Calculatoare și Tehnologia Informației - Engleză
Academic year 2022 - 2023

CALORY – FOOD TRACKER MOBILE APPLICATION

Candidate: Bogdan, Tatu

Scientific coordinator: Conf.dr.ing. Dan Pescaru

Session: September 2023

Table of Contents

1. **INTRODUCTION** 6
2. **SPECIFICATIONS, REQUIREMENTS & DESIGN** 7
2.1. SIMILAR APPLICATIONS 7
2.1.1. MyFitnessPal 7
2.2. PROJECT REQUIREMENTS 7
2.3. PROJECT SPECIFICATIONS 8
2.4. BLOCK DIAGRAM 9
2.5. CLASS DIAGRAMS 10
2.5.1. Controllers 10
2.5.2. Repositories 12
3. **TECHNICAL BACKGROUND** 13
3.1. PROGRAMMING LANGUAGES 13
3.1.1. TypeScript (JavaScript) 13
3.1.2. Rust 13
3.2. FRONTEND TECHNOLOGIES 14
3.2.1. React Native + Expo 14
3.2.2. Nativewind (Tailwind CSS) 14
3.2.3. Axios 15
3.2.4. React Query 15
3.2.5. React Native Reanimated 15
3.2.6. Expo Libraries 16
3.3. BACKEND TECHNNOLOGIES 16
3.3.1. SQLx and diesel_cli 16

3.3.2. Tide & Serde 17
3.3.3. Surf 17
4. **APPLICATION UTILIZATION** 18
5. **IMPLEMENTATION** 26
5.1. BACKEND 26
5.1.1. Database Diagram 26
5.1.2. Migrations 26
5.1.3. Models 28
5.1.4. Repositories 32
5.1.5. Controllers 38
5.1.6. Data Transfer Objects 43
5.1.7. Main 44
5.2. FRONTEND 46
5.2.1. App.tsx 46
5.2.2. Layouts 48
5.2.3. Services 52
5.2.4. Components 54
5.2.5. Types 56
5.2.6. Utils 56
5.2.7. Assets 57
6. **CONCLUSIONS** 58
6.1. FINAL WORDS 58
6.2. FUTURE WORK 58
7. **REFERENCES** 59

ABSTRACT

Teza intră în detaliu despre procesul de dezvoltare și implementare a aplicației mobilă full-stack Calory. Aplicația se ocupă de o mulțime de sarcini diferite legate de sănătate, cum ar fi urmărirea alimentelor (calorii, macronutrienți, micronutrienți), adăugând alimente verificate din Baza de date USDA FoodCentral sau alimente personalizate, făcute de utilizator, urmărirea apei, statistici și diagrame de progres.

Motivația muncii este de a ajuta oamenii să ia deciziile corecte atunci când vine vorba de hrană și nutriție și de a înțelege mai bine profilurile reale de macro și micronutrienți ale alimentelor consumate.

În timpul creării aplicației au fost folosite diverse instrumente, cum ar fi GitHub, pentru gestionarea controlului versiunilor.

În ceea ce privește implementarea aplicației, backend-ul a fost realizat folosind limbajul de programare Rust, folosind o abordare bazată de date în primul rând la crearea tabelelor bazei de date.

RDBMS utilizat la crearea aplicației PostgreSQL 15.

React Native a fost folosit la realizarea front-end-ului aplicației mobile, construit pe deasupra cadrului Expo și scris în TypeScript.

Comunicarea dintre client și server s-a realizat printr-un API REST, iar interfața și backend-ul comunică direct cu API-ul USDA FoodCentral Database.

ABSTRACT

The thesis goes into detail about the development and implementation process of Calory, the full-stack mobile application. The application handles a lot of different health related tasks, such as food tracking (calories, macronutrients, micronutrients), by adding verified food from the USDA FoodCentral Database or custom, user-made food, water tracking, statistics, and progress charts.

The motivation of the work is to help people make the right decisions when it comes to food and nutrition, and to get a better grasp on the actual macro and micronutrien profiles of the foods consumed.

Various tools were employed during the creation of the application, such as GitHub, for managing version control.

Regarding the implementation of the application, the backend was made using the Rust programming language, using a database-first approach when creating the database tables.

The RDBMS used in the creation of the application PostgreSQL 15.

React Native was used in the making of the frontend of the mobile application, built on top of the Expo framework, and written in TypeScript.

The communication between the client and the server was done through a REST API, and the frontend and backend communicate directly with the USDA FoodCentral Database API.

1. INTRODUCTION

Calory is a mobile application that assists users in managing their eating patterns and promoting a healthy lifestyle. It provides a comprehensive toolkit for users to trace their food consumption, monitor their water intake, engage in physical activity by counting their steps, and access detailed statistical insights for a specific day or time period. This multifunctional software is a convenient and effective companion for those who wish to maintain a balanced diet and maximize their overall health.

Through Calory's user-friendly interface, users can easily add food items to their digital diary, ensuring that their nutritional consumption is accurately recorded. Barcode recognition is incorporated into the app, allowing users to rapidly retrieve nutritional information for packaged foods. By utilizing this function, users can easily make informed dietary decisions and monitor their progress toward attaining personalized health objectives.

In addition to food monitoring, Calory enables users to construct customized meals, which facilitates efficient meal planning and promotes a nutritionally balanced lifestyle. Individual meal components can be assembled by the user, allowing for an exhaustive overview of their dietary composition and a more accurate assessment of nutritional intake.

Calory includes a dedicated water intake tracker due to the significance of hydration in maintaining optimal health. Users can easily track and register their daily water intake, allowing them to remain adequately hydrated throughout the day. This function serves as a helpful reminder and encourages users to prioritize proper hydration, which is essential to their overall health.

In order to encourage an active lifestyle, Calory includes a step counter that enables users to monitor their physical activity levels and establish personal objectives. By recording the number of steps taken, users can evaluate their daily activity and endeavor to increase their overall movement, thereby improving their fitness and health.

The statistical analysis function of Calory provides users with extensive insight into their dietary habits, physical activity levels, and overall progress. Users can access exhaustive statistics for a specific day or select a desired time period, allowing them to assess their behaviors, identify trends, and make informed adjustments to effectively achieve their health goals.

Calory is a comprehensive mobile application that caters to the requirements of individuals attempting to maintain a healthy lifestyle due to its vast array of functionalities. Calory enables users to make informed dietary decisions, monitor their progress, and embark on a path to enhanced health and well-being by combining user-friendly features and comprehensive data analysis.

2. SPECIFICATIONS, REQUIREMENTS & DESIGN

2.1. SIMILAR APPLICATIONS

2.1.1. MyFitnessPal

MyFitnessPal, developed by Under Armour, was one of the original calorie tracker apps on the market, and is still one of the most used applications for this today.

It had a lot of features when it came out and has garnered even more to this day. It has features related to health in general, like an intermittent fasting tracker, workout routines, a health and nutrition newsfeed. It has a gigantic database of foods and recipes to choose from that is easy to search, either by text, or by scanning barcodes or the food itself, using AI to detect the meal and what it’s made of.

MyFitnessPal also has functionality for adding workouts and exercises and can be paired to other apps like Apple’s Health App, Fitbit, Strava.

What it does lack is a polished design on older screens that have yet to change, the application being actually unintuitive to use in some places by cluttering the screen with as much information as possible. In addition to this a lot of useful but simple features are blocked behind a paywall, like custom macronutrient tracking.

2.2. PROJECT REQUIREMENTS

A user will be able to:

- create a profile with data about their body.
- use the application on multiple mobile platforms, such as iOS and android.
- get a meal plan with a given calorie and macronutrient goal depending on their personal information and weight goal.
- adjust their calorie and macronutrient goal manually.
- search a curated list of foods with nutritional information to add to their meals.
- have a good user experience, with a user-friendly interface that is easy to navigate.
- have a responsive and smooth UI, with screen transitions and animations.
- search for food by their name or brand.
- see their recently tracked.
- favourite foods for easier tracking.
- scan foods by their bar code to bring up the nutritional information.
- log their daily water intake.

- see a daily summary with the number of eaten calories and macronutrients, in comparison to their goal and a micronutrient breakdown.
- see their calorie, macronutrient, water, weight progress for a given period of time.
- see their profile with their user data and personal details.
- adjust their water habits.
- change their dietary needs and preferences.

2.3. PROJECT SPECIFICATIONS

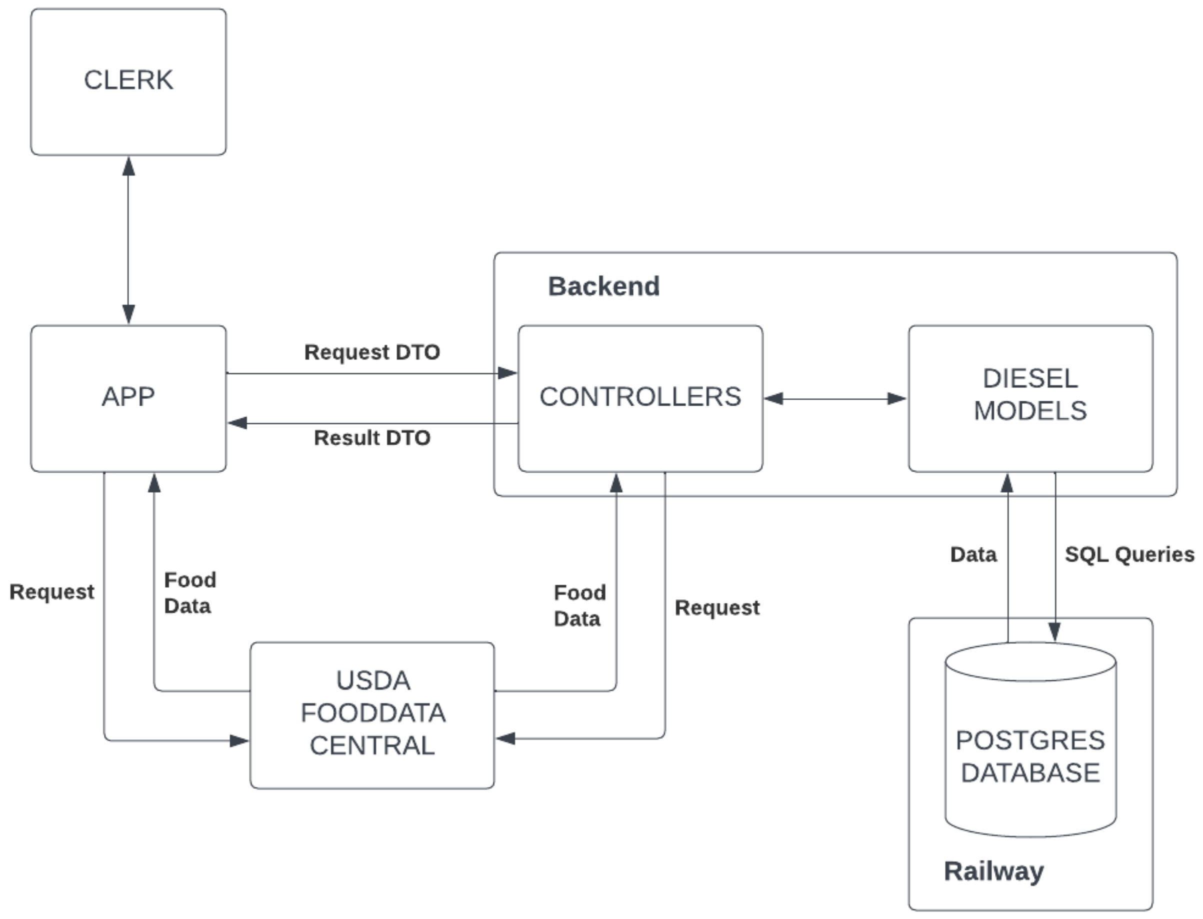
The frontend of the mobile application will be built using the Expo framework on top of React Native, with TypeScript, for multiple platform support, but prioritising iOS.

The backend will be written in Rust, utilizing Tide, together with Serde handling the serialization and deserialization of data, to build a fast, and reliable REST API to handle requests from the client. It will also use a database-first approach with SQLx as the database driver to communicate and query the database in a type-safe manner.

User authentication and management is done using Clerk, which also handles OAuth and user persistence using JSON Web Tokens.

The database will be a relational database, using the PostgreSQL relational database management system. It will be hosted on and managed by Railway, for convenience during the development process.

2.4. BLOCK DIAGRAM



The system is comprised of multiple components: the application, the Clerk platform, the USDA FoodData Central API, the backend, that has the controllers, the DTOs and Diesel Models, and the Postgres database managed by Railway.

App (Frontend) – The frontend is the interface of the user that communicates with clerk to get the user data, with the USDA API directly to get data on a specific food or search for food. It also communicates with the backend through a variety of routes to request data to be displayed to the user.

Backend – The backend is the component responsible for processing requests sent from the frontend, performing calculations and business logic, handling data retrieval to generate responses through DTOs (Data Transfer Object) to be sent back to the client-side of the application. It is made up of:

DTOs – Data transfer objects that serve as containers for data that allow for information to be transported between different layers of the system. They can either be sent as a request from the client-side to the server-side and deserialized by the backend, or vice-versa, to be serialized into JSON as a response back to the client.

Controllers – The controllers are a group of functions that receive and handle the HTTP requests dispatched to them by Tide, that ties each controller to a given route. They handle GET, POST, PUT and DELETE requests, perform business logic, and send back appropriate responses to the client-side. Some controllers also communicate with the USDA API to handle more complex requests containing the food data that should be processed in the backend.

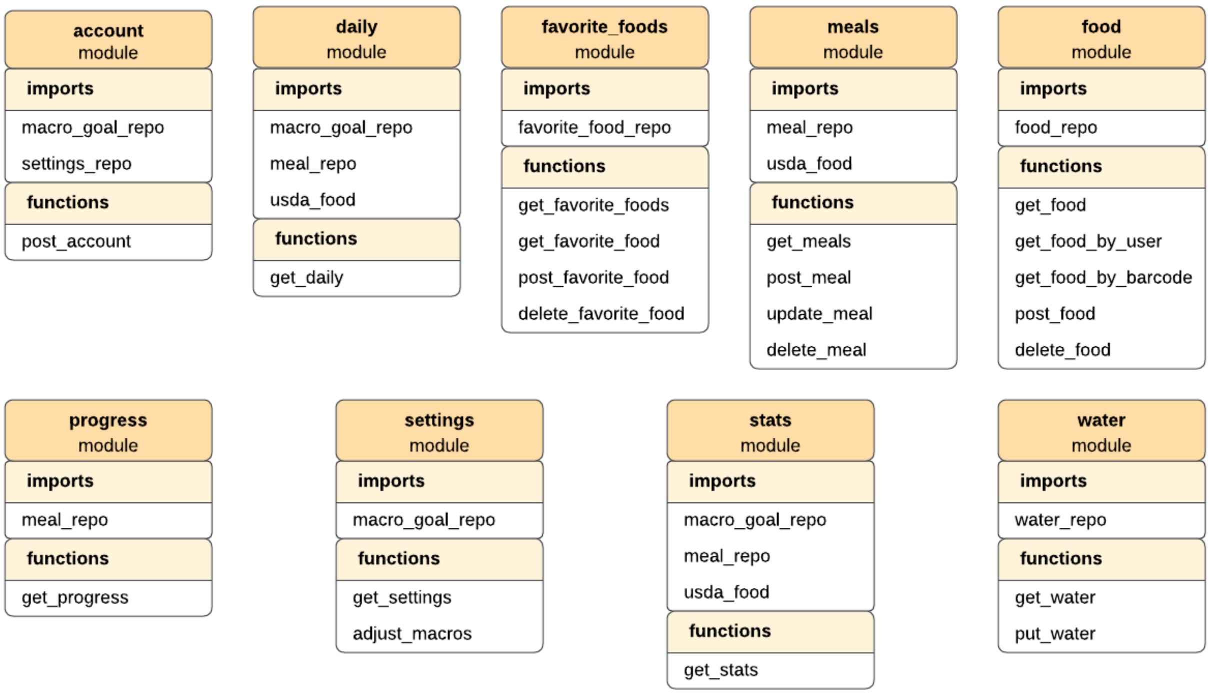
Diesel Models – Diesel is an ORM that abstracts the database system and allows for querying and data manipulation of the database in an object-oriented paradigm. The code is translated by Diesel into the appropriate SQL query to perform CRUD operations over the database.

Postgres Database – The database hosted on Railway stores and manages part of the data used in the application.

USDA FoodData Central API – An external API provided by the United States Department of Agriculture that contains relevant nutritional data for a whole array of foods. Either handles requests directly from the client-side for simple tasks, such as displaying the raw data, or from the server-side for more processing-intensive ones, that need to alter the data before sending it to the client.

2.5. CLASS DIAGRAMS

2.5.1. CONTROLLERS



Rust, being a multi-paradigm language, doesn’t force the developer into grouping controller functionality into a class, so, naturally, each function is bundled up into a module, using repositories and/or the ‘usda_food’ service as imports.

They handle the HTTP requests that are routed by Tide to them and return the appropriate response.

The controllers are:

account – handles account creation actions, such as creating the settings of a user and calculating their macronutrient goals.

daily – used to congregate the information of a user’s daily meals and goal.

favorite_foods – handles operations related to users’ favourite foods.

meals – handles the management of users’ meals, creating, reading, updating, and deleting.

food – handles the retrieval of the food by different criteria and the modification of food.

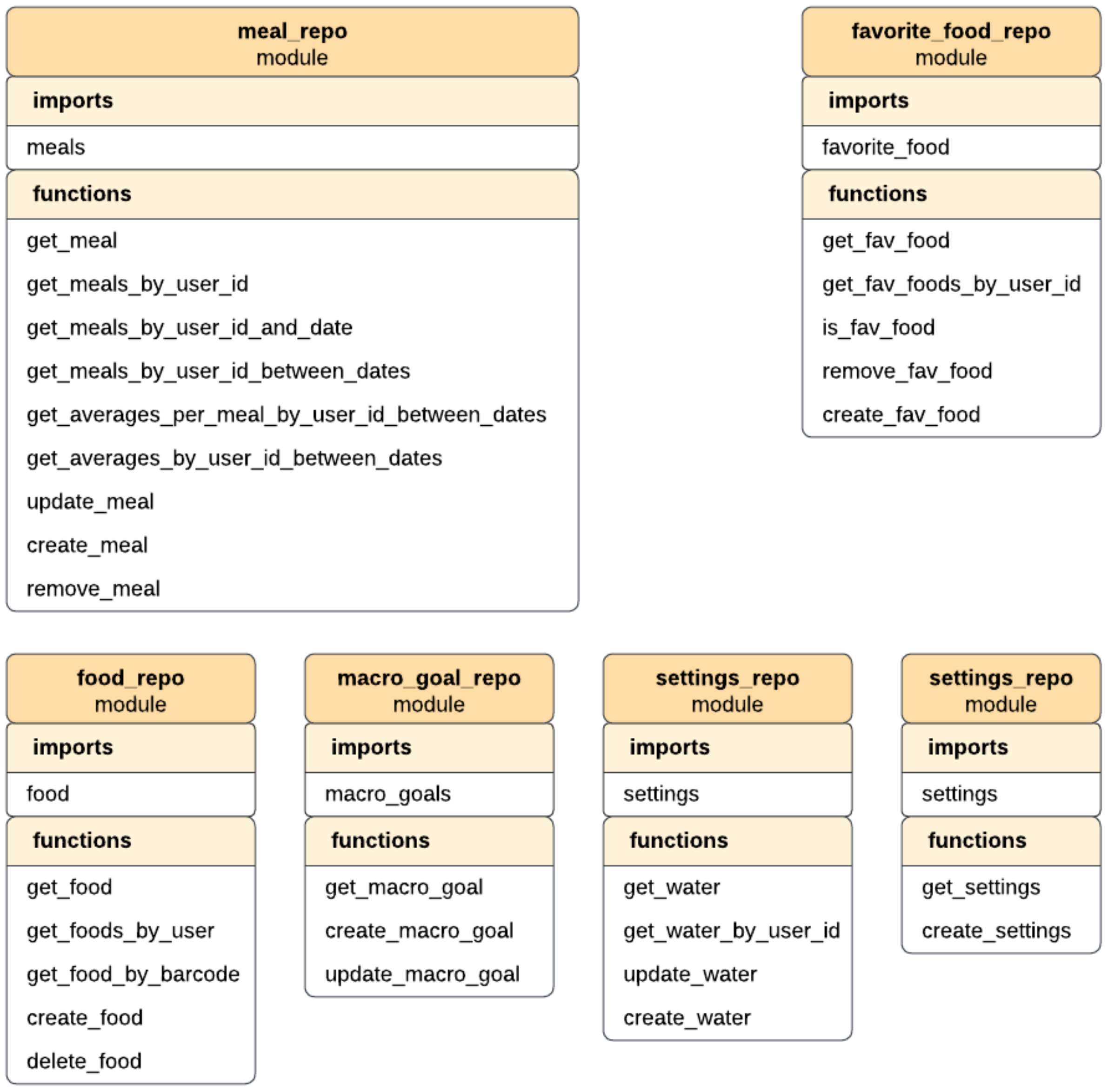
progress – handles the calculation and statistical operations necessary to provide information on a user’s progress over time.

settings – represents the operations tied to the reding and updating of a user’s settings.

stats – gives insight on a user’s daily statistics.

water – manages the user’s daily water intake.

2.5.2. REPOSITORIES



In the backend, each repository is grouped intro their own module. They connect the controllers to the Diesel models and the database, providing them data access and storage by creating SQL queries using Diesel's ORM.

3. TECHNICAL BACKGROUND

3.1. PROGRAMMING LANGUAGES

3.1.1. TypeScript (JavaScript)

TypeScript is a high-level programming language and developer tool what adds static typing with optional type annotations to JavaScript, allowing developers to define and enforce types for variables, function parameters, return values and more. JavaScript, TypeScript's underlying programming language, offers a variety of programming paradigms, including but not limited to imperative programming, object-oriented programming, supporting principles such as encapsulation, inheritance, and polymorphism, functional programming, event-driven programming, with its integrations with the Document Object Model (DOM), having the ability to respond to user interactions, and asynchronous programming, through mechanisms such as callbacks, Promises and the async/await syntax.

Due in part to the language's ease of use, but also JavaScript's dominance on the web, since it was originally designed to be one of the core technologies powering web browsers, TypeScript has amassed a vast ecosystem of third-party packages for UI and UX design and development, such as frameworks (e.g., Angular, Svelte, Astro) and libraries (e.g., React, SolidJS). It also offers a multitude of utility libraries for tasks such as animations, handling HTTP requests, graph and chart visualizations, machine learning, and some that provide utility functions for working with arrays, objects, functions, and other data types in a functional programming style.

3.1.2. Rust

Rust is a modern, system-level programming language that prioritizes memory safety, through its use of the borrow checker, and performance, claiming to be a blazingly fast programming language. It was created to address the difficulties of writing dependable and efficient software, particularly in systems programming where low-level control and high performance are essential.

The language utilises the borrow checker, a novel approach to solving memory related problems, a static code analysis tool that enforces rules at compile time, checks for the passing of ownership of variables to functions, ensuring that no race conditions occur on references to values, and tracking the lifetime of references. Unlike other low-level programming languages such as C or C++, that utilise manual memory allocation and freeing, where developers fully manage memory access, which can be very difficult, and is the source of many runtime errors.

Even though Rust is not as popular as Python or JavaScript, it has steadily grown in popularity over the years, making it a great option for building applications, with a moderate number of third-party packages and sufficient documentation to begin programming in it. Rust offers a multitude of libraries to aid in the design of the backend for creating APIs (e.g., Rocket, Actix Web, Axum, Tide), ORMs and database drivers (e.g., Diesel, SeaORM, sqlx), HTTP clients (e.g., reqwest, hyper, surf), error handling (e.g., anyhow, thiserror) and many more use cases.

3.2. FRONTEND TECHNOLOGIES

3.2.1. REACT NATIVE + EXPO

The frontend of the application was built on top of React Native in conjunction with Expo. React Native is a JavaScript-based framework for building native mobile applications. It facilitates the development of applications that are compatible with both iOS and Android. Following a component-based architecture. React Native enables the creation of reusable JavaScript user interface (UI) components written in JSX (JavaScript XML) by integrating markup and logic in the same file and separating the UI into independent units.

React Native also enables the development of mobile applications with performance and user experience comparable to native apps by providing access to native APIs.

Adopting React Native for my thesis application allows me to leverage my existing knowledge of React, saving me time and effort in comparison to developing distinct apps for each platform.

I chose to integrate Expo into my development process in addition to React Native. Expo is a comprehensive suite of tools and services that facilitates the development of React Native mobile applications. Expo provides a vast library of pre-built UI components, facilitating the construction of an app's user interface, but also provides access to a variety of device APIs, including camera, haptics, status bars, and push notifications, allowing for the seamless integration of device-specific features.

Expo's Over-the-Air (OTA) updates make it simple to distribute app updates without requiring users to obtain a new version from the app store. Expo simplifies the development and deployment process by offering tools to generate standalone binary files for iOS and Android platforms and enabling the sharing of apps via QR codes.

3.2.2. NATIVEWIND (TAILWIND CSS)

Tailwind is a utility-first CSS framework. In contrast to other CSS frameworks like Bootstrap or Materialize CSS it doesn’t come with predefined components. Instead, Tailwind CSS operates on a lower level and provides a set of CSS helper classes. Tailwind CSS is not opinionated and lets developers and designers create their own unique design.

NativeWind uses Tailwind CSS as scripting language to create a universal styling system and. It processes your styles during your application's build and uses a minimal runtime to selectively apply responsive styles.

NativeWind allows for a high level of customization, enabling the fine-tuning and extension of utility classes to conform to particular design criteria. Customization options include color schemes, spacing configurations, typographical elements, and other visual aspects.

I chose Nativewind in order to have granular control over the design process of the application, to obtain a consistent, yet distinct design compared to component libraries, without having to manually type out the complete React StyleSheet styles.

3.2.3. Axios

Axios is a widely used Promise-based HTTP client library that provides a high level of abstraction, making it simple to send HTTP requests to REST endpoints on the backend.

Axios abstracts away the complexities of XMLHttpRequests, CORS (Cross-Origin Resource Sharing) restrictions, and the serialization and deserialization of JavaScript objects to JSON.

Axios is used to handle requests and responses to the backend and to request data from the FoodData Central API in this project. It is utilized in conjunction with React Query to simplify data retrieval, state management, and error handling.

3.2.4. React Query

React Query by TanStack is a data-fetching utility designed to manage the state of server-requested data. It offers a plethora of additional tools and options, including data caching, infinite and paginated queries, status indicators of active queries, placeholder query data, error handling, and complex typing of states and functionality.

React Query's fetching mechanisms are agnostically built on Promises and can be used with Axios and any other asynchronous data fetching client.

It was used throughout the application to facilitate the process of working with remote data from the backend by modifying the UI based on the request state, caching queries and invalidating caches on data mutation, and handling data prefetching in anticipation of the user's use.

3.2.5. React Native Reanimated

Reanimated is a concise React Native animation library used to create performant and easy to write animation code that runs on the iOS or Android’s native animation engine.

It handles infinite and event-driven animations written in a declarative manner with utility functions such as withTiming, withSequence, withDecay. The API also contains implementations for number and colour interpolation, as well as easing functions to build complex animations.

3.2.6. EXPO LIBRARIES

The Expo SDK provides access to device and system functionality such as contacts, camera, phone sensors, haptics, and so on, in the form of packages.

The application uses many of the libraries provided by expo, but I will only mention a few of the more important ones.

Expo-haptics was utilized to provide haptic feedback to users in order to improve their experience. It provides tactile feedback of varying strengths and patterns to inform the user of the occurrence of an event or to indicate the success or failure of an action.

Expo-router, which was developed on top of react navigation, provides a file-based routing solution that supports both static and dynamic routes with optional query parameters. It includes components used for routing, linking, unmatched routes, splash displays, and Head components to provide meta tags for the application. The inclusion of layouts that provide a common interface and transitions for components (e.g., navigation layout) is also noteworthy.

Lottie-react-native is another library listed in the expo sdk which was used for smaller animations to give personality to the application.

Expo-sensors provides access to Android's and iOS' system sensors to get the user's step count, and also allows the application to subscribe to pedometer updates. The step count is used to calculate the calories burned by the user in the context of the application.

These are only a few of the dozens of packages that Expo has to offer and that were used in the making of the applicaiton.

3.3. BACKEND TECHNNOLOGIES

3.3.1. SQLX AND DIESEL_CLI

The most prevalent database driver crate for Rust is SQLx. It provides static checking of queries, eliminating the possibility of improper database interactions at compilation time, without a DSL (domain-specific language) or API for building queries.

SQLx handles the connection to a database or a database pool, with various options for configuring the pool, such as specifying a maximum and/or minimum number of concurrent connections to be maintained by it, setting a maximum lifetime for all connections before closing or an idle timeout.

SQLx is not an ORM (object-relational mapping), meaning the developer has to write raw SQL queries. It mitigates the drawbacks by providing a compile-time query checker, by connecting to the development database at compile time to have the database itself verify the SQL queries and not allowing for the program to be built until the queries correct.

It also comes with a command line interface tool that handles database migrations, but I stuck with ‘diesel_cli’, from the previous version of my backend, as my migration tool, made by the creators of Rust’s Diesel ORM crate, to handle modifications made to the databases’ schemas.

SQLx is a great choice for a database driver, since Rust’s ORMs only provide ease when writing simple queries, even after the hurdle of learning their own API. For more complex queries, ORMs don’t have the necessary functionalities in order to implement them, having to resort back to raw SQL, in turn losing the immense benefits of type safety that come with using ORMs.

3.3.2. TIDE & SERDE

Tide is a web framework for Rust that provides routing, pre-processing of requests and post-processing of responses, through serialization and deserialization of data by utilising the Serde crate.

Its primary responsibility is to listen to incoming web requests to the designated domain, dispatch the requests to the application code, and return a response to the client by following a lifecycle of: routing, to determine which request handler to invoke by matching route attributes defined in the application, validation of the request against types present in the matched route, otherwise forwarding the request to the next matching route, until it either finds one or invokes the error handler, processing done by the application code, and returning a HTTP response to the client.

Tide also provides interfaces for writing custom middleware, which have been used in the application for tasks such as logging incoming requests and outgoing response results, user validation before passing on the requests to controllers and error handling.

Shared application scoped state is also a useful feature that Tide provides, primarily for sharing the database pool between routes, not having to start up a different connection with every request.

3.3.3. SURF

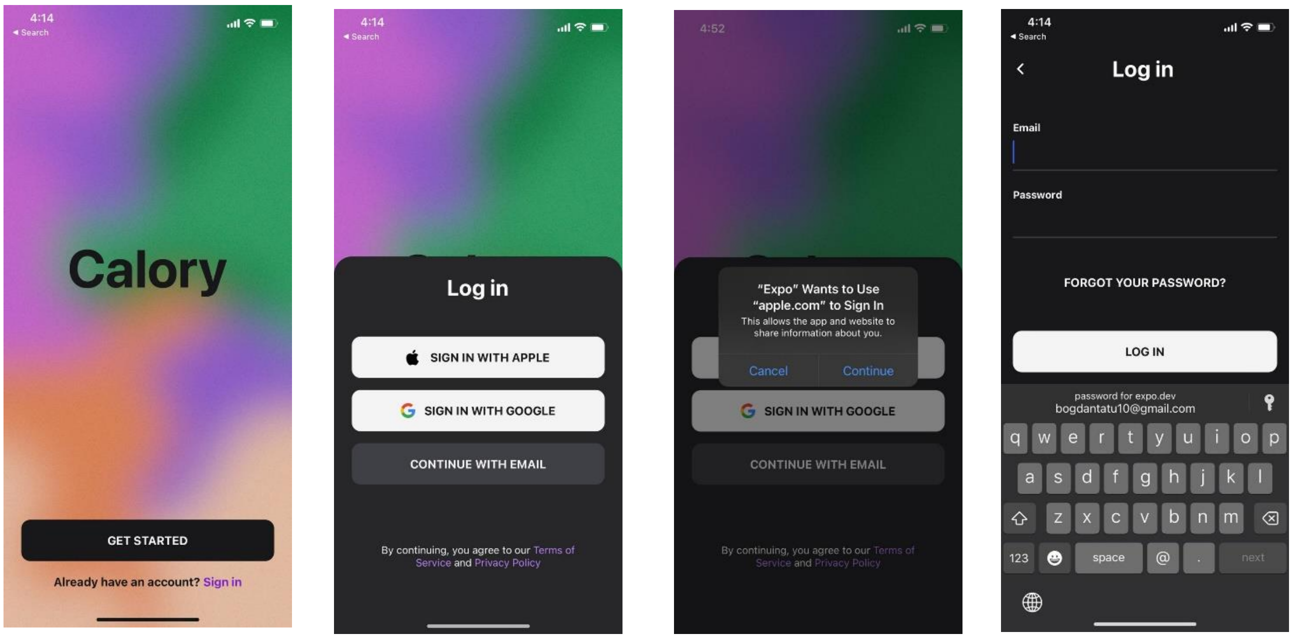
The Surf crate provides a higher-level HTTP client for handling requests.

It handles serialization and deserialization automatically using the Serde crate and it allows for the creation of custom middleware, with an inbuilt logger that prints out information about the requests made and their results.

It is the equivalent of the frontend package, axios, and is only used to make request to the FoodData Central API.

4. APPLICATION UTILIZATION

Upon launching the application, the user is presented with a title screen from which he can either go through the account creation process or log in.



By continuing, you agree to our Terms of Service and Privacy Policy

This allows the app and website to share information about you.

Figure 4-5.2-1. Title / Log In Modal / Log In Oauth / Log In screens

When pressing the sign in button, a modal pops up with different modes of signing in, through Apple, Google or with email address. When the user presses either the sign in with apple/google buttons, an alert appears on the screen telling the user that the application wants to use apple.com / google.com to sign in. Pressing continue prompts the iOS device to open face scan to sign in, and the user is redirected to the home page (diary).

Choosing the email sign in option brings the user to a different screen where he has to enter his/her credentials to log in.

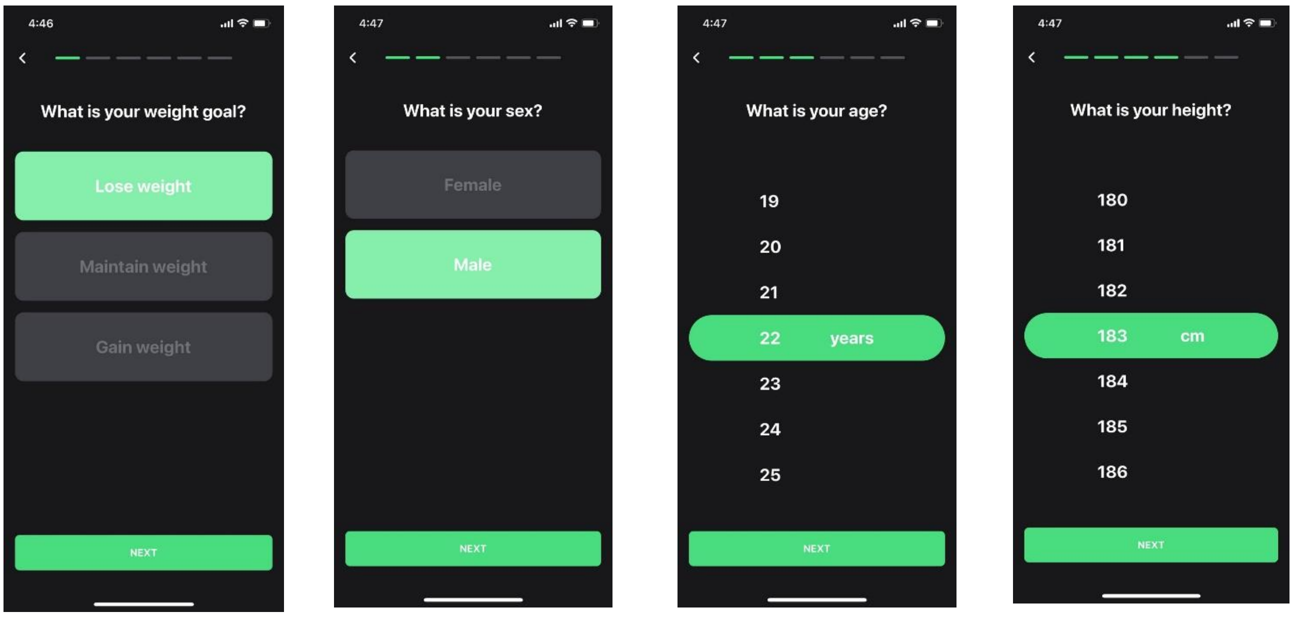


Figure 4-2. Weight Goal / Gender / Age / Height screens

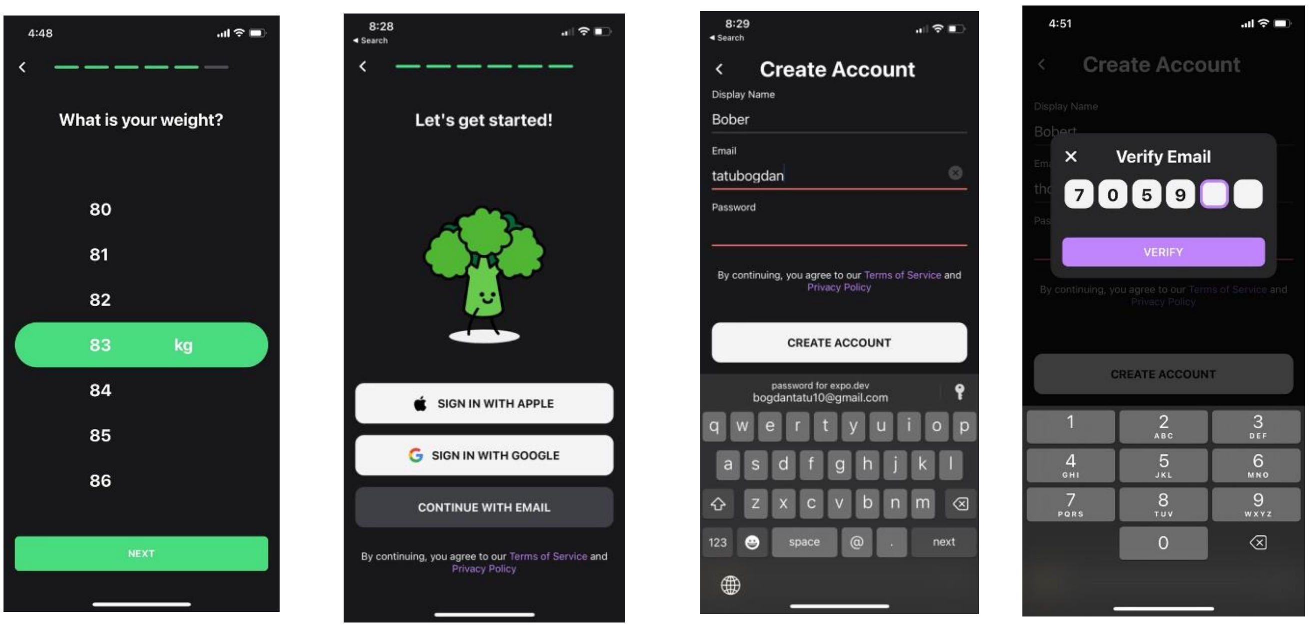


Figure 4-3. Weight / Sign Up / Email Sign Up / Verification screens

Going the other route of signing up opens up a stack of screens that facilitate the account setup, the user having to select from multiple options for personal details, such as: weight goal, gender, age, height, and weight.

Finally, after the user goes through all the form screens, he gets to the sign-up screen that has options to sign up with Apple, with Google or by email. As before, signing up with OAuth brings up an alert and after its completion the user is once again redirected to the home page.

Choosing to sing up with email, the user is transported to a new screen where he must enter their credentials to create an account (username, email, and password). If the credential entered are wrong the field borders will turn red to indicate a mistake.

When a user creates a legitimate account, a validation modal appears, and they receive an email with a verification number. When the user enters the incorrect code, red borders appear to show that something went wrong; when they enter the correct code, an account is created, and they are taken to their diary.

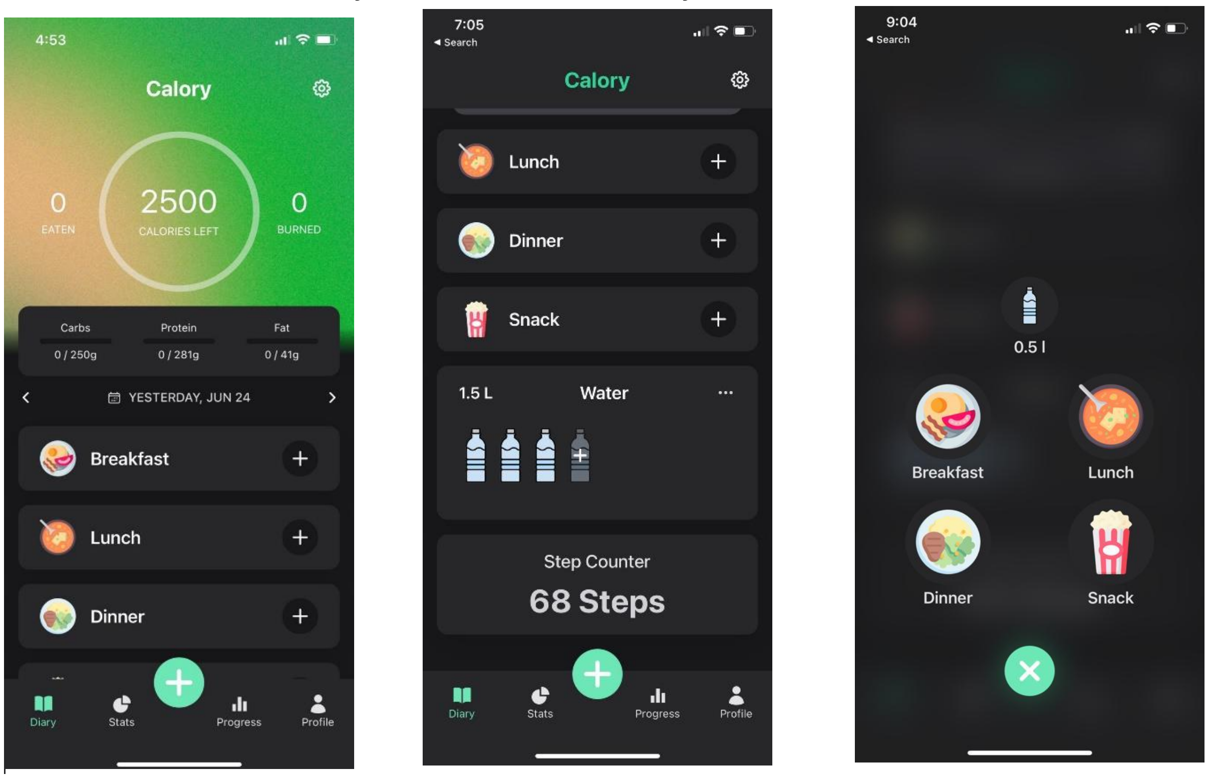
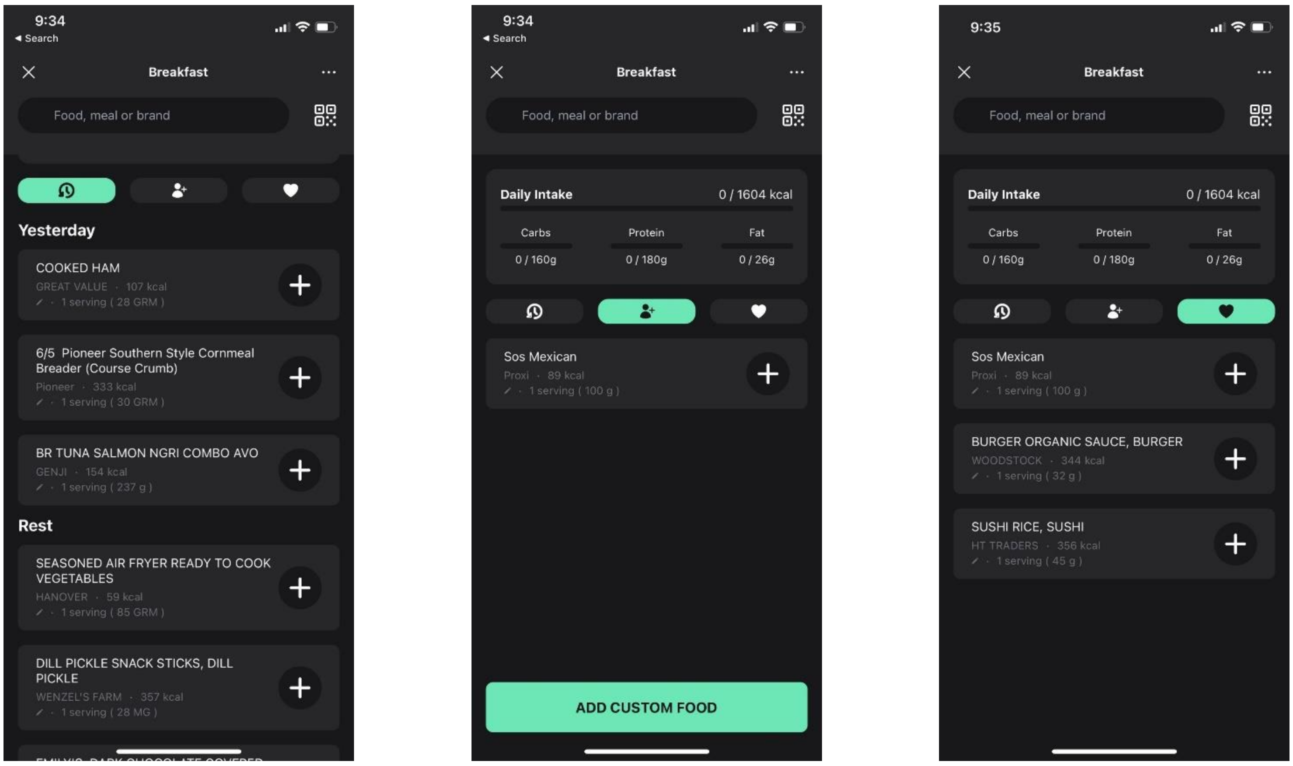


Figure 4-4. Diary / Diary Water / Diary Modal screens

The home page will present the user to diary of their meals, starting off as blank, with only the calories remaining that were calculated from the user's details, buttons to add meals (breakfast, lunch, dinner, snack), an interface to add their daily water intake, a navigation layout, and a card that they can use to turn the step counter on.

Tapping the plus button on the navigation layout brings up a hazy modal for rapidly adding to a meal or a bottle of water.



Yesterday

COOKED HAM
GREAT VALUE · 107 kcal
· 1 serving (28 GRM)

6/5 Pioneer Southern Style Cornmeal Breader (Course Crumb)
Pioneer · 333 kcal
· 1 serving (30 GRM)

BR TUNA SALMON NGRI COMBO AVO
GENJI · 154 kcal
· 1 serving (237 g)

Rest

SEASONED AIR FRYER READY TO COOK VEGETABLES
HANOVER · 59 kcal
· 1 serving (85 GRM)

DILL PICKLE SNACK STICKS, DILL PICKLE
WENZEL'S FARM · 357 kcal
· 1 serving (28 MG)

Daily Intake		0 / 1604 kcal
Carbs	Protein	Fat
0 / 160g	0 / 180g	0 / 26g

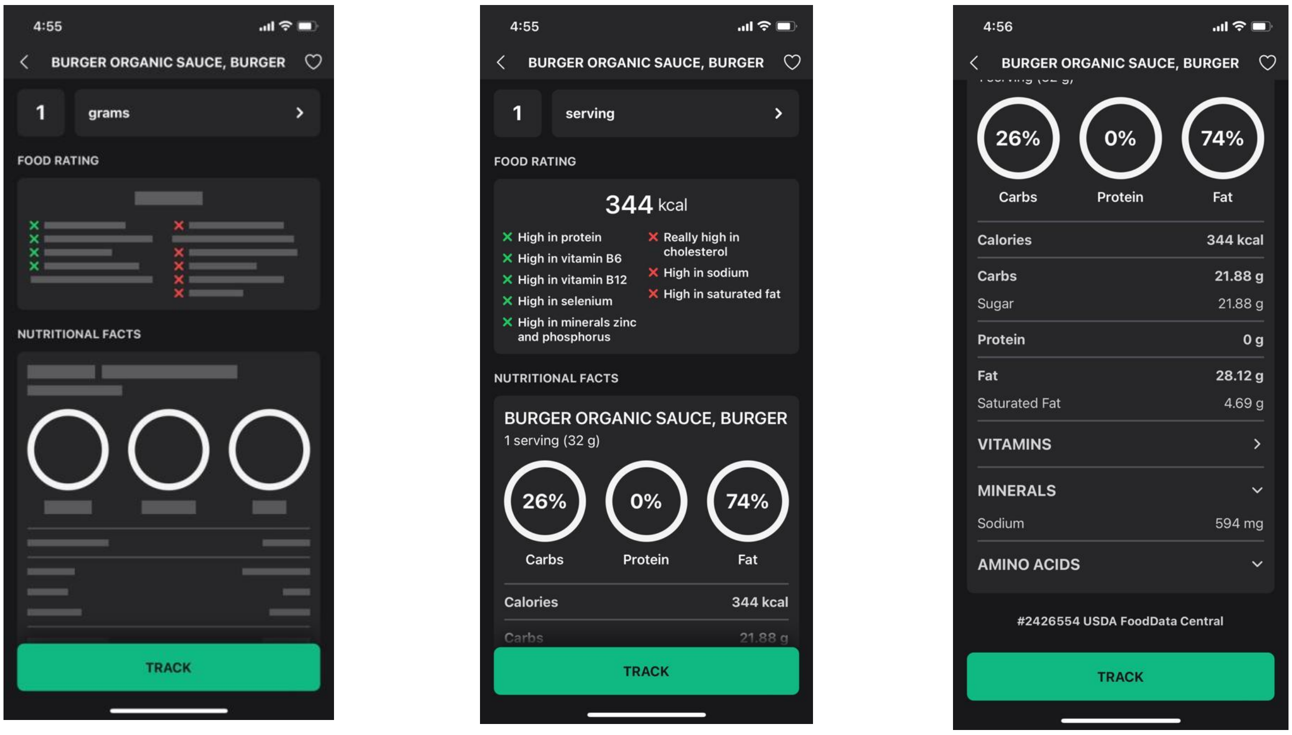
Sos Mexican
Proxi · 89 kcal
· 1 serving (100 g)

BURGER ORGANIC SAUCE, BURGER
WOODSTOCK · 344 kcal
· 1 serving (32 g)

SUSHI RICE, SUSHI
HT TRADERS · 356 kcal
· 1 serving (45 g)

Figure 4-5. Recent / Created / Favorite

Pressing the plus next to the meals or in the blurry modal, brings the user to a page where they can add food either by searching for it using the search bar, or by scanning the barcode of the food. The screen also contains a macronutrient progress card and options for recent, user-created, and favourite foods to be displayed on the bottom. Searching for a food opens up a result section of the found food, that can be directly added, or the user can go to the food page.



344 kcal

- High in protein
- High in vitamin B6
- High in vitamin B12
- High in selenium
- High in minerals zinc and phosphorus
- Really high in cholesterol
- High in sodium
- High in saturated fat

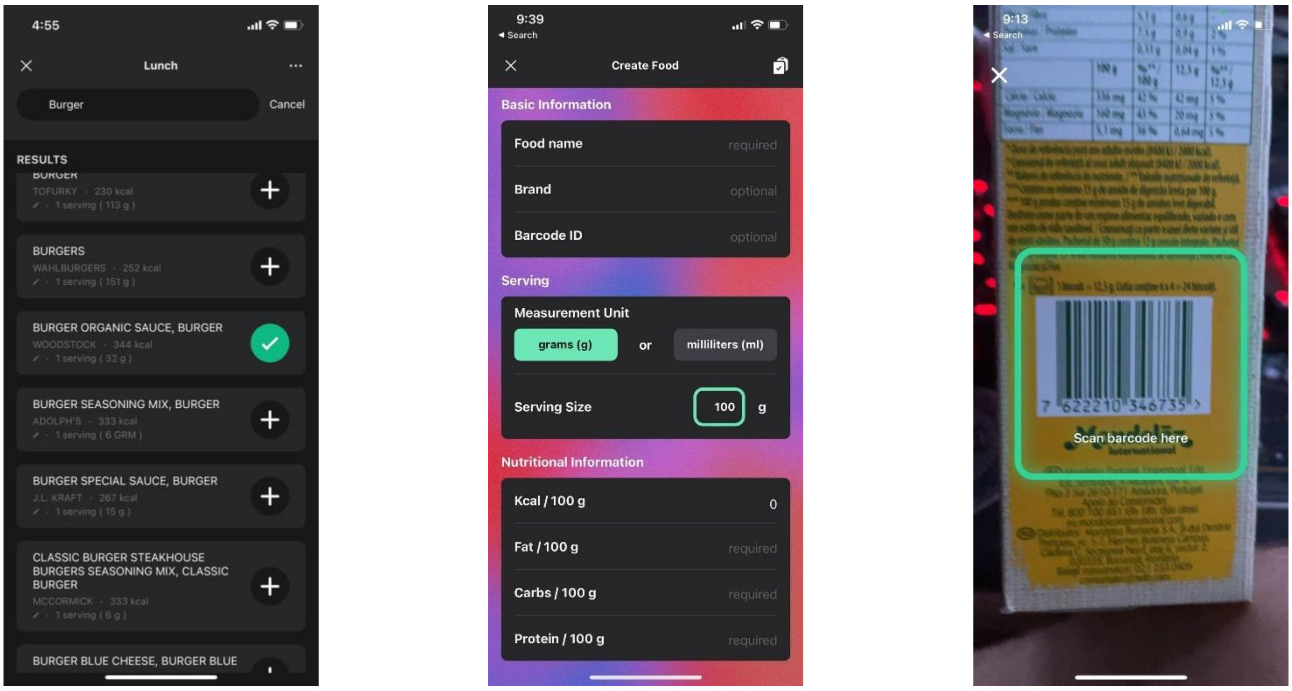
Calories	344 kcal
Carbs	21.88 g

Calories	344 kcal
Carbs	21.88 g
Sugar	21.88 g
Protein	0 g
Fat	28.12 g
Saturated Fat	4.69 g
VITAMINS	
MINERALS	
Sodium	594 mg
AMINO ACIDS	

#2426554 USDA FoodData Central

Figure 4-6. Placeholder / Id Food Rating / Id Nutrition

The food page contains data regarding the viewed food, such as the rating of the food, with pros and cons and the nutritional data (calories, carbs, protein, fat, vitamins, minerals, amino acids). The user can change the portion size, track the food, or favourite the food.


4:55
Lunch
Burger — Cancel
RESULTS

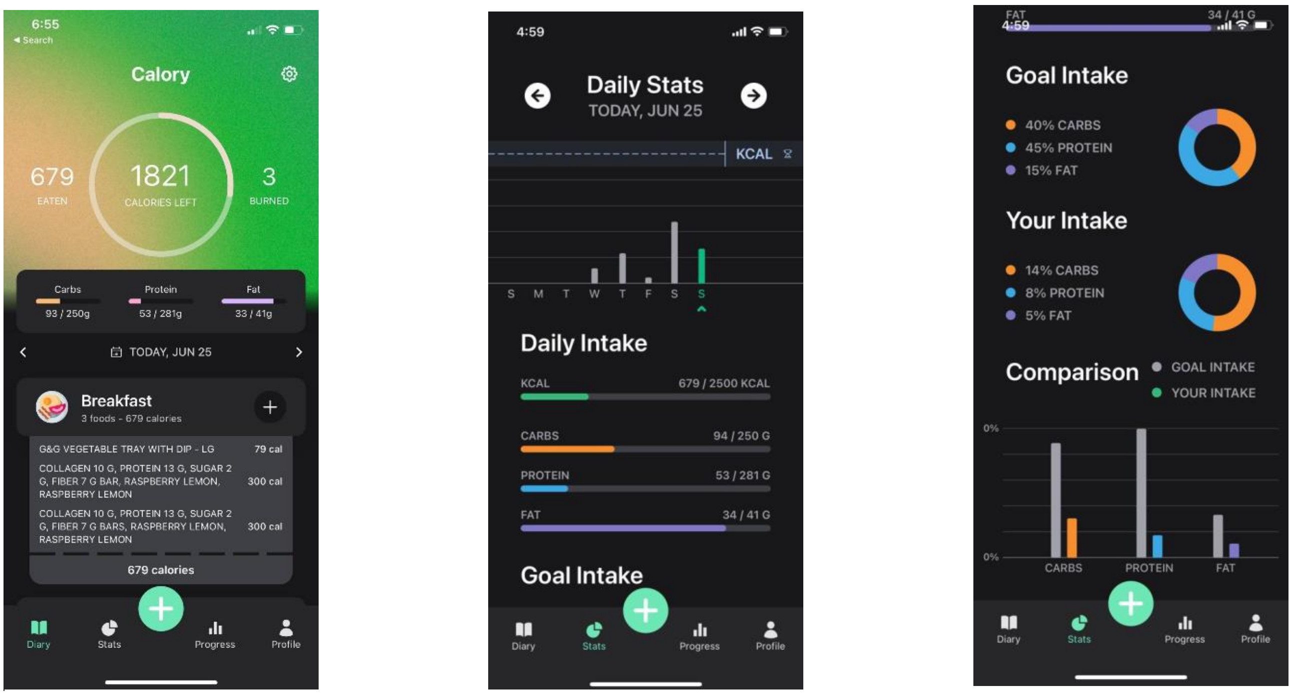
BURGER — TOFURKY · 230 kcal · 1 serving (113 g)
BURGERS — WAHLBURGERS · 252 kcal · 1 serving (151 g)
BURGER ORGANIC SAUCE, BURGER — WOODSTOCK · 344 kcal · 1 serving (32 g)
BURGER SEASONING MIX, BURGER — ADOLPH'S · 333 kcal · 1 serving (6 GRM)
BURGER SPECIAL SAUCE, BURGER — J.L. KRAFT · 267 kcal · 1 serving (15 g)
CLASSIC BURGER STEAKHOUSE BURGERS SEASONING MIX, CLASSIC BURGER — MCCORMICK · 333 kcal · 1 serving (6 g)
BURGER BLUE CHEESE, BURGER BLUE

9:39
Create Food
Basic Information
Food name — required
Brand — optional
Barcode ID — optional
Serving
Measurement Unit: grams (g) or milliliters (ml)
Serving Size: 100 g
Nutritional Information
Kcal / 100 g: 0
Fat / 100 g — required
Carbs / 100 g — required
Protein / 100 g — required

9:13
Scan barcode here

Figure 4-7. Search / Create Custom / Scan Barcode

Navigating to the statistics tab reveals the daily intake of calories and macronutrients, the user's most recent week of calories monitored, a comparison between the goal and actual intake percentages, and a breakdown of macronutrients and micronutrients, vitamins, minerals, and amino acids consumed.

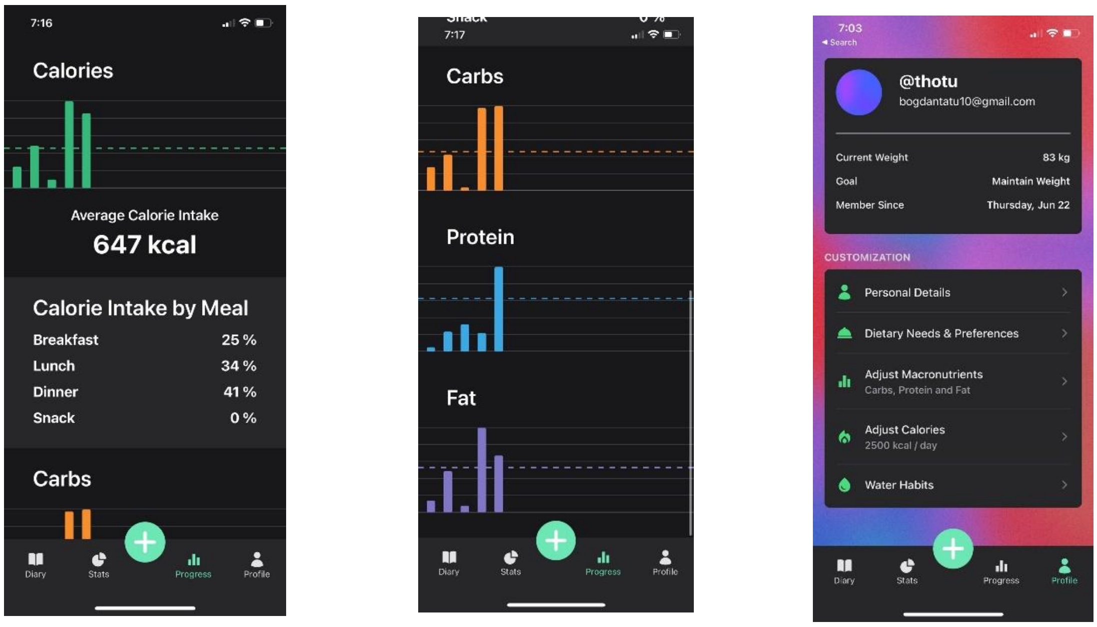

6:55
Calory
679 EATEN — 1821 CALORIES LEFT — 3 BURNED
Carbs 93 / 250g — Protein 53 / 281g — Fat 33 / 41g
TODAY, JUN 25
Breakfast — 3 foods - 679 calories
G&G VEGETABLE TRAY WITH DIP - LG — 79 cal
COLLAGEN 10 G, PROTEIN 13 G, SUGAR 2 G, FIBER 7 G BAR, RASPBERRY LEMON, RASPBERRY LEMON — 300 cal
COLLAGEN 10 G, PROTEIN 13 G, SUGAR 2 G, FIBER 7 G BARS, RASPBERRY LEMON, RASPBERRY LEMON — 300 cal
679 calories

4:59
Daily Stats — TODAY, JUN 25
KCAL
Daily Intake
KCAL — 679 / 2500 KCAL
CARBS — 94 / 250 G
PROTEIN — 53 / 281 G
FAT — 34 / 41 G
Goal Intake

Goal Intake
40% CARBS
45% PROTEIN
15% FAT
Your Intake
14% CARBS
8% PROTEIN
5% FAT
Comparison — GOAL INTAKE / YOUR INTAKE

Figure 4-8. Diary Completed / Stats / Stats Comparison screens

The progress pane displays the user's progress over a longer period of time as well as a breakdown of the percentage of calories consumed per meal type.



Average Calorie Intake
647 kcal

Calorie Intake by Meal

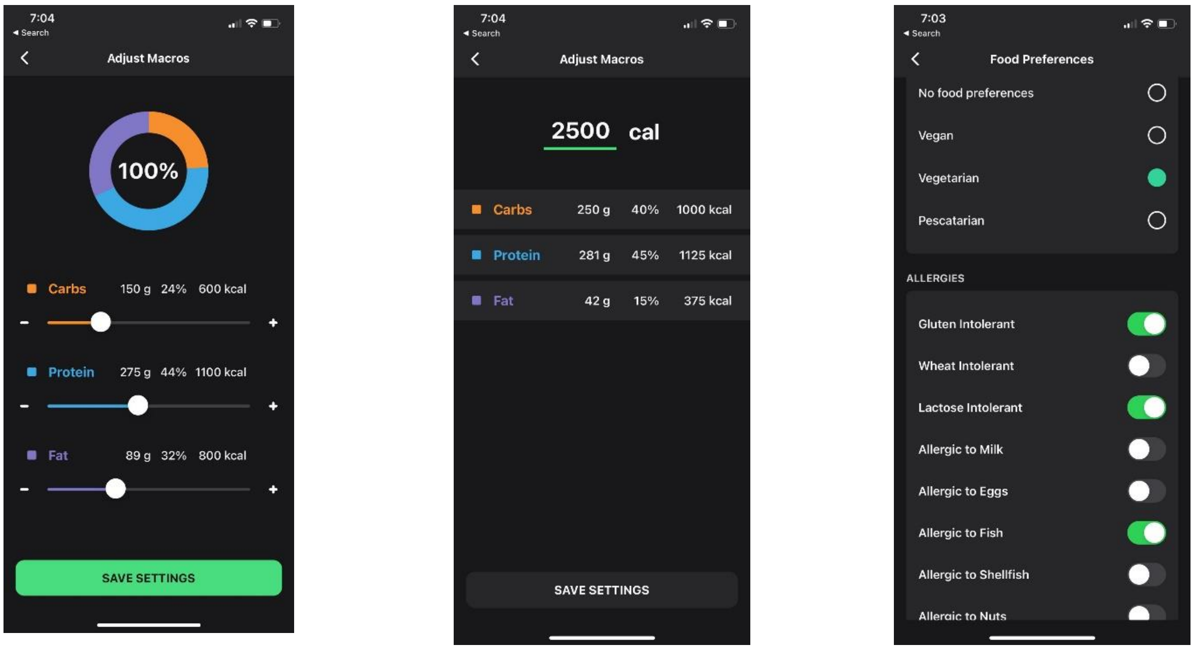
Breakfast	25 %
Lunch	34 %
Dinner	41 %
Snack	0 %

@thotu
bogdantatu10@gmail.com

Current Weight	83 kg
Goal	Maintain Weight
Member Since	Thursday, Jun 22

Figure 4-9. Progress / Progress Macros / Profile screens

On the profile tab, the user can view their username, email address, current weight, weight target, and account creation date. The user can navigate to modify his or her dietary requirements and preferences, adjust the daily macronutrient goal or daily caloric intake, and alter his or her water consumption habits.



Carbs	150 g	24%	600 kcal
Protein	275 g	44%	1100 kcal
Fat	89 g	32%	800 kcal

Carbs	250 g	40%	1000 kcal
Protein	281 g	45%	1125 kcal
Fat	42 g	15%	375 kcal

No food preferences
Vegan
[✓ Vegetarian]
Pescatarian

ALLERGIES

Gluten Intolerant	✓
Wheat Intolerant	
Lactose Intolerant	✓
Allergic to Milk	
Allergic to Eggs	
Allergic to Fish	✓
Allergic to Shellfish	
Allergic to Nuts	

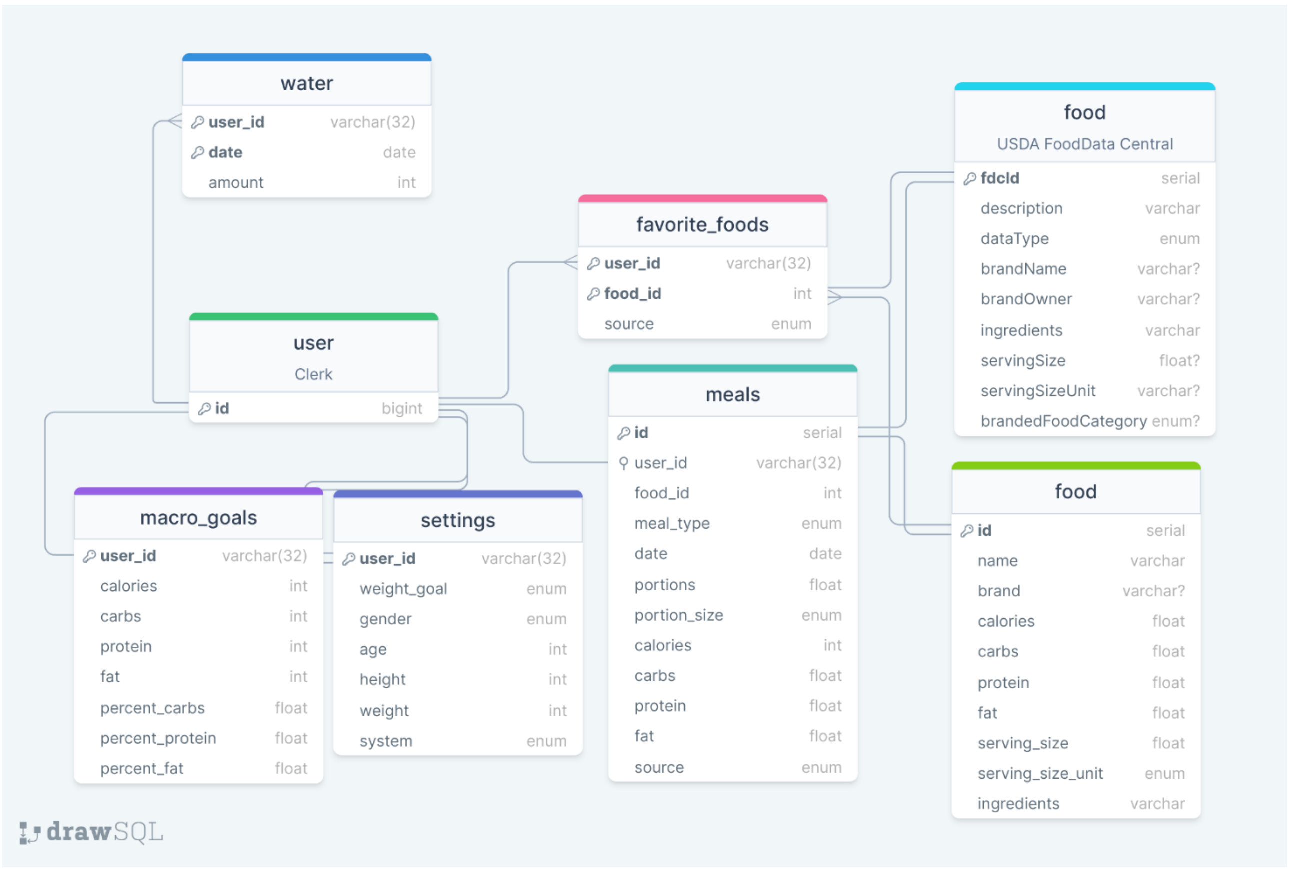
Figure 5.2-2. Adjust Macros / Adjust Calories / Food Preferences screens

The screens for adjusting macronutrients and calories only permit saving if the values have been altered, and in the case of macronutrients, the percentages must add up to 100 percent.

5. IMPLEMENTATION

5.1. BACKEND

5.1.1. DATABASE DIAGRAM



As it can be seen, the meals and favorite_foods databases are tied to either the food database food or a food from the USDA database. The returned user id of Clerk’s user object is used to set the “user_id” fields of most of the databases.

5.1.2. MIGRATIONS

Handling migrations was done using the diesel_cli tool.

A migration is a set of SQL statements that describes the modifications to the database structure. Adding, removing, or altering database objects like tables, columns, or constraints enables the database schema to change over time, without losing precious data or causing disruption to the functionality of the application. Diesel's migrations are often used to maintain consistency between the database and the application's data model, by providing a version control system.

Migrations are generated using the command “diesel migration generate <migration_name>”, creating an up.sql and a down.sql in the migrations folder.

To run migrations, you can use the “diesel migration run” command, which will execute the most recent migration.

After running, it is advised to run the “diesel migration redo” which calls the down.sql then the up.sql again to make sure that the down.sql is correct.

```sql
-- down.sql
DROP TABLE macro_goals;

-- up.sql
CREATE TABLE macro_goals (
    user_id VARCHAR(32) PRIMARY KEY,
    calories INT NOT NULL,
    carbs INT NOT NULL,
    protein INT NOT NULL,
    fat INT NOT NULL,
    percent_carbs REAL NOT NULL DEFAULT 0.5,
    percent_protein REAL NOT NULL DEFAULT 0.3,
    percent_fat REAL NOT NULL DEFAULT 0.2
)
```

In this example, we have an up.sql file, that represents the change that we want to make to the database schema, that being the creation of the macro_goals table. The down.sql file represents the SQL query used to undo the changes made by up.sql in case of a rollback or a redo.

```sql
-- down.sql
ALTER TABLE favorite_foods DROP COLUMN source;

-- up.sql
ALTER TABLE favorite_foods ADD COLUMN source source NULL;
UPDATE favorite_foods SET source = 'usda';
ALTER TABLE favorite_foods ALTER COLUMN source SET NOT NULL;
ALTER TABLE favorite_foods DROP CONSTRAINT favorite_foods_pkey;
ALTER TABLE favorite_foods ADD CONSTRAINT favorite_foods_pkey PRIMARY KEY
(user_id, food_id, source);
```

This is another example of a database migration, in which the intention was to add another column to the favorite_foods database in order to allow for controlling of the source of the added favourite food, which before was only possible for usda food, thus having to set the source of old data to ‘usda’.

This is only possible by setting source to NULL, then setting the source to the desired value, and only after that setting it to NOT NULL, unless we want to set a default value for source, which we don’t want to in this case.

Next the altering of the primary key of the table takes place, since now there should be able to exist more instances of the same ‘user_id’ and ‘food_id’ if their source is different, but no two entries should have identical ‘user_id’, ‘food_id’ and ‘source’. This is done by removing the primary key constraint and adding a new one.

5.1.3. MODELS

The models represent the inputs or the outputs of a database query. They are used for data capture and are passed to sqlx’s “query_as!” macro in order to perform the compile time checking of queries. The majority of models implement the following traits, which are generated by the "derive" macro but are omitted from the examples for brevity reasons.

```
#[derive(FromRow)]
```

The FromRow trait indicates that a record cand be queried from the database and the types which implement the trait represent the result of a SQL query.

```
#[derive(Type)]
#[sqlx(type_name = "enum_name", rename_all = "snake_case")]
```

The Type trait is used to represent a PostgreSQL enum type. It is used in tandem with the sqlx attribute, to bind the Rust enum to a PostgreSQL enum and to rename the variants of the enum using a given strategy (“snake_case”, “camel_case”, etc.) when they are used in a query.

5.1.3.1. Meal model

```
pub enum MealType {
    Breakfast,
    Lunch,
    Dinner,
    Snack,
}
pub enum PortionSize {
    Serving,
    Gram,
    Ml,
}
pub enum Source {
    User,
    Usda,
}

pub struct Meal {
    pub id: i32,
    pub user_id: String,
    pub food_id: i32,
    pub meal_type: MealType,
    pub date: chrono::NaiveDate,
    pub portions: f32,
    pub portion_size: PortionSize,
    pub calories: i32,
    pub protein: f32,
    pub carbs: f32,
    pub fat: f32,
    pub source: Source,
}
```

The ‘meal-model’ module defines several data structures and enumerations.

The ‘Meal’ struct represents a user-added meal with its own id field and primary key for a given user with ‘user_id’ and the added food with ‘food_id’. It also contains information about the meal type that it was added for, that being the meals of the day, breakfast, lunch and dinner, or a snack, specified by the ‘MealType’ enum, the date on which the meal was added, and the proportion of the meal, comprised of the number of portions and the portion size.

The ‘source’ field denotes the database from which the food was added, be it a user made food, or a food from the USDA database.

Additionally, the struct also contains duplicate fields that could’ve been acquired from the ‘Food’ specified by ‘food_id’ and calculated given the ‘portion’ and ‘portion_size’ fields. These fields in question are “calories”, “carbs”, “protein”, and “fat” – the duplication serving the purpose of mitigating the cost of another request to the USDA API to get the 'Food' data for some requests.

5.1.3.2. SETTINGS MODEL

```
pub enum Gender {
    Male,
    Female,
}
pub enum System {
    Metric,
    Imperial,
}
pub enum WeightGoal {
    Lose,
    Maintain,
    Gain,
}

pub struct Settings {
    pub user_id: String,
    pub weight_goal: WeightGoal,
    pub gender: Gender,
    pub age: i32,
    pub height: i32,
    pub weight: i32,
    pub system: System,
}
```

This module also defines multiple enums that pertain to the 'Settings' struct.

The struct represents the initial personal details provided by the user during the account creation process that are used to create an informed dietary plan for the user, with a given macronutrient intake.

The details provided represent the weight goal of the user, this being the intention to lose, maintain or gain weight, the gender and age of the user. It also contains details with meanings dependant on the system of measurement used (imperial or metric), height representing centimetres in the metric system and inches in imperial system, respectively, weight, that represents either kilograms or pounds.

The primary key of the model is the 'user_id' since each user will have one and only one 'Settings' data tied to them.

5.1.3.3. Macro Goal model

```rust
pub struct MacroGoal {
    pub user_id: String,
    pub calories: i32,
    pub carbs: i32,
    pub protein: i32,
    pub fat: i32,
    pub percent_carbs: f32,
    pub percent_protein: f32,
    pub percent_fat: f32,
}
```

The struct ‘MacroGoal’ represents the goal macronutrient intake of the user, inserted into the database alongside the ‘Settings’ row.

The fields of this struct represent the goal caloric intake for a day, ‘calories’, measured in kilocalories, and the target weight of each macronutrient (in grams), ‘carbs’, ‘protein’, and ‘fat’.

The remaining three fields represent the target percentage of macronutrients eaten throughout the day, ‘percent_carbs’, ‘percent_protein’, and ‘percent_fat’. Each of these fields are, in part, set by default to values of 0.5 (50%), 0.3 (30%), respectively 0.2 (20%).

5.1.3.4. Favourite Food model

```rust
pub struct FavoriteFood {
    pub user_id: String,
    pub food_id: i32,
    pub source: Source,
}
```

The ‘FavoriteFood’ struct represents a single instance of one of the favourite foods set by the user. Its primary key is given by both fields combined.

It is an association between a person and a food, and it forms a many-to-many relationship connectiong them.

The source is the same as in the meals example.

5.1.3.5. Water model

```rust
pub struct Water {
    pub user_id: String,
    pub date: chrono::NaiveDate,
    pub amount: i32,
}
```

The 'Water' struct represents the water intake of a user on a given day, indexed by the compound primary key given by 'user_id' and 'date'.

5.1.4. Repositories

Each repository is associated with one or more models and represents the capabilities to query and retrieve data, insert new or updated data, or delete data from the database. The functions are turned into SQL by the ORM.

```
        pub fn function_name(connection: &PgPool, . . . ) -> sqlx::Result<_>
```

All functions will be passed a "connection" parameter representing the PostgreSQL connection pool, so that the appropriate queries will be passed to and executed by the database management system.

All repository functions return a generic "sqlx::Result<T>" type, returning either the generic type specified by the developer or an "sqlx::Error", denoting errors during the querying of the database (row not found, pool closed, decoding error, etc.)

For the sake of brevity, the connection parameter has been left out of the following repository code examples.

5.1.4.1. Macro Goal repo

The macro goal repository module defines the functionality for handling users' macronutrient goal settings. It includes the following functions:

```rust
pub async fn get_by_uid(uid: &str) -> Result<Option<MacroGoal>> {
    let macro_goal = sqlx::query_as!(
        MacroGoal,
        r#"
        SELECT user_id, calories, carbs, protein, fat, percent_carbs,
            percent_protein, percent_fat
        FROM macro_goals
        WHERE user_id = $1
        "#,
        uid
    )
    .fetch_optional(conn)
    .await?;

    Ok(macro_goal)
}
```

The "get_by_uid" function retrieves the macronutrient goal of a user from the database given by the user's id and also the table's primary key, "uid".

It uses the "query_as!" macro to statically check the validity of the query and to type-check the return of the query, so that the MacroGoal struct is compatible with the "SELECT" statement. The macro also binds the rest of the parameters to each of the “$” variables in the query.

After creating the query, “fetch_optional” tries to fetch the first row generated by the query. It returns a Some(row) variant if it succeeds, a None variant if the query was successfully ran, but the database returned 0 rows, or an error otherwise, that is propagated up the call chain.

```rust
pub async fn create(new_macro_goal: &MacroGoal) -> Result<()> {
    sqlx::query!(
        r#"
        INSERT INTO macro_goals (user_id, calories, carbs, protein, fat)
        VALUES ($1, $2, $3, $4, $5)
        "#,
        new_macro_goal.user_id,
        new_macro_goal.calories,
        new_macro_goal.carbs,
        new_macro_goal.protein,
        new_macro_goal.fat,
    )
    .execute(conn)
    .await?;

    Ok(())
}
```

The “create” function performs a query to insert all the necessary values into the “macro_goals” table.

Since the query doesn't return any data, it doesn't need to type-check the query's return value, therefore it utilizes the "query!" macro rather than the "query_as!" from the prior example.

Additionally, every field of the MacroGoal object is bound at the positions "$1", "$2", "$3", "$4", and "$5". Then the query is executed with the given connection and all the errors are passed up to the caller of the function.

```
pub async fn update(macro_goal: &MacroGoal) -> Result<()> {
    sqlx::query!(
        r#"
        UPDATE macro_goals
        SET calories = $2, carbs = $3, protein = $4, fat = $5
        WHERE user_id = $1
        "#,
        macro_goal.user_id,
        macro_goal.calories,
        macro_goal.carbs,
        macro_goal.protein,
        macro_goal.fat,
    )
    .execute(conn)
    .await?;

    Ok(())
}
```

The “update” function has similar syntax to the “create” function.

Both functions either return an error, or Rust’s unit primitive type, “()”, since no data has to be passed back to the controllers, the controllers only needing to know if the query succeeded.

5.1.4.2. MEAL REPO

The meal repository contains a lot of functionality related to the user created meals, to be able to get meals depending on a range of criteria.

```
       pub async fn get_by_user(uid: &str) -> Result<Vec<Meal>> {
           let meals = sqlx::query_as!(
               Meal,
               r#"
               SELECT id, user_id, food_id, meal_type AS "meal_type: _", date,
                   portions,  portion_size  AS  "portion_size:  _",  calories,
protein,
                   carbs, fat, source AS "source: _"
               FROM meals
               WHERE user_id = $1
               "#,
               uid
           )
           .fetch_all(conn)
           .await?;

           Ok(meals)
       }
```

The “get_by_user” function gets all the meals of a given user.

After creating the query, the "fetch_all" function is invoked in order to extract all the rows returned by the query, in the form of a Vec<Meal> type. If no rows have been found, it simply returns an empty vector.

```
        pub async fn get_total_macro_intake_per_day_between_dates_for_user(
            uid: &str,
            start_date: &chrono::NaiveDate,
            end_date: &chrono::NaiveDate,
        ) -> Result<Vec<MealGroup>> {
            let meals = sqlx::query_as!(
                MealGroup,
                r#"
                   SELECT  date,  CAST(SUM(calories)  AS  INTEGER)  AS  calories,
SUM(protein)
                    AS protein, SUM(carbs) AS carbs, SUM(fat) AS fat
                FROM meals
                WHERE user_id = $1 AND date >= $2 AND date <= $3
                GROUP BY date
                ORDER BY date ASC
                "#,
                uid,
                start_date,
                end_date
            )
            .fetch_all(conn)
            .await?;

            Ok(meals)
        }
```

The aptly named "get_total_macro_intake_per_day_between_dates_for_user" function is a bit more complex. First off, it finds the meals of the user between the given dates, it groups the results by date, and calculates the sum of each of the macronutrients of each day, then it orders the results in an ascending manner by their date.

The final output is loaded into the MealGroup struct represented by:

```
        pub struct MealGroup {
            pub date: chrono::NaiveDate,
            pub calories: Option<i32>,
            pub protein: Option<f32>,
            pub carbs: Option<f32>,
            pub fat: Option<f32>,
        }
```

All the macronutrient fields need to be Options since the "SUM" SQL function can error, which is transformed by SQLx into a None variant.

```
pub async fn get_average_macro_intake_per_meal_type_between_dates_for_user(
    uid: &str,
    start_date: &chrono::NaiveDate,
    end_date: &chrono::NaiveDate,
) -> Result<Vec<MealAveragePerMealType>> {
    let meals = sqlx::query_as!(
        MealAveragePerMealType,
        r#"
        SELECT meal_type AS "meal_type: _", CAST(AVG(calories) AS INTEGER) AS
            calories, CAST(AVG(protein) AS REAL) AS protein, CAST(AVG(carbs)
            AS REAL) AS carbs, CAST(AVG(fat) AS REAL) AS fat
        FROM meals
        WHERE user_id = $1 AND date >= $2 AND date <= $3
        GROUP BY meal_type
        "#,
        uid,
        start_date,
        end_date
    )
    .fetch_all(conn)
    .await?;

    Ok(meals)
}
```

The “get_average_macro_intake_per_meal_type_between_dates_for_user” filters the database for meals of the user between the specified dates, groups the results by meal type (“breakfast”, “lunch”, “dinner”, “snack”) and gets the average of all the meals for each of the meal types to show the user’s distribution of macronutrients per meal type.

```
pub async fn delete(mid: i32) -> Result<()> {
    let _result = sqlx::query!(
        r#"
        DELETE FROM meals
        WHERE id = $1
        "#,
        mid
    )
    .execute(conn)
    .await?;

    Ok(())
}
```

The “delete” function is similar to both creating and updating functions.

5.1.4.3. FAVORITE FOOD REPO

```rust
pub async fn get_by_user(uid: &str) -> Result<Vec<i32>> {
    struct FoodId {
        food_id: i32,
    }

    let ids = sqlx::query_as!(
        FoodId,
        r#"
            SELECT food_id
            FROM favorite_foods
            WHERE user_id = $1
        "#,
        uid
    )
    .fetch_all(conn)
    .await?
    .into_iter()
    .map(|f| f.food_id)
    .collect();

    Ok(ids)
}
```

The "get_by_user" function retrieves the ids of a user's favorite foods by selecting the "food_id" column from the resultant rows.

The “query_as!” macro cannot be used with primitive types such as i32, accepting only struct, variant or union types, meaning that we have to create a local struct like “FoodId” to store the return data of the query. The process of transforming a Vec<FoodId> into a Vec<i32> is straightforward, by iterating over all of the initial values, mapping a function to extract the “food_id” fields of the objects and collecting them back into a vector.

The selection of only the ids was done because favorite foods can have different sources, one being the internal food table and the other being the external USDA database, from which the controller will request the food data for the specified ids.

```
pub async fn get_created_by_user(uid: &str) -> Result<Vec<Food>> {
    let foods = sqlx::query_as!(
        Food,
        r#"
        SELECT id, food.user_id, name, brand, barcode, calories, carbs,
            protein, fat, serving_size, serving_size_unit AS
            "serving_size_unit: _", ingredients
        FROM food
        INNER JOIN favorite_foods ON food.id = favorite_foods.food_id
        WHERE favorite_foods.user_id = $1 AND favorite_foods.source = 'user'
        "#,
        uid
    )
    .fetch_all(conn)
    .await?;

    Ok(foods)
}
```

The “get_created_by_user” function gets all user-created favorite foods of a user.

It does this by executing an INNER JOIN between the food and the favorite_foods tables, binding them through the food id, the primary key in the food table, for a given user and the source being that of ‘user’, since the ‘usda’ foods are not stored inside our database.

5.1.5. Controllers

Controllers are the modules that handle the HTTP requests sent from the client-side. The functions have this signature:

```
pub async fn controller_fn(req: Request<PgPool>) -> tide::Result
```

They take in a Request object with a PgPool as the state, from which the query parameters, query body and state can be extracted. The functions return a “tide::Result”, that either unwrap into a “tide::Response”, which is just a status code, with an optional body and header, or a “tide::Error”.

5.1.5.1. Error Handling

Error handling is made easier with the creation of a generic trait “MapErrorToServerError<T>”, to convert from one crate’s error type to another’s.

```
pub trait MapErrorToServerError<T> {
    fn map_err_to_server_error(self) -> Result<T, tide::Error>;
}

impl<T> MapErrorToServerError<T> for Result<T, sqlx::Error> {
    fn map_err_to_server_error(self) -> Result<T, tide::Error> {
        self.map_err(|e| tide::Error::new(StatusCode::InternalServerError, e))
    }
}
```

The trait needs to be created in order to implement it for a type that isn’t the developer’s own. In this case for the Result struct from the standard library, for transforming an “sqlx::Error” database error into a “tide::Error” to make the code easier to follow. It also keeps the original error message, but also attaches an Internal Server Error status code to the new error.

Transforming this:

```
let food = match food_repo::get_by_id(connection, food_id).await {
    Ok(Some(food)) => food,
    Ok(None) => {
        return Ok(/* Error Message */)
    }
     Err(e) => return Err(Error::new(StatusCode::InternalServerError, e)),
};
```

Into this:

```
let Some(food) = food_repo::get_by_id(connection, food_id).await
    .map_err_to_server_error()?
else {
    return Ok(/* Error Message */)
};
```

Getting rid of the match statement entirely. Rust also has a “let-else” pattern, making the code cleaner and more readable.

Another way to make the code cleaner was to use declarative macros to generate responses, error messages and errors.

```
macro_rules! response {
    ($status:expr, $body:expr) => {
        tide::Response::builder($status)
            .body(serde_json::json!($body))
            .build()
    };
    ($status:expr) => {
        tide::Response::builder($status).build()
    };
}
```

The “response!” macro handles the creation of “tide::Response” objects with a status code and an optional body.

```
response!(StatusCode::Ok, foods)
response!(StatusCode::Created)
```

```
pub struct ErrorMessage {
    pub message: String,
    pub error: String,
}

impl ErrorMessage {
    pub fn new(error: &str, message: &str) -> ErrorMessage {
        ErrorMessage {
            message: message.to_string(),
            error: error.to_string(),
        }
    }

    pub fn res(self, status: StatusCode) -> Response {
        let mut res = Response::builder(status).body(json!(self)).build();
        res.set_error(Error::new(status, anyhow::Error::from(self)));

        res
    }
}

macro_rules! error_message {
    ($status:expr, $error:expr, $message:expr) => {
        $crate::controllers::utils::tide::ErrorMessage::new($error, $message)
            .res($status)
       };
}
```

The “error_message!” macro is also used to create a Respone with a payload that has the status code, an error, and a message to let the client know what exactly caused the error, rather than only sending back a status code. It also sets an error onto the response for logging purposes.

```
error_message!(
    tide::StatusCode::BadRequest,
    "invalid-source",
    "Invalid source. Must be one of: user, usda"
)
```

5.1.5.2. Macro Goal Controller

```
pub async fn get_macro_goal(req: Request<PgPool>) -> Result {
    let user_id = req.param("uid")?;

    let conn = req.state();

    let Some(macro_goal) = macro_goal_repo::get_by_uid(conn, user_id).await
        .map_err_to_server_error()?
    else {
        return Ok(error_message!(
            tide::StatusCode::BadRequest,
            "no-macro-goal",
            "No macro goal found for user"));
    };

    Ok(response!(StatusCode::Ok, macro_goal))
}
```

An example of a concise controller function can be this one, handling the request for the macronutrient goal data for a given user.

It extracts the “uid” request parameter, gets the connection pool from the requests state, tries to fetch the macronutrient goal data from the database given the user id, and returning a response with an Ok status code and the fetched data.

The function exits early if it encounters any errors, either by returning in the else statement or through Rust’s “?” operator, that propagates the error.

```
pub async fn get_progress(req: Request<PgPool>) -> Result {
    let user_id = req.param("uid")?;
    let date_from = req.param("date_from")?;
    let date_to = req.param("date_to")?;

    let Ok(date_from) = NaiveDate::parse_ymd(date_from) else {
        return Ok(error_message!(
            StatusCode::BadRequest,
            "invalid-date-from-format",
            "Invalid date from format. Format must be YYYY-MM-DD"
        ));
    };
    let Ok(date_to) = NaiveDate::parse_ymd(date_to) else {
        return Ok(error_message!(
            StatusCode::BadRequest,
            "invalid-date-to-format",
            "Invalid date to format. Format must be YYYY-MM-DD"
        ));
    };

    let connection = req.state();

    let Some(goals) = macro_goal_repo::get_by_uid(connection, user_id)
        .await
        .map_err_to_server_error()?
    else {
        return Ok(error_message!(
            StatusCode::NotFound,
            "macro-goal-not-found",
            "Macro goal not found"
        ));
    };

    let meals = meal_repo::
        get_total_macro_intake_per_day_between_dates_for_user(
            connection, user_id, &date_from, &date_to,
        )
        .await
        .map_err_to_server_error()?;

    if meals.is_empty() {
        return Ok(response!(StatusCode::Ok, ProgressDto::empty()));
    }

    let averages_per_meal_type = meal_repo::
        get_average_macro_intake_per_meal_type_between_dates_for_user(
            connection, user_id, &date_from, &date_to,
        )
        .await
        .map_err_to_server_error()?;

    let data = ProgressDto::new(goals, meals, averages_per_meal_type);

    Ok(response!(StatusCode::Ok, data))
}
```

A more complex request can be denoted by the HTTP get request for the progress of a user between two dates.

Firstly, it extracts the “user_id”, “date_from” and “date_to” parameters and validates them. It gets the connection pool. It fetches the macronutrient goals of a user and the macronutrient intake of the user between the dates. If no totals were found, the function returns an Ok status code and an empty ProgressDto, otherwise it fetches the users averages per meal type between the dates and creates a ProgressDto from all of the fetched data.

5.1.6. Data Transfer Objects

Data Transfer Objects are used to transfer object to and from the server-side. DTOs encapsulate data and provide a standard format for data transmission between different components and between the server and the client. They offer a concise representation of the data that needs to be transferred by only including the necessary field and excluding any irrelevant or sensitive data.

An excellent example is the "FoodDto", which not only serves as a conduit between the frontend and the backend, but also has a common structure for combining the use of USDA foods and user-created foods.

```rust
pub struct FoodDto {
    pub id: i32,
    pub name: String,
    pub brand: Option<String>,
    pub calories: f32,
    pub serving_size: f32,
    pub serving_size_unit: String,
    pub alternative_serving_size: Option<String>,
    pub verified: bool,
    pub nutrients: NutrientsDto,
    pub vitamins: HashMap<String, f32>,
    pub minerals: HashMap<String, f32>,
    pub amino_acids: HashMap<String, f32>,
    pub source: Source,
}
pub struct NutrientsDto {
    pub carbs: f32,
    pub fiber: Option<f32>,
    pub sugar: Option<f32>,
    pub protein: f32,
    pub fat: f32,
    pub saturated_fat: Option<f32>,
    pub unsaturated_fat: Option<f32>,
}
```

The DTOs contain all the information needed of a food to be displayed on the client-side and is a common ground between the two food structs. Some DTOs also implement Into and From traits to make it easier to convert between objects returned from the database and DTO objects.

```
impl From<USDABranndedFoodItemDto> for FoodDto
impl From<Food> for FoodDto
```

5.1.7. Main

The main file is where the server is created and gets launched from.

```
#[async_std::main]
async fn main() -> tide::Result<()>
```

The async runtime is handled by the async_std crate, by specifying the async_std::main attribute applied to the main function.

```
async fn main() -> tide::Result<()> {
    femme::start();

    start_server().await?;

    Ok(())
}
```

Femme, a rust logging crate, is started to provide a logger for both Tide’s and Surf’s middlewares. Then the server is created and started, and any error or panic it produces is handled by the async_std runtime.

```
let mut app: tide::Server<()> = tide::new();
```

The server object is created, and the necessary middleware is attached to it, alongside the REST API routes.

```
app.with(LogMiddleware::new());
app.with(After(|res: Response| async move { … }));
```

The logging and the error converter middlewares are atttched to the root of the server.

```
app.at("/").get(|_| async { Ok("Running!") });
```

A testing route is created and attatched to see if the server is running.

```
app.at("/api/v2").nest({
    let mut api = tide::with_state(db::create_pool().await);
    /* routes */
});
```

The actual routes are nested under “/api/v2” path, and each lower route will be created using a tide Server with the state containing the database connection pool.

```
pub async fn create_pool() -> PgPool {
    dotenv().ok();

    let database_url = env::var("DATABASE_URL")
        .expect("DATABASE_URL must be set");

    let options = PgConnectOptions::from_str(&database_url).unwrap();

    PgPoolOptions::new()
        .max_connections(5)
        .connect_with(options)
        .await
        .unwrap_or_else(|_| panic!("Error connecting to {database_url}"))
}
```

The connection to the PostgreSQL Database is done with the help of “dotenvy”, a package for working with ‘.env’ files. The “DATABASE_URL” environment variable is read from the file and the PgPool object is created by establishing a connection with the url, also specifying the number of maximum connections to the pool.

```
app.at("/api/v2").nest({
    let mut api = tide::with_state(db::create_pool().await);

    api.at("/user/:uid").nest({
        let mut user = tide::with_state(db::create_pool().await);

        user.with(UserMiddleware::new());

        user.at("/meals").nest({
            let mut meals = tide::with_state(db::create_pool().await);

            meals.at("/:date").get(meals::get_meals);
            meals.at("/recent").get(meals::get_recent_meals);
            meals.at("/").post(meals::post_meal);
            meals.at("/:mid").delete(meals::delete_meal);

            meals
        });
        / * ... * /
        user;
    });
    / * ... * /
    api
});
```

The user path is nested below the API, together with the UserMiddleware. Below that, the meals path is nested with endpoints created for different REST request types.

The first endpoint becoming:

```
"api/v2/user/:uid/meals/:date"
```

With two request parameters, “uid” and “date”.

```
app.listen("0.0.0.0:3000").await?;
```

The server is made to listen on localhost on port 3000.

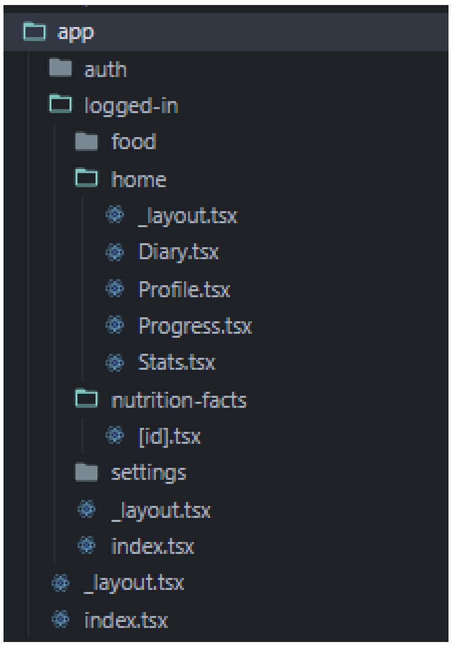
5.1.7.1. MIDDLEWARE

Middleware structs need to implement the Middleware<State> trait provided by Tide, which takes in a Request object

```
#[async_trait::async_trait]
impl Middleware<State> for UserMiddleware {
    async fn handle(&self, req: Request<State>, next: Next<'_, State>) ->
        tide::Result
}
```

5.2. FRONTEND

The frontend of the application, that was developed using React Native with Expo, with the routs handled by Expo Router, that utilises a file-based routing technique that maps each route to a specific file in the app directory on the server, providing static or dynamic routing capabilities.



```
app
  auth
  logged-in
    food
    home
      _layout.tsx
      Diary.tsx
      Profile.tsx
      Progress.tsx
      Stats.tsx
    nutrition-facts
      [id].tsx
    settings
    _layout.tsx
    index.tsx
  _layout.tsx
  index.tsx
```

The example above presents some of the routes or the screens used in the front end, with the entry point of the application being the index.tsx file that is the child of the app directory. Expo router also handles dynamic routes, such as the [id].tsx file that represents "app/logged-in/nutrition-facts/<id>", which can utilize the id in the code to get nutritional information for the food with the given id. Expo also comes with the ability of specifying _layout.tsx files that execute their code in all the routes that are at or below that file.

5.2.1. APP.TSX

App.tsx is the entry point of the application, which is designated inside the application by the 'index.tsx' file that is changed by Expo Router into 'App.tsx'. It is wrapped by the Root component, which is contained in the topmost '_layout.tsx' file, shown below, and gives functionality to it:

```
const queryClient = new QueryClient();

const Root = () => {
    return (
        <ClerkProvider
            publishableKey={CLERK_PUBLISHABLE_KEY}
            tokenCache={tokenCache}
        >
            <QueryClientProvider client={queryClient}>
                <Slot />
            </QueryClientProvider>
        </ClerkProvider>
    );
};
```

The Slot component is the Expo Router alternative to React’s children prop, specifying that each component that is used with this layout will go in the place of the Slot component.

The ClerkProvider and the QueryClientProvider contexts are made available to the lower routes via the use of the Root component, which encapsulates the whole of the application.

The QueryClientProvider, imported from TanStack’s React Query, provides the application with the librarie’s hooks that handle asynchronous data fetching and caching and the multitude of state changes that take occur during the function calls.

The usage of the ClerkProvider context enables the use of its hooks, which concentrate on user management and authentication inside the application. To provide user persistence between sessions, it requires an API key and a token cache and utilizes the JWT tokens described below:

```
export const tokenCache = {
    async getToken(key: string) {
        try {
            return SecureStore.getItemAsync(key);
        } catch (e) {
            console.log(e);
            return null;
        }
    },
    async saveToken(key: string, value: string) {
        try {
            return SecureStore.setItemAsync(key, value);
        } catch (e) {
            console.log(e);
        }
    }
}
```

It defines an asynchronous getter and setter (getToken and saveToken) of the Token using Clerk's SecureStore for persistence.

```
const App = () => {
    const { isSignedIn, isLoaded } = useUser();

    if (!isLoaded) {
        return <LoadingPage />;
    }

    if (!isSignedIn) {
        return <Redirect href={page.auth.authentication} />;
    }

    return <Redirect href={page.home.diary} />;
};
```

The App component only handles the redirects to the home page (page.home.diary) if the user is signed in, otherwise it redirects the user to the authentication page (page.auth.authentication). The useUser() hook handles the user's authentication status, and if it has not been loaded, the application displays a loading page until everything is finished loading.

5.2.2. Layouts

5.2.2.1. Account-setup layout

Another notable layout used in the application includes the layout in account-setup:

```
const Layout: React.FC = () => (
    <SetupProvider>
        <CreateAccountLayout>
            <Stack
                initialRouteName='accout-creation/WeightGoal'
                screenOptions={{
                    headerShown: false,
                    animation: 'slide_from_right'
                }}
            />
        </CreateAccountLayout>
    </SetupProvider>
)
```

The layout makes use of a stack layout from Expo Router, which is comparable to the Slot component but has the advantage of offering customized headers and screen transitions.

The Stack is wrapped inside of the CreateAccountLayout, which is a UI layout that displays a progress bar during the account creation process, a question prompt for the setup fields, and a back and next button to transition between screens.

The SetupProvider is the context that envelops the stack of screens, to provide easier access to state and relevant functionality throughout the account creation screens, making use of the created useSetup() hook, without having to pass down state throughout the components.

```
interface SetupContext {
    . . .
}

const SetupContext = createContext({} as SetupContext);

export const useSetup = () => {
    return useContext(SetupContext);
};
```

Firstly, we create the context the the useSetup() hook that uses that context.

```
export const SetupProvider: React.FC<{
    children: React.ReactNode;
}> = ({ children }) => {
    . . .
    const [weightGoal, setWeightGoal] = useState<WeightGoalOptions>('');
    const [gender, setGender] = useState<GenderOptions>('');
    const [age, setAge] = useState<number>(20);
    const [height, setHeight] = useState<HeightOptions>({
        type: 'metric',
        cm: 170
    })
    const [weight, setWeight] = useState<WeightOptions>({
        type: 'metric',
        kg: 70
    })
    . . .
}
```

Then we declare the SetupProvider component, which contains the state of the fields—weight goal, gender, age, height, and weight—that the user will alter during the account creation process.

```
    const { startOAuthFlow: oauthFlowGoogle } = useOAuth({
        strategy: 'oauth_google'
    })
    const { startOAuthFlow: oauthFlowApple } = useOAuth({
        strategy: 'oauth_apple'
    })
    const { signUp, setActive, isLoaded } = useSignUp();
```

Next, we take the objects and functions returned by Clerk's useOAuth() and useSignUp() hooks. They provide functionality for user sign up using an email and password or an OAuth method (Google or Apple).

```
    const createAccountWithApple = async () => {
        const { createdSessionId, setActive } = await oauthFlowApple({});
        if (!setActive) throw new Error;

        await setActive({ session: createdSessionId });
    }
```

Then we create functions to handle the OAuth sign up, that wait for the user's input and set an active session. The same is done for the Google OAuth sign up.

```
const createAccountWithEmail = async (
    username: string, emailAddress: string, password: string
) => {
    await signUp.create({ username, emailAddress, password })
    await signUp.prepareEmailAddressVerification()
}

const verifyEmail = async (code: string) => {
    const { createdSessionId, createdUserId } = await
        signUp.attemptEmailAddressVerification({
            code: code,
        })
    if (!createdSessionId || !createdUserId) throw new Error;
    await setActive({ session: createdSessionId })
    return await finaliseSetup(createdUserId)
}
```

Then we create functions for sign up by email, with a username, email, and password, that creates a temporary user and sends an email verification code to the provided email address. The email is then validated if the code inputted by the user and passed to the email verification function is valid, a user is created, and the session is set.

```
const finaliseSetup = async (userId: string) => {
    if (gender === '' || weightGoal === '') throw new Error;

    return await createAccount({
        user_id: userId,
        weight_goal: weightGoal,
        gender: gender,
        age: age,
        . . .
    })
}
```

After every method of sign up, the finaliseSetup function is called to handle the creation of the account in the database with all the information that the user entered during the setup process.

5.2.2.2. Logged-in layout

This represents the layout of the routes that can be accessed by users that are signed in.

```
const Layout = () => {
    const { user, isSignedIn } = useUser();

    if (!user || !isSignedIn) {
        return <Redirect href={page.auth.authentication} />
    }

    return (
        <DateProvider>
            <UserProvider user={user}>
                <Stack
                    screenOptions={{ headerShown: false, animation: 'fade' }}
                />
            </UserProvider>
        </DateProvider>
    );
};
```

It contains another stack component which fades as the transition between screens.

It also contains the “DateProvider” context which handles the date of the day the user chooses to be shown in the food diary screen, that also effects the daily statistics screen and the screens in which the user adds food for that specific day.

The “UserProvider” context is also included to provide all of the screens that utilise the active user a non-null version of the user object obtained from Clerk, for a better TypeScript experince, by using the useAuthedUser() hook.

5.2.3. Services

All of the necessary functions for communicating between the frontend and the backend of the application, initiating API calls to the server, and managing requests to the USDA FoodData Central API are located in the folder titled "services." Axios is used throughout all of the files to manage the HTTP requests and responses.

5.2.3.1. Endpoints

The API endpoints for both the USDA API and our backend are provided in the constants/routes directory.

```
const address = '172.22.118.161'
const port = '6000'
const domain = `https://${address}:${port}/api` as const

export const api = {
    favorite_foods: `${domain}/favorite-foods`,
    progress: `${domain}/progress`,
    adjust_macros: `${domain}/settings/adjust-macros`,
    . . .
} as const;
```

An address and a port are defined for the backend address, which are interpolated into the domain. The endpoints are then concatenated to the domain to create the api constant. The use of ‘as const’ at the end of the domain and api constants are for better developer experience working on the services. This will ensure that the whole endpoint string will be shown on hover over the variable in the editor.

```
const usda_domain = 'https://api.nal.usda.gov/fdc/v1';

export const usda_api = {
    search: `${usda_domain}/foods/search`,
    food: `${usda_domain}/food`,
    foods: `${usda_domain}/foods`,
} as const;
```

The endpoint routes for the USDA API are created in the same manner, the API providing three different endpoints for the food-related operations.

5.2.3.2. Requests

Requests are handled by Axios, which provides a simple API for handling asynchronous HTTP requests and also supports the typing of the response data.

```
export const getMealsForDay = async (
    userId: string,
    date: Date
): Promise<Meal[]> => {
    const res = await axios.get<Meal[]>(api.meals, {
        params: {
            user_id: userId,
            day: date.toISOString(),
        },
    });

    return res.data;
}
```

The above example represents one of the requests from the meal.ts file, which is a straightforward HTTP GET request passed to the “/api/meals” endpoint on our backend. Its purpose is to fetch the meal data of a user for a given date, information which is passed to the query parameters ‘user_id’ and ‘day’, with the date being passed in the standard ISO format. The request will return AxiosResponse that should contain an array of objects of type Meal, but, in case the request fails, the error will be caught by the calling function and displayed appropriately to the user through the user interface.

```
export const updateMacroGoal = async (macroGoal: UpdateMacroGoalDto) => {
    await axios.put(api.adjust_macros, macroGoal)
}
```

Let’s have a look at a different example, this one coming from the settings.ts file, that performs a PUT request routed to the “/api/settings/adjust-macros” (api.adjust_macros) endpoint, that should change the macronutrient goal of a user with data stored in the UpdateMacroGoalDto object. We are only interested in the status of the response, therefore we do not anticipate receiving data, so we don’t need to return anything, and TypeScript will infer the return type to a “Promise<void>”. Axios will take care of the status on its own automatically, and the function will throw an exception if it receives an error code in the response.

5.2.4. Components

Components are the building blocks of React Native, which provide snippets of independent and reusable UI code that combine form and function.

First off, we have a component that uses the useQuery and useMutation hooks from React Query to provide easier state management and UI updates driven by the state of the requests.

```
const Water = () => {
    const [water, setWater] = useState(0)
    const { user } = useAuthedUser()
    const { dateYMD } = useDate()

    const { isLoading } = useQuery(['water'],
        () => getWater(user.id, dateYMD), {
        onSuccess: (data) => {
            setWater(data)
        },
        onError: () => {
            setWater(0)
        }
    })

    const { mutate: mutateWater } = useMutation(
        (amount: number) => putWater(user.id, dateYMD, amount))

    const setWaterAmount = (amount: number) => {
        setWater(amount)
        mutateWater(amount)
    }

    return (
        <View className='flex-col'>
            . . .
        </View>
    )
}
```

It gathers user and date data from the contexts higher up the component tree to pass down to the query function and as the query key.

The initial state of the water is set on 0 until a response is fetched using the useQuery hook. The UI waits for the request to be done loading to allow the user to interact with the the interface. On response completion, the water is set to the value retrieved from the database.

The useMutation hook creates a mutate function that can be used when the user triggers an event. When the water is set in the UI by the user, its state is changed proactively to provide responsive behaviour and after that the request is made.

A more design-oriented component comes in the form of “HorizontalProgressBar.tsx”.

```
export const HorizontalProgressBar: React.FC<{
    barClassName: string;
    value: number;
    goal: number;
}> = ({ barClassName, value, goal }) => {
    const progress = goal === 0 ? 0 : Math.min(value / goal, 1) * 100;

    const progressValue = useSharedValue(0);

    useEffect(() => {
        progressValue.value = withTiming(progress, {
            duration: 800,
            easing: Easing.bezier(0.5, 0, 0.5, 1)
        });
    }, [progress]);

    const progressStyle = useAnimatedStyle(() => {
        return {
            width: `${progressValue.value}%`
        };
    });

    return (
        <Animated.View
```

```
                style={progressStyle}
                className={barClassName}
            />
        )
}
```

It represents an animated progress bar that animates whenever the progress value changes, which corresponds to a change in either the value of the “goal” or the “value” props.

The animation is triggered by a change of the progress value by using useEffect and setting the animated value to the new progress value with a timing function so that the animation doesn’t jump directly to the final position. The withTiming function makes it possible to set a duration of 800ms and a Bezier easing function to the animation, making it look more natural.

The style is then processed by the useAnimatedStyle hook in order to set the percentage width of the bar using the animated value, which is then applied to the Animated.View component.

The component also takes in a barClassName prop to provide the bar with a static style, but also making it more reusable in different components, being used throughout the application for bar charts (Stats, Progress, ProgressComparison components) and as progress bars (Stats, Macros, SearchHome components).

5.2.5. TYPES

It is the client-side equivalent of the dto module on the server, supplying TypeScript types that correspond to DTOs.

```
export type MealType = 'breakfast' | 'lunch' | 'dinner' | 'snack';
export type PortionSize = 'serving' | 'gram' | 'ml';

export interface Meal {
    id: number;
    user_id: string;
    food: Food;
    meal_type: MealType
    date: string;
    portions: number;
    portion_size: PortionSize;
}
```

Represents a meal that will be obtained as a response to a request made to the server.

5.2.6. UTILS

Groups utility functions into a single directory, like the implementation of the token cache, and the transformation of a USDA food into a FoodDto, by searching for each nutrient by its USDA identifier.

```
const findSearchNutrient = (
    nutrients: AbridgedFoodNutrientDto[],
    id: number
) => {
    return nutrients.find((nutrient) => nutrient.nutrientId === id);
}

const getSearchNutrientValue = (
    nutrients: AbridgedFoodNutrientDto[],
    id: number
) => {
           const nutrient = findSearchNutrient(nutrients, id);
           return nutrient?.value || 0;
       }

       export const usdaSearchFoodToFood = (food: SearchResultFoodDto): Food
```

5.2.7. Assets

The directory in which fonts, frontend images for icons and background, and lottie animations stored as JSONs.

6. CONCLUSIONS

6.1. FINAL WORDS

In conclusion, Calory provides a user-friendly mobile app for managing dietary patterns, monitoring physical activity, and making informed health decisions. Calory enables users to effortlessly monitor food consumption, construct personalized meals, track water intake, count steps, and access statistical insights with its user-friendly interface and extensive features. By advocating a holistic approach to healthy living, Calory enables users to take charge of their nutrition and lifestyle decisions, thereby nurturing better health. Future work may involve the integration of peripheral devices and machine learning algorithms to improve functionality and personalize recommendations. Calorie is a valuable instrument for those who wish to live healthier lifestyles.

6.2. FUTURE WORK

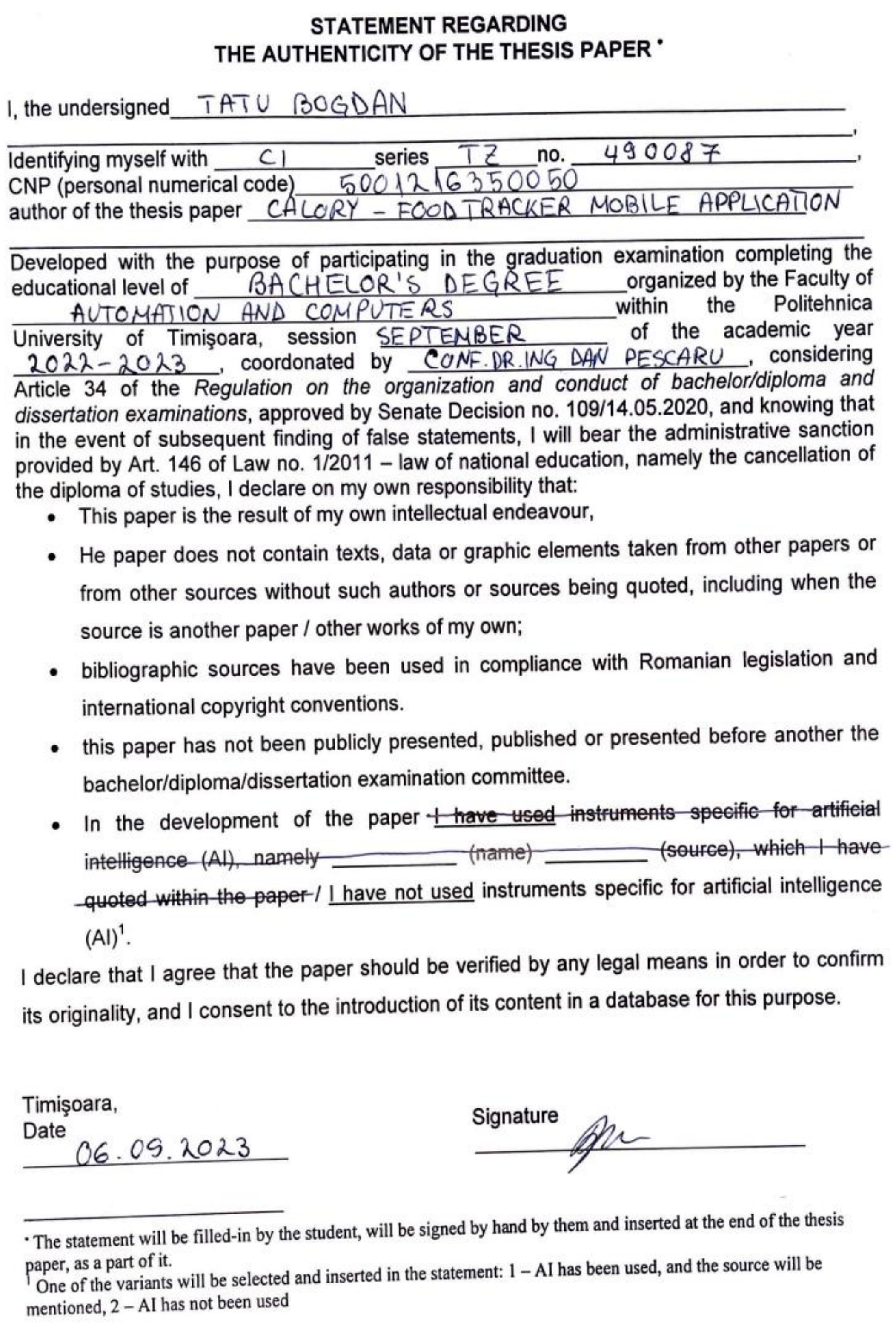
The future of Calory contains promise for additional enhancements and developments. Calory can investigate integrations with wearable devices and smart scales to collect and analyze additional health-related data as technology continues to advance. This expansion would provide users with a more in-depth and real-time comprehension of their overall health, allowing them to make more informed decisions regarding their daily routines.

In addition, Calory can consider incorporating machine learning algorithms in order to provide personalized recommendations and insights based on user-specific data. By utilizing artificial intelligence, the app can provide customized recommendations for optimal nutrition, exercise routines, and goal setting, thereby assisting users further in attaining their health goals. Other models have also been trained through image recognition to recognize meals from photos taken by users and calculate their nutritional values, which could be a prospect for the application.

Calory provides a seamless user experience and a variety of features that facilitate mindful dining, physical activity monitoring, and data-driven decision making. Calory has the potential to continuously support individuals on their journey toward enhanced health, well-being, and long-term sustainable behaviors through ongoing developments and future enhancements.

7. REFERENCES

[1] "React Native · Learn once, write anywhere." https://reactnative.dev/
[2] "ClErk | Authentication and User Management," *Clerk*. https://clerk.com/
[3] "Tailwind CSS - Rapidly build modern websites without ever leaving your HTML.," *Tailwind CSS*. https://tailwindcss.com/
[4] "Material symbols and icons - Google Fonts," *Google Fonts*. https://fonts.google.com/icons?icon.platform=web
[5] S. Benitez, "Rocket - Simple, Fast, Type-Safe web framework for Rust." https://rocket.rs/v0.4/
[6] "TanStack Query | react query, solid query, svelte query, vue query." https://tanstack.com/query/latest
[7] "Introduction | Expo Router." https://expo.github.io/router/docs/
[8] "Railway," *Railway*. https://railway.app/
[9] "Introduction - Rust By Example." https://doc.rust-lang.org/rust-by-example/index.html
[10] "Overview · Serde." https://serde.rs/
[11] "Illustrated Guide to SQLX." https://jmoiron.github.io/sqlx/
[12] "tide - Rust." https://docs.rs/tide/latest/tide/
[13] "surf - Rust." https://docs.rs/surf/latest/surf/
[14] "Build, Collaborate & Integrate APIs | SwaggerHub." https://app.swaggerhub.com/apis/fdcnal/food-data_central_api/1.0.1#/FDC
[15] "PostgreSQL 16RC1 Documentation," *PostgreSQL Documentation*, Aug. 31, 2023. https://www.postgresql.org/docs/16/index.html
[16] "Diesel is a Safe, Extensible ORM and Query Builder for Rust." https://diesel.rs/
[17] "reqwest - Rust." https://docs.rs/reqwest/latest/reqwest/
[18] "npm: react-native-health," *Npm*. https://www.npmjs.com/package/react-native-health
[19] "Expo documentation," *Expo Documentation*. https://docs.expo.dev/
[20] "Getting started | Axios Docs." https://axios-http.com/docs/intro



STATEMENT REGARDING THE AUTHENTICITY OF THE THESIS PAPER *

I, the undersigned TATU BOGDAN,

Identifying myself with CI series TZ no. 490087,
CNP (personal numerical code) 5001216350050
author of the thesis paper CALORY – FOOD TRACKER MOBILE APPLICATION

Developed with the purpose of participating in the graduation examination completing the educational level of BACHELOR'S DEGREE organized by the Faculty of AUTOMATION AND COMPUTERS within the Politehnica University of Timişoara, session SEPTEMBER of the academic year 2022-2023, coordonated by CONF. DR. ING DAN PESCARU, considering Article 34 of the *Regulation on the organization and conduct of bachelor/diploma and dissertation examinations*, approved by Senate Decision no. 109/14.05.2020, and knowing that in the event of subsequent finding of false statements, I will bear the administrative sanction provided by Art. 146 of Law no. 1/2011 – law of national education, namely the cancellation of the diploma of studies, I declare on my own responsibility that:

- This paper is the result of my own intellectual endeavour,
- He paper does not contain texts, data or graphic elements taken from other papers or from other sources without such authors or sources being quoted, including when the source is another paper / other works of my own;
- bibliographic sources have been used in compliance with Romanian legislation and international copyright conventions.
- this paper has not been publicly presented, published or presented before another the bachelor/diploma/dissertation examination committee.
- In the development of the paper ~~I have used instruments specific for artificial intelligence (AI), namely ________ (name) ________ (source), which I have quoted within the paper~~ / I have not used instruments specific for artificial intelligence (AI)[1].

I declare that I agree that the paper should be verified by any legal means in order to confirm its originality, and I consent to the introduction of its content in a database for this purpose.

Timişoara,
Date 06.09.2023

Signature [illegible]

* The statement will be filled-in by the student, will be signed by hand by them and inserted at the end of the thesis paper, as a part of it.

[1] One of the variants will be selected and inserted in the statement: 1 – AI has been used, and the source will be mentioned, 2 – AI has not been used