Wasteless

Student: Oancea Eduard

**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. 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 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.

Notifications/reminders will be done in the second assignment.

# Functional Requirements

The app should allow the user to:

* sign-up
* sign-in
* perform CRUD operations on groceries
* donate food
* notify the user in the case of waste
* display reports on his waste

# Non-functional Requirements

- performance

- usability: the interface is easy to use and provides validation messages

- security: data can only be accessed by authenticated users

- platform compatibility: can be deployed on mac/linux/win and runs in any modern browser

- availability: the data is persisted in a database

- use an ORM and a DI container

- use an OOP language: Typescript (though most types miss on the backend as I have some configuration issues which I intend to fix in the next assignment)

- use a layered architecture

- use the abstract factory pattern for weekly/monthly reports

- the data is validated

2. Use-Case Model

Use case: Sign-up

Level: user-goal level

Primary actor: Anonymous user

Main success scenario: The user inputs a new email address and the password twice -> the new account is created, and the user is redirected to the sign-in page.

Extensions: The user inputs an email address and 2 different passwords, validations fail -> the user is alerted of the validation error. The user inputs an already used email address and 2 different passwords, the API returns the validation error and the user is alerted of the validation error.

A close up of text on a white background

Description automatically generated

3. System Architectural Design

**3.1 Architectural Pattern Description**

The application is a Web Application modeled using the Client-Server architecture. The Client-Server architectural pattern splits the functionality into two components: a client and a server. The client side is responsible with rendering a user interface and allowing the user to interact with the application. It communicates with the server by sending http requests and updating its’ internal state on receiving the responses. The server is concerned with processing, validating and then storing the data in a relational database (for this particular application). The Client-Server architecture is a layered architecture and has 2 tiers. The 2 tiers are the client and the server themselves, while we have many other layers that construct them. The communication is done using Representational State Transfer (REST).

The client is built on React and utilizes the flux architectural pattern with the context API.

The server is built on Express with Sequelize as the ORM. It has the following layers:

* Controller layer: it maps the resources from the request to the service layer (could be further split in a routing layer as well)
* Service layer: it contains the business logic and does the ORM operations (that should ordinarily be done by the repository layer)

The database used is PostgreSQL as it is open source and has great support.

**3.2 Diagrams**

A close up of a piece of paper

Description automatically generated

4. UML Sequence Diagrams

The numbers here have no meaning but I didn’t find a way to remove them in starUML.

*A screenshot of a cell phone

Description automatically generated*

5. Class Design

**5.1 Design Patterns Description**

The design patterns I used are:

* Dependency injection, via bottlejs, a dependency injection container for node.js applications and using react hooks, as well as via module imports. For the DI container, it creates and manages all of the dependencies of the backend application
* Abstract factory for generating reports. For this, I created a factory interface that is implemented by the 2 concrete factory classes. I also created an interface for the reports themselves, 1 generic report class and 2 classes that extend this class. While it is not “textbook” abstract factory, there would be a lot of duplicated code if I didn’t make the Report class.
* ORM for accessing the database. I have also created migrations files to create the database and have checkpoints similar to git commits.
* The factory pattern for creating certain objects

**5.2 UML Class Diagram**

**A close up of a map

Description automatically generated**

I did not include the class diagram for the client as I will do that for the next assignment.

6. Data Model

*A close up of a white wall

Description automatically generated*

I considered having 1 list per user for now. It can be extended in the next assignments.

7. System Testing

During the development process, I validated my work with postman to check the API endpoints, while using console logs to debug. For the client I used console logs and did manual testing. I have set-up a testing environment on the backend that recreates the test database after each test is run. Only 1 test is written for accessing an endpoint.

8. Bibliography