<WasteLess A2>
Analysis and Design Document

Student: Andrei Rusu
Group: 30431

Table of Contents

1. Requirements Analysis

1.1 Assignment Specification

The purpouse of this assignment was to create an application using any language that would fullfill a few requirements, detailed below.

Functional Requirements

The app needs to have a login capability, where a user can authenticate and add grocery lists with the items they want to consume. The system should, based on this lists, calculate how much food is being wasted by the user, based on their calorie burn rate, notify them about expiring items and offer options for the excess food to be donated to charities where possible.

The application needs to work with a database, within a Client-Server architecture. This means that only the server has acces to the database and performing tasks, while the client only performs requests to this server and displays data.

More details in the UML class diagram section.

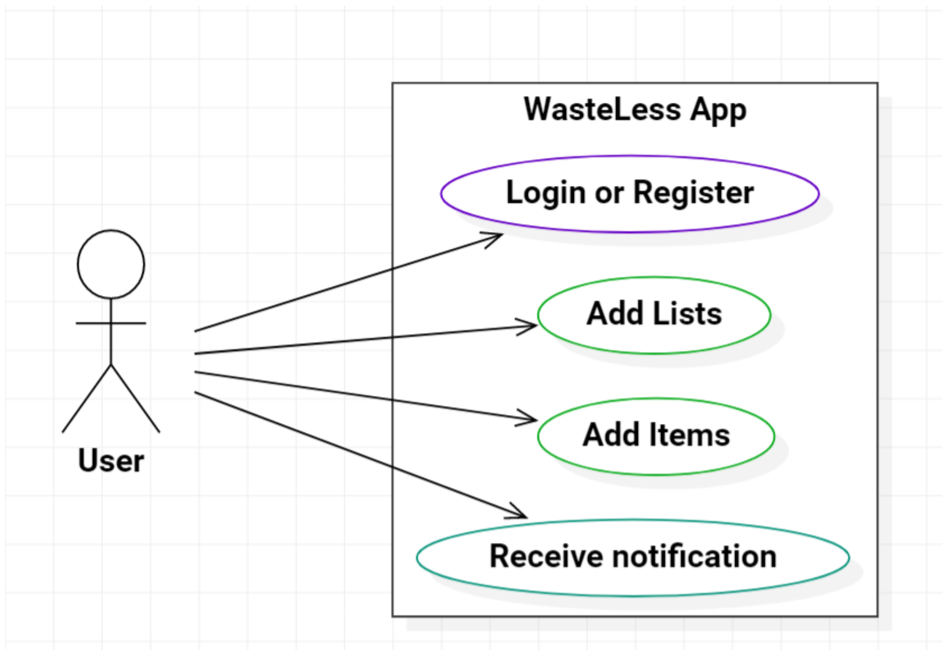
1.2 Non-functional Requirements

The application was required (if not explicitly, then by common sense) to be maintainable (hence the modularity), testable, have good usability, and also, from a later development standpoint, to be scalable and extensible further on. Many of these requirements are fullfiled by using a good architecture, logic and coding style, which were employed during the course of this assignment.

2. Use-Case Model

Use case: <Add an item to a list>
Level: <use-goal level>
Primary actor: <the user of the application>
Main success scenario: <Start the application, login or register, add a new list or select an existing one, fill out the new item details and add it to the current list>
Extensions: <failure if wrong data is input>
]

The use-case diagram is presented below:



3. System Architectural Design

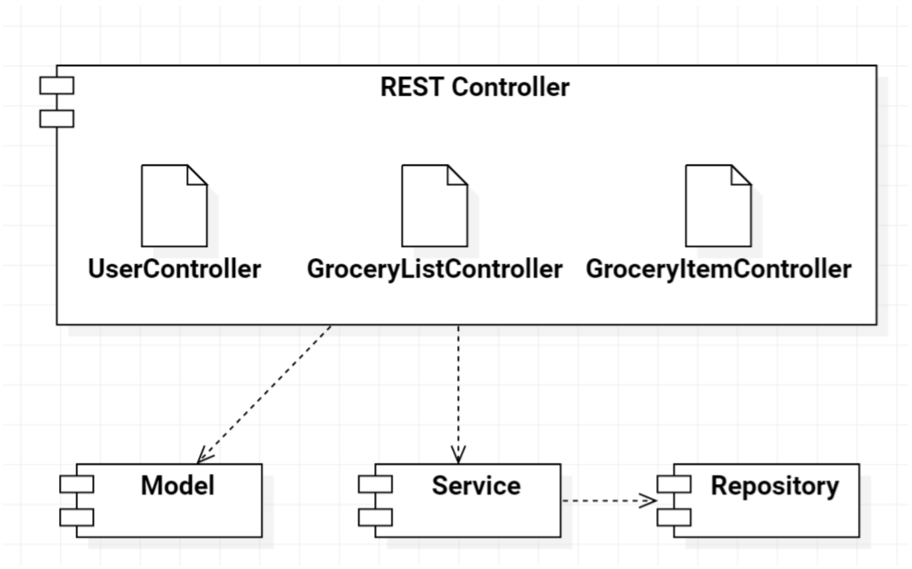
3.1 Architectural Pattern Description

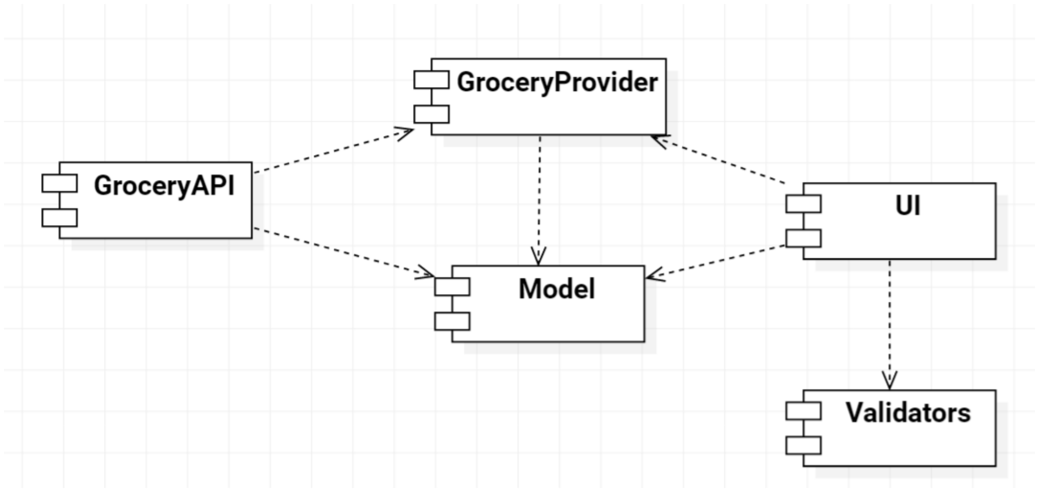
The system is based on a client-server architecture:

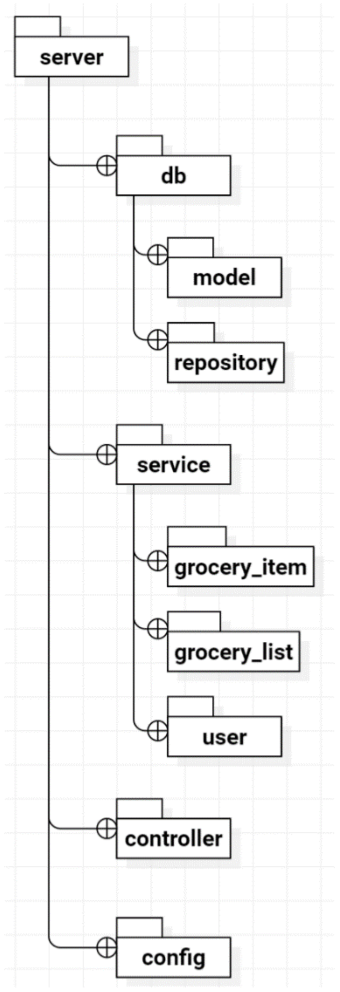
- The **Server** is a Java application based on the Spring MVC framework. It includes an ORM (the same Hibernate I used in the previous assignment), dependency injection containers and, more importantly, request mapping capabilities
- The **Client** is an Android application written in Kotlin runnning Retrofit2(for communicating with the server), Koin (Kotlin dependency injecton framework) and various Android Jetpack components such as Navigation (for navigating through the screens)

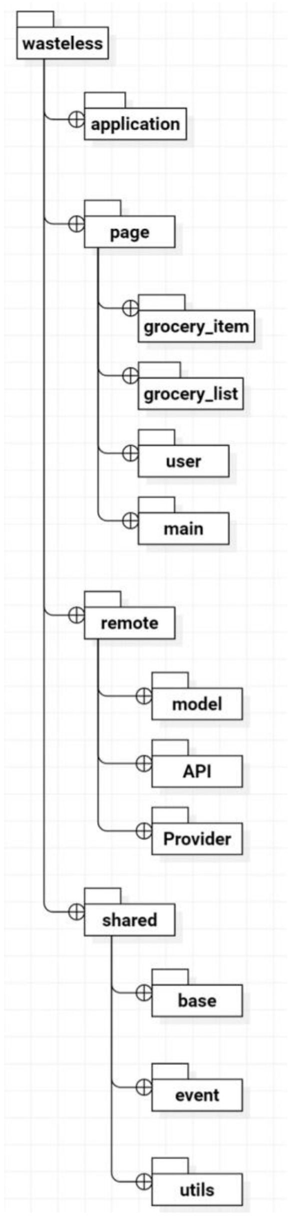
3.2 Diagrams

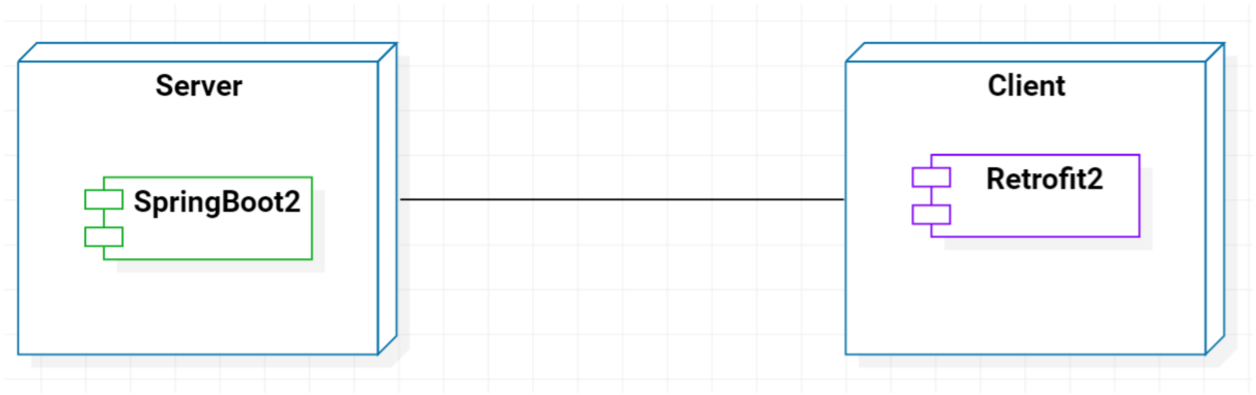
The component, package and deployment diagrams are shown below, for both the client and the server:





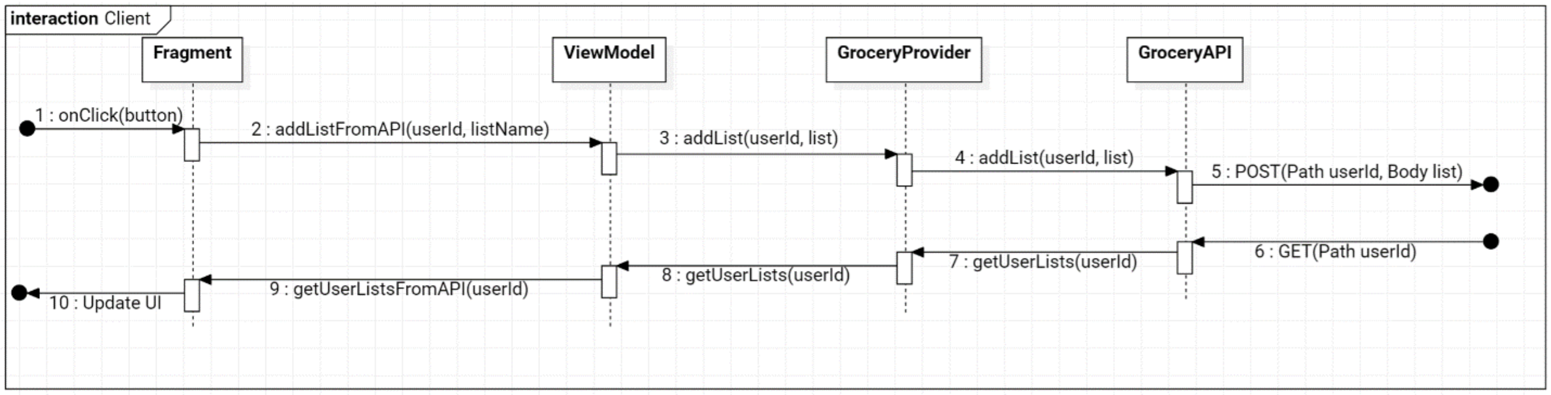

server
db
model
repository
service
grocery_item
grocery_list
user
controller
config

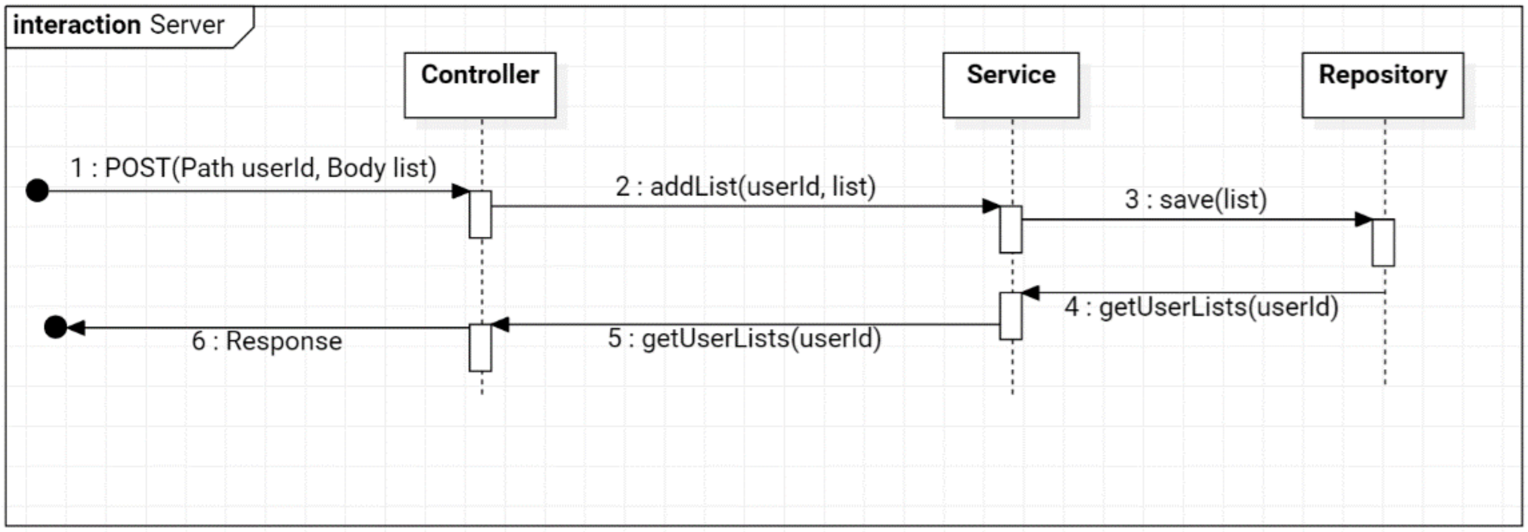

wasteless
application
page
grocery_item
grocery_list
user
main
remote
model
API
Provider
shared
base
event
utils


Server
SpringBoot2
Client
Retrofit2

4. UML Sequence Diagrams

The following diagram illustrates the scenario of a user adding a new grocery list and receiving the new lists.





5. Class Design

5.1 Design Patterns Description

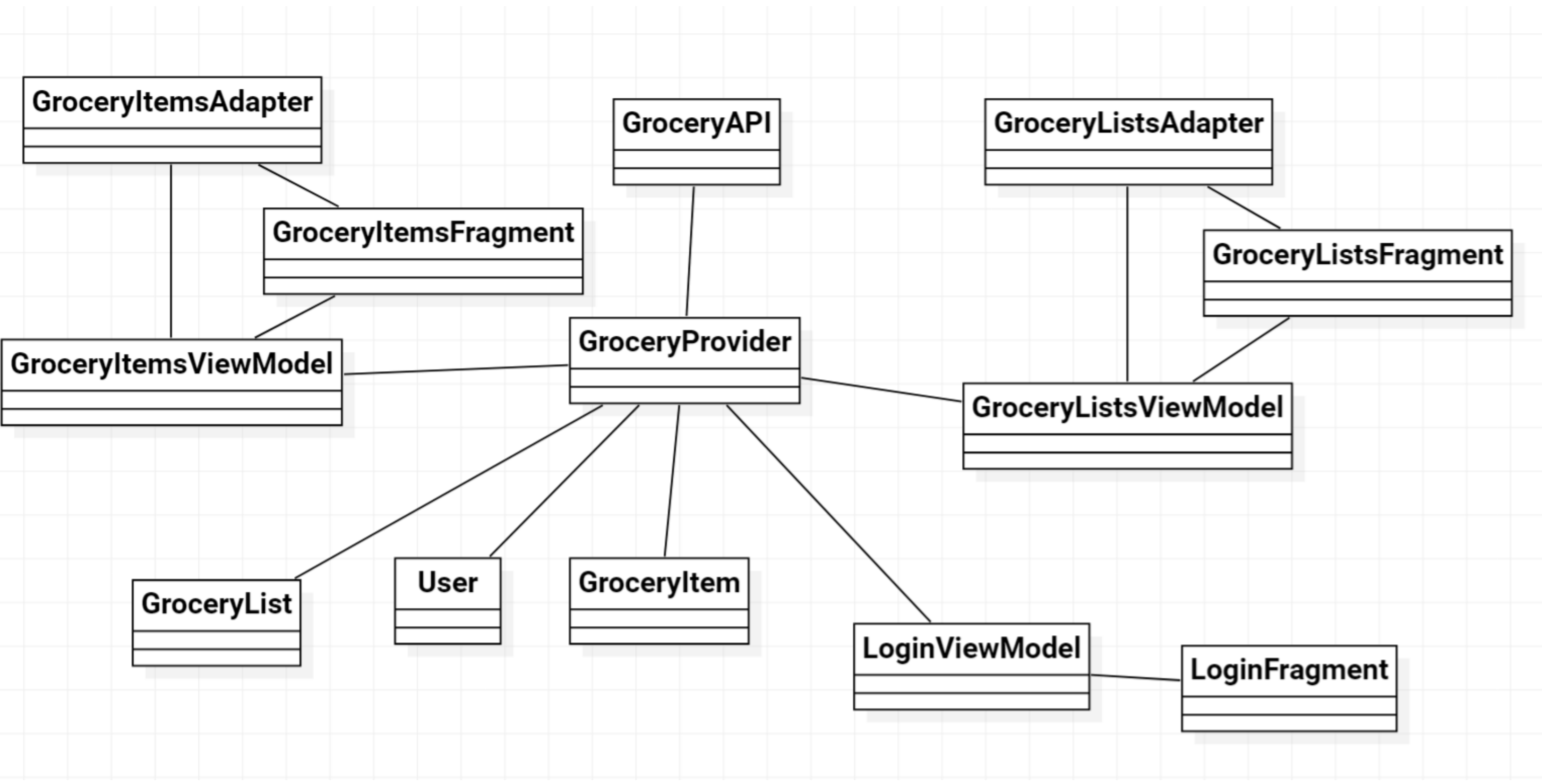
The Observer design pattern is used for notifying the user when their waste levels are too high.
In this pattern, a subject is observed by one or more observers that get notified when the state of the subject changes. This way, events can be triggered, and actions performed when something happens.
In this project, the observer pattern is implemented on the client-side, in the Android application. Here, whenever a new grocery item is added to a grocery list, a call to the server is made, which computes and return the waste levels and stores them in a special type of data type, called LiveData. LiveData is an Android wrapper class that wraps desired data types into observable objects and allows event-driven actions to be performed on them. We can use one of the “observe” methods of this class to know when the underlying data has changed. When the LiveData’s value is updated, the method is triggered and we can do something about it, like display a toast notification (a small temporary notification).
Furthermore, this design pattern is used in other parts of the application, namely in settings the adapters for the recycler views (the lists in the UI). This can be seen in the *setupObservers* method of the fragments.

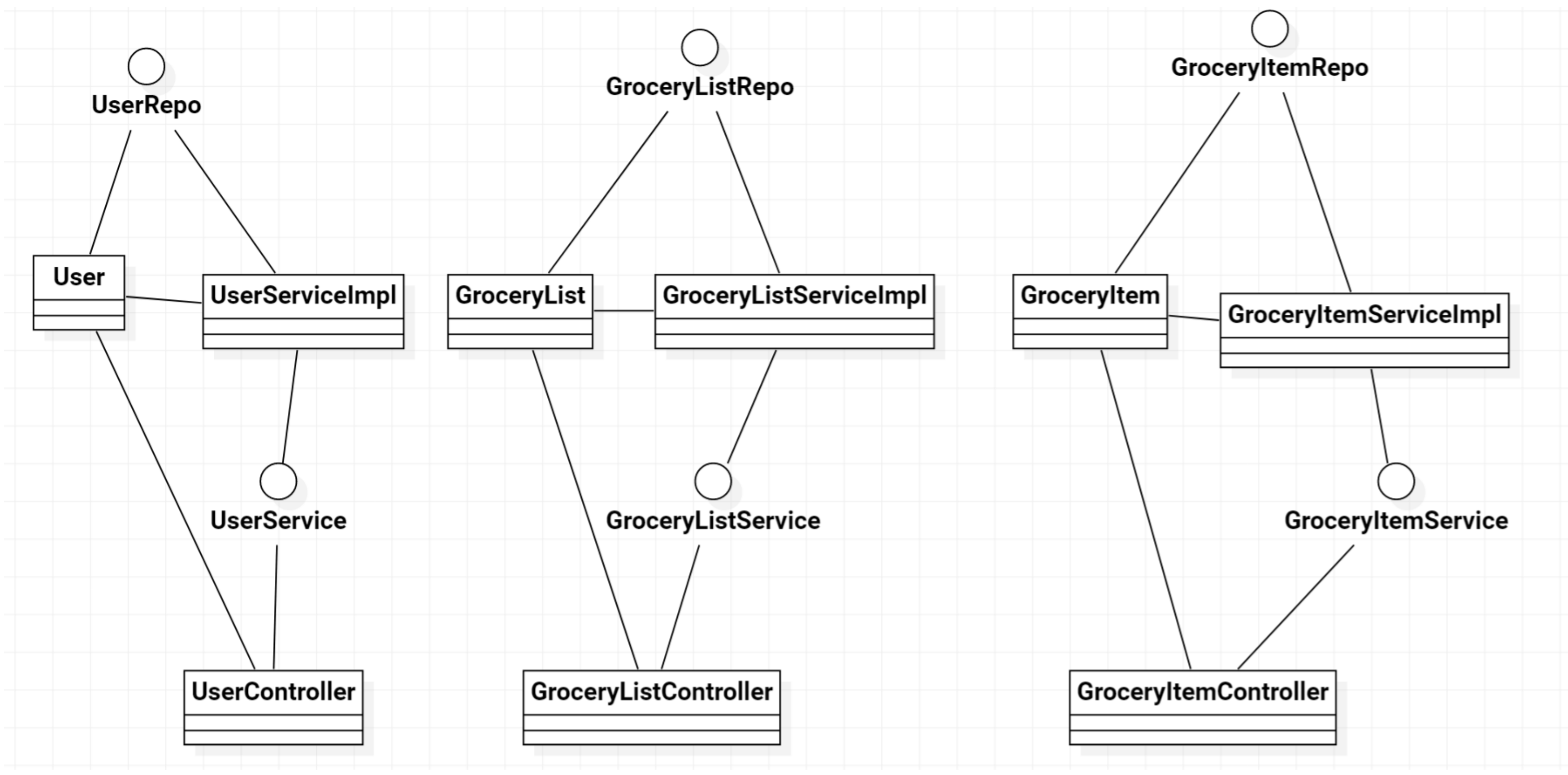
5.2 UML Class Diagram

The main feature of this architecture is the decoupling of the data manipulation from the data visualisation. The client handles data visualisation, and gets that data by performing requests to the server in a RESTful manner.

The Spring Server app’s main components are the Model, Service, Repository and REST Controller. The Model is the data of the application, meaning the User, their grocery lists and their grocery items. The Service is the interface (and implementation) thtat specifies what types of operations can be performed on this data. Methods in this interface use the Repository to perform database operations. The REST Controller is the class which maps, using annotations, the requests coming from the client to methods declared in the service class. Here, the URI of the request is specified, along with the request method, which can be of type GET (for obtaining information), PUT (for adding something new), POST (for modifying something already present) and DELTE (self-explanatory). These classes define the behaviour of the server and the data that is transmitted between it and the clients. They are defined for every model separately.

The Android app’s main components are the Model again, the UI components, the Provider and the API. The Model corresponds to the one on the server-side. The UI is made up of Activities and Fragments (lightweight UI components) that are used to store data and display it to the user. Data is stored in ViewModels and displayed in Fragments. The API is the bridge between the application and the server. It uses Retrofit2 to send requests to the server and obtain the response. The Provider is the bridge between the ViewModels and the API, and specifies asynchronous methods for getting information from the API. This can be visualised in the sequence diagram above.
Furthermore, the Adapters seen in the class diagram act as a way for fragments to display list data in a custom format, readable by the user.


GroceryItemsAdapter
GroceryAPI
GroceryListsAdapter
GroceryItemsFragment
GroceryListsFragment
GroceryItemsViewModel
GroceryProvider
GroceryListsViewModel
GroceryList
User
GroceryItem
LoginViewModel
LoginFragment


UserRepo
GroceryListRepo
GroceryItemRepo
User
UserServiceImpl
GroceryList
GroceryListServiceImpl
GroceryItem
GroceryItemServiceImpl
UserService
GroceryListService
GroceryItemService
UserController
GroceryListController
GroceryItemController

6. Data Model

The data is structured in the following manner, using data classes:

The User

- ID
- first name
- last name
- password
- a caloric intake goal

The Grocery List

- ID
- name
- user index

The Grocery Item

- ID
- name
- quantity
- caloric value
- the purchase date
- the consumption date
- the expiration date - hopefully chronologically in that order :)
- a list index

7. System Testing

I performed unit testing on the server-side by adding users and then deleting them, adding lists, adding items to lists, and then checking wheter they had really been added.
I performed integration testing on the client-side Android app by testing the same operations and seeing if they succesfully completed on the server side, by checking the database.

8. Reference

https://spring.io/projects/spring-boot
https://www.tutorialspoint.com/spring/
https://spring.io/guides
https://square.github.io/retrofit/
https://insert-koin.io/