PERSONAL MEAL PLANNER

A meal planner with focus on
personal dietary goals and food waste

DEPARTMENT OF COMPUTER SCIENCE

CS-23-SW-7-05, Fall 2023

SEMESTER PROJECT

STUDENT REPORT

AALBORG UNIVERSITY

Title:
Personal Meal Planner

Theme:
Recipes to reduce waste and cost

Project Period:
Fall 2023

Project Group:
CS-23-SW-7-05

Participants:
Andreas Hove Paludan,
Ibrahim Alaiddin Abu Rached,
Julian Aarup Larsen,
Kristoffer Møller Gregersen,
Magnus Jørgensen Harder Christensen,
Martin Langgaard Jacobsen,
Peter Schwartz Lauridsen,

Supervisors:
Dalin Zhang,
Dazhuo Qiu

Keywords:
Semester Project, Computer Science, Meal Planning, Food Waste, Microservices

Page Count:
63

Date of Completion:
19/12/2023

Electronics and IT
Aalborg University
http://www.aau.dk

Abstract:

This report presents a food waste reduction and healthy eating initiative via a user-friendly food planner app targeting Danish households. It addresses global food waste issues and emphasizes designing a health-focused planner.
The requirement specification outlines essential features, including user login and personalized dietary, health, and food inventory data, alongside a meal plan generator. Project management challenges such as communication issues and deviations from the project management agreements are discussed. Architectural choices, like adopting a microservices approach. The report concludes with a reflection on imbalanced time allocation between architecture setup and feature development, which ultimately led to a suboptimal solution. Future work recommendations emphasize iterative improvements to the meal plan service, API gateway functionality, and potential architecture reconsideration for a better balance.

AALBORG UNIVERSITY
STUDENT REPORT

Contents

1. PREFACE

This project report was produced by the software group CS-23-SW-7-05 from Aalborg University in the fall of 2023. We chose the topic ”Recipes to reduce waste and cost” as we found it interesting, and we thought that we might use the solution ourselves.

The goal of this project was to develop a meal planning web application that could generate meal plans from a user’s dietary goals while taking the food that they have in their home into account to reduce food waste.

The project describes the process of finding problems with existing solutions and coming up with requirements for our solution that tackle some of the issues that the existing ones do not. We proceeded to work with an architecture that was not suited for the time allotted to us, which ultimately resulted in a solution that did not meet all of our must have requirements.

We would like to thank our supervisors Dalin Zhang and Dazhuo Qiu for the support and feedback on this report as we worked on it.

2. INTRODUCTION

This project details the issues of food waste and healthy eating and presents a proposed solution to combat both issues. We recognize the importance and scale of both issues, particularly food waste being one of the leading causes of climate change. We seek to analyze both the problem of food waste and healthy eating, resulting in a problem statement that defines the goal of our proposed solution.

Upon deciding on an architecture, we design all the components of our application, both frontend and backend. With a clear goal and design, we move to our development phase, where we implement our requirements and describe interesting aspects of our application. To ensure the correctness and stability of the code, we carry out a testing phase to conclude our development process. This leads to a discussion of our findings and the various aspects of the project. Finally, we conclude our project to assert whether we achieved what we set out to do. This semester includes elective courses, which may cause time management issues and inter-group communication. We aim to combat this by outlining measures to avoid any issues that this split schedule may spawn. With challenges of time management and inter-group communication stemming from elective courses, we set clear project management guidelines and chose an architecture to best suit this split schedule and potentially avoid poor time management and communication.

3. PROBLEM ANALYSIS

With the initial project proposal revolving around the issue of food waste, we want to look at the overall waste issue. The initial project proposal suggested a possible solution of an application that suggests recipes to a user based on the food already in their fridge or pantry. This could potentially help reduce food waste and help users save money on food expenditures. We have a common interest in keeping healthy through physical exercise and keeping track of our nutritional intake. Therefore, we also want to investigate integrating nutritional tracking into the project. For this, we want to look into the field of health in general and nutritional tracking to assert whether there is a broader interest in health tracking.

The following problem analysis will first investigate food waste, followed by an analysis of the issue of healthy eating and nutritional tracking. From this problem analysis, we conclude whether there is demand for an application that seeks to reduce food waste and keep track of users' nutritional intake.

3.1 FOOD WASTE

In this section, we investigate the problem of food waste. We look at the global and domestic sides of the issue and the different movements that have emerged to combat the issue.

Food waste is one of the leading causes of climate change, producing around 270 million tons of CO_2 annually[1]. Up to 40% of the world's food production is wasted, equating to 2.5 billion tons of food wasted annually[2]. In Denmark, the common households make up for 28.8% of the country's total food waste[3]. This number spikes to 43% in the United States[4]. The issue of food waste has led to many articles being written in recent years about how to reduce food waste and the benefits of doing so. The two largest benefits of reducing food waste for the average household are a reduced carbon footprint and a reduced grocery expenditure[5].

The issue of food waste is prevalent in all parts of the world, and there is a global focus on food waste reduction[6][7]. To reduce food waste around the world, many movements have emerged to help spread awareness of the issue and give advice on how to reduce food waste. One of the most well-known movements focusing on spreading awareness and giving advice on how to reduce food waste in Denmark is Stop Spild Af Mad(Stop Food Waste)[8]. There are many other movements both in Denmark and around the world, all to reduce global food waste[1, 9, 10].

With worldwide organizations spreading awareness and educating the population on the issue of food waste, it is clear that the problem is very important. As stated, food waste is one of the leading causes of climate change, making it a problem worth looking at solving.
As the initial project proposal suggested, one way to reduce food waste in households would be to recommend recipes based on the ingredients the users already have at their disposal. This is a path we want to investigate further. With this approach, we also want to attempt to recommend healthy meals to users. Therefore, we want to investigate the topic of healthy eating, and as an extension, we want to delve deeper into nutritional tracking.

3.2 HEALTHY EATING

In this section, we investigate the topic of healthy eating and nutritional tracking.

Obesity rates have been rising since their initial reporting. In Denmark, the prevalence of obesity has risen from 6.1% in 1987 to 18.4% in 2021[11]. In the United States, the obesity rate is as high as 42% based on data from 2017 and 2020[12]. The Centers for Disease Control and Prevention attribute the potential causes of prevalent obesity to unhealthy eating habits, lack of physical activity, and insufficient sleep[13]. For this reason, it makes sense for us to promote healthy eating habits when recommending recipes. Our definition of healthy eating habits is eating within a set daily caloric window to lose or maintain body weight. Within the caloric window, calories should be well-balanced between proteins, carbohydrates, and fats, meaning the body gets all the macronutrients it needs to function optimally.

One way to ensure healthy eating habits is to keep track of one's nutritional intake. When eating in a caloric surplus, the body stores the unnecessary calories as body fat. If this pattern of overeating continues for long enough, it will eventually lead to obesity[14]. Keeping track of one's daily caloric intake makes it easier to avoid overeating.

Calorie counting and nutritional tracking have spawned many articles describing the benefits and the hows of keeping track of daily food intake. Many articles consider the main benefit of tracking calories, such as a better overview of food consumption and the caloric value of various foods[15]. With a better overview and knowledge of the energy contents, choosing healthier food options and making better grocery shopping decisions becomes easier.

With the benefits of calorie counting and food waste reduction established, we now want to investigate existing solutions that tackle the problems of food waste and healthy eating habits.

3.3 TARGET DEMOGRAPHIC

As described in sections 3.1 and 3.2, food waste and healthy eating are global issues affecting many people. However, we do not find it feasible to tackle these issues on a global scale. For this reason, we want to limit our target demographic to Denmark.

In Denmark, the target demographic for reducing food waste is very large, with food waste being present both in the industry and in households. Since we are considering food waste and healthy eating, we find it reasonable to limit our target demographic to Danish households.

Given that the obesity rate in Denmark has risen to 18.4% as of 2021, it is sensible to conclude that the issue of healthy eating is present in Denmark. Therefore, In Denmark, there is a potential demographic interested in healthier eating and getting a better overview of how much they consume.

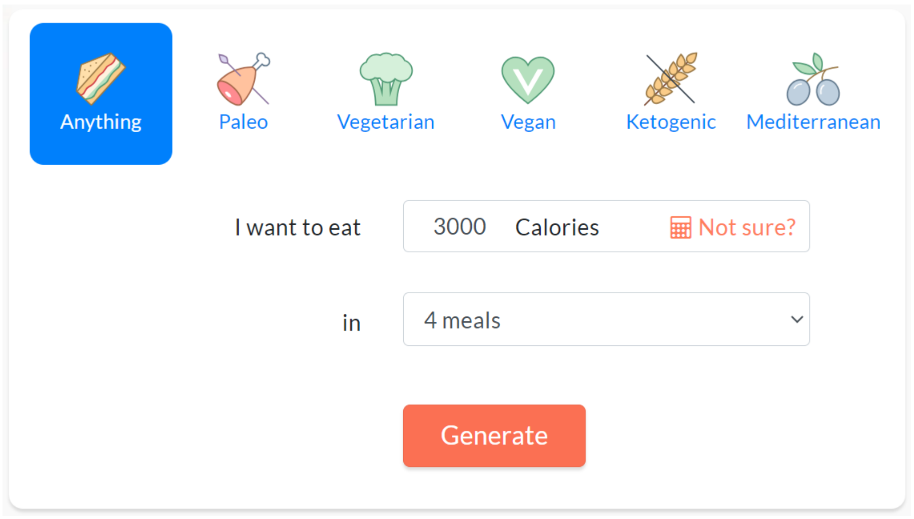
Based on our analysis of food waste and healthy eating, we have narrowed our target demographic to Danish households and individuals who want to eat healthier while reducing their food waste.

3.4 EXISTING SOLUTIONS

In this section, we look at existing solutions for the problem of food waste extending to the issue of healthy eating. This will lead to a comparison of the existing solutions, which will determine if any of them have the potential to solve the problems described in the previous sections.

3.4.1 EAT THIS MUCH

Eat This Much is a website that generates meal plans summing up to a user-specified amount of calories. Figure 3.1 shows the meal plan generation UI of Eat This Much. As is seen, there are a variety of dietary options available, such as vegetarian, ketogenic, and vegan. Choosing a dietary option, a calorie target, and a number of meals lets the user generate a meal plan. A meal plan has meals with various details such as cooking time and calories. Appendix A.1 shows a generated meal plan with one meal. Clicking a meal displays a pie chart describing the macronutrient distribution of the meal. There is a list of directions and automatic ordering of ingredients from AmazonFresh. The website is designed for users in the United States, as it uses the imperial measurement system and only supports ordering ingredients from AmazonFresh[16].



I want to eat 3000 Calories

in 4 meals

Figure 3.1: Eat This Much meal plan generation page[16]

3.4.2 NO WASTE

No Waste is an app that tracks the expiration dates of food in the fridge and pantry. The user manually enters food items and their expiration dates. When the food is about to expire, a notification is sent to the user. The food is sorted by expiration date, name, or category. The user can also perform a search to find food items in their inventory list and track the food waste by deleting the food as eaten or expired[17].

3.4.3 KITCHE

Kitche is an app that helps track food at home and recommends recipes based on the user's diet and selected food items. Food items can be added to the inventory by scanning a food barcode or a supermarket receipt. Users can request recipe recommendations based on different dietary options and selected food items from their inventory.[18].

3.4.4 MYFITNESSPAL

MyFitnessPal is an application focused on nutritional tracking and health monitoring. Users can track their calorie and macronutrient intake, weight change, and exercise. The application has a barcode scanner for easy food tracking. This helps users keep an overview of their eating habits to stay or get healthy.[19]

3.4.5 COMPARISON OF EXISTING SOLUTIONS

With the existing solutions investigated, we want to assert whether they solve the problems described in our problem analysis. For this reason, we compare the existing solutions on three criteria. Firstly, we want a solution that attempts to reduce food waste. Secondly, we want the solution to help monitor health through calorie tracking. Thirdly, we want the solution to recommend recipes to the user.

Table 3.1 shows the functionalities of the existing solutions we have investigated alongside our proposed solution. As can be seen, none of the described applications tackles all the issues covered in the problem analysis. Our proposed solution should solve or partially solve all the issues we have discussed.

	Reduce food waste	Health monitoring	Recipe recommender
Eat This Much		X	X
No Waste	X		
Kitche	X		X
MyFitnessPal		X	X
Our Solution	X	X	X

Table 3.1: Comparison of different apps based on what problems they can solve

While the existing solutions address individual issues described in our problem analysis, none tick all the boxes. The No Waste application focuses on reducing food waste by tracking the expiration dates of foods in the users' homes. Eat This Much and MyFitnessPal focus more on health tracking and recommending recipes but do not address the issue of food waste. Our goal is to combine the ideas of the current partial solutions and create an application that tackles the issue of reducing food waste, tracking health, and recommending recipes. This leads us to our problem statement.

4. PROBLEM STATEMENT

This chapter describes our problem statement which then leads us to a requirement specification. Lastly, we cover our project management guidelines, which we will use to effectively manage our time and workforce.

4.1 PROBLEM STATEMENT

Throughout our analysis, we investigated the issues of food waste and healthy eating. Here we found that food waste is a leading cause of climate change and that obesity rates have been rising in the past four decades. This led us to investigate existing solutions that tackle these issues, and we found that no current solution directly tackles the issues. This has then led us to our problem statement:

We want to develop an application that generates personalized meal plans for users based on personal dietary needs and available ingredients to promote healthy eating and minimize food waste.

4.2 REQUIREMENTS

Now that we have established our problem statement, we move on to define our requirements. We have prioritized these requirements by using the MoSCoW method. We exclude won't have requirements. Our reasoning for this is that we do not find it relevant to include requirements that we have no intention of implementing during this iteration of the project.

4.2.1 MUST HAVE

Our must have requirements are listed below:

- User login system
- User health profile
- User ingredient inventory
- Personal dietary data
- Food database from supermarkets

- Generate meal plan from user inventory and personal dietary data
- Save meal plan to database
- Use recipes from recipe websites

These requirements describe what we need to implement to have a satisfactory MVP[1]. Firstly, we need our application to be usable by users, who have unique profiles. This is described through our first four must have requirements, listed below. These are a login system, a health profile, personalized users with dietary data, and an ingredient inventory. The dietary data and health profile are aimed at tackling the health monitoring aspect of our application. The ingredient inventory is aimed at reducing food waste, by keeping track of which foods the users have in their fridge or pantry. Our fifth requirement, a food database from supermarkets, is then used to supplement the ingredient inventory, with knowledge of the foods' macronutrient contents.

The remainder of our must have requirements tackle recipe recommendations. We firstly want to be able to generate meal plans based on a user's inventory, and their dietary data. The dietary data consists of a user's daily required calories and daily macronutrient targets. We then want to be able to save the generated meal plans to a database. Each meal plan should be linked to the user that generated the meal plan. Then, to generate the meal plans, we need a way of getting recipes, which our next requirement states that we want to get from websites and vendors.

4.2.2 SHOULD HAVE

Our should have requirements are listed below:

- Show macronutrient contents for recipes
- Food consumption statistics
- Use leftover ingredients for multiple recipes
- Discount fetcher
- A more user-defined meal plan

These requirements aim at improving our MVP by adding valuable functionality to our solution. Our first should have requirement is to show macronutrient content for recipes in the generated meal plans. Alongside our second should have requirement, food consumption statistics, these are aimed at helping users keep track of how many calories they consume. The third requirement is to use leftover ingredients for multiple recipes. For this requirement, we need to have a clear overview of the quantity of each ingredient the user has in their inventory. Then it is a matter of updating their inventory, each time a recipe needs a certain amount of an ingredient from the user's inventory. We also want to implement a discount fetcher that finds any nearby discounted ingredients needed for a recipe in a user's meal plan. Finally, we need a more user-defined meal plan. From our must have requirements, users are only able to generate a

[1]MVP: Minimal Viable Product

meal plan based on their user information. With this should have requirement, users will be able to choose how many days the meal plan should span and which meals they want each day.

4.2.3 COULD HAVE

Our could have requirements are listed below:

- Barcode and receipt reader
- Additional macronutrient logging
- Optional dietary constraints for meal plans

These requirements are those which add meaningful improvements to our solution but are of lesser importance than the should have requirements. Firstly, we want to add a barcode and receipt reader to make it easier for users to update their ingredient inventory. With a barcode and receipt reader, a user would only need to scan either the receipt from their grocery shopping or the barcodes of the foods that they purchase. We also want to add the option for users to log other sources of calories and macronutrients outside of the generated meal plans. This would further improve the health monitoring aspect of our application. Furthermore, we want to further improve the flexibility of the meal plans by allowing users to set specific dietary restrictions, such as vegetarian or carnivore, or if they have any food allergies.

4.3 PROJECT MANAGEMENT

This section details our project management, detailing both our risk- and time management processes.

In the development process, numerous decisions must be made, each presenting opportunities for both success and error. At every critical step of the process, we need to decide between different options. A single misstep in decision-making could result in a substantial loss of valuable development time or, in the worst-case scenario, yield a subpar final product that fails to meet the minimum requirements.

To preempt the occurrence of poor decisions, we want to establish comprehensive guidelines for decision-making throughout the development cycle. Each decision made should be backed by sound reasoning and a thorough thought process. This approach compels us to engage in critical thinking and research before finalizing any critical decision, thereby minimizing the risk of making suboptimal choices.

To effectively manage our time within the limited project timeframe, we will be using GitHub's backlog board feature. This backlog will serve as a central repository for tracking tasks, including new, ongoing, and completed ones. Assigning team members to specific tasks on the backlog board will enhance communication and provide a clear overview of each member's current responsibilities.

Given our team size of seven members, we recognize the potential for parallel work on different aspects of the product and project. To capitalize on this, we plan to adopt some of the agile development principles by forming smaller sub-groups when working on distinct parts of the product. This strategy aims to increase overall productivity and accelerate progress. However, to maintain effective communication, we must ensure continuous interaction between these sub-groups.

In summary, our risk management is:

1. **Decision-Making Guidelines:** Establishing comprehensive guidelines for decision-making, ensuring each choice is supported by sound reasoning and critical thinking.
2. **Time Management:** Implementing a GitHub-hosted backlog system to effectively manage time and provide a clear overview of tasks, fostering communication and oversight.
3. **Parallel Work and Communication:** Adopting agile principles by forming sub-groups for parallel work on different aspects of the project, with a commitment to maintaining continuous communication between these sub-groups.

By implementing these risk management measures, we aim to enhance the overall efficiency of our development process and minimize the likelihood of setbacks.

5. DESIGN

This chapter describes the design phase of the project, including our choice of architecture, architectural design, frontend design, and database design.

5.1 CHOOSING AN ARCHITECTURE

In this section, we present and describe commonly used software architectures in order to decide on an architecture for our project.

When considering the right architectural choice for our project, we want an architecture that is easy to scale in the future, to both tackle the increasing number of users and new features that might hinder the performance of the application. This means that we are looking for an architecture that is easily adaptable for future requirements and scalability. Many of the group members have different courses and the time that we have where we are all available and working, is limited, we also want an architecture that can facilitate this, so that the different group members are not entirely dependent on other group members' work.

5.1.1 MONOLITHIC

Monolithic architecture represents a unified, self-contained software unit, where an independent application is built upon a singular code base. This approach offers advantages in terms of streamlined deployment and simplified end-to-end testing, requiring only a single executable file for deployment and facilitating debugging with all code centralized. However, the monolithic architecture is not without its challenges. Scaling individual components becomes more intricate as all code resides in a single location, limiting adaptability. Additionally, flexibility is constrained by the technologies upon which the monolith is built, and even minor code changes mandate a complete redeployment of the application [20].

A monolithic application uses a stateful architecture. A stateful architecture stores and maintains user data inside the server as a session, while a stateless architecture stores user sessions in a database layer. In a stateful architecture, scalability is an issue, since user sessions have to be replicated on new servers, as the application is scaled up[21].

The benefits of a monolithic architecture, rooted in its singularity, are evident in the efficiency of interactions within the application and the ease of troubleshooting [22]. Nevertheless, the simplicity of this architecture presents its own set of drawbacks. The singular code base, while initially advantageous, may pose challenges in long-term maintenance as the application's complexity expands. Furthermore, collaborative development becomes challenging when teams need to work on different aspects of the application [22].

5.1.2 MICROSERVICES

A microservice architecture is broken up into a collection of independent software units called services. These services are independently deployable and loosely coupled. This means that each service is deployable without depending on other services to be deployed alongside it. Each service is its own executable file with its own code base. The decoupled nature of the code, makes it highly maintainable and testable, though end-to-end testing will become more tedious[23]. With each service being independent, it allows development teams to use the best-suited programming language for each service.

The microservice architecture excels at providing frequent and reliable software delivery. This is in large part due to the independent nature of the services and the ease of testing each service. Where a microservice architecture falters, is in its end-to-end testing across the different services and its increase in resource requirements as more services are added. As each service requires its own code base and individual database, it will require a lot of resources. Since microservices are intended for larger applications and are generally considered too time-consuming to implement for smaller applications, the resource requirements will be a relevant issue[24]. The initial development overhead, when adopting a microservice architecture is also larger than a monolithic or layered approach. This is in large part due to the need for well-defined services and their responsibilities, their APIs, and how they are integrated. The need for well-defined architectural choices and reasonings is paramount for a successful microservice application[23].

Another large selling point of the microservice architecture is its scalability, with each service being individually scalable in response to demand. It is also possible to implement dynamic automated scaling through monitoring of a microservice system[25]. Microservices are also typically stateless, meaning that each Service simply receives a request, processes it, and sends back a response without storing anything internally in the Service. Instead, services store states externally, typically in a database[26]. The stateless nature of microservices also lets the architecture be scalable without potential loss of user data, as the data is stored externally.

5.1.3 LAYERED

A Layered architecture organizes code into distinct layers, each representing a logical division with communication facilitated through requests and services. While the number of layers may vary, common layers include the presentation layer, housing the frontend, and the database layer. Communication between these layers involves the presentation layer sending requests downward through intervening layers to the database layer, which then responds with the requested data[27].

There are different variants of the layered architecture which can be chosen depending on the requirements of the project. The most common variants are the strictly layered and the relaxed layered architecture. The strictly layered enforces layers to only communicate with layers directly above and below themselves, whereas a relaxed layered architecture allows for higher order layers to communicate with any layer below it.

The layered architecture promotes modularity, making it easier to organize and maintain code by separating concerns into distinct layers. The logical division into layers enhances the comprehensibility of the codebase, aiding developers in understanding and navigating the system. Different layers focus on specific functionalities, facilitating the separation of concerns and enhancing code maintainability[27]. The advantages of using a strictly layered architecture are the clear distinction of responsibility and the maintainability also increases. In the relaxed variant, it can become harder to understand the responsibilities of each layer and it can also be harder to get an overview.

The layered architecture is deployable as a single unit or as several units, depending on the details of implementation. Depending on how the architecture is implemented, deployment can be difficult, especially for larger applications. Changing one thing in a component can potentially require a redeployment of either a large part of the system or the whole system. This means that the architecture is not necessarily well-suited for continuous delivery[28]. The need for communication between layers introduces potential latency and complexity, particularly as the number of layers grows[27].

A layered architecture can be either stateless or stateful, depending on the needs of the system. If the system requires scalability, the architecture should be stateless. If the system should instead enable a richer user experience, the architecture should be stateful[21].

5.1.4 COMPARISON

In this subsection, we will delve deeper into the aspects of each architecture mentioned at the start of this section. We will be focusing on the scalability, requirement adaptability, and flexibility of the architectures for working in a disjoint environment.

SCALABILITY

The reasoning for focusing on scalability in our application stems from the ease of development and the feature development in the future, which is also one of the goals of the study plan for this semester[29].

Before continuing we need to have an understanding of what the different options are for scaling an application. These options are explained below:

Horizontal scalability is when one will increase the amount of systems that the application is run on so that there is a distributed net of systems that are running the application. These systems are typically managed by a request load balancer that monitors the traffic of each system and redirects requests to systems with smaller loads[30]. This results in less downtime, because if a system goes down the load balancer simply redirects to a system that is running. This also offers the flexibility of continuous upgrades as the systems can be gradually redeployed over time. The reliability is also a great benefit of using this method of scaling[31].

The horizontal scaling also has some downsides. Buying multiple systems costs more money than buying a single system, resulting in a higher initial cost.

Having multiple systems also increases the cost of running and maintaining them. It can also be harder to maintain multiple systems[31].

The applications that are best for horizontal scaling are stateless, as they do not lose any information needed to perform their actions when they experience downtime.

Vertical scalability refers to increasing the computing power of the system that the application is run on. If an architecture using this model goes down the downtime might be greater, as there is only one machine. This also does not provide the great flexibility of continuous upgrades, as the system needs to be shut down and restarted, and as there is only one system, clients are unable to access the application in the meantime. There is also only a single point of failure, meaning if the machine's hardware breaks, the whole application is down for the time it takes to repair the machine.

The benefit of this approach is that it is cost-effective since there is only a need for one machine to facilitate the application. There is also no need for a whole load-balancing layer on top of it. This decreases the complexity required to run and maintain the application [31]. Opposite to horizontal scaling, stateful applications are great for vertical scaling as they tend to follow the same ideology.

We have chosen to measure the architecture's ability to scale horizontally, as this can, in the future, facilitate many concurrent users and continuous feature additions, with its distributed approach.

- **Monolithic** applications are single executables and tend to be stateful, which is a direct match for the vertical scaling approach. This is not great when wanting an application for many concurrent users as it is limited to the hardware available. The computing power of single components is limited.

- A **microservices** application with components as a service approach is great for running multiple services on multiple machines and having them communicate, making microservices a great choice if the goal is to have a highly scalable application.

- **Layered** applications are similar to monolithic applications but are split up into layers that have a broader responsibility. A typical layered application is comprised of frontend, backend, and database layers, which are individually scalable. However, this does not make a layered application as scalable as microservices, as the backend responsibility tends to be divided between the microservices in a microservice approach.

REQUIREMENT ADAPTABILITY

As requirements might change and evolve, we want the architecture to be able to easily adjust to these changes, while staying online or having the smallest amount of downtime.

The **monolithic approach** does not facilitate a gradual shift to newer versions of the application, and it has to have downtimes as all the components in the application are tightly coupled. The monolithic architecture is also constrained to the technologies it is built upon in its initial release. This creates a lot of technical debt and needs a lot of refactoring if a framework or internal design patterns need to change. When working with a monolithic architecture it is also important to be aware of where features are implemented in the internal structure to maintain the responsibilities of each component.

In a **microservice approach**, a gradual shift to newer versions of the application is easier. However, if the interfaces each service exposes change, upgrading the application becomes a bigger task. In the microservice architecture, the responsibilities of each service are well defined and if the area of responsibility for a service increases significantly it is better to split the service than to have it grow. The focus on service responsibility is high and is very important when planning to implement a new feature. When keeping the responsibilities low of each service framework, the tasks of upgrading and migrating are smaller compared to a monolithic approach.

The **layered approach** can be hard to adapt to new requirements similar to the monolithic approach as implementations of layered tend to follow monolithic approaches. Changes in separate layers can be made, if the interface that it exposes stays the same or only exposes new functionality. If one would change an already functional interface function, it could require more refactoring on other layers[28].

FLEXIBILITY

Given that this semester has elective courses, resulting in group members having a shifted schedule, we want to have great flexibility when working on the project. Therefore, we want a flexible architecture to facilitate that each group member does not need to be present for the development of the application to go forward.

- The **monolithic architecture** has tightly coupled components, meaning that the whole group needs to be gathered to make decisions as this can create changes in multiple components which is not ideal. It is also harder to divide the group into teams, with specific areas of responsibility.
- **Microservices** are tailored for teams as each service is its own project and codebase with its own responsibilities, only requiring an interface, through which the other services can communicate with it. This means that a smaller subset of the group can work on a service, and not complicate another service's work.
- A **layered architecture** can be divided into frontend, backend, and database layers, enabling the group to be divided into three teams working on a layer each, independently of the other layers.

5.1.5 SUMMARY AND DECISION

Following the above analysis and comparison of the monolithic, microservice, and layered architectures, We now summarize the comparison and decide on which architecture we adopt.

1. **Scalability**
    - **Monolithic**: Limited scalability due to its stateful nature and reliance on vertical scaling. Components are limited by the hardware available.

- **Microservices**: Highly scalable with a distributed approach, allowing for individual scalability of each microservice.
- **Layered**: Can be scaled individually, but less scalable than microservices.

2. **Requirement Adaptability**

   - **Monolithic**: Limited adaptability due to tight coupling. Gradual shifts and upgrades may lead to significant downtimes.
   - **Microservices**: Facilitates gradual shifts to newer versions. Well-defined responsibilities of microservices make upgrades more manageable.
   - **Layered**: similar to monolithic it can be hard to adapt new requirements to their similar approach. Changes internally are easier.

3. **Flexibility**

   - **Monolithic**: Tightly coupled components require the whole group for decision-making. The division into teams with specific responsibilities is challenging.
   - **Microservices**: Tailored for teams, allowing smaller subsets to work independently on specific microservices.
   - **Layered**: Divisible into frontend, backend, and database layers, enabling teams to work independently on each layer.

The following table shows our classification of each architecture with the comparison metrics:

Architecture	Scalability	Requirement Adaptability	Flexibility
Monolithic	Limited	Limited	Limited
Microservices	High	High	High
Layered	Medium	Limited	High

Table 5.1: Classification of architectures

Here we have compromised the above listing into a table that shows the best-suited architecture, from our comparison. In conclusion, the microservice architecture aligns well with our project's goals, offering horizontal scalability, adaptability to changing requirements, and flexibility in a disjoint working environment.

5.2 ARCHITECTURAL DESIGN

This section describes the architecture of the application from the microservices, communication, and security aspects.

Our microservice architecture is designed to be scalable, so that each of our microservices can function independently of our other services, except for our meal plan service, which relies on both the inventory- and recipe service, to function. This is also the case for the databases as our microservices can work with independent databases, thus allowing for partial availability from our API.

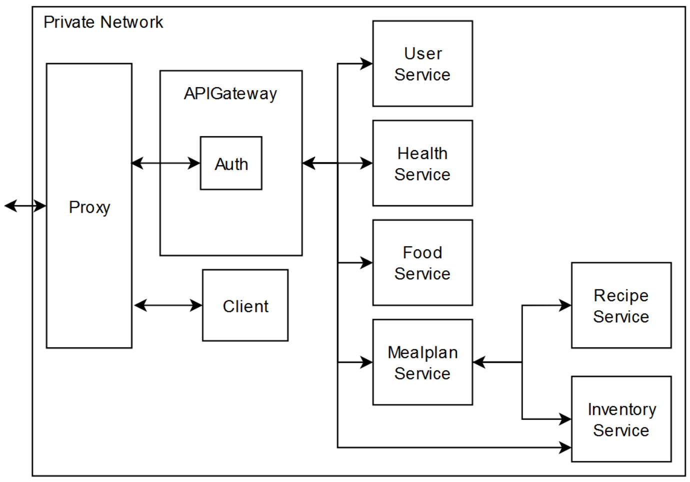


Figure 5.1: Our microservice architecture visualization

The following sections will explain Figure 5.1 in detail.

5.2.1 INTERNAL COMMUNICATION

The services must be designed in such a way that it is easy for them to communicate with each other. There are several ways of implementing communication between the services, such as gRPC or HTTP.

gRPC is often used as it provides a standard for communication in the terms of gRPC Services[1] with methods[2] and messages[3], which are defined in protocol buffers[4] that are independent from the language being used[33]. This means that these buffers can be used by all services. This also provides a well-defined messaging protocol between each service. The downside of gRPC is that there needs to be an implementation available for the language that is being used for each service. Otherwise one has to manually implement the messaging protocol. In short, there is a lot of overhead for defining gRPC compatible services.

Implementing the communication with basic HTTP requests provides a fast development cycle as languages commonly used to write web servers have HTTP libraries available. And there is not a lot of overhead for listening to HTTP requests and sending them. The drawback of using HTTP as the communication is that the microservices that can communicate with each other need to manually write the classes corresponding with the response and request messages, whereas gRPC and protobuf often have code generators.

We chose to have our microservices communicate with each other through HTTP as a lot of us have used it before.

[1]gRPC service is the system that can be communicated with through gRPC
[2]gRPC method defines the input and output for the communication endpoint
[3]gRPC Message is the data being sent through methods
[4]Protocol buffer is a method of serializing structured data[32]

The microservices can communicate freely by using HTTP, as the only entry for the microservices by outside users is from the API gateway. This allows for unencrypted communication.

5.2.2 USER SERVICE

The user service is responsible for all actions users can perform on their immediate account, through endpoints.

The user service exposes the following endpoints:

- **POST** `/user`: This endpoint creates a new user from a user object containing the following information:
  - username
  - email
  - password
  - gender
  - birthday
  - daily calorie target
  - daily protein target
  - daily carbohydrate target
  - daily fat target
- **DELETE** `/user/{id}`: Deletes the user with a specified id.
- **PUT** `/user/{id}`: Updates the user with a new user object.
- **GET** `/user/{id}`: Gets the user object with the specified id, without password.
- **POST** `/validate`: validates a user from their username and password.

Since the application is targeted at people wanting to reduce food waste and be aware of their physical health, we give the users the option to include information on their daily calories, protein, carbohydrates, and fats targets. Our other microservices use this information to produce meal plans specific to the individual user's needs.

5.2.3 HEALTH SERVICE

The health service provides additional functionality for the user. This service lets the users make health entries, detailing their height, weight, fat percentage, muscle percentage, and body water percentage. Since the application is aimed at users with the desire to maintain or improve their physical health, these health entries serve as a way for users to keep track of their body composition. The health service makes it possible for users to see and track their progress over time, with the option to add a new health entry at any time.

Health Service has the following endpoints:

- **GET** `/health/history`: This request allows the user to obtain all their health entries.

- **POST** `/health`: This request takes the user's input and creates a new health entry for them.
- **DELETE** `/health/{id}` This request deletes a health entry from the user.

5.2.4 INVENTORY SERVICE

The inventory service lets the user keep track of which ingredients the user has in their fridge, freezer, and pantry. The user inserts all ingredients and their expiration date. This lets the application keep track of which ingredients are going bad and allows the application to choose recipes for the user, that use the available ingredients to prevent food waste.

The inventory has the following endpoints:

- **GET** `/inventories/user/{userId}`: Returns all inventories for a specific user.
- **POST** `/inventories`: Is used to create a new inventory.
- **POST** `/inventories/{id}`: This endpoint is used to create a new ingredient in an inventory.
- **DELETE** `/inventories/{id}`: Deletes an inventory along with all its ingredients.
- **DELETE** `/inventories/{id}/{foodId}`: Deletes one ingredient from a specific inventory.

5.2.5 FOOD SERVICE

The food service is responsible for keeping a database of all ingredients and their macronutrients. This is done through publicly available APIs, that provide information on various foods and their respective macronutrients. Food Service has some private endpoints that are not exposed to the front end. These include endpoints used to delete and add new food items to the database that should only be used by the administrators.

Food service has the following endpoints exposed:

- **GET** `/foods?query={foodname}`: This endpoint is used to get a list of foods whose name matches the query.
- **POST** `/foods/list`: Takes a list of food ids and returns the details of all foods in the list.
- **GET** `/foods/discounted?zipcode={zip}`: Returns a list of foods that are about to expire in stores located in the provided zip code.

5.2.6 RECIPE SERVICE

The recipe service provides recipes that match query parameters of calories, protein, fat, and carbohydrates. These parameters can be adjusted by an error constant so that the recipe service can find recipes inside a certain range. This allows for some deviance in the recipes being sent instead of an exact match. The recipe service also has the ability to fetch new recipes from a specified range of recipe APIs and add them to its database.

Recipe Service has the following API endpoints:

- **GET** `/recipe/{id}`: This endpoint allows for getting specific recipes from their id
- The following items are query endpoints that use the before mentioned parameters for querying the recipe database
  - **GET** `/recipe/random`: This endpoint exposes the ability to get a random recipe from the query parameters
  - **GET** `/recipe`: While the previous just gets one random recipe from the query parameters, this endpoint gets all recipes that match the query.

5.2.7 MEAL PLAN SERVICE

The meal plan service is responsible for the handling of meal plans. This includes generating meal plans, fetching one or all meal plans, and deleting meal plans for users. To generate meal plans, the meal plan service sends requests to the recipe service, with user specified calories and macronutrients as well as a specified meal type. The users choose the number of days and the number of meals for each day up to three per day. For each day, the user chooses a split for their daily target calories across the different meals. Once all parameters are specified by the user on the front end, they are sent to the meal plan service through the API gateway, which sends requests to the recipe- and inventory services and generates a complete meal plan. The meal plan is then sent back to the user and saved to the meal plan database.

- **GET** `/mealplan/{userID}`: Gets the latest meal plan for a specified user id.
- **GET** `/mealplans/{userID}`: Gets all meal plans for a specified user id.
- **POST** `/mealPlan/`: Inserts a meal plan id into the Mealplan table in the database.
- **POST** `/mealPlanRecipe/`: Inserts a plan id and a recipe id into the MealPlanRecipe table in the database.
- **POST** `/mealsPerDay/`: Inserts a meal count and the total macronutrients for a day in a meal plan into the MealsPerDay table in the database.
- **POST** `/generate/`: Generates a new meal plan for a user with the specified parameters and inserts it into the database.
- **DELETE** `/mealPlan/{planID}`: Deletes a meal plan with a specific plan id, for an authenticated user.

5.2.8 AUTHENTICATION AND API GATEWAY

While each microservice can communicate with each other freely, the client needs to communicate with the microservices through an API gateway, as some of these might expose sensitive endpoints and require authentication.

We have designed an API gateway to control the access to each microservice, this has been done in a way so the client needs to provide a JSON Web Token (JWT)[5] to access these sensitive endpoints.

The client is responsible for keeping the authorization token in a safe space, and safely disposing of it.

The endpoints of the API gateway are designed in such a way that the client never has to specify what the user's id is, as it is stored in the authorization token. This mitigates the risk that users can do something to items owned by other users. This also means that no microservice endpoint is directly exposed.

The endpoints defined for the API gateway are the following:

- **GET** `/user`: Retrieves user information.
- **POST** `/user`: Creates a new user.
- **DELETE** `/user`: Deletes a user.
- **POST** `/login`: Logs in a user.
- **POST** `/health`: Creates a health entry.
- **DELETE** `/health/{id}`: Deletes a health entry by id.
- **GET** `/health/history`: Retrieves health history.
- **GET** `/inventories`: Retrieves all inventories for the logged in user.
- **POST** `/inventories/{inv_id}`: Adds an item to a specific inventory.
- **POST** `/inventories`: Creates a new inventory.
- **DELETE** `/inventories/{inv_id}`: Deletes an inventory.
- **DELETE** `/inventories/{inv_id}/{item_id}`: Deletes an item from a specific inventory.
- **GET** `/foods`: retrieves all the food items that matches a query.
- **GET** `/foods/{id}`: Retrieves information about a specific food item.
- **GET** `/foods/discounted`: Retrieves discounted foods.
- **POST** `/generate`: Generates a meal plan.
- **GET** `/mealPlan`: Retrieves the current meal plan.
- **GET** `/mealPlan/all`: Retrieves all meal plans.
- **DELETE** `/mealPlan`: Deletes a meal plan.

Routing all API endpoints through the API gateway ensures that the endpoints are consistent when working with the API from the frontend.

[5]JWT is a way to safely transfer, with data integrity features[34]

5.2.9 CROSS-ORIGIN RESOURCE SHARING (CORS)

We do not want other websites to use our application API, as this could lead to cross-site request forgery (CSRF) which is where a user logs into a website that they think is ours but is another website, this can lead to unwanted requests to our API, such as user deletion or user updates, which the actual user did not make themselves.

Web browsers implement a security feature called CORS, which can restrict web pages to make requests to a web server. Restricting this to only include our client, mitigates the CSRF attacks from happening.

5.2.10 APPLICATION PROXY

For the API gateway and the client to be served through the same domain, we need to have a proxy that allows for redirection, depending on the endpoint the user tries to access. This also allows the application to seem to come from the same server, from the user's perspective.

The proxy is also responsible for serving the application through HTTPS with an SSL certificate. This enables encryption of data when the client and API gateway are communicating with each other. The SSL certificate also ensures that the server is trusted and has a secure connection.

5.3 FRONTEND DESIGN

This section details the design process of the UI for our application. We have created low-fidelity wireframes for each major page of the application to better visualize the layout.

We wanted to follow a mobile-first approach when developing the UI. This means that the UI is firstly designed for smaller screens and then adapted for larger screens. The decision was based on the fact that 58% of global internet traffic comes from mobile devices[35] and considering our use case, this percentage is probably even higher for our application.

When opening the website for the first time, a non-authenticated user will be presented with the login and create account page as seen in Figure 5.2(a). User account creation begins with entering the essential account information such as name, email, password, and location. The user can opt-in to use food waste discounts to customize the meal plan generation for using discounts from local supermarkets. Finally, four input boxes are used to enter daily macronutrient targets.

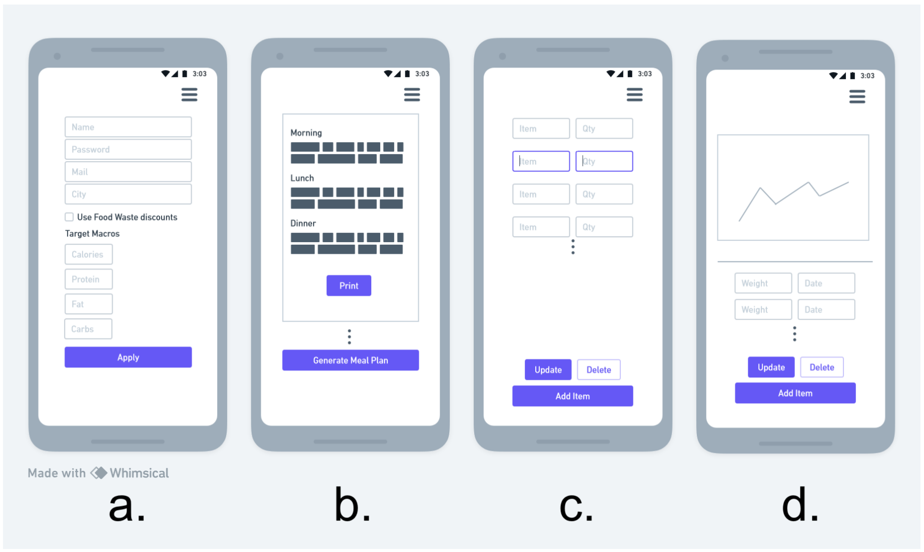


Figure 5.2: Low-fidelity wireframes. **a**: create account page, **b**: meal plans page, **c**: inventory page, **d**: health page. Can also be found in Appendix A.3.

The central activity for users is to check their current meal plan. When checking the recipe for the meals of the day, it should be quick and easy to see what ingredients to buy. Accordingly, the "meal plans" page as shown in Figure 5.2(b) is the application's home page. This page should provide an overview of the recipes planned for the next couple of days and show ingredients, instructions, and macronutrients for each dish to satisfy our requirement to show macronutrients for recipes.

The page also contains a button, that when clicked, opens a popup where you can create new meal plans, satisfying our requirements of generating and saving meal plans. This popup should contain a form where the user can specify their preferences like allergies and dietary restrictions, in addition to how many days to generate.

To satisfy our requirements for the ingredient inventory and ingredient database, we need a page where the user can manage their inventory. Our goal was to provide a simple interface for users to easily add new ingredients to their inventories with just a few clicks. The wireframe for the inventory screen, shown in Figure 5.2(c), consists of a simple list of ingredients with the name and quantity shown, in addition to options to edit and delete the item. Below is a button that opens a new page or a popup that allows the user to insert new ingredients. This page would consist of a form with the following input fields: name, quantity, expiration date, and a submit button. This should enable users to quickly insert new ingredients into their inventory. Keeping inventories up to date aids the meal plan generator in recommending recipes that minimize food waste. Maximizing the use of existing ingredients aids in fulfilling our requirements of generating meal plans from user inventories and using leftover ingredients for multiple recipes.

To fulfill our requirement of a user health profile, we want to implement a dedicated health tracking page as seen on Figure 5.2(d). This page provides an overview of the user's health and the option to input health data, including weight, fat percentage, muscle percentage, etc. The graph visualizes the user's health progress over time and the user will have the ability to toggle between which health metric is shown on the graph.

5.4 DATABASE DESIGN

This section describes our database designs for meal plan- and recipe services. This includes our reasoning and thought process when designing the databases and their Entity Relation Diagrams (ER Diagrams)

5.4.1 MEAL PLAN DATABASE

A meal plan is defined as a plan that can span over multiple days, comprised of one to three meals per day. The attributes required to have all the relevant information of a meal plan are stored in the meal plan database. These are:

- Start date
- End date
- Day
    - Total calories
    - Total protein
    - Total carbohydrates
    - Total fat
    - Meals
        - Recipes

The start and end date provides us with a timespan of the meal plan. The day attribute covers each day in the meal plan. Within each day, we have total calories, protein, carbohydrates, fat, and meals for that day. Lastly, each meal has a recipe.

Considering the list above, we need a `MealPlan` table in the database to define a meal plan with a start date and end date. Considering a meal plan can span multiple days, we have another table `MealsPerDay` that represents the individual days in a meal plan. This table describes the number of meals assigned to that day and the nutritional attributes, which are the total calories and macronutrients respectively. Lastly, we have a third table `MealPlanRecipes` that constitutes the recipes tied to a meal plan. This way, the table avoids the data redundancy that would come with inserting multiple recipes into the table.

We have illustrated the relations between the tables in the ER Diagram in Figure 5.3. Both the `MealsPerDay` and `MealPlanRecipes` entities have a one-to-many relation to `MealPlan` because there will always be just one meal plan, but there can be multiple days and recipes in a meal plan. To reduce the amount of attributes present in the diagram the total calories, protein, carbohydrates, and fat have been reduced as a single `nutrition` attribute.

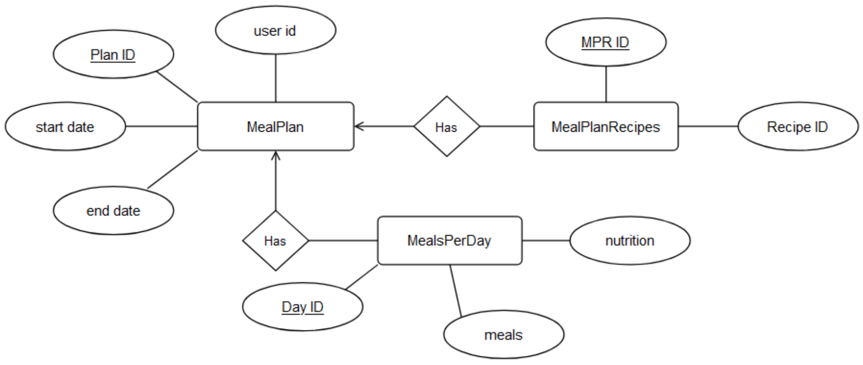


Figure 5.3: Entity Relationship (ER) diagram of the meal plan database schema

5.4.2 RECIPE DATABASE

Storing recipes in a database can be a difficult task as a recipe consists of many different parts. For our recipe database, we want our recipes to include the following attributes:

- Title
- Servings
- Instructions
- Ingredients
  - item
  - unit
  - amount
- Energy
  - Calories
  - Fat
  - Protein
  - Carbohydrates
- tags

We decided that this is the minimal amount of information a recipe is required to have, to be useful for our meal planning feature. The meal planning feature was designed to take the energy contents of a recipe into account when selecting recipes for the meal plan. The meal planner will also use tags to categorize the recipes into breakfast, lunch, and dinner. The Ingredients are both used so the user can make sense of the recipe and also so the meal planner can take into account what the user has in their inventory.

To represent these recipes in a relational database we need to design each entity that has a minimal amount of redundancy and flexibility.

From the above list of recipe attributes, we have chosen to split the attributes into several tables. First is a `Recipe` table which includes the title, servings, and instructions. Then to avoid redundant entries in the `Recipe` table, we created an `Ingredient` table, with entries of item, unit, and amount. A lot of recipes can have the same ingredient item and unit, creating a logical split of two more tables for `items` and `units` respectively, so that we would not have redundant item and unit attributes. The `Energy` attribute also has its own table, which we saw as a logical separation from the recipe, this also accommodates recipes that might not have energy information. The last attribute `Tags` has also been made into their own table with the same reasoning as `items` and `units`.

In Appendix A.2 the relations between the different tables have been described with an ER Diagram. Each `Recipe` and `Energy` has a one-to-one relation as it is unlikely that two recipes have the same energy contents. To represent that a recipe can have multiple ingredients we have used a one-to-many relation to `Ingredient`, so each ingredient has a reference to the recipe they belong to. Each `Ingredient` has a many-to-one relationship to the `Item` and `Unit` entity, as items and units are not unique to each ingredient. for example, two ingredients in different recipes can both include a tomato. It would result in redundant data if they were to be designed as an attribute. `Recipe` and `Tag` have a many-to-many relationship, to accommodate that a recipe can have many tags, and many tags can be used by multiple recipes.

5.4.3 RELATIONSHIPS ACROSS MICROSERVICES

In the microservices we have decided to use a database per service approach[36], to decouple the services completely. We have done this to allow each service to use the most appropriate database management system for its feature area. This also means that we cannot do relations traditionally.

In our databases, we have some fields that serve as conceptual foreign keys. However, since the databases are separate, we are unable to enforce the foreign keys traditionally. Instead, the fields will be provided when the API gateway sends a request to the service. This approach will make the fields act as foreign keys.

6. IMPLEMENTATION

This chapter details our implementation process, only covering our most interesting implementation details. Firstly, we discuss our five different services, followed by the implementation of our frontend to backend communication module, namely the API gateway. Towards the end of the chapter, the request handling on the frontend is detailed. Lastly, we show our visual design.

6.1 RECIPE SERVICE

The recipe service is a central part of the application as this is the part that is responsible for querying the recipe database and finding recipes that are inside the range of users' macronutrient needs.

The service includes functionality to get, create, and delete recipes by id, which we need for the maintainability of the recipe database. We could also use these functionalities in an administrator panel.

The most crucial function of the recipe service is the `recipe_by_query` function that exposes a friendly interface for querying recipes. The function is involved in both "**GET** `/recipe/random`" and "**GET** `/recipe`" endpoints, where these endpoints expose the same interface as the `recipe_by_query` function.

The formal parameters used in the endpoint are the following:

- Macronutrient parameters, which is the amount of macronutrients that the recipe must have, all these parameters are floating point numbers.
  - `calories`
  - `protein`
  - `fat`
  - `carbs`
- `energy_error`: this parameter is used to adjust how precisely the macronutrients must match when querying the database. The parameter is a decimal number between 0 and 1 where 1 is 100% error and 0 is 0% error.
- `tags`: This defines a list of tags that the recipes being found must match.
- `ingredients`: This defines a list of ingredients that the recipes being found must have.

The parameters of this function can get the recipes from the database from a simple interface.

The inner workings of `recipe_by_query` is defined as follows:

```
1  function recipe_by_query(...) do
2      calories_min, calories_max = minMax(calories, energy_error)
3      # do the same for all macronutrient parameters
4      recipes = database.filter_recipes_where(
5                  recipe.calories <= calories_max,
6                  recipe.calories >= calories_min,
7                  # similar for all other macronutrients
8                  )
9      if tags is not empty:
10         recipes = recipes.filter_tags_where_tag_match_one(tags)
11
12     if ingredients is not empty:
13         recipes = recipes.filter_include_all_ingredients(ingredients)
14     return list(convet_to_schema(recipes))
```

Listing 6.1: Recipe query function

In Listing 6.1, we first find minimum and maximum values of each macronutrient from the `energy_error`. For example, an `energy_error` is 0.2 and the calories are 560. In this case, the minimum is $560 - (560 \cdot 0.2) = 448$, and the maximum is $560 - (560 \cdot 0.2) = 672$, meaning that we now have a range the calories of recipes must be inside. in the case that the query inputs calories that are 0 the range that the recipe must be in are the range $[0, \infty]$. We show this range calculation on line 2.

Once we have calculated all ranges for the macronutrients, we remove all recipes not inside the ranges from the recipe selection. We do this on lines 4 to 8.

If the tags given to the function are not an empty list, we filter the selected recipes again and remove the non-matching recipes from the selection on lines 9 to 12. On lines 14 to 17, we perform the same procedure for the ingredients, excluding recipes that do not include all ingredients from the selection.

The final selection of recipes is converted to recipe schemas and returned as a list.

6.2 MEAL PLAN SERVICE

This section describes our implementation of the meal plan service, with a focus on the `generate_meal_plan` function. We do not cover the other meal plan functions since the `generate_meal_plan` function has the most prevalent implementation.

6.2.1 GENERATE MEAL PLAN

The `generate_meal_plan` function takes three parameters, as shown in Listing 6.2. We use the `user_id` to ensure the meal plan gets created for the correct user. The `targets` are the user-specified daily calorie and macronutrient targets. `split_days` is a list that details the number of days the meal plan will span and how the user wants to split their targets throughout the day. An example of a `split_days` list containing information for two days could be

$[0, 0.4, 0.6, 0.2, 0.3, 0.5]$. Each day has three entries in the list, corresponding to the three meals available for each day. The first entry 0 corresponds to breakfast on the first day. A 0 means the user has allocated none of its targeted daily calories to that meal. A 1 in the `split_days` means a user has allocated 100% of their daily calories and macronutrients to the meal.

To calculate the desired calories and macro-nutrients from the `split_days` list, we have defined a helper function called `calculate_split`, which takes in the meal split and the calorie and macronutrient targets. Then, we return the macronutrients needed for the meal. We calculate how much of the user's daily macronutrients and calories are available for each meal by $daily\_macronutrient \cdot day\_split$.

Each day in a meal plan contains at most three meals: breakfast, lunch, and dinner. To get the meals for each day of a desired meal plan, we use the `generate_meal_plan` function. To ensure the recipes correspond to the correct day.

The function `generate_meal_plan` is called each time a user makes a `POST` request on the endpoint `/generate` on the API gateway. The function takes a user id (`user_id`), the user's macronutrient targets (`targets`), and the list of daily splits `split_days`. On line 2, the `energy_error` is set to 0.1 to allow for some deviance from the user's actual goal. If we do not use an energy error, we risk that no meal plan suits the user's targets.

On lines 3 and 4, we initialize an empty dictionary for the total macronutrient information and a list of recipes corresponding to the days the user has specified in the `split_days`.

The for loop on line 7 loops over each breakfast, lunch, and dinner split, creating an empty dictionary for the daily recipes on line 8. We calculate the `breakfast_parameters` in the before mentioned `calculate_split`, which corresponds to the macronutrient parameters defined in section 6.1. We perform this process for all meals in the meal plan.

Then, on lines 12 to 17, we fetch random recipes from the recipe service with the breakfast, lunch, and dinner parameters combined with the `energy_error` and a tag for each meal type.

Once we have calculated all the daily splits and fetched all the recipes, we create a response dictionary with total macronutrients and daily recipes. Lastly, we insert the created dictionary into the meal plan database.

```
1  function generate_meal_plan(user_id, targets, split_days) do
2    energy_error = 0.1
3    total_macronutrients = empty dictionary
4    day_mealplan = empty list
5
6
7    for (breakfast_split, lunch_split, dinner_split) in split_days do
8      recipes = empty dictionary
9      breakfast_parameters = calculate_split(breakfast_split, targets)
10     # do similar for lunch and dinner split
11
12         breakfast_recipe = random_recipe_by_query(
13                 breakfast_parameters,
14                 energy_error,
15                 tags=["Breakfast"],
16                 ingredients=[]
17                 )
18     # fetch lunch and dinner recipes with same method as above
```

```

    recipes["Breakfast"] = breakfast_recipe
    # same approach for other recipes types

    for recipe in (breakfast_recipe, lunch_recipe, dinner_recipe) do
      total_macronutrients["calories"] += breakfast_recipe["calories"]
      # same for all other macronutrients

    day_mealplan.append(recipes)

  response = empty dictionary
  response["total_macronutrients"] = total_macronutrients
  response["days"] = day_mealplan

  insert_mealplan_into_database(user_id, response)

  return response
```
Listing 6.2: Generation of meal plan

6.3 USER SERVICE

The user service is a central part of the application. While each service is an independent program with its own database, we still need to provide user-specific data. We still do this by storing the user id in the database, but only the user service can manage the user database.

The central functionality of the user service is validating users, which is done by asking the user service for a username and password pair, which it then compares to the one in the database. We encrypt the password in the database with bcrypt.

The bcrypt algorithm is designed to be slow when encrypting text, which will make it harder for attackers to brute force the stored passwords even if they get a database dump[37]. An encrypted password using bcrypt looks like this:

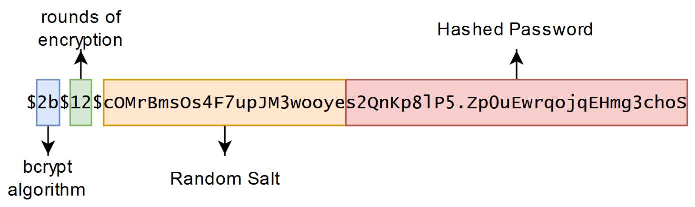


Figure 6.1: bcrypt hash composition[37]

All the information used to encrypt the password is stored directly in the password string, which can be seen in Figure 6.1. The bcrypt algorithm can be used with different amounts of encryption passes, which increases the computational time that an attacker has to use to crack the password. In this example, the amount of passes that are required to produce the hashed password is $2^{12} = 4096$. This makes it harder for the attacker to bruteforce the passwords. But for when we need to check if the password provided by the user is the correct one bcrypt takes the algorithm, and encryption rounds, and the random salt, encrypts the provided password and performs a byte comparison between the stored password and the newly hashed password.

The bcrypt algorithm was specifically designed to be used in a time when the computational power of computers is always increasing. The designers made it possible to use different amounts of encryption rounds, so the algorithm is a viable solution to password storage, even as the computational power increases[37].

6.4 API GATEWAY

The API gateway is implemented as a Web API where it performs authentication on requests from the client and redirects the request to the correct microservice.

The API gateway is composed of three different components, which are responsible for the encoding of JWT tokens, the communication with microservices, and exposing the API, respectively. The API gateway also holds all the request and response schemas from the microservices.

6.4.1 SERVICE COMMUNICATION

For the API gateway to communicate with microservices easily, a wrapper class around an HTTP library has been implemented, which can be replaced with mocks or other implementations of the service class. This means that the API gateway takes in service interfaces and is not concerned with the implementation. It also uses generics for the response types of the service being communicated with.

```
function request(method, url, res_model, res_type, data) do
  res = HTTPclient.send(method, url, data)

  if res.status_code in range(400, 599) do
    details = error_details()
    # check if there is error details,
    # if not, make generic error details
    send_HTTP_error(res.status_code, details)
    return

  switch(res_type) do
        case LIST do return res_model(unpack_list(data))
        case DICTIONARY do return res_model(unpack_dict(res.data))
        case PRIMITIVE do return res_model(res.data)

  return None
```

Listing 6.3: Service request function

Listing 6.3 shows pseudocode for the service class which handles all general cases that the services might send.

The `request` function takes in which HTTP method should be used on the microservice and the URL it should send the request to. Then it takes `res_model` which is the response model, which can be any type. It also takes the `res_type` which is the method of unpacking the request method should use when creating the `res_model`. If the request requires a message body, the `request` method also has an optional parameter `data` of type string which is used as the body.

On line 2 it sends the request to the service and receives the response. Then, if the response is an HTTP error message, the function handles it on line 4 by trying to get the HTTP error detail message, and if there is none, create a generic HTTP error message for the client. This part is implemented so that we do not need to explicitly handle any errors the services might raise.

Otherwise, if all is well, it converts the type to the `res_model`. Depending on the programming language this implementation is made in there might be more unpacking types. The `unpack` methods on line 12 to 14 are responsible for unpacking the data so that the constructor of `res_model` can interpret them correctly.

6.4.2 JWT ENCODING AND DECODING

The authentication process of users is complicated and is prone to error if it is implemented by ourselves, therefore we use a library to encode and decode the JWT tokens. In the pseudo-code, we refer to it as `jwt`.

```
1 function encode(username, id, expire, secret, algorithm) do
2   return jwt.encode(username, id, expire, secret, algorithm)
3
4 function decode(token, secret, algorithm) do
5   payload = jwt.decode(token, secret, algorithm)
6   if payload is corrupt or expired do
7     return None
8   return payload
```

Listing 6.4: JWT encoding/decode process

The method `encode` of Listing 6.4 on line 1 takes the username and id of the user and an expiration date. Then we use the JWT library to encode the users info into a token with a secret and an algorithm. The secret is a random string that is only known to us, and the algorithm is a JWT-supported algorithm, which in most cases is `HS256`. We package both the username and id in the token so the client never has to send the username or id. The `decode` method on line 4 takes in the token and decodes the token into a payload with our secret and specified algorithm. It then checks if the payload is corrupt or expired and if it is, return `None`. The payload corruption usually happens if the user tries to modify the token. The API gateway can then use the payload.

6.4.3 API IMPLEMENTATION

The API gateway is implemented using data schemas that define how the data should be structured. This structure automatically declines any malformed data and sends an error to the client. The actual implementation of the API gateway uses the FastAPI and Pydantic packages from Python to handle the validation of user inputs. The schemas have four different variants; namely base, create, update, and info schemas. These variants ensure that the users can modify their data at will. The schemas have first been defined in the microservices and then the API gateway uses them to correctly communicate with the microservices.

BASE SCHEMAS

Base schemas are responsible for having the absolutely bare minimum of fields for a resource. The base health schema is as follows:

```
schema BaseHealthEntry
    datestamp: datetime
    height: float #Optional
    weight: float #Optional
    fatPercentage: float #Optional
    musclePercentage: float #Optional
    waterPercentage: float #Optional
```

Listing 6.5: Base health schema

Here are the minimum fields a request including a health entry must have. This schema is then used to extend the other schema types for health entry modifications.

CREATE SCHEMAS

Create schemas that hold information about the owner of the resource along with the base schema information.

```
schema CreateHealthEntry extends BaseHealthEntry
    userID: int
```

Listing 6.6: Create Healths schema

The only extra information required for a health entry to be created is the user's id, which is added when the request reaches the API gateway as mentioned in subsection 5.2.8. The user id is used to mitigate unauthorized modification and creation of resources owned by other users.

UPDATE SCHEMAS

Update schemas are similar to the create schemas as they also hold information about the user but they might have different fields, as some fields in the schema might be constants. For example, when updating a user, the base schema will hold an entry for birth date, but the updated schema will not, as a user's birth date is constant. All health entry fields are variables and can change, therefore the update schema is just an extension of the `CreateHealthEntry`.

INFO SCHEMAS

Info schemas are schemas that are sent to the client, which then would be used to display the data in the schema on the frontend. They are usually an extension of the create schema and sometimes have nested schemas. The info schemas are also just named after the resource they represent, so the info health entry schema's implementation name is `HealthEntry`.

```
schema HealthEntry  extends CreateHealthEntry
    id: int
```

Listing 6.7: Create Healths schema

The reason that the `HealthEntry` is an extension of the `CreateHealthEntry`, is that the `CreateHealthEntry` extends the base class and also includes fields that are needed in the Info schema. The id is included for debugging.

6.5 FRONTEND IMPLEMENTATION

In this section, we will describe the implementation and content of the application's frontend based on the wireframes described in section 5.3.

FRONTEND FRAMEWORK

We have chosen to use React to create the frontend of our application. React is a free and open-source JavaScript framework for building user interfaces based on components[38]. We feel React lends itself well to our project, as our different services are representable as React components. React components are pieces of code that are independent and reusable. A component works like a JavaScript function but only returns HTML[39]. Each React component works separately, enabling us to change parts of the application without updating everything.

SERVICE INTERFACES

For every service, we define an interface to the required functions we need to implement for using the service. For example, for the inventory service, the interface `IInventoryService` is structured as seen in Listing 6.8. There are five functions that, when given a concrete implementation for them, will send a request to the API gateway.

```
interface IInventoryService
    function GetAllForUser(userId: number) returns list of Inventory
    function Post(userId: number, name: string) returns status_code
    function PostToInv(invId: number, foodId: number, expirationDate:
    string) returns status_code
    function DeleteItem(invId: number, itemId: number) returns
    status_code
    function DeleteInv(inv: Inventory) returns status_code
```

Listing 6.8: Inventory service interface

Using an interface rather than a concrete implementation for an argument in a function or class constructor means that the function or class is not dependent on the specific implementation. When we use an interface, the functions and classes are only dependent on the inputs and outputs of the interface. Using interfaces lets us implement the functions to make calls to an internal mock API when testing.

The `IInventoryService` interface in the Listing 6.8 defines functions for communicating with all inventory endpoints on the API gateway, which we described in subsection 5.2.8. The concrete implementation of the interface functions maps to the following endpoints:

- `GetAllForUser`: Maps to **GET** `/inventories`
- `Post`: Maps to **POST** `/inventories`
- `PostToInv`: Maps to **POST** `/inventories/{inv_id}`
- `DeleteItem`: Maps to **DELETE** `/inventories/{inv_id}/{item_id}`
- `DeleteInv`: Maps to **DELETE** `/inventories/{inv_id}`

AUTHENTICATION

We send an authentication token with the header as a JWT that is then stored in the web browser's local storage when a user logs in. The token gets deleted from the local storage when logging out. The functions seen in Listing 6.9 are used for token handling.

```
function GetJwt() do
    return localStorage.getItem("jwt") or empty string

function setJwt(jwt) do
    localStorage.setItem("jwt", jwt)

function getDefaultHeader() do
    return Headers({"Authorization": "Bearer" + GetJwt()})

function ClearJwt() do
    localStorage.removeItem("jwt")
```

Listing 6.9: JwtService class

We also need a token for authentication purposes before sending an API request. The function `getDefaultHeader` is then used to get the JWT token from the local storage if it exists. Otherwise, it would just return an empty string. In Listing 6.10, the pseudo-code for the function `DeleteItem` is shown, where we use the `getDefaultHeader` function and send the result with the request as seen on line 5.

```
1  function DeleteItem(invId, itemId) do
2      headers = getDefaultHeader()
3      return fetch(baseUrl/inventories/{invId}/{itemId}
4          method: "Delete"
5          headers: headers
6          )
```

Listing 6.10: DeleteItem method

6.5.1 FRONTEND CONTENT

In this subsection, we describe the design and content of the application's frontend.

SIGN UP

At the bottom of Figure 6.3 we show the option for creating a new account. Pressing the ”Sign up” link redirects the user to the signup page.

After completing all required user information for creating the account, clicking the ”next” button will direct you to the ”energy targets” page where the user can enter their target for calories, protein, carbohydrates, and fat, as seen in Figure 6.2. Pressing the ”Sign up” button then creates the new account. If successful, we redirect the user to the login page, where they log in with the newly created credentials.

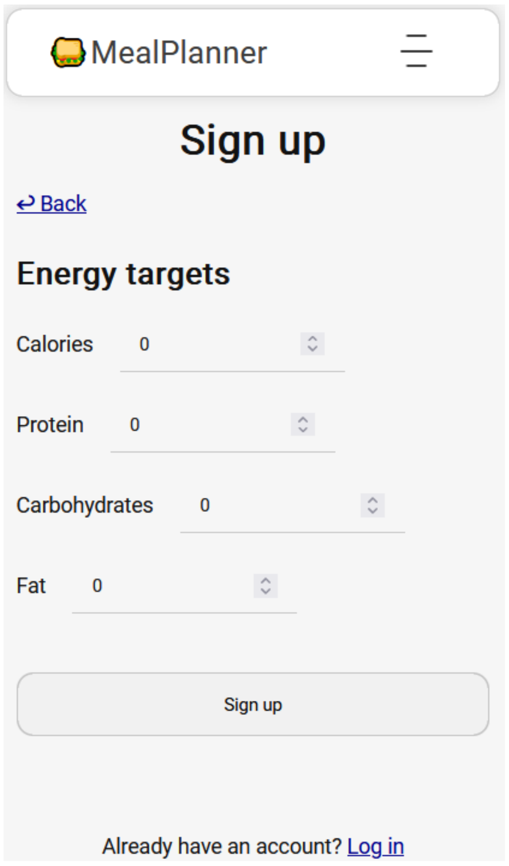


Figure 6.2: Energy targets page

LOGIN

We only allow authenticated users to use the application. To achieve this, we have implemented a simple login page. When a user enters incorrect credentials, we display an informative message in red, indicating something went wrong, as shown in Figure 6.3.

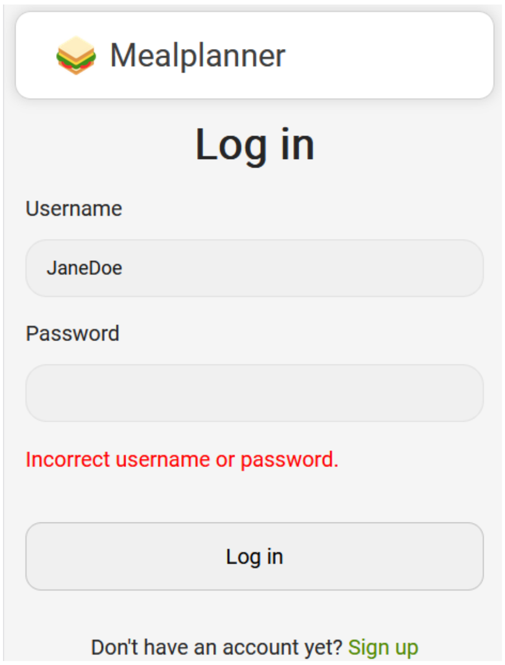

Mealplanner
Log in
Username
JaneDoe
Password
Incorrect username or password.
Log in
Don't have an account yet? Sign up

Figure 6.3: Login view when entering wrong login credentials

SIDEBAR

The application interface has a sidebar that contains four different elements to choose from. Clicking on one of the sidebar elements navigates to that page. We have made the sidebar responsive for a better user experience on all screen sizes. Figure Figure 6.5 and Figure 6.4 show how the sidebar adapts to different screen sizes. The four sidebar elements represent the various pages in our application, some of which we describe in the subsections below.

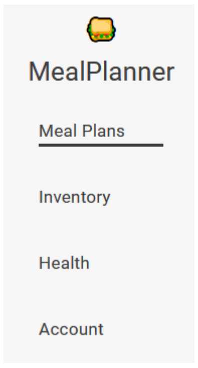


Figure 6.4: Sidebar in desktop view

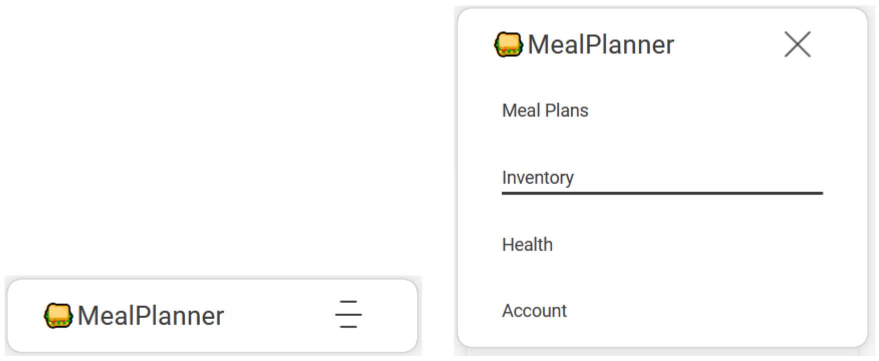
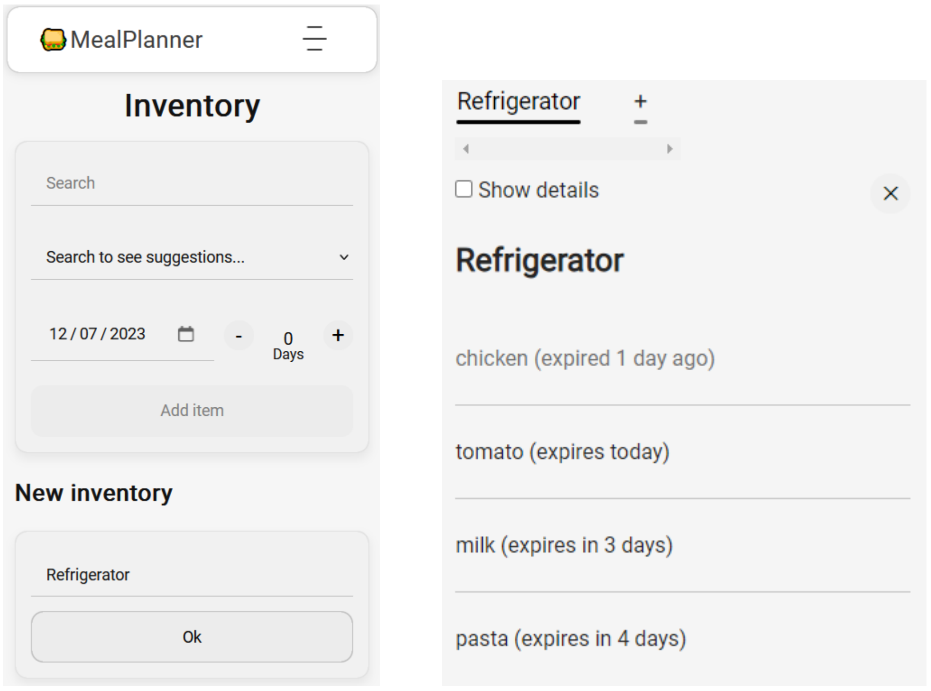


Figure 6.5: Sidebar in mobile view

INVENTORY

The inventory page provides users with an overview of the contents of their inventories.

When new users add items to their inventory, they first need to create a new inventory. Users can assign a name to the inventory, allowing them to categorize inventories with labels such as ”Refrigerator” and ”Pantry” for different food types.



(a) New inventory (b) Inventory items

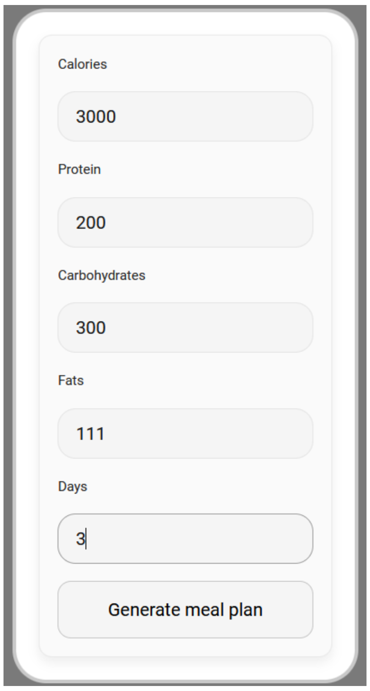
Figure 6.6: Inventory page

Adding items to an inventory is done by entering the item name in the search field and selecting the specific item from a dropdown menu. Users can also set an expiration date before clicking the ”Add item” button.

As shown in Figure 6.6, clicking on the inventory name reveals the items it contains, displaying their names and expiration dates sorted by the expiration date. Once an item exceeds its expiration date, it becomes less visible to the user, indicating that it has expired. The user can see more details about the item, such as fat, protein, and carbohydrates, by pressing the ”details” checkbox.

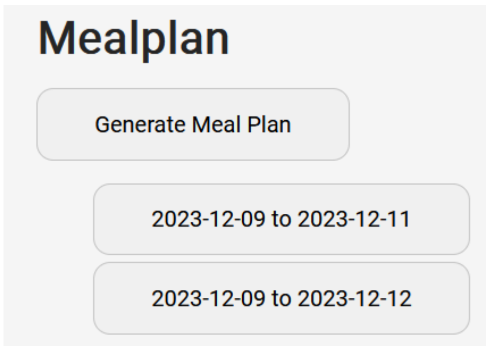
MEAL PLANS

The ”meal plans” page grants users an overview of their meal plans. In addition, it also gives the ability to generate a new meal plan. When a user wants to create a meal plan, the user must input the calories, macronutrients, and the number of days the meal plan should span. We show this process in Figure 6.7(a).



Calories	3000
Protein	200
Carbohydrates	300
Fats	111
Days	3

(a) Generate mealplan



2023-12-09 to 2023-12-11
2023-12-09 to 2023-12-12

(b) Mealplan list

Figure 6.7: Meal plan page

We show all the meal plans assigned to the user in a list-style format, where every row is a button with the text ”from the start date to end date” as shown in Figure 6.7(b). Upon choosing a meal plan, a window appears with the option to select a day or delete the specific meal plan. When the user chooses a day, we show the day’s total calories and macronutrients and detailed information about the recipes, such as the ingredients and instructions.

HEALTH

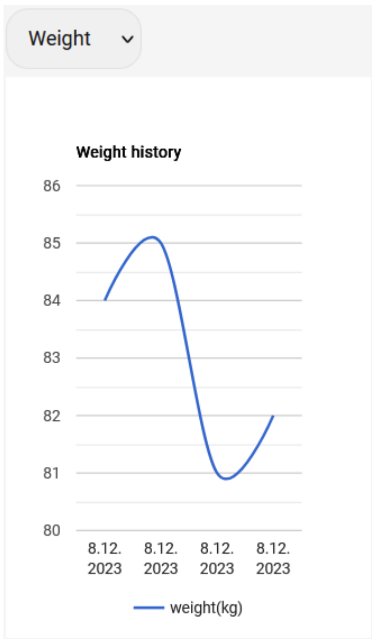


Figure 6.8: Health graph showing a user's weight history

We provide an overview of the user's weight, height, fat, muscle, and water percentage over time on the health page. Here, we use a drop-down menu to select the type of health information the graph should display. The time-series graph, illustrated in Figure 6.8, visually represents these health entries. Users also have the option to add or delete a specific health entry for a given date.

7. TEST AND EVALUATION

7.1 UNIT TESTING

All the services have implemented unit testing for the internal functions to verify that they work as intended. This has been done in a black box fashion, where we only focus on getting the expected output from the input and verifying that this is correct.

The services that we have implemented are simple and can be thought of as an interface to the databases they serve. This means that, when unit testing, we test the outcome from the databases. The tests have run using a test database using a flat SQLite database that is created and deleted for each unit test. We do this to have identical entries in the database to ensure the results are always the same.

7.2 INTEGRATION TESTING

To ensure that our services work as expected, we have also created integration tests for each service that test the service endpoints. This is done by creating an instance of the service in a local environment and a test client that acts as an HTTP client, which then sends requests to the service. The service does not know that this client is a test client, which results in tests that are as close to creating real requests from a client as possible.

We test the service's response on expected HTTP responses and content. We also test them on invalid input to see if they respond with the correct error messages.

7.3 CONTINUOUS INTEGRATION

Each service has its own repository on GitHub, and here we have used GitHub actions to automatically run the previously mentioned tests whenever someone commits to the repositories. These actions ensure that the code is working as intended. It also ensures the commits that might have broken the code are easily located.

The purpose of the described tests was to ensure the correctness of our code, both in isolated and integrated contexts. Our testing phase helped ensure correctness and revealed several minor issues in our code, which we then corrected.

7.4 USABILITY TEST

In this section, we cover our usability test. The goal of the test is to identify possible usability problems that could be used to refine the design of the application. The first step was to define some tasks for a participant to complete. Finding the tasks was done by listing all possible features that can be performed on the different pages of our application, with a focus on the most important ones. Those tasks are shown in the list below:

- Make a new account and log in.
- Use the inventory page.
  - Create a new inventory.
  - Add a few ingredients with different expiration dates.
  - See the macronutrients of the added ingredients.
  - Create another inventory and delete it afterward.
- Create a meal plan and review the newly created meal plan.
- Insert a health entry and view the data on the graph.
- Sign out

The second step was to find the participant who should complete the predefined tasks. The participant chosen was an extern with no influence on the application development. The participant is also considered to be in our target group and has an above-average technical knowledge. Throughout the usability test, the participant was encouraged to think aloud to enable continuous feedback. Due to time constraints, we only performed this usability test.

The result of the usability test is feedback on the application and observations about the participant's behavior, which can be used to detect usability problems and gain an overall view of what works well. The detected usability problems are assigned into three categories[40] describing how severe the problem is:

- Critical: The user is unable to continue without intervention from the test moderator.
- Serious: The user experiences a delay of several minutes and causes significant irritation.
- Cosmetic: The user experiences a small delay but it does not have a big impact on the user experience.

#	Usability problem	Category
1	Confused about why the generated meal plan was not visible	Critical
2	Confused about why he could not see new health entries	Critical
3	During the usability test, generating a meal plan or adding a health entry has no visible indication that the action has been performed	Critical
4	Did not know their energy targets	Serious
5	Confused about why input fields on the health page are not optional	Serious
6	Confused about the search of inventory page	Serious
7	Clicked the wrong button to add new items to inventory	Cosmetic
8	Confused about why the inputs in the meal plan popup were prefixed with a zero	Cosmetic
9	Confused about why they have to enter energy targets again when generating a meal plan	Cosmetic
10	Could not find the sign out button	Cosmetic
11	Wondering why the expiration date field on the inventory page does not reset	Cosmetic
12	Noticed how some input fields allowed negative numbers	Cosmetic

Table 7.1: Usability problems

We will now move on to a more detailed description of the usability problems shown in Table 7.1

1: When the participant was tasked with generating a new meal plan, the UI did not update automatically after submitting the form. This caused confusion about whether the meal plan was successfully created and made the participant submit the form twice, thus creating two meal plans. Only when we suggested refreshing the page could the participant continue with the task.

2: When the participant was asked to add a new health entry, the Health page did not update automatically after submitting the form, where it was supposed to make some changes to the graph. This confused the participant about whether the new health entry was added to the graph or not. As this problem was similar to **1** the participant tried to update the page manually which led to the new health entry added to the graph.

3: The participant thought that the website froze when he attempted to generate a meal plan and add a health entry. The participant mentioned how he expected some indication that the action had been successfully applied.

4: This problem appeared during the signup process and caused a significant delay for the participant. The participant did not know what a reasonable number of daily calories, protein, etc. is. To continue the task, we provided some numbers to use, but because the participant could figure this out on his own with some research, this problem is categorized as serious and not critical.

5: When the participant wanted to submit a health entry, he was confused as to why every field was mandatory. Given that he did not know every metric, he had difficulties filling out the health entry fields. The participant noted that it would be beneficial to leave some fields optional, to accommodate users, that do not know every metric that is present on the health page.

6: Entering ”mælk” in the search field on the inventory page produced no matches because the names of items are very specific such as ”skummetmælk 0.1%” or ”sødmælk 3.5%”.

7: The participant was momentarily confused about which ”add” button adds a new item to the inventory. Initially, he clicked on the ”+” button which created a new inventory, but quickly figured out the correct one.

8: The participant noted that the input fields on the meal plan popup were prefixed with a 0. It did not affect any functionality and is purely a cosmetic error.

9: The participant disliked that he had to enter energy targets again when creating a meal plan.

10: When the participant was asked to sign out from the application he got confused about where the sign-out button was located, but in the end, he managed to find it on the account page.

11: Given that the search input field on the inventory page resets after adding a new item, the participant expected the expiration date input field to do the same.

12: The participant discovered that some input fields allowed negative numbers. It caused no errors, but the graph on the health page went into the negatives.

From the above observations, it is evident that our frontend is not up to our standards regarding a production ready application. The majority of the issues that the user experienced were related to the application’s lack of feedback whenever the user performed a task. The design of the graphical user interface made the user confused about which actions the buttons performed. This means that our design also needs more iterations to get the most intuitive UI and the best user experience. All these issues have created new tasks, but due to our limited time, we have to list them as future work.

8. DISCUSSION

In this chapter, we discuss various aspects of our project. We discuss what went well and which aspects of the project were unsatisfactory.

8.1 PROJECT MANAGEMENT

In this section, we discuss the project management guidelines described in section 4.3, which guidelines we adhered to, which we did not, and the outcome of our guideline decisions.

During our decision-making process, we partially adhered to our guidelines. We conducted extensive research before finalizing decisions, yet the depth of critical analysis was insufficient, resulting in some suboptimal choices. In particular, choosing to work with microservices initially looked like a good choice but ultimately turned out to be a poor decision. If we had engaged in more critical thought, we may have made a different decision, which is discussed further in section 8.2.

For time management, we utilized the backlog feature on GitHub for both the report and the application, facilitating a clear overview of pending tasks, the distribution of work, and the identification of report sections ready for review. While we utilized the backlog, we did not always specify time frames for each backlog item, resulting in some items taking longer to finish than anticipated.

Regarding team organization, we initially formed smaller, functional groups. These groups were intended to distribute tasks and provide a support system for questions. Group composition was based on shared course schedules to enable collaborative work on the program. However, the effectiveness of these groups varied. This was due to both communication issues across the different groups and the periodic absence of group members. We have previously encountered issues with group members failing to show up without providing any reasons. This has resulted in instances where only a single team member is present on certain days, consequently postponing work and reducing team communication. If work is continually delayed, it pushes other features further down the development pipeline, resulting in some features not being implemented before the deadline. The issue could have been mitigated if we had spent more time during the design phase of our application. Here, we should have defined well-formed interfaces and inputs and outputs of the various components of our application. This would have helped us maintain a clear vision of the product and know the format to work with and base our functions around. We believe this, alongside the decision to use a microservice architecture, has had a major role in the issues we faced during the development process.

We initially wanted to use Scrum principles to manage group communication and divide our work into sprints. While we agreed to adopt Scrum, ultimately, we did not adopt Scrum as part of our project management. We believe this is partially due to the staggered work schedule of the group, resulting in us being unable to set specific times each week for Scrum- and daily stand-up meetings. In previous semesters, we have had success with Scrum processes. We believe we would have had a better group communication structure if we had adopted the processes during this semester. In hindsight, implementing Scrum would have been better for knowing which tasks were being worked on and by whom.

Given that we chose to divide the group into smaller sub-teams, some of these teams adopted pair programming. As some group members worked alone, it was not feasible to adopt group-wide pair programming. For the teams that adopted pair programming, it proved effective in producing code and helped expedite the problem-solving process.

While we divided our group into different teams, we gathered in the group room a few times each week because group members had chosen divergent elective courses. We held weekly meetings where we discussed and shared the progress of the different components of the application and the report. As our group had different elective courses, we believed working in the same room, when possible, was the best course of action. As the elective courses wrapped up, we continued to work in the same room to facilitate communication.

8.2 ARCTITECTURE

When designing our application architecture, we were very focused on the vertical scalability of our application, which may have hindered the development of the first version of our program.

8.2.1 DECISION

We chose the microservice architecture for our program because it introduces loosely coupled and separate codebases. Our rationale was that we could divide ourselves into small teams and work on each microservice in a language of choice decided within these teams.

At the beginning of the development process, we experienced success with the architecture since we could split the group depending on what courses each group member had to attend. This strategy ensured that we always had someone else to work and communicate with. However, this also meant that we had limited time where the whole group was present, making it difficult for the teams to efficiently communicate and figure out how the different services should work together.

When we reached the point where the services were up and running individually, we realized that we had not defined a good and stable interface for each service to communicate with each other. Combined with the limited time together as a team, this resulted in work frequently getting postponed. For example, the health service was still under development when the work on the frontend began, resulting in delayed work on the health aspect of the frontend.

8.2.2 COMPLEXITY

The application complexity also increased when we needed to have multiple microservices running simultaneously, as this was difficult to configure and manage manually. We began the development of the API gateway and a complete project setup using Docker and Docker Compose, intending to use the container orchestration tool called Docker Swarm to manage and load balance many instances of the services. During this process, we had to change many minor things on each service for them to be compatible with the Docker Compose setup. The purpose of the project setup was to have a test environment with Docker Compose and use Docker Swarm for production. In the test environment, we had a single MySQL database instance talking to each service instead of having an independent database for each service. This also resulted in issues with services having to be adjusted slightly to be usable with the database.

All the small changes to each service, both for the project setup and the necessary changes to the service database integrations, added a sizeable overhead for us as each of our services had its git repository to not conflict with the work of the other teams. This resulted in us managing many repositories, which was not ideal given the timeframe of our project.

8.2.3 LAYERED APPROACH

The benefits of having multiple codebases and communication between each other began to fade away when the implementation of the service architecture became greater than the feature implementation, which happened when we had to implement service communication and the containerization of each service.

The layered architecture might have been a better approach in the timespan we had to work with. Here, we would have had only one codebase each for the frontend and backend to work with and a single database instance. This would be the most manageable architecture choice for a layered architecture with only three layers. Here, we may have had a better overview of the application as the whole API would be in a single repository.

Setting up two repositories and creating two application projects would have been faster than having to do it many times with the microservices. This would have been a better approach in the early stages of development to implement the necessary features of our program. Here, we would also have been able to take advantage of relational databases instead of having to manage relations over many databases. The layered approach would also give us a greater overview, as the interfaces and implementation details would have been less scattered across multiple code repositories.

While the layered approach is often deemed slower in deployment and scalability, it is easy to set up and begin the development fast, whereas the microservices approach is a slow beginning, with the tedious setup with multiple projects and communications between them. However, it is faster in feature development once the infrastructure is built. This architecture would have allowed us to increase the development iterations greatly. The scalability would not be as good as in the microservices architecture, but looking back, we should have focused on the features instead of the scalability.

8.3 INDIVIDUAL EXPERIENCE

With a project group consisting of seven team members, each with their level of experience, it would have been beneficial to consider the expertise of each member before beginning the development process. The experience level and forte of each team member vary. With a large project, it may have been possible to assign tasks and split the development team into smaller sub-teams based on the expertise of the team members.

As it is, some team members were stuck on assigned tasks for longer than expected. This, in turn, caused the development process to slow down significantly and made us miss set deadlines for services being finished. We might have been able to mitigate this issue by assigning tasks based on expertise and experience.

During the development process, we initially split the group based on the elective courses of the members. The reasoning was that this was the only way for the sub-teams to have the same working hours, thus optimizing communication within the sub-teams. While this did not consider the experience level of each member, we believe our approach was the most optimal while we still had courses. As the elective courses wrapped up, we would have been able to redistribute our workforce based on the experience of each team member. However, we did not take this approach and continued to work in the groups established at the beginning of the development process. As some of the functionalities were either finished or nearing completion, we believe reallocating group members would be more of a hindrance than an advantage.

8.4 FRONTEND IMPLEMENTATION

The frontend development is crucial to make the product usable by our target group. In this section, we will discuss the positive and negative outcomes of the development of the frontend.

The UI design of the website is lackluster and not very appealing, and the User Experience (UX) design is not optimal for easy navigation. Our initial expectations of the ease of development with React and TypeScript were that the frontend implementation would not be very complex. However, due to our lack of knowledge of structuring and designing websites with React and TypeScript, we ended up with an unfulfilling UI and UX. The styling was especially troubling for us when styling the placement of elements on the website. Furthermore, the absence of design guidelines created inconsistencies in the UI, making it difficult to interact with the application. It caused less reuse of CSS classes, which led to slower development.

The UX would have been greatly improved if we had implemented indications whenever the user performs actions like generating a meal plan. This was particularly visible in our usability test, where the participant initially thought the meal plan was not generated. It was also visible on the health page, where the graph does not update before refreshing the page. The inventory page did not have these issues. However, the participant was confused about how to add new items to the inventory page, which is also a big issue.

Throughout the development, we had a mobile-first approach. Unfortunately, while we emphasized optimizing for smaller screens, we neglected the layout on larger screens. The application works on larger screens but could use the available space more efficiently.

However, the general functionality of the website in regards to displaying the user data from the services was a success.

8.5 REQUIREMENTS

This section discusses our requirement specification, which requirements we implemented, and which we did not.

We have fully implemented most of our must-have requirements. However, we have not entirely implemented the meal plan generator. Currently, the meal plan generator is not linked to the user inventory or the users' dietary data. In our current implementation, users manually write targeted daily calories and macronutrients when generating a meal plan. While this works, it does not satisfy our must-have requirement of being able to generate a meal plan with both the user inventory and their dietary data taken into consideration. As stated in section 4.2, our must-have requirements describe our MVP. Considering that we have not fully implemented our meal plan generator, we can not state that we have successfully satisfied our MVP.

The reason for the meal plan generator not being implemented as stated in our must-have requirements, is partly due to issues with communication between the different services. For the meal plan generator to be successfully implemented, we need communication between the meal plan service, the user service, and the inventory service. Due to pushed deadlines and issues with getting all services up and running, we ended up deciding to let the meal plan service run independently of the user- and inventory services. Without the ingredient inventory being used in the generation of meal plans, we do not successfully tackle the issue of food waste.

We have only fully implemented one of our should-have requirements. We show macronutrient contents for the recipes in the generated meal plans. This partially tackles the issue of healthy eating by giving users an overview of how many calories and macronutrients are in the meal plan. However, it was intended to work in tandem with the food consumption statistics. Had we implemented both requirements, we believe they would have given users a clearer picture of their consumption over time. However, we feel we successfully tackled the issue of healthy eating, given that we have implemented a meal plan generator that generates meal plans based on user-inputted caloric goals and then shows the macronutrient contents of each recipe to the user.

When we initially created our requirement specification, we had high ambitions, which ultimately turned out to be too great. We underestimated the size of various tasks throughout the project, including our requirements. Choosing to work with a microservice architecture also resulted in an added learning curve, slowing the development time of several of our requirements.

9. CONCLUSION

In this project, we set out to combat the issues of food waste and healthy eating. During our analysis in chapter 3, we found that both problems are widespread and significant. Our analysis led us to investigate existing solutions, and we found that none of them solved the specific issue we wanted to solve. This resulted in our problem statement:

We want to develop an application that generates personalized meal plans for users based on personal dietary needs and available ingredients to promote healthy eating and minimize food waste.

Based on our problem statement, we created a requirement specification using MoSCoW. These requirements described our MVP and further improvements that would satisfy our problem statement. Afterward, we initiated a design phase, starting with a choice of architecture. Throughout this process, we looked at the benefits and drawbacks of different architectures, ultimately resulting in us choosing to adopt a microservice architecture. Following this, we defined our components and designed frontend wireframes.

Our application only implements some of our must-have requirements. The application creates meal plans based on user inputs of a desired number of days, calories, and macronutrients. We achieve this by having users manually fill in the required information. The current implementation does not meet our requirement of generating meal plans based on users' personal dietary data and food inventory.

The reasons for failing to implement our must-have requirements are poor decisions and subpar time management. The time management suffered partly due to decisions made regarding our choice of program architecture.

Based on the issues described and the missing requirements, we conclude that we have not solved our problem statement to an acceptable degree.

10. FUTURE WORK

This chapter details our suggested future work for the project. The chapter describes the areas in which our application could be improved and how we propose the improvements be made. The improvements are based on our MoSCoW requirements and our findings throughout the implementation and testing process.

10.1 MEAL PLAN SERVICE

The current implementation of the meal plan service can generate meal plans based on user inputs on the frontend of the application. The inputs are limited to desired calorie- and macronutrient targets and a desired amount of days. While this works, the functionality is minimal, and there are certain features we would like to have added or changed if more time was available.

Currently, a user is able to generate mealplans with three meals per day, limited to only breakfast, lunch, and dinner. The backend picks a split of 33% for all macronutrients instead of letting the user input the split themselves. Future iterations of this service should add an input box for the split values and allow the user to pick between more types of meals, such as snacks. This could also allow users to choose more meals per day than the current limit of three meals.

With added options for choosing daily meals, it would also be beneficial for users to have more control over their desired macronutrient split. Currently, the users pick a complete split of their total daily energy target. This means that a user can pick a split such as 0.4 of their total energy target for a given meal, resulting in the meal planner splitting all targets, meaning calories, protein, fat, and carbohydrates. The drawback is that recipes are forced to fit a specific macronutrient split, resulting in many recipes being unfit for any user generated meal plan.

To improve on this feature, the users should be able to choose both their desired calorie split per meal and their splits for protein, carbohydrates, and fat.

Another feature that would add to the functionality of the meal plan service is the ability for users to add tags to their meals when generating. Tags such as ”vegan” and ”halal” would allow for more flexible dieting, help adhere to specific dietary constraints, and make our application more inclusive for users with allergies and other dietary restrictions. We would also like to add a way for users to give their generated meal plans a name to differentiate between their various meal plans. These features, alongside the previously mentioned features, would implement our should have requirement of a more user-defined meal plan.

The ”generate meal plan” popup currently requires the user to input the desired calorie and macronutrient amount for the meal plan. However, during user creation, the user already specifies their desired daily calories and macronutrients. In future iterations, the meal plan service should fetch these values from the user service instead, which would save the user time and effort and avoid unnecessary repeats of already specified parameters.

Currently, we do not take the user inventory into account or use their target macronutrients from their account. Getting their inventory from the inventory service would greatly improve the meal plan service by further targeting the issue of food waste. Therefore, this would be a high priority addition to our product in future iterations.

10.2 API GATEWAY

The API gateway currently lacks support for updates on certain endpoints, such as the user, health, and inventory endpoints. This makes the users unable to update entries that they have made. To update their inventories and health points are required to delete the entries and inventories and recreate them with new information. The services do facilitate these operations, but they have not been implemented in the API gateway and are thus unused. Future iterations of this project should implement updates on the endpoints that are currently missing the feature. This enhancement will significantly improve the user experience when users want to change their inventories and health entries.

10.3 ARCHITECTURE

We mentioned in the discussion that there is a lot of overhead in the microservice architecture and that it hinders feature development. If we were to continue the development of this project, it might be a viable solution to change the architecture to the layered architecture, as our team would not increase enough to facilitate the team responsibility areas for each service.

We think that this would help with the future feature development.

10.4 FRONTEND

During the testing phase of this project, we found many of the issues in our application happened in the UI and UX. The most critical of these problems is based on the user actions causing the viewable information to be outdated. But several ”quality of life” issues were also found.

In the current state of the application, the frontend was primarily developed with the mobile platform in mind. While this is still usable on other devices, it does not adapt the layout to be used efficiently on larger screens. We want to change this for the application to be easier to use on devices other than mobiles.

In addition, we want to update the current design for mobile to be more user friendly.

Confusion can occur when generating a meal plan or updating health entries. Both of these issues happen because the application does not refresh or update the current screen. This would be first in line to solve on the frontend. Each respective action that changes the current frontend, should also update what the user is able to see on the current page. This would result in the user not creating several entries due to not knowing if it was a success.

BIBLIOGRAPHY

[1] STOPWASTE. *Stop food waste.* `https://stopfoodwaste.org/food-and-climate`. (Accessed on 09/22/2023).

[2] Josh Jackman. *Food Waste Facts and Statistics.* Accessed on November 23, 2023. 2023. URL: `https://www.theecoexperts.co.uk/home-hub/food-waste-facts-and-statistics`.

[3] Stop Spild Af Mad. *Madspild i tal.* Accessed on November 24, 2023. 2023. URL: `https://stopspildafmad.org/om-madspild/madspild-i-tal/`.

[4] rts. *Food Waste in America in 2023.* Accessed on November 24, 2023. 2023. URL: `https://www.rts.com/resources/guides/food-waste-america/`.

[5] United States Environmental Protection Agency. *Preventing Wasted Food At Home.* Accessed on November 23, 2023. 2023. URL: `https://www.epa.gov/recycle/preventing-wasted-food-home`.

[6] AHISHA GHAFOOR. *Stop food waste movement Denmark.* `https://www.scandinaviastandard.com/the-best-ways-to-reduce-food-waste-in-denmark/`. (Accessed on 09/22/2023). 2021.

[7] FDA. *Stop food waste movement Denmark.* `https://www.fda.gov/food/consumers/tips-reduce-food-waste`. (Accessed on 09/22/2023). 2022.

[8] Stop Spild Af Mad. *Om Stop Spild Af Mad.* Accessed on November 23, 2023. 2023. URL: `https://stopspildafmad.org/`.

[9] *Stop food waste day.* `https://www.stopfoodwasteday.com/en/about.html`. (Accessed on 09/22/2023).

[10] *Stop food waste movement Denmark.* `https://stopwastingfoodmovement.org/`. (Accessed on 09/22/2023).

[11] Stine Schramm. *Time trends in body mass index distribution in the general population in Denmark from 1987 to 2021.* Accessed on November 24, 2023. 2023. URL: `https://ugeskriftet.dk/dmj/time-trends-body-mass-index-distribution-general-population-denmark-1987-2021`.

[12] Emily Laurence. *Obesity Statistics.* Accessed on November 24, 2023. 2023. URL: `https://www.forbes.com/health/body/obesity-statistics/`.

[13] Centers for Disease Control and Prevention. *Causes of Obesity.* Accessed on November 24, 2023. 2022. URL: `https://www.cdc.gov/obesity/basics/causes.html`.

[14] NHS. *Obesity - Causes*. Accessed on December 13, 2023. URL: https://www.nhs.uk/conditions/obesity/causes/.

[15] trainingcor. *5 BENEFITS OF COUNTING CALORIES*. Accessed on November 24, 2023. 2023. URL: https://www.trainingcor.com/5-benefits-counting-calories/.

[16] *The Automatic Meal Planner - Eat This Much*. https://www.eatthismuch.com/. (Accessed on 09/22/2023).

[17] Kasper Hjortsballe. *No Waste*. https://www.nowasteapp.com/. (Accessed on 09/22/2023).

[18] *Kitche*. https://kitche.co/the-app/. (Accessed on 09/22/2023).

[19] MyFitnessPal. *Essential Guide to MyFitnessPal Premium*. https://blog.myfitnesspal.com/essential-guide-to-myfitnesspal-premium/. (Accessed on 09/22/2023).

[20] Rahul Awati. *monolithic architecture*. Accessed on October 2, 2023. 2022. URL: https://www.techtarget.com/whatis/definition/monolithic-architecture.

[21] Milad Rezaeighale. *Stateful and stateless architecture design*. Accessed on December 7, 2023. 2022. URL: https://www.linkedin.com/pulse/stateful-stateless-architecture-design-milad-rezaeighale/.

[22] Chris Richardson. *Pattern: Monolithic Architecture*. Accessed on December 6, 2023. 2023. URL: https://microservices.io/patterns/monolithic.html.

[23] Chris Richardson. *What are microservices?* Accessed on December 5, 2023. 2023. URL: https://microservices.io/.

[24] Anna Dziuba. *Guide to Implementing Microservices Architecture On AWS [With Examples]*. https://relevant.software/blog/microservices-on-aws/. (Accessed on 10/02/2023). 2021.

[25] Mayank Modi. *Designing for Scalability with Microservices: How to build scalable systems using microservices architecture*. Accessed on December 7, 2023. 2023. URL: https://www.linkedin.com/pulse/designing-scalability-microservices-how-build-scalable-mayank-modi/.

[26] Joseph Ingeno. *Stateless versus stateful microservices*. Accessed on December 7, 2023. 2023. URL: https://www.oreilly.com/library/view/software-architects-handbook/9781788624060/c47a09b6-91f9-4322-a6d4-9bc1604b1bdf.xhtml.

[27] Sushant Kumar. *Layered Pattern*. Accessed on October 2, 2023. 2023. URL: https://www.scaler.com/topics/software-engineering/layered-pattern/.

[28] O'Reilly. *Layered Architecture*. Accessed on December 18, 2023. 2023. URL: https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/ch01.html.

[29] Study Board for Computer Science AAU. *Internet Study Plan*. https://moduler.aau.dk/course/2023-2024/DSNSWK120. Accessed: December 7, 2023. 2023.

[30] Amazon Web Services. *What is Load Balancing?* `https://aws.amazon.com/what-is/load-balancing/`. Accessed on 7 Dec 2023. 2023.

[31] nOps. *Horizontal vs Vertical scaling: An in-depth Guide.* `https://www.nops.io/blog/horizontal-vs-vertical-scaling/`. Accessed on 7 Dec 2023. 2023.

[32] Google LLC. *Protocol Buffers Documentation.* Accessed on December 11, 2023. URL: `https://protobuf.dev/`.

[33] gRPC. *gRPC Core Concepts.* Accessed on December 11, 2023. URL: `https://grpc.io/docs/what-is-grpc/core-concepts/`.

[34] Internet Engineering Task Force (IETF) et al. *JSON Web Token (JWT).* Accessed on December 15, 2023. 2015. URL: `https://datatracker.ietf.org/doc/html/rfc7519`.

[35] statista. *Global mobile traffic 2022.* Accessed on December 15, 2023. URL: `https://www.statista.com/statistics/277125/share-of-website-traffic-coming-from-mobile-devices/`.

[36] Chris Richardson. *Database per Service.* https://microservices.io/patterns/data/database-per-service.html. Accessed on December 14, 2023. 2018.

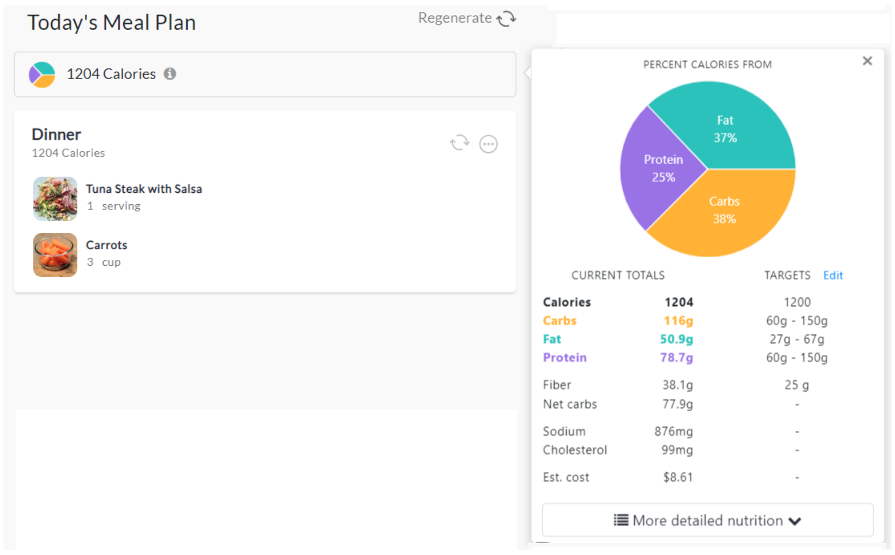
[37] *A Future-Adaptable Password Scheme.* Accessed on December 11, 2023. 1999. URL: `https://www.usenix.org/legacy/events/usenix99/provos/provos.pdf`.

[38] *What is React.js?* Accessed on December 12, 2023. URL: `https://codeinstitute.net/global/blog/what-is-react-js/`.

[39] w3schools. *React Components.* Accessed on December 14, 2023. URL: `https://www.w3schools.com/react/react_components.asp`.

[40] Timothy Merritt. *DEB SW3 lecture 8 slides.* Accessed on December 14, 2023.

A. APPENDIX


Today's Meal Plan

1204 Calories

Dinner
1204 Calories

Tuna Steak with Salsa
1 serving

Carrots
3 cup

PERCENT CALORIES FROM

	CURRENT TOTALS	TARGETS
Calories	1204	1200
Carbs	116g	60g - 150g
Fat	50.9g	27g - 67g
Protein	78.7g	60g - 150g
Fiber	38.1g	25 g
Net carbs	77.9g	-
Sodium	876mg	-
Cholesterol	99mg	-
Est. cost	$8.61	-

Figure A.1: Eat This Much generated meal plan[16]

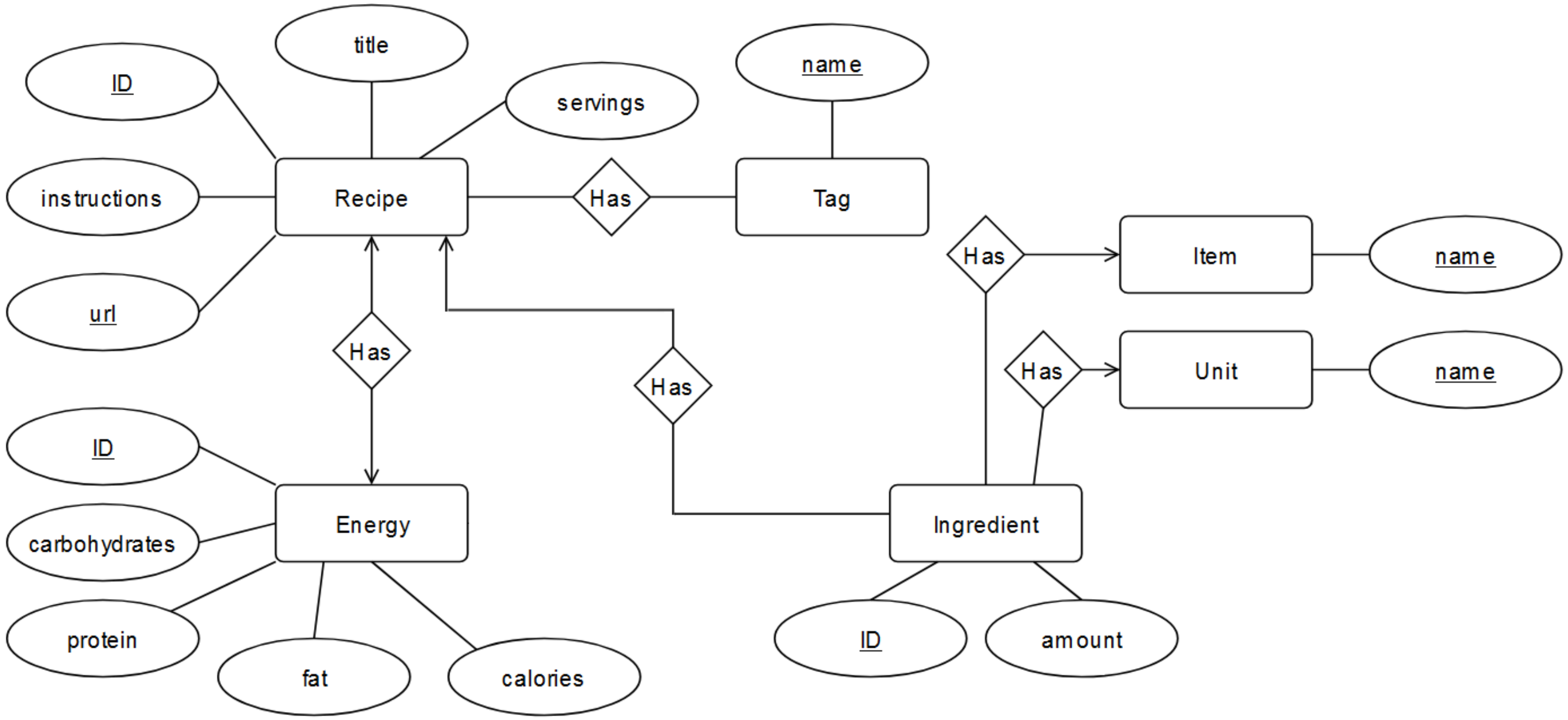


Figure A.2: Recipe ER Diagram

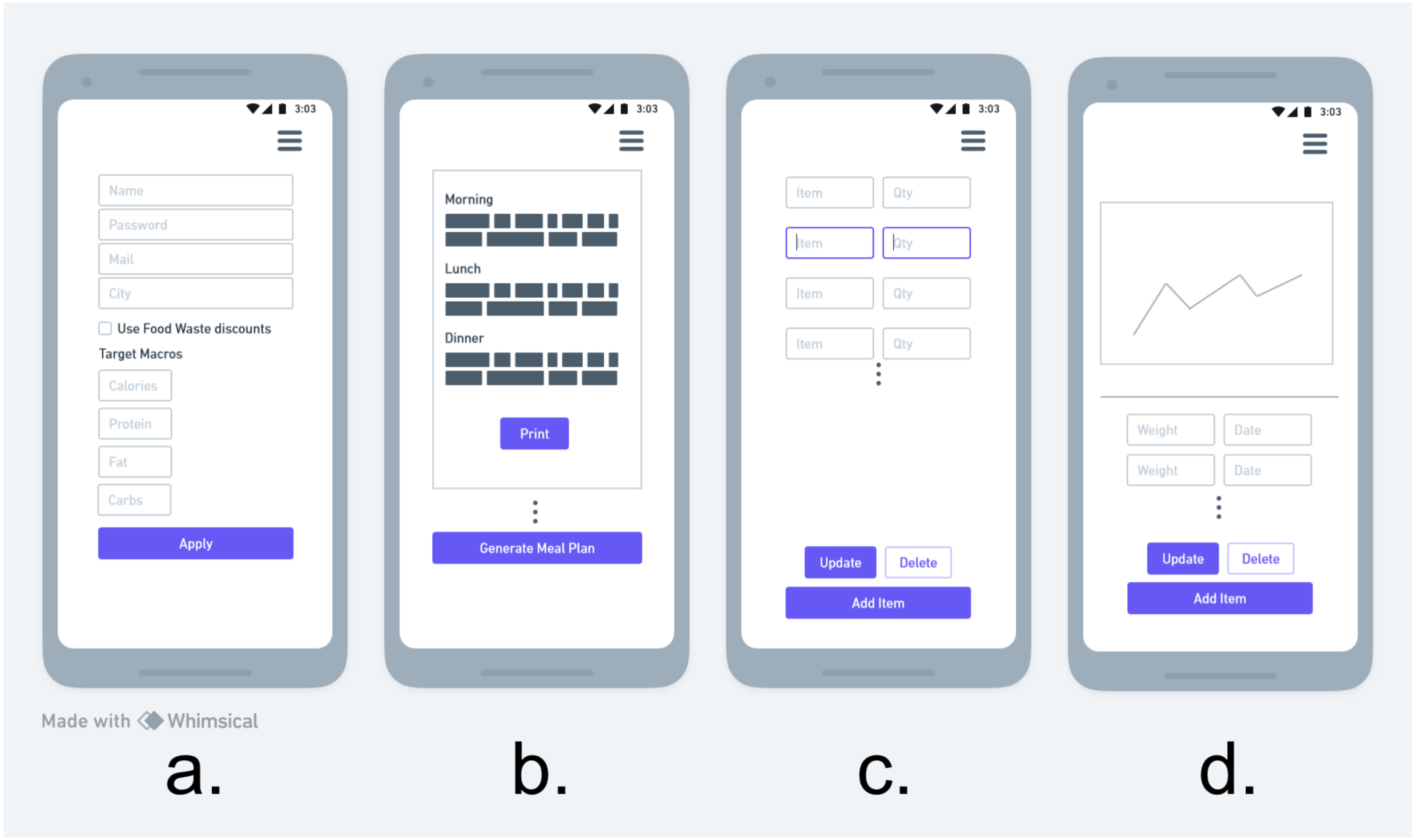


Figure A.3: Low-fidelity wireframes. **a**: create account page, **b**: meals plan page, **c**: inventory page, **d**: health page.