Development of a Full Stack Mobile Application for Dynamic Quiz Generation with AI-Driven Hints using ChatGPT

**Bogdan TATU**

Computers and Information Technology   
**Faculty of Automation and Computers**

**Polytechnic University of Timisoara**

Timișoara, 06, 2023

Table of Contents

[1 The Quizzdos Project 11](#_Toc138577236)

[1.1 Introduction 11](#_Toc138577237)

[2 Specifications, requirements and design 11](#_Toc138577238)

[2.1 Similar applications 11](#_Toc138577239)

[2.1.1 MyFitnessPal 11](#_Toc138577240)

[2.1.2 Rust 11](#_Toc138577241)

[2.2 Project Requirements 11](#_Toc138577242)

[2.3 Project Specifications 12](#_Toc138577243)

[2.4 Block diagram 12](#_Toc138577244)

[2.5 Class diagrams 13](#_Toc138577245)

[2.5.1 Controllers 13](#_Toc138577246)

[2.5.2 Repositories 15](#_Toc138577247)

[3 Technical background 15](#_Toc138577248)

[3.1 Programming languages 16](#_Toc138577249)

[3.1.1 TypeScript (JavaScript) 16](#_Toc138577250)

[3.1.2 Rust 16](#_Toc138577251)

[3.2 Frontend technologies 17](#_Toc138577252)

[3.2.1 React Native + Expo 17](#_Toc138577253)

[3.2.2 Nativewind (Tailwind CSS) 17](#_Toc138577254)

[3.2.3 Axios 18](#_Toc138577255)

[3.2.4 React Query 18](#_Toc138577256)

[3.2.5 React Native Reanimated 18](#_Toc138577257)

[3.2.6 Expo Libraries 18](#_Toc138577258)

[3.3 Backend technologies 18](#_Toc138577259)

[3.3.1 Diesel 18](#_Toc138577260)

[3.3.2 Rocket & Serde 18](#_Toc138577261)

[3.3.3 Reqwest 18](#_Toc138577262)

[3.3.4 Tokio 18](#_Toc138577263)

[4 Application utilization 19](#_Toc138577264)

[4.1 Account creation 19](#_Toc138577265)

[4.2 Student experience 19](#_Toc138577266)

[5 Gantt Chart 19](#_Toc138577267)

[6 Implementation 20](#_Toc138577268)

[6.1 Backend 20](#_Toc138577269)

[6.1.1 Schema 20](#_Toc138577270)

[6.1.2 Models 20](#_Toc138577271)

[6.1.3 Database Diagram 24](#_Toc138577272)

[6.2 Frontend 26](#_Toc138577273)

[6.2.1 App.tsx 27](#_Toc138577274)

[6.2.2 Layouts 28](#_Toc138577275)

[6.2.3 Services 32](#_Toc138577276)

[6.2.4 Constants 34](#_Toc138577277)

[6.2.5 Screens 34](#_Toc138577278)

[7 Conclusions 35](#_Toc138577279)

[8 References 36](#_Toc138577280)

Illustrations

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.

1. Specifications, requirements and design
   1. Similar applications
      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.

* 1. 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.
  1. Project Specifications

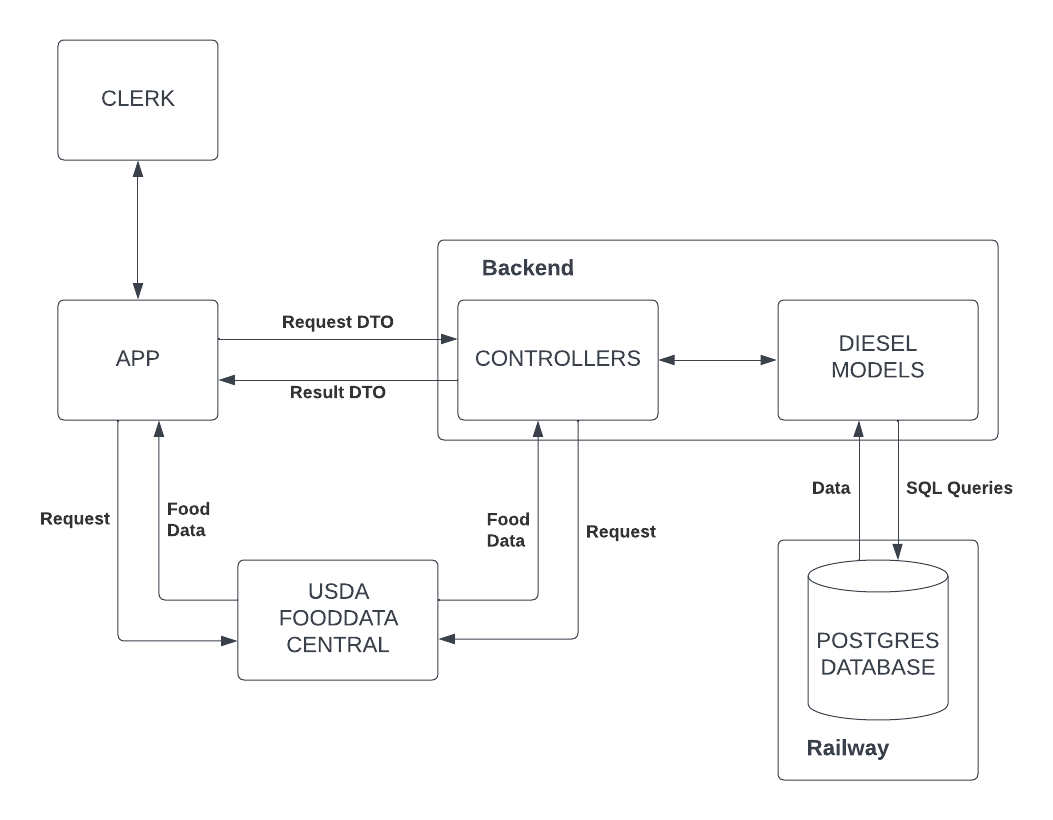
The frontend of the mobile application will be built using the Expo framework on top of React Native, with TypeScript, for multiple platform support, but prioritising iOS.

The backend will be written in Rust, utilizing Rocket, to build a type-safe, fast, and reliable API to handle requests from the application. It will also use a database-first approach with Diesel as an ORM (Object Relational Mapper) to communicate and query the database.

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.

* 1. 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.

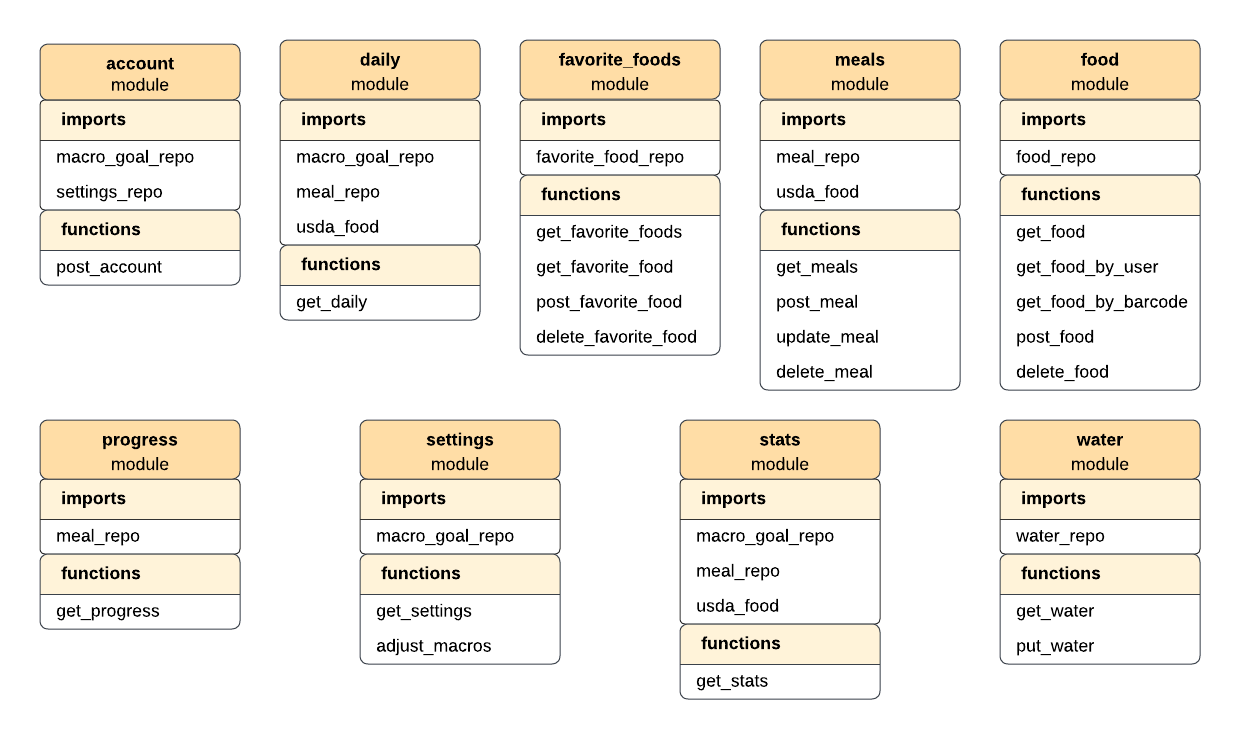
Controllers – The controllers are a group of functions that receive and handle the HTTP requests dispatched to them by Rocket, that ties each controller to a given route. They handle GET, POST, PUT and DELETE requests, perform business logic, and send back appropriate responses to the client-side. Some 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.

* 1. Class diagrams
     1. Controllers

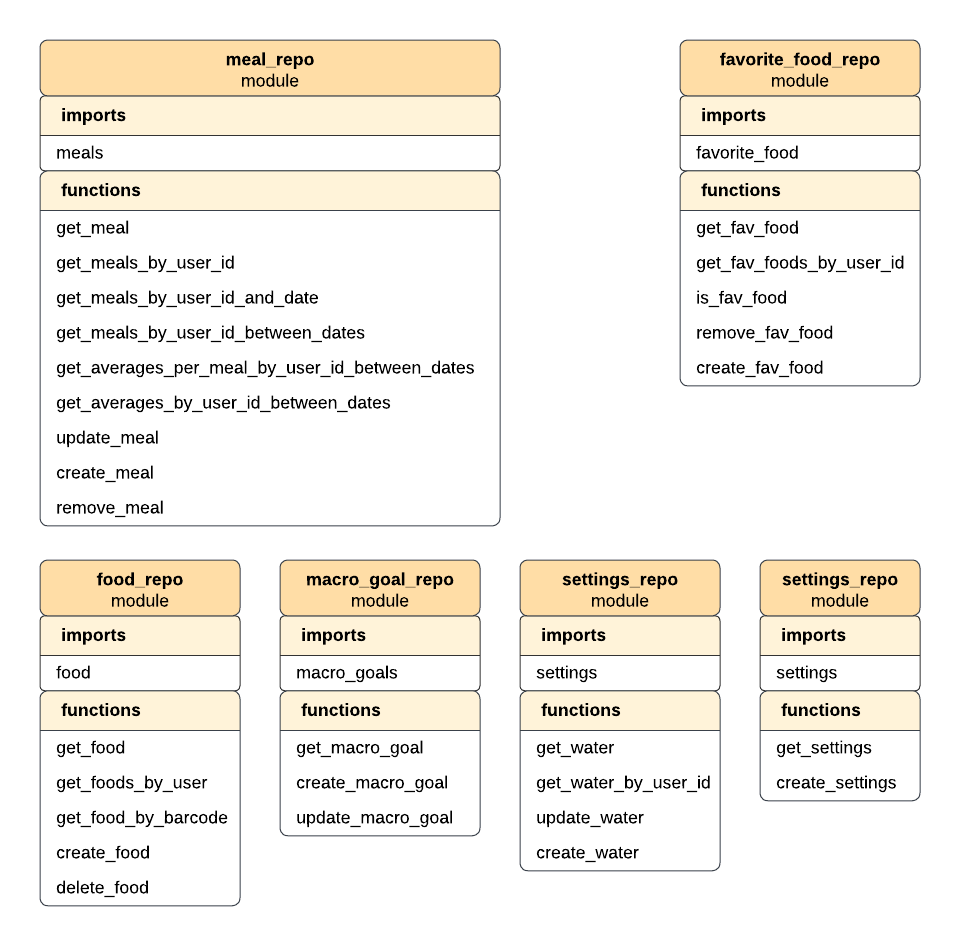


Rust, being a multi-paradigm language, doesn’t force the developer into grouping controller functionality into a class, so, naturally, each function is bundled up into a module, using repositories and/or the ‘usda\_food’ service as imports.

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

The controllers are:

1. account – handles account creation actions, such as creating the settings of a user and calculating their macronutrient goals.
2. daily – used to congregate the information of a user’s daily meals and goal.
3. favorite\_foods – handles operations related to users’ favourite foods.
4. meals – handles the management of users’ meals, creating, reading, updating, and deleting.
5. food – handles the retrieval of the food by different criteria and the modification of food.
6. progress – handles the calculation and statistical operations necessary to provide information on a user’s progress over time.
7. settings – represents the operations tied to the reding and updating of a user’s settings.
8. stats – gives insight on a user’s daily statistics.
9. water – manages the users’ daily water intake.
   * 1. 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.

1. Technical background
   1. Programming languages
      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.

* + 1. 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), ORMs and database drivers (e.g., Diesel, SeaORM, sqlx), HTTP clients (e.g., reqwest, hyper), error handling (e.g., anyhow) and many more use cases.

* 1. Frontend technologies
     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.

* + 1. 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.

* + 1. 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.

* + 1. 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.

* + 1. 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.

* + 1. 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.

* 1. Backend technologies
     1. Diesel

The most prevalent ORM and query builder library for Rust is Diesel. It provides type safety, eliminating the possibility of improper database interactions at compilation time.

It offers a high-level query builder with zero-cost abstractions and is designed to be abstracted over, allowing developers to write code that is reusable, making it the clear victor among the alternatives.

Diesel also comes with a command line interface tool called ‘diesel\_cli’ that handles database migrations, generating, running, and redoing them. It also generates the code required to map PostgreSQL types to its own typing system and then to Rust’s.

Being still in production, it is not a catch-all solution and replacement for SQL for querying databases from within Rust, and it allows developers to write raw SQL queries, but this undermines the type-safety guarantees of the crate.

* + 1. Rocket & Serde

Rocket 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 erro handler, processing done by the application code, and returning a HTTP response to the client.

* + 1. Reqwest

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

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

1. Application utilization

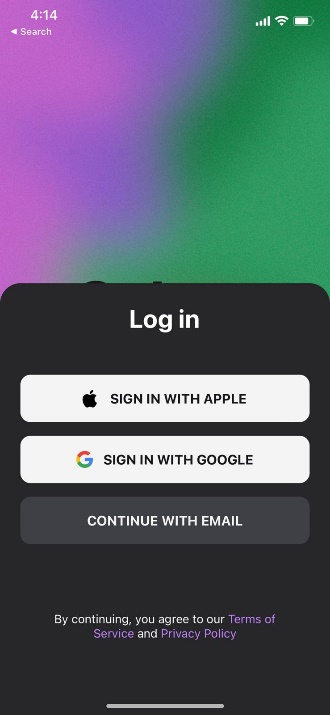
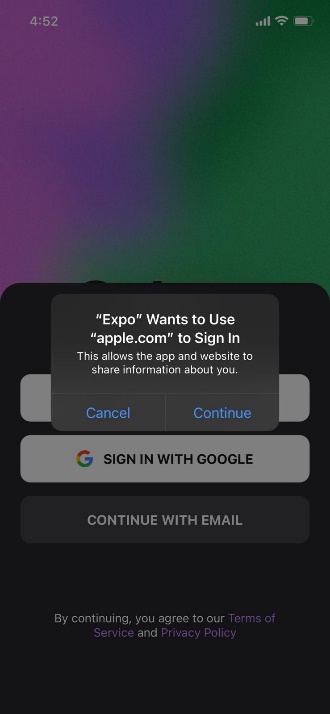
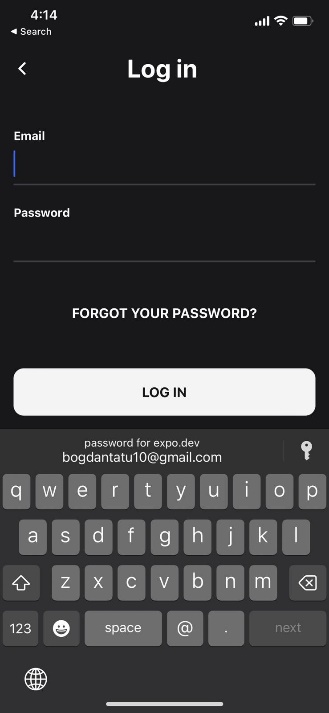
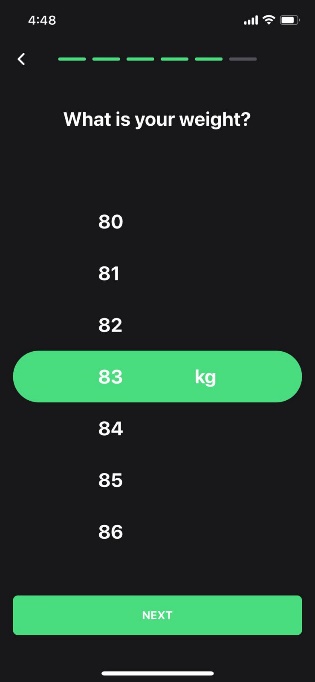
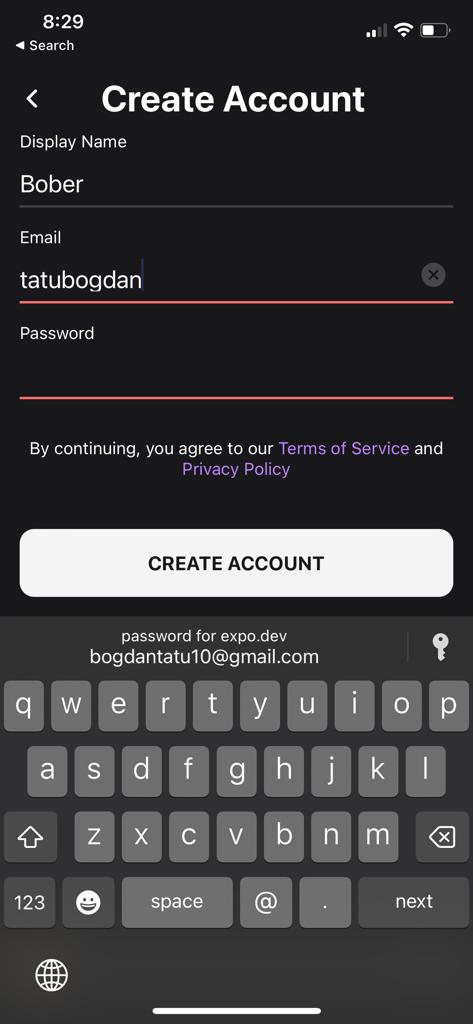
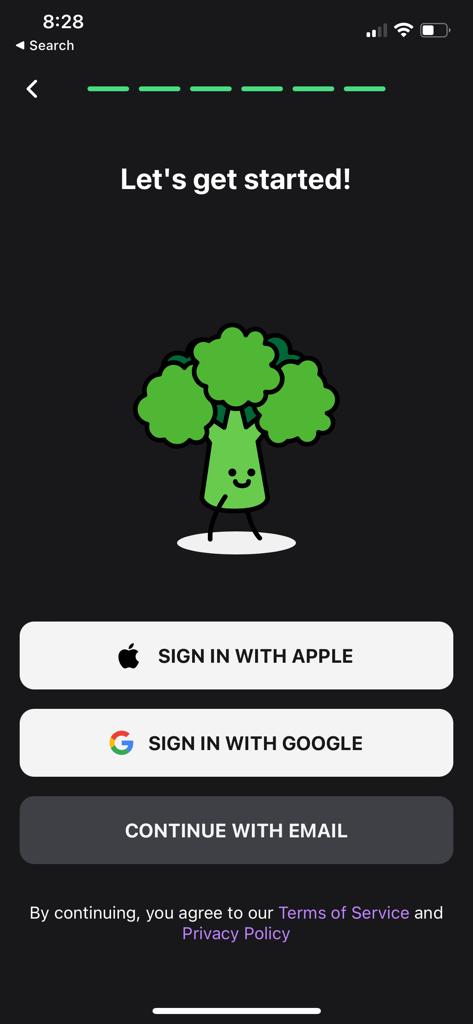
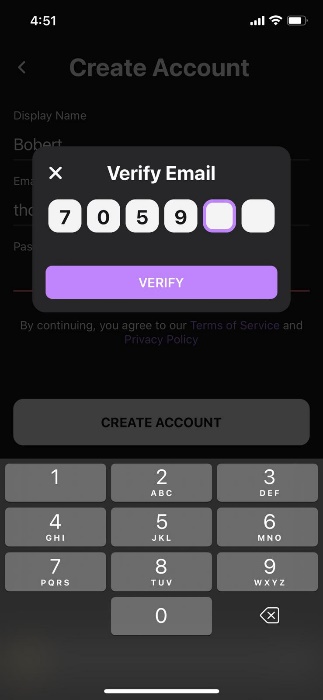
Upon launching the application, the user is presented with a title screen from which he can either go through the account creation process or log in.

Figure 4.1 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.

A screenshot of a cell phone

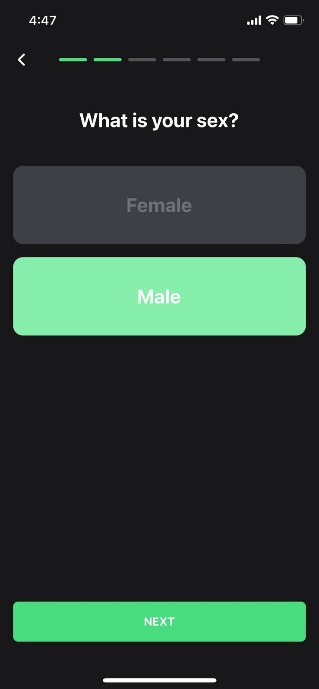
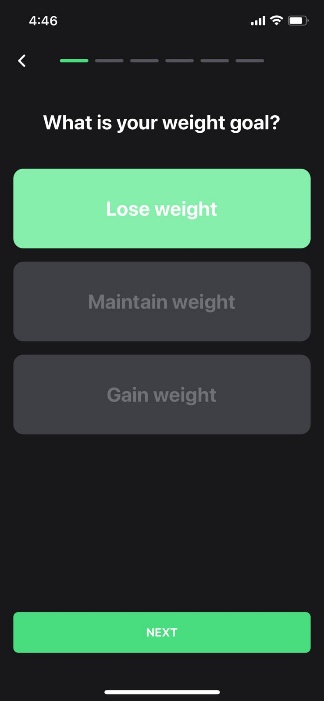
Description automatically generated with medium confidenceGoing the other route of signing up opens up a stack of screens that facilitate the account setup, the user having to select from multiple options for personal details, such as: weight goal, gender, age, height, and weight.

Figure 4.2 Weight Goal / Gender / Age / Height screens

Figure 4.3 Weight / Sign Up / Email Sign Up / Verification screens

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.

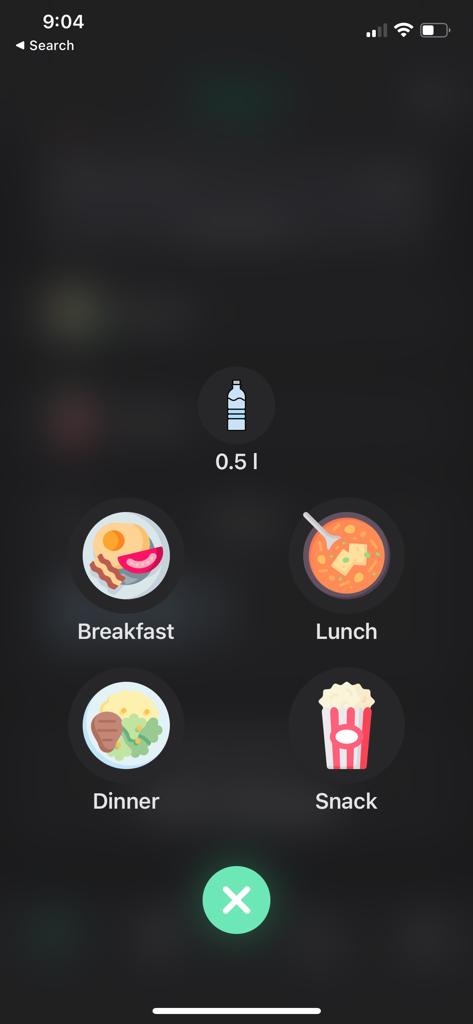
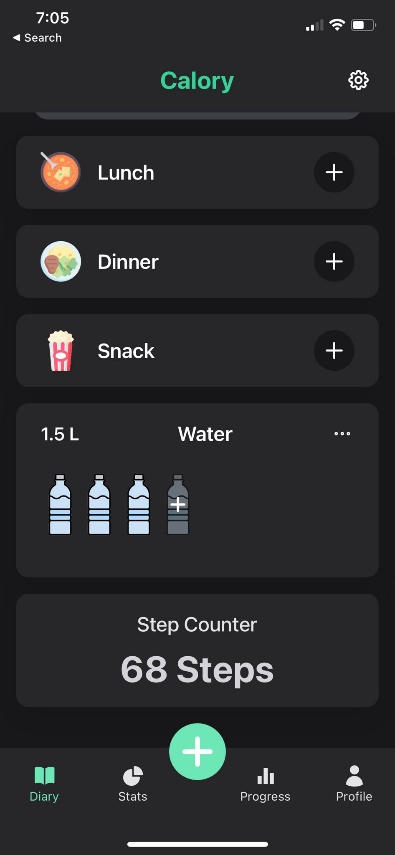
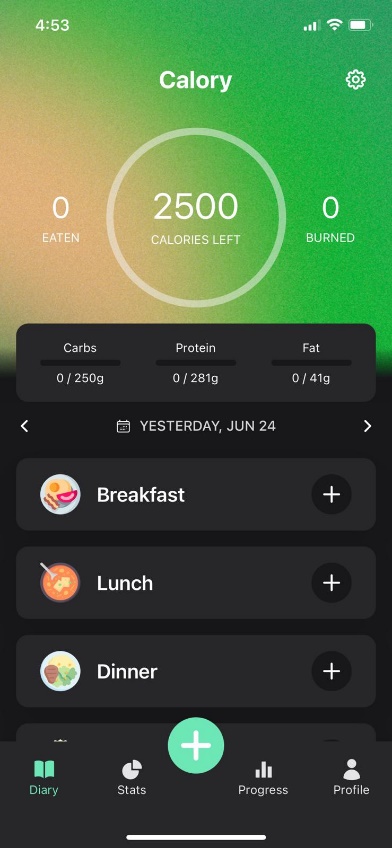
When a user creates a legitimate account, a validation modal appears, and they receive an email with a verification number. When the user enters the incorrect code, red borders appear to show that something went wrong; when they enter the correct code, an account is created, and they are taken to their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

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 their page where they can see nutritional values, food rating and change the portion, and track the food.

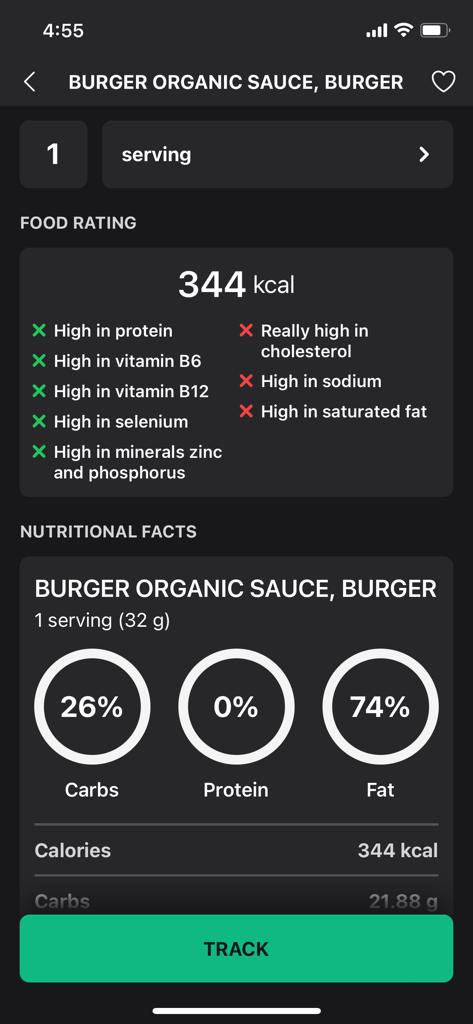
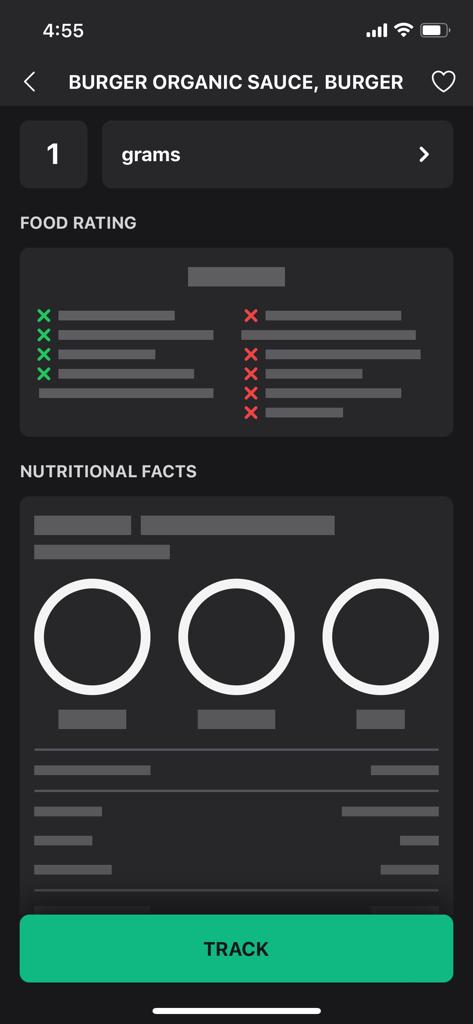
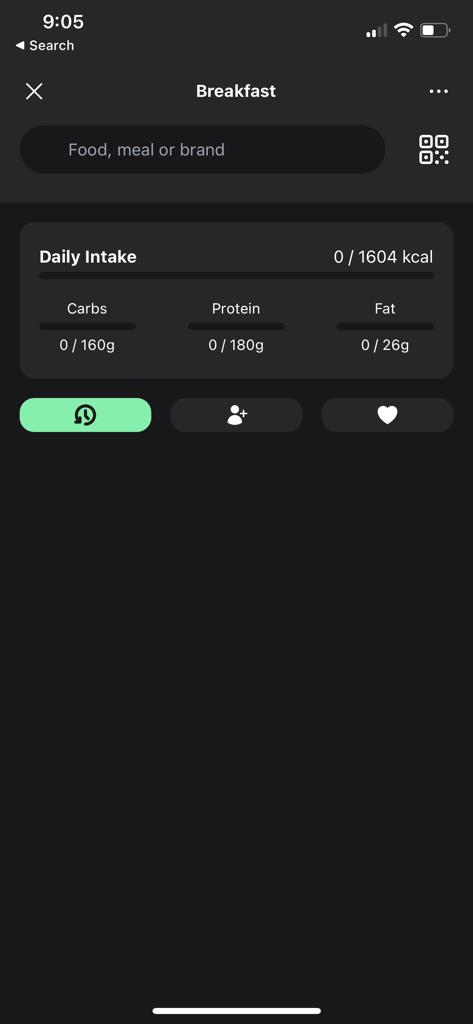
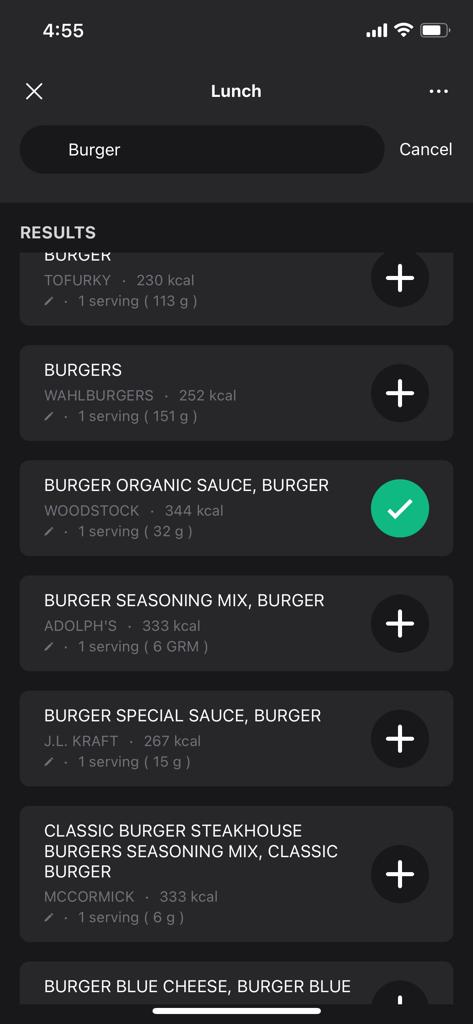
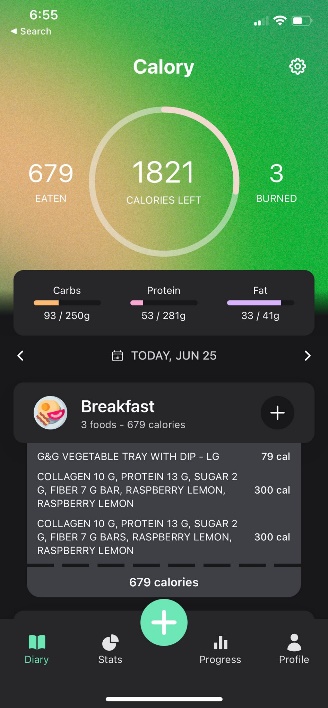
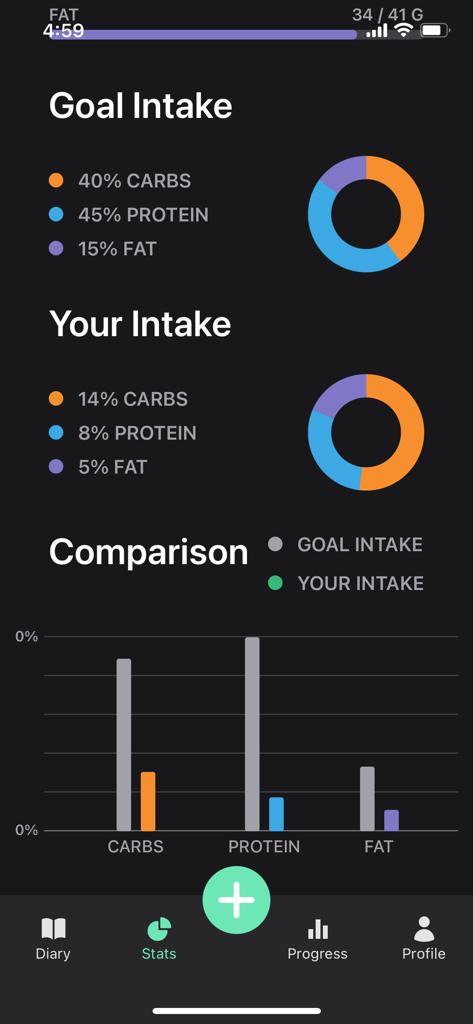


Figure 4.6 Placeholder Id / Id / Id Stats screens

Figure 4.5 Search / Search Results / Scan screens



Going back to the diary we can see the changes made by adding the foods.

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.

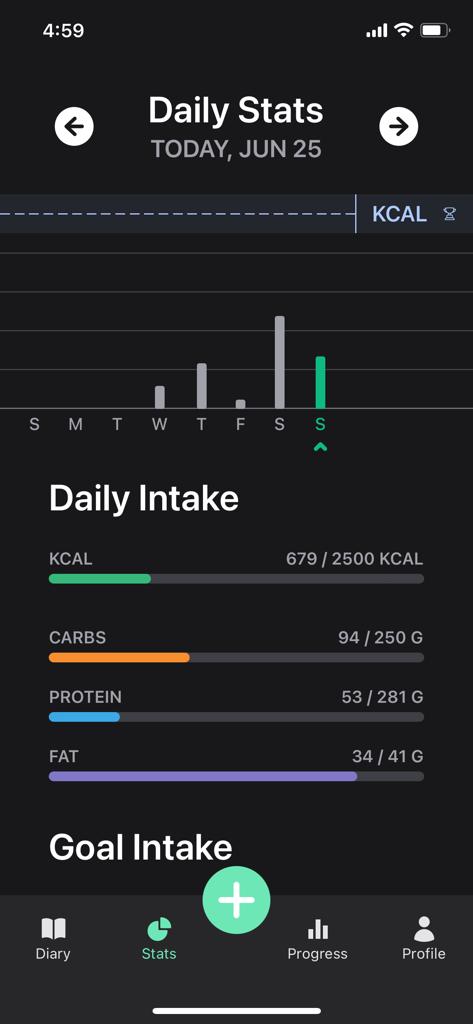


Figure 4.7 Diary Completed / Stats / Stats Comparison screens

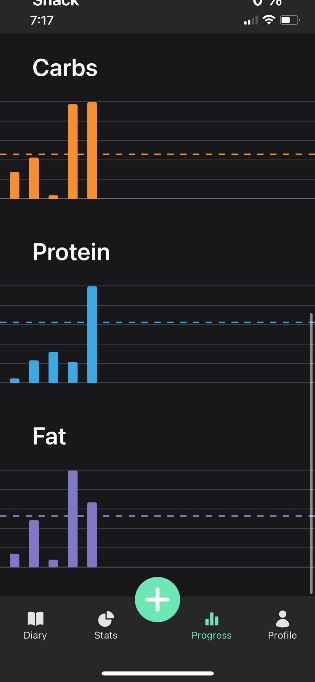
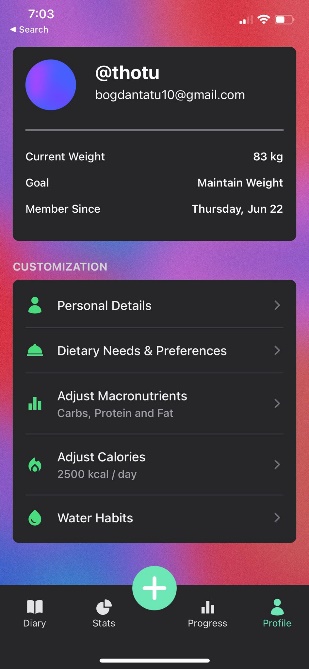
The progress pane displays the user's progress over a longer period of time as well as a breakdown of the percentage of calories consumed per meal type.

Figure 4.8 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.

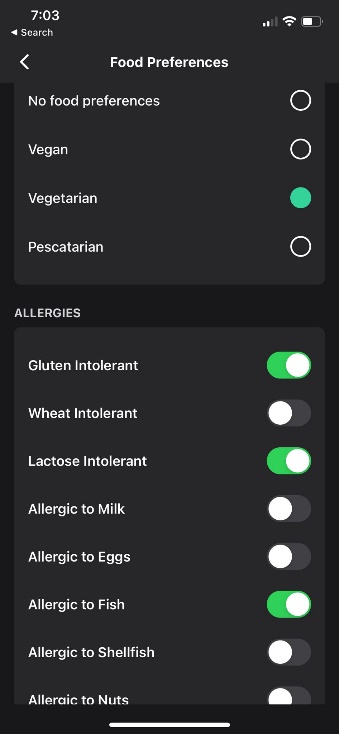
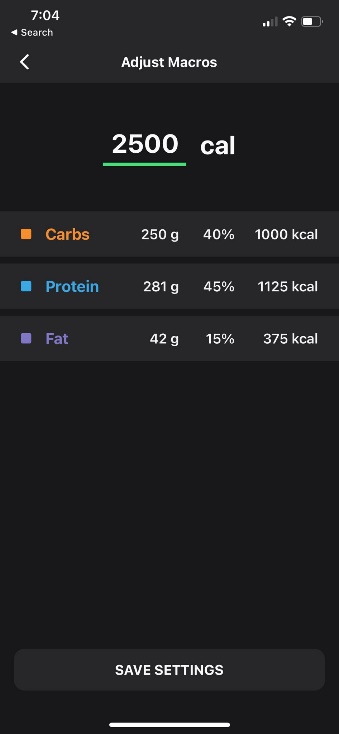
The screens for adjusting macronutrients and calories only permit saving if the values have been altered, and in the case of macronutrients, the percentages must add up to 100 percent.

Figure 4.8 Adjust Macros / Adjust Calories / Preferences screens

1. Implementation
   1. Backend
      1. Schema

The database schema is automatically generated by using the diesel cli “diesel print-schema” command and directly after running or redoing migrations.

diesel::table! {

*use* diesel::sql\_types::\*;

*use* *super*::sql\_types::ServingSizeUnit;

    food (id) {

        id -> Int4,

        #[max\_length = 32]

        user\_id -> Bpchar,

        #[max\_length = 100]

        name -> Varchar,

        #[max\_length = 100]

        brand -> Nullable<Varchar>,

        barcode -> Nullable<Int8>,

        calories -> Float4,

        carbs -> Float4,

        protein -> Float4,

        fat -> Float4,

        serving\_size -> Float4,

        serving\_size\_unit -> ServingSizeUnit,

        ingredients -> Nullable<Text>,

    }

}

This example depicts one of the tables (the food table) generated in the schema.rs file by the command, which is transformed into multiple structs with implemented traits by the procedural macro "table!" during the build. Diesel has its own data types that correspond to PostgreSQL types and are represented in the models by Rust types.

* + 1. Models

The models represent the mapping of Rust types to Diesel’s generated tables. They are used for data capture and related database operations. 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(Queryable)]

The Queryable 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(Insertable)]

The Insertable trait is used to represent a structure that can be used to insert a new row into the database.

#[derive(AsChangeset)]

The AsChangeset trait denotes that the structure can be used to update a row from the database.

* + - 1. Meal model



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.

* + - 1. Settings model



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.

* + - 1. Macro Goal model



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%).

* + - 1. Favourite Food model



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.

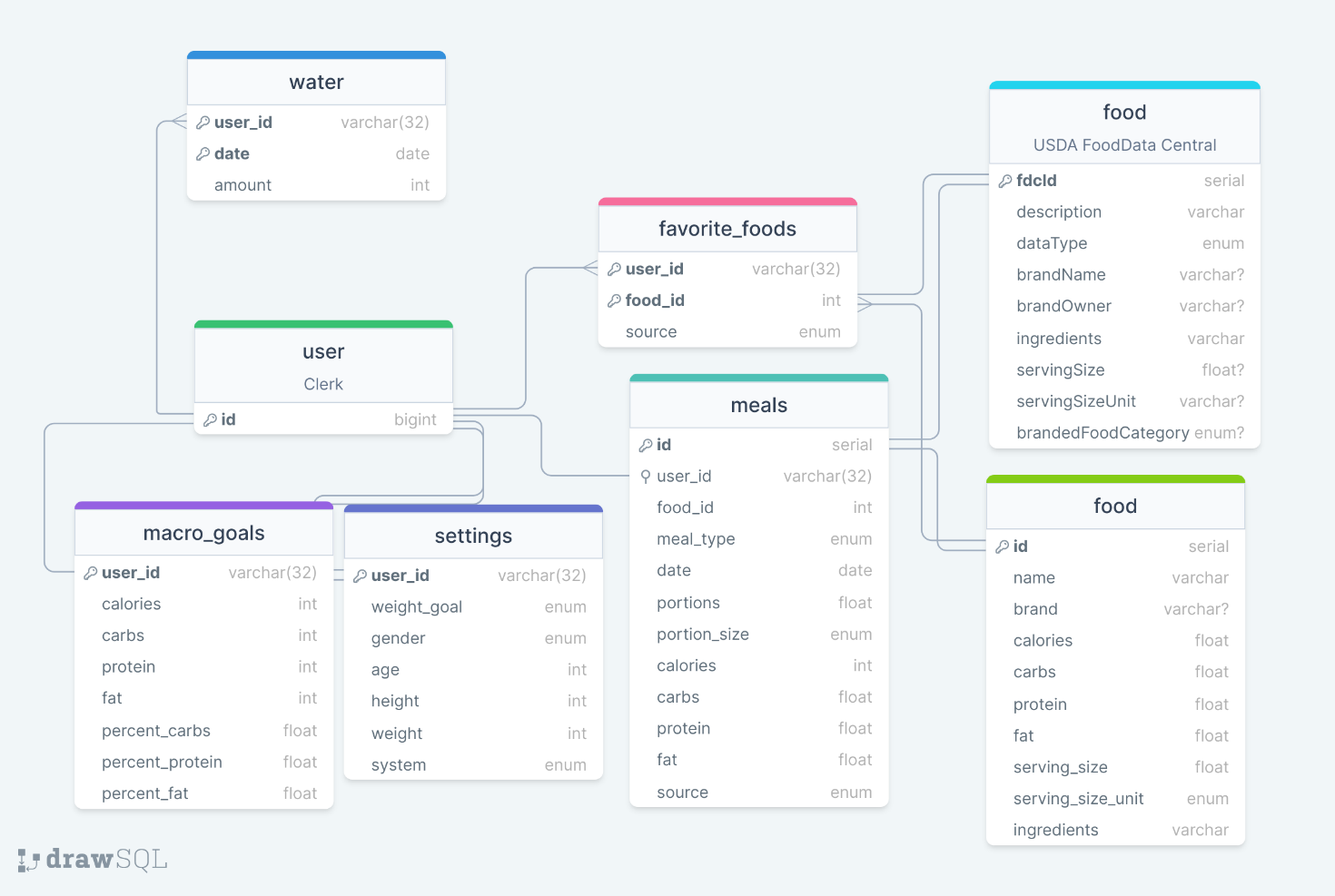
The source is the same as in the meals example.

* + - 1. Water model



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’.

* + 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.

* + 1. 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.



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.



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.

* + 1. 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: &*mut* PgConnection, . . . )

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

Controllers establish a connection to the database:

*let* connection = &*mut* db::establish\_connection();

The connection is transferred to repositories to execute SQL queries.

*pub* *fn* establish\_connection() -> PgConnection {

    dotenv().ok();

*let* database\_url = env::var("DATABASE\_URL")

.expect("DATABASE\_URL must be set");

    PgConnection::establish(&database\_url)

.expect(&format!("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 PgConnection object is created by establishing a connection with the url.

* + - 1. Food repo

The food repository module defines the functionality for handling user created food. It includes the following functions:

*pub* *fn* get\_food(conn: …, id: i32) -> Food {

*return* food::table

        .filter(food::id.eq(id))

        .first::<Food>(conn)

        .expect("Error loading food");

}

The “get\_food” function retrieves the food from the database given by its primary key, “id”, and is transformed into a straightforward “SELECT \* WHERE food.id = id” query. The function panics if the requested food with id cannot be found in the database.

*pub* *fn* get\_foods\_by\_user(conn: …, uid: &String) -> Vec<Food> {

*return* food::table

        .filter(food::user\_id.eq(uid))

        .load::<Food>(conn)

        .expect("Error loading foods");

}

The function performs a query to get all the foods created by the user and loads them into a vector.

*pub* *fn* get\_food\_by\_barcode(conn: …, barcode: i64) -> Option<Food> {

*return* food::table

        .filter(food::barcode.eq(barcode))

        .first::<Food>(conn)

        .ok();

}

The “get\_food\_by\_barcode” function is similar in function to the other ones, but instead of panicking when a food with that barcode id cannot be found, it returns an optional food, which could have the food data, represented as Some(food) or None otherwise.

In this case, the optional is returned because users may request data on a product, through the barcode, that is not in the database.

*pub* *fn* create\_food(conn: …, new\_food: &NewFood) -> bool {

*return* diesel::insert\_into(food::table)

        .values(new\_food)

        .execute(conn)

        .is\_ok();

}

This function inserts "new\_food" into the database utilizing diesel's "insert\_into" method and returns a boolean value based on the result returned by the execute method.

*pub* *fn* delete\_food(conn: …, id: i32) -> bool {

*return* diesel::delete(food::table.filter(food::id.eq(id)))

        .execute(conn)

        .is\_ok();

}

The “delete\_food” removes the row from the database that contains the specified id and returns true if the operation was successful.

* + - 1. 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* *fn* get\_meal(conn: …, mid: i32) -> Meal

Gets the meal by its primary key from the database.

*pub* *fn* get\_meals\_by\_user\_id(conn: …, uid: &String) -> Vec<Meal>

This function gets all the meals of a given user.

*pub* *fn* get\_meals\_by\_user\_id\_and\_date(conn: …, uid: &String, date:

chrono::NaiveDate) -> Vec<Meal>

The “get\_meals\_by\_user\_id\_and\_date” function performs multiple filters on the database to retrieve the meals of a user for a specified date.

*pub* *fn* get\_total\_macro\_intake\_per\_day\_between\_dates\_for\_user(

    uid: &String,

    start\_date: &chrono::NaiveDate,

    end\_date: &chrono::NaiveDate,

) -> Vec<MealGroup> {

*return* meals::table

        .filter(meals::user\_id.eq(uid))

        .filter(meals::date.ge(start\_date))

        .filter(meals::date.le(end\_date))

        .group\_by(meals::date)

        .select((

            meals::date,

            sql::<sql\_types::Integer>("CAST(SUM(calories) AS INTEGER)"),

            sql::<sql\_types::Float>("SUM(protein)"),

            sql::<sql\_types::Float>("SUM(carbs)"),

            sql::<sql\_types::Float>("SUM(fat)"),

        ))

        .order\_by(meals::date.asc())

        .load::<MealGroup>(conn)

        .expect("Error loading meals by user id and date");

}

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 day to calculate the sum of each of the macronutrients of each day and it orders the results in an ascending manner by their date.

Since Diesel does not support more complex queries without workarounds and a large amount of boilerplate code, the queries for SUMs are executed in ordinary SQL.

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

*pub* *struct* MealGroup {

*pub* date: chrono::NaiveDate,

*pub* calories: i32,

*pub* protein: f32,

*pub* carbs: f32,

*pub* fat: f32,

}

The “From” trait needs to be implemented for the struct so that it can be automatically converted from the tuple output by the select method into the desired struct.

*impl* From<(chrono::NaiveDate, i32, f32, f32, f32)> *for* MealGroup {

*fn* from(

        (date, calories, protein, carbs, fat):

(chrono::NaiveDate, i32, f32, f32, f32),

    ) -> *Self* {

        MealGroup { date, calories, protein, carbs, fat }

    }

}

*pub* *fn* get\_average\_macro\_intake\_per\_meal\_type\_between\_dates\_for\_user(

    uid: &String,

    start\_date: &chrono::NaiveDate,

    end\_date: &chrono::NaiveDate,

) -> Vec<MealAveragePerMealType>

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* *fn* remove\_meal(conn: …, mid: i32) -> bool

*pub* *fn* create\_meal(conn: …, meal: NewMeal) -> bool

The last two queries, “remove\_meal” and “create\_meal” are used for the deletion and the creation of a meal.

* + - 1. Favorite food repo

*pub* *fn* get\_fav\_foods\_by\_user\_id(conn: …, uid: &String) -> Vec<i32> {

*return* favorite\_foods::table

        .filter(favorite\_foods::user\_id.eq(uid))

        .select(favorite\_foods::food\_id)

        .load::<i32>(conn)

        .expect("Error loading favorite foods by user id");

}

The "get\_fav\_foods\_by\_user\_id" function retrieves the ids of a user's preferred foods by selecting the "food\_id" column from the resultant rows. This was done because favorite foods can have different sources, such as the internal database or the external USDA database, from which the controller will request the food data for the specified ids.

*pub* *fn* is\_fav\_food(uid: &String, fid: i32, source: Source)

The "is\_fav\_food" function receives a user id, a food id, and a source and examines the database for a row containing all three to determine if the food is the user's favorite.

*pub* *fn* remove\_fav\_food(uid: &String, fid: i32, source: Source) -> bool

*pub* *fn* create\_fav\_food(favorite\_food: NewFavoriteFood) -> bool

Lastly, the “remove\_fav\_food” and “create\_fav\_food” remove and add a favourite to and from the database.

* + - 1. Water repo

*pub* *fn* get\_water(uid: &String, day: chrono::NaiveDate) -> Option<Water>

“get\_water” retrieves the water intake of a user for a given day, which may or may not be set.

*pub* *fn* get\_water\_by\_user\_id\_between\_dates(

    uid: &String,

    start: chrono::NaiveDate,

    end: chrono::NaiveDate,

) -> Vec<Water>

The “get\_water\_by\_user\_id\_between\_dates” function gets all the water data of the user between the dates.

*fn* get\_average\_water\_intake\_between\_dates\_for\_user(

    uid: &String,

    start\_date: &chrono::NaiveDate,

    end\_date: &chrono::NaiveDate,

) -> f32

This function gets the average water intake of a user in the range of the dates.

*pub* *fn* update\_water(conn: …, updated\_water: NewWater) -> bool {

*return* diesel::insert\_into(water::table)

        .values(&updated\_water)

        .on\_conflict((water::user\_id, water::date))

        .do\_update()

        .set(water::amount.eq(updated\_water.amount))

        .execute(conn)

        .is\_ok();

}

The “update\_water” tries to perform the insertion of new water intake data into the database.

If it is the first insertion for the user on that specific date, it simply adds the row into the database. If the user has already logged their intake, it will result in a conflict on the water table’s primary key (user\_id, date composite key) and will instead set the amount of the existing instance to the new amount.

* + 1. Controllers

Controllers are the modules that handle the HTTP requests sent from the client-side. The functions are denoted by the attributes above them:

#[get("/route")]

#[post("/route", format = "json", data = "<data>")]

#[put("/route", format = "json", data = "<data>")]

#[delete("/route")]

These attributes specify the HTTP request method, along with the static and dynamic routes that may include query parameters, which provide the API's endpoints, and the request body and its format.

* + - 1. Water Controller

#[get("/water?<user\_id>&<date>")]

*pub* *fn* get\_water(user\_id: String, date: String) -> Json<i32> {

*let* connection = &*mut* db::establish\_connection();

*let* date = NaiveDate::parse\_from\_str(&date, "%Y-%m-%d").unwrap();

*match* water\_repo::get\_water(connection, &user\_id, date) {

        Some(water) => Json(water.amount),

        None => Json(0),

    }

}

An example of a concise controller function can be this one, handling the request for the water data for a given user on a specific date, returning the amount if the water is found in the database or 0, if it has not been created yet.

#[get("/progress?<user\_id>&<date\_from>&<date\_to>")]

*pub* *async* *fn* get\_progress(

    user\_id: String,

    date\_from: String,

    date\_to: String,

) -> Json<ProgressDto> {

*let* connection = &*mut* db::establish\_connection();

*let* date\_from = chrono::NaiveDate::from\_iso(&date\_from);

*let* date\_to = chrono::NaiveDate::from\_iso(&date\_to);

*let* meals = meal\_repo::

get\_total\_macro\_intake\_per\_day\_between\_dates\_for\_user(

        connection, &user\_id, &date\_from, &date\_to,

    );

*if* meals.is\_empty() {

*return* Json(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,

        );

*let* data = ProgressDto::new(meals, averages\_per\_meal\_type);

    println!("{:#?}", data);

*return* Json(data);

}

A more complex request can be denoted by the HTTP get request for the progress of a user between “date\_from” and “date\_to”. Establishing the connection to the database, transforming the dates from strings to an actual date type, getting the macro intake per day between the dates for the user. If that returns nothins it means that the user has no meals and can return an empty DTO back, then getting the average macro intake per meal type between the given dates to put it all together into a ProgressDto to send back to the user.

* + 1. DTOs

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.

*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* *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 of the DTOs also implement Into and From traits to make it easier to convert between other objects and DTOs.

*impl* From<USDABranndedFoodItemDto> *for* FoodDto

*impl* From<Food> *for* FoodDto

* + 1. Main

The main file is where the server gets launched from. Rocket handles the creation and runtime of the server by building it, attaching the CORS policies to the server, and mounting all the defined routes from the controller modules.

#[launch]

*fn* rocket() -> \_ {

    rocket::build().attach(cors::CORS).mount(

        "/api",

        routes

            favorite\_foods::get\_favorite\_foods,

            favorite\_foods::get\_favorite\_food,

            . . .

            food::get\_food\_by\_barcode,

            food::post\_food,

            food::delete\_food,

        ],

    )

}

* 1. Frontend

![A screenshot of a computer

Description automatically generatedThe 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.

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.

* + 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:



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:



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



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.

* + 1. Layouts
       1. Account-setup layout

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



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.



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



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.



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).



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.



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.



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.

* + - 1. Logged-in layout

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



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.

* + 1. 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.

* + - 1. Endpoints

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



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.



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

* + - 1. 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.



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.



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.

* + 1. 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).

* + 1. 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.

* + 1. 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

* + 1. Assets

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

1. Conclusions
   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.

* 1. 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.

1. References
2. React Native · Learn once, write anywhere - <https://reactnative.dev/>
3. Clerk | Authentication and User Management - <https://clerk.com/>
4. Tailwind CSS - Rapidly build modern websites without ever leaving your HTML. - <https://tailwindcss.com/>
5. Material Symbols and Icons - Google Fonts - <https://fonts.google.com/icons?icon.platform=web>
6. Rocket - Simple, Fast, Type-Safe Web Framework for Rust - <https://rocket.rs/v0.4/>
7. DrawSQL - Database schema diagrams - <https://drawsql.app/>
8. TanStack Query | React Query, Solid Query, Svelte Query, Vue Query - <https://tanstack.com/query/latest>
9. Introduction | Expo Router - <https://expo.github.io/router/docs/>
10. Railway - <https://railway.app/>
11. Introduction - Rust By Example - <https://doc.rust-lang.org/rust-by-example/index.html>
12. Overview · Serde - <https://serde.rs/>
13. food-data\_central\_api | 1.0.1 | fdcnal | SwaggerHub - <https://app.swaggerhub.com/apis/fdcnal/food-data_central_api/1.0.1#/FDC>
14. PostgreSQL: Documentation: 16 - <https://www.postgresql.org/docs/16/index.html>
15. Diesel is a Safe, Extensible ORM and Query Builder for Rust - <https://diesel.rs/>
16. reqwest – Rust - <https://docs.rs/reqwest/latest/reqwest/>
17. react-native-health – npm - <https://www.npmjs.com/package/react-native-health>
18. About React Native Reanimated | React Native Reanimated - <https://docs.swmansion.com/react-native-reanimated/docs>
19. Expo Documentation - <https://docs.expo.dev/>
20. Getting Started | Axios Docs - <https://axios-http.com/docs/intro>
21. Tokio - An asynchronous Rust runtime - https://tokio.rs/