<Assignment 3>

Student: Trif Marina Bianca

**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

The application is called WasteLess and provides help for users in order to manage the food they buy. This is done by allowing them to create items from their groceries and add details about each of them:

* name of the product
* quantity in grams
* nr of calories
* the purchase date
* the expiration date
* the consumption date

The user must first login in order to have access to any of the feature, therefore a JWT login service is provided for this part.

The application generates a report by showing the user relevant numbers about the items that expired such that food waste management can be improved.

The user can also create lists by selecting items and adding them to a list. All the grocery lists are displayed on the main page/dashboard, but the user can select one grocery list, see the items that it contains and/or make modifications.

The user has the option to view warnings regarding the items that will expire the following day. If there are many, there is a donate button which provides user with donation options in case the food waste might be too big. This conclusions can also be drawn by looking at the report, which is generated and highlighted by the system.

All data is stored in a database.

# Functional Requirements

Functional requirements for users management:

* The system must provide a form for the users to register
* The system must register the user details and store them in the database
* The system must provide a form for the users to login
* The system must verify if the username and password provided match with the ones in the database
* The system must display the user details, except the password, on the dashboard
* The system must provide users options for donating excess food
* The system must warn users if there are items close to expiring

Functional requirements for items management:

* The system should provide users the possibility to add an item to the database, with their desired details
* The system must display all the items added on the dashboard alongside with their details
* The system must compute and analyze the data about items in order to notify the user about its conclusion on food waste
* The system must provide a report based on the ideal burndown rate and on the expiration details

# Non-functional Requirements

Performance

- The app’s response time is good and is able to register/log a user in less than 2s.

- The app should render the dashboard within 2 seconds from logging in

Security

- By using the JWT model of authentication, the user cannot access data that is not meant for them to be seen. It also keeps user data secure and only passes the necessary information to the frontend

- The app should display the personal details only for that user

- The app should end the session when the user opts to log out, so other users can’t access their personal data

Data integrity - The app generates exceptions if the data provided is not compatible with the type that it’s supposed to have

Availability - The app should render each request of the user within 2 seconds from the time of the request

-The app should immediately update the database after a new item/list is added in order for it to be available and displayed on the dashboard right after the user requests the addition of the item, without having to refresh the page to see the new content

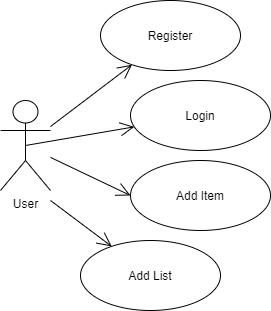
Capacity – The user can create as many items/lists as they want, storage being locally limited since the application is not yet deployed

Portability - The app should be portable on each browser and the user must have the same/similar experience and the same access to all features

Maintainability - The app should be easily maintained on each of its layer by having components as separate as possible, such as controllers, services, models and context.

Scalability – Since the chosen architecture is a layered one, the division between the presentation layer, the business layer and the data access layer makes it fairly easily scalable. Therefore, other features can be added without changing much of the structure of the application, including big modifications such as adding another architectural layer or another design pattern.

2. Use-Case Model



Use case: Add Item

Level: user-goal level

Primary actor: user

Main success scenario:

* User clicks on Add Item
* User enters item name
* User enters quantity
* User enters calories
* User enters expiration date
* User enters purchase date
* User enters consumption date
* User attempts to add item
* Item is created
* User is redirected to dashboard to see all items

Extensions:

* User clicks on Add Item
* User enters all data except item name
* User attempts to add item
* User is not allowed since item name is required
* User must add item details again

3. System Architectural Design

**3.1 Architectural Pattern Description**

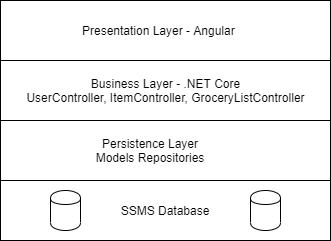
The main architectural pattern used is the layered architecture. The presentation layer is represented by the client side, which is built with Angular 9.

The backend is implemented with .NET Core and EF Core. The business layer is represented by the controllers which manage the data and retrieve information or send it to the persistence layer. The persistence layer consists of the models and the repositories which access the database context.

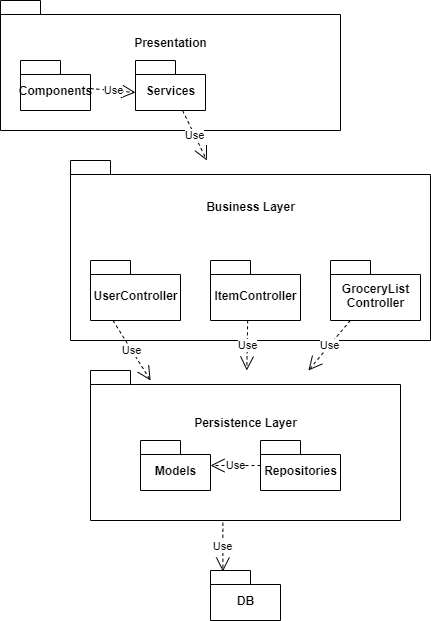
For storing the database SQL Server was used.

**3.2 Diagrams**

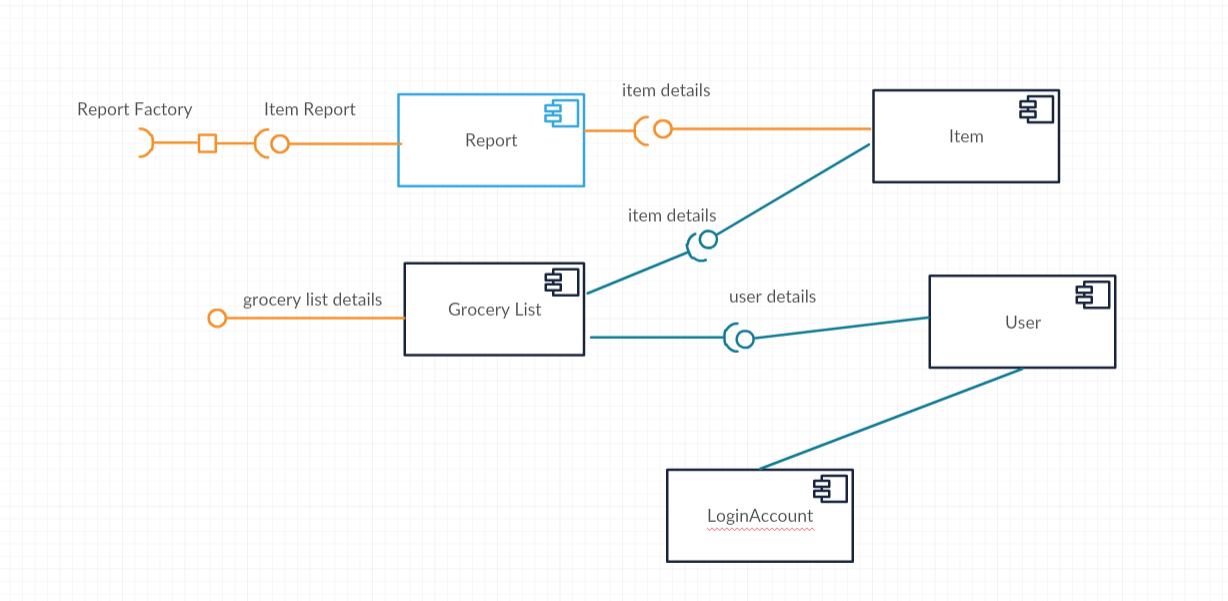
Layered Architecture diagram



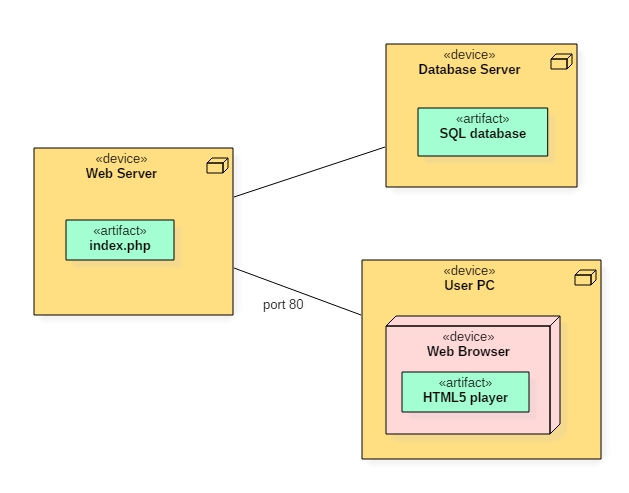
Package Diagram



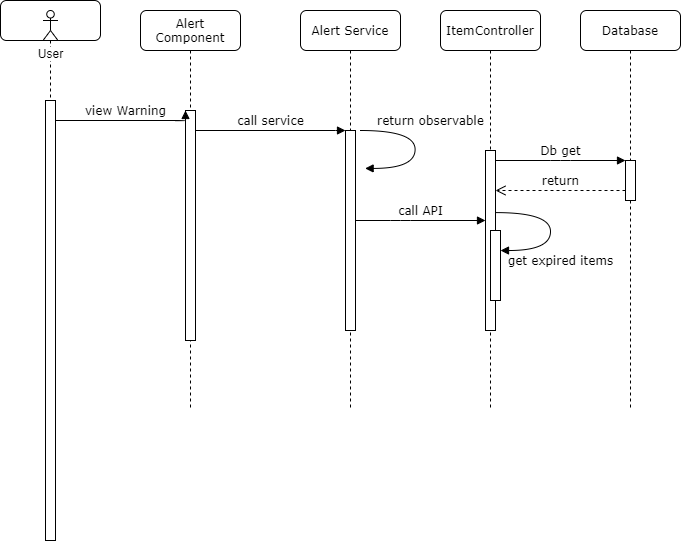
Component diagram



Deployment diagram



4. UML Sequence Diagrams



5. Class Design

**5.1 Design Patterns Description**

Since EF core implements the repository pattern, a layered approach was chosen. Therefore, the data access layer uses the context and LINQ to query the database and make changes to the tables. In this application, the Item repository, Grocery List Repository and User Repository deal with data access.

The next layer is given by the Controllers, which have no access to the database context, but receive HTTP requests and manage them by calling the methods in the data layer.

The requests come directly from the client by using REST API services. The client side implements these services and renders the presentation layer by the use of components.

Angular implements the Singleton pattern when declaring a service by using the @Injectable decorator and setting the provideIn property to ‘root’. For the service to be Singleton it is included in AppModule or in module that is only imported by AppModule.

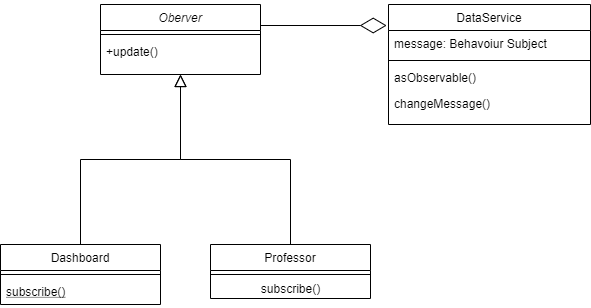
For passing data between components I used the Observer design pattern and implemented it with the help of RxJs BehaviourSubject. Observers can subscribe to the subject to receive the last (or initial) value and all subsequent notifications.

The data service provides the message source as a Behaviour Subject sending it as Observable to whoever injects the data service. The dashboard component subscribes on this message the list id for the view list page. The view list component subscribes on the local string message whatever received from the dashboard component through the data service. The reason the communication is done this way is because the two components that need to pass data between one another are not in ant other way related. Hence, the data service as a third party can be used.

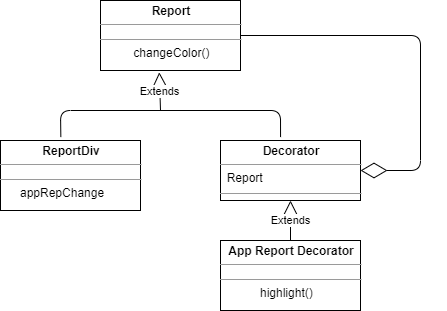
The decorator pattern was used for changing the color of the report generated by the application when a certain condition is met. It was done by making use of the Directive decorator, which takes an element, in this case a div, and changes its color based on a host listener. The decorator is placed on the html div, based on the selector given by the input.

**5.2 UML Class Diagram**

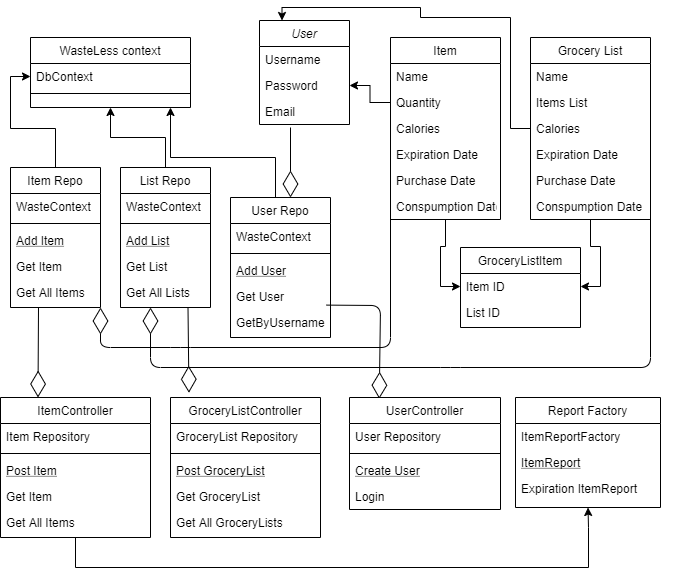
Observer pattern



Decorator pattern



Class diagram



6. Data Model

The data models are the following:

User model:

public class User

{

[DatabaseGenerated(DatabaseGeneratedOption.Identity)]

public int UserId { get; set; }

public string Username { get; set; }

public string Email { get; set; }

public string Password { get; set; }

public ICollection<GroceryList> GroceryLists { get; set; }

}

Item model:

public class Item

{

[DatabaseGenerated(DatabaseGeneratedOption.Identity)]

public int ItemId { get; set; }

public string ItemName { get; set; }

public int Quantity { get; set; }

public int Calories { get; set; }

public DateTime PurchaseDate { get; set; }

public DateTime ExpirationDate { get; set; }

public DateTime ConsumptionDate { get; set; }

public IList<GroceryListItem> GroceryListItems { get; set; }

}

Grocery List Model

public class GroceryList

{

[DatabaseGenerated(DatabaseGeneratedOption.Identity)]

public int GroceryListId { get; set; }

public string GroceryListName { get; set; }

public IList<GroceryListItem> GroceryListItems { get; set; }

}

We use a Grocery List Item model in order to define the many to many relationship between Item and Grocery List, needed since it was a code first approach:

public class GroceryListItem

{

public int GroceryListId { get; set; }

public GroceryList GroceryList { get; set; }

public int ItemId { get; set; }

public Item Item { get; set; }

}

We define a Login model in order to only get the username and password from the database and send it to the front end:

public class LoginModel

{

public string Username { get; set; }

public string Password { get; set; }

}

The front-end part is divided in 11 components:

* The app component/ app module which contains all the other and the router outlet.
* The alert component which generates the user warnings/notifications
* The auth component used for logging in/out
* The dashboard component which renders the most information and acts as the home page
* The donate component which renders the donation option
* The grocery list component for adding new grocery lists
* The item component for adding new items
* The login component for performing login/logout
* The registration component for adding new users
* The report component for rendering the reports
* The view list component for viewing every grocery list details

Data models that are used are similar to the ones declared on backend such that the field matching can be done automatically when necessary.

7. System Testing

Validation testing is done for every object/model sent through an HTTP post request to the controller. If the object sent was null the API method returns a BadRequest exception alongside with a significant message: for example, “Item is null”.

[HttpPost]

public ActionResult Post([FromBody] Item i)

{

if (i == null)

{

return BadRequest("Item is null");

}

\_itemRepository.createItem(i);

return NoContent();

}

Field validation is done on Angular forms on the front end by specifying whether a certain field is required or not. If it is, the form cannot be submitted unless the user provides a valid input for each field.

<input class="form-control" #***UserName***="ngModel" name="UserName" [(ngModel)]="formModel.UserName" required>

8. Bibliography

[1] - <https://docs.microsoft.com/en-us/ef/core/>

[2] - <https://www.entityframeworktutorial.net/efcore/configure-many-to-many-relationship-in-ef-core.aspx>

[3] - <https://angular.io/guide/architecture>

[4] - <https://www.geeksforgeeks.org/abstract-factory-pattern/>

[5] - <https://refactoring.guru/design-patterns/abstract-factory>

[6] - <https://blog.angular-university.io/angular-jwt-authentication/>

[7] - <https://www.entityframeworktutorial.net/efcore/entity-framework-core-migration.aspx>

[8] - <https://angular.io/guide/observables>

[9] - <https://jasonwatmore.com/post/2019/07/05/angular-8-alert-toaster-notifications>

[10] - <https://fireship.io/lessons/sharing-data-between-angular-components-four-methods/>

[11] - <https://xgrommx.github.io/rx-book/content/subjects/behavior_subject/index.html>