Student: Socaci Radu Andrei

**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

Design and implement an application that helps users manage food waste.

Once a user is authenticated, he can input grocery lists and see reports of how much food is wasted weekly and monthly. The system also allows users to track goals and minimize waste by sending reminders if waste levels are too high based on ideal burndown rates.

The ideal burndown rate for 100 calories worth of groceries due to expire in 5 days is 20 calories worth of groceries per day.

The system should provide you with options to donate excess food to various local food charities and soup kitchens and notify you of them prior to item expiration.

# Functional Requirements

The application should have authentication (http basic in this case). In order to access the application, the user needs to provide valid credentials (username/password).

Once authenticated, the user can perform the following operations:

* Perform CRUD on grocery lists (view, create, delete, update)
* Perform CRUD on specific items inside a grocery list
* Generate reports (monthly and weekly reports)
* Donate excess food to charitable organizations
* Set a goal and get notified whenever the number of calories is exceeded.

# Non-functional Requirements

The app did not impose any development restrictions, from a framework/technology point of view. However, a few non-functional requirements still aroused:

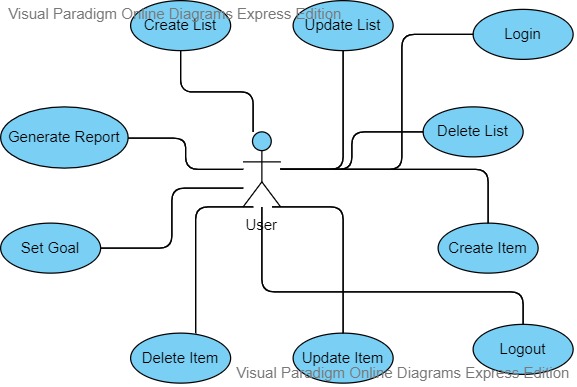
* The data should be persisted in a database
* The application should use a Client-Server architecture
* The data access level should be implemented using an ORM
* Perform input validation before saving the data into the database
* Use a CQRS architecture, use a mediator pattern to handle requests
* Use a decorator pattern for changing the color of the report (green for above the ideal

rate and red for under)

Moreover, the app has other typical non-functional requirements, such as: availability, performance, ease of use, etc.

The application uses Spring Security coupled with an interceptor for http basic authentication. For simplicity there is no CSRF protection and the token itself is kept in sessionStorage (which means that SSO will not be possible and the user will have to login each time the browser is closed). This approach is not particularly secure for production (XSS attacks are possible) but is quite simple to implement.

2. Use-Case Model



Use case: Login

Level: User-goal level

Primary actor: The user

Main success scenario:

1. The user provides a username/password combination when prompted to log in
2. If the credentials are not valid, the user will be notified, and the flow will stop
3. If the credentials are valid, the user will be redirected to the home page

Extensions: In case of failure, the user will be able to log in again, with different credentials

3. System Architectural Design

**3.1 Architectural Pattern Description**

The application uses a 3-tier Layered RESTful architecture server-side, coupled with an Angular 9 centered client (presentation). While the server cannot be said to have an actual presentation layer, controllers are used to communicate with the client (by means of a DispatcherServlet). However, Spring Framework was used, and taking into considerations Spring best practices, I decided to further split the application, adding a Service layer, splitting the data access layer into Repository and Model packages. Also, part of the business logic layer is the Factory package, which hold all the logic related to creating reports.

The application uses a client-server architecture. The presentation part is handled by an Angular 9 client. The server sends JSON responses to the client, which the later one renders on the screen (browser). As a consequence, this version of the application, as opposed to the one presented during the first assignment, is faster. The html will not be statically requested each time but will be dynamically changed depending on user events (SPA) using javascript.

Other design patterns used were embedded into the data access layer, since they closely relate to the domain model classes (Builder and Observer). There is a more classical observer on the client-side as well, which listens to the websocket connection for any incoming notifications. The server-side observer uses Spring Events and is therefore not suited for the requirements of the assignment.

The server-side code was adapted to better emphasize SRP (Single Responsibility Principle) by means of the CQRS architecture. The mutators (commands) and queries have been separated and the communication between the application entry point (controller) and the actual handler which handles the specific request is facilitated by a custom-built mediator. The mediator simply maps a request to its handler, whereas the latter one maps the request to its response and returns it to the application controller for further processing (client-side).

**3.2 Diagrams**

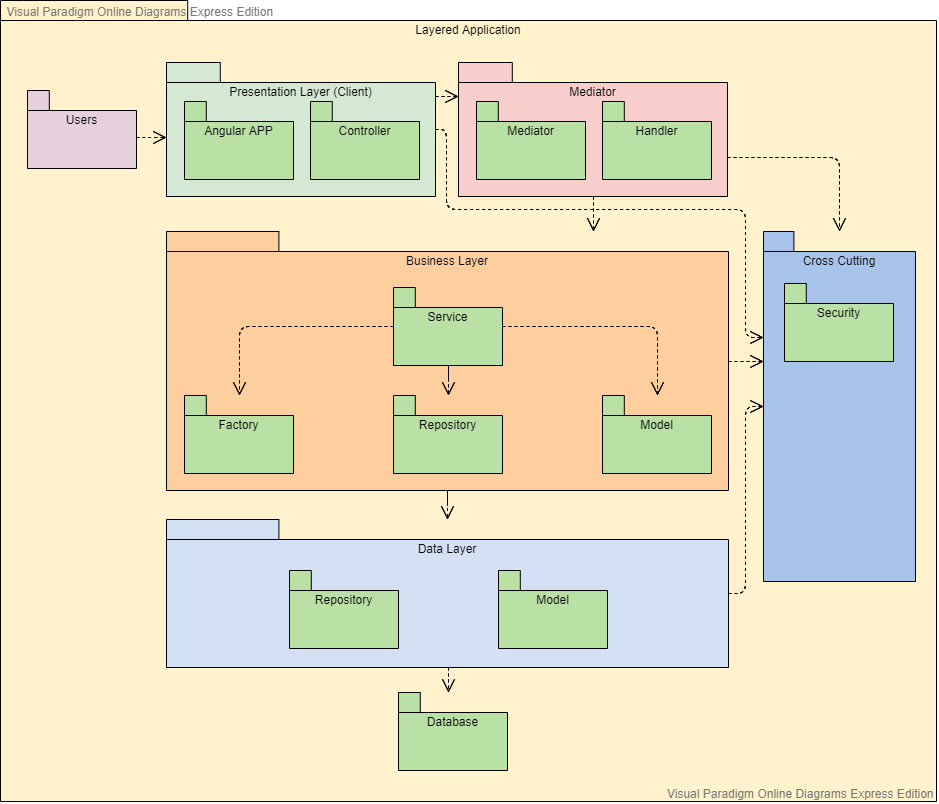
The presentation layer will contain a package for the controllers of the application and a package for other useful classes for the views. The presentation layer depends on the DTO (Data Transfer Object) classes and on the business layer.

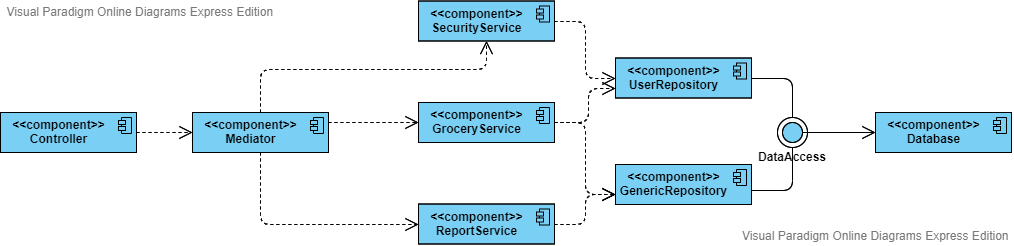
The business layer package contains packages for the services, where logic and validations of data are implemented, and another package where the factory design pattern is implemented for generating Reports. The business package also depends on the DTO package, as well as on the model package, which maps a POJO to every table in the database.

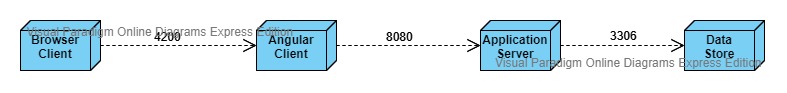
The data access layer contains the repositories used for obtaining data from the database and performing CRUD operations on them. It depends as well on the model package, with its respective classes, since these directly map to tables in the database.

The client application directly communicates with the server by means of http requests, which are handled by spring’s server (tomcat) and by means of a DispatcherServlet and a RequestMapper, mapped to a specific method in a controller. The controller returns all the necessary data, which the client can then render for the user to see.

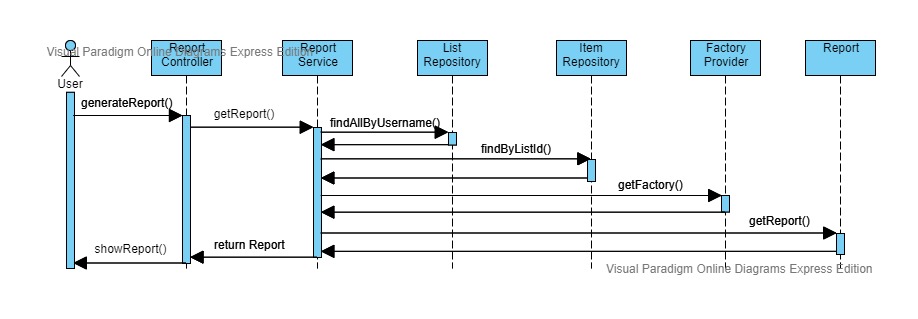
Furthermore, the service layer handling the business logic was redesigned to use CQRS architecture, for a more concise service implementation, resulting in a more SRP compliant application and loose coupling. Routing the request to the CQRS handlers is done by the mediator, represented in its own package, connecting the presentation and the business layers.







4. UML Sequence Diagrams



5. Class Design

**5.1 Design Patterns Description**

First, the Abstract Factory design pattern was used for report generation. There is a factory provider, which instantiates a factory, based on the request parameter given. After the correct factory was instantiated, the factory returns an instance of type Report, which can either be a MonthlyReport or a WeeklyReport. These in term extend from an abstract class, Report, which was used to encapsulate the common functionality, to avoid code duplication. A ReportDTO is used for communication with the view layer.

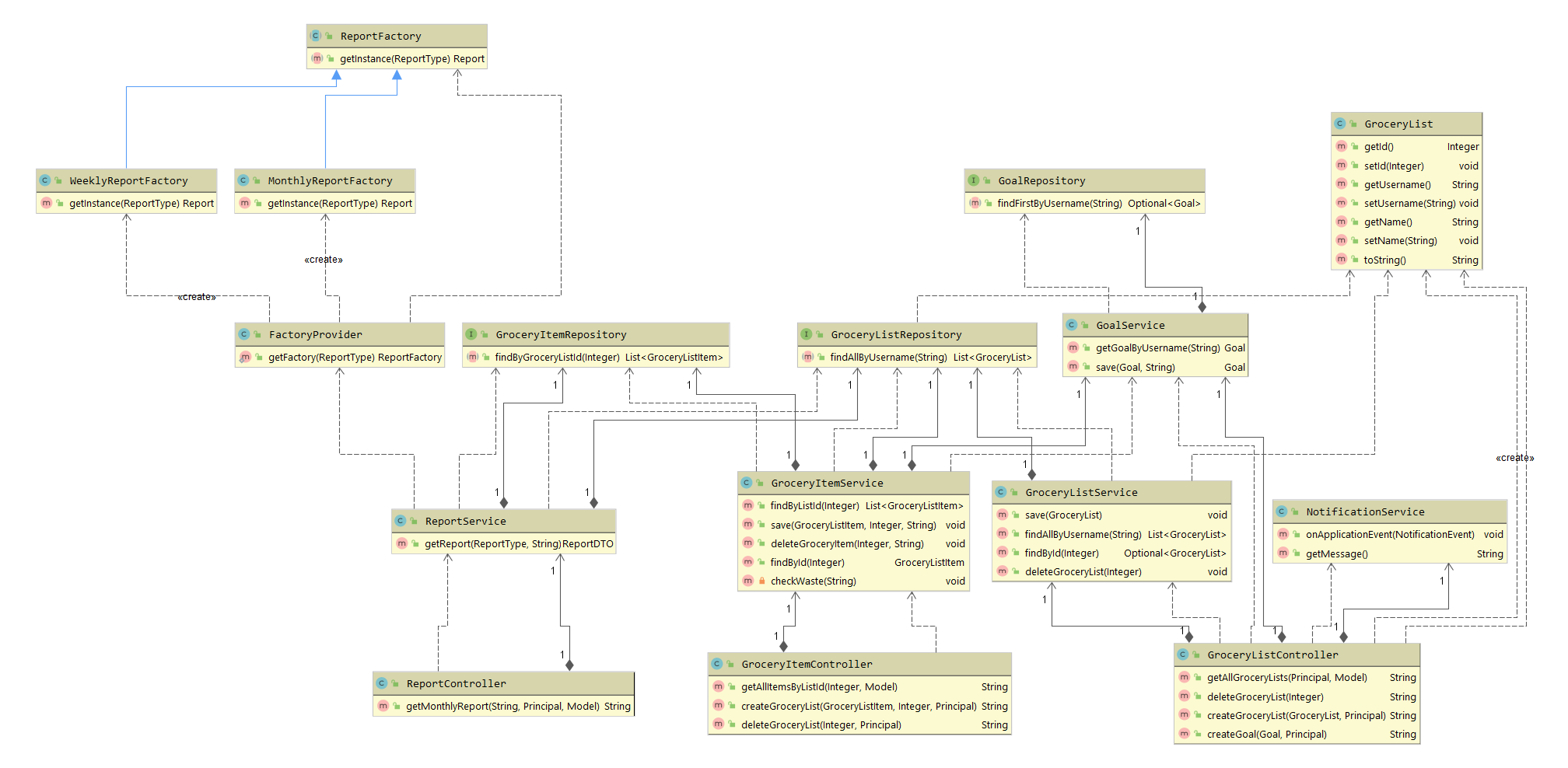
Second, the observer design pattern was used for reminders (called notification in the app). Spring does not allow for a classic implementation of the Observer design pattern. It is possible, by predefining the beans (subscriber) and subscribing to the observable. However, this approach is both error prone and not recommended. A better implementation can be achieved by use of Spring Events, which were used in this case.

In order for the assignment to be compliant with the requirements, another Observer (a classic one this time) was implemented on the client-side. The observer is not needed in this case, as the websocket architecture, coupled with sockJS allows for streaming data to be both received and sent. However, it was implemented just to present the bare concept of this pattern.

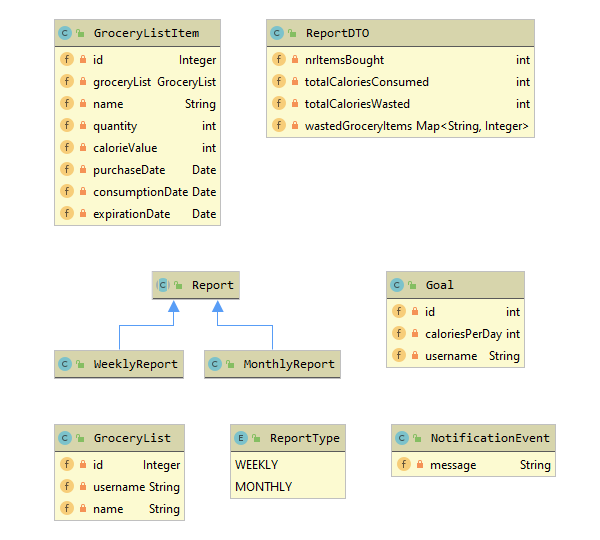
To complement the report generation functionality of the application, the Builder design pattern was also implemented to facilitate the creation of ReportDTOs’.

The “waste/no waste” notification is appears red when there is a waste and green when there is now waste, and the result is logged using a decorator on the client side (custom annotation).

**5.2 UML Class Diagram**



6. Data Model



7. System Testing

Testing was mainly conducted on service classes. The Unit Testing was implemented using tools like JUnit and Mockito. JUnit was used for implementing the unit tests and Mockito for creating mocks of the repositories.

Some of the test cases include:

* Save goal test: The goal repository is mocked. Any goal class given to it will return a predefined mock. We assert that the value of the username of the actual instance that was saved in the database is the same as the one returned by the service.
* Get goal by username: Given a username, the repository returns an optional of goal. We assert that the returned instance’s username is the same as the one given to the service in the initial query.

8. Bibliography

* <https://spring.io/docs>
* <https://docs.spring.io/spring-security/site/docs/current/reference/html5/>
* <https://www.baeldung.com/>