
Bilkent University

Department of Computer Engineering

# Senior Design Project

*Foodster: Maintain your diet easier*

## High Level Design Report

Khasmamad Shabanovi, Gledis Zeneli, Balaj Saleem, Ibrahim Elmas, Perman Atayev

Supervisor: Ozcan Ozturk
Jury Members: Dr. Çigdem Gündüz-Demir, Dr. Can Alkan

# Contents

# 1. Introduction

Food is the first of all fundamental human needs, yet how many times today have you, personally, consciously thought of what you have and will be eating today, what its nutritional value is and how it fits into your greater nutritional, fitness, and lifestyle goals? With the ease of falling into a routine and the abundance of staple food, keeping track of all these variables, and making sure that you plan your meals such that your priorities are heeded, becomes increasingly mundane, monotonous, and just unnecessary extra work.

With the fast-paced lifestyle of the 21st century, monotony and unnecessary work are the last things an individual needs in his/her busy life. Figuring out what to eat, to order or to cook, how to cook it, where to get the ingredients from, and how beneficial this meal would be for your body are questions that would take precious time and energy that could be better employed elsewhere.

Furthermore, the complications of searching for the nutritional data, planning a healthy diet according to one's needs, planning the budget for such a diet, and finding recipes to support this time is truly a cumbersome endeavor.

This is where Foodster comes in.

## 1.1 Purpose of the system

Foodster will simplify the complicated process of eating healthy and take the thinking out of the simple and fundamental process of having food. This means the complete integration of planning meals with one's constraints, listing the ingredients and requirements for such meals, automated ordering of these ingredients or automated ordering of these meals. This draws inspiration from apps such as myfitnesspal which help the user reach their nutritional goals with the least possible effort. All this will be amalgamated into a cross platform mobile application.

## 1.2 Design goals

### 1.2.1 Usability

- Users should be able to find it intuitive to use the system.
- Users should not face any major navigation and operational challenges while using the system.
- Users should be able to use the system to get meaningful and insightful results quickly.

### 1.2.2 Supportability

- Users should be able to use the application using android and iOS Operating systems.
- Users should be able to use the application with phones and tablets that support iOS and Android.

### 1.2.3 Reliability

- No information that was gathered from the user should be lost.
- No meal plan, grocery list that was generated for the user should be lost.
- There will be no down times of the server and all services will always be available.

### 1.2.4 Efficiency

- No functionality of the application should take longer than 1s.
- The response time of the heavy and non-heavy users should not differ drastically, meaning that both should still be able to use any functionality under 1s.

#### 1.2.5 Security

- Users passwords should be hashed to ensure accounts' security.
- Users data should only be used for the purposes of serving users' requests and making the application better.
- Authentication should be required to change any sensitive data of a user.
- Any sensitive data regarding users' payments and cards should be securely stored in the database that can only be accessed by the admins of the system.

#### 1.2.6 Scalability

- If the Latency of the application increases with the number of users, the number of hardware that is serving users as well as the number of databases that are storing should be increased to decrease the latency back to 1s in the worst case scenario.
- If the location of users starts affecting their latency, CDNs should be used to improve their experience with the application.

#### 1.2.7 Extensibility

- The application should have logic separate from the UI, so that it's easy to work on the two concurrently to add new functionalities to the application.
- The application should be coded to be as modular as possible, so that changes in one part of the application does not significantly affect other parts.

### 1.3 Definitions, acronyms, and abbreviations

- CDN - Content Delivery Network.
- REST api - Representational State Transfer. A software architectural style that defines a set of constraints to be used for creating Web services.
- GUI - Graphical User Interface
- Facade class - a software design architecture technique which acts as a facade or a proxy for a larger set of services and classes.
- Http - HyperText Transfer Protocol. Network protocol which regulates information sharing and handling.
- JWT - JSON Web Token. Web standard for secure information transfer between two parties in a network.

### 1.4 Overview

Foodster will be used to reduce the friction of maintaining a healthy diet for users with the goals they have in mind such as losing or gaining weight, or following diets such as paleo, keto and vegan. Also, foodster will make it easier for people to order their groceries as well as keeping track of which groceries they have, order their meals, keep track of their meal plans and more.

## 2. Proposed software architecture

### 2.1 Overview

This section will go over our design choices with regards to Foodster’s software architecture. Section 3.2 will consist of a high level subsystem decomposition, where we outline the modules of our system and how they interact with each other. A more detailed on the services provided by these modules will be provided in section 4. Section 3.3 explains with the help of a UML diagram how our software will be distributed and deployed to the relevant hardware. Section 3.4 and 3.5 cover how data will be managed so that it is persistent and securely stored and accessed, respectively. In 3.6 we explain how the workflow of our software will be controlled. Section 3.7 explores how our software will handle the boundary conditions of initialization, termination, and failure.

## 2.2 Subsystem decomposition

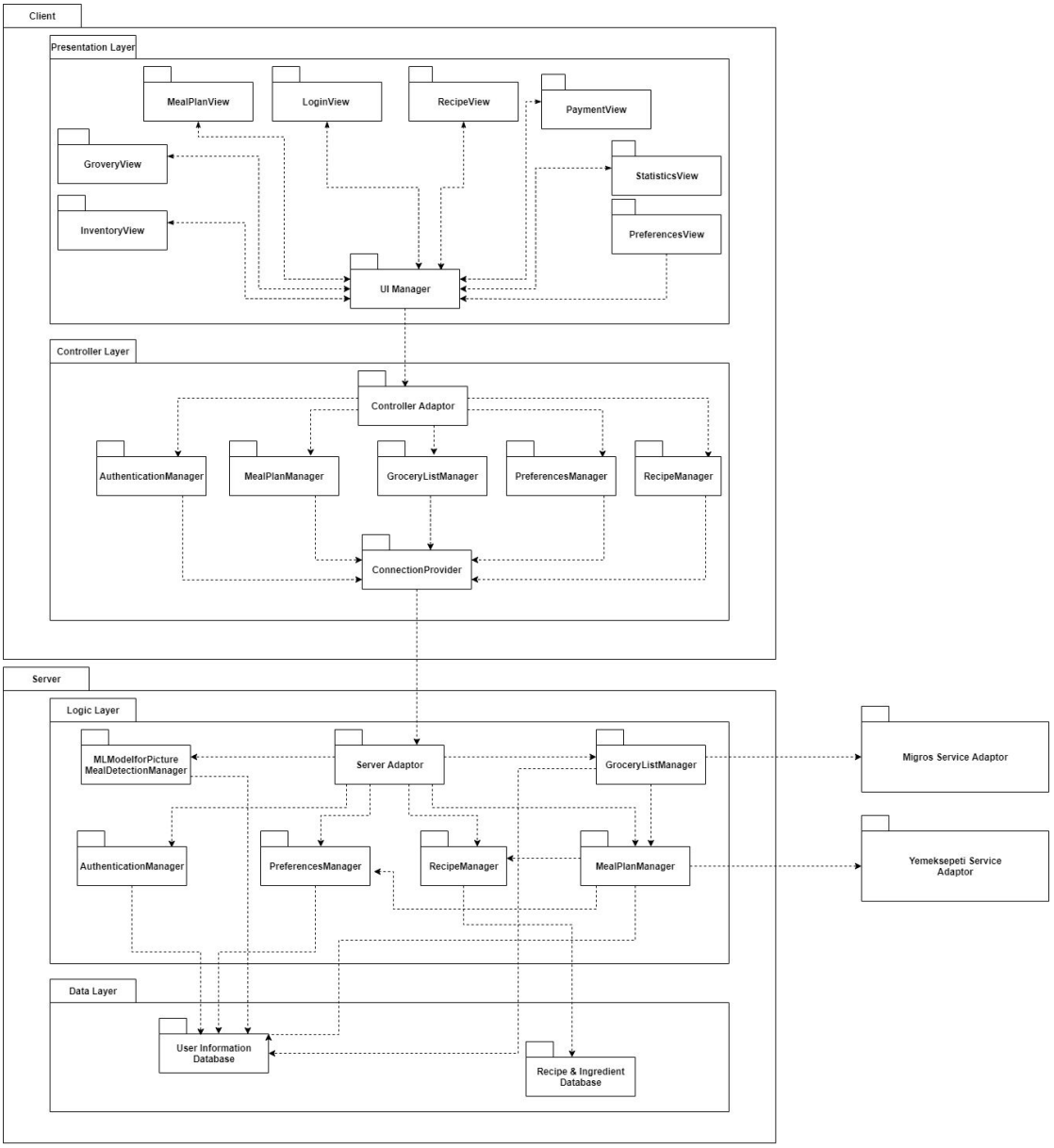


Figure 1: Subsystem Decomposition

On the highest level our software will be structured as a Client-Server architecture with some caveats. The user's side will interact with the client where GUI and REST api calls are made to the server. The Client subsystem is itself broken down into a Presentation module which consists of UI handling components and their listeners. The Presentation layer of the Client communicates with the Logic layer of the Client which processes the raw information and sends it to the server to actually be used.

The Server subsystem, similarly to the client, is itself broken down down to a Logic and Data layer. The Logic layer of the Server is where the majority of the processing of our software will be made. It contains all the required components to provide the processing functionalities of our software with exception to the Migros Service Adaptor and YemekSepeti Service Adaptor. This design choice is deliberate as we intend for these components to be completely self sufficient and not necessarily part of our main application stack. The Logic layer of the Server communicates with the Data layer of the Server to retrieve and update information. The Data layer will store all the information of our software ranging from user info to the data about recipes and ingredients.

All communication between different layers in our system are done through facade classes to decrease coupling. We also use a form of opaque layering where each layer only communicates with the one before and after it. These design choices bring great flexibility in case we wish to refactor some components or change functionalities. Opaque layering causes some overhead through the required transition/facade modules, but it also allows for easier testing as we would only need to mock/stub two layers max for testing any single layer.

Some other consideration worth exploring is why organise the subsystems as Client-Server where both the Client and Server have two additional layers composing them, when 4-Tier architecture could have potentially been used as well. Our decision was influenced by the planned deployment of these layers as 4 Tier architecture suggests the software components are distributed across 4 different systems and must be managed separately, but that is not the case with us (more on this in 3.3).

## 2.3 Hardware/software mapping

The subsystem decomposition diagram presented in Figure x is helpful to illustrate the hardware/software mapping of our system as it is 1-to-1 with what would be the deployment diagram of our system. The Client component and its subcomponents will be deployed on the user mobile device which would be running Android or iOS. The Server component and its subcomponents will be hosted in a cloud server running Linux. There will be no need for special hardware to host the database as the database will be run in Docker containers within the OS of the cloud server. The Logic subcomponent of the Client and the Logic subcomponent of the Server will communicate via http. The Yemek Sepeti Service Adapter and Migros Service Adapter will be deployed in separate cloud servers and will communicate with the Logic subcomponent of the Server via http protocol as well.

## 2.4 Persistent data management

Our application will store all of its user and food related data in a database in the server. The main reason for choosing a database over a file system is querying. Querying is an important part of our application as its services mainly consist of searching and filtering information. Using a file system would be inconvenient due to the large amounts of data and the complicated queries we need to run. The database will guarantee the persistency and handle the concurrent writes and reads for us.

In addition to the database, local device storage will be used to store some small amount of the data fetched from the server to improve user experience and loading time. Such data could include login credentials, meals schedules for the day etc., and they need to be updated consistently based on the information retrieved from the server.

## 2.5 Access control and security

First, let's consider access control. Communication with the server will be done through REST api. Users will be able to register and authenticate freely through the REST api, but every other call must be authenticated via JWT tokens. JWT tokens will be used to regulate user sessions which makes sure that every user has access to only the information and services intended for that user. Even knowing the server address, one cannot use its services without registering and authenticating as a user. Security is also enforced by such a system because nobody else other than the user themselves, and us who have access to the server directly, can access the user’s preference and meal plan data. The decision to not encrypt user data was made after deciding that the data was not critical and applying encryption to that potentially large body of data would cause performance issues. The only piece of user data which is encrypted in the password. Bcrypt encryption is used to hide the password [1]. Bcrypt performs worse than more mainstream encryption methods like SHA256, but it is superior in the security it provides [2].

## 2.6 Global software control

Centralized event driven control will be used for Foodster. These events will be majorly user driven. The events will be triggered by user interaction with the GUI, and send requests to retrieve or update information to the server. The event triggering and handling

in the server will be asynchronously handled. This system fulfills all our functional requirements. The application need only react to user actions and there is no need for procedure driven control. Centralized control is convenient and intuitive to implement, though we may need to use tools such as load balancers to account for increased numbers of users in the future.

Some rarer events could also be model driven, which are triggered on the server side from us developers to notify users about campaigns and promotions.

## 2.7 Boundary conditions

Foodster consists of two major, largely independent, components, the client side software and the server side software. Therefore, it is worth it to consider how the boundary conditions play out for both these software components.

### 2.7.1 Initialization

The initialization state occurs on the client side whenever the user opens the application. On initialization, the application could be in one of two states. Determining which of the two states is the current one is done through looking for JWT Token stored in the device local storage. This requires the initialization of the classes which manage local storage, and the classes which send requests to the server so the authentication token can be verified.

First, when a token is not there or it has expired, the user is not signed in, and they are displayed a login in page. Only the relevant UI is initialized since the classes to communicate with the server are already initialized. When a successful login happens, a series of initialization similar to the second case happens.

Secondly, the token is authenticated so the user is already logged in and is shown the meal plan page. All the primary UI components (referring to the different menu options) are initialized at this stage. The UI is filled with the 'cache like' data stored in the device local storage while information is being retrieved and cross validated from the server. The local storage management and server communication classes are already initialized by this point so no extra work needs to be done.

The initialization state on the server is very rare, and only occurs once on deployment, on complete failures, or after the servers have been taken down for maintenance. The initialization consists of starting the Docker containers and exposing the database to a port accessible from the server. Then the server software itself is configured, started, and exposed to a port; then the REST api endpoints are set up, concluding the server side software initialization.

### 2.7.2 Termination

Termination of the client software happens whenever the user closes the app. No special action needs to be taken within the software as the OS of the device will handle the resource deallocation. If any requests made by the application are still under process in the server, their changes will be stored, but their return values will be ignored as the application is no longer running. This is ok because there are no critical operations the server could be performing with regards to sending information back to the user.

Termination on the server side is rare and would happen when the server would require maintenance. All new requests would be blocked, and there would be a need to wait for all running requests and queries to terminate so as not to lose any data. Once there is no longer any activity on the server, the database Docker containers would need to be stopped and the server software be terminated.

### 2.7.3 Failure

Failure on the client side could happen on three levels. First, uncaught exceptions could cause the program to crash; in such scenarios a crash report would be sent to the server. The application would report the failure to the user, and the user would be allowed to continue using the app from the last stable state. Second, connection errors could cause

activity to halt; it would cause inconvenience, but no error in data storage or any other process. The user would be notified and they could try again later. Third, the device OS may decide to abruptly terminate the application to free resources for other applications; this is not problematic as this case is almost symmetric to the termination case of the client side software.

Failure on the server side could really only happen due to unhandled exceptions. If the hardware fails and the server fails is out of our control, and many cloud services have insurance packages to account for data loss damages in case this rare scenario happens. Due to the operations in the server being uniform and asynchronous, only that single request would fail, its failure would be logged and reported to the user, and the other processes would continue to operate normally. Restarting the server in such cases could be problematic.

# 3. Subsystem services

Foodster is composed of the interactions of two separate systems: Client (End User) and the Server

## 3.1 Client

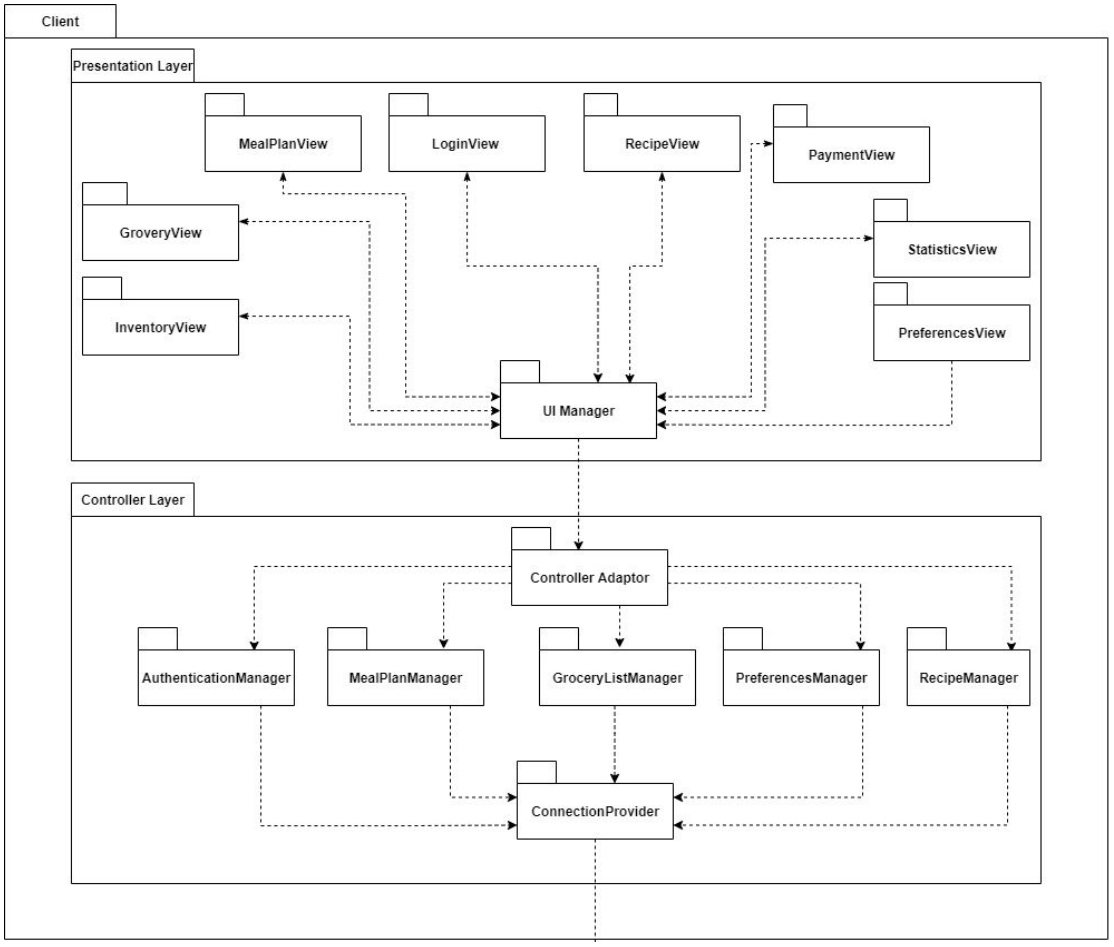


Figure 2: Client Subsystem

The client can be used with the Android and IOS operating systems via smartphones or tablets. The client comprises the subsystems of the presentation and controller. Presentation layer is responsible for presenting our functionalities to the users in a user-friendly interface. Controller layer is responsible for fetching data or posting data to the server when events are initiated by the view class. Both presentation and controller layers are going to be discussed next.

### 3.1.1 Presentation Layer

The layer of the presentation includes views.

**MealPlanView:** The view class for presentation of meal plans and interactions of users with meal plans like adding a meal to plan or changing meal plan.

**LoginView:** The view class for authentication - sign in, sign up-. Takes input from the user for authentication processes and interacts with UIManager to check the credentials like username, password.

**RecipeView:** The view class for presentation of recipes and their ingredients, filtering recipes according to some filters such as nutrition quantity - 20g protein per 100g -. Interacts with UIManager for fetching recipe and nutrition data.

**GroceryView:** The view class for presentation of current grocery list and grocery list operations such as generating grocery list for chosen days or specific meals in a day. Interacts with UIManager to fetch current grocery list data, meal plan data for generating grocery list and generating a grocery list and sending it to the server.

**InventoryView:** The view class for presentation of current inventory and operations such as adding an item/ingredient to current inventory or removing an item. Interacts with UIManager to fetch inventory data and send inventory changes to the server side.

**PaymentView:** The view class for presentation of payment operation. Interacts with UIManager to check payment details like card number and make the payment.

**StatisticsView:** The view class for presentation of statistics of the user such as average daily protein intake in last 5 days, daily nutritional intakes. Interacts with UIManager to fetch statistics data from the server.

**PreferencesView:** The view class for presentation of preferences like desired daily protein intake, allergies. This view class is also the presentation class for preference change operations. Interacts with UIManager to fetch preferences data and send preference changes to the server.

**UIManager:** The class that manages the views and provides communication between controller layer and view classes. Interacts with the view class to get updates to send the server.

### 3.1.2 Controller Layer

The controller layer includes manager classes for the view classes in the presentation layer. These manager classes are responsible for communication between the view classes and server.

**ControllerAdaptor:** The facade class for the communication of UIManager with the manager classes. Interacts with all of the managers to provide an api to the presentation layer.

**AuthenticationManager:** The class that manages authentication operations like signin in and signing up. Interacts with ConnectionProvider to send credentials to the server.

**MealPlanManager:** The class that manages meal plan operations like meal plan data fetches, meal plan changes and new meal plan generations. Interacts with ConnectionProvider for fetching meal plan data, sending meal plan changes that are made in the presentation layer to the server and sending details for a new meal plan to request a new meal plan.

**GroceryListManager:** The class that manages grocery list operations like fetching current grocery list data, generation of new grocery list. Interacts with ConnectionProvider for fetching grocery list data and meal plan data and sending grocery list generation details like chosen days and meal to the server for generation of grocery list.

**PreferencesManager:** The class that manages preferences operations like fetching preferences data from the server and providing this data to the presentation layer and sending preferences changes that are made in the presentation layer to the server. Interacts with ConnectionProvider to fetch preferences data and send preferences changes to the server.

**RecipeManager:** The class that manages recipe operations like fetching recipe and ingredient data, search queries with some filters. Interacts with ConnectionProvider for fetching recipe/ingredient data and sending search queries to the server.

**ConnectionProvider:** The class that is used by the manager classes to communicate with the server. Interacts with the ServerAdaptor class of the server via REST api.

## 3.2 Server

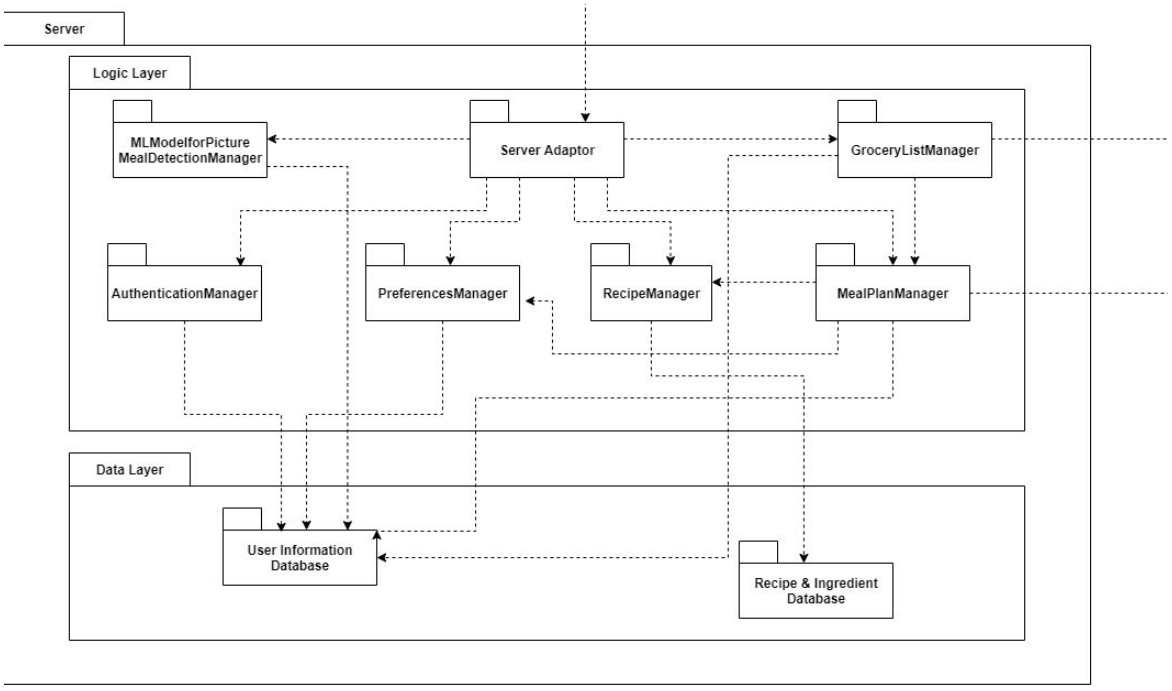


Figure 3: Server Subsystem

### 3.2.1 Logic Layer

The logic layer includes the business logic of our system. There are separate manager classes for major functionalities of our system like meal planning, grocery lists. This layer has also a facade class ServerAdaptor which is the doorway of the server to client.

**ServerAdaptor:** The class that is the doorway of the server to client-side. Interacts with all of the controllers to provide their functions to client-side.

**PictureMealDetectionManager:** The class that detects the ingredients from the given picture. Interacts with User Information Database to log these ingredients.

**GroceryListManager:** The class that contains business logic for grocery list operations like fetching grocery list data, generating a new grocery list. Interacts with MealPlanManager to get the meal plan data of the given days and specific meals to get recipes and their ingredients. Interacts with Migros Service Adaptor for required operations to order the current grocery list. These operations are

- checking if all of the ingredients are in stock
- searching for alternatives for the ingredients that are out stock at the moment

- preparing the shopping cart
- making payment

Interacts with User Information Database for fetching current grocery list data, saving new generated grocery list.

**AuthenticationManager:** The class that contains business logic for authentication operations - signing in, signing up-. Interacts with User Information Database to check credentials or store them.

**PreferencesManager:** The class that contains business logic for preferences operations like fetching preferences data, making preferences changes that are made in the client-side. Interacts with User Information Database to perform these operations.

**RecipeManager:** The class that contains business logic for operations related to the recipe and ingredients like fetching recipe data and data of the ingredients of this recipe, searching for recipes with some filters. Interacts with Recipe & Ingredient Database for these operations.

**MealPlanManager:** The class that contains business logic for meal plan related operations like fetching current meal plan data, adding an extra meal to current plan, generation of a new meal plan and changing some meals in the current meal plan. Interacts with RecipeManager to get some recipes with some filters that are adjusted with respect to preferences of the users like daily nutritional intake goals, unliked meals, favorite meals and allergies. Interacts with User Information Database for fetching current meal plan, saving the generated meal plan and updating the current meal plan with given changes.

#### 3.2.2 Data Layer

Data layer handles the main storage of the server

**User Information Database:** This database contains tables and other database structures required for storing user account data, user preference data, meal plan data and grocery list data. We will have separate collections to store these datas.

**Recipe & Ingredient Database:** This database contains recipe and ingredient datas.

## 4. Consideration of Various Factors in Engineering Design

Our product revolves around optimising the convenience and health factors of cooking. Therefore, a big part of our design process will be food and health related considerations ranging from more serious conditions like allergies and diabetes to milder concerns like making sure the users receive a well balanced variety of nutrients. These will dictate how we configure our recommendation algorithms. Depending on how one looks at it, these may also be considered as safety concerns especially when certain foods might risk the well being of the users.

One of the features of our app is financial optimization of cooking. We will focus on designing a system which achieves this while maintaining quality of product recommendations.

In order to make the experience most tailored to the user, we will design our recommendations as such to take into account the food culture of the region where the user lives and has grown up with. We believe this will have a great impact on the user experience with the app.

Table 1. Factors that can affect analysis and design.

| | Effect level | Effect |
|---|---|---|
| Public health | 10 | Improving the health of the users is a primary concern. We will have to configure our recommendations so that they provide the greatest benefits to our users. |
| Public safety | 4 | We will have to put reliable failsafes to protect users who can have adverse health effects from specific foods. |
| Public welfare | 4 | Designing our tool so we both collect optimal data and give proper recommendations to help our users make cheaper and qualitative choices. |
| Global factors | 0 | Our app is quite personal in the experience it provides and therefore its design is not really reliant on global factors. |
| Cultural factors | 8 | In order to provide the best experience possible our recommendations will have to be tweaked such that they account for the cultural elements of local cuisine. |
| Social factors | 0 | We did not find any social factors which may be critical to consider during our design process. |

# 5. Teamwork Details

## 5.1 Contributing and functioning effectively on the team

We have determined our goals beforehand and made sure that all of us are aware of these goals, so that we can make decisions faster and spend more time on development. Also, we divide tasks into modules to ensure proper teamwork. These modules are assigned to 3 people at most. Each of these sub-team has a leader assigned. By means of modulation and structure of our sub-teams, we can specialize in that field and give proper support to each other. What is more important, we always include all team members in the planning processes for the team to function effectively. So that nobody feels excluded and all members will share their ideas and be a part of not only work, but also decision making throughout the project. We prefer short frequent weekly meetings instead of long discussion meetings, which we found to be unproductive. These meetings are used to inform each other on our progress on our assigned tasks.

## 5.2 Helping creating a collaborative and inclusive environment

In all of our work, we always keep others up to date on our progress to track each other's work and not to miss any important information. Besides informing what we have done on our task, we also let our team know about bad situations like not being able to handle the task assigned to him/her. So that we can check our division again and make a better distribution of tasks to ensure that all members are contributing. What is more, before we submit our work, each one of the tasks is assigned to some other team-mate for review and feedback. So that we ensure the quality of the work and give each other support.

### 5.3 Taking lead role and sharing leadership on the team

We did not have a single leader throughout the project. We have decided that every member will be the leader of the organizational proceedings of the team on a weekly basis. The thinking behind this strategy is that everybody will lead the team sometimes and will be able to dedicate themselves toward our project's success. Similarly, we have also divided our project into work packages, and these packages also have assigned leaders who organize the work done and coordinate with other package leaders on the implementation.

## 6. References

[1] https://www.usenix.org/legacy/events/usenix99/provos/provos_html/node1.htm

[2] https://www.cs.princeton.edu/~appel/papers/verif-sha.pdf