecture outline. Saguaro has two services: UserService and GroceryService, whose jobs are to take in data and perform some type of manipulation. This is where all core logic occurs in the application. The services are able to directly manipulate entities as needed, and interface with repositories to store and retrieve entities. Notice that the services do not depend on the layers above them, meaning that they can be used regardless of specific controller configurations, satisfying the CLEAN architecture dependency rules.

The innermost layer of Saguaro consists of entities, of which there are four: User, Role, GroceryItem, GroceryList. These classes encapsulate data relevant to the entity. They also define rules specific to the entity, for example that usernames cannot be null values. These entities are not dependent on any other part of the application, meaning that changes in those parts will never affect how entities are defined.

SOLID Design

Cereus:

1) Single Responsibility Principle

It sticks to the Single responsibility Principle by separating classes that have different actors from each other. For example, we have split the controller layers into two controllers: UserSystem and GroceryListSystem. The UserSystem handles creating user, login, logout and the other methods that are related to handling the user, while GroceryListSystem handles creating grocery lists, getting grocery lists' items and names and other methods that are related to managing grocery lists. We separated these two controllers because they have different

responsibilities. Thus, if we have to change the methods relating to grocery lists, the functionality of the user-related methods are not affected.

2) Open/Closed Principle

Cerceus adheres to the Open/Closed Principle because of our use of interfaces to separate the layers of our infrastructure. By simply creating new implementations for interfaces, instead of changing the main functions of our controller, we were able to easily change our UI from a Command Line Interface to an Android App and our database from an EntityRepository class to an online server. The controllers were designed with the knowledge that the UI and database it interacted with were going to be replaced, thus allowing for a painless extension of our UI and database, while leaving the controllers closed for modification.

3) Liskov Substitution Principle

Cereus adheres to the Liskov Substitution Principle by frequent use of interfaces where necessary to separate our layers. The interfaces AuthAdapter and GroceryAdapter allowed us to easily replace interaction with a repository class with interaction with a server, because the implementation of the interfaces, WebAuthAdapter and WebGroceryAdapter, fit into the controllers UserSystem and GroceryListSystem without requiring any major modifications. Thus, most of the controller code written in Phase 0, which had been properly established, easily carried over to Phase 1 because our architecture was upholding the Liskov Substitution principle.

4) Dependency Inversion Principle

In Phase 0, the Controller Layers were built with a strong focus on separating them from the User Interface and Database, because those were temporary and would be switched out with an Android App and server, which we have now implemented. For the databases, the interfaces AuthAdapter and GroceryAdapter allowed us to easily switch out the classes that relied on EntityRepository class to classes that interacted with our server. The UI layer, which implemented a Command Line interface earlier, was easily switched out to an Android App without having to change a single method in the Controllers because the main functionality was established, and only what methods we call have to change. Thus, the Dependency Inversion principle is upheld in Cereus.

5) Interface Segregation Principle

Our main use of interfaces is within the use case layers, with interfaces AuthAdapter and GroceryAdapter that must interact with the database to store and manage users and their grocery lists. We made the decision to separate our interaction with the database to have one interface that handled user management and one that handled grocery list management because the functionalities are not related. For example, if the Adapters were one, then we would have a login method alongside a method to retrieve grocery items. Hypothetically, if anybody wanted to use our login method, they would have to implement functionality for grocery lists, thus requiring implementation of methods that are irrelevant to the desired function, and making our design clunky and hard to modify. Thus, we have separate interfaces that ensure the Interface Segregation Principle is fulfilled.

Saguaro:

Saguaro was designed to follow the SOLID design principles as much as possible. It adheres to the Single Responsibility Principle by splitting different areas of functionality into different classes, in addition to having layered classes of functionality. For example, all user related endpoints are handled by a dedicated UserController, and user related logic such as login and registration is handled by the UserService. The same is true for grocery list management operations. This means that should, for example, the logic for login require changing, only the user related classes require modification. None of the grocery list management classes share responsibilities with or depend on these classes, so that code remains unaffected by changes.

The design of Saguaro follows the Open/Closed Principle by virtue of being a Spring application. Additional functionality can easily be added to the application by extending Spring features. For example, additional endpoints can be defined by creating a new controller class and annotating it with @RestController. Similarly, we can create more services and repositories by leveraging Spring annotations, as needed.

Saguaro is aided in following the Liskov Substitution Principle once again by Spring. Much custom functionality is created by extending base classes provided by Spring. Custom token authentication is implemented in part by extending the AbstractAuthenticationProcessingFilter class, and then overriding some relevant authentication methods. Importantly, the child class only modifies the underlying implementation of the same behaviour as the parent class, meaning that any other filter could be substituted without altering the program's desired property of requiring authentication.

Interfaces are not heavily made use of in Saguaro, somewhat diminishing the relevance of the Interface Segregation Principle. However, it is interesting to note the use of this principle in the design of the Spring framework. Interfaces provided by the framework are extremely specific, and define methods relevant only to a single feature. This is seen in action in the TokenAuthenticationProvider, which implements the Spring interface AuthenticationProvider. This interface is solely responsible for defining an authentication implementation, and contains two methods: "authenticate" and "supports", the latter of which simply specifies what types the implementation can work with.

Finally, one weakness of Saguaro is that it does not adhere to the Dependency Inversion Principle very well. It does make full use of Spring's inbuilt dependency injection, wherein classes are not themselves responsible for creating their dependent objects. However, since most dependencies within the application are quite linear, further abstraction using interfaces was deemed unnecessary. Although this does mean slightly more coupled classes, the extra verbosity that would have been introduced was less desirable given the current small scale of the application.

Packaging Strategies

Cereus:

The Cereus module uses a “by layer” packaging strategy. This strategy uses the four packages entities, adapters (use cases), systems (controllers), and ui, which separate the project into the layers of CLEAN architecture. This was done so that classes within the same layer of CLEAN architecture would be in the same package and thus be able to access each other and not the other layers directly. We went with this packaging strategy because it allowed us to clearly define the functions and scope of our project, and divide the workload between layers. This meant that each person would be working on one package for the most part and so when changes were made, others would not be affected by those changes. Moreover, they really emphasized the dependency rules of CLEAN architecture for us, and allowed us to easily maintain that structure.

Saguaro:

The Saguaro module uses a “by layer” packaging strategy. This strategy was chosen so classes with the same “levels” of functionality can be grouped together. From an organizational standpoint, this makes looking for files extremely easy. For example, if changes need to be made to the way data access occurs, then the relevant files will be in the repository package. From a technical perspective, the “by layer” packaging strategy allows layer-internal classes to be hidden from other layers. For example, the CustomGroceryListRepository interface should only ever be used by the GroceryListRepository, in the same layer. Marking it package-private means that the controllers and services cannot access this interface, as intended.

Design Patterns

Cereus:

One of the design patterns that we have used is the facade design pattern. We have two controller classes namely UserSystem and GroceryListSystem where UserSystem is responsible for controlling the user and GroceryListSystem is responsible for controlling the grocery lists and items. We have a facade class named UserInteractFacade which wrapps these two controllers into one class. This facade class has the two controller classes as its attribute and its methods are simple and just call the corresponding method in each controller. This facade class has been used in the android part. If we have not used the facade design pattern then the activities in our classes would have to have both UserSystem and GroceryListSystem as their attributes. However, since we have implemented a facade class, the android activities have UserInteractFacade as their attribute.

Cereus makes use of the Adapter Design Pattern in the use case class. The use case class is made up of two adapter interfaces, the AuthAdapter and GroceryAdapter, whose purpose is to interact with whatever database we are using, and to transform the data into a format that can be used by the controller. The WebAuthAdapter and WebGroceryAdapter are implementations that interact with the server, and send and receive Json Objects that are then mapped to entities that can be used by the controllers to login, create and logout users, and create and edit grocery lists. Thus, controllers are the clients, and the service we are trying to access is the server implemented in the Saguaro module. The AuthAdapter and GroceryAdapter serve as our client interface, and the WebAuthAdapter and WebGroceryAdapter are our adapters that wrap the server so that it is essentially hidden from the controllers, or our client.

Accessibility

1. **For each Principle of Universal Design, write 2-5 sentences or point form notes explaining which features your program adhere to that principle.**

   Principle 1: Equitable Use

   This app is intended for making grocery lists as a free app on an app store. There is no part of the design, privacy, or security that would exclude people. There is an option to add friends but not having friends to add on the app does not subtract from the app as all features are still available to every user.

   Principle 2: Flexibility in Use

   You can share your list with others. You can create a default list of items, aka templates. There are options to sort the lists and items in alphabetical or reverse-alphabetical order to give the user some options. In the future we could have more sorting options. The app also responds to the phone's system dark mode status. If the phone has the dark mode on, the app page elements will reflect that.

   Principle 3: Simple and Intuitive Use

   Most of the elements on each page follow the same style from page to page so that there is little confusion as to what each element would do. Also each element looks like what it will do. For instance, buttons look like traditional buttons, text boxes look like text that is editable, and plain text does not look editable. This avoids confusion for the user and allows them to use the app intuitively the right way.

   Principle 4: Perceptible Information

   There are many titles for any page that has lists of some elements and for descriptions of some pages. Each button is labeled with its use and text boxes have hints to tell the user what they are to enter. There are some parts of the app where

clicking a button does not do anything as there is some issue or that action can't take place. When this occurs, we display a text popup that will tell the user what happened so they don't wonder.

Principle 5: Tolerance for Error

One feature we could have included for this principle was to have popup warnings for when users try to delete lists, templates, or items. This would keep the user from accidental deletions and would make it clear to them what action they are about to take.

Principle 6: Low Physical Effort

The main action that a user will make when using this app is clicking buttons and typing text and so it takes very little physical effort to do actions. Also deleting lists and creating lists do not take many clicks to do. Many actions take one or two clicks to do.

Principle 7: Size and Space for Approach and Use

We spaced the UI elements apart so that they were easy to click while making it hard to misclick. It allows the user to take the actions they intended to do. The elements are very reachable as the user can click anywhere on the screen.

2. **Write a paragraph about who you would market your program towards, if you were to sell or license your program to customers. This could be a specific category such as "students" or more vague, such as "people who like games". Try to give a bit more detail along with the category.**

The intended target when designing was for students who live together as this app would allow them to make grocery lists that could be shared between them so that everyone can be on the same page. As we developed the app it was apparent that anyone who uses lists when going out for groceries, so most if not all adults, could use this app. If we market towards all adults, we could focus more on the features that assist in making lists and making the shopping experience smooth. If we market towards students, we could put our focus towards the friends and make it almost like a grocery list social media.

3. **Write a paragraph about whether or not your program is less likely to be used by certain demographics. For example, a program that converts txt files to files that can be printed by a braille printer are less likely to be used by people who do not read braille.**

People who don't rely on grocery lists and buy whatever they see, or buy a set list of things they remember, would not benefit or would not find a particularly good use for this app. It is unlikely that these people would change the way they shop solely based on the existence of

our app. Also, people who do not go out shopping for groceries, like kids, would not find a good use for this app either.

Testing

The WebAuthAdapter and WebGroceryAdapter rely on the server to return the object specified by their description. Thus, we could not write unit tests for the adapter classes, as there is no way to feasibly ensure the functionality of the code without either running the server or using a Mocking framework.

Saguaro is extensively tested, using a combination of unit and integration tests. In general, it uses unit tests to confirm desired functionality for specific methods, while integration tests are used to ensure data persistence is working as intended. Trivial functions such as getters and setters are assumed to be correct and are left untested. The robust testing ensures that new functionality behaves correctly, but also that in the process of development old behaviour is maintained.

Refactoring

In the implementation of WebAuthAdapter and WebGroceryAdapter, there was duplicate code used to send the request to servers for each command. After preliminary attempts to refactor some of the User methods, for phase 2 we overhauled the refactoring by creating a new class HttpUtil. The new class factored out two methods, the process of mapping out the input Json string into a RequestBody for the HttpRequest, and making the http request to the server itself. Both were operations that created a lot of duplicate code and cluttered up the actual differences between each method. Thus, the refactoring was really beneficial to the testing and the clarity of the code.

For the controllers, we initially had a grocery list system and user system that the UI layer would call depending on a given function they needed. This was very inefficient and not very clean as the UI layer would have to know which controller had which method and there were values passed from one to another through the UI layer. To fix this we refactored the controllers to be a part of the user interaction facade. This allowed the UI to communicate only with the facade and the facade would handle the calls to the two controllers and the communication between the two. This allowed the development of the android side much smoother and cleaner.

There was also a simple refactoring on the android side that helped the project be cleaner. We have many pages in the android app and for each we need a NameActivity file where it handles the logic of the specific page. The problem was that many of these files had repeated code and similar actions. This problem became larger when we wanted to handle actions for only specific pages. For instance we wanted the toolbar buttons to redirect to the given pages when logged in but for users on the login or signup page we wanted to tell them to login before using the

buttons. To solve this, we made an abstract activity class that would hold the duplicate code and have methods for the display and the button actions of the toolbar. This meant that for pages that we did not want using the toolbar functionality, we could override the method and print a warning to the screen.

Use of Github Features

For our project we made sure that we utilized github as much as possible to make our work come together smoothly. The main way we did this was through branches. Any new feature, fix, etc would have a new branch and for the most part everyone's work was done in separate branches. We would notify our group members of branches that were complete and ready to merge and get them to review our pull requests. This allowed everyone to know what was being added to main and catch any errors, or poor design decisions before they reached the main branch.

Progress Report

Individual Contributions:

Charles implemented new server functions such as being able to add friends, share lists, and create templates. Most server-side code he wrote was rigorously tested and documented. In addition, he was responsible for hooking up the Android UI to backend logic. He refactored existing classes to throw exceptions on error, and handle them where appropriate.

Major server-side PR:

https://github.com/CSC207-UofT/course-project-cactus/pull/65

Added functionality to interact with friends.

Major Android client-side PR:

https://github.com/CSC207-UofT/course-project-cactus/pull/76

The work on this branch includes work done on the UI by other group members. Charles worked on a major refactor of logic classes, including abstraction of common operations and adding exception throwing and handling. The major feature added in this PR was a user profile page, which allows a user to edit their details and add friends.

Ronit updated the use case layer, implementing the methods in the adapters, WebGroceryAdapter and WebAuthAdapter, that add friends and share lists while communicating with the server alongside Charles. He also updated the UI for the list pages, adding the list name and whether the user was the owner or editor of the shared list. Moreover, Ronit wrote code in the controller to check for the owner.

Major pull request for initial implementation of WebAuthAdapter:

https://github.com/CSC207-UofT/course-project-cactus/pull/38

This pull request represents a significant contribution to the team because it involved implementing a crucial part of the use case layer, and is the code that allows the app to communicate with the server for User based operations.

Major pull request for initial implementation of WebGroceryAdapter:

https://github.com/CSC207-UofT/course-project-cactus/pull/51

This pull request represents a significant contribution to the team because it involved implementing a crucial part of the use case layer, and is the code that allows the app to communicate with the server for Grocery based operations.

Caleb updated the android UI look for all pages. He designed the look for view elements by creating custom shapes and layouts. He also added the toolbar to the top of every page. He implemented the AbstractActivityClass and the refactoring of all other activities to implement this class. He helped debug android logic. He added the CreateListActivity, CreateTemplateActivity, and AddFriendActivity.

Link to major pull request for Caleb:
https://github.com/CSC207-UofT/course-project-cactus/pull/67

In this pull request, I was able to add the AbstractActivity class which helped future android activities. Another thing this pull request did was overhaul the android look to make the app more appealing and also abide by the accessibility principles.

Dorsa helped with the implementation of UserProfileAcitivity which is an interface for changing the name and password of the user. She helped with the design and some part of the logic for this Android activity. She also helped with the design and implementation of AddFriendActivity which is responsible for adding friends. She also implemented and designed sort options for grocery items, grocery lists and grocery templates.

Link to major pull request for Dorsa:

https://github.com/CSC207-UofT/course-project-cactus/pull/86

In this pull request, she added the sorting options for grocery lists, grocery items and grocery template where there are buttons for sorting in each corresponding activity and the sorting method and the button text will change every time a user clicks on the sort button.