Wasteless 3

Analysis and Design Document

Student: Aldea Roxana Ioana

**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

Wasteless is a client-server architecture based application designed in order to reduce food waste by allowing users to track their groceries and set goals based on the ideal burndown rate. First of all, the system is provided with a login system in order to allow only registered users to enter the application. In order to authenticate, the user has to introduce its credentials, meaning username and password. The user is able to create a grocery list by providing a name or to select an existing list and choose to edit it. Edit list means add items or donate them. A grocery list item has a name and a quantity as well as a calorie value, purchase date, expiration date and consumption date. The system should provides the user with options to donate excess food to various local food charities and soup kitchens.

One important feature of this application is that each time the user logs in, he/she is notified about the items which expire soon, this way, those items can be consumed, reducing food waste.

Also, it the user makes waste, the colour of the report changes to red, otherwise, it is green.

# Functional Requirements

A user can authenticate by providing the right credentials (username and password) and after that it can:

* View, create, edit a new grocery list
* View, create, donate grocery items
* Set a goal to minimize food waste
* Generate reports

# Non-functional Requirements

The data should be stored in a database

The system architecture should be a layered one

In order to generate two types of reports, the abstract factory design pattern should be used

The data access should be implemented using ORM

The dependencies should be resolved using a DI containter

2. Use-Case Model

Use case: Add Item

Level: user-goal level

Primary actor: user

Main success scenario: The user is allowed to add new grocery items to a newly created list or to an existing list. It has to introduce some features as (name, quantity, calorie value, purchase date, expiration date and consumption date). If the user provides valid data, for example positive values for quantity and calories and valid consumption date, not before the purchase date or after expiration date, the new item will be added in the database.

Extensions: In case the user introduces invalid data, as I have said above, bad values for quantity, calories or consumption date, an error message will appear on the top of page.

*O imagine care conține desen

Descriere generată automat*

3. System Architectural Design

**3.1 Architectural Pattern Description**

One of the architectural patterns used in this application is the Client-Server Architecture. This architectural pattern was chosen because it has important advantages over the simple Layered Architecture. The Client Side is decoupled from the Server Side. The Client, the front end has been developed in Angular using Typescript and the the back end, in Java & Spring Boot. The backend has a 3-tier layered architecture, split in dao, business and presentation layers, which contain the specific Spring classes, Repository in dao, Service in business and Controller in presentation. The application uses Tomcat as server.

Along with the client-server architectural pattern, CQRS was used in order to separate the command requests: create, update, delete from the query commands: retrieve. This was achieved by splitting the service in two components, two packages: one which handles the command requests and another one which handles the queries. This fact can be observed by looking at the package diagram.

**3.2 Diagrams**

**Component Diagram**

**O imagine care conține captură de ecran

Descriere generată automat**

**Client-Server Architectural Pattern**

**O imagine care conține captură de ecran

Descriere generată automat**

**CQRS Architectural Pattern**

**O imagine care conține captură de ecran

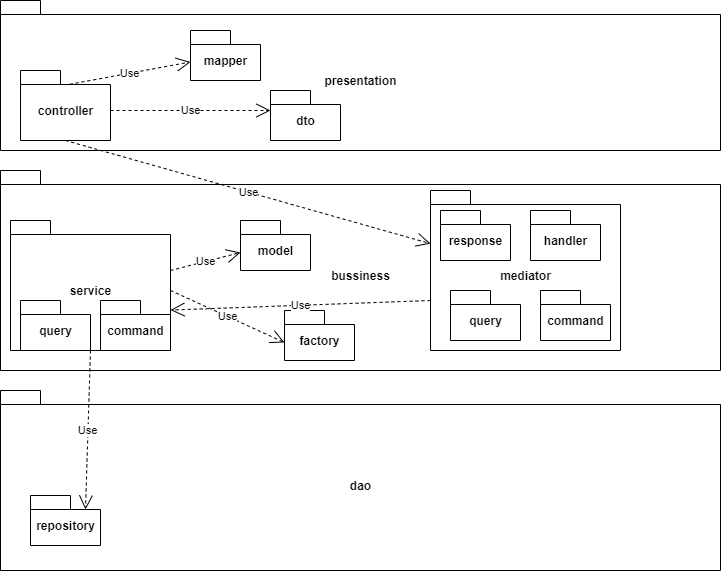
Descriere generată automat**

**Deployment diagram**

O imagine care conține captură de ecran

Descriere generată automat

**Package diagram**

****

**Sequence Diagram for insert item use-case**

O imagine care conține captură de ecran

Descriere generată automat

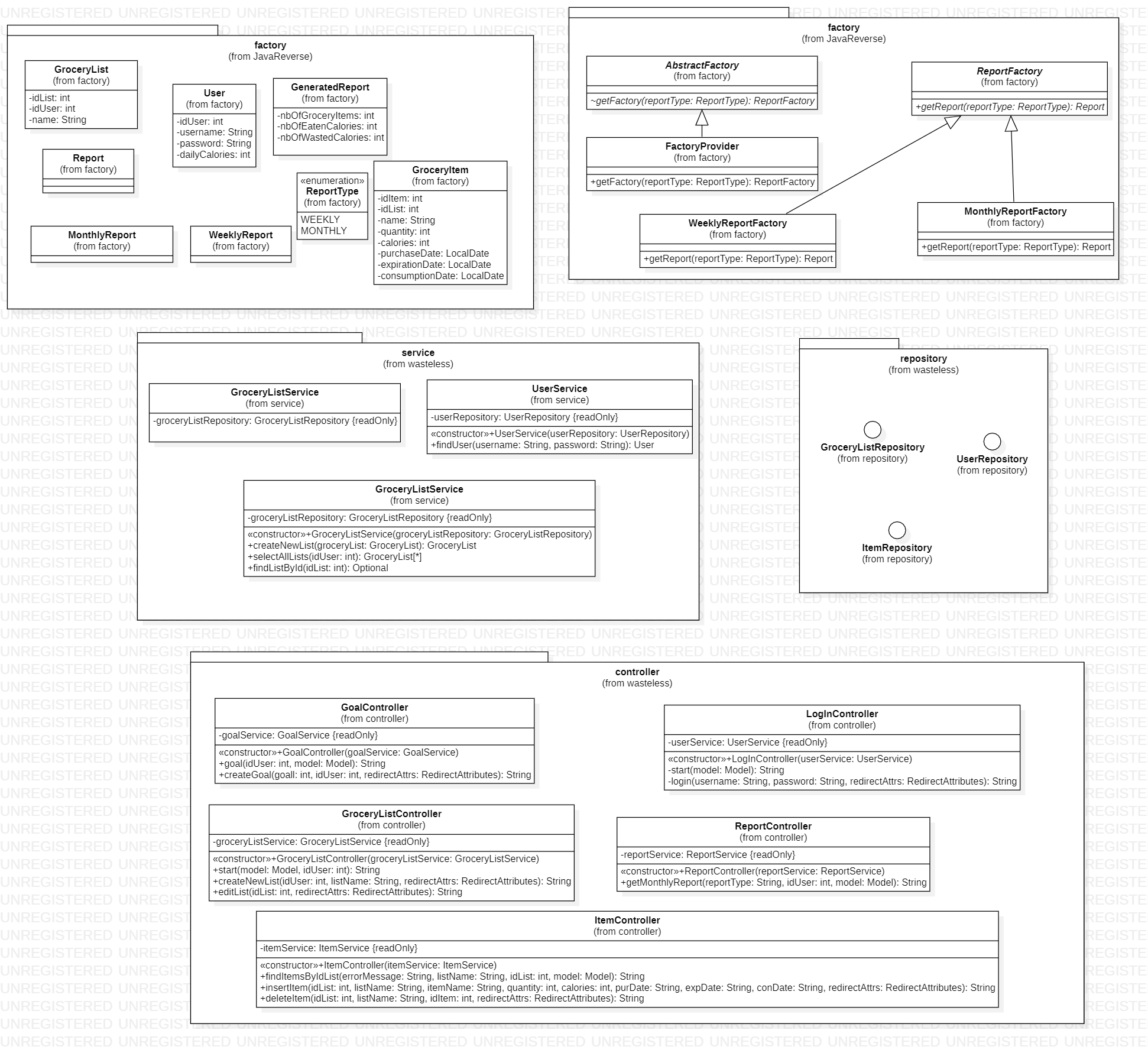
5. Class Design

**5.1 Design Patterns Description**

Along with the design patterns described in the previous versions of this application, Observer, for managing the notification system which tells the user about the products which expires soon, Abstract Factory for creating different types of Reports, Weekly reports and Monthly reports, the application uses the Decorator Design Pattern. The functionality brought by using this DP is that the colour of the report is changed. If the user has grocery items unconsumed and expired, wasted, the colour of the report will be red. Otherwise, it will be green. The classes used in order to implement this DP are located in the report/decorator package on the client side.

The Mediator Pattern was also used in this app. As it was mentioned above, the application uses the CQRS architectural pattern. With the mediator pattern, communication between objects is encapsulated within a mediator object. Objects no longer communicate directly with each other, but instead communicate through the mediator. This reduces the dependencies between communicating objects, thereby reducing coupling. It was implemented in the business layer, but it is used in the presentation layer, handling the requests and responses. It can be observed in the package diagram. The mediator package contains the command, query packages for requests, a response package, a handler package and some interfaces in order to make use of Polymorphism.

**5.2 UML Class Diagram**



6. Data Model

O imagine care conține captură de ecran

Descriere generată automat

7. System Testing

The application was tested using Junit framework. For the components which use other components, I have been used Mockito to mock the functionality of that specific component. There are just some tests, for GroceryListController. The Mediator class uses ApplicationContext in order to create a handler bean at run time, this way I have annotated the ApplicationContext field with @Mock annotation.

The Service component has not been tested because uses Repository interface which extends Jpa Repository and as it is known, using Spring Boot eases very much the writing of queries and they are done by default using the model classes, mapped through ORM to the database entities.

8. Bibliography

* <https://stackoverflow.com/>
* <https://www.callicoder.com/spring-boot-rest-api-tutorial-with-mysql-jpa-hibernate/>
* <https://spring.io/docs>
* <https://angular.io/guide>