

Wasteless Application Analysis and Design Document

**Student: Carla-Maria Rusu
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	11
5. Class Design	11
6. Data Model	14
7. System Testing	16
8. Bibliography	18

1. Requirements Analysis

1.1 Assignment Specification

This application helps users manage food waste. Users can log in, add grocery items, view weekly and monthly reports, and track their goals. The application offers the possibility of donating food in case of food wastage. For each unconsumed item, a burndown rate is computed such that the user may know how much to consume before the item expires.

1.2 Functional Requirements

- The user has the possibility to register
- The user can log in if they already have an account
- The user can view their groceries list
- The user can set a goal
- The user can add a grocery item
- The user can add a consumption date to an item
- The user can view monthly and weekly reports
- The user is provided burndown rates for each grocery item
- The user can log out

1.3 Non-functional Requirements

- The burndown rate is computed by the system
- The user should have their own groceries and should not be able to view other users' accounts
- The user should not be able to provide wrong inputs to the login page, the add groceries page, the add consumption date form or set a negative goal (the inputs should be reasonable)

2. Use-Case Model

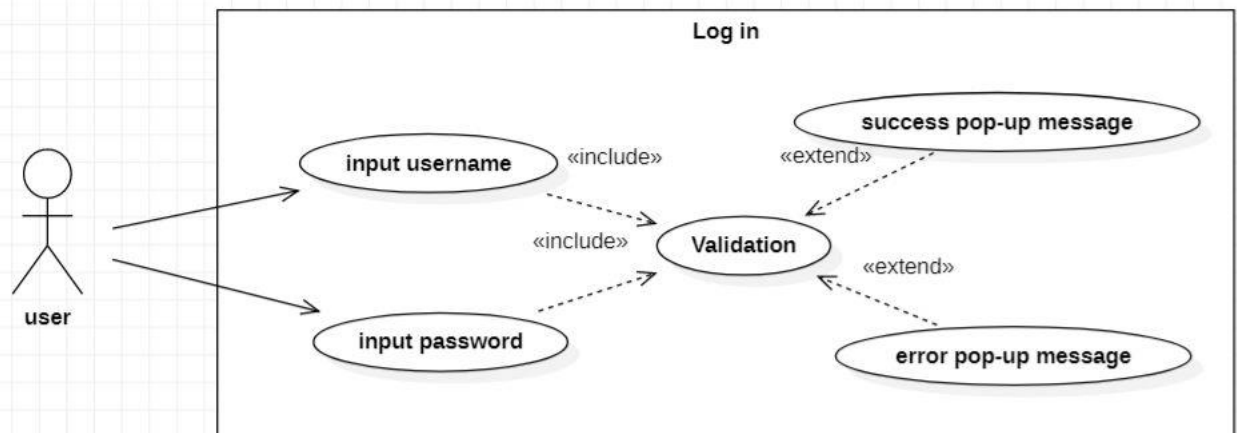
Use case: Log in

Level: user-goal level

Primary actor: user

Main success scenario: the user introduces an existing account username and password and logs in. A success pop-up message appears

Extensions: the user introduces incorrectly formatted data or data that is not in the database. An error pop-up message appears to inform the user of the status



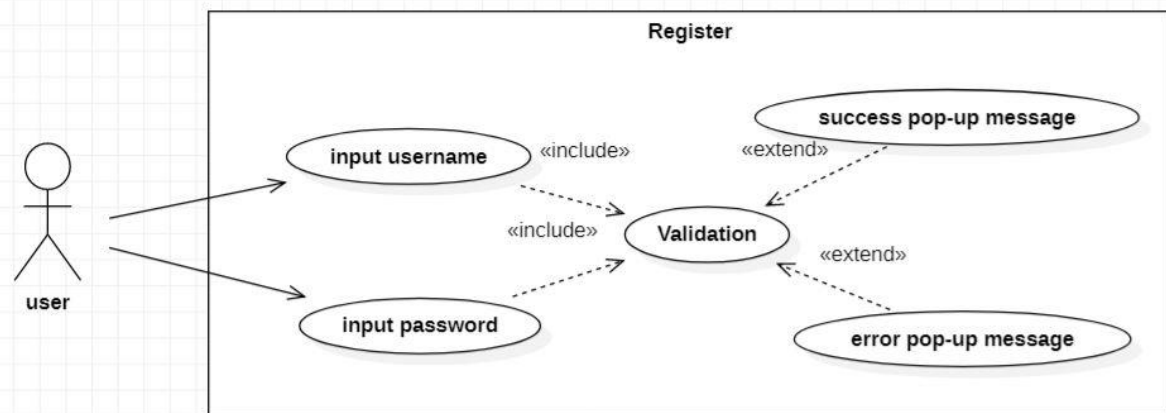
Use case: Register

Level: user-goal level

Primary actor: user

Main success scenario: the user introduces a non-existing username and password and registers. A success pop-up message appears

Extensions: the user introduces incorrectly formatted data or a duplicate username. An error pop-up message appears to inform the user of the status



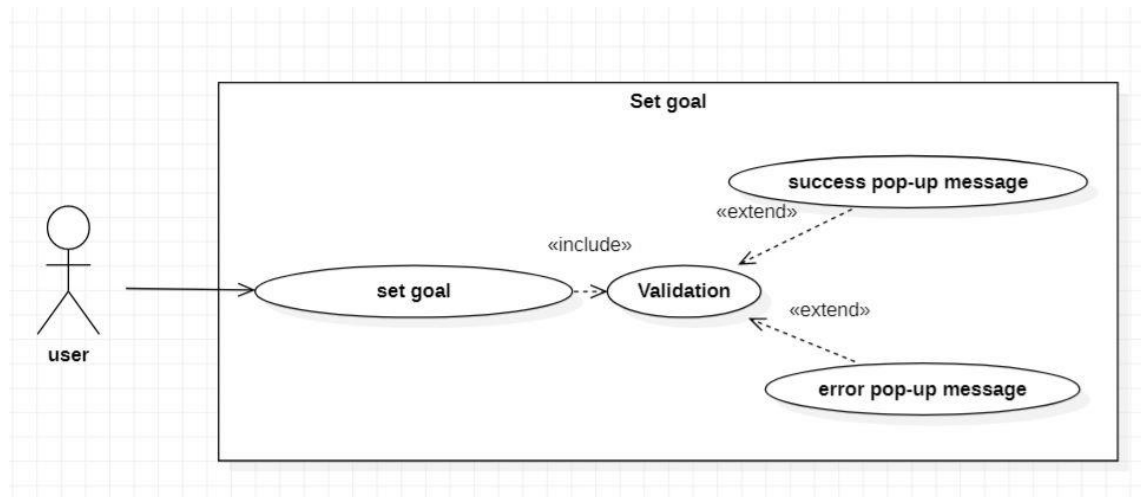
Use case: Set goal

Level: user-goal level

Primary actor: user

Main success scenario: the user introduces strictly positive number. A success pop-up message appears. The goal is updated.

Extensions: the user introduces incorrectly formatted data. An error pop-up message appears to inform the user of the status

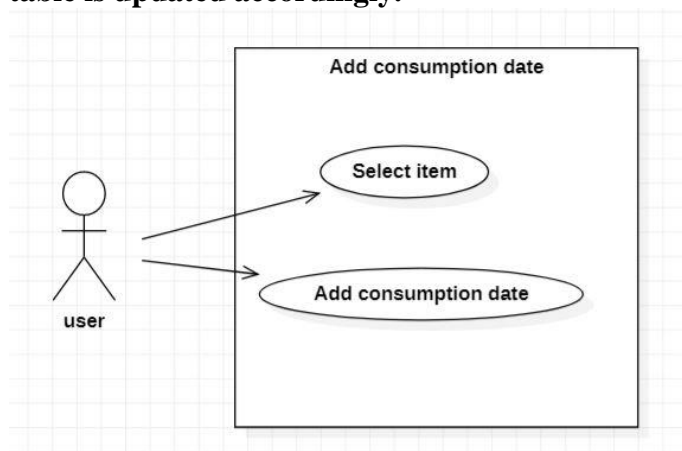


Use case: Add consumption date

Level: user-goal level

Primary actor: user

Main success scenario: the user introduces a consumption date for a selected item. The table is updated accordingly.

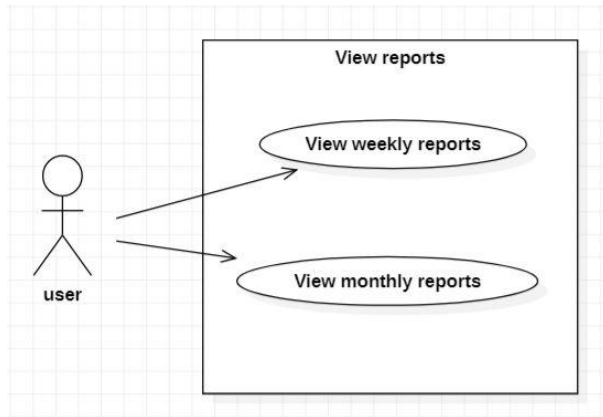


Use case: View reports

Level: summary

Primary actor: user

Main success scenario: the user selects either the weekly or the monthly report buttons. The selected report is displayed below the table



Use case: Set goal

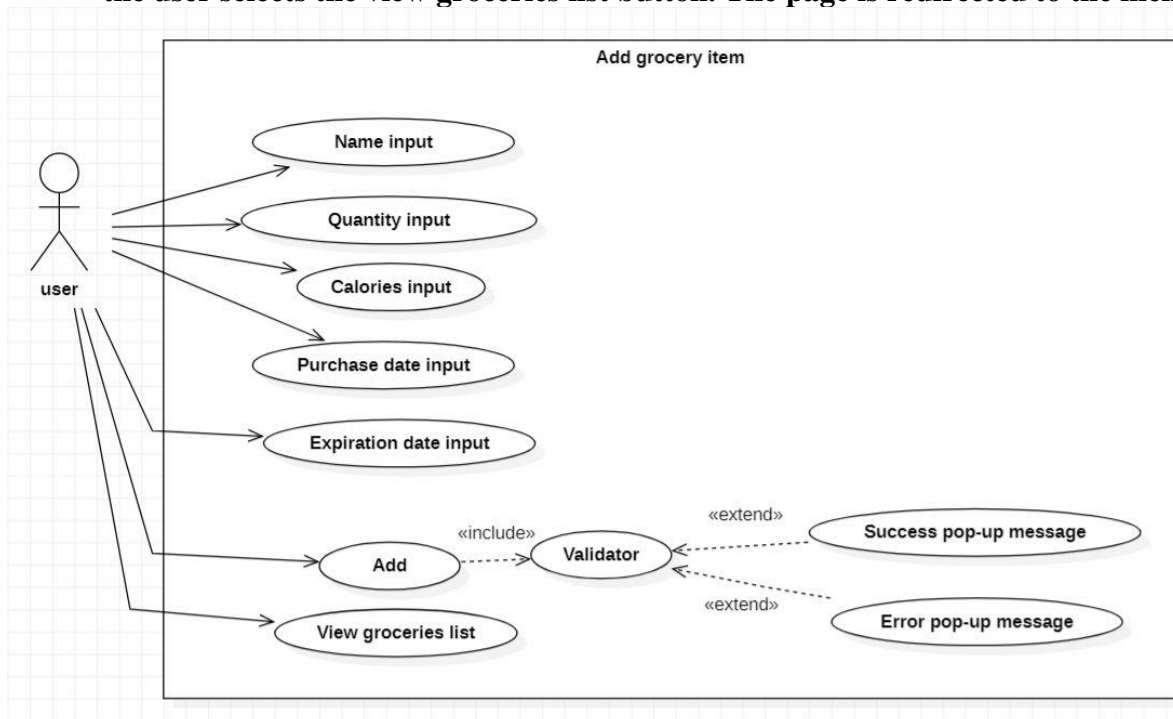
Level: user-goal level

Primary actor: user

Main success scenario: the user introduces the input data correctly and selects the add button. A success pop-up message appears. The grocery list updated.

Extensions:

- the user introduces incorrectly formatted data. An error pop-up message appears to inform the user of the status
- the user selects the view groceries list button. The page is redirected to the menu.



3. System Architectural Design

3.1 Architectural Pattern Description

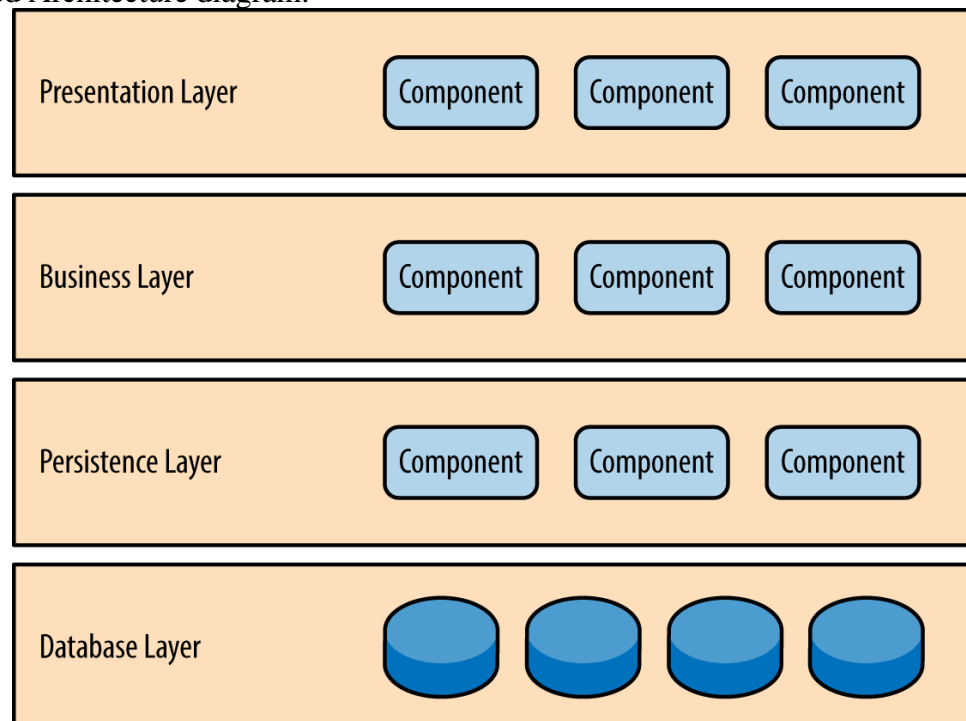
The application is built using the **Client-Server architecture**. The frontend represents the client and the backend hosts the server. The client-server architecture has the advantage of having more **clients connected to a single server**. The frontend was made with Angular and the backend with Spring Boot.

The backend is structured using the **Layered Architecture** pattern. This architecture separates the layers such that packages are loosely coupled: a layer may interact only with the layer beneath it. The layers are: **Presentation layer, Business layer, Persistence layer and Database layer**. The **utilities package** is not included in the layers.

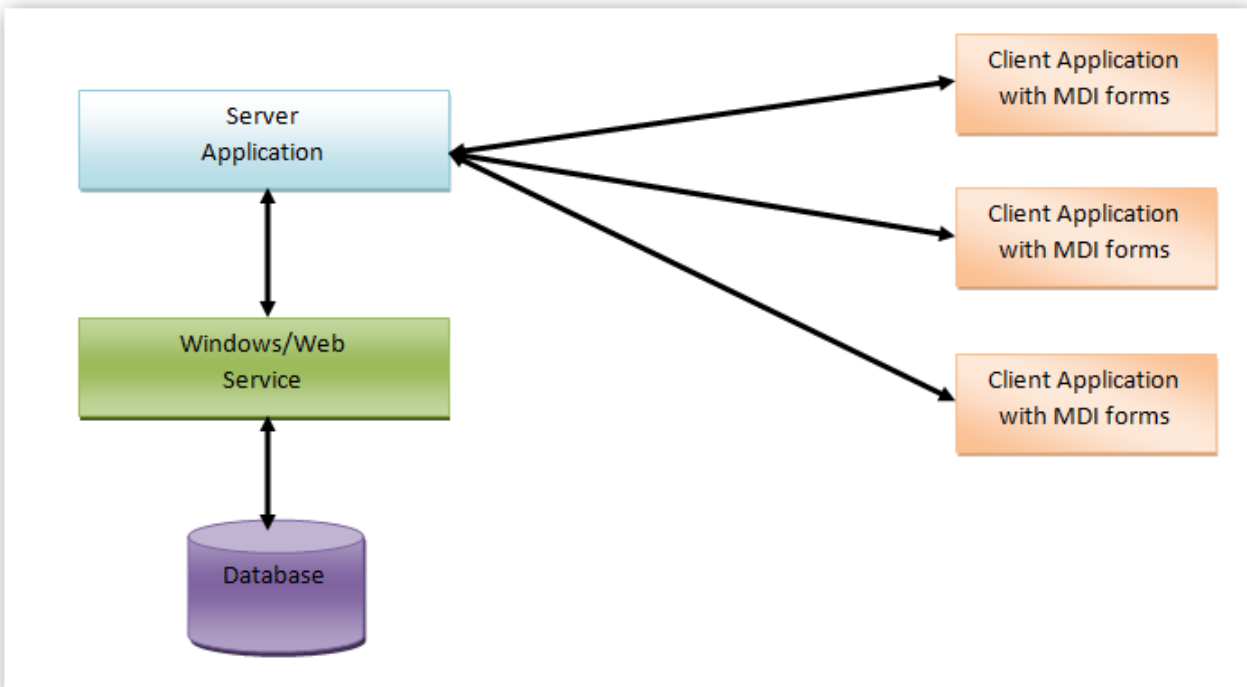
The backend side of the application also employs **CQRS architecture**. CQRS stands for **Command Query Responsibility Segregation**. CQRS is an architectural pattern that **separates command and query requests**. Commands encompass create, update, and delete actions, while queries represent read actions. This pattern adds a layer of complexity, whilst de-coupling the code. The architecture was designed using the **Mediator pattern**. The Mediator pattern promotes code reusability and enables loosely coupled components. The mediator represents the link between **requests and responses**, i.e. a **handler**. The handler applies an action to the request in order to obtain the required response. A mapping is created between each request and its handler. As such, the controller's in the presentation layer communicate through the mediator with the corresponding services in the business layer.

3.2 Diagrams

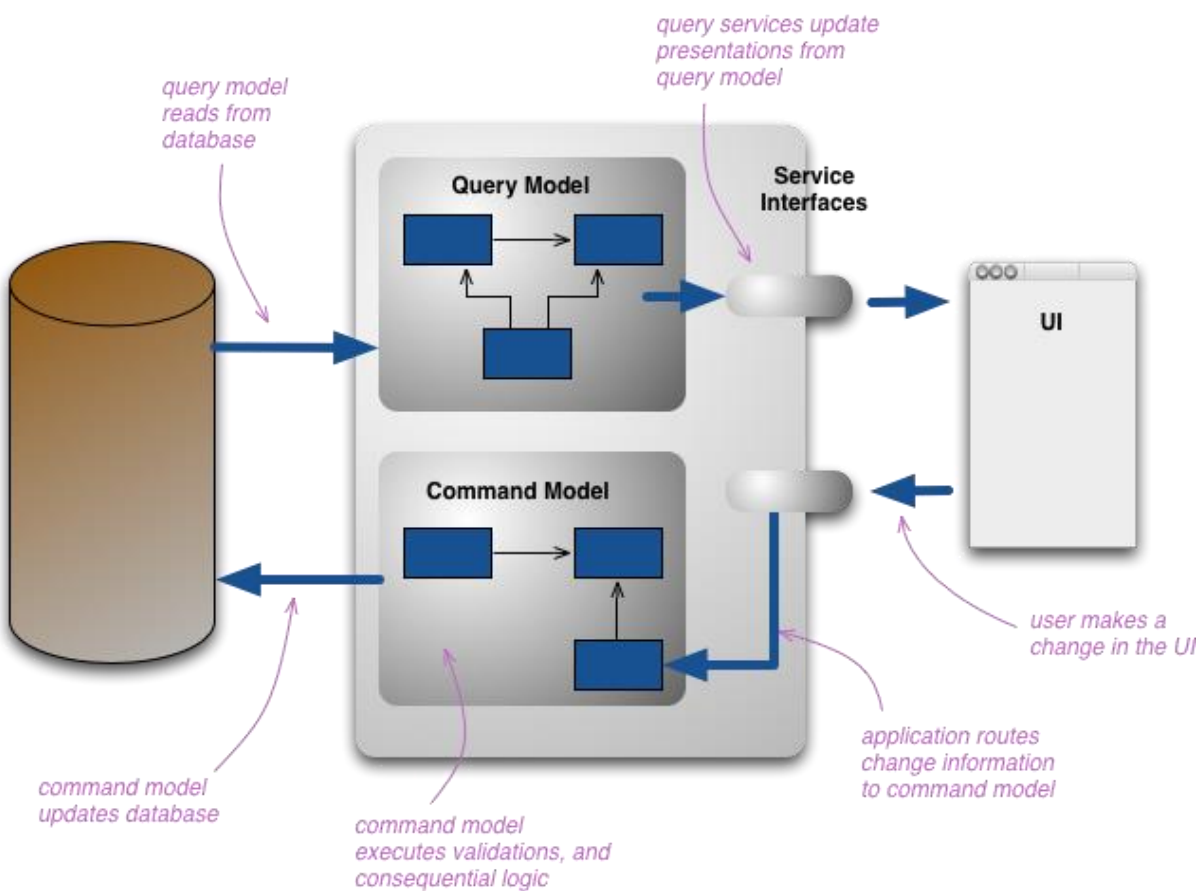
Layered Architecture diagram:



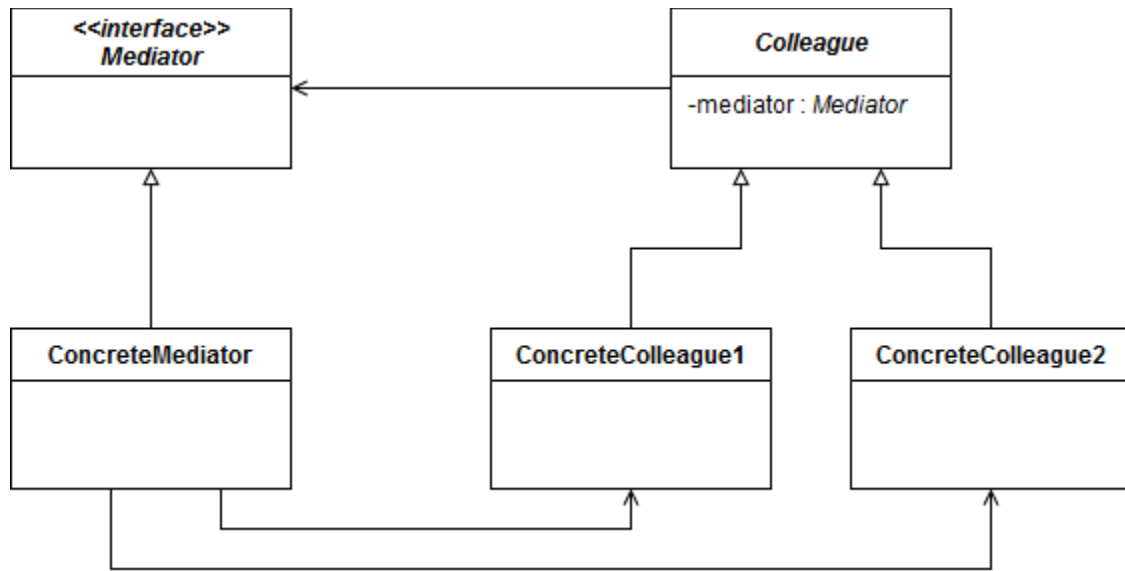
Client-Server diagram:



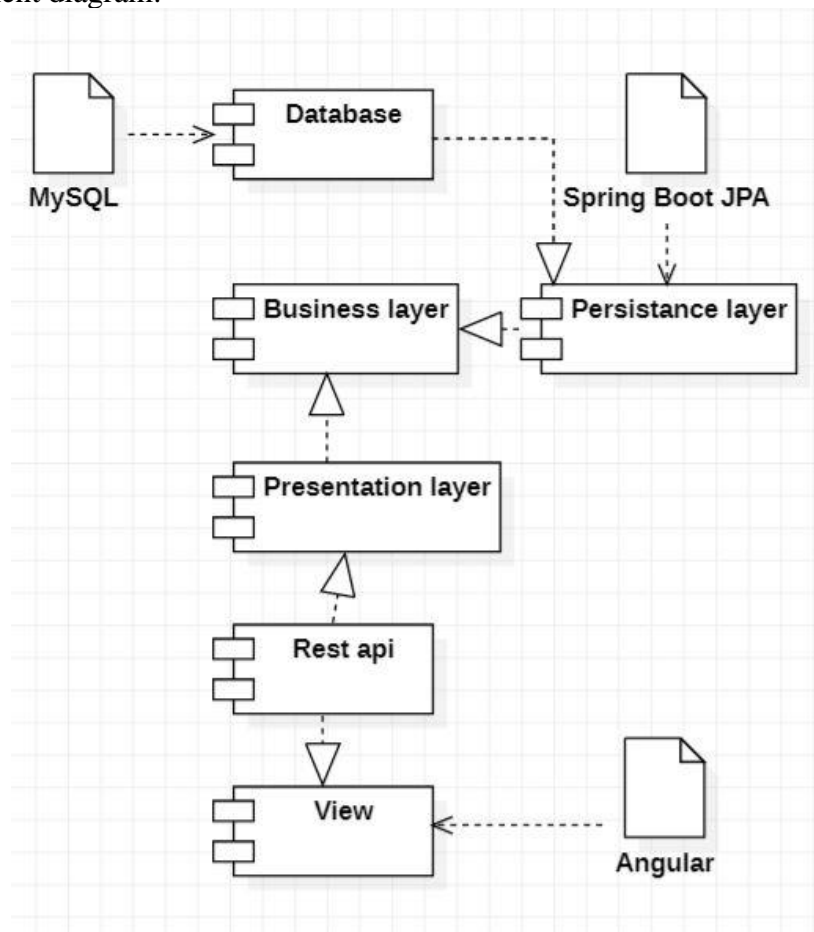
CQRS diagram:



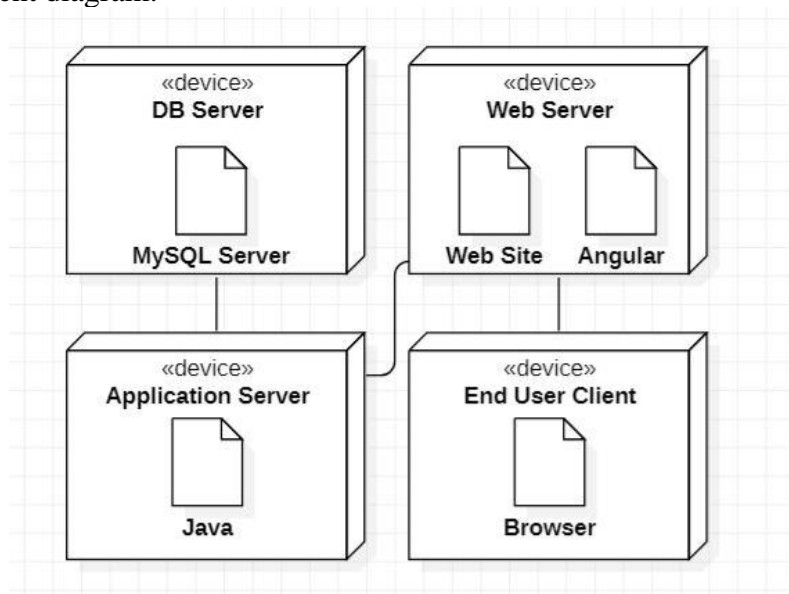
Mediator pattern:



Component diagram:

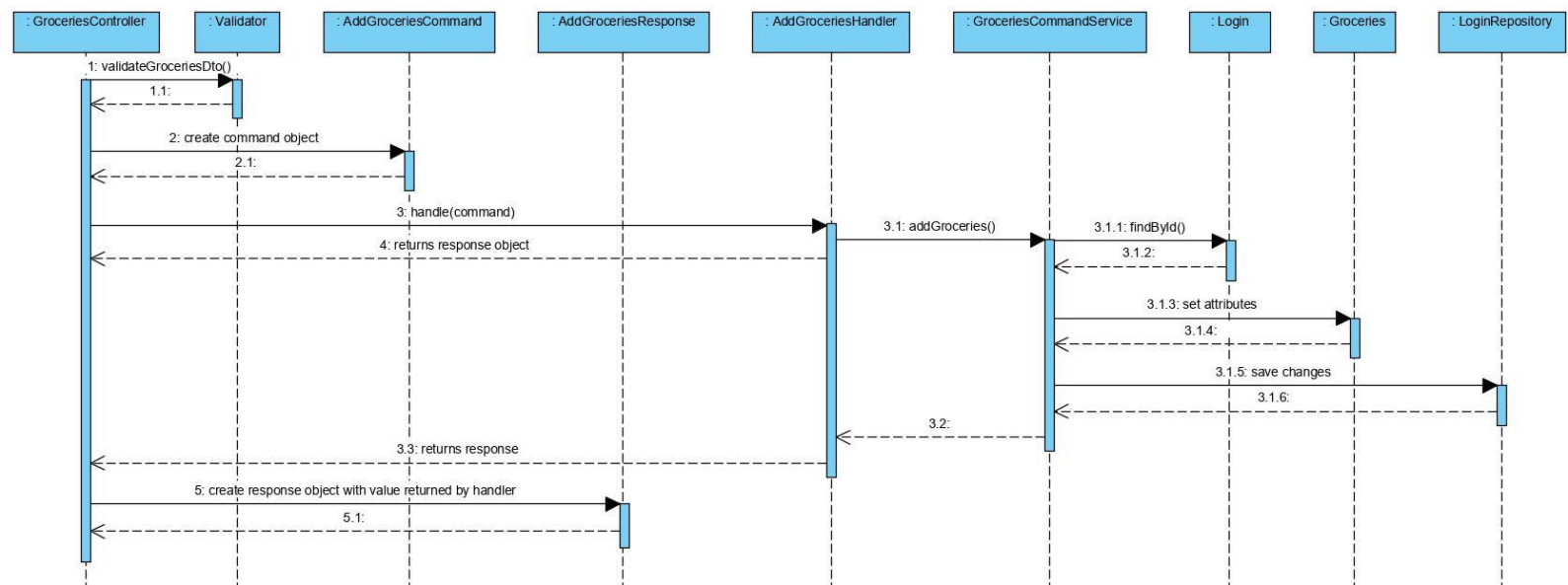


Deployment diagram:



4. UML Sequence Diagrams

sd Add grocery item

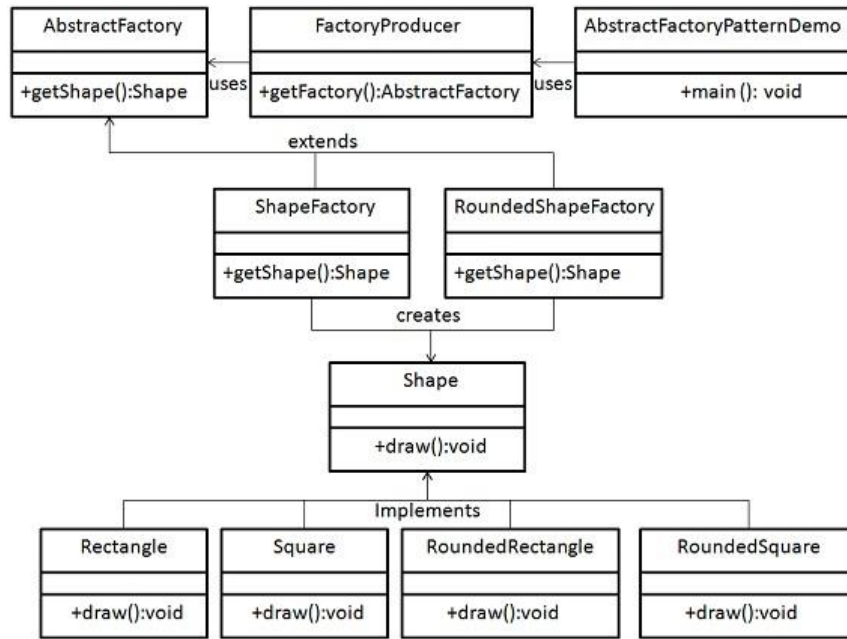


Sequence diagram for adding a grocery item to a user's groceries list.

5. Class Design

5.1 Design Patterns Description

The **Abstract Factory Pattern** was implemented in this application to aid in the creation of weekly and monthly reports. This pattern is a creational pattern. It is responsible for creating a factory of related objects without having to specify their classes. This pattern is useful in providing a more loosely coupled code, that is easy to extend and maintain. Unlike the factory method pattern, the abstract factory pattern uses abstraction in the development of the factories as well. Such that it is able to provide families of related objects, not only single objects. The abstract factory pattern uses composition in its architecture.

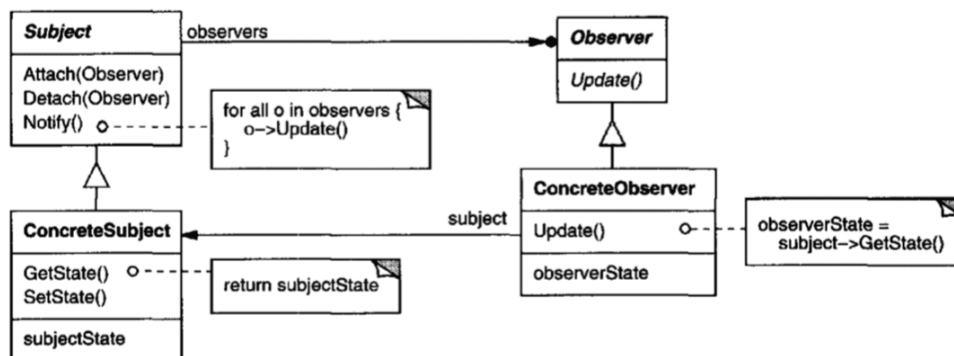


1.3-1 Abstract factory pattern

The **Observer Pattern** was implemented on the **client-side** of the application in order to notify the user of food waste based on the sum of ideal burndown rates of individual items and the goal value. The Observer Pattern is a behavioral pattern. It is used when one object needs to automatically notify one or many other objects of some changes within it.

In this implementation, there are two interfaces, one for the Subject and one for the Observer. They contain the method definitions that any subject and observer must implement. Namely, **attach**, **detach** and **notify** for the subject, and **update** for the observer.

Concrete classes implement these interfaces: **ConcreteSubject** and **ConcreteObserver**. The concrete subject contains some business logic in which the sum of the burndown rates is computed and compared with the goal. If the sum exceeds the goal, the state of the subject is set to 1, otherwise it is set to 0. The concrete observer is notified, and the **excess** variable is set accordingly. Finally, the excess variable is checked in the html file to see if it is necessary to notify the user of excess food or not.

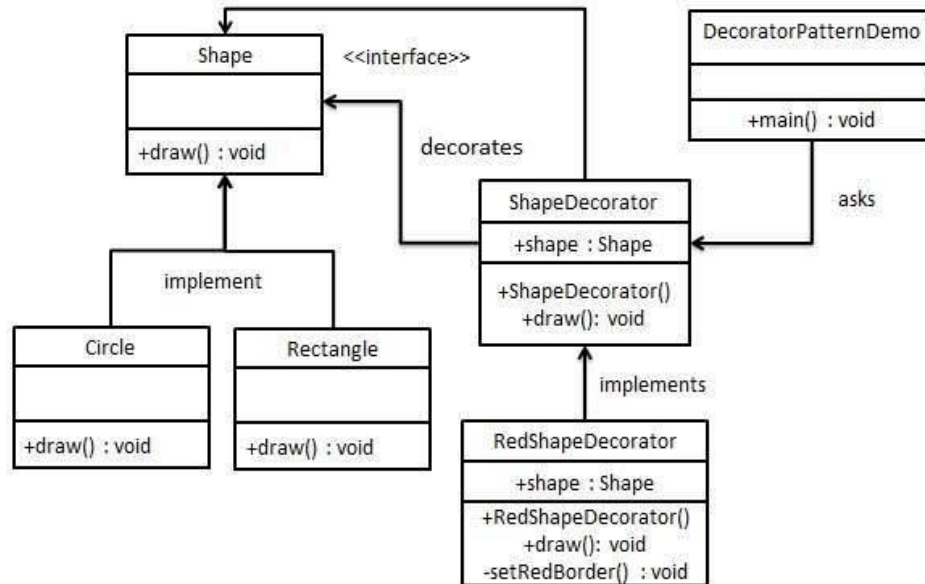


1.3-2 Observer pattern

The **Decorator pattern** is used to differentiate between reports in which the burndown rate is under the goal value (GREEN – signaling no food wastage) and above the goal value (RED – signaling food wastage).

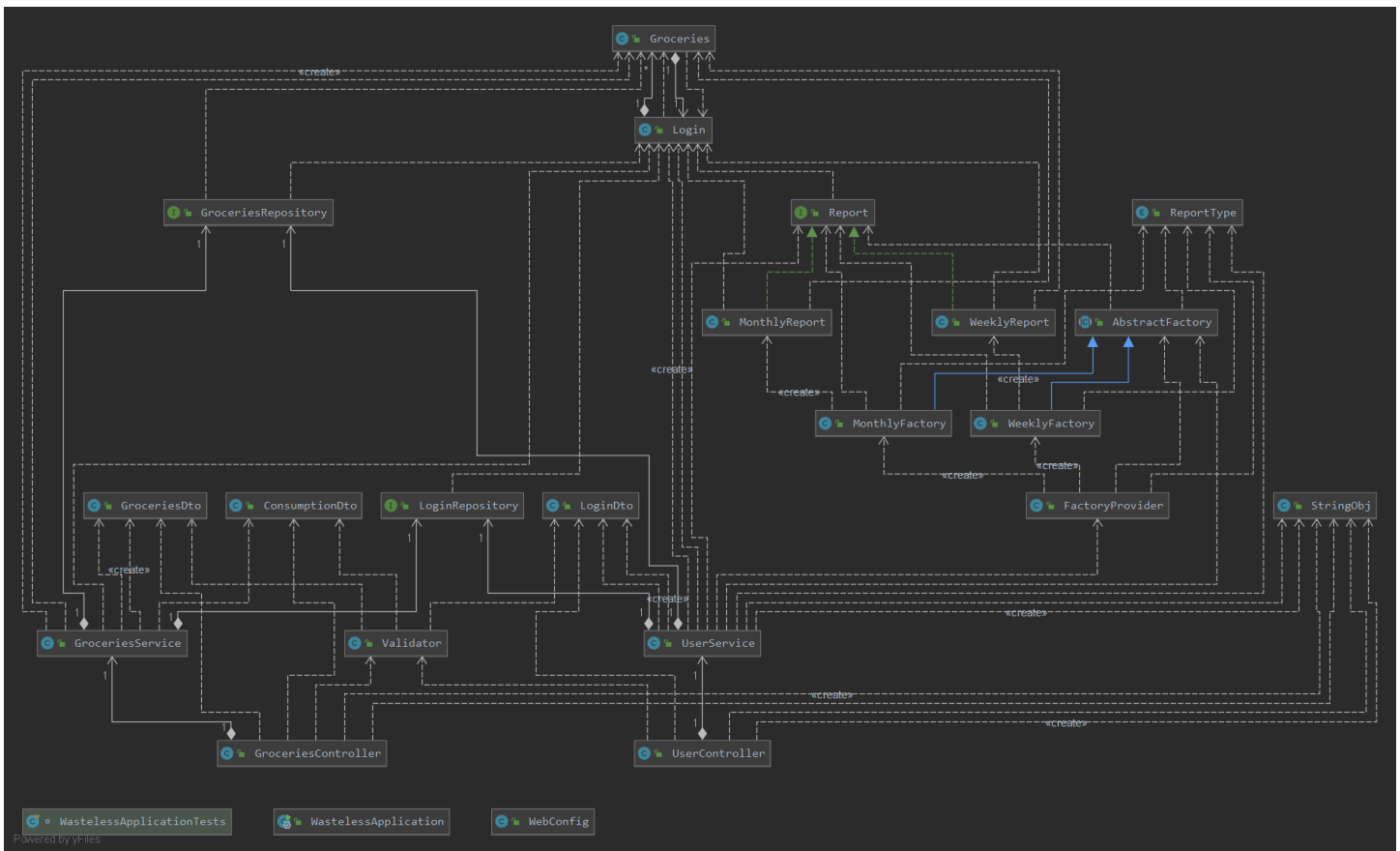
The pattern was implemented by implementing the Report interface in the ReportDecorator class. The ReportDecorator class should also have a report object as its attribute. The GreenDecorator and RedDecorator extend the ReportDecorator and each provides an extra method used to decorate the classic reports by adding a specific string at the beginning of the report text (either “Not wasting groceries!” or “Wasting groceries!”).

In the client side, the UI will display a green text report if the report contains “Not” as the first 3 characters, or a red text otherwise.



1.3-3 Decorator pattern

5.2 UML Class Diagram



The client interacts only through the **Presentation layer** represented by the controllers **UserController**, **GroceriesController** and the **DTOs** (the controllers receive and transmit information requested by the user in a special format called data transfer object which contains only the necessary information of a specific interaction). The **Business layer** communicates with the presentation layer. It contains the methods and the models necessary for obtaining the data the user requests. It consists of services **GroceriesService** and **UserService**, entities **Groceries** and **Login**, and the **Factory** used for the creation of the **Reports**. The next layer, that the business layer connects to on the other end, is the. **The Persistence layer** contains the repositories **GroceriesRepository** and **LoginRepository**. The repositories **Persistence layer** transmits to and receives queries from the **Database layer**. The Database layer is the final one. It consists of a **MySQL database** which stores the users' information and their grocery lists.

Other packages that are used are: the config package, which contains the necessary information to configure the server and the client. The final package is the **utilities** one. It contains the **Validator** used in the presentation layer, in the controllers, to parse and validate the user's input.

[updated class diagram is available in the project folder – UML.png]

6. Data Model

- Groceries model
 - Represents the grocery item of a grocery list
 - Name
 - Quantity
 - Calories
 - Purchase date
 - Expiration date
 - Consumption date
- Login model
 - Represents the user's credentials, groceries and goal. Has a dependency to Groceries model
 - Username
 - Password
 - Goal
 - Groceries list

Groceries		
f	id	int
f	name	String
f	quantity	int
f	calories	int
f	purchase_date	Timestamp
f	expiration_date	Timestamp
f	consumption_date	Timestamp
f	loginFK	Login
m	getId()	int
m	setId(int)	void
m	getName()	String
m	setName(String)	void
m	getQuantity()	int
m	setQuantity(int)	void
m	getCalories()	int
m	setCalories(int)	void
m	getPurchase_date()	Timestamp
m	setPurchase_date(Timestamp)	void
m	getExpiration_date()	Timestamp
m	setExpiration_date(Timestamp)	void
m	getConsumption_date()	Timestamp
m	setConsumption_date(Timestamp)	void
m	getLoginFK()	Login
m	setLoginFK(Login)	void
m	getInterval()	int
m	getTotalCalories()	int
m	computeBurndownRate()	int
m	toString()	String

Login		
f	id	int
f	username	String
f	password	String
f	goal	int
f	groceryLists	List<Groceries>
m	getGoal()	int
m	setGoal(int)	void
m	getUsername()	String
m	setUsername(String)	void
m	getPassword()	String
m	setPassword(String)	void
m	getId()	int
m	setId(int)	void
m	getGroceryLists()	List<Groceries>
m	setGroceryLists(List<Groceries>)	void
m	toString()	String

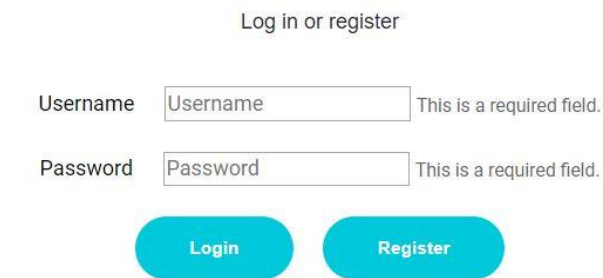
7. System Testing

Unit testing was used to test the functionality of the application.

- Log in/Register page

Has input validation integrated. Requires user to input a username and a password.

Wasteless application



The screenshot shows a web form titled "Wasteless application" with a subtitle "Log in or register". It contains two input fields: "Username" and "Password". Each field has a placeholder text and a validation message "This is a required field." to its right. Below the fields are two blue buttons: "Login" and "Register".

- Add grocery item page

Has:

- Input validation
- Form for grocery item
- Name; must be made of words containing letters and spaces
- Quantity; must be a strictly positive number
- Calories; must be a strictly positive number
- Purchase date
- Expiration date; must be after purchase date
- Add button, updates groceries list
- View groceries list button; directs to menu of user profile

User: user1

Fill to add grocery item

Name	<input type="text" value="Burgeri"/>
Quantity	<input type="text" value="2"/>
Calories	<input type="text" value="300"/>
Purchase Date	<input type="text" value="08/04/2020"/>
Expiration Date	<input type="text" value="11/04/2020"/>

[Add](#)

[View groceries list](#)

- Menu page
 - Has:
 - Goal display and set goal button
 - Log out button
 - Add consumption date button, with drop-down select of item name and data input
 - Add grocery item button. Redirects to the add groceries page
 - View weekly and monthly report buttons. Displays corresponding result underneath table.
 - Table of grocery items. Displays current grocery lists of user and affiliated data.

User: user1

Goal: 150

Log out

Groceries List

Burndown rate: 116

New goal 150

Set goal

Grocery item Burgeri ▼

Consumption Date 11/04/2020

View weekly reports

Add consumption date

Add grocery item

View monthly reports

Name	Quantity	Calories	Purchase Date	Expiration Date	Consumption Date	Burndown Rates
Burgeri	2	300	2020-04-08	2020-04-11	2020-04-10	200
Paste Integrale	1	150	2020-04-10	2020-04-14	N/A	37
File somon	4	200	2020-04-11	2020-04-14	2020-04-11	266
Lapte	3	450	2020-03-17	2020-04-03	N/A	79

User: user1

Weekly report

Goal: 2000

Number of groceries purchased in the last week: 3

Number of groceries consumed in the last week: 2

Number of groceries expired in the last week: 1

Number of calories consumed in the last week: 1400

Number of calories wasted in the last week: 600

8. Bibliography

- <https://dzone.com/articles/factory-method-vs-abstract>
- <https://docs.oracle.com/javase/8/docs/api/java/sql/Timestamp.html>
- https://www.tutorialspoint.com/design_pattern/abstract_factory_pattern.htm
- <https://refactoring.guru/design-patterns/observer/typescript/example>
- https://www.tutorialspoint.com/design_pattern/observer_pattern.htm
- <https://culttt.com/2015/01/14/command-query-responsibility-segregation-cqrs/>
- https://www.tutorialspoint.com/design_pattern/decorator_pattern.htm
- <https://www.baeldung.com/java-mediator-pattern>