<Wasteless Assignment 3>

Analysis and Design Document

Student: Iacob Andrei Ștefan

**Group:30431**

Table of Contents

1. Requirements Analysis 3

1.1 Assignment Specification 3

1.2 Functional Requirements 3

1.3 Non-functional Requirements 3

2. Use-Case Model 3

3. System Architectural Design 3

4. UML Sequence Diagrams 3

5. Class Design 3

6. Data Model 3

7. System Testing 3

8. Bibliography 3

1. Requirements Analysis

# Assignment Specification

Wasteless3 is a software application built using the client-server architecture. Both parts are developed in Java, making use of GraphQL for querying . It provides users a way to better organize their grocery lists, entering all the products that have been bought into the application . The products from all the lists will then be available for visualization and the user will be able to perform multiple operations on them : check that a specific item has been consumed (and when) , set a goal regarding how many calories he/she intends to consume daily, see when the food that is available exceeds the values needed for his goal. Also, the user can see weekly reports showing how much food has been wasted in the last 7 days ( what products and how many calories). In order for the application to be complete, a person is able to donate any food that he/she would like to (the application will make suggestions in this sense) to local charities. The application can be used by multiple users , as it also provides a sign up system . So new accounts can be made and once you are logged in to the app, you are free to use any of its functionalities.

# Functional Requirements

Some of the main functional requirements of the application:

-New user registration

-Authentication

- New grocery list creation , adding products to the new list

-Specifying for each item the name ,caloric value, expiration date ,purchase date, quantity

-Visualizing all the products that have been purchased

-Having the ability to donate excess foods to local charities

-See reports of how much food is wasted weekly

-Setting a goal regarding the number of calories the user intends to consume daily

-Sending reminders when food waste levels are too high based on ideal burndown rates

# Non-functional Requirements

● CQRS architecture, mediator pattern to handle requests

● decorator pattern for changing the color of the report (green for above the ideal

rate and red for under)

● Data stored in a database

● Client-Server architecture

● All the inputs of the application validated against invalid data before submitting

the data and saving it in the database.

2. Use-Case Model

Use case: donate food item

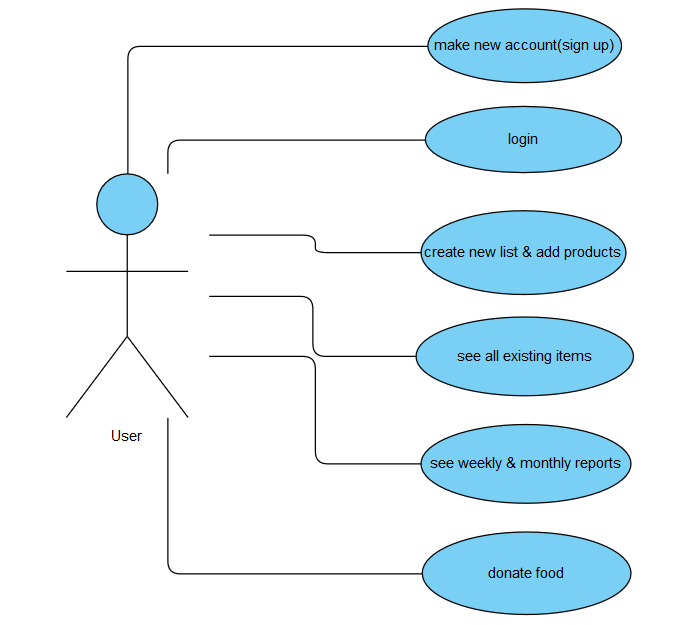
Level: user-goal level

Primary actor: user

Main success scenario: The logged in user chooses from the main menu the button which directs him to the Donation section. Now the user is notified which items are due to expire in the next 2 days . He is able to choose one of the items and donate it . This operation will remove the item from the database, as we consider that it is no longer in his possession.

Extensions: The user must own food items (and have them in the application , so say) to be able to donate. Otherwise the items list will be empty.

]



3. System Architectural Design

**3.1 Architectural Pattern Description**

For the development of this application the client-server architectural pattern has been used. On a ‘micro’ level , the client side is developed in Java and respects the layered architecture(presentation , business, data).

The client-server architecture is a computing model in which the server hosts, delivers and manages most of the resources and services to be consumed by the client. This type of architecture has one or more client computers connected to a central server over a network or internet connection . Client computers provide an interface to allow a computer user to request services of the server and to display the results the server returns. Servers wait for requests to arrive from clients and then respond to them.

Also, the client-server architecture is a producer/consumer computing architecture where the server acts as the producer and the client as a consumer.

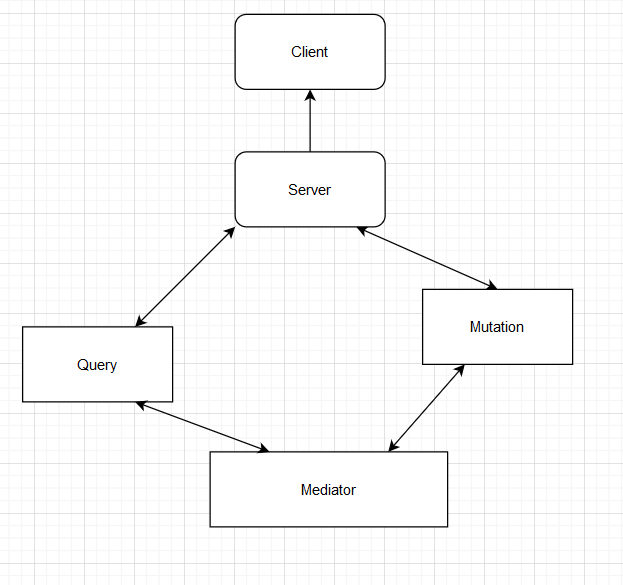
Both parts have been developed in Java and the connection between them is enabled by using GraphQL for querying. Json requests are sent from the Client to the Server , either for a query or a mutation. The ‘concerns’ are logically splitted by default in GraphQL on the server side, as we have to distinguish from the beginning the queries( asking for some elements ) from mutations (update, add , delete) .

**3.2 Diagrams**

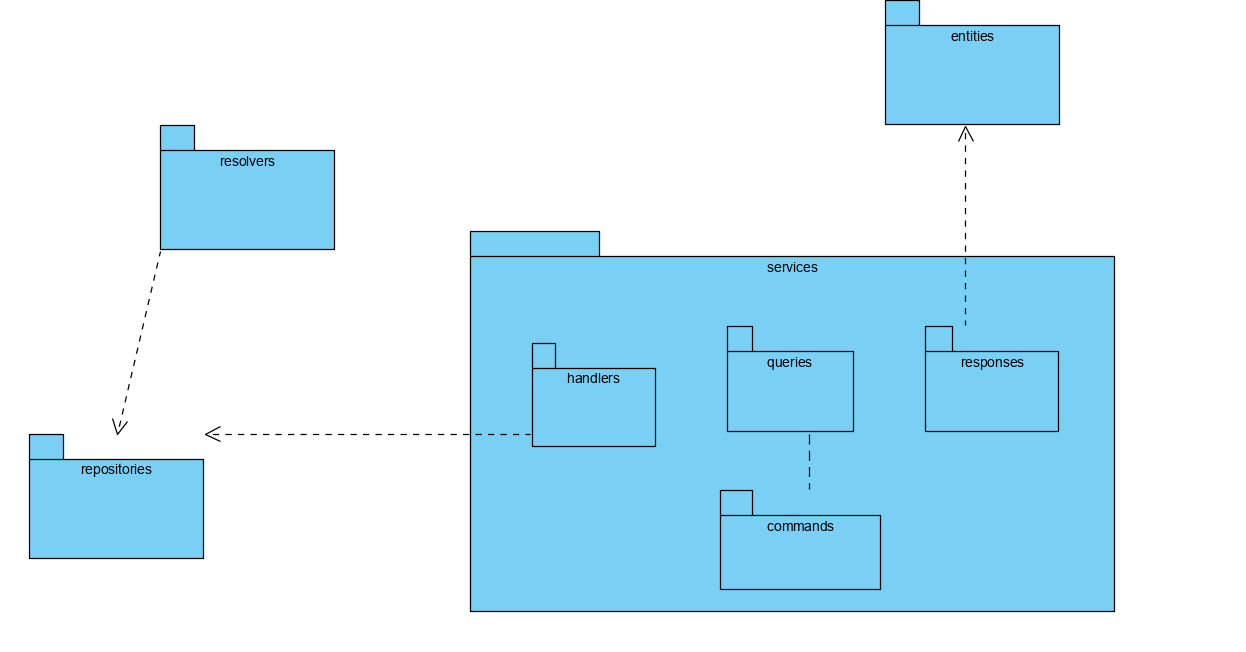


Although it is not a very specific conceptual diagram , it shows the point of a client-server architecture. The client is using an interface through which he is basically sending queries that will later be solved in the server side. This communication , in my application, is performed via the GraphQL with queries sent as JSON. The server is ‘placed’ on localhost port 8081( I am using 8080 for another project). Once the server is done , he will provide a response that can be used by the client side afterwards( something to be printed on the UI etc.). For the server, the application is developed using Spring Boot . The logic is mainly implemented in the Query and the Mutation classes, specific to GraphQL.

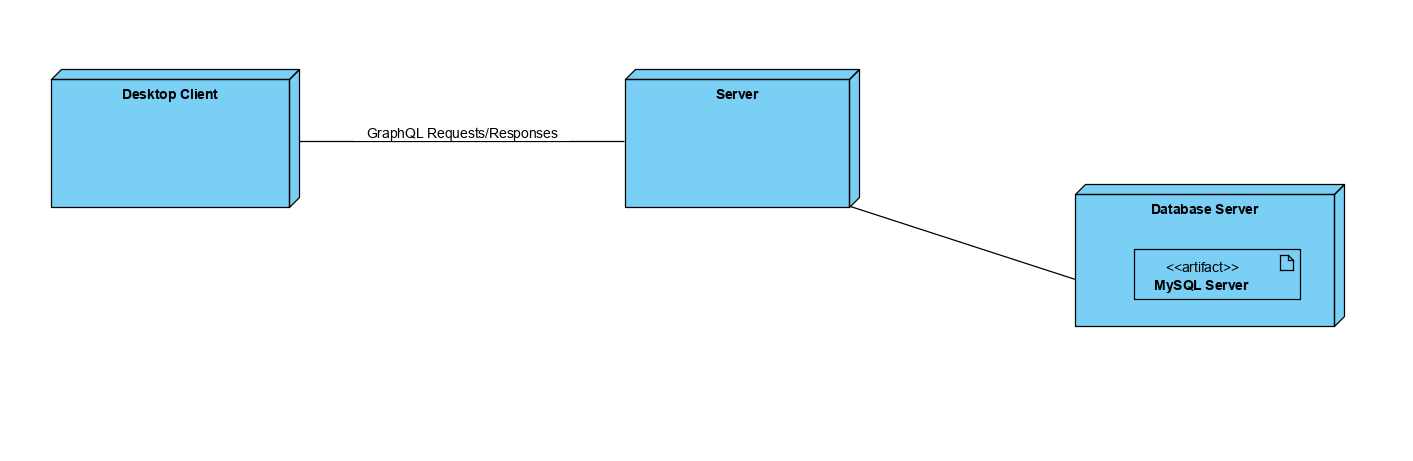
The CQRS architecture ,on another hand , is a pattern which focuses on splitting the requests in order to better emphasize one of the SOLID principles : that of single responsibility. This comes very natural , in my opinion , for an application that has been developed using GraphQL , because the same idea is maintained : queries for retrieving data, commands for operations that modify the database( add, delete , update). I have sketched the way in which this implementation works:



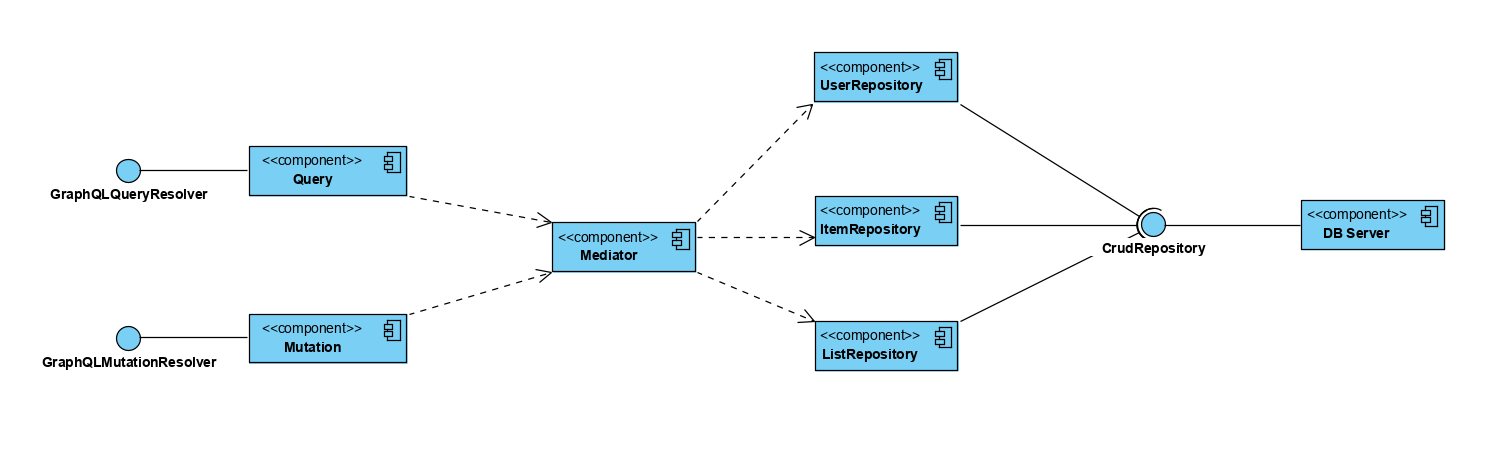
*Package Diagram*

**

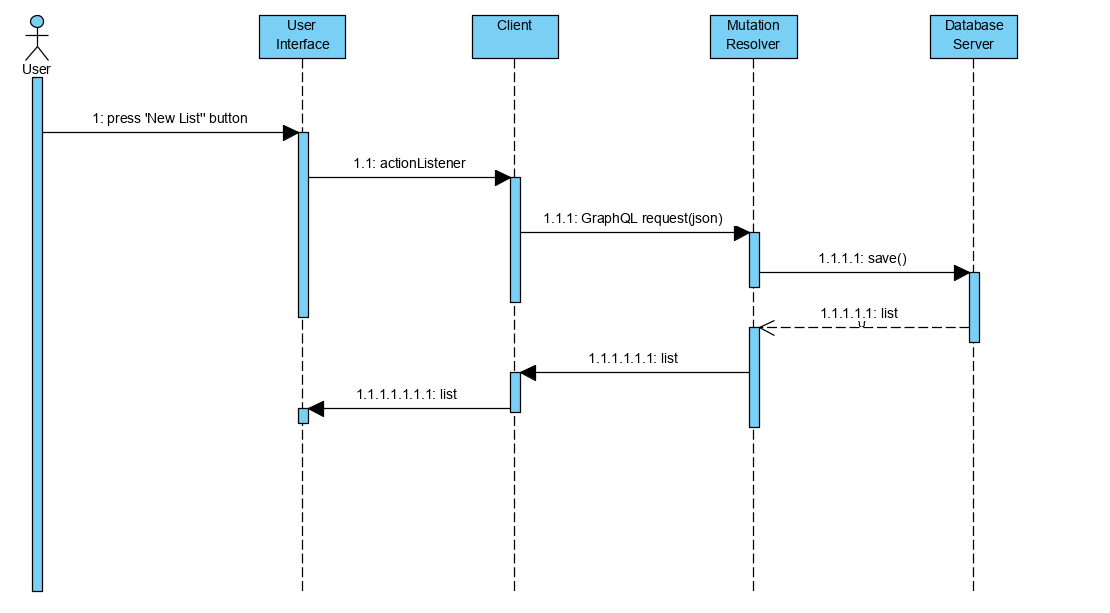
Deployment:



Component:



4. UML Sequence Diagrams



5. Class Design

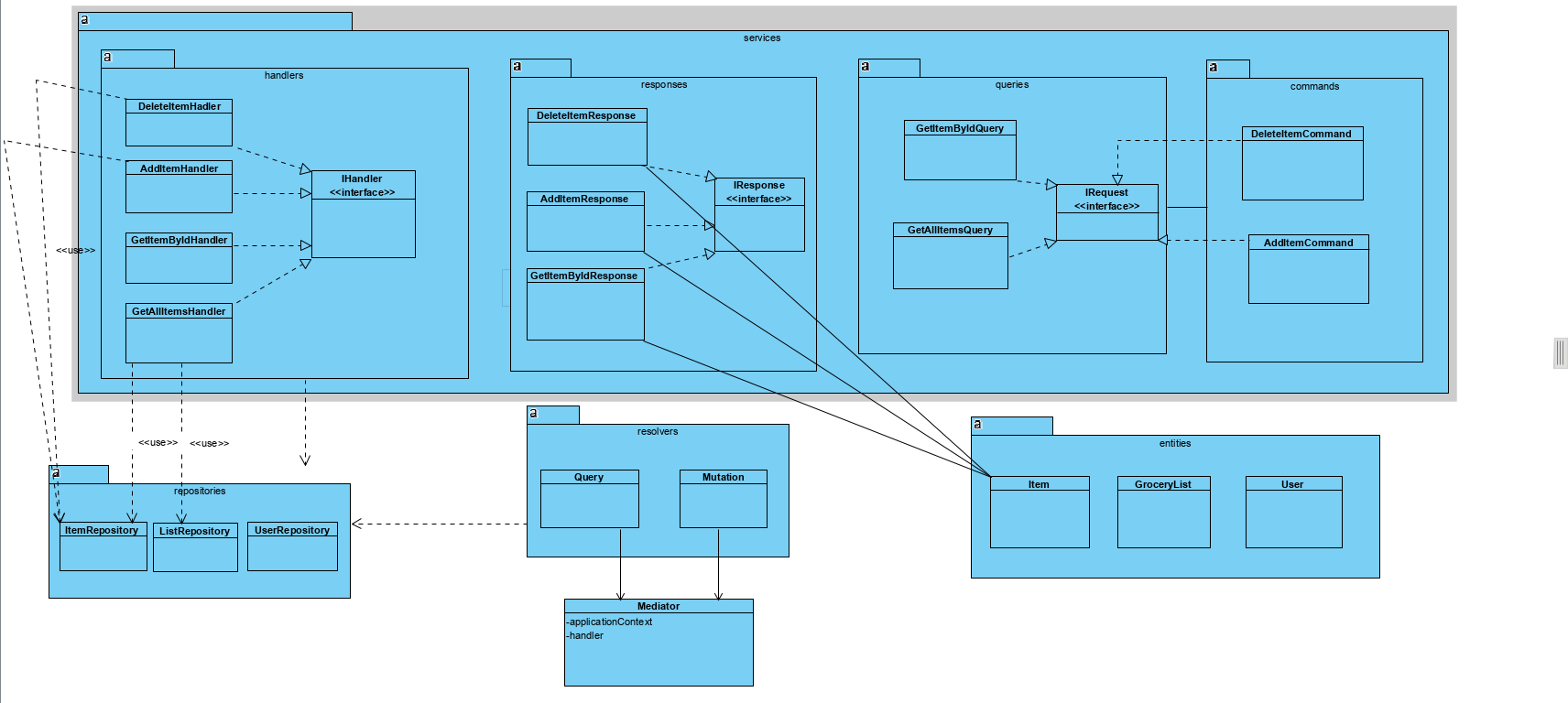
**5.1 Design Patterns Description**

The **decorator pattern** is a design pattern that allows behavior to be added to an individual object, dynamically, without affecting the behavior of other objects from the same class. The decorator pattern can be used to extend (decorate) the functionality of a certain object statically, or in some cases at run-time. This is achieved by designing a new *Decorator* class that wraps the original class.

The mediator pattern is used to reduce communication complexity between multiple objects or classes. This pattern provides a mediator class which normally handles all the communications between different classes and supports easy maintenance of the code by loose coupling. It falls under the behavioral patterns category.

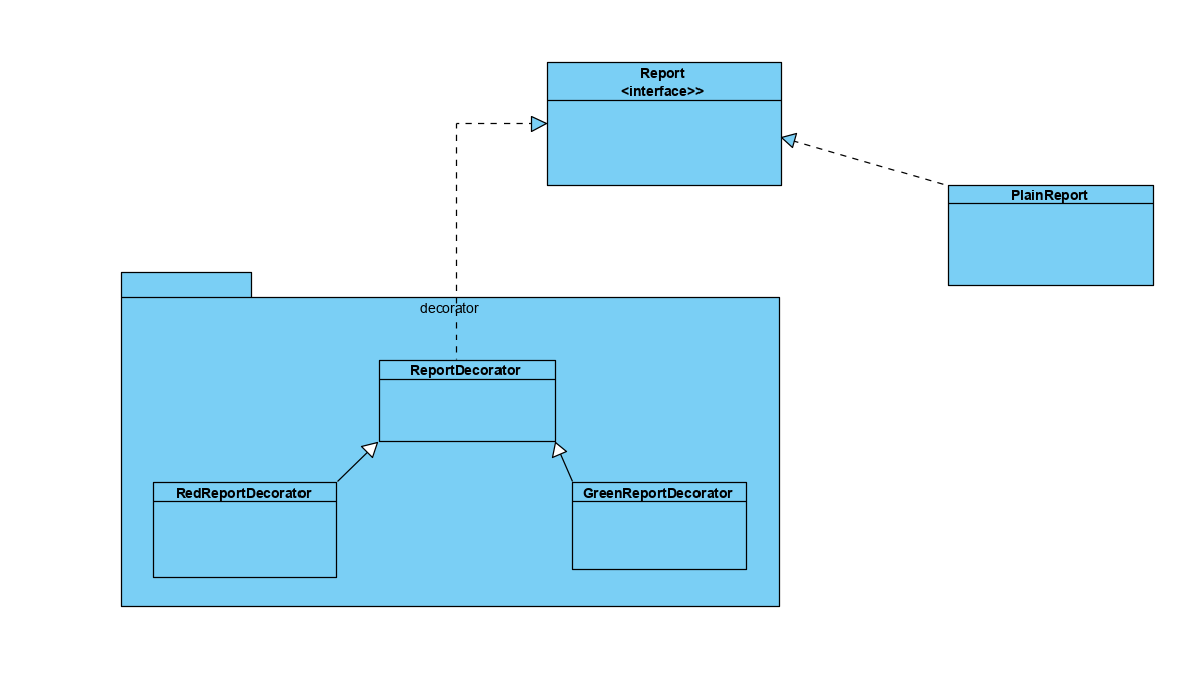
**5.2 UML Class Diagram**

For the server side I have omitted some classes from the implementation because the diagram would have been too large and unreadable . But still , it can be seen that the CQRS architecture have been implemented , as each request that the client is making and that appears on the server is dealt with in the required manner. More specifically , the “concerns have been separated” , so to say, as the handlers , responses and queries each fulfill a specific task . The Meidator pattern comes into place here . It is basically the way in which we are building the connection between the query and the response , via a handler. The handler will be the one which contains the actual ‘logic’ , the solving of our issue, while the queries and responses act more like terminals( query as input and response as output). The mediator actually consists of a hash map which matches a given query to a specific response. Since I have used GraphQL for building the server, the Mediator is primarily used by the Query and Mutation classes.



The implementation of the client side is very similar to the one from the second assignment , so I thought there is no need to redo the whole diagram . Instead , I have put emphasis on the implementation of the Decorator design pattern. It is used for the newly generated reports , as there will be two types ( green report and red report) . The type of the report is established in the BurndownRateCalculator class .

When it comes to the pattern implementation , the ‘center’ can be considered the Report interface . This interface will be implemented by the ReportDecorator, an abstract class defining the main methods which have to be implemented in the ‘concrete’ decorators(RedReportDecorator, GreenReportDecorator). And the final needed element is the PlainReport class. Basically our report starts from the PlainReport , which may contain the ‘start point’ of the report, how it is in the beginning. After that , the decorators are the ones which add a specific attribute to the plain entity( in our case, a color).

****

6. Data Model

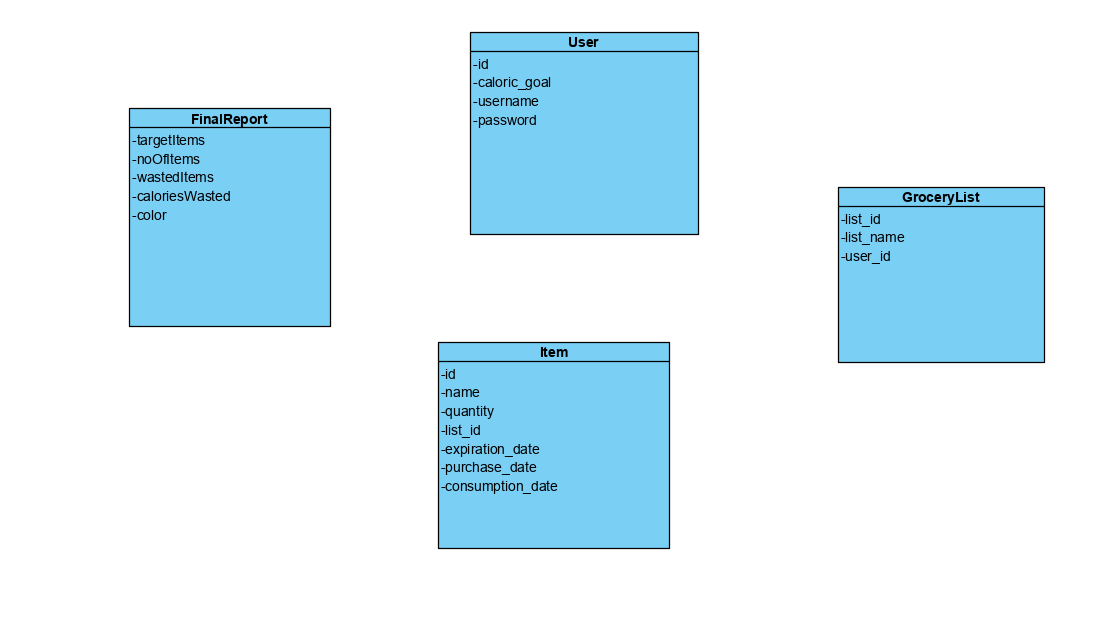
User: contains an id , calories goal , an username and a password.

Item: id, name , quantity, dates(expiration , purchase , consumption) represented as Strings , list id .

GroceryList: list\_id, list\_name , user\_id .

FinalReport : targetItems , noOfItems , wastedItems , caloriesWasted , color.

Even though the dates are represented as strings here , they are still checked on the validators. I implemented this way because it was easier to parse the response received from the server .



7. System Testing

Each component has been tested individually , immediately after being developed. Once the components were working as expected, they were put together.

Because I have worked with GraphQL , I have been able to build the server and run queries directly on the graphiql interface . So the server has been built priorly and tested with priority to ensure that the main functionalities( database connection , mutations) are met. After adding the client and providing requests-responses, the data-flow testing has been used.

8. Bibliography

<https://refactoring.guru/design-patterns/decorator>

<https://www.youtube.com/watch?v=j40kRwSm4VE>

<https://docs.microsoft.com/en-us/azure/architecture/patterns/cqrs>

samples from laboratory drive