Frankfurt University of Applied Sciences
Fachbereich 2: Informatik und Ingenieurwissenschaften
Studiengang: Informatik (B.Sc.)

Project Report

Process Project Programming Exercises

Andrew Belmonte Sato
Matrikelnummer: 1340196
Huy Anh Dang
Matrikelnummer: 1407958
James Belmonte Sato
Matrikelnummer: 1340604
Martin Komarnicki
Matrikelnummer: 1260603

Submitted on: 14. Juli 2022

Lecturer: Ralf-Oliver Mevius
Module 22: Programming Exercises
Summer semester 2022

Eigenständigkeitserklärung

Hiermit erklären wir, dass wir die vorliegende Arbeit eigenständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel verwendet sowie die aus fremden Quellen direkt oder indirekt übernommenen Stellen/Gedanken als solche kenntlich gemacht haben. Diese Arbeit wurde noch keiner anderen Prüfungskommission in dieser oder einer ähnlichen Form vorgelegt. Sie wurde bisher auch nicht veröffentlicht.

Hiermit stimmen wir zu, dass die vorliegende Arbeit von der Prüferin/ dem Prüfer in elektronischer Form mit entsprechender Software auf Plagiate überprüft wird.

X Belmonte Sato	X [illegible]
Andrew Belmonte Sato	Huy Anh Dang
X J. Belmonte	X Komarnicki
James Belmonte Sato	Martin Komarnicki

Contents

List of Figures

List of Tables

Listings

List of Abbreviations

API	Application programming interface
REST	Representational state transfer

Abstract

This project report serves as documentation of the computer science project at Frankfurt University of Applied Sciences in the bachelor's degree program in computer science in the summer semester of 2022.

Nowadays teamwork and the ability to work in a team are essential in working life. A social workspace helps the productivity and collaboration between colleagues [4]. Furthermore learning to handle social interactions and problems between colleagues can only be learned by practically having social confrontations. Thus working in a team of multiple people and trying to coordinate to get a project done in a certain time-frame is an experience that brings many newly learned skills with itself. By using this point of view while developing a solution for the given task, many interactions can be taken along and used later in a working environment.

In addition, using structured frameworks for project management such as scrum or agile is essential in solving problems and creating a product. Using the scrum methodology in practice to finish a project will show the benefits of scrum: It encourages the product to be built faster, since a set of goals have to be completed in each sprint. Through the sprints it automatically requires frequent planning and setting goals for every week. This helps the team focus only on current sprint goals to increase the productivity.

Lastly using tools like Zoom, Postman, ClickUP, Overleaf and Github greatly increases the producivity of a team. They are not just used in every working environment, but also encourage to work in a group. Documentation in a team is not just important, but also complicated. Using different tools to handle documentation and communication changes how working is nowadays compared to the past. Thus using these practices in a project is good preparation for working later.

1 Introduction

1.1 Motivation

The module "Programming Exercises" is carried out in the 4th semester of the Bachelor's program in Computer Science. Upon successful completion of the module, students should have learned certain competences, such as planning and executing the software engineering process, as well as being able to program together at an in-depth level. Students should also be able to work together to form a team and adhere to a self-created schedule, as well as communicate at a high technical level to achieve results as a team. If unexpected complications arise both technically and socially, as a team this hurdle should be overcome. Furthermore students should have the ability to realize an application that cover the aspects of distributed systems, graphics and sound, user interaction, and a relational database management system. Additionally, structurizing and producing documentation using appropriate english terminology should also be achieved. Therefore this project report was created to serve as a project result and documentation to reflect the skills learned.

1.2 Problem Definition

In this course students should imagine the following scenario:

You are an employee at a software company and work in the development department that implements innovative database systems. Due to the very high competition among database producers you have to develop a new software for the CeBit. In the following project meeting your development team will be given the task to develop the design. You will work directly with the colleagues that are tasked in programming and implementing your requirements.

1.3 Structure of Project Report

In chapter 2 of this project work, we will write about the general procedure of the project. After a short presentation of our project goal, the concrete procedure within the group is explained. Furthermore, the definition of the milestones/sprints as well as the tools used will be discussed. In chapter 3 the project execution is explained and in the subchapters the achieved results are presented, which also complement the individual topics of the respective functionalities. Chapter 5 forms with the conclusion an outlook into the possible future for this application.

2 Project

2.1 Procedure

Appropriate distribution of tasks in the collective brought many advantages for the work situation and the team members. Using the maximum capacity of each team member was an effective way to improve work efficiency. In order to understand the desires and capabilities of each member, conversations and discussions were held early on. Assigning tasks that matched each person's productivity helped members work more effectively and with a more pleasant spirit. Members were assigned specific tasks with deadlines. A permanent meeting was held each week via Zoom or on-site. Spontaneous meetings could also be held with the highest spirit and concentration. The analysis of task assignments was important to understand what needed to be done and what tools were necessary. Knowledge sharing helped members to acquire knowledge immediately and use it effectively. After a successful analysis, our team began to develop a plan. The work was divided and controlled by the project manager.

2.2 Assessment of Sprints

In the first meeting of the group, it was decided to define multiple sprints over a longer period of time (see figure 2.1).The reason for this was to construct a clear thread in the project work in order to be able to deal with the amount of information in a structured manner. In addition the sprints helped give an overview of what was reached at the end of each week and which tasks would have to be done for the next week. Each group member could assign himself to a given task and thus tasks could be evenly divided under each team member:

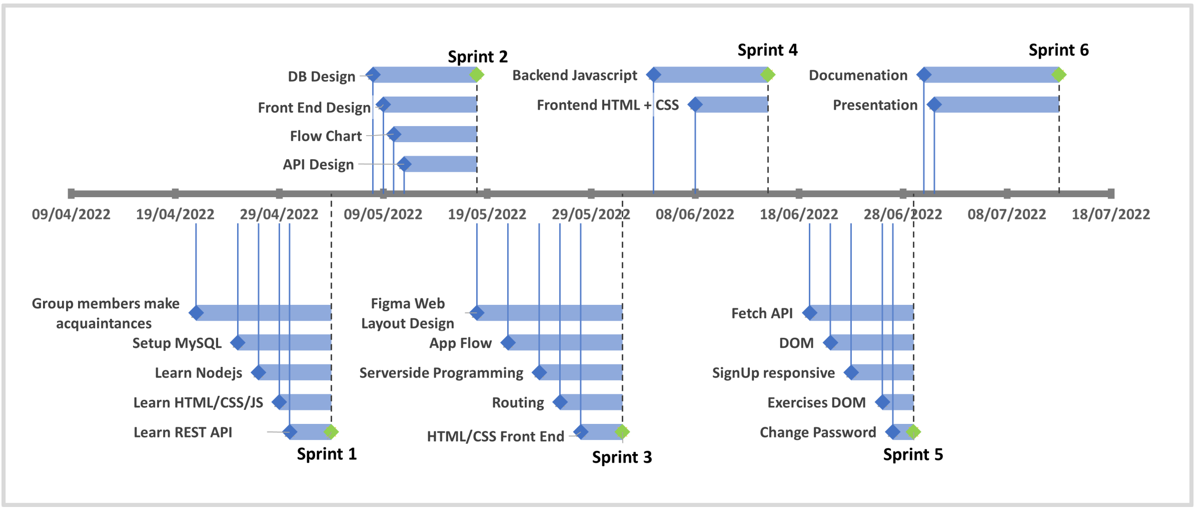


Figure 2.1: Schedule of the project

Sprint 1

- realizing a MySQL database using Amazon RDS
- Collect information and study the topic REST API
- Learn possibly important programming languages like HTML/CSS/JS
- Deadline: 04.05.2022

Sprint 2

- Created a an early database design diagram
- made first version of frontend design using igma
- made a flow chart diagram showing how the website will be accessed by user and admin
- Deadline: 18.05.2022

Sprint 3

- finished frontend design using figma layour design
- discussed the previously created flow chart diagram and about the app flow
- serverside programming and routing
- first implementation of frontend using HTML/CSS
- Deadline: 01.06.2022

Sprint 4

- starting of backend using Javascript

- at the same time analysing and correcting mistakes in frontend with HTML/CSS
- Deadline: 15.06.2022

Sprint 5

- further backend development using fetch API, DOM
- checked if SignUp request in REST API is completed
- implementation of the DOM for exercises
- implementation of the ability to change password
- Deadline: 29.06.2022

Sprint 6

- analysing the results and creating documentation
- preparation for presentation
- Deadline: 13.07.2022

Common IT project management methods, such as the Scrum method, made it possible to achieve results early on. As a result, there was more time to reflect on minor details at the end of the project work.

2.3 Used Tools

The hardware and software environments used for implementation and evaluation are briefly described below.

2.3.1 Figma

Figma is an interface design software that can run in a browser. It is structured in a manner to be used in a team-based, collaborative environment. It provides the necessary tools for the design phase of a project including tools that illustrate certain functions of a website, prototyping capabilities and code generation. Figma can only be used online, which brings benefits like real life collaboration. You can work together with colleagues at the same time using the same project. This means that you do not have to worry about transferring files between team members because the project is constantly being synchronized [11]. Considering all the benefits that were mentioned, figma is the optimal tool for designing and implementing the first ideas of the team.

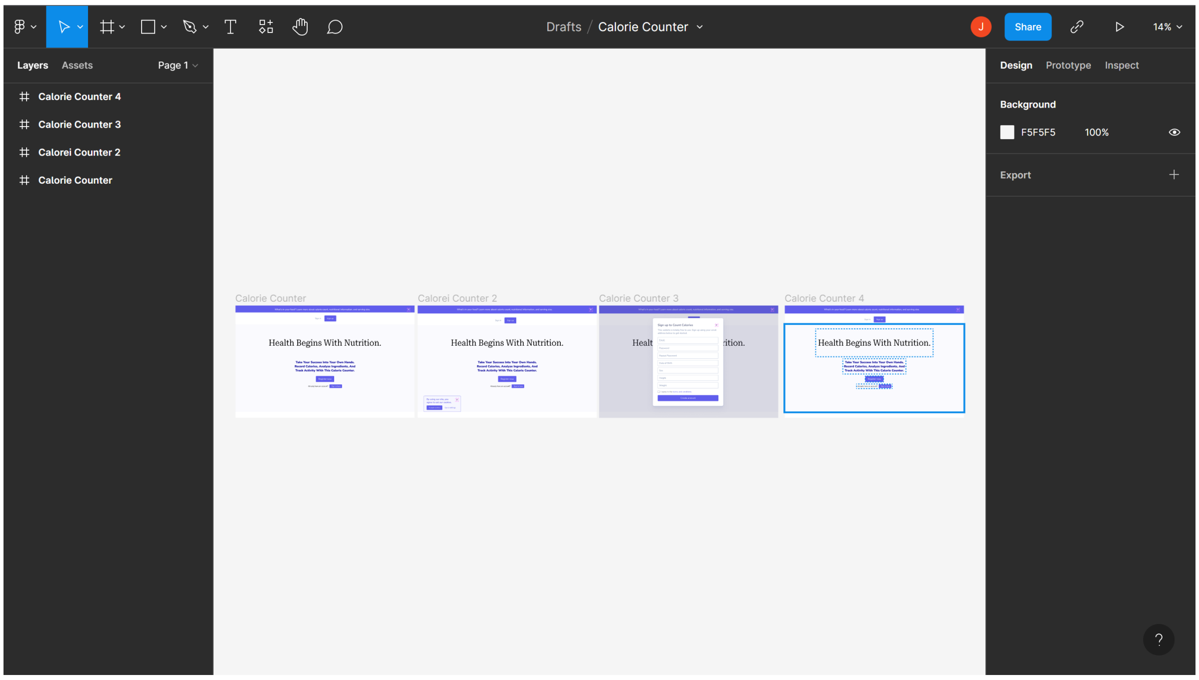
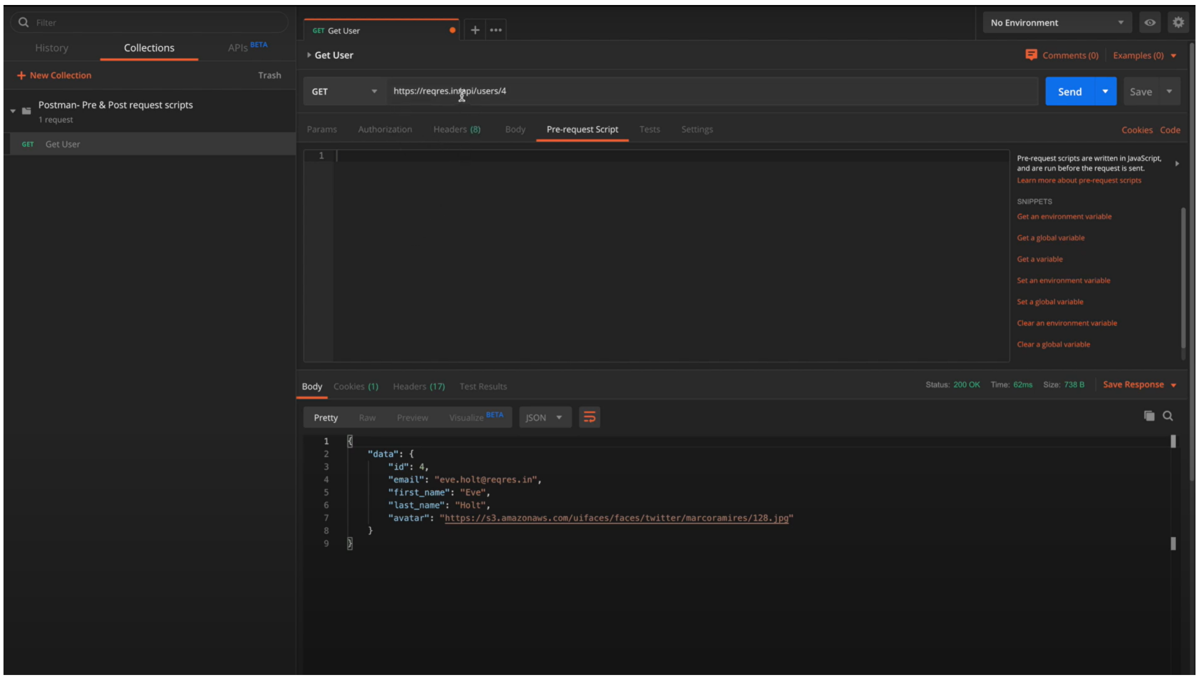


Figure 2.2: Figma interface

2.3.2 Postman

Postman is used to send requests from a client to a server. It is a tool which is used to test an API. You can develop, test, share and document APIs. Postman uses an interface for creating requests or getting responses of requests. The created requests can be stored as collections, which can be shared and sent to colleagues. This allows for a more efficient and less tedious work between the team.

The postman application can handle GET requests to retrieve data from an API, POST requests to send new data to an API, PUT and PATCH requests to update data that already exists and DELETE requests that removes data that already exists [10]. After creating a request through postman, it will return a response body and a response code (see figure 2.3).



Pre-request scripts are written in JavaScript, and are run before the request is sent.
Learn more about pre-request scripts

```
{
    "data": {
        "id": 4,
        "email": "eve.holt@reqres.in",
        "first_name": "Eve",
        "last_name": "Holt",
        "avatar": "https://s3.amazonaws.com/uifaces/faces/twitter/marcoramires/128.jpg"
    }
}
```

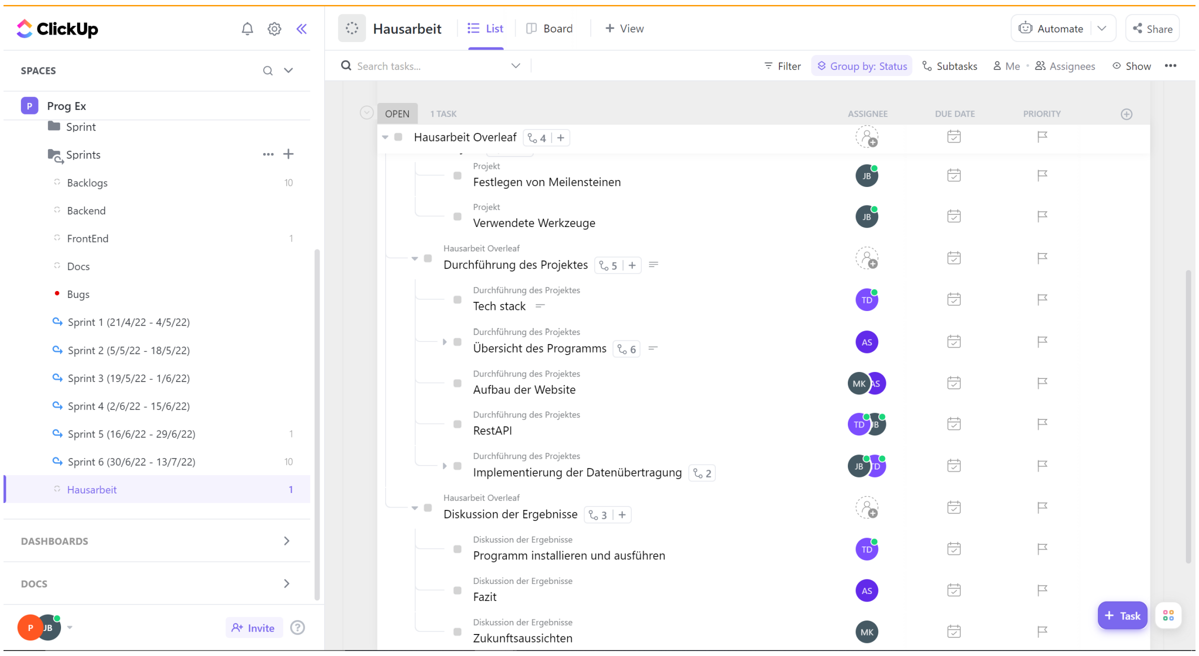
Figure 2.3: Postman interface

The response code can be: 200 for a successful request, 201 for a successful request and that data was created, 204 for an empty response, 400 for a bad request, 401 for unauthorized access, 403 for forbidden or access denied, 404 for data not found, 405 if the method is not allowed or if the requested method is not supported, 500 for internal server error, 503 if the service is unavailable [9].

Postman can also use other key features such as request cURLs, request tests, request collection and request runners [10].

2.3.3 ClickUp

Clickup is a project management tool to help teams collaborate, manage projects, set goals and create and share documents [6]. You can create custom sprints and tasks that can be assigned to anyone. Using ClickUp weekly and assigning each colleague to a task, helped keep an overview of what had to be done each week. With the help of ClickUp the team could organize and structure the project goal in way that was evenly distributed between each student. Figure 2.4 shows the ClickUp interface. On the left of the page you can see each sprint that was created during each weekly meeting. Each sprint has tasks that were divided between group members. Furthermore in the middle of the page each task can have an assignee that should finish the task during the sprint. You can also set a priority for each task.



Hausarbeit Overleaf 4

Projekt
Festlegen von Meilensteinen JB

Projekt
Verwendete Werkzeuge JB

Hausarbeit Overleaf
Durchführung des Projektes 5

Durchführung des Projektes
Tech stack TD

Durchführung des Projektes
Übersicht des Programms 6 AS

Durchführung des Projektes
Aufbau der Website MK AS

Durchführung des Projektes
RestAPI TD JB

Durchführung des Projektes
Implementierung der Datenübertragung 2 JB TD

Hausarbeit Overleaf
Diskussion der Ergebnisse 3

Diskussion der Ergebnisse
Programm installieren und ausführen TD

Diskussion der Ergebnisse
Fazit AS

Diskussion der Ergebnisse
Zukunftsaussichten MK

Figure 2.4: ClickUp interface

3 Project Implementation

3.1 Techstack

3.1.1 Frontend

The foundation of websites is provided by HTML, CSS and Javascript. HTML stands for Hyper Text Markup Language. It is the standard markup language for creating Web pages and describes the structure of a Web page. On other hand, CSS stands for Cascading Style Sheets, and is used to design our website appearance.

3.1.2 Backend

Node JS

Node JS is an open-source backend runtime environment for Javascript. With Node JS we can develop applications on the server using Javascript. Node JS is extremely scalable, light, quick, and data-intensive. It uses the Chrome v8 engine, which turns Javascript code into machine code. Node JS receives requests from clients, processes them in a single thread, and then gives back the response. Node JS uses the idea of threads to manage requests or I/O activities. A thread is a set of operations that the server must carry out. In order to deliver the information to lots of clients with low latency, it runs concurrently on the server. Node JS uses event loop to handle concurrent requests with a single thread without blocking it for one request [13].

We use Node JS because of its reusability. Both frontend and backend can be written in JavaScript. With the help of framework like Express.js as backend (a Node.js framework). It increases productivity and is faster to set up, develop our web application. Because we used only one language for both frontend and backend, we don't waste time to switch between context of multiple languages. Finally, the NPM, which stands for Node Package Manager. NPM is a Node JS's package ecosystem. It is the world's largest ecosystem of open-source libraries. Thousands of open source developers contribute to NPM every day, and it is free to use. An out-of-the-box command-line tool is included with NPM. The package we require can be easily found on the NPM website, where it can then be installed with a single command. This command-line tool allows us to manage the versions of your package, examine dependencies, and even set up special scripts for our project.

Express

Express is a Node JS web application framework, it offers a variety of features for creating both web and mobile applications. A single page, multipage, or hybrid web application can be created with it. It is a layer added to Node JS that aids in managing servers and routes [12].

Express was developed to make it simple to create APIs and web apps. It cuts the amount of coding work required practically in half while still producing effective web and mobile applications. Another benefit of utilizing express is that it is written in Javascript, which is simple to learn. Using Node JS with Express helped us to develop our backend in a very short time.

```
const path = require("path");
const morgan = require("morgan");
const express = require("express");
const cookieParser = require("cookie-parser");
const cors = require("cors");
const router = require("./router/router");

const app = express();

app.use(morgan("dev"));

app.use(express.json()); //parse data from body
app.use(cookieParser());

app.use(
  cors({
    credentials: true,
    origin: "http://localhost",
    sameSite: "none",
  })
);

//API Routes
app.options("*");
app.use("/api/v1", router);

module.exports = app;
```

Listing 3.1: app.js

```
const app = require("./app");
const dotenv = require("dotenv");
dotenv.config();

// Server connection
const port = 3000;
```

```
app.listen(port, () => {
  console.log("Listening to port", port);
});
```

Listing 3.2: server.js

Amazon MySQL

MySQL is the world's most popular open source relational database and Amazon RDS makes it simple to deploy, manage, and grow MySQL deployments on the cloud. We can quickly and affordably install highly scalable MySQL servers with resizable hardware capacity using Amazon RDS. Amazon takes care of time-consuming database maintenance duties like backups, software patches, monitoring, scaling, and replication, Amazon RDS for MySQL gives us more time to concentrate on developing applications [1].

3.2 Project Overview

The following sections are dedicated to the illustration and explanation of the project by means of providing adequate diagrams which should contribute to a high level understanding of the architecture alongside the used classes. Starting with the *Use case diagram* requirements are created to ensure the platform fulfills its' the intended functions. After that an *App flow diagram* is shown to ensure the understanding of how the Application is used from the perspective of a user. Following that diagram is the *Architecture diagram* that illustrates the functionalities and communication between the frontend and backend. In addition to this the *Class diagram* shows an in-depth view of the working classes of the backend due to these being the components that do the most work. With the help of a *Database diagram* existing tables are shown to help understand the database.

3.2.1 Use case diagram

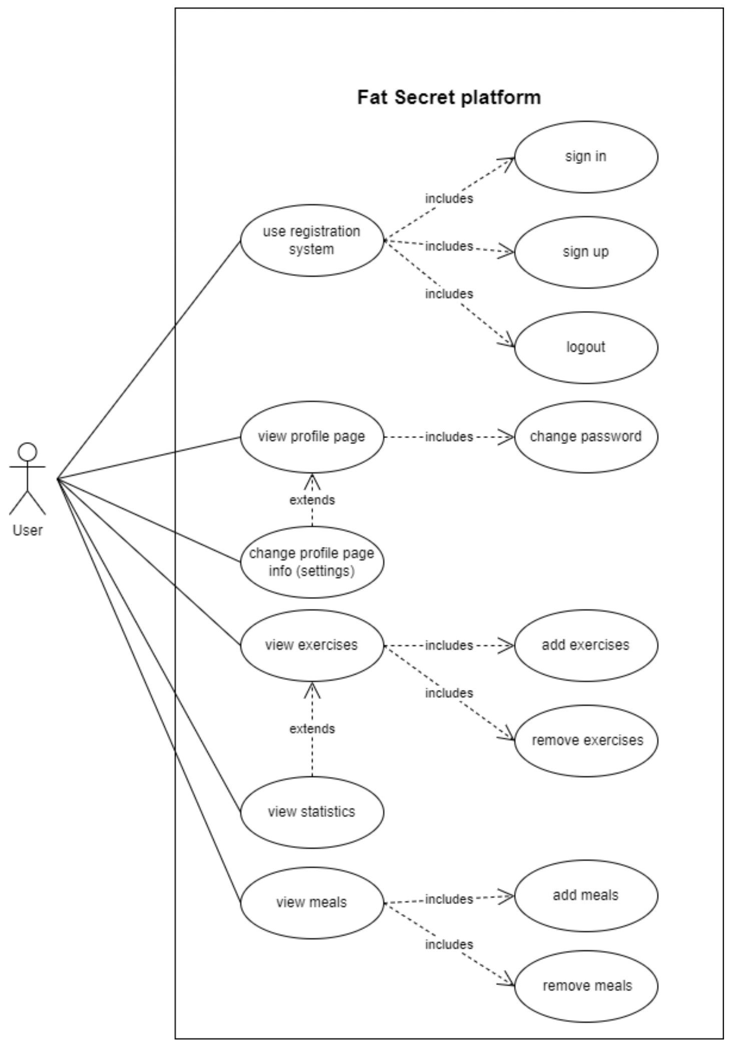


Figure 3.1: Use case diagram

Within the use case diagram you can see the projects' requirements. A user should be able to use a registration system that allows for the signing up of new accounts and their subsequent signing in, followed up by an eventual logout when the user is done. A user should be able to view their own profile page and if necessary change their password easily. Within a settings page all other info of the user should be changeable. A page should be provided where the user can add and remove meals from their provided meals table. Similar to viewing meals the user should be able to view all the exercises they have completed and add new or remove old ones. An extra page that extends the "view exercises" use case shows the progression of the user by displaying the users statistical information within a graph that shows the amount of burnt calories for each day. The final requirement is the ability of the user to view all eaten meals and have the option to add new or remove ones.

3.2.2 App flow

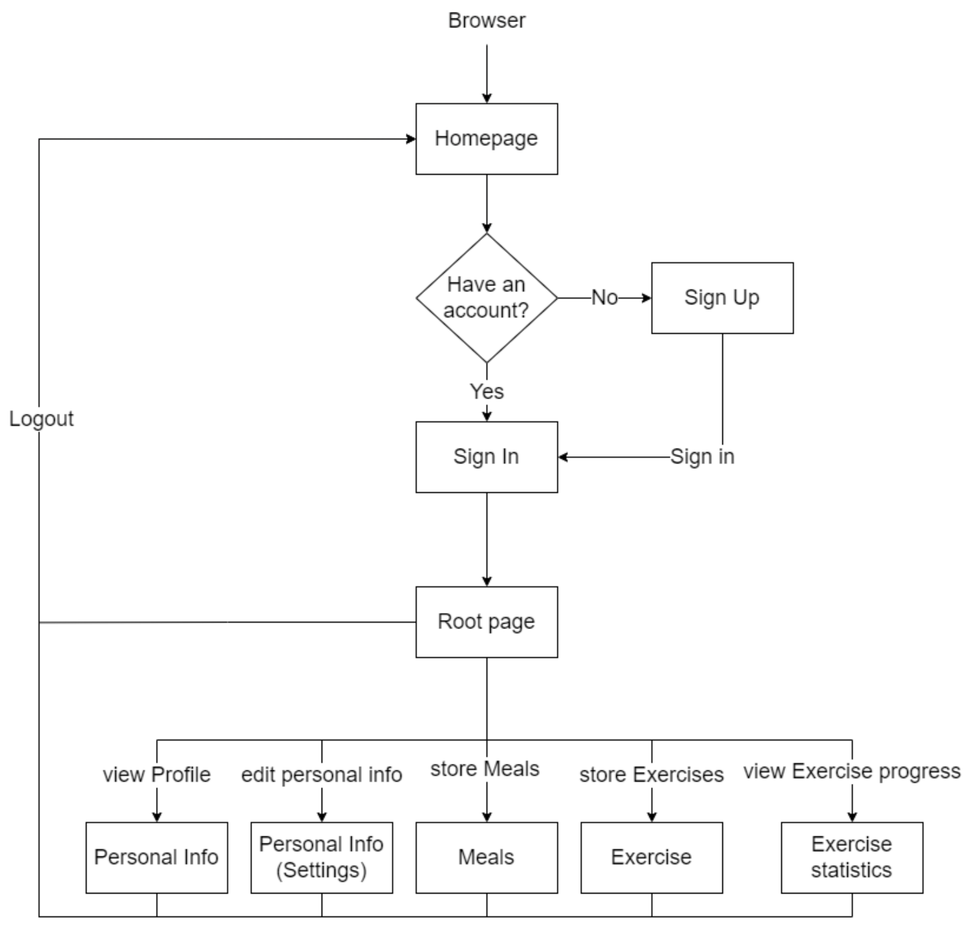


Figure 3.2: App flow diagram

The App flow diagram shown above describes the users' behaviour by connecting all possible pathways of the application. It acts as a map by showing how to get from each page to any other. Since our program uses an intuitive design, achieving an easy and simplistic understanding of the program is guaranteed. Using a browser web client the user can access the *homepage* of the application. Here he/she is prompted to get started. A user can directly sign into an account if one exists. In the case of a user not already having an account there is the option to *Sign Up*. As soon as the user selects this option a pop-up window opens that asks for the users information which will be stored and used within the app itself. The needed information is an email address, a password, the date of birth, the sex, the height and the weight of the user. Once this is completed an account is created and registered within the database. Following the a successful *Sign*

In process the user is directed to the "root page" section of the application. By default this is the *Personal Info* page. Here the user can view the accounts profile information and if needed change the current password. By accessing the *Personal Info (Settings)* page, which is symbolised with a cog within the app, the user can update false or old information. Data such a first or last name, sex and weight can be updated easily. The *Meals* page

3.2.3 Architecture diagram

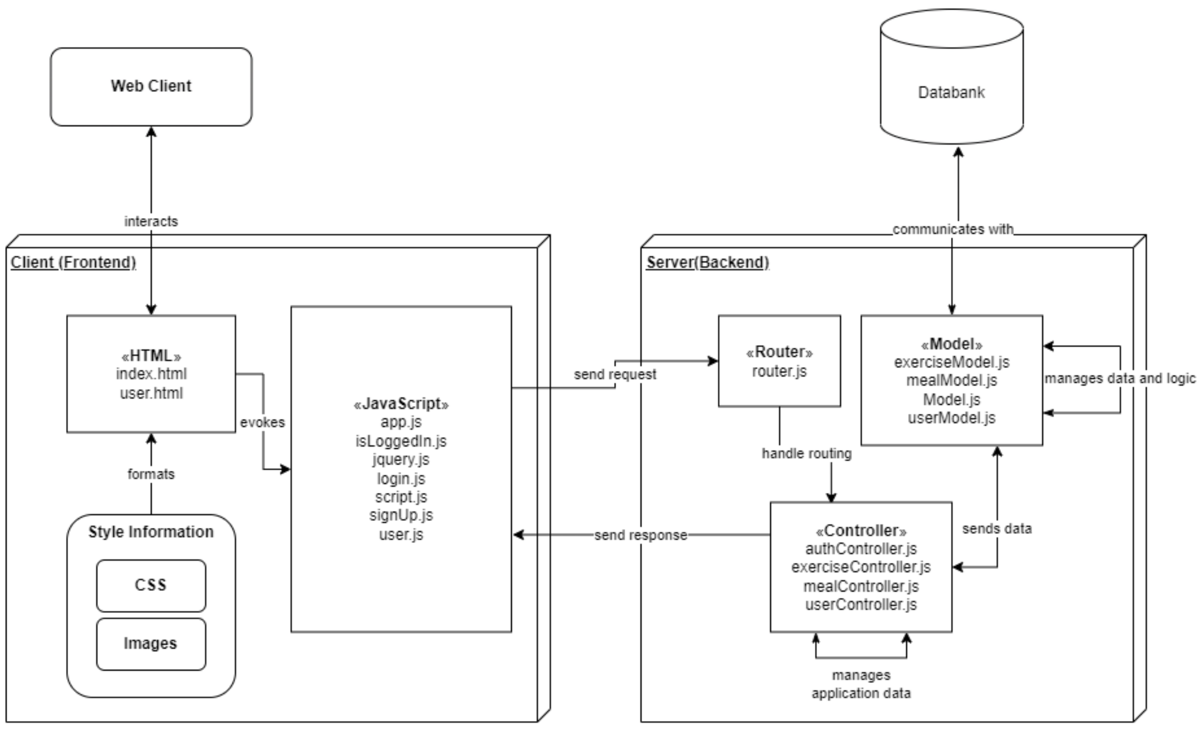


Figure 3.3: Architecture diagram

The figure above depicts the Architecture diagram which is used as a high level overview of the project. The design abides to the traditional understanding of the implementation of a frontend/backend and shows the interactions between these two main bodies of working. The frontend encapsulates all components intended to be seen by a client. Once a user accesses the frontend via a web browser the HTML files will serve as the intermediate between the users interactions and the functioning of the program. These HTML files are formatted using the *Style Information* component which encloses the necessary CSS files and images which are displayed onto the users view. As soon as a user interacts with a displayed button on the webpage an according *JavaScript* script is evoked. This script communicates with the *backend* by sending the request to exchange or get data. This request is sent to the *Router*, which handles and allocates the routing to the needed destination. In this sense it acts as an intermediate between the backend and frontend for receiving requests. Instead of letting the user change data within the server directly the data is controlled within the *Controller* which then communicates with the *Model* using the MVC-pattern (see Chapter 3.4.1 Model-View-Controller for more precise information). Once data is changed within the database the response is communicated back through the *Model* and then through the *Controller*. The *Controller* sends the response to the acting *JavaScript* scripts which in return updates the users view.

3.2.4 Class diagram

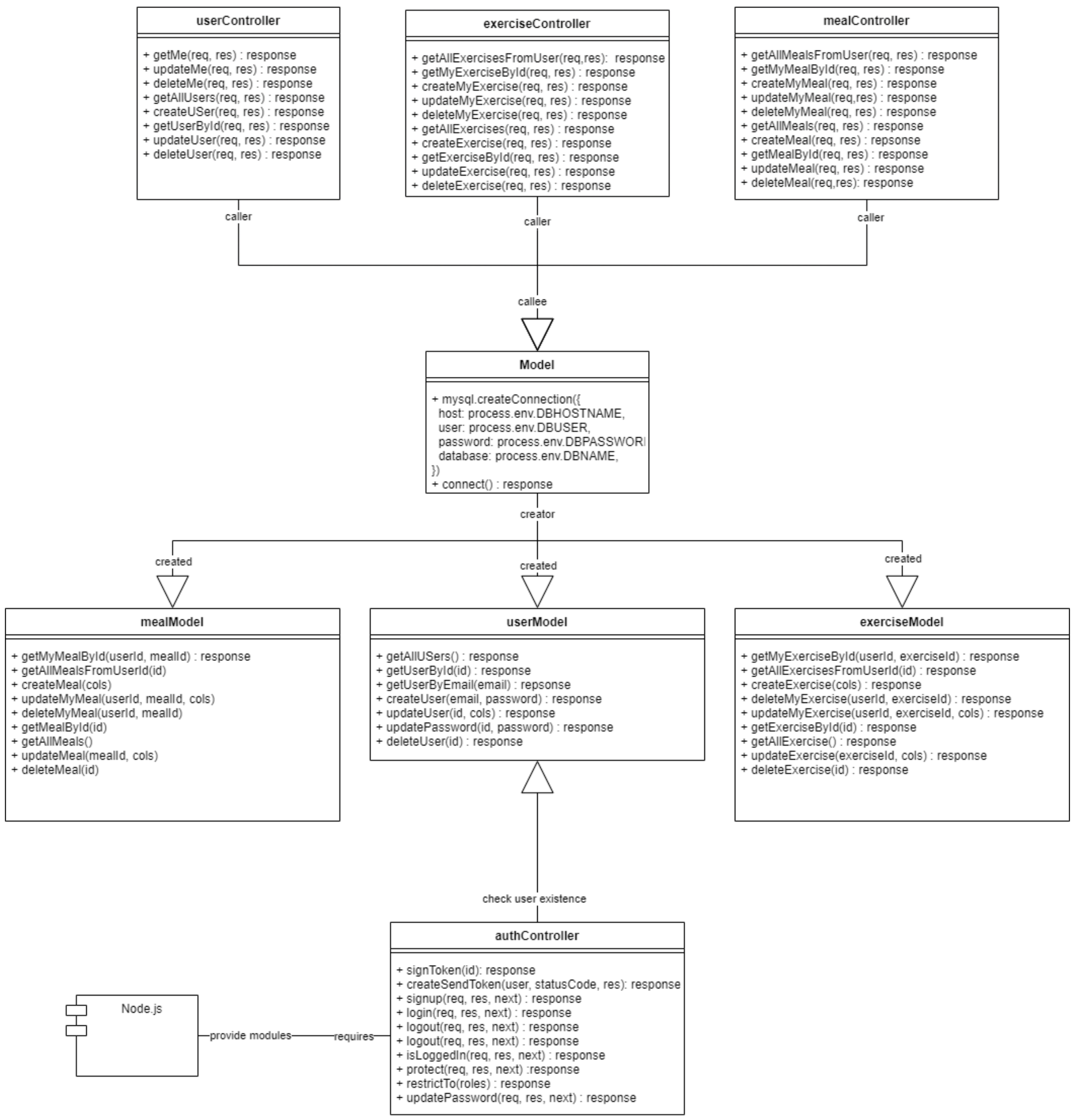
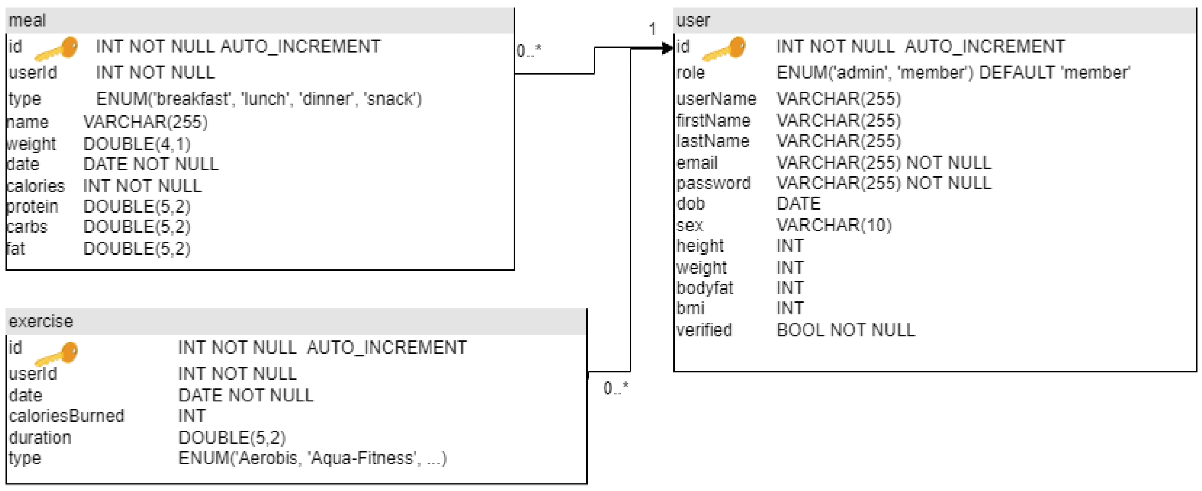

userController
+ getMe(req, res) : response
+ updateMe(req, res) : response
+ deleteMe(req, res) : response
+ getAllUsers(req, res) : response
+ createUSer(req, res) : response
+ getUserById(req, res) : response
+ updateUser(req, res) : response
+ deleteUser(req, res) : response
exerciseController
+ getAllExercisesFromUser(req,res): response
+ getMyExerciseById(req, res) : response
+ createMyExercise(req, res) : response
+ updateMyExercise(req, res) : response
+ deleteMyExercise(req, res) : response
+ getAllExercises(req, res) : response
+ createExercise(req, res) : response
+ getExerciseById(req, res) : response
+ updateExercise(req, res) : response
+ deleteExercise(req, res) : response
mealController
+ getAllMealsFromUser(req, res) : response
+ getMyMealById(req, res) : response
+ createMyMeal(req, res) : response
+ updateMyMeal(req,res) : response
+ deleteMyMeal(req, res) : response
+ getAllMeals(req, res) : response
+ createMeal(req, res) : repsonse
+ getMealById(req, res) : response
+ updateMeal(req, res) : response
+ deleteMeal(req,res): response
caller
caller
caller
callee
Model
+ mysql.createConnection({
host: process.env.DBHOSTNAME,
user: process.env.DBUSER,
password: process.env.DBPASSWORI
database: process.env.DBNAME,
})
+ connect() : response
creator
created
created
created
mealModel
+ getMyMealById(userId, mealId) : response
+ getAllMealsFromUserId(id)
+ createMeal(cols)
+ updateMyMeal(userId, mealId, cols)
+ deleteMyMeal(userId, mealId)
+ getMealById(id)
+ getAllMeals()
+ updateMeal(mealId, cols)
+ deleteMeal(id)
userModel
+ getAllUSers() : response
+ getUserById(id) : response
+ getUserByEmail(email) : repsonse
+ createUser(email, password) : response
+ updateUser(id, cols) : response
+ updatePassword(id, password) : response
+ deleteUser(id) : response
exerciseModel
+ getMyExerciseById(userId, exerciseId) : response
+ getAllExercisesFromUserId(id) : response
+ createExercise(cols) : response
+ deleteMyExercise(userId, exerciseId) : response
+ updateMyExercise(userId, exerciseId, cols) : response
+ getExerciseById(id) : response
+ getAllExercise() : response
+ updateExercise(exerciseId, cols) : response
+ deleteExercise(id) : response
check user existence
authController
+ signToken(id): response
+ createSendToken(user, statusCode, res): response
+ signup(req, res, next) : response
+ login(req, res, next) : response
+ logout(req, res, next) : response
+ logout(req, res, next) : response
+ isLoggedIn(req, res, next) : response
+ protect(req, res, next) :response
+ restrictTo(roles) : response
+ updatePassword(req, res, next) : response
Node.js
provide modules
requires

Figure 3.4: Class diagram

The class diagram shown above depicts how the acting classes of the program interact with each other. According to the MVC-pattern (see chapter 3.4.1 Model-View-Controller) the data being transferred to the backend should not be directly changed by frontend interactions. Thusly this pattern was implemented by making *Controller* classes, which interact with a *Model* interface instead of directly changing data within

the database. At the very top of the diagram are the three *Controller* classes: *userController*, *exerciseController* and *mealController*. Each class has methods that can either be evoked by only an admin or by users. Starting here the diagram will be described in a top to bottom manner. The *userController* has in total eight methods, the first four of which can only be used by a user while the other four can be evoked by an admin for administrational purposes. The methods fulfill all functions needed by a user from their point of view such as getting, updating and deleting data from the users profile page, while additional methods allow an admin to make changes to a users profile as well. As an example: users can update their own profile page using the method *updateMe(req, res)*. As an admin you can use the *method updateUser(req, res)* to update a users' profile. The other two controllers *exerciseController* and *mealController* work in a similiar manner. The *exerciseController* is responsible for all things related to making changes in the exercise category such as getting, deleting and updating a users' exercises in the existing exercise table. This class uses ten methods to implement this functionality, the first five of which are again reserved for the user while the other five are for the admin. The *mealController* has ten methods that work in the same way as the *exerciseController*. The main difference is that it operates solely with everything related to the category of meals instead of exercises.
All of the mentioned controller classes call the *Model* class which acts as an interface between controller classes and models class. Here a connection to the database is established first created and then established. This interface class creates for each controller class a corresponding Model class which interacts directly with the database. As an example: when the *mealController* evokes a method its' destination is the *mealModel*, which as a kind of middleman interacts with the database using methods that fulfill the true function of the controller class.
In addition to the other controller classes the *authController* (i.e. authentification controller) is responsible for the sign-up, sign-in and JWT token (see chapter 3.4.2) system. Here users are created during the sign-up procedure and subsequently stored within the database through the *userModel*. During the sign-in process it checks if the given user email address exists and if the given password is correct. To complete this procedure this class uses modules provided by Node.js.

3.2.5 Database diagram



meal	
id	INT NOT NULL AUTO_INCREMENT
userId	INT NOT NULL
type	ENUM('breakfast', 'lunch', 'dinner', 'snack')
name	VARCHAR(255)
weight	DOUBLE(4,1)
date	DATE NOT NULL
calories	INT NOT NULL
protein	DOUBLE(5,2)
carbs	DOUBLE(5,2)
fat	DOUBLE(5,2)

exercise	
id	INT NOT NULL AUTO_INCREMENT
userId	INT NOT NULL
date	DATE NOT NULL
caloriesBurned	INT
duration	DOUBLE(5,2)
type	ENUM('Aerobis, 'Aqua-Fitness', ...)

user	
id	INT NOT NULL AUTO_INCREMENT
role	ENUM('admin', 'member') DEFAULT 'member'
userName	VARCHAR(255)
firstName	VARCHAR(255)
lastName	VARCHAR(255)
email	VARCHAR(255) NOT NULL
password	VARCHAR(255) NOT NULL
dob	DATE
sex	VARCHAR(10)
height	INT
weight	INT
bodyfat	INT
bmi	INT
verified	BOOL NOT NULL

Figure 3.5: Database diagram

The database diagram shows the created tables that store data about the users, about their meals and their exercises. The meal table stores data about each users' meals. To do this the variable *userId* acts as a foreign key that refers to the user tables' *id* primary key. By doing so each you can link each created meal to a user. Apart from this the meal table includes all necessary data to keep track of a diet such as carbohydrates, proteins and fats. Other important data like the name, weight and when the meal was consumed is also stored.
Taking a look at the user table you can see that it holds all information the platform needs to create a sensible profile. Variables such as the names, sex, body features and accessing information (email, password) are stored.
Within the exercise table data about the exercises such as duration, type and date is stored.

3.3 Website structure

3.3.1 Structure of website

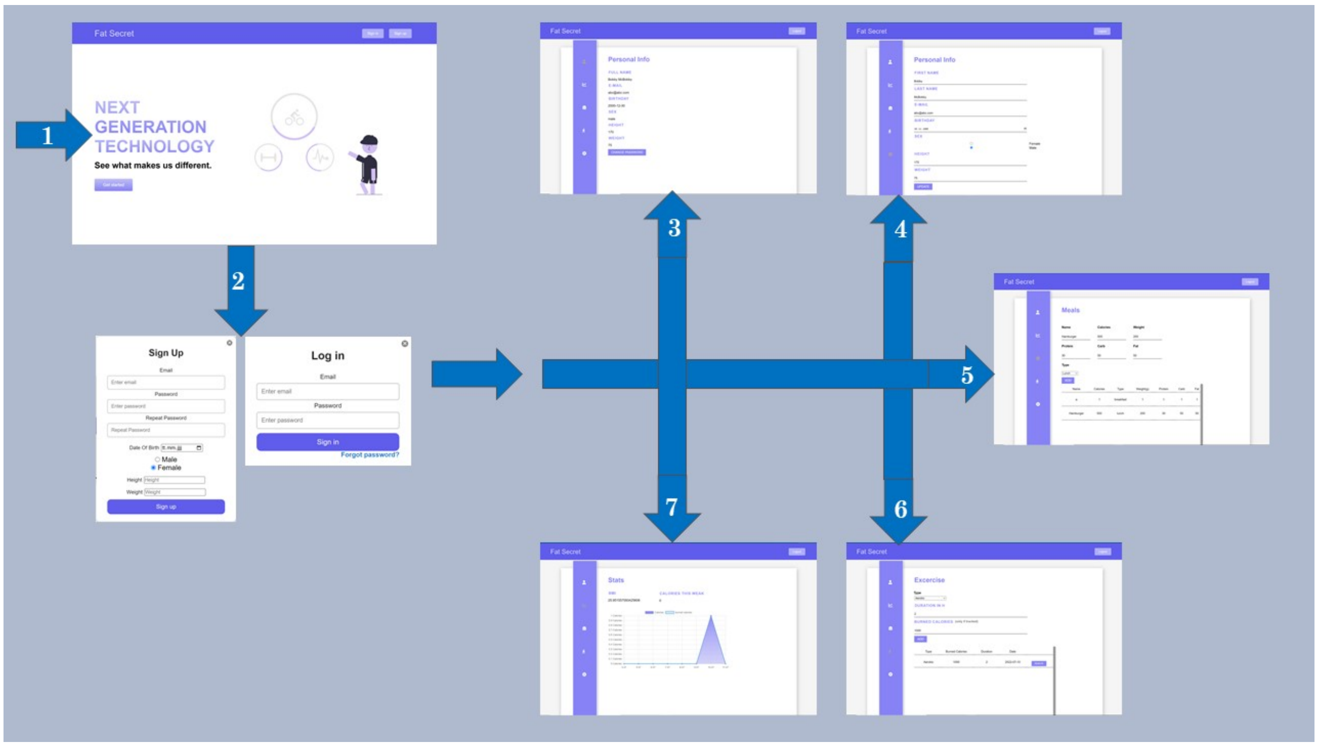
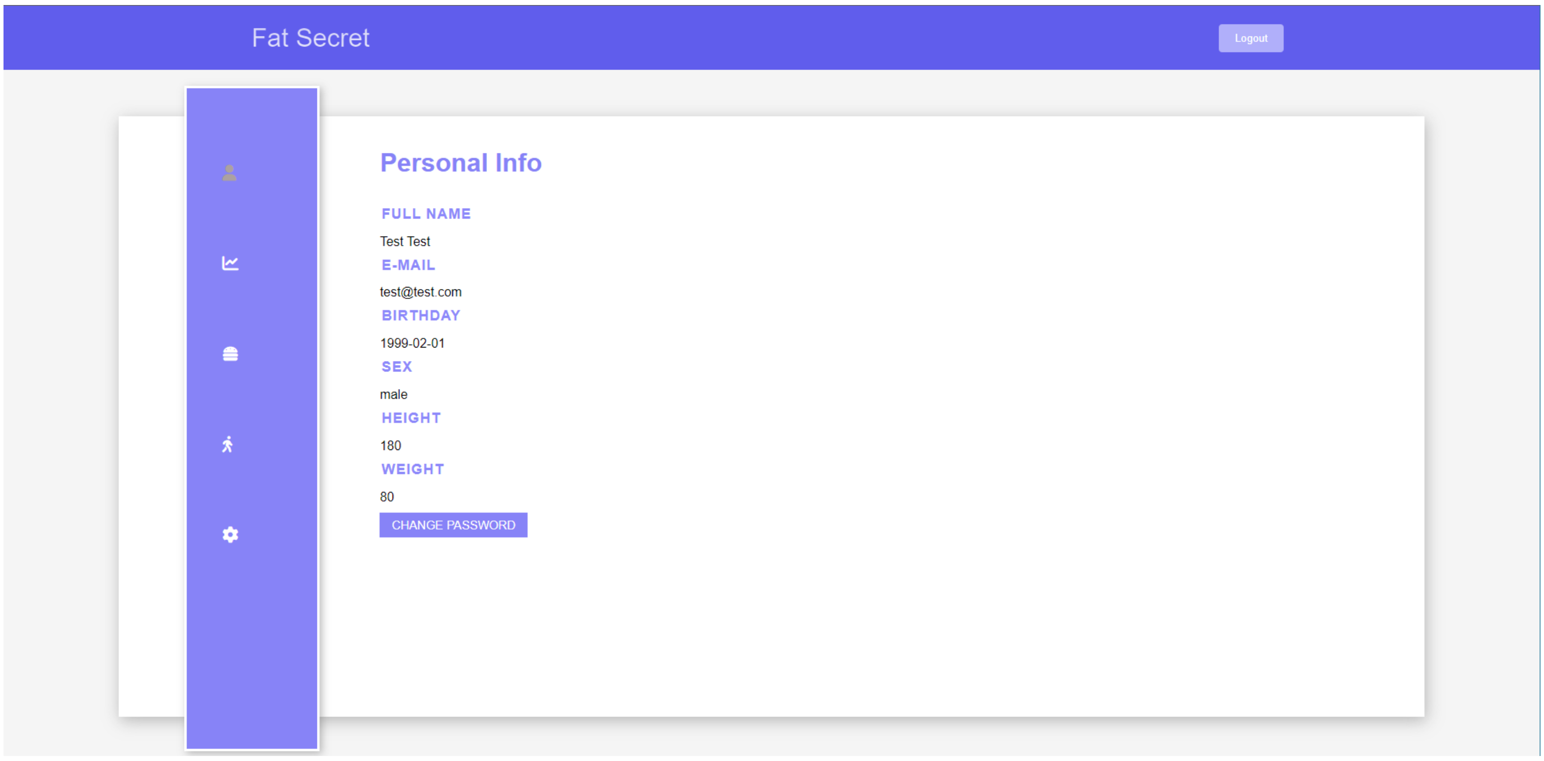


Figure 3.6: Website structure

In total the website consists of six pages and two pop-ups. Follow the numbered arrows in the figure above to see the structure and setup of these pages. (1) When opening the website you are first of all confronted with the homepage. Once either the sign-in or sign-up button is selected a corresponding pop-up will be shown. (2) Here you can fill out the forms with the appropriate information and continue to the next pages. All of these pages can easily be located using the icons on the left side of each page. These icons represent the functions of the pages. (3) The profile page lists the all known user information: full name, e-mail address, sex, height and weight with the option to change the users password at the bottom. (4) The settings page contains the same information with the important difference being that the user is able to interact with the fields. This can be done to update the profile via the "update" button at the very bottom of the page. (5) The meals page has several input fields on the top half of the page. These are used to fill in the information about the meal that is to be added: name, weight, amount of calories, proteins and fats. There is also the option to categorise the meal into: lunch, breakfast and dinner. Below the input fields is the table that displays all the added meals. (6) The exercise page has three fields to interact with. Here you can input the duration of an exercise, the type of exercise and the amount of burned calories. Below the input fields a table displays all user exercises that have been added. (7) The last page is the statistics page. Here the user is graphically shown the amount of calories burned each day alongside the BMI calculation result.

3.3.2 User Page



FULL NAME
Test Test
E-MAIL
test@test.com
BIRTHDAY
1999-02-01
SEX
male
HEIGHT
180
WEIGHT
80

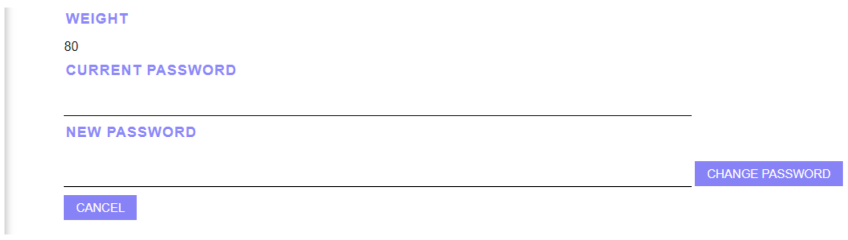
Figure 3.7: Personal Info

After the login the user is Presented with the userpage. The Page is Structured in different sections. the icons on the left side are indicating which specific topic a section is covering. The icons are from the free kit from the website fontawesome.com. To use these icons there must be a script included in header part of the user.html. The navigation bar has only the Logout button as an interactive part (see 3.7).

The entire user part is an HTML document. After clicking on an icon, the user is not redirected to a new HTML page, only a different part of the document is displayed or output through the user.js document.

3.3.2.1 Personal Info

The first page which get displayed after a login is the Personal Info. For the information the data get pulled from the data bank. This is accomplished through a method in the user.js. The function *renderProfile()* is called when the profile icon is clicked. All buttons get their interaction aspect through the functions *getElementbyId("id")*(the id is defined in the HTML part) and the function *addEventListener("click",...)*.In *renderProfile()* a string gets defined with HTML code and the data from the user (Full name, Email, Birthday,Sex,Height, Weight). So the visible part of the user data Section is fully programmed in JavaScript . The user data is read with the method *getMe()*. *getMe()* is part of the communication between frontend and backend. In addition, a button for changing the password is defined. and when this button is clicked, the function *renderChangePassword()* is called. The function *renderChangePassword()* extends the page with input fields for the old password and the desired new one (see 3.8).


WEIGHT

80

CURRENT PASSWORD

NEW PASSWORD

CHANGE PASSWORD

CANCEL

Figure 3.8: Change Password extension

Also two more buttons gets displayed for changing the password and to cancel the action.

Cancel "closes" the input fields by only showing the HTML part for the button output in this function and not for the input fields.

By clicking on change password a payload gets defined with the input currentPassword and newPassword. The payload gets send do the backend through the function *changePassword(payload)*.

3.3.2.2 Stat Page

The section for the statistical overview is where the user can see his BMI (body mass index) a measurement of someone's weight in relation to their height. for the calculation the user height and weight is read from the database.

$BMI = weight/(height^2/10000)$

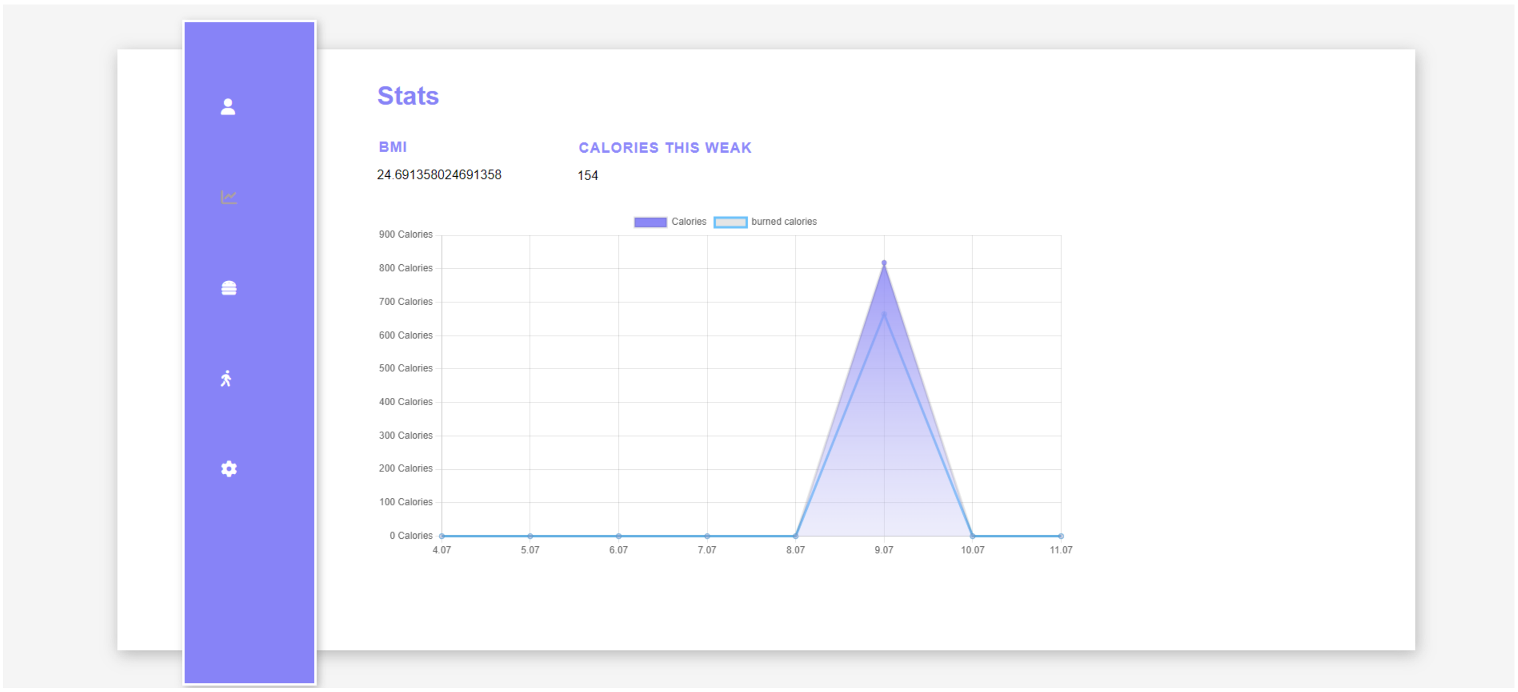


Figure 3.9: Statistics Page

The section also shows a diagram which indicates the calories eaten and calories lost through exercise in the last six days plus the current day. Two labels which describes what the two different graphs stand for are shown above the diagram. The violet graph with a background gradient shows the calories eaten. The blue line shows the calories lost. There is the possibility to deactivate one of those through clicking on one of the labels. The howl style of graph is described in the user.js and not in the user.css. For the diagram we used the chart.js framework which is well documented on the website chartjs.org [2]. To display a chart a canvas has to be defined in HTML. The diagram is then displayed at this defined position.

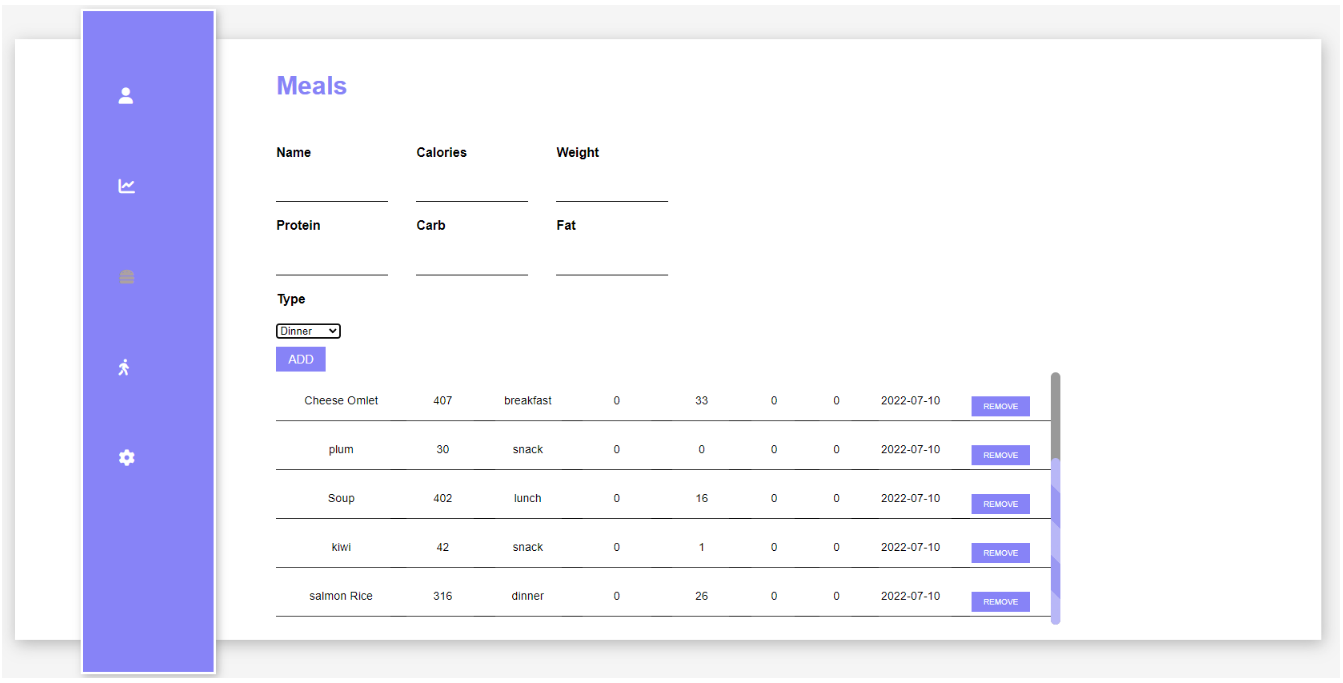
To select the data for the desired period, all meals and all exercises of the user are run through two loops. It is checked whether these are added on the respective date. When there are added at a specific date, all calories are summed up and get saved in an array. This array is so structured that the position 0 stands for the date six days ago and 1 five days ago.

"Calories This Week" is the total value of calories which the user has gained or lost (deficit). The total calories for the meals and exercises are calculated in the same loop were the right data gets selected for the diagram.

The entire HTML part of the "Stats" section is written in JavaScript in the *renderStat* function. This functions gets called when the user clicks on the Stat icon shown in 3.9.

3.3.2.3 Meals Page

The idea behind this section is that the user can add an eaten meal to the database. A meal consists of a name the calories, proteins, carbs, fat which it contains and the weight. This variables get stored in the database with the date and selected type (Breakfast, Lunch, Dinner, Snack). Although we only use Calories and Date, we have added the other input options for potential extension of the program. Section is shown in 3.10



Cheese Omlet	407	breakfast	0	33	0	0	2022-07-10	REMOVE
plum	30	snack	0	0	0	0	2022-07-10	REMOVE
Soup	402	lunch	0	16	0	0	2022-07-10	REMOVE
kiwi	42	snack	0	1	0	0	2022-07-10	REMOVE
salmon Rice	316	dinner	0	26	0	0	2022-07-10	REMOVE

Figure 3.10: Meals Page

The input fields, type selection and add button are written in user.html, the bottom side where all meals get listed with the remove button to delete a specific meal is generated in JavaScript *renderMeals*. The listing is possible because with CSS. In user.css is described that all meals should be displayed in a grid and all eight elements should be placed automatically. This means that all elements of a meal are placed next to each other.

The add button adds an Meal to the database. All input values are read in with *document.getElementById().value* and the date is created. Then a if statement checks whether a name has been entered for the meal. If not, a note is left in the console. If a name has been entered the payload will be created with the variables. If no value has been entered for calories, weight, protein, carbs and fat, than these values become 0 in the payload. The *createMeal* function is then called. This is there for the communication with the backend.

The scrollbar appears after a certain height in pixels is reached by the listed meals. And ensures that the elements entered do not exceed a certain level. Therefore a div (class

= scroll-container-meal) was added around the listing which is described in the CSS file. Here is the definition of height important which describes how high the defined area is in which the list is displayed as well as *overflow:scroll*, this describes that after the height gets exceeded a scrollbar appears and this brings with it the possibility of scrolling. *overflow-x:hidden* only says that the x-axis scrollbar is hidden.

The scrollbar was personalized in user.css with the options *::-webkit-scrollbar-[...]*.

The Remove button is created for every Meal with a loop every button is tied together with a mealId. After a click on the remove button the function *deleteMeal(mealId)* is called. *deleteMeal(mealId)* is used to communicate with the backend. The Path for the restAPI is defined. After the delete function the renderMeals function is called to update the Page.

```

for (let meal of meals) {
    const deleteMealButton = document.getElementById(‘meal-delete-${
       meal.id}‘);

    deleteMealButton.addEventListener("click", async () => {
      const mealId = deleteMealButton.getAttribute("data-mealId");
      await deleteMeal(mealId);
      await renderMeals();
    });
  }
```

Listing 3.3: loop to create remove button in user.js

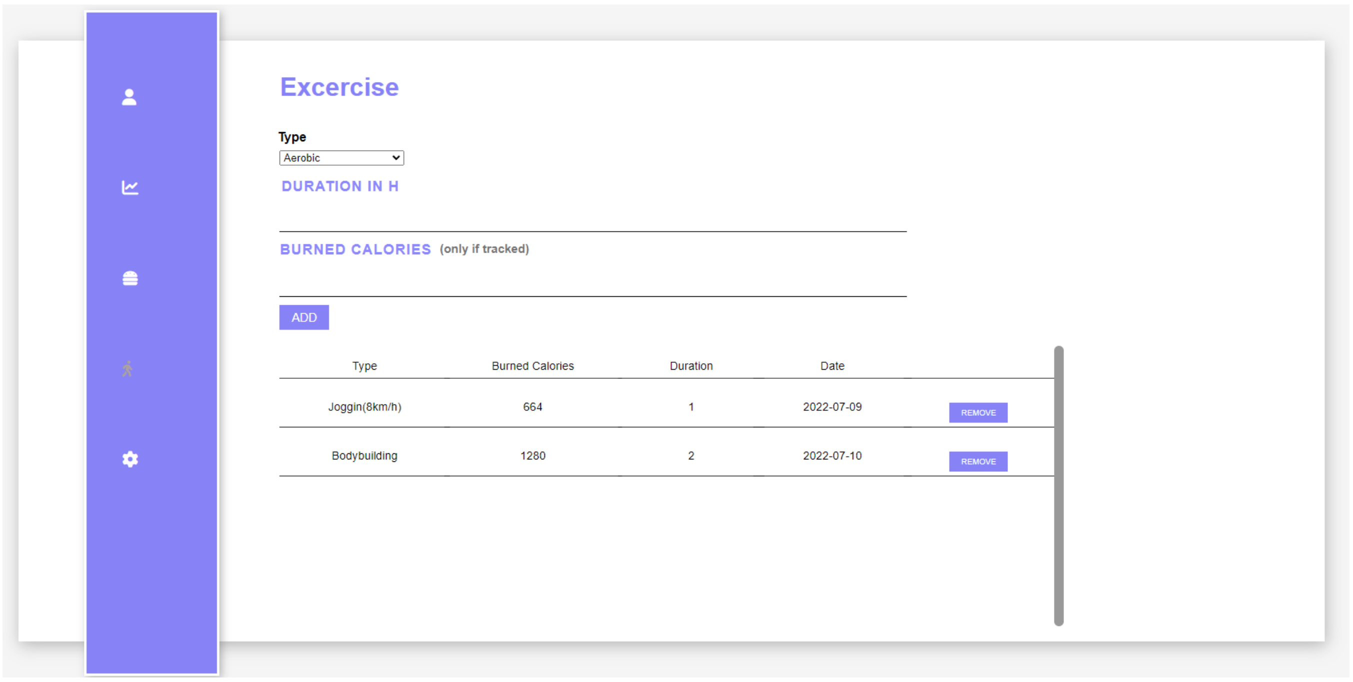
3.3.2.4 Exercise Page

The Idea behind the exercise section is that the user can add completed exercises to his user profile. So that the user is able to track the burned/lost calories 3.11.

The section consists of three inputs. The type of exercise which is a selection, the duration in hours and an optional input field burned calories. The input for the burned calories is for the case that the user wore a smartwatch during the exercise, which measured the calories lost. If the user has entered a value in this field, that value will be saved in the database, but if the field is empty, the calories lost will be calculated.

For the calculation the unit MET (metabolic equivalent task) is used to compare energy expenditure in different physical activities. 1 MET corresponds to a turnover of 3.5 ml of oxygen per kilogram of body weight per minute in men and 3.15 ml/kg/min in women. Moderate physical activity such as walking or slow dancing requires energy consumption of 3-4 METs, strenuous sport such as soccer 8 METs - fast running at 16 km/h even brings together 16 METs.

The type section for an exercise which is in user.html passes the name and the MET value for the calculation.



Type	Burned Calories	Duration	Date	
Joggin(8km/h)	664	1	2022-07-09	REMOVE
Bodybuilding	1280	2	2022-07-10	REMOVE

Figure 3.11: Exercise Page

Example: Aerobic has a MET value of 7.3. The calculation would be $7.3 * weight * duration = burnedcalories$

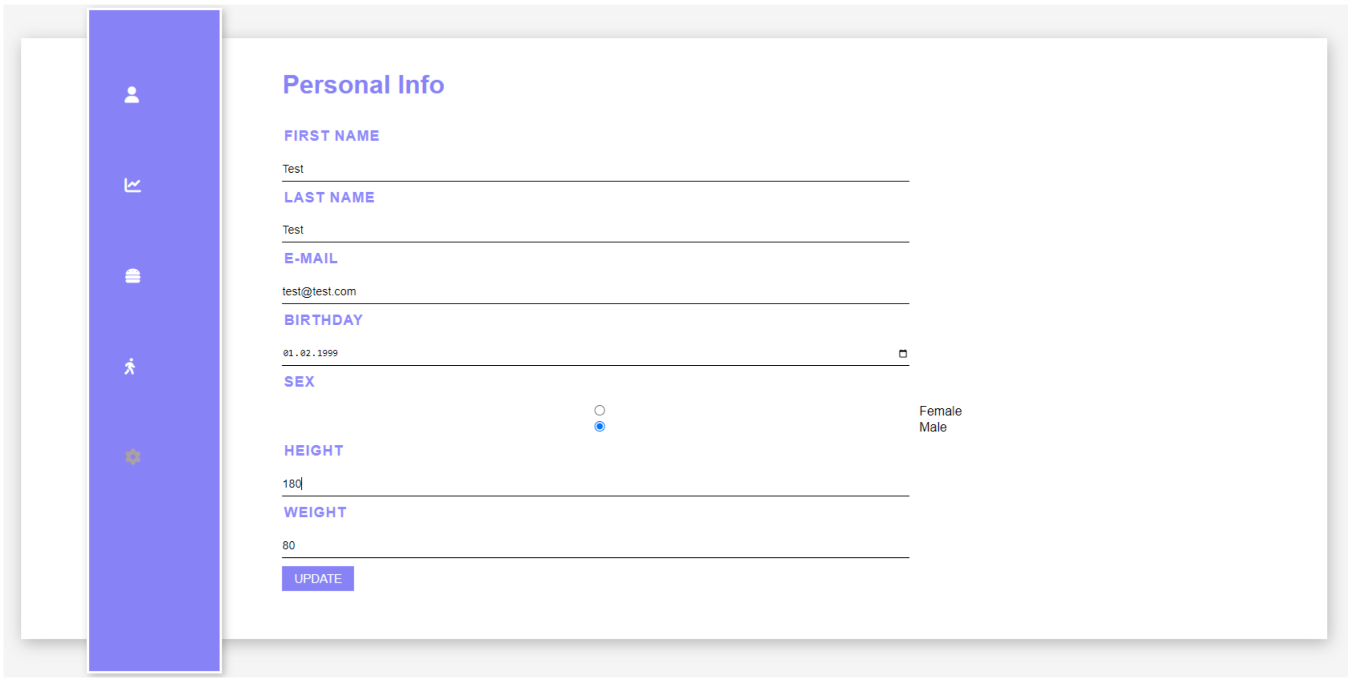
How an exercise is added and how it is displayed is the same as for the meals. Only with fewer variables and a duration must be entered instead of the name. The name corresponds to the type.

3.3.2.5 Change Profile Data

The last section is for changing the data from the user. The user has to click on one of the input fields to delete the previous text and enter the desired new data. In the case of gender, however, only the selection of the other is required. The user has the possibility to change his first name, last name, gender, e-mail address, date of birth as well as height and weight as seen in [3.12].

As seen in previous methods, a function *renderEdit* is used to display this page. In this function, the user data is read from the database and the output HTML text is defined. Important for the display is also the definition by an if statement, which controls whether the user is male or female. Depending on the which case occurs, the output of the gender is then displayed.

To change the data, the update button must be clicked. Then *updateUser()* is called. This function creates the payload with all user data, which are read from the input fields and written to the database through the restAPI.


Personal Info

FIRST NAME
Test

LAST NAME
Test

E-MAIL
test@test.com

BIRTHDAY
01.02.1999

SEX
Female
[✓ Male]

HEIGHT
180

WEIGHT
80

Figure 3.12: Change Profile Data

3.4 Implementation of the Data Transmission

To understand how data is being transferred through the application, first we need to show how we structured and developed our project. Our project is developed using a design pattern called Model-View-Controller.

3.4.1 Model-View-Controller

MVC is an architectural or design pattern applied to software engineering. While not a rigid guideline, this pattern helps us narrow our attention to one particular aspect of our program at a time. The primary objective of MVC is to divide complex programs into distinct portions, each with a distinct function, as we will demonstrate in our project. Let's first examine what each feature of the pattern offers [7].

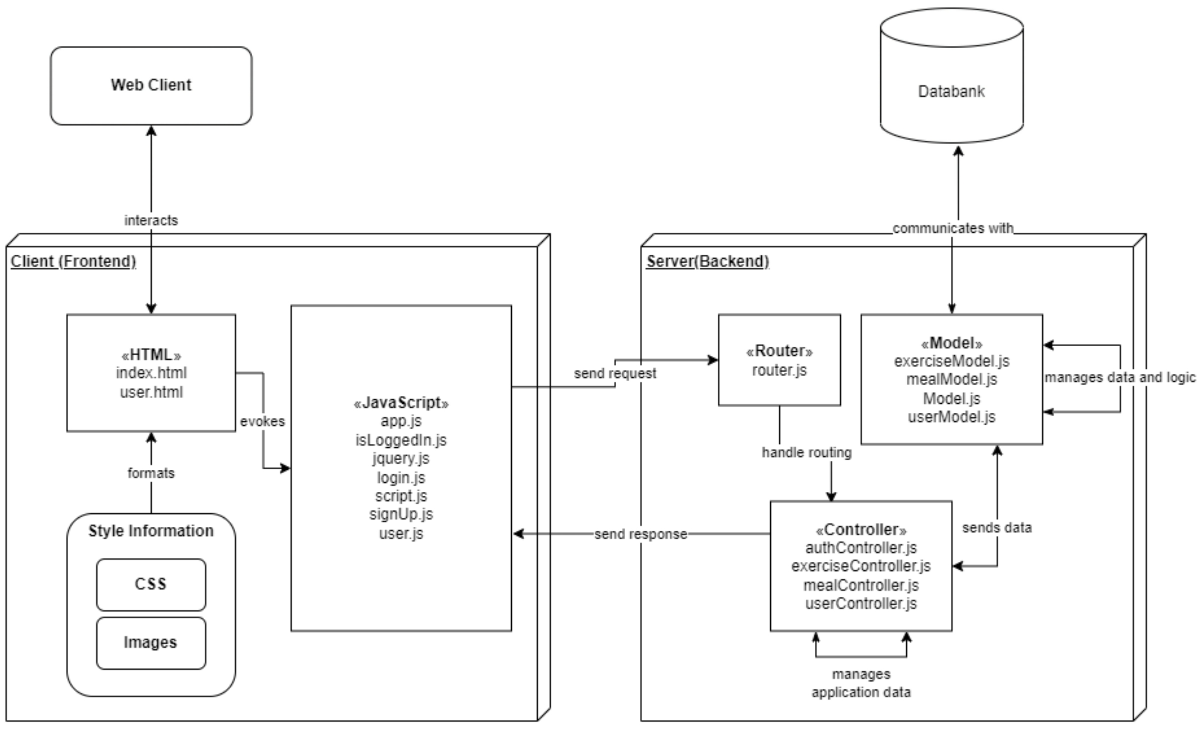


Figure 3.13: Model-View-Controller design pattern

Model

A Model is, as its name suggests, a design or structure. With MVC, a component of the program that communicates with the database is defined by the model, which also dictates how a database is organized. The properties of a user that will be stored in our database will be defined here.

In _dbHelper.js in Code 3.4 file we connect to our database webapp and create our tables: user, meal, exercise. Because we only need to connect to our database and create our tables once at the beginning or right after database schema changes, we only need to execute the script once in such an event.

```
const mysql = require("mysql");
const dotenv = require("dotenv");

dotenv.config();

// Database connection
const db = mysql.createConnection({
  host: process.env.DBHOSTNAME,
  user: process.env.DBUSER,
  password: process.env.DBPASSWORD,
});

db.connect(function (err) {
  if (err) throw err;
  console.log("Connected!");
});

const createDatabase = function () {
  const query = "CREATE DATABASE IF NOT EXISTS webapp;";

  db.query(query, (err, data) => {
    if (err) {
      console.log(err);
    } else {
      console.log("Database created!");
    }
  });
};

const connectToDB = function () {
  const query = "USE webapp;";
  db.query(query, (err, data) => {
    if (err) {
      console.log(err);
    } else {
      console.log("Use database webapp!");
    }
  });
};

const createUserTable = function () {
  const query = `CREATE TABLE IF NOT EXISTS user (
          id INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
          role ENUM('admin', 'member') DEFAULT 'member',
          username varchar(255) NOT NULL,
          firstName varchar(255),
          lastName varchar(255),
          email varchar(255) NOT NULL,
          password varchar(255) NOT NULL,
          dob DATE,
          sex varchar(10),
          height INT,
          weight INT,
          bodyfat INT,
          bmi INT,
          verified BOOL NOT NULL
      )
      ;`;

  db.query(query, (err, data) => {
    if (err) {
      console.log(err);
    } else {
      console.log("User table created successfully");
    }
  });
```

```
};

createDatabase();
connectToDB();
createUserTable();
db.end();
```

Listing 3.4: _dbHelper.js

The files userModel.js, exerciseModel.js and mealModel.js are where we create, modify and manage the application database using MySQL query and MySQL APIs from mysql package. The example below (Code 3.5) shows how the creating user APIs in userModel.js is executed.

```
  async createUser(email, password) {
    const query = "INSERT INTO user (email,password) VALUES (?);";

    const values = [email, password];
    return await new Promise((resolve, reject) => {
      this.database.query(query, [values], (err, results, fields) => {
        if (err) {
          reject(err);
        } else {
          resolve(results);
        }
      });
    });
  }
```

Listing 3.5: userModel.js

View

The View is where end users interact within the application. Simply put, this is where all the HTML template files go. The client program is separate Node JS program. We use Node JS simply for application routing (Code 3.6).

```
const router = express.Router();

router.get("/", (req, res) => {
  res.sendFile(path.join(__dirname + "/public/html/index.html"));
});

router.get("/user", (req, res) => {
  res.sendFile(path.join(__dirname + "/public/html/user.html"));
});
  }
```

Listing 3.6: router.js

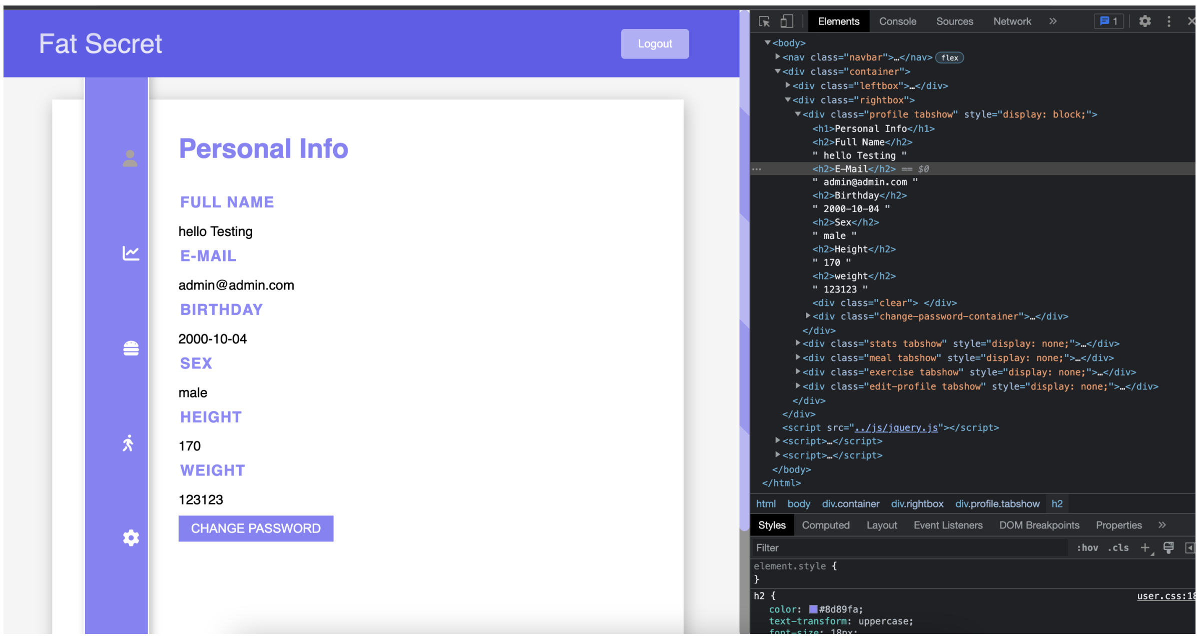
To achieve dynamic, interactive website we use Javascript technique called DOM to manipulate HTML and CSS. For example, the following lines of code will render user's profile data whenever the user click to view his profile (Code 3.7).

```
const renderProfile = async () => {
  const user = await getMe();
  let str = `
  <h1>Personal Info</h1>
  <h2>Full Name</h2>
  ${user.firstName} ${user.lastName}
  <h2>E-Mail</h2>
  ${user.email}
  <h2>Birthday</h2>
  ${user.dob}
  <h2>Sex</h2>
  ${user.sex}
  <h2>Height</h2>
  ${user.height}
  <h2>weight</h2>
  ${user.weight}
  <div class="clear"> </div>
  <div class="change-password-container">
  <button class="btn" id="change-password">Change Password</button>
  </div>

  `;
  const profileElement = document.querySelector(".profile");
  profileElement.innerHTML = "";
  profileElement.insertAdjacentHTML("afterbegin", str);

  const changePWButton = document.getElementById("change-password");
  changePWButton.addEventListener("click", () => {
    renderChangePassword();
  });
};
```

Listing 3.7: user.js



Personal Info

FULL NAME

hello Testing

E-MAIL

admin@admin.com

BIRTHDAY

2000-10-04

SEX

male

HEIGHT

170

WEIGHT

123123

CHANGE PASSWORD

```
<body>
  <nav class="navbar">…</nav> flex
  <div class="container">
    <div class="leftbox">…</div>
    <div class="rightbox">
      <div class="profile tabshow" style="display: block;">
        <h1>Personal Info</h1>
        <h2>Full Name</h2>
        " hello Testing "
        <h2>E-Mail</h2> == $0
        " admin@admin.com "
        <h2>Birthday</h2>
        " 2000-10-04 "
        <h2>Sex</h2>
        " male "
        <h2>Height</h2>
        " 170 "
        <h2>weight</h2>
        " 123123 "
        <div class="clear"> </div>
        <div class="change-password-container">…</div>
      </div>
      <div class="stats tabshow" style="display: none;">…</div>
      <div class="meal tabshow" style="display: none;">…</div>
      <div class="exercise tabshow" style="display: none;">…</div>
      <div class="edit-profile tabshow" style="display: none;">…</div>
    </div>
  </div>
  <script src="../js/jquery.js"></script>
  <script>…</script>
  <script>…</script>
</body>
</html>
```

html body div.container div.rightbox div.profile.tabshow h2

```
element.style {
}
h2 {                                   user.css:18
  color: #8d89fa;
  text-transform: uppercase;
  font-size: 18px;
```

Figure 3.14: User's profile page

Controller

The Controller communicates with the Model and provides the View with functionality and response. When an end user submits a request, the Controller, which works with the Model, receives it. Consider the Controller as a waiter taking care of the orders of the customers, in this instance the View. After that, the waiter, who represents the Model/Database, goes to the kitchen to get food to serve the customers.

For example: The function login in userController.js take care of the login process of the application. The end-user send login credential via browser. This credentials is sent to the server as JSON and is attached in the request body. userController then takes the credential in request body, calls the database to check the validity of the credential. If everything is correct, controller then sends back credential token to end-user. Otherwise it will send back an error code (Code 3.8).

```
const { User } = require("../model/Model");
exports.login = async (req, res, next) => {
  try {
    const { email, password } = req.body;

    //1 Check email and pw exist
    if (!email || !password) {
      return res.status(400).json({
        status: "fail",
        message: "Please provide email and password",
      });
    }

    //2 Check if the user exist and Check if the password is correct
    const user = await User.getUserByEmail(email);

    if (!user || !(await checkPassword(password, user.password))) {
      return res.status(400).json({
        status: "401",
        message: "Incorrect email or password",
      });
    }

    //3 If everything ok , send token to client
    createSendToken(user, 200, res);
  } catch (error) {
    res.status(500).json({
      status: "fail",
      message: error.message,
    });
  }
};
```

Listing 3.8: userController.js

3.4.2 JSON Web Token

For authenticating and authorizing user's credentials, we use technique called JSON Web Token or JWT. JWT is an open standard that allows a client and a server to exchange security-related data. Every JWT has a set of encoded JSON objects, including claims. To ensure that the claims cannot be changed after the token is issued, JWTs are signed using a cryptographic technique [3].

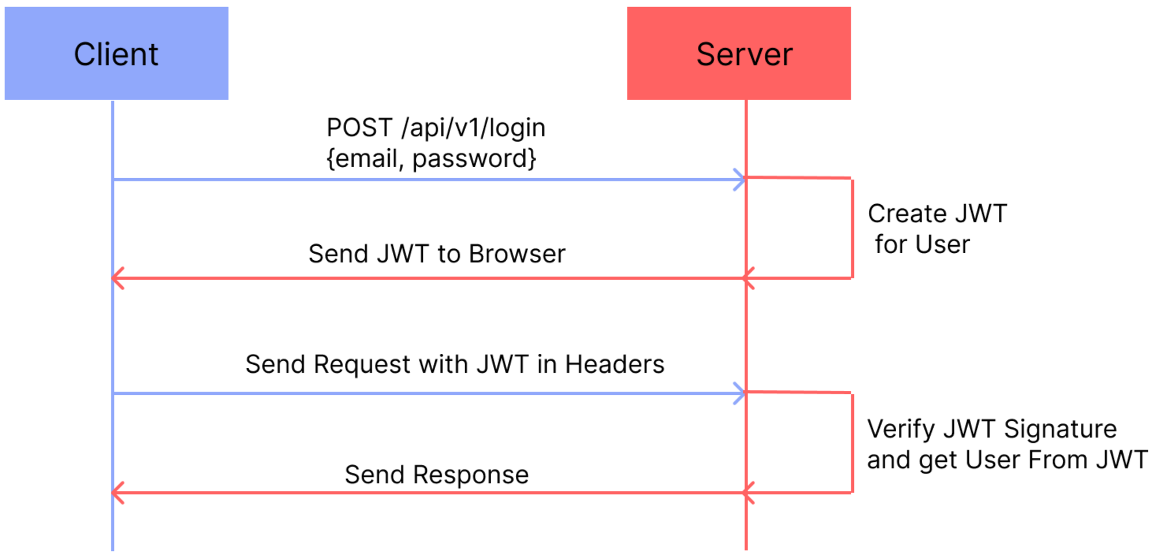


Figure 3.15: How JWT works

After user sent to the server his credentials using email and password. The server then verifies the credentials provided by the user, if the credentials are correct, the server creates the JWT with user's ID and sends it to the browser. These processes are implemented in the functions signToken, createSendToken and login in authController.js. Now whenever the user make a request, which requires authorized permission, the server will validate the JWT in the request headers or cookie. The server searches for the user with user's ID in the token. If everything is correct, the server grants access to the authorized resources (see Code 3.9).

```
const signToken = (id) => {
  return jwt.sign({ id }, process.env.JWT_TOKEN, {
    expiresIn: process.env.JWT_EXPIRES_IN,
  });
};

// Send JWT token in response cookie
const createSendToken = (user, statusCode, res) => {
  const token = signToken(user.id);
  const cookieOptions = {
    expires: new Date(
      Date.now() + process.env.JWT_COOKIE_EXPIRES_IN * 24 * 60 * 60 *
        1000
    ),
    httpOnly: true,
  };

  res.cookie("jwt", token, cookieOptions);

  //remove the password in the output
  delete user.password;

  res.status(statusCode).json({
    status: "success",
    token,
    expires: new Date(
      Date.now() + process.env.JWT_COOKIE_EXPIRES_IN * 24 * 60 * 60 *
         1000
    ),
    data: {
      user,
    },
  });
};

exports.login = async (req, res, next) => {
  try {
    const { email, password } = req.body;

    //1 Check email and pw exist
    if (!email || !password) {
      return res.status(400).json({
        status: "fail",
        message: "Please provide email and password",
      });
    }

    //2 Check if the user exist and Check if the password is correct
    const user = await User.getUserByEmail(email);

    if (!user || !(await checkPassword(password, user.password))) {
      return res.status(400).json({
        status: "401",
        message: "Incorrect email or password",
      });
    }

    //3 If everything ok , send token to client
    createSendToken(user, 200, res);
  } catch (error) {
    res.status(500).json({
      status: "fail",
      message: error.message,
    });
  }
};

```

```
exports.protect = async (req, res, next) => {
  try {
    // 1) Get the token and check if it exists
    let token;
    if (
      req.headers.authorization &&
      req.headers.authorization.startsWith("Bearer")
    ) {
      token = req.headers.authorization.split(" ")[1];
    } else if (req.cookies.jwt) {
      // authen users also by req.cookies send via browser not just req
         .header
      token = req.cookies.jwt;
    }

    if (!token) {
      return res.status(401).json({
        status: "fail",
        message: "You are not logged in!",
      });
    }

    //2) Validate the token
    const decoded = await promisify(jwt.verify)(token, process.env.
       JWT_TOKEN);

    //3 Check if the user still exists
    const currentUser = await User.getUserById(decoded.id);
    if (!currentUser) {
      return res.status(401).json({
        status: "fail",
        message: "The user belonging to this token does no longer exist
          ",
      });
    }

    req.user = currentUser;

    next();
  } catch (error) {
    return res.status(401).json({
      status: "fail",
      message: error,
    });
  }
};
```

Listing 3.9: authController.js

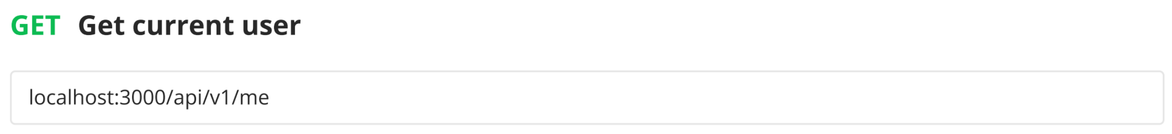
3.5 REST APIs

An application programming interface (API) defines a set of rules on how multiple autonomous systems can connect and communicate with each other. An API uses a representational state transfer architectural style (REST) to follow the given design principles and is then defined as a REST API. Through REST developers get a higher level of flexibility when connecting applications with each other. [8]

The following chapter will explain in detail how the requests work and what purpose they serve. Request details include the method, authorization type, URL, headers, request and response structures and examples. The key-value pairs for request parameters will also be displayed.

3.5.1 Authorized access APIs

When using an application, a user should have the possibility to access his profile. Using a GET request (see figure 3.16) a user can retrieve his current profile. The request uses the bearer token authentication, a security token that can be used by any party that has access to it [5].

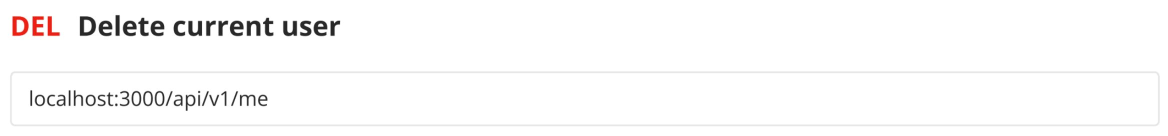


GET Get current user

localhost:3000/api/v1/me

Figure 3.16: GET **Get current user** request

Every user should also have the option of delete the user profile. With the DEL request (see figure 3.17) a user can delete their user profile. This request uses the bearer token authentication.



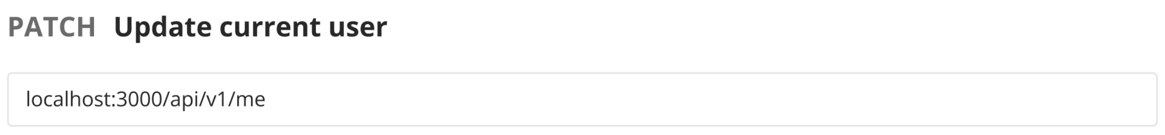
DEL Delete current user

localhost:3000/api/v1/me

Figure 3.17: DEL **Delete current user** request

As shown in chapter 3.4.1, an existing user has multiple attributes like a username, first name, last name, an email address, a password, date of birth etc. Especially in an application for counting calories, the user has to change the profile values to dynamically.

Using the PATCH request **Update current user**, these values can be changed (see Figure 3.18).

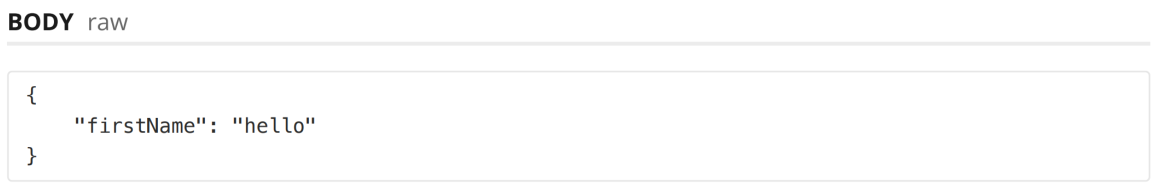


PATCH Update current user

localhost:3000/api/v1/me

Figure 3.18: PATCH **Update current user** request

Figure 3.19 shows the data that is passed within the PATCH request. In this example, the user has changed the attribute **firstName** to **hello**. This request also uses the bearer token authentication.

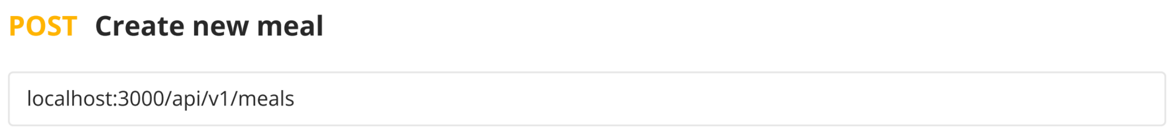


BODY raw

```
{
    "firstName": "hello"
}
```

Figure 3.19: PATCH **Update current user** body request

Furthermore a user also has to be able to create a meal to track calories. A meal consists of the attributes shown in the database diagram in figure 3.5. To create a meal, the POST request **Create new meal** is used. The URI "/meals" is used to address the resources. The POST request uses the bearer token authentication.

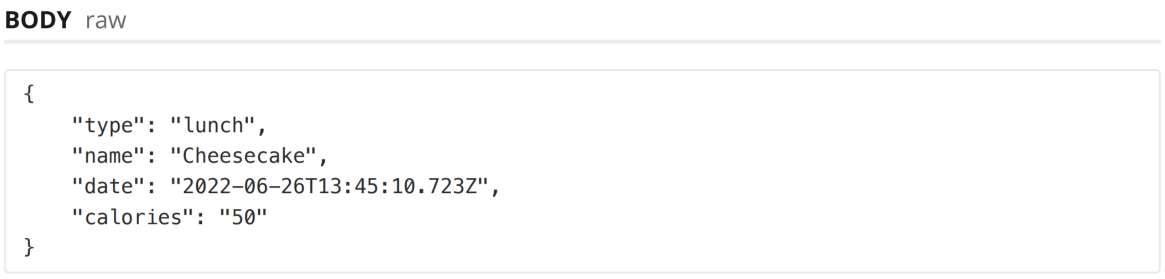


POST Create new meal

localhost:3000/api/v1/meals

Figure 3.20: POST **Create new meal** request

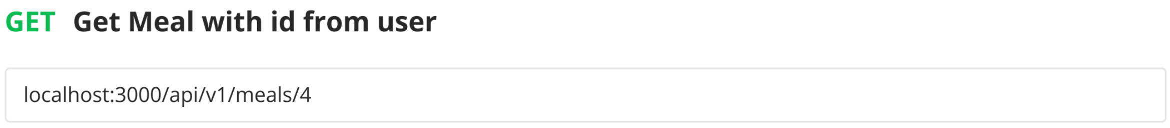
Figure 3.21 shows what was added to each attribute.


BODY raw

```
{
    "type": "lunch",
    "name": "Cheesecake",
    "date": "2022-06-26T13:45:10.723Z",
    "calories": "50"
}
```

Figure 3.21: POST **Create new meal** body request

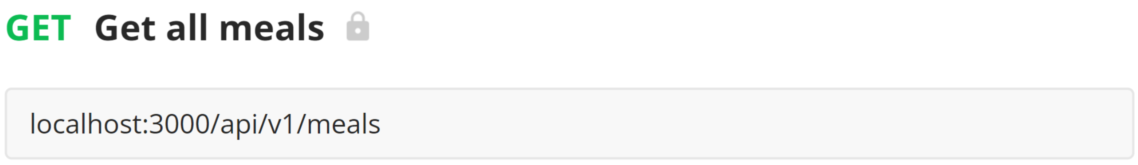
Additionally meals from a user can be retrieved using the GET **Get Meal with id from user** request (see figure 3.22).


GET Get Meal with id from user

localhost:3000/api/v1/meals/4

Figure 3.22: GET **Get Meal with id from user** request

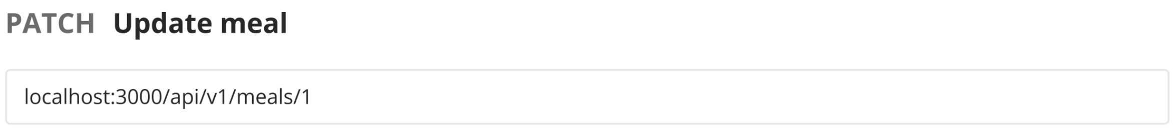
All meals can be retrieved using the GET **Get all meals** request (see figure 3.23).


GET Get all meals

localhost:3000/api/v1/meals

Figure 3.23: GET **Get all meals** request

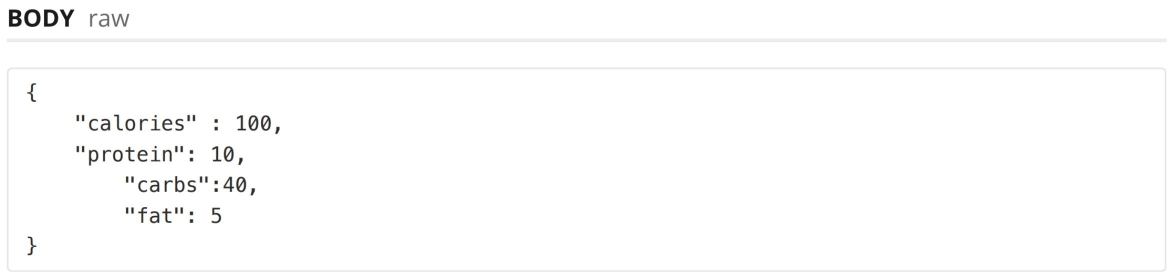
If the wrong attributes of a meal have been added, then a meal can be updated with the new values. Using the PATCH request a meal will be updated with new values. This way you do not have to delete a meal and create a new one.


PATCH Update meal

localhost:3000/api/v1/meals/1

Figure 3.24: PATCH **Update meal** request

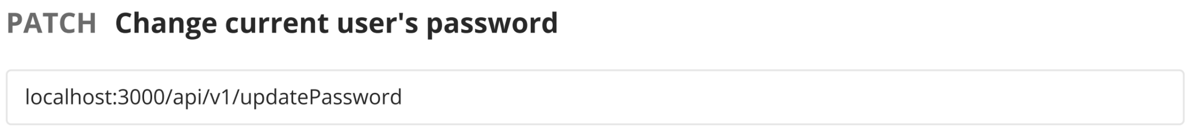
Figure 3.25 is an example what information is transferred with a PATCH request.


BODY raw

```
{
    "calories" : 100,
    "protein": 10,
        "carbs":40,
        "fat": 5
}
```

Figure 3.25: PATCH **Update meal** body request

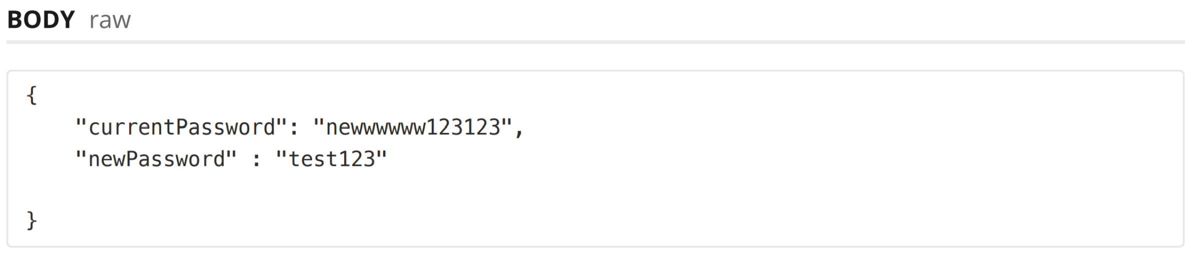
The user also has the ability of changing the first entered password, which is stored in the user attribute **password**. With the PATCH **Change current user's password** request, the password is updated.


PATCH Change current user's password

localhost:3000/api/v1/updatePassword

Figure 3.26: PATCH **Change current user's password** request

Figure 3.27 shows the information passed with the PATCH request. The current password and a new password have to be entered. If the current password is incorrect, then the user password is not changed.


BODY raw

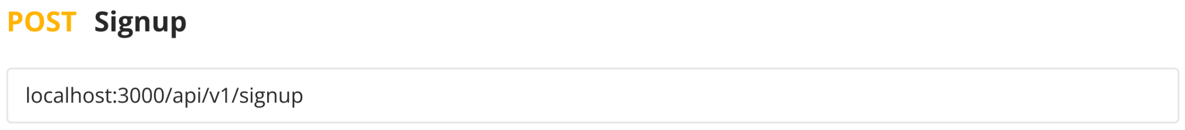
```
{
    "currentPassword": "newwwwww123123",
    "newPassword" : "test123"

}
```

Figure 3.27: PATCH **Change current user's password** body request

3.5.2 Public access APIs

To use the application, a user needs to create a profile. Registering a profile requires an email and a password. Using the POST **Signup** request, the user registers with his email and a password (see figure 3.28).

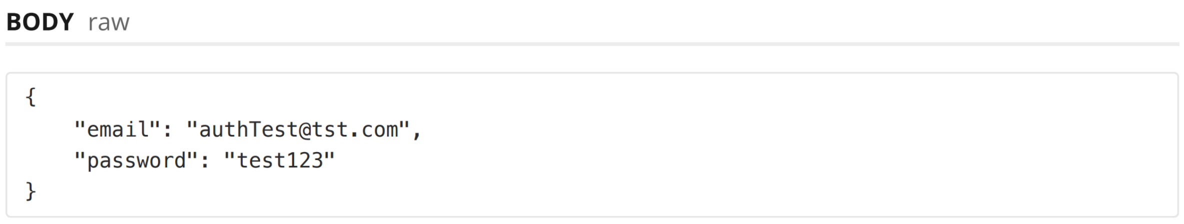


POST **Signup**

localhost:3000/api/v1/signup

Figure 3.28: POST **Signup** request

Figure 3.29 shows what data is transmitted with the POST request. In this example the attribute **email** is given the input **authTest@tst.com** with the attribute **password** is given the input **test123**.

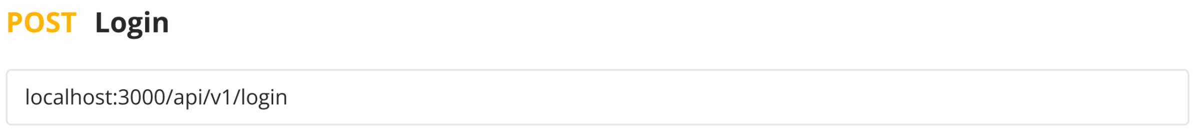


BODY raw

```
{
    "email": "authTest@tst.com",
    "password": "test123"
}
```

Figure 3.29: POST **Signup** body request

After successfully registering, a user can use the POST **Login** request to log into the previously created account.

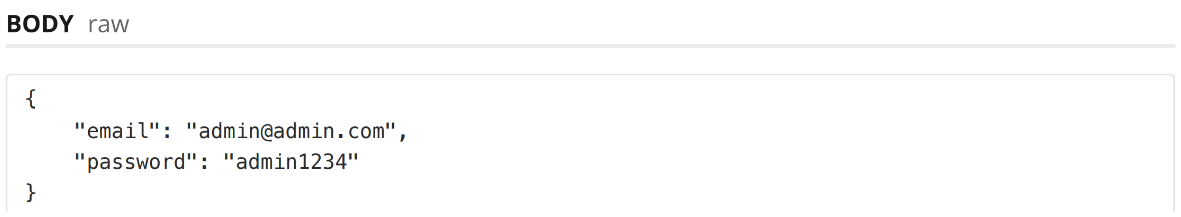


POST **Login**

localhost:3000/api/v1/login

Figure 3.30: POST **Login** request

The example in figure 3.31 is an admin logging into the application. The admin uses the attributes **email** and **password** to login. If the response status of the POST request is an error, then the wrong email or password has been entered.

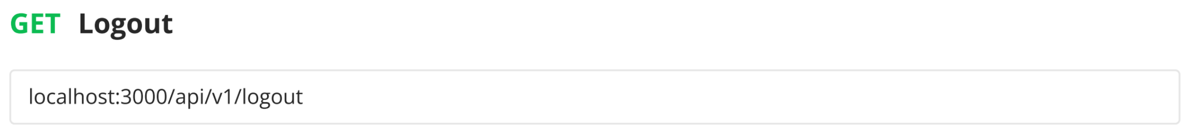


BODY raw

```
{
    "email": "admin@admin.com",
    "password": "admin1234"
}
```

Figure 3.31: POST **Login** body request

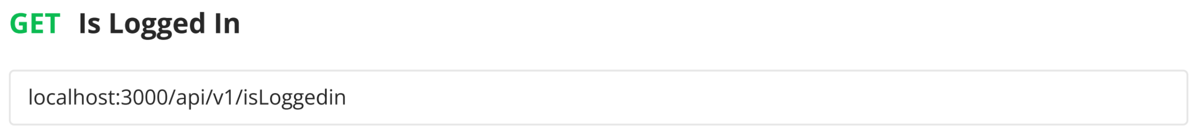
A user can log out using the GET **Logout** request.


GET Logout

localhost:3000/api/v1/logout

Figure 3.32: GET **Logout** request

The GET **Is Logged In** request checks whether the user is logged in.

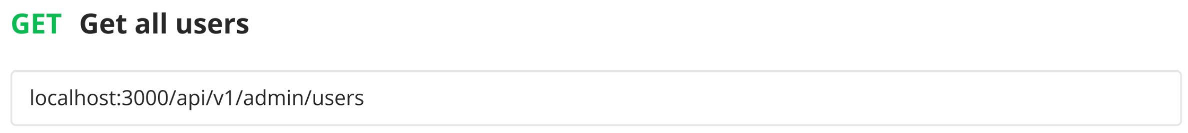

GET Is Logged In

localhost:3000/api/v1/isLoggedin

Figure 3.33: GET **Is Logged In** request

3.5.3 Admin only APIs

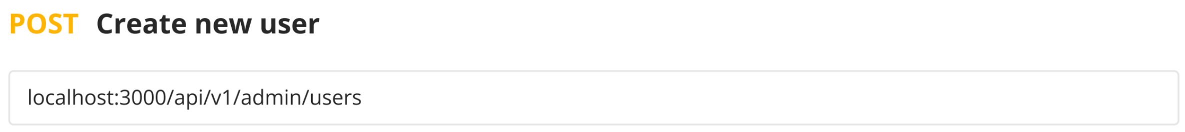
The purpose of the administrator REST API is to provide an interface for administrative tasks. Thus an admin needs to be able to execute more tasks then a normal user. For example, an admin can retrieve all users from the application (see figure 3.34).


GET Get all users

localhost:3000/api/v1/admin/users

Figure 3.34: GET **Get all users** request

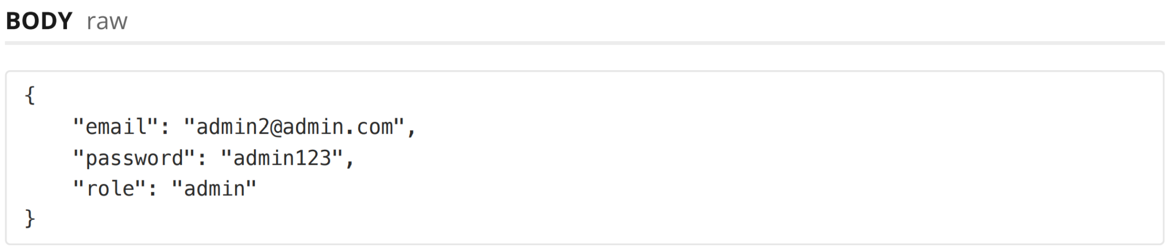
An admin also can create a new user using the POST **Create new user** request.


POST Create new user

localhost:3000/api/v1/admin/users

Figure 3.35: POST **Create new user** request

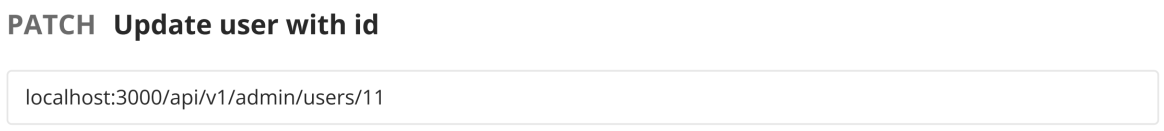
Figure 3.36 shows the information that the admin needs enter using the POST request. Furthermore the admin also can assign each newly created user a role.


BODY raw

```
{
    "email": "admin2@admin.com",
    "password": "admin123",
    "role": "admin"
}
```

Figure 3.36: POST **Create new user** body request

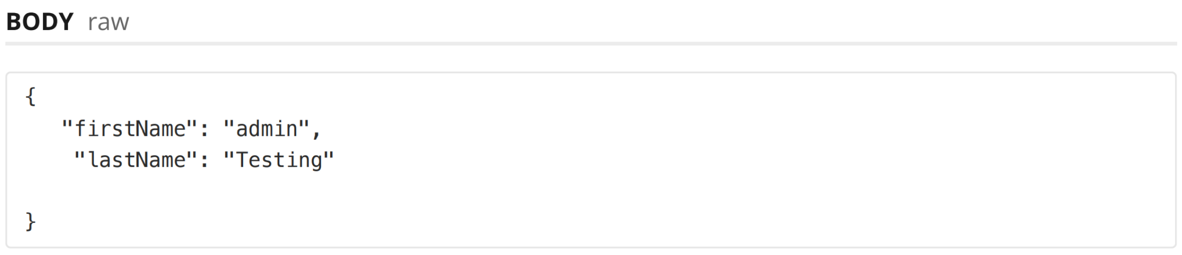
Using the ID assigned to a user, the admin can change information of a user.


PATCH Update user with id

localhost:3000/api/v1/admin/users/11

Figure 3.37: PATCH **Update user with id** request

The admin changes the first and last name of the user with the ID=11 in figure 3.38.

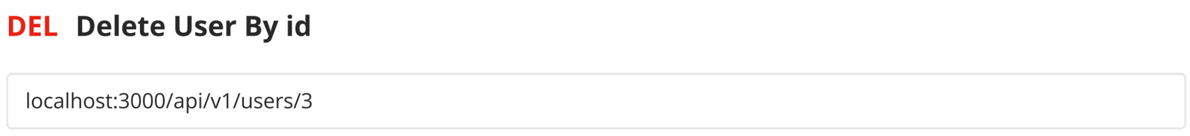

BODY raw

```
{
   "firstName": "admin",
    "lastName": "Testing"

}
```

Figure 3.38: PATCH **Update user with id** body request

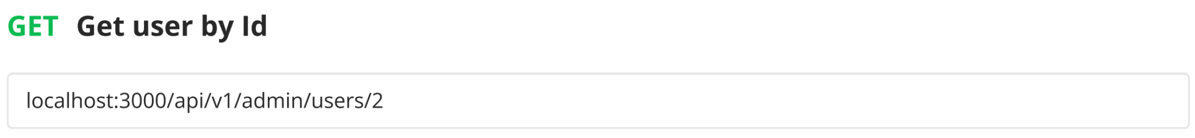
An admin also can delete a user using the ID. With the DEL **Delete User By id** request, the admin deletes a user.


DEL Delete User By id

localhost:3000/api/v1/users/3

Figure 3.39: DEL **Delete user by id** request

An admin can fetch a user by using the user ID and the GET **Get user by ID** request.

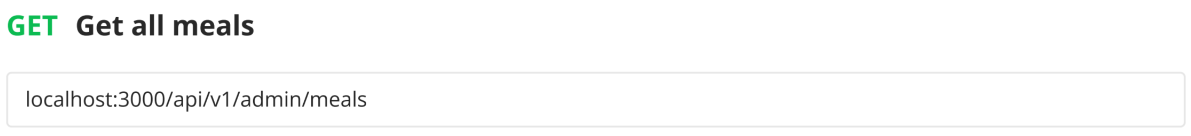


GET Get user by Id

localhost:3000/api/v1/admin/users/2

Figure 3.40: GET **Get user by id** request

An admin can access all meals of a user.

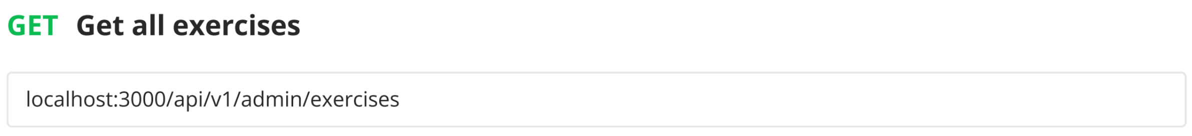


GET Get all meals

localhost:3000/api/v1/admin/meals

Figure 3.41: GET **Get all meals** request

An admin can access all exercises of a user.

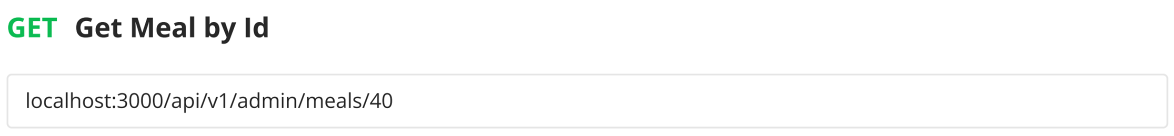


GET Get all exercises

localhost:3000/api/v1/admin/exercises

Figure 3.42: GET **Get all exercises** request

Certain meals with can also be fetched with the ID.

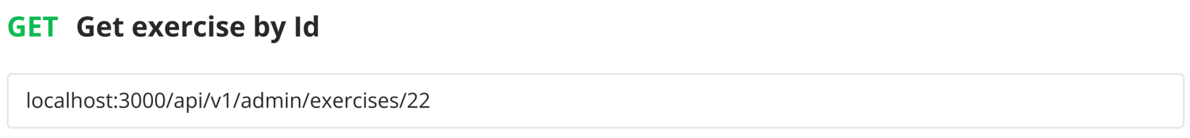


GET Get Meal by Id

localhost:3000/api/v1/admin/meals/40

Figure 3.43: GET **Get Meal by Id** request

In addition admins can also retrieve certain exercises using the GET **Get exercise by Id** request.

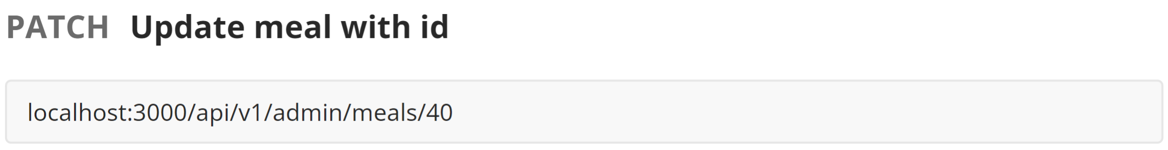


GET Get exercise by Id

localhost:3000/api/v1/admin/exercises/22

Figure 3.44: GET **Get exercise by Id** request

An admin also has the ability to change entered meals. This is done using the PATCH **Update meal with id** request.

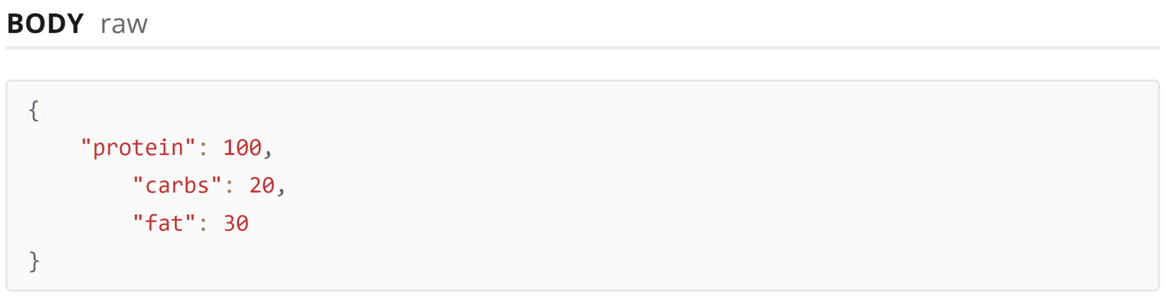


PATCH **Update meal with id**

localhost:3000/api/v1/admin/meals/40

Figure 3.45: PATCH **Update meal with Id** request

Figure 3.46 is an example of what a PATCH request could contain. The attributes **protein**, **carbs** and **fat** will have new values after the PATCH request.

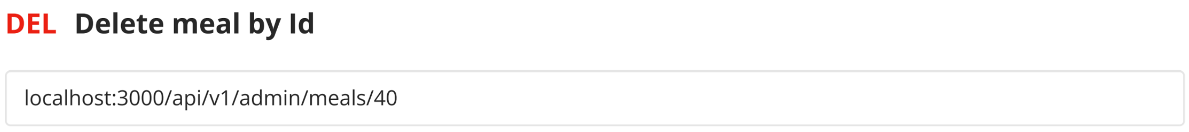


BODY raw

```
{
    "protein": 100,
        "carbs": 20,
        "fat": 30
}
```

Figure 3.46: PATCH **Update meal with Id** body request

An admin can delete a meal using the ID and the DEL **Delete meal by Id** request.

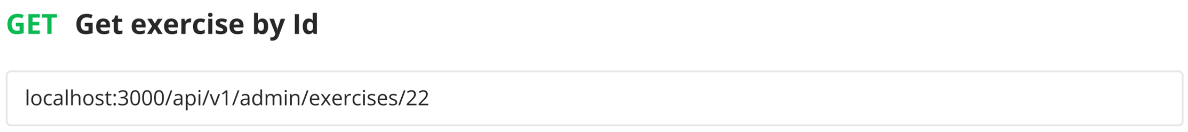


DEL **Delete meal by Id**

localhost:3000/api/v1/admin/meals/40

Figure 3.47: DEL **Delete meal by Id** request

Specific exercises can also be fetched using the ID and the GET **Get exercise by Id** request.

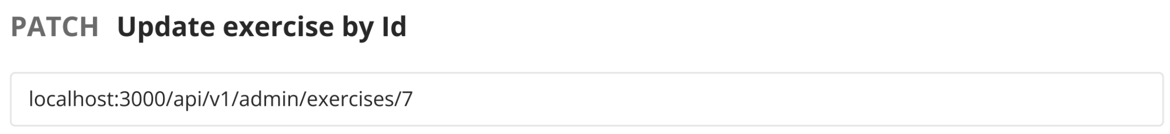


GET **Get exercise by Id**

localhost:3000/api/v1/admin/exercises/22

Figure 3.48: PATCH **Get exercise by Id** request

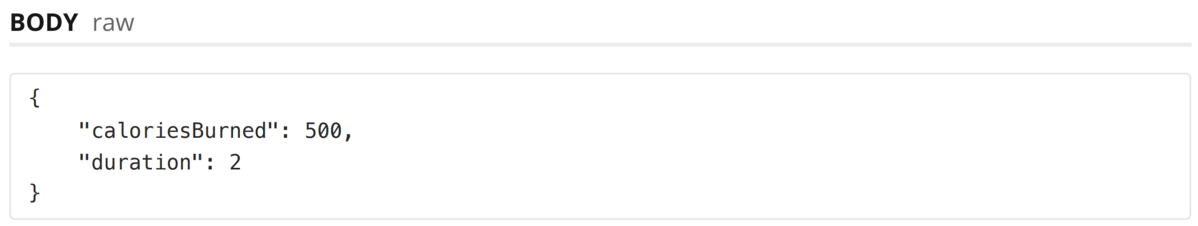
Exercises can also be updated using the id of an exercise and a PATCH **Update exercise by id** request.


PATCH Update exercise by Id

localhost:3000/api/v1/admin/exercises/7

Figure 3.49: PATCH **Update exercise by Id** request

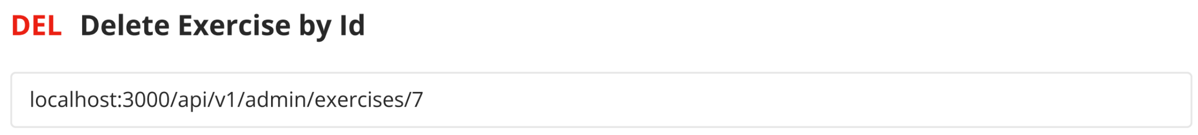
As shown in the example 3.50 the attributes **caloriesBurned** and **duration** will be changed with the PATCH request.


BODY raw

```
{
    "caloriesBurned": 500,
    "duration": 2
}
```

Figure 3.50: PATCH **Update exercise by Id** body request

Lastly an admin can delete an exercise using the ID and the DEL **Delete Exercise by Id** request.


DEL Delete Exercise by Id

localhost:3000/api/v1/admin/exercises/7

Figure 3.51: DEL **Delete Exercise by Id** request

4 Discussion of the Results

4.1 Install and run the project

Download and install node on your computer in case it is not installed https://nodejs.org/en/.

Go to project directory /caloriestracker using Terminal (Mac OS, Linux based Systems) or CMD (Windows). Go to /caloriestracker/server, run the command

```
$ npm install
```

to install all required packages and

```
$ npm start
```

to start application server

Go to /caloriestracker/client, run the command

```
$ npm install
```

to install all required packages and

```
$ npm start
```

to start application frontend

The application is now running at http://localhost.

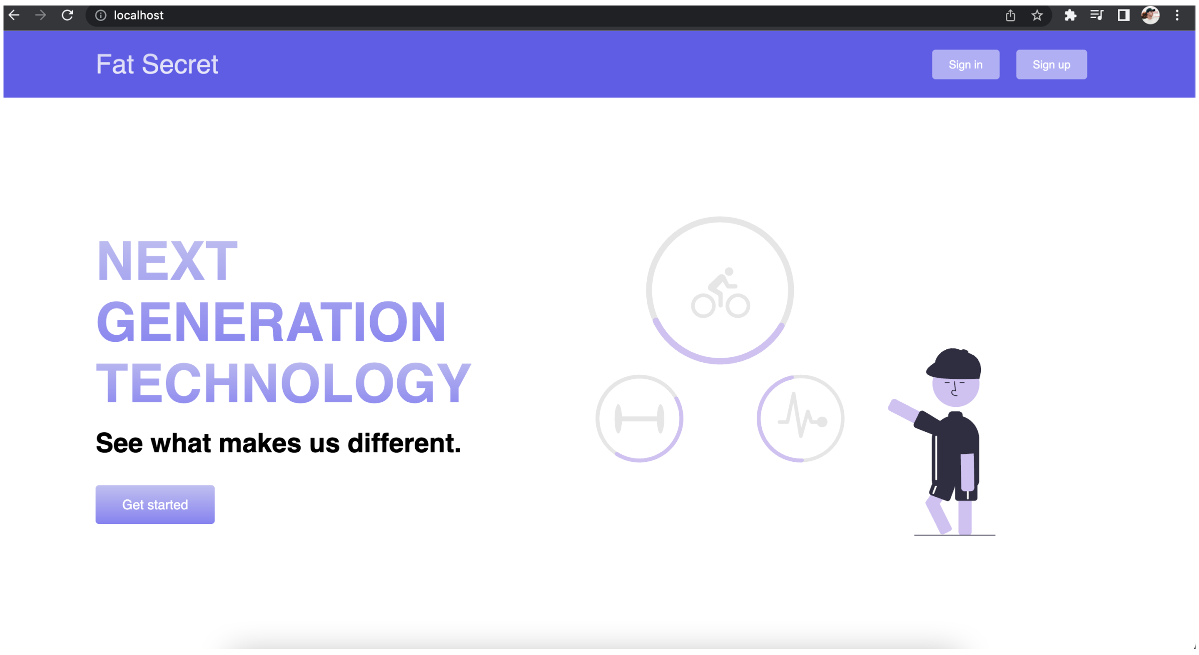


Figure 4.1: Application homepage

4.2 Conclusion

The planning, learning and subsequent execution of this programming project taught each participant how the inner workings of web development function and how to coordinate a multitude of tasks in an efficient way so that a successful collaboration can be guaranteed. By exchanging experiences about previously used online planning and writing tools the group was able to see which of these were sensible to use and which were better ignored. In addition the insight of learning new concepts and languages together within the group, instead of each member learning alone, was invaluable. By doing so questions that arose could be easily answered by another member, saving time and energy that could be used elsewhere. This effect was not limited to the theoretical part but was extended to the practical implementation of the newly learned due to every member having learned the same material. In return this resulted in a system that was not fully compartmentalized which was good for a project of this size. Since compartmentalization is almost always guaranteed within bigger projects, that take place in companies, getting a view of the whole program was of great learning value. The final product was the result of the cooperation between each member and their creative choices. Not only did the group learn how a database, a backend and a frontend communicate and work together but also how to use pleasant design choices to satisfy a product user.

All in all this programming exercise broadened the horizons of each team member by showing them where their weaknesses and gaps in their knowledge lie. This can only be achieved through practical experiences because one can only realize and analyse their own mistakes after the mistakes have happened. This important aspect is usually overlooked when you only learn how something works in theory. Overall this way a nice change of pace in terms of learning due to the previously mentioned reason and the independent form of working. Unlike other modules where you have someone looking over your shoulder and inspecting every step you take, here you have the freedom to explore and try new things. These new things were combined with existing knowledge of known concepts and software engineering to create the final product.

4.3 Future prospects

In regards to the future the program can be extended to hold functionalities that should be commonplace nowadays within the fitness-app industry such as interactive diagrams and exercise recommendations based on the users diet. An interactive statistical interface that is not limited to tracking calories per day but also for a longer duration could be used to show users their progress. By doing so users can be motivated to keep better track of their own way of life while simultaneously interacting more with the application. This statistical interface does not have to be limited to the calories category but can be extended to show the diet of the user. By showing the diets ratio of each macronutrient compound (carbohydrates, fats and proteins) it can convey whether the users diet is appropriate for their current lifestyle or not.
Another addition to the meals page would be a barcode scanner. This practise is already widespread especially within mobile applications. With this the user can easily scan a products barcode, categorise it and make it available for every other user. This means that as soon as a product (for example an apple) is categorized once, it does not need to be categorized again, since all the needed information is already stored on the database. In practise this means that all you have to do to add a meal to your table is to scan the product and insert the weight.

Bibliography

[1] *Amazon RDS for MySQL*. 2021. URL: https://aws.amazon.com/rds/mysql/?nc1=h_ls (visited on 07/01/2022).

[2] *How ChartJs Works*. URL: https://www.chartjs.org/ (visited on 07/13/2022).

[3] *How JWT works*. 2021. URL: https://blog.miniorange.com/what-is-jwt-json-web-token-how-does-jwt-authentication-work/ (visited on 07/01/2022).

[4] *How social workspaces help productivity and collaboration*. 2021. URL: https://penkethgroup.com/knowledge-centre/how-social-workspaces-help-productivity-and-collaboration/ (visited on 06/13/2022).

[5] Rajesh Kumar. *What is a bearer Token and how does it work?* 2021. URL: https://www.google.com/search?q=AUTHORIZATION+Bearer+Token+Token+%5C%3Ctoken%5C%3E&rlz=1C1CHBF_deDE977DE977&oq=AUTHORIZATION+Bearer+Token+Token+%5C%3Ctoken%5C%3E&aqs=chrome..69i57.382j0j7&sourceid=chrome&ie=UTF-8 (visited on 07/01/2022).

[6] Brenna Miles. *What is ClickUp*. 2021. URL: https://www.webopedia.com/definitions/clickup/ (visited on 07/09/2022).

[7] *MVC*. 2021. URL: https://blog.logrocket.com/building-structuring-node-js-mvc-application/ (visited on 07/01/2022).

[8] *Representational State Transfer*. 2022. URL: https://de.wikipedia.org/wiki/Representational_State_Transfer (visited on 07/11/2022).

[9] *Response in Postman*. 2021. URL: https://www.javatpoint.com/response-in-postman#:~:text=Once%20you%20send%20the%20request,which%20is%20called%20the%20response. (visited on 07/01/2022).

[10] Gustavo Romero. *What is Postman API Test*. 2021. URL: https://www.encora.com/insights/what-is-postman-api-test (visited on 06/11/2022).

[11] Adrian Twarog. *What is Figma? A Design Crash Course [2021 Tutorial]*. 2021. URL: https://www.freecodecamp.org/news/figma-crash-course/ (visited on 07/11/2022).

[12] *What is Express*. 2021. URL: http://expressjs.com/ (visited on 07/01/2022).

[13] *What is Node JS*. 2021. URL: https://www.geeksforgeeks.org/explain-the-working-of-node-js/#:~:text=It%20is%20a%20used%20as,fast%2C%20and%20data%2Dintensive. (visited on 07/01/2022).