Technical specification

Edibl

Favour Yahaya favour.yahaya2@mail.dcu.ie 23369606
Hubert Iwanow hubert.iwanow2@mail.dcu.ie 23422334

Table of contents

1. Introduction

1.1 Overview
Provides a brief (half page) overview of the system / product that was developed. Include a description of how it works with other systems (if appropriate).

1.2 Glossary
Define and technical terms used in this document. *Only include those with which the reader may not be familiar.*

2. System Architecture

This section describes the high-level overview of the system architecture showing the distribution functions across (potential) system modules. Architectural components that are reused or 3rd party should be highlighted. Unlike the architecture in the Functional Specification - this description must reflect the design components of the system as it is demonstrated.

3. High-Level Design

This section should set out the high-level design of the system. It should include system models showing the relationship between system components and the systems and its environment. These might be object-models, DFD, etc. Unlike the design in the Functional Specification - this description must reflect the design of the system as it is demonstrated.

4. Problems and Resolution

This section should include a description of any major problems encountered during the design and implementation of the system and the actions that were taken to resolve them.

5. Installation Guide

This is a 1 to 2 page section which contains a step by step software installation guide. It should include a detailed description of the steps necessary to install the software, a list of all required software, components, versions, hardware, etc.

1.Introduction

1.1 Overview

The system developed is a virtual pantry ios app which tracks a user's food products. The app keeps track of information such as the expiration date of the product, the storage location of the food and the category of the food e.g (bread, yogurt). Products in the user's pantry are stored in a local database for offline usability. Users are also given an extended shelf life value for the products which displays how long a product is safe to eat after expiry date based on the storage location of the product and whether it's been opened. Information to tell whether food has expired is also given which provides key texture , look ,smell and textile indicators. Users receive notifications regarding when product is soon to expire or has expired via accessing ios permissions. The app also includes a recipe recommendation system which recommends users possible recipes which can be made with products in the user's pantry. Once the user selects a recipe to make, the ingredients used are automatically removed from the database and the recipe is added to the user's recipe history. The recipe recommendation system keeps track of previously made recipes and won't recommend them again unless no other recipes are available. Users are able to rate recipes and this affects which recipes are re-recommended. Users are also able to save recipes. The app uses an external api to find recipe information. The system has two databases hosted. Product database which holds over 13,000 food products sold in Ireland all with their barcode. When a user scans a barcode this system is queried. The system also has a food information database which contains over 600 different food types , its extended shelf life past expiry date in the pantry, fridge and freezer and also general information regarding the food. The system also has unit tests regarding the essential functionalities of the app.

1.2 Glossary of terms

SF: Safety window, how long after the best before date the food is safe to consume, this is calculated with the use of the food database and its expiry data
SQL: Structured Query Language
API: Application Programming Interface
DB: data base
Item: Any food item which the user may add using a barcode or manual entry, example: Tayto, cheese and onion, 85g
Pantry: This is where all of the items are stored for the user
Food Database: A large database which contains all food information such as the name, barcode number, item type (example: dairy), safety guidelines of item type, user survey ratings. This database will be hosted online and use Postgres
Jest: Javascript based testing framework

2. System Architecture

2.1 High-Level Architecture Overview

Here's an overall description of the system structure.

- **Client Side:** Ios App. This is the main user interface which users will be interacting with. Calls upon backend and external services for certain functionalities e.g. retrieving data from hosted databases or getting recipe information. Also responsible for local management of user data e.g. food in users pantry and its information. Created using expo for cross platform development.

- **Notifications:** within Client Side.
- **Local Storage Layer:** Local SQLite database used for the storage of user's information. Specifically, the local as a database is lightweight and allows for offline usability of the app.
- **External services:** Spoonacular, third party recipe API which recommends users recipes based on ingredients provided. IOS permissions are also accessed and enabled to use the camera and notification system.
- **Backend Services:** Cloud hosted database with two tables. Product information and Food information.
- **Testing Services:** Utilizing Jest-based test scripts for client. Tested after each change integrated. Used basic test script with imported asserts module for server.

2.2 Core System Modules

Breaking our system architecture into functional modules.

1. Pantry Management Module

- Local SQLite database for storing and retrieving the items and information of the user's products
- Tracks information of product such as its expiration date, safety window, current location and category
- Location changing of products

2. Notification Module

- Notification system which monitors expiration data of products.
- The notification system schedules alerts based on proximity to expiration date.
- IOS notification system which allows for notification systems to be shown on the phone.

3. Recipe Recommendation Module

- External recipe API integration in order to obtain information. Third party.
- Recipe recommendation algorithm which prioritises recommending recipes using the most soon to expire food products.
- Recipe recommendation system which alters which recipes are recommended to the user later on.
- Recipe history tracking to prevent over repetition in recommendation.

4. Backend service (Hosted Database)

- Hosted on supabase. Platform which provides Baas services.
- The product table contains information regarding 13,000+ Irish food products alongside their barcodes.
- The food table contains information about 600+ food types and their shelf life data.
- Both tables use direct querying for the access of information

5. Barcode scanner module

- Camera permissions need to be granted by user in order to scan
- Barcode recognition
- Uses expo-camera

6. Testing Module

- Unit testing scripts placed in the client and server of the app.
- Testing covers basic functionalities of the app such as pantry manipulation, querying and recipe manipulation, notifications and accessing hosted databases.
- Using Jest as it is a javascript framework and provides reports regarding coverage.

2.3 Reused components

Client Application:

- SQLite - local database
- Expo notifications - user notifications
- Expo camera- Camera access and barcode scanning

Backend

- Supabase - Hosting database
- PostgreSQL framework

2.4 Deployment Architecture

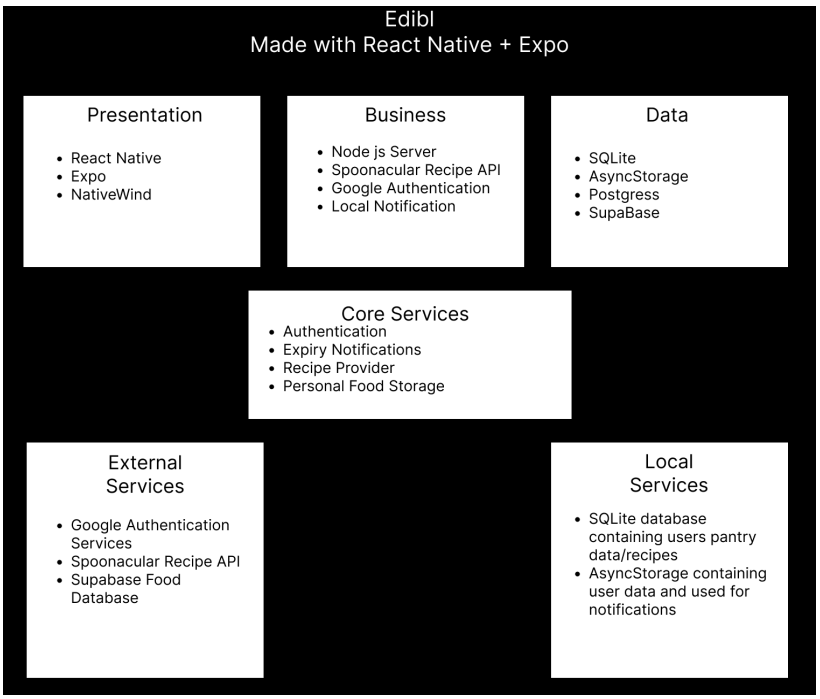
- Ios device- Hosts application and local database itself.
- Cloud servers -Host product information and food information tables
- Internet - Hosts external recipe API service

3. High-Level Design

3.1 System Design Overview

Edibl is a React Native and Expo mobile application which features both client and server architecture for storing data. An internet connection is needed in order to use all of the features but can also be used offline but sacrifices features such as, getting new recipes, logging in with Google Authentication or logging scanned food automatically with SupaBase food database.

Below is the architecture diagram of Edibl.



3.2 System Components

3.2.1 Presentation Layer

The presentation layer within Edbil is built using Expo and React Native. Each page features styling which is implemented using Nativewind.

Key UI Components:

- **Login screen:** Initial screen a user sees upon first entering the application, features Google's OAuth for logging in with Apple Id.
- **Homepage:** Expo-Router based landing page, which features navigation to all of the other pages, only available after logging in.
- **Barcode Scanner:** Camera view which can only be operated once granted user permissions, used for scanning food barcodes.
- **Pantry management screen:** The users own view of the food items within their pantry which is sorted via expiry date, where items with earliest expiry appear near the top.
- **Suggested recipes screen:** Displays recipes based on users current items within their pantry and items which are near expirations are prioritised, history of recipes is kept to not provided the same recipes multiple times.
- **Detailed recipe screen:** Full-screen view of a given recipe after pressing a recipe from the suggested recipe list, users can save their favourite recipes for later.
- **Detailed item screen:** Full-screen view of a given item within the users pantry, provides details on how to prolong its shelf life.
- **Saved/Favourite recipes screen:** A list view of the users saved recipes which can be clicked in order to go to recipe details of given recipe.

The UI is made up of a container/presentational component where the presentational component is used for rendering what the user sees while the container component manages the state.

3.2.2 Business Layer

This layer contains the logic of the application which is split into various components.

Authentication:

- Interface provided by Google OAuth 2.0
- This manages the user session when logging in with the use of AsyncStorage
- Has functions such as saveUser(), getUserInfo(), logOutUser(), checkLogIn()
- Does not store the user login information after logging out, it is provided with the use of Google OAuth 2.0

Recipe Suggestions:

- Provides users with recipes with the use of Spoonacular's free API plan, an internet connection is required
- Matches the users ingredients which are soon to expire to be prioritised in the recipe making process
- An algorithm checks whether a recipe had been provided recently in order to provide the users with a better recipe selection each time
- Recipe can also cached and be saved to be viewed offline
- After recipe is made the ingredients the user had in pantry are then removed

Notifications:

- Local notifications are made possible with expo-notifications
- Daily expiry notifications are sent out with items in pantry which have less than 3 days from their expiry, checks for all items at 9:00 am daily
- Removes items from pantry automatically if not used or removed by user, a notification is sent out to throw away expired items mentioned

Local Database:

- Uses SQLite to manage both food and saved recipes of each user
- Utilises special queries for expiring items in pantry and their subsequent removal
- Allows users to manually add items into pantry without an internet connection

3.2.3 Data Layer

This layer will outline the various databases used for each of their respective uses.

SQLite:

- Used for local storage of each user, user can add food manually whilst offline
- Contains information regarding the food, most notably the id, category, storage location, expiry date, safety markers
- Allows users to save their favourite recipes, which are cached and can be viewed on a separate page

AsyncStorage:

- Used for storing authenticated user info

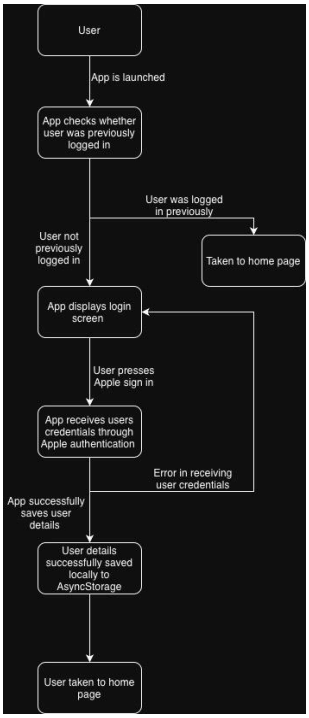
- Gets the users state across the whole app allowing to view certain pages for only logged in users

Postgres:

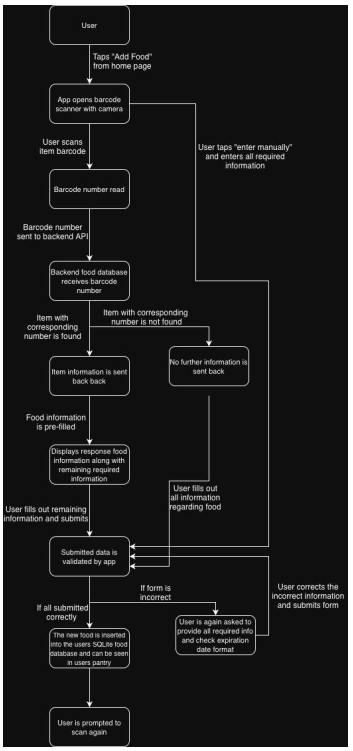
- Main database storage for food which contains more than 13,000 different individual items
- Hosted on SupaBase, does require an internet connection in order to work
- After a barcode is scanned, its id is then checked whether it can be found within the food database and if it is all of its information is then sent to the client side

3.3 Data Flow Diagrams

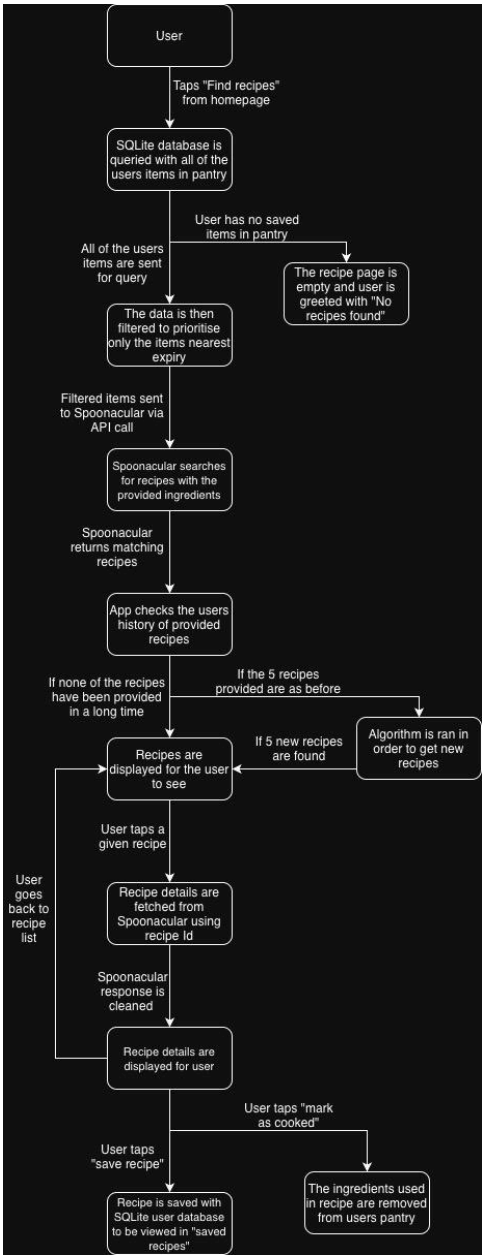
3.3.1 User Authentication



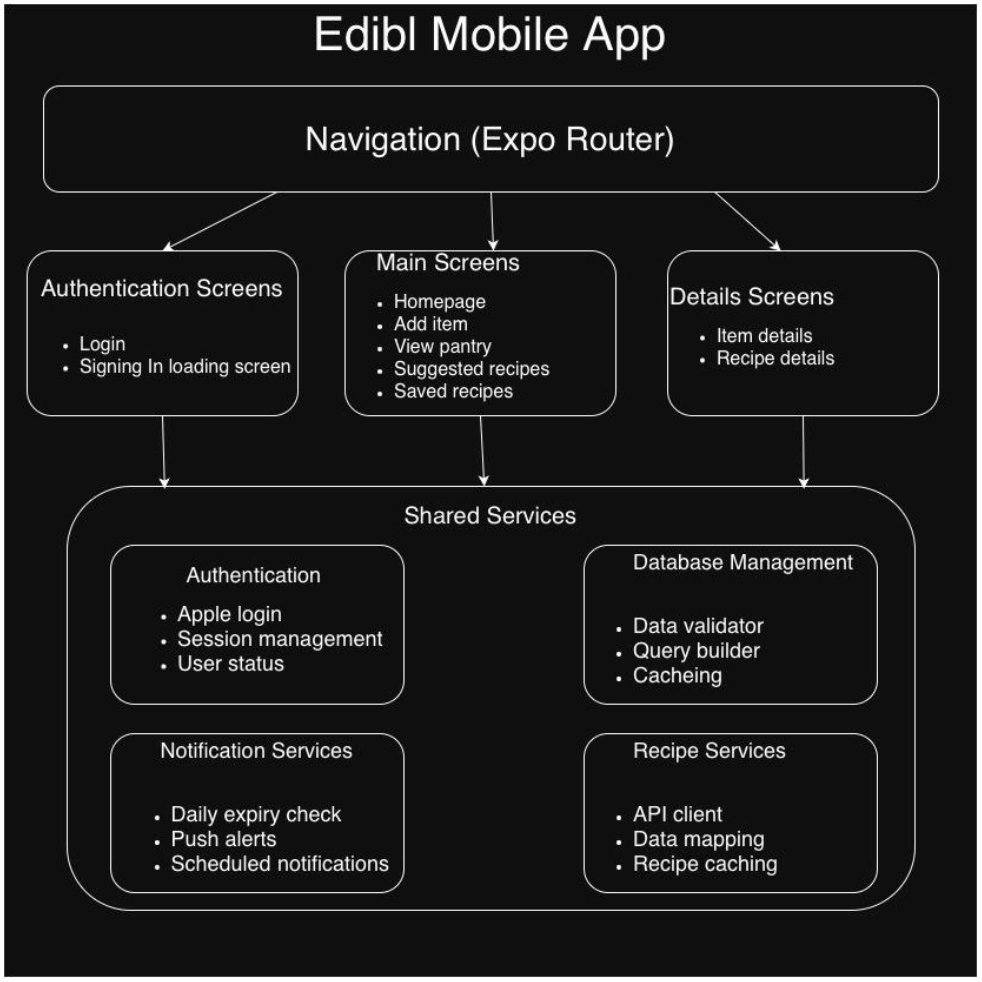
3.3.2 Adding Item to Pantry



3.3.3 Recipe Suggestions Flow



3.4 Component Interaction Diagram



3.5 External System Integration

3.5.1 Google Authentication

- App receives the users profile via an OAuth token
- Data is secure as user data is not stored on any server, which improves security
- Easy for users to sign in, simply press one button and they are logged in

3.5.2 Spoonacular Recipe API

- Limited to 50 requests per day
- Response is in JSON format which needed to be cleaned before displaying the response information to users
- Matches recipes which use the most amount of provided ingredients, responds with recipe Id
- A recipe Id is then used to find further information about said recipe such as its instructions

3.5.3 SupaBase Food DB

- Managed via Node.js, a custom made schema
- Used for barcode lookup of items scanned by users
- Runs using SupaBase, needs an internet connection in order to fetch the food information after scanning barcode number

- Takes in barcode number and returns the corresponding food information
- Manual entry required by user if the barcode number is not found in the database

4. Problems and Resolutions

Here's a list of the problems we faced throughout the development and implementation of the system described above.

Ios app development on windows

This problem first occurred when downloading dependencies and initialising the react native app. We soon realized that developing an ios app on windows would be difficult. You are able to develop the app on windows however you can't build or run the ios app. This was a huge problem as this would prohibit us from testing and configuring the app.

Our solution was to use a react native expo app instead of just a react native app. Using expo made the development process substantially easier as it allowed us to run the app on any device using the expo go app via a QR code.Expo essentially worked as an intermediate. Although only one of us could create the final build for the app since they had a macbook, using expo was our best and most realistic option.

Product database verboseness

We collected all the data for our product database from open food facts, an open source database home to over 4 million unique food products. Our initial plan was to collect all the data from the database however we soon realized how unrealistic this goal was. Collecting this much data would already take far too long without considering rate limits. 4 million entries is also unnecessarily big.

We decided to reduce the scope of the data collected. We only collected products from Ireland which reduced the total number of entries to 13,000. Although there are obviously many products in Ireland imported from other countries, this approach reduced time spent collecting data while also maximizing our coverage.

Local database

Initially Redis was meant to be used for push notifications to alert users of upcoming expiry dates of food. This was prior to learning about expo's great built in libraries which included expo-notifications. This allowed for much easier implementation of notifications and once again didn't require a server to run the Redis database. They are not actually push notifications as they are sent . Using expo's built in library was a simpler and more sensible solution

Barcode Scanner

Our barcode scanner worked well, always recognizing barcodes at different angles. One problem we found was that the barcode would be less effective in darker lighting. In order to negate this issue we integrated manual entry of food products. Another issue we faced at the start with our barcode scanner was at the start, only a single barcode could be scanned before the app would essentially break and not allow for further scans. This was because the scanner would scan the barcode multiple times a

second and the user would need to click 10-20 times for the “scan again” button to disappear. This was because the app did not know the scanner had scanned an item so a clause of whether it had been scanned or not was added to prevent this. So we added a function to set the state the scanner is in so if a barcode has been scanned it will no longer look for other barcodes to scan.

Syncing food and product db

Our biggest problem during this week was figuring out how to map the data from our product database and our food database. Although this seems easy at first, just match items based on name, eg. matching primary category bread = bread, and greek yogurt to yogurt however this gets complicated very quickly eg. Using this logic would lead to soy sauce and soy beans falling under the same category of soy.

Our first solution was a fuzzymatching approach, categorising using name patterns and keeping a confidence level for each matching. If the confidence level was below the threshold around 70% , we would manually review the mapping and either accept or alter it. Although this approach seemed promising it turned out to be very inaccurate. Our food database had around 13000 entries and our product database had around 200 which meant the fuzzy matching model did not have enough data to produce accurate results.

Our best and final solution was a heuristic approach. We compiled a list of the 500 most popular descriptors (primary and secondary) in our app. We then filtered out all of the descriptors which we deemed too vague eg: meats. For each descriptor remaining we manually matched it to its best fitting category in our food database. Finally we created another database which contained these mappings, so they could be queried. Although this approach took more time, it was definitely our best choice as it resulted in the largest widespread coverage. It also ensured a high level of accuracy since each mapping was manually chosen.

Recipe recommendation and history

We needed to develop an algorithm for recommending recipes to users. Showing users the same recipes repeatedly would result in reduced user satisfaction. We also wanted to ensure recipes maximized the ingredients available to the user.

In order to solve these issues we :

- Created a recommendation algorithm which prioritized recipes with soon to expire foods, ensuring they were used before expiry.
- Added a database to keep track of recipes previously made by users. We then created a page where users could see their previously made recipes and rate them. From this, we developed a basic recommendation system system.
- To allow for variety in a user’s recipe selection, we wouldn’t recommend the same recipes twice. In the case that there weren’t any new recipes to recommend, we will supply them with recipes made before starting from the highest rated decreasing.

Best practices

A problem throughout development was following best practices when it came to naming. This problem becomes especially apparent when pushing changes when we get confused by the similarities in the names of variables and functions e.g food, foodData, Food, getFoods, foodNames. Around halfway through our project we decided to analyze our code and alter and fix any names which could lead to any possible confusion. This resulted in greater clarity in the second half of our project

Collecting data for food information database

Towards the beginning, a main goal of ours was initialising our food information database. This database would store information about different foods such as how long after their expiry date are they still safe to consume and how to identify if the food is no longer safe to eat. We firstly needed to find a source for this information.

We had quite a few options for where to find this data however we decided to go with eatbydate as it contains a large amount of data and the structure of the site is consistent meaning it was easier to webscrape. We compiled a list of all foods outlined on the site and ran a script which would visit each foods individual page, webscrape its information and finally store this data into our food database also hosted on supabase.

Google auth

This issue occurred when developing our login page. We decided to use google and apple as methods of logging in however there were many problems with the google authentication system. The user was stuck on the login screen even after logging in successfully and could not go anywhere else in the app. Not the intended outcome as the user should've been taken to the homepage once logged in. But if the app was closed and then opened the user is logged in and taken to the home page upon entering the app again. The problem was that the user state was not being changed so even if the app receives the correct response the user is stuck on the login screen. Added an additional file which monitors the users current status within the app.

Setting up clients was also a challenge and had to follow a youtube tutorial in order to set everything up correctly. when a user tried to login using google they would get the following error:

This was because the webClient was not set up correctly but also hadn't added the email address into the test users. This took quite a while to find the solution online.

5. Installation Guide

Prerequisites
The following must be installed before setup:

- Node.js 18 or higher,
- npm (included with Node),
- Git, and
- Expo Go (on a physical device) or Xcode (iOS) / Android Studio (Android) for a simulator.

App Setup

1. Clone the repository: git clone <repository-url>
2. cd edibl
3. Install dependencies: npm install
4. Change url in config.ts file to your ip address
5. Start the development server: npm start
6. From the Expo menu, press i for iOS, a for Android, w for web, or scan the QR code with Expo Go on a physical device.

The app uses native modules e.g camera, SQLite, notifications. If prompted, build a development client first by running npm run ios or npm run android instead of npm start.
Server Setup

1. Navigate to the server folder: cd back to code directory cd server
2. Install dependencies: npm install
3. Create a .env file in the server folder with the following variables — obtain values from the project administrator or Supabase dashboard: DATABASE_URL, SUPABASE_URL, SUPABASE_KEY, PORT = 3000.
4. Generate the database client: npx prisma generate
5. Start the server: npm run dev

Recommendation
We would recommend using expo go for starting the app as it allows for running the app regardless of platform. All you need to do is scan the QR code.

Running tests
client(edibl): npx jest
Server: node --test server.test.js

Troubleshooting

- Port conflict: run npx kill-port 8081 then retry.
- Native module error: build a dev client with npm run ios or npm run android.
- Android emulator not detected: start the emulator in Android Studio and run adb devices.
- Database connection error: verify the .env values match your Supabase project settings.