UNIVERSITY OF LONDON

(CM3050)
Mobile Development
Final Report

Nouri

Ellen Faustine

220570273

Table of Contents

Concept development

Overview

Nouri is a mobile nutrition tracking application developed with React Native and Expo. It helps users track their daily meals and snacks by logging foods through barcode scanning or manual entry. By scanning a barcode, users can also instantly view the nutrition facts of any product. They can set and customize daily nutrition goals, explore a variety of recipes, and bookmark favorites for quick access. Nouri displays nutrient intake with animated progress rings and keeps a daily log of foods consumed. Users can view and edit their profile, including their display name and avatar. With its clean and interactive design, Nouri aims to make nutrition tracking simple and engaging, supporting users as they build healthier eating habits.

Background & Motivation

Maintaining a healthy diet is an essential part of overall well-being, but many people struggle to monitor their nutrition consistently. Although there are many nutrition tracking apps available, some are cluttered, difficult to use, or lack features that fit into daily routines, such as easy food entry and recipe suggestions. As a student who aspires to eat healthily, I sometimes forget to eat or lose track of what I have consumed because of the demands of school life. Furthermore, I noticed that many existing apps require a subscription, focus only on calorie counting, or make it difficult to log foods quickly, especially snacks or homemade meals. These experiences motivated me to create Nouri, a solution that prioritizes user experience by making food logging simple and visually engaging.

Aims & Objectives

This project aims to design and develop a mobile nutrition tracking application that makes monitoring daily food intake simple, engaging, and accessible. It also seeks to support healthier eating habits by providing users with tools to log meals, set goals, and explore recipes that fit their lifestyle, as well as to demonstrate technical skills in React Native and Expo through the creation of a complete and fully functioning mobile app.

The objectives of this project include:

- Implementing flexible food logging options, including barcode scanning that allows users to check a product’s nutrition breakdown, as well as manual entry through a form
- Providing visual nutrition feedback using animated progress rings and detailed daily intake logs.

- Enabling custom daily nutrition goals with persistent storage, so users can personalize the app to their needs
- Building a clean, interactive interface that emphasizes usability, consistency, and accessibility.

Target Audience

The target audience for this app includes individuals who want to track their daily nutrition intake and become more mindful of what they eat. This app is designed for users who prefer a simple and accessible way to log their meals and keep a record of their daily consumption. It is particularly useful for students, like myself, who may forget meals or struggle to track their diet during busy schedules. It is also suited for working professionals and other busy individuals who want a quick and convenient method to monitor their nutrition without using complicated or subscription-based apps.

Scope & Limitations

This project enables users to log meals and snacks by scanning barcodes or entering details manually, with all nutrition data stored locally for offline access. Users can check the nutrition facts of any product, set custom nutrition goals or use predefined FDA values, and track their progress through animated nutrition rings. The app also provides recipe exploration through preset categories, with each recipe showing nutrition details, dietary tags, bookmarking options, and the ability to add recipes to daily intake. In addition, users can review their food history by date, receive reminder notifications, and manage their profile by editing their name, avatar, and bio.

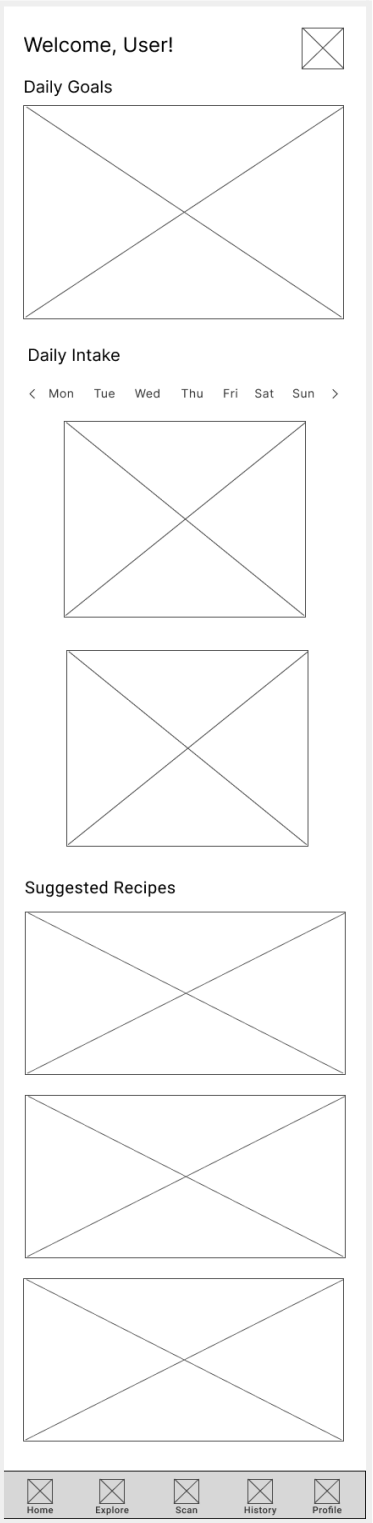
At this stage, the app supports only single-user tracking, and profile data will reset if the prototype is cleared. Features such as recipe search, cloud synchronization, multi-user support, social or sharing functions, and third-party health app integration are not yet included. Although these advanced capabilities remain outside the current scope, the application delivers the essential aspects of nutrition tracking and provides a strong foundation for future improvements.

Wireframing

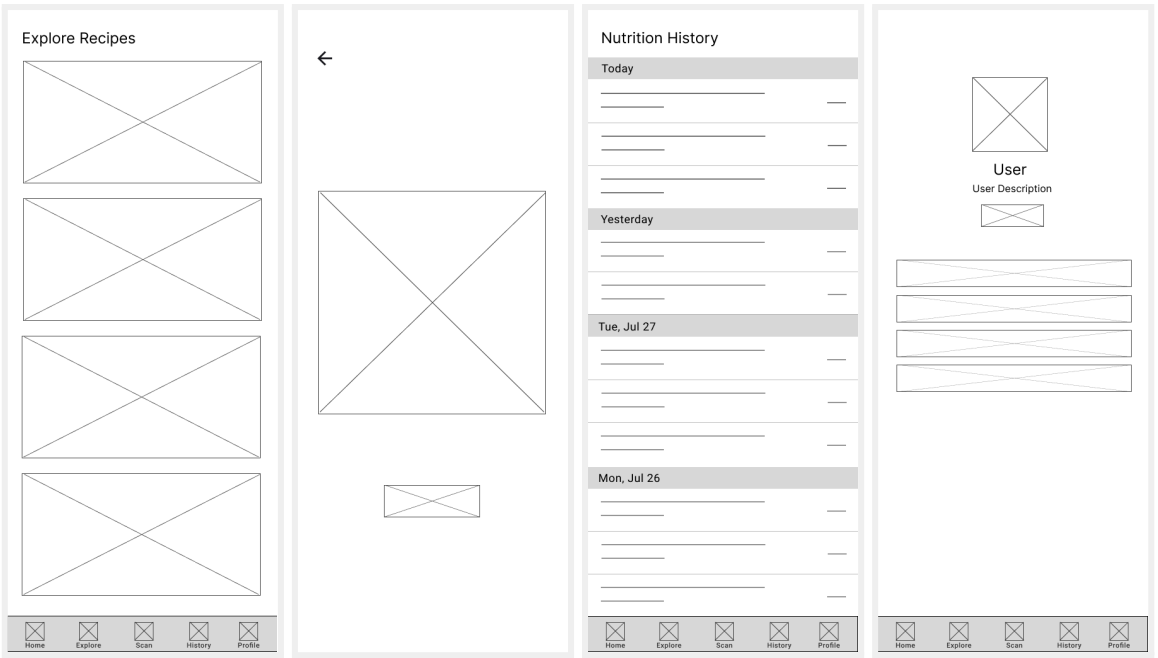
For this app, I created three sets of wireframes for both primary and secondary screens in Figma and developed interactive prototypes for user testing. The process progressed from low-fidelity to mid-fidelity and finally to high-fidelity, incorporating user feedback at each stage to refine the subsequent version.

Low-Fidelity Wireframes

Starting with low-fidelity wireframes, I created simple and basic layouts to initially map out the look, structure, and navigation of the app. I focused on the five core screens: Home, Explore, Scan, History, and Profile, which are shown below.

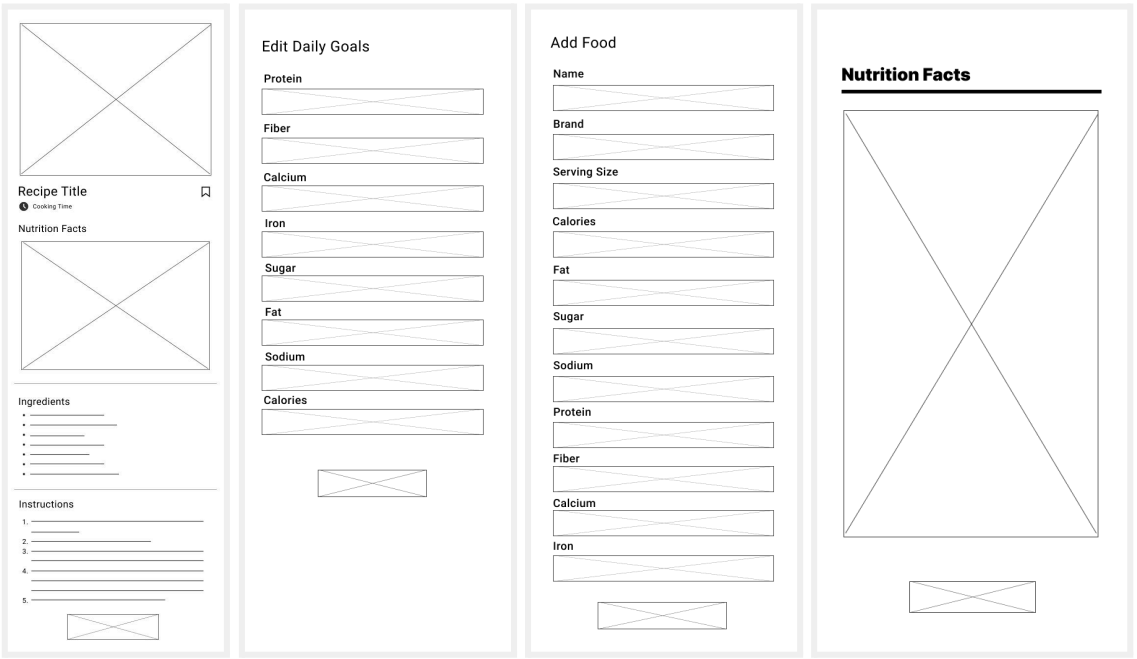


Home Screen



Explore Screen Scan Screen History Screen Profile Screen

Next, I created wireframes for the secondary screens, including the recipe details screen, food entry screen, nutrition label, and edit daily goals screen, which are also shown below.

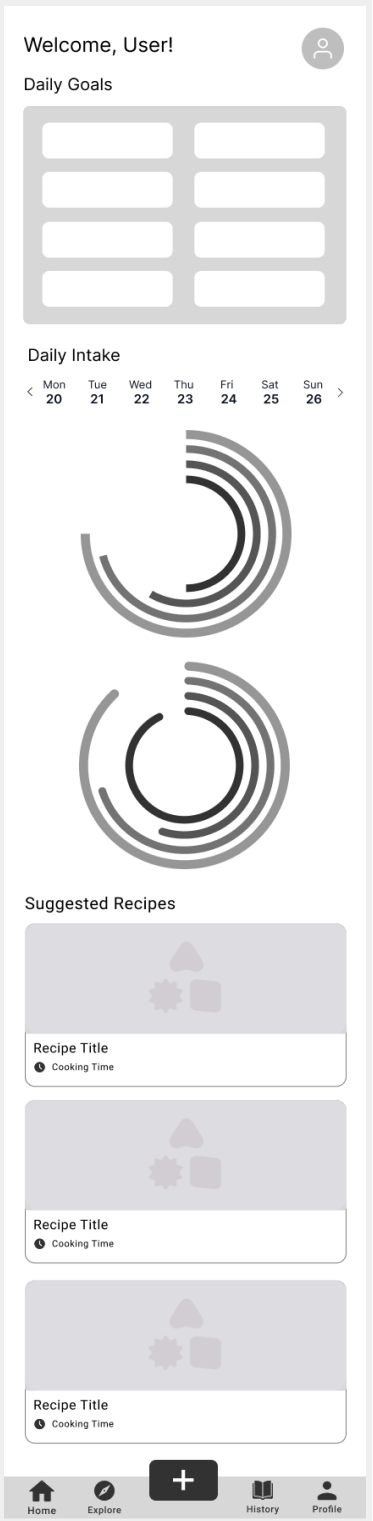


Recipe screen Edit daily goals Food entry Nutritional label

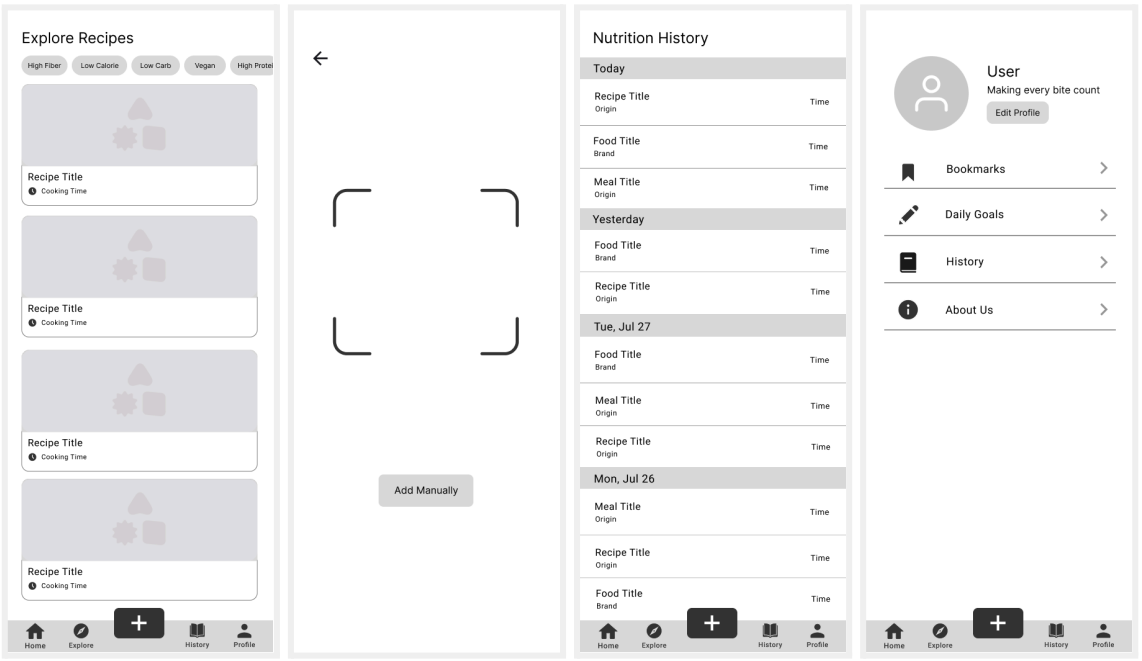
Mid-Fidelity Wireframes

After creating the low-fidelity wireframes, I conducted the first round of user testing to gather feedback on the layouts and navigation. The feedback indicated that the

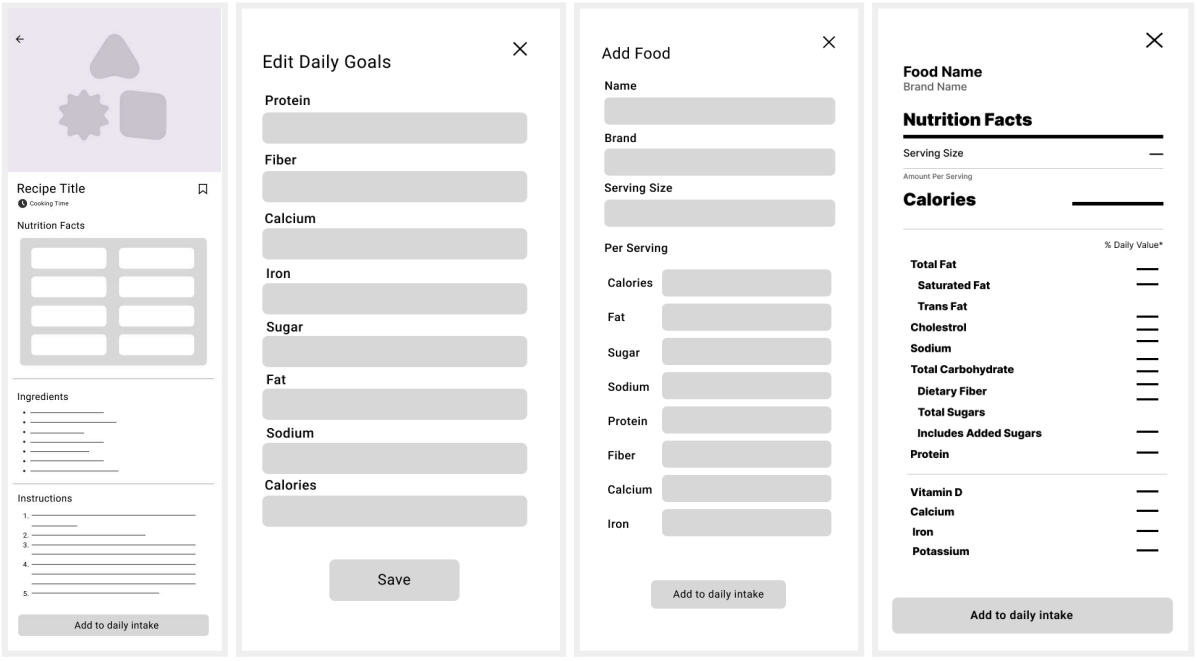
overall structure was good but lacked detail. Using these insights, I created the mid-fidelity wireframes, incorporating more detail and refinement. I began again with the primary screens and then proceeded to update the secondary screens.



Home Screen



Explore Screen | Scan Screen | History Screen | Profile Screen

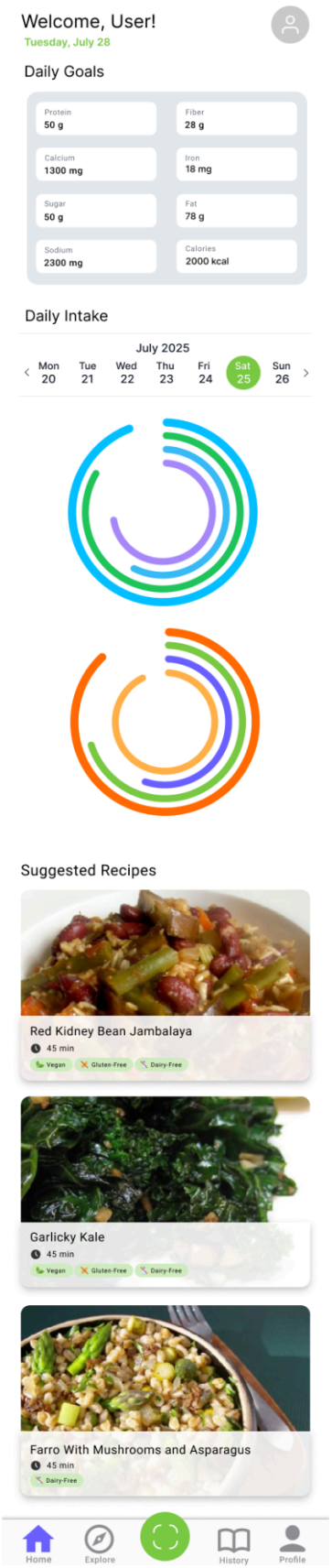


Recipe screen | Edit daily goals | Food entry | Nutritional label

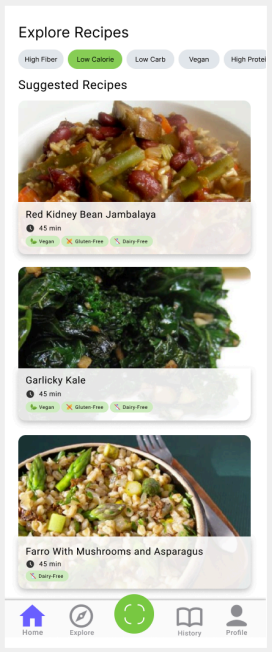
High-Fidelity Wireframes

Just like before, I conducted another round of user testing to gather feedback on the updated wireframes. I used this feedback to refine and improve the layout and structure while progressing to the next set of wireframes, which are the high-fidelity versions. These include images, colors, and additional details, bringing the design

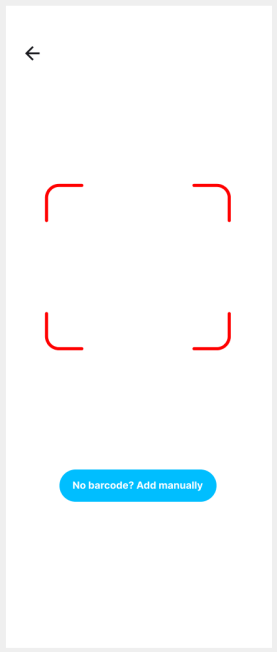
vision to life. I started with the primary screens and then moved on to the secondary screens.



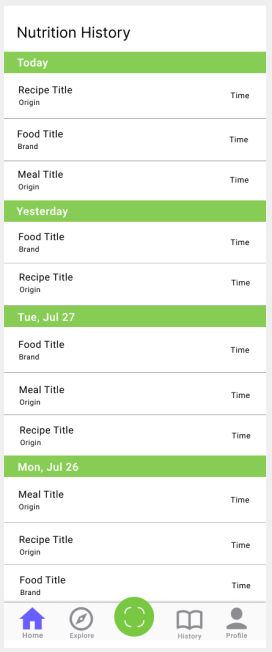
Home Screen



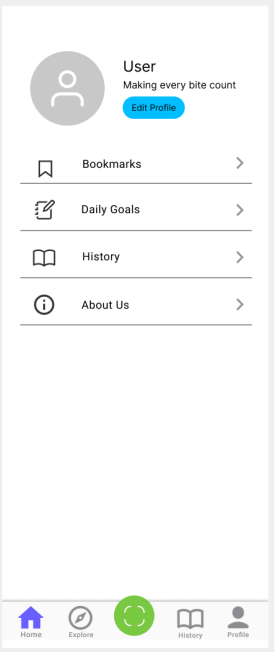
Explore Screen



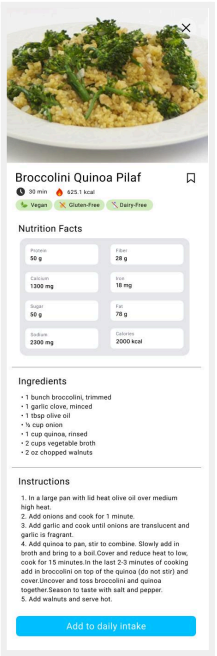
Scan Screen



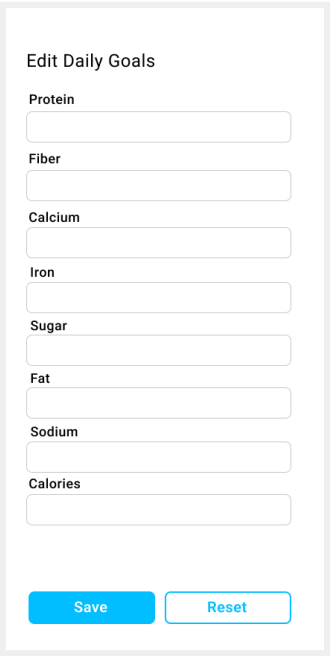
History Screen



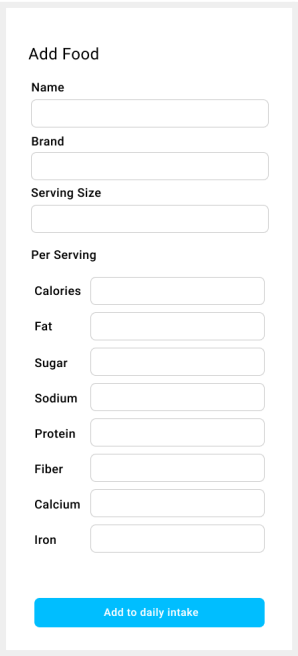
Profile Screen



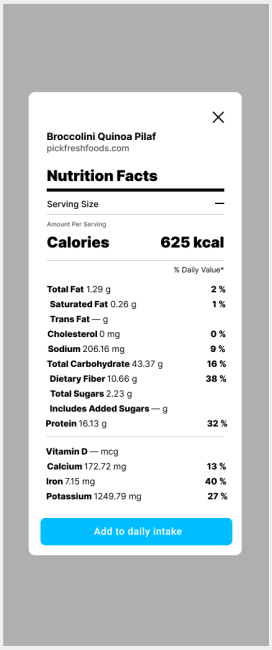
Recipe screen



Edit daily goals



Food entry



Nutritional label

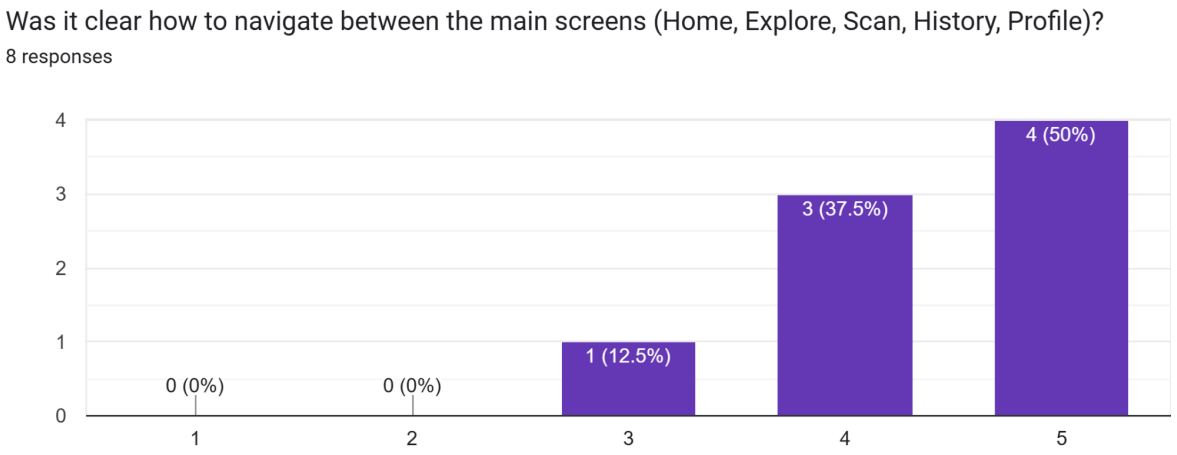
User feedback

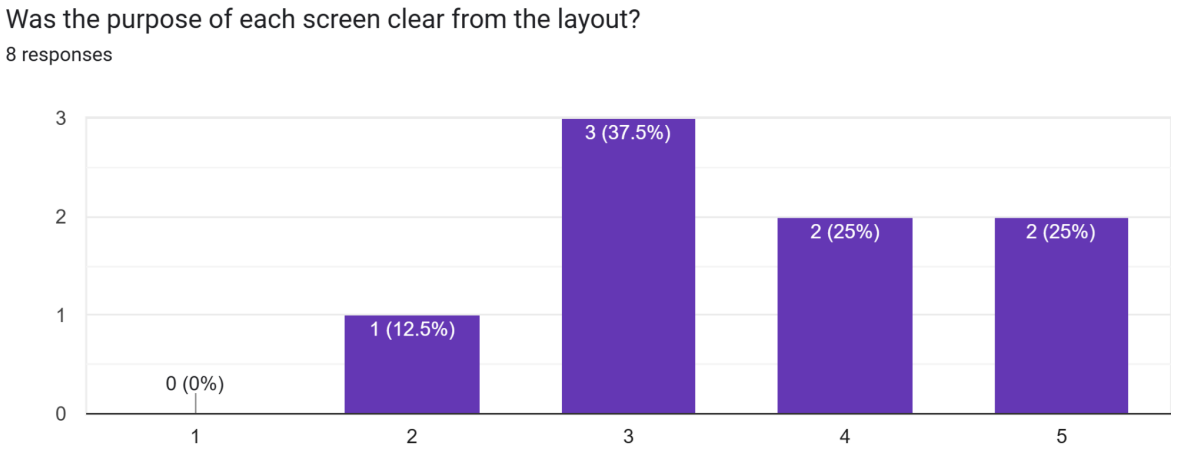
To evaluate the usability and design of the app, I conducted three rounds of user testing using the wireframes. Each round focused on a different stage of fidelity, starting with low fidelity, then mid fidelity, and finally high fidelity, so feedback could be gathered on layout, navigation, and overall user experience. The goal of these

tests was to identify potential issues, understand user preferences, and use the insights to iteratively refine the app's design and functionality.

Low-Fidelity Testing

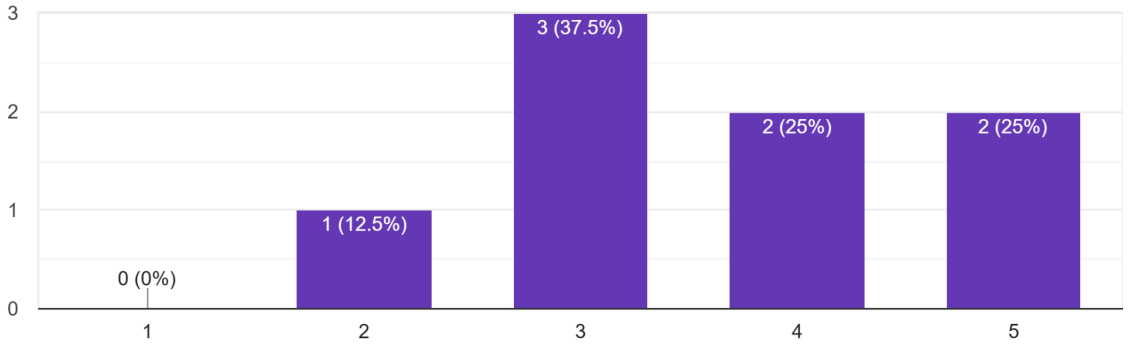
During the first round of user testing with the low-fidelity wireframes, participants were asked to complete basic tasks such as navigating between screens, adding a food entry, and viewing daily nutrition goals. Overall, users found the navigation intuitive, and the core layout of the Home, Explore, Scan, History, and Profile screens was well understood. However, some issues were identified, including unclear labeling of certain buttons, difficulty understanding the purpose of some screens, and challenges with logging new food entries. Participants suggested adding clearer headings and buttons, visual indicators, and more detail for each screen's design. This feedback provided valuable insights that informed improvements in the mid-fidelity wireframes, particularly in refining the layout and enhancing visual cues for navigation. The results of this testing are presented below.





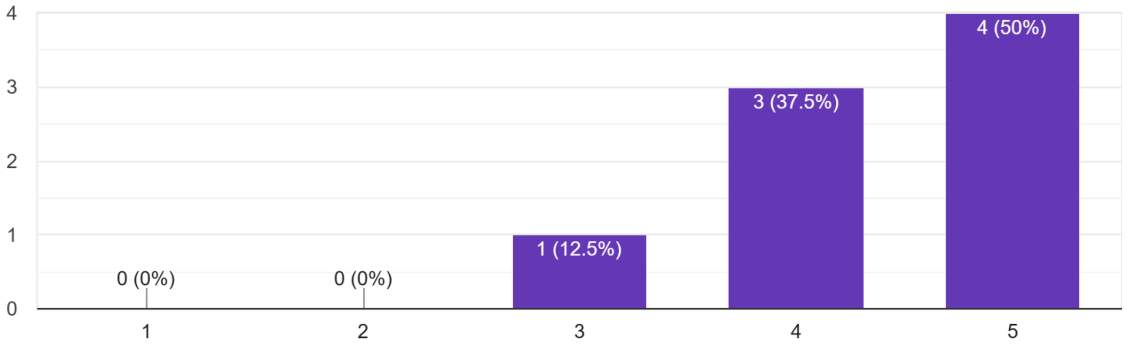
How easy was it to find where to log a new food?

8 responses



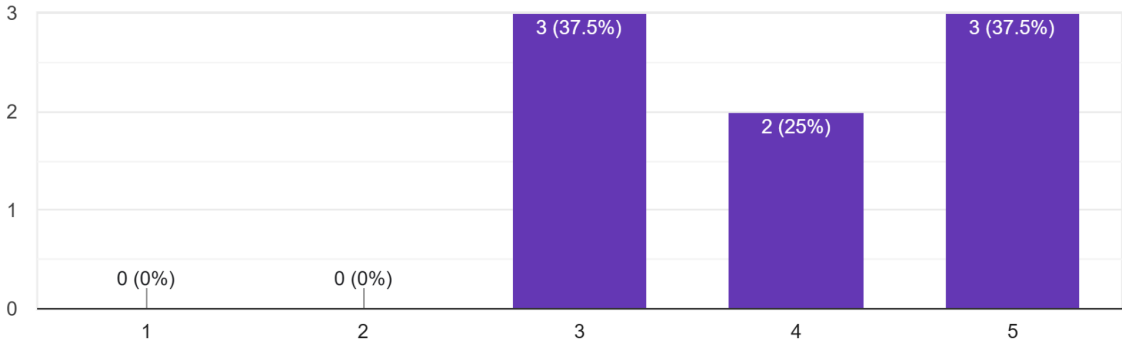
How easy was it to navigate to a recipe's details page?

8 responses



Did the overall layout make sense for completing basic tasks?

8 responses



What are your first impressions of the overall layout and structure?

8 responses

- The wireframe lacks context and is still very vague
- simple and easy to navigate
- it looks empty and plain
- I like it, seems intuitive and simple!
- It's a good first design for the app and I can generally guess what the buttons are for.
- I like the structure, but some screens look a bit empty
- very simple and easy
- I found it a little hard to tell what some buttons and screens, such as the scan or log screen, would do at this stage of the design.

Do you have any suggestions or comments about this layout?

8 responses

- include more textual detail to be more clear
- layout and navigation is good but lack detail/contex
- the layout for the edit daily goals and the food entry screen seems boring and redundant
- Nope, good to continue
- a good starting point! layout is easy to understand but more information regarding the buttons could help
- It might help to add example content and more detail for the btutons
- no problems at all
- please add more design and make it look better

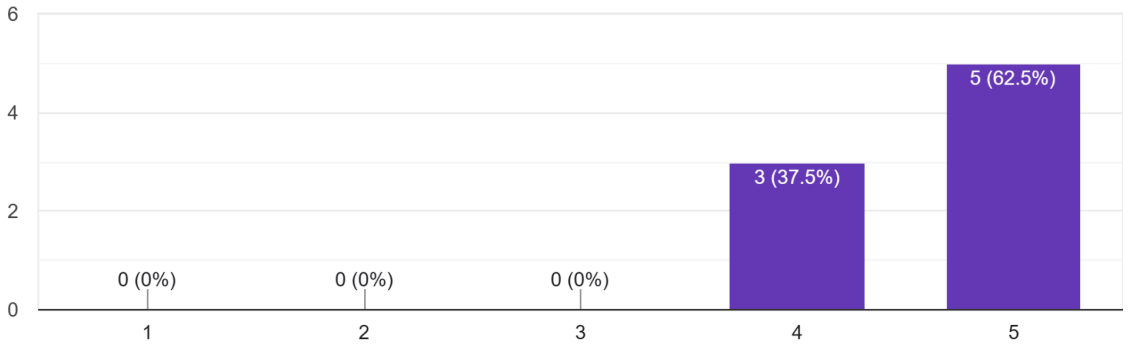
Mid-Fidelity Testing

During the mid-fidelity user testing, eight participants interacted with the prototype to complete key tasks such as navigating screens, adding food, viewing history, and exploring recipes. Overall, feedback was positive, with most users finding navigation clear, icons and labels understandable, and the layout visually improved from the

previous version. Users highlighted the clarity, simplicity, and intuitiveness of the design, particularly praising the nutrition rings and the updated add food form. Some issues were noted, including confusion with the scan screen frame, a non-clickable edit button, and suggestions to add more information to recipe cards, display the current month in the calendar, and make certain buttons more prominent. This feedback was used to refine the interface and prepare the high-fidelity wireframes.

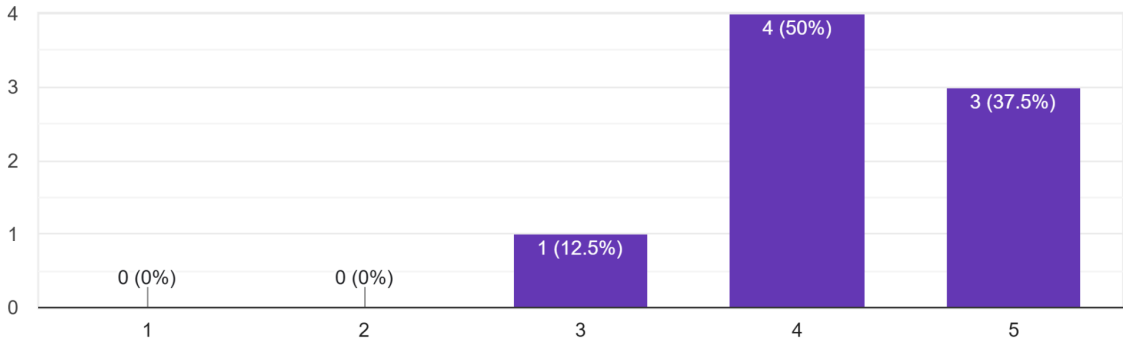
How clear was it to navigate between different screens in the updated prototype?

8 responses



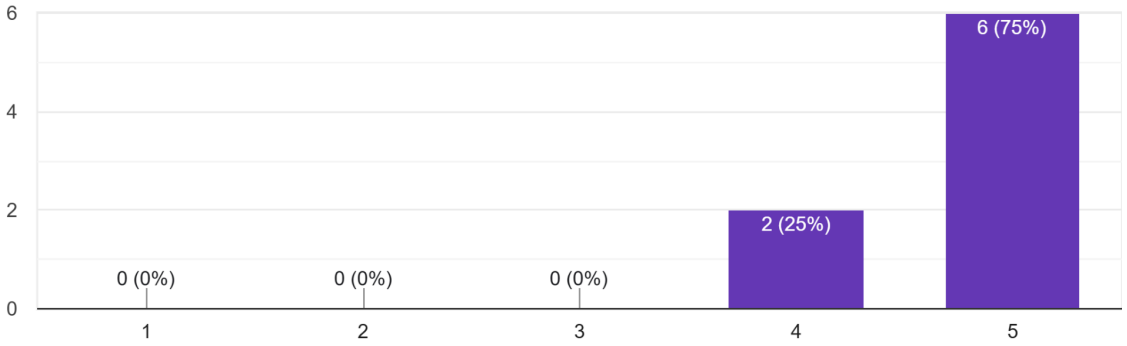
How easy was it to complete each of the example tasks (e.g., adding food, viewing history, exploring recipes)?

8 responses



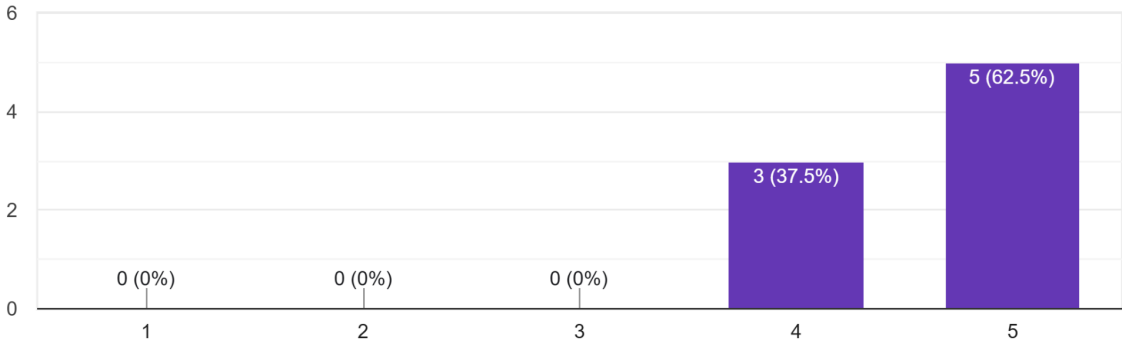
Were the icons and labels on each screen clear and understandable?

8 responses



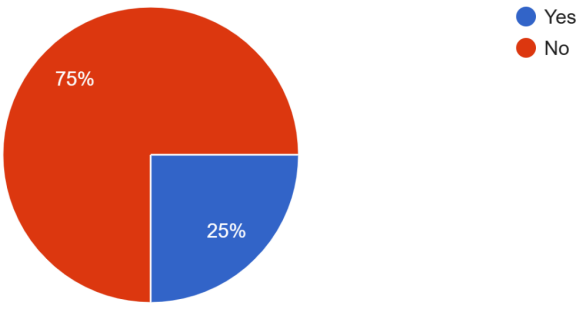
Do you think the visual design and layout improved compared to the previous version?

8 responses



Is there any part of the app that still feels confusing or hard to use?

8 responses



If you selected "Yes", please specify which part and why

4 responses

-

no

the edit button is not clickable

i didn't understand the box frame on the scan screen

What do you like most about the current design?

8 responses

Love the clarity!

navigating between screens feels straightforward and easy

the design is simple and intuitive

i love the use of the rings and the updated add food form!

I love the simple yet not boring look and it looks very clean!

very easy to understand and use, good improvement from the previous one

i love the home page and the nutrition label design

looks very clean!

What would you still improve or change in the app?

8 responses

- Nope
- include more icons or info in the recipe card, it looks a bit empty
- consider adding tags like spicy/vegan/gluten tags for the recipes
- put the current month on the calendar
- maybe make the buttons to be bigger(?)
- incude the month and year for the calendar/week thing
- make the edit profile screen
- nothing

Any other suggestions or comments to make the app easier or more enjoyable to use?

8 responses

- -
- Good to go, love it
- everything looks good!
- use different colours for the rings
- i think make the scan button to be more prominent maybe make it into a square rather than a small rectangle
- good work
- very simple and intuitive design

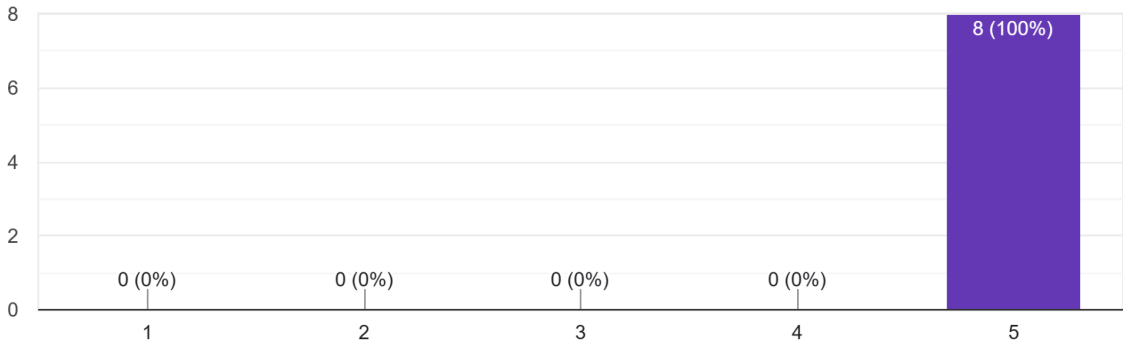
High-Fidelity Testing

During high-fidelity testing, eight participants interacted with the fully designed prototype. Users found navigation clear, the overall layout visually appealing, and logging meals and adjusting daily goals intuitive. Positive feedback highlighted the realistic nutrition label, circular scan button, and clean interface. Minor suggestions

included adding more icons or information to recipe cards, improving the home screen welcome section, and enhancing button visibility. Overall, the testing confirmed that the app provides a clear, engaging, and user-friendly experience while identifying small areas for refinement.

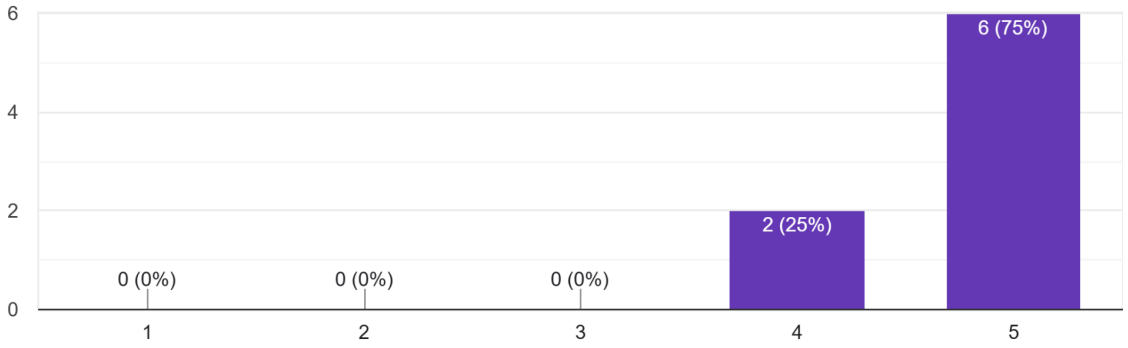
How easy was it to navigate between different sections (Home, Explore, Scan, History, Profile)?

8 responses



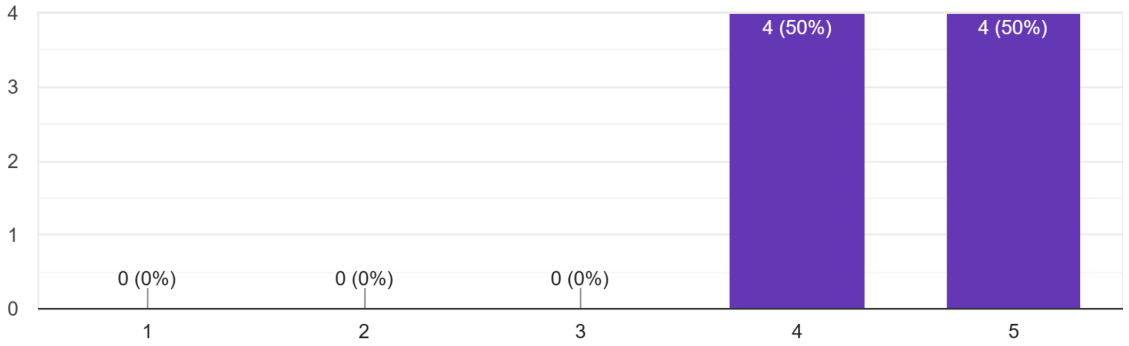
How visually appealing do you find the overall design of the app?

8 responses



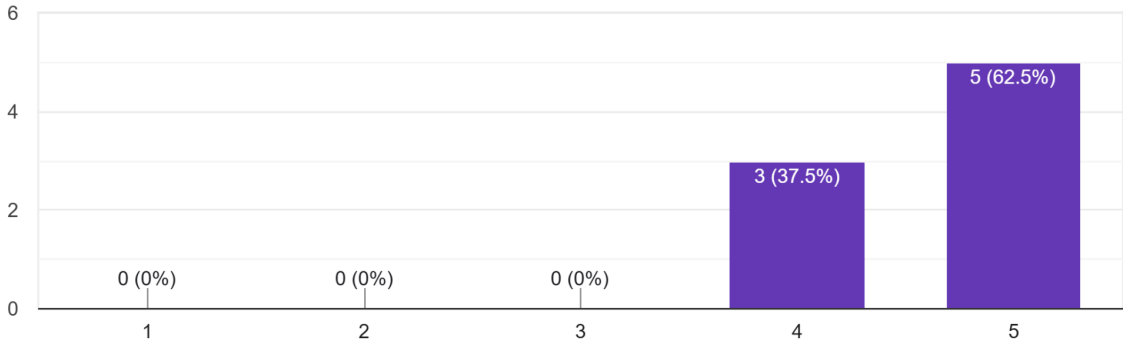
How easy was it to log a new meal or snack using the current design?

8 responses



How intuitive was it to set or change your daily nutrition goals?

8 responses



What do you like most about the updated design and layout?

8 responses

the nutriional label looks very realilstic

very clean and colourful without it being too overpowering

i love the circular button instead of the rectangle

more context

Everything is easy to navigate and use

everything is so straightforward and easy to use

everything is organized and looks good

the clean look

Is there anything you would still change or improve in the app?

8 responses

-

nope

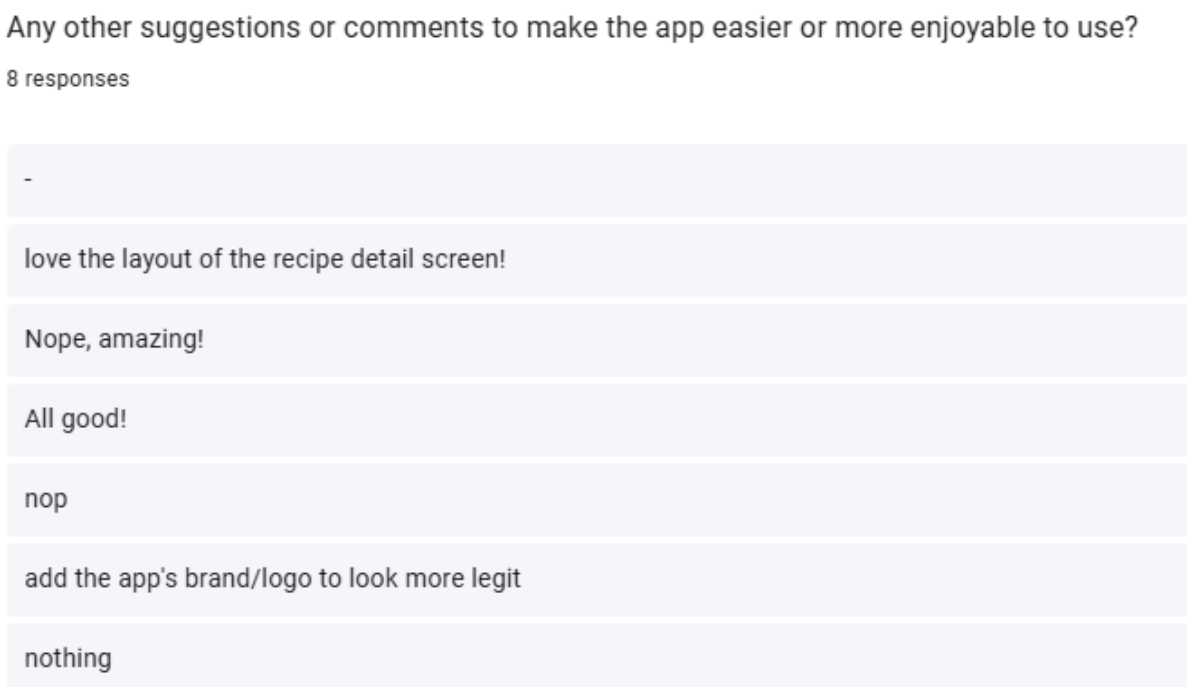
maybe use more colours(?)

Nope

No

the welcome part onthe screen looks a bit empty

nope alls good!



Prototyping

Prototyping for Nouri was conducted using Figma to create interactive versions of the app at different stages of fidelity. Low-fidelity wireframes were first developed to map out the basic layout and navigation. These were then refined into mid-fidelity prototypes with more detail and visual cues, followed by high-fidelity prototypes that incorporated full color, images, and interactive elements to closely represent the final app experience. The prototypes allowed for user testing at each stage, providing valuable feedback to improve usability and interface design. The interactive Figma prototype can be accessed at the following link: https://www.figma.com/design/DlRMVEkUw8I7lZOmF9baLy/Nouri?node-id=75-781&t=7ASEgg3Jk5z1aybi-1

Development

App.js

The App.js file serves as the entry point of this app, handling initial setup and global configurations before rendering the main navigation. It first loads a custom font using expo-font to maintain consistent typography across the app. While the font is loading, a splash screen is displayed for 1.2 seconds to provide visual feedback to the user. Notifications are also set up in this file using expo-notifications. A daily reminder is scheduled at 12:00 pm to prompt users to log their meals or snacks. Permissions are requested from the user, and any previously scheduled notifications are cleared to prevent duplicates. Notification behavior is configured so that alerts are shown without playing sounds or updating the app badge. Finally, once the fonts

are loaded and the splash screen is dismissed, App.js renders the main app interface using a NavigationContainer and the TabNavigator, which manages the primary navigation structure of the app.

Components

In this section, I discuss selected components that implement key functionality and demonstrate techniques learned during the module. These components highlight custom logic, interactive features, and dynamic visualizations that contribute significantly to the app's functionality and user experience.

Nutrition Rings

The NutritionRings.js file defines a component that visually represents the user's nutrient intake using circular progress rings. Each ring corresponds to a specific nutrient and shows the user's current intake relative to their daily goal. The component uses the react-native-svg library to draw the rings and React Native's Animated API to smoothly animate the progress whenever the values change.

Multiple rings are displayed concentrically, with their spacing and radius calculated dynamically based on the component size. The currently selected nutrient is highlighted at the center, showing its name and current intake value. Users can tap on a ring to change the selected nutrient, which updates the central label accordingly. By combining animations, interactive selection, and clear visual feedback, this component offers an intuitive and engaging way for users to monitor their daily nutrient intake at a glance.

Nutrition Label

The nutritional label modal component, which displays detailed nutrition information for a food item, is implemented in the NutritionLabel.js file. It shows both the nutrient amounts and their corresponding percent daily values based on a standard 2000-calorie diet. The component extracts nutrient values such as calories, fats, carbohydrates, protein, vitamins, and minerals, and formats them for clear presentation. Moreover, its design is inspired by the skeuomorphic style of actual nutrition facts labels found on food packaging, making it more familiar and intuitive for users to read.

Each nutrient is displayed in a row with its name, amount per serving, and percent daily value when applicable. The 'formatValue' function ensures numbers are shown with a maximum of two decimal places, and percent daily values are calculated using utility functions that reference standard daily values.

The modal also displays product details including the food name, brand, and serving size. Users can scroll through the nutrient list and close the modal via a dedicated

close button. Additionally, if an ‘onAddToDailyIntake’ function is provided, a button is shown that allows the user to add the food to their daily intake.

The component is styled to enhance readability and usability using React Native’s Modal, ScrollView, and custom layout components. By combining familiar label design with interactive features, this component provides an intuitive interface for users to review detailed nutrition information before adding foods to their daily logs.

Loading Overlay

The LoadingOverlay.js file defines a fullscreen loading overlay component that provides visual feedback when the app is performing a background task. It displays a spinner using React Native’s ActivityIndicator and can optionally show a message below the spinner. The overlay is rendered only when the visible prop is true, preventing it from appearing unnecessarily. It covers the entire screen with a semi-transparent background to focus the user’s attention on the loading state while dimming the rest of the interface. This component is designed to provide a consistent and reusable loading indicator throughout the app.

Screens

In this section, I will discuss the implementation and development of the primary screens, as well as a few selected secondary screens. The focus will be on how these screens were structured, the key components and functionality they include, and the techniques applied during development to create a responsive and user-friendly interface.

Home Screen

The Home screen is the core of the app and serves as the main point of interaction. It defines the main dashboard, providing users with an overview of their daily goals, nutrient intake for today and previous days, and suggested recipes. Users can also select different dates to view the intake for a specific day.

The daily goals section displays the user’s target nutrient intake using a ‘NutritionCard’ component, and users can tap the card to navigate to the edit goals screen to adjust their daily targets. The daily intake section allows users to view nutrient consumption for the selected day, with a ‘WeekSelector’ component enabling easy navigation between different days of the week. Nutrient intake is visualized using two ‘NutritionRings’ components, which display animated circular rings representing the user’s progress for each nutrient relative to their goals. The Suggested Recipes section presents a selection of recipes retrieved from the Spoonacular API. These are displayed as ‘RecipeCard’ components, and the list is shuffled to provide variety and keep recommendations dynamic.

Data for the home screen is loaded and updated whenever the screen gains focus. This includes fetching the user profile, daily intake, and goals from AsyncStorage, ensuring that the displayed information is always up to date. Randomized recipe suggestions are fetched when the screen mounts, providing users with relevant and varied recommendations.

Explore Screen

The Explore screen defines the interface for browsing and discovering recipes within the app. It presents a list of recipe categories at the top and a corresponding list of recipe results below, allowing users to filter and explore meals according to their dietary preferences. The categories are currently predefined and are displayed as interactive horizontal bubble buttons.

When a category is selected, recipes are fetched from the Spoonacular API. To manage API usage and improve performance, the data is cached locally and refreshed only once every 24 hours. The fetched recipes are then shuffled to provide variety and displayed as ‘RecipeCard’ components in a FlatList, with staggered animations for a dynamic visual effect. While the data is loading, the ‘LoadingOverlay’ component is shown to provide feedback to the user.

Scan Screen

The Scan screen allows users to scan food barcodes and log nutritional information. It enables quick capture of a food item’s data using the device camera, fetches nutrition details from the OpenFoodFacts API, and adds the item to the user’s daily intake.

The screen requests camera permissions on mount and provides fallback UI with instructions and a “Go Back” option if access is denied. The main camera view displays a scanning frame, and the barcode scanner detects supported codes. Upon a successful scan, the app fetches product data from the API, normalizes the nutrient values, and displays them in a ‘NutritionLabe’l modal. Haptic feedback and a ‘LoadingOverlay’ component offer responsive and interactive feedback during scanning and data fetching.

Users can also navigate to a manual food entry screen if a barcode is unavailable or if they prefer to log meals manually. The component manages state for scanned products, nutrient data, modal visibility, and loading indicators, ensuring a smooth and consistent user experience. By combining real-time camera scanning, API integration, and modal interactions, the scan screen provides an intuitive interface for efficiently logging nutritional information.

History Screen

The History screen displays the user's nutrition history by showing all previously logged food entries grouped by date, allowing users to review and keep track of their nutrient intake over time.

Daily logs are fetched from AsyncStorage whenever the screen gains focus, ensuring the information is current. The entries are organized into sections, with each section representing a specific day and labeled using user-friendly date strings. If no history exists, a placeholder message informs the user.

Food entries within each section are displayed as pressable items showing the food name, brand, and the time it was logged. Selecting an entry opens a 'NutritionLabel' modal, presenting detailed nutrient information for that item and enabling users to inspect past foods in a clear and interactive way.

The screen uses a SectionList to efficiently render grouped entries and includes styling for headers, items, and modals to ensure readability and visual clarity. By combining historical data, interactive modals, and organized sections, the history screen provides an intuitive interface for tracking and reviewing nutritional intake over time.

Profile Screen

The profile screen provides the user interface for viewing and managing personal information and app-related settings. At the top, it displays the user's avatar, name, and bio, giving a clear overview of their profile. Users can tap the "Edit Profile" button to navigate to the edit profile screen and update their information.

Below the profile header, the screen presents a list of profile options, including bookmarks, daily goals, history, and about us. These options are displayed as pressable rows with icons and labels, allowing users to navigate quickly to the respective screens. The options are organized using a SectionList to ensure efficient rendering and consistent spacing.

The component fetches the user's profile data from AsyncStorage whenever the screen gains focus, ensuring that the displayed information is always up to date. While the data is loading, a LoadingOverlay component is displayed to indicate progress and provide user feedback.

Recipe Screen

The Recipe screen provides the interface for viewing detailed information about a selected recipe. It displays the recipe image, title, preparation time, nutritional information, dietary tags, ingredients, and cooking instructions, offering users a comprehensive overview of the meal.

The screen fetches recipe details from the Spoonacular API when mounted, and the data is also cached locally and refreshed only once every 24 hours to manage API usage limits and improve performance. Nutrient values are normalized using utility functions. Users can view whether the recipe is bookmarked and toggle its status, with bookmarks stored persistently. Recipes can also be added directly to the daily intake, with haptic feedback and confirmation alerts providing responsive interaction.

Dietary tags such as vegan, gluten-free, and dairy-free are displayed visually to give users quick insights into the recipe's suitability. Nutrition information is presented by reusing the NutritionCard component, summarizing key nutrients like calories, fats, protein, vitamins, and minerals. Ingredients and instructions are displayed clearly in scrollable sections, supporting both structured steps and raw text instructions.

The screen uses a ScrollView with a sticky header and shadow effect for improved navigation, and a fixed footer contains the "Add to Daily Intake" button. Loading states are managed with a LoadingOverlay component to maintain responsiveness during API requests. Overall, the Recipe screen combines API integration, interactive components, and dynamic styling to provide an intuitive, informative, and user-friendly recipe detail interface.

Food Entry Screen

This screen provides a form for users to manually enter a food, snack, or meal without using a barcode. It allows users to input details such as the food name, brand, serving size, and nutrient amounts, offering a flexible way to log items that may not exist in the barcode database. Nutrient inputs include calories, fat, sugar, sodium, protein, fiber, calcium, and iron, all of which are normalized and validated before being recorded. The layout also features a fixed footer with helper text indicating required fields and an "Add to Daily Intake" button.

A ScrollView and KeyboardAvoidingView are used to ensure input fields remain accessible when the keyboard is active. Each input is controlled by state, with helper functions to update values and convert inputs to numeric types. Users must provide a name for the food item, and the screen provides haptic feedback and alert messages to guide data entry and confirm successful logging. Once submitted, the food entry is added to the daily intake via the logFoodEntry function.

The Edit Daily Goals screen follows a similar layout and input structure but serves a different purpose, allowing users to set or modify their daily nutrient targets rather than log individual food items. By reusing consistent input patterns and UI components, both screens offer intuitive and interactive experiences tailored to their respective tasks.

Navigation

Navigation in this app is done using two ways, the main navigation is handled using the tab navigator to access the 5 main screens while a stack navigator is used to manage the navigation flow between related screens within each main tab of the app. This setup keeps navigation organized, makes deep linking possible, and allows each section to handle its own navigation logic independently.

Tab Navigator

The main navigation of the app is managed by a bottom tab navigator, which allows users to quickly switch between the primary screens including Home, Explore, Scan, History, and Profile. This setup is implemented using the React Navigation library's bottom tab navigator, ensuring a familiar and user-friendly navigation experience. Each tab is connected to a stack navigator or individual screen that manages the navigation flow within its own section, keeping each area of the app organized and easy to access.

Each tab uses a custom icon from the Ionicons library for clear and familiar navigation. The Scan tab features a custom floating action button placed at the center of the tab bar, giving users quick access to the barcode scanning feature from anywhere in the app.

Stack Navigator

The navigation of each main page, screen, or section of the app is managed by its own stack navigator. This approach keeps navigation organized within each tab, allowing users to move smoothly between related screens while maintaining navigation history and a consistent user experience.

- HomeStack manages the Home section and includes the main dashboard, daily goal editing, and recipe detail screens.
- ExploreStack handles the Explore section, allowing users to browse recipes and view recipe details.
- ScanStack manages screens related to barcode scanning and manual food entry. The main Scan screen lets users scan barcodes, while the Food Entry screen is presented as a modal with a close button for easy navigation.
- ProfileStack oversees the Profile section, providing access to the profile page, editing profile information, viewing bookmarks, editing goals, viewing recipe details, and accessing the about page.

All stacks are built using the native stack navigator from React Navigation and follow a consistent pattern, making it easy to maintain and extend navigation for each section. Modal presentations and custom header actions, such as close buttons, are used where appropriate to enhance usability and focus.

Storage

All data in this app is managed with AsyncStorage, which saves information directly on the device. This ensures that a user's data remains available even after closing or restarting the app. The storage folder is responsible for handling all persistent data. It saves and retrieves user profiles, daily nutrition logs, nutrition goals, bookmarks, and cached recipes. By storing this information locally, the app makes sure user data stays accessible even when offline. The storage folder contains five files. Each file is dedicated to a specific type of data, which are bookmarksStorage.js, goalsStorage.js, nutritionStorage.js, profileStorage.js, and recipesStorage.js.

Bookmarks Data

All bookmarks data are managed by the file bookmarksStorage.js, which contains all the functions related to saving, loading, and updating the user's bookmarked recipes. It uses the storage key 'user_bookmarks' to organize all bookmarked recipes in one place. The 'getBookmarks' function retrieves the current list of bookmarked recipes and returns an array of recipe objects. The 'setBookmarks' function, meanwhile, saves the entire list of bookmarked recipes by updating local storage with a new array, replacing any previous data.

```
const bookmark_key = 'user_bookmarks';

// Get all bookmarked recipes (returns an array)
Tabnine | Edit | Test | Explain | Document
export async function getBookmarks() {
  const raw = await AsyncStorage.getItem(bookmark_key);
  return raw ? JSON.parse(raw) : [];
}

// Save the full list of bookmarks
Tabnine | Edit | Test | Explain | Document
export async function setBookmarks(list) {
  await AsyncStorage.setItem(bookmark_key, JSON.stringify(list));
}
```

The 'addBookmark' function allows users to add a new recipe to their bookmarks. Before adding, it checks if the recipe already exists in the list to prevent duplicates. If the recipe is not already bookmarked, it adds it to the array and updates storage accordingly. Furthermore, the 'removeBookmark' function deletes a specific recipe from the bookmarks using its unique recipe ID. It filters out the selected recipe from the list and updates storage so the change is saved.

```
// Add a recipe to bookmarks
Tabnine | Edit | Test | Explain | Document
export async function addBookmark(recipe) {
  const bookmarks = await getBookmarks();
  if (!bookmarks.some(r => r.id === recipe.id)) {
    bookmarks.push(recipe);
    await setBookmarks(bookmarks);
  }
}

// Remove a recipe from bookmarks by id
Tabnine | Edit | Test | Explain | Document
export async function removeBookmark(recipeId) {
  let bookmarks = await getBookmarks();
  bookmarks = bookmarks.filter(r => r.id !== recipeId);
  await setBookmarks(bookmarks);
}
```

Goals Data

The goalsStorage.js file manages all operations related to user nutrition goals. All goals data is organized under the storage key ‘user_goals’ to keep each user’s preferences together and separate from other information.

The ‘getUserGoals’ function retrieves the user’s saved nutrition goals from storage and returns them as an object. If no custom goals are found, it returns null so that the app uses the default recommended goals instead. The ‘setUserGoals’ function saves a new set of nutrition goals by updating the storage with the latest values provided by the user. In addition, the ‘resetUserGoals’ function removes any saved goals from storage, allowing the app to return to the default goals.

```
const goals_key = 'user_goals';

// Get saved goals (returns object or null if not set)
Tabnine | Edit | Test | Explain | Document
export async function getUserGoals() {
  try {
    const raw = await AsyncStorage.getItem(goals_key);
    return raw ? JSON.parse(raw) : null; // null means use app default goals
  } catch {
    return null;
  }
}

// Save user goals object
Tabnine | Edit | Test | Explain | Document
export async function setUserGoals(goalsObj) {
  await AsyncStorage.setItem(goals_key, JSON.stringify(goalsObj));
}

// Remove all saved goals (resets to default)
Tabnine | Edit | Test | Explain | Document
export async function resetUserGoals() {
  await AsyncStorage.removeItem(goals_key);
}
```

Nutrition Data

The nutritionStorage.js file is responsible for managing all food and nutrition entry data in the app. Each day's entries are saved under a unique storage key that begins with 'user_dailyintake:' followed by the date. This structure allows the app to organize daily logs individually and access any day's records when needed.

```
// Build the storage key for a given date
const getKeyForDate = (dateStr) => `user_dailyintake:${dateStr}`;

// Get the storage key for today
const getTodayKey = () => getKeyForDate(toDateStr());
```

The 'logFoodEntry' function adds a new food or snack entry to the current day's log. Each entry includes details such as the name, brand, serving size, nutrients, the source of the entry, which may be scanning, manual input, or a recipe, and the time it was recorded. The 'getDailyIntake' function calculates the total amount consumed for each nutrient by summing the values from all food entries logged for the current day. Additionally, the 'getIntakeByDate' function retrieves all food entries for a specific

date, allowing users to review their nutrition intake for any day, which is then displayed by the nutrition rings in the app.

```
// Add a new food entry to today's log
Tabnine | Edit | Test | Explain | Document
export async function logFoodEntry({
  name,
  brand,
  serving,
  nutrients = {},
  source = 'manual',
  date = new Date(),
}) {
  try {
    const key = getKeyForDate(toDateStr(date));
    const existing = await AsyncStorage.getItem(key);
    const current = Array.isArray(JSON.parse(existing)) ? JSON.parse(existing) : [];

    const now = new Date();

    // Create food entry object
    const entry = {
      name,
      brand,
      serving,
      source,
      timestamp: now.toISOString(),
      date: toDateStr(date),
      nutrients,
    };

    current.push(entry);
    await AsyncStorage.setItem(key, JSON.stringify(current));
  } catch (e) {
    console.error('Error logging food entry', e);
  }
}
```

The ‘resetDailyIntake’ function removes all food entries for the current day, allowing the user to clear and restart their log if needed. This function is mainly a helper to make development and testing easier. Lastly, the ‘getAllDailyIntakes’ function collects all daily logs saved in storage and presents them in a format that can be easily grouped and displayed on the history page. This approach ensures that all nutrition data is organized by date, can be tracked over time, and remains available to the user even when offline.

Profile Data

All data operations related to the user's profile, including saving and loading profile information, are managed by the profileStorage.js file. Profile data such as the user's name, avatar, and bio is stored under the storage key 'user_profile', which provides a central and organized location for all profile-related information.

The 'getProfile' function retrieves the saved user profile from storage. If no profile exists yet, it creates a default profile with a placeholder name, a default bio, and no avatar, then saves this default to storage. The 'setProfile' function saves a complete user profile and replaces any existing profile information. Meanwhile, the 'updateProfile' function updates specific fields in the profile by merging any new changes with the current saved data, allowing for partial updates without overwriting the whole profile.

```
const profile_key = 'user_profile';

// Get saved profile, or return default if not set
Tabnine | Edit | Test | Explain | Document
export async function getProfile() {
  let json = await AsyncStorage.getItem(profile_key);
  if (json) return JSON.parse(json);

  // Default profile if none found
  const profile = {
    name: 'User',
    avatar: null,
    bio: 'Making every bite count',
  };
  await AsyncStorage.setItem(profile_key, JSON.stringify(profile));
  return profile;
}

// Save the full profile (replaces any existing profile)
Tabnine | Edit | Test | Explain | Document
export async function setProfile(profile) {
  return AsyncStorage.setItem(profile_key, JSON.stringify(profile));
}

// Update profile (merge fields)
Tabnine | Edit | Test | Explain | Document
export async function updateProfile(updates) {
  const profile = await getProfile();
  const merged = { ...profile, ...updates };
  await setProfile(merged);
  return merged;
}
```

Recipes Data

All operations related to fetching, storing, and caching recipe data and recipe categories in the app are managed by the recipesStorage.js file. This file defines several recipe filter categories that are used to fetch data from the Spoonacular API, including options like high protein, low calorie, vegan, and gluten-free. These categories appear on the Explore screen to help users easily find recipes that match their dietary preferences.

```
// Recipe filter categories for explore screen
export const recipeCategories = [
  { key: 'high_protein', label: 'High Protein', query: 'minProtein=20' },
  { key: 'low_cal', label: 'Low Calorie', query: 'maxCalories=400' },
  { key: 'low_carb', label: 'Low Carb', query: 'maxCarbs=20' },
  { key: 'high_fiber', label: 'High Fiber', query: 'minFiber=5' },
  { key: 'vegan', label: 'Vegan', query: 'diet=vegan' },
  { key: 'gluten_free', label: 'Gluten-Free', query: 'glutenFree=true' },
  { key: 'dairy_free', label: 'Dairy-Free', query: 'dairyFree=true' },
  { key: 'under_15_min', label: 'Under 15 Min', query: 'maxReadyTime=15' },
];
```

The ‘getRecipes’ function fetches a list of recipes from the Spoonacular API based on the selected category. Before making an API call, it checks if a cached version of the recipes already exists and is less than 24 hours old. If valid cached data is available, it returns the cached recipes immediately, which reduces repeated network requests and improves the user experience. If no fresh cache is found, the function fetches new data from the API, saves it in storage with a timestamp, and returns the updated list of recipes.

```
// Fetch recipes by category (with caching for 24h)
Tabnine | Edit | Test | Explain | Document
export async function getRecipes(categoryKey = null) {
  const now = Date.now();
  const storageKey = `explore_recipes_${categoryKey || 'default'}`;
  const category = recipeCategories.find(cat => cat.key === categoryKey);
  const query = category ? category.query : 'diet=healthy';

  // Try to load from cache first
  const cached = await AsyncStorage.getItem(storageKey);
  if (cached) {
    const { timestamp, data } = JSON.parse(cached);
    if (now - timestamp < 24 * 60 * 60 * 1000 && Array.isArray(data) && data.length > 0) {
      return data;
    }
  }

  // Fetch from API if no valid cache
  const url = `https://api.spoonacular.com/recipes/complexSearch?number=20&addRecipeInformation=true&includeNutrition=true&${query}&apiKey=${api_key}`;
  const res = await fetch(url);
  const data = await res.json();
  const results = Array.isArray(data) ? data : data.results || [];
  await AsyncStorage.setItem(storageKey, JSON.stringify({ timestamp: now, data: results }));
  return results;
}
```

The ‘getRecipeDetails’ function retrieves detailed information for a specific recipe using its unique ID. Similar to ‘getRecipes’, it first checks for a recently cached version before fetching from the API. This approach helps keep the app responsive and reduces data usage. The ‘clearAllRecipeCaches’ function removes all cached

recipe lists and details from storage which is mainly used during development and testing to clear outdated data or force a refresh when needed.

```
// Fetch full recipe details by id (with caching for 24h)
Tabnine | Edit | Test | Explain | Document
export async function getRecipeDetails(recipeId) {
  const detailKey = `explore_recipe_detail_${recipeId}`;
  const now = Date.now();
  const cachedDetail = await AsyncStorage.getItem(detailKey);
  if (cachedDetail) {
    const { timestamp, data } = JSON.parse(cachedDetail);
    if (now - timestamp < 24 * 60 * 60 * 1000) return data;
  }
  const url = `https://api.spoonacular.com/recipes/${recipeId}/information?includeNutrition=true&apiKey=${api_key}`;
  const res = await fetch(url);
  const data = await res.json();
  await AsyncStorage.setItem(detailKey, JSON.stringify({ timestamp: now, data }));
  return data;
}

// Remove all cached recipes and details
Tabnine | Edit | Test | Explain | Document
export async function clearAllRecipeCaches() {
  const keys = await AsyncStorage.getAllKeys();
  const exploreKeys = keys.filter(k => k.startsWith('explore_recipes_') || k.startsWith('explore_recipe_detail_'));
  if (exploreKeys.length > 0) await AsyncStorage.multiRemove(exploreKeys);
}
```

Utils

This folder holds files that contain helper functions and default values for the app, such as default nutrition goals, daily values, normalization functions for data from different sources, and calculations for nutrient amounts per serving.

Date

The date.js file contains a set of utility functions that support date manipulation and formatting throughout the app. These helpers are essential for features like tracking daily nutrition logs, organizing historical data, and improving the user interface with friendly date displays.

The ‘addDays' function takes a date and a number of days, then returns a new date that is that many days in the future or past. This is used for calendar navigation, allowing users to move between previous or upcoming days in their calendar. The ‘startOfWeek’ function returns the date for the Monday of the week for any given date, helping to standardize weekly views and ensuring that the week selector always begins on Monday, regardless of which day the user selects.

The ‘toDateStr' function formats a JavaScript Date object as a "YYYY-MM-DD" string. This format is used as a storage key for daily data and provides a consistent way to save and retrieve entries across the app. The ‘formatFriendlyDate’ function takes a date string in the "YYYY-MM-DD" format and converts it into a more user-friendly label, such as “Today” for the current date and “Yesterday” for the

previous day. For other dates, it displays a short, readable string with the weekday, month, and day. This makes date information easier to understand in features like the nutrition history log.

Nutrition

The nutrition.js file provides constants, helper functions, and normalization logic for handling nutrition data throughout the app. This file is essential for ensuring that all nutrition-related values are consistent, accurate, and easy to use across different features.

It contains the default daily nutrition goals for key nutrients such as proteins, fiber, calcium, iron, sugars, fat, sodium, and calories. These values are based on the recommended daily amounts published by the FDA (U.S. Food and Drug Administration, 2022) and follow the official guidelines for nutrition labeling, as provided on the FDA website. They serve as the app's initial targets for users but can be customized to fit individual needs. The file also includes nutrient definitions, with display names, measurement units, and the color used for each nutrient in charts and progress rings.

Standard daily values for nutrition labels are included to enable the calculation of percent daily value, making it easy to show users how much of each nutrient they have consumed compared to recommended amounts, which are also based on FDA guidelines. The 'pct' function handles the percentage calculation and there are also utility functions for converting between different units, such as grams, milligrams, and micrograms.

A core part of this file is its normalization functions. These functions extract and standardize nutrient data from various sources so the app can handle data from different APIs and user inputs in a unified format. The 'getPerServing' helper is used within this process to determine the amount of each nutrient per serving. If a direct per-serving value is not available, it calculates the amount based on the nutrient value per 100 grams and the serving size. The 'normalizeNutrientsFromOpenFoodFacts' function parses nutrition data from OpenFoodFacts API products, while 'normalizeNutrientsFromManual' processes values entered manually by users. The 'normalizeNutrientsFromSpoonacular' function standardizes nutrition information fetched from Spoonacular recipes. This approach ensures that all food entries, regardless of their origin, are represented in a consistent structure for display and tracking.

APIs

Nouri integrates external APIs to enhance functionality, particularly for recipe exploration and barcode-based food logging. The primary APIs used in this project are the Spoonacular API and the OpenFoodFacts API.

The Spoonacular API is used to fetch detailed recipes, including nutritional information, ingredients, preparation time, and dietary tags such as vegan, gluten-free, and dairy-free. This API enables the Explore screen to provide users with a wide selection of recipes that can be filtered based on categories and dietary preferences. Recipe data is normalized within the app to ensure consistent display across different screens.

The OpenFoodFacts API is used for barcode scanning. When a user scans a product, the app retrieves product details including serving size, brand, and nutrient content. This allows users to quickly log foods without manual entry and provides immediate nutritional feedback. The nutrient data is normalized using utility functions so that it can be added accurately to the user's daily intake.

Limitations of the API usage include:

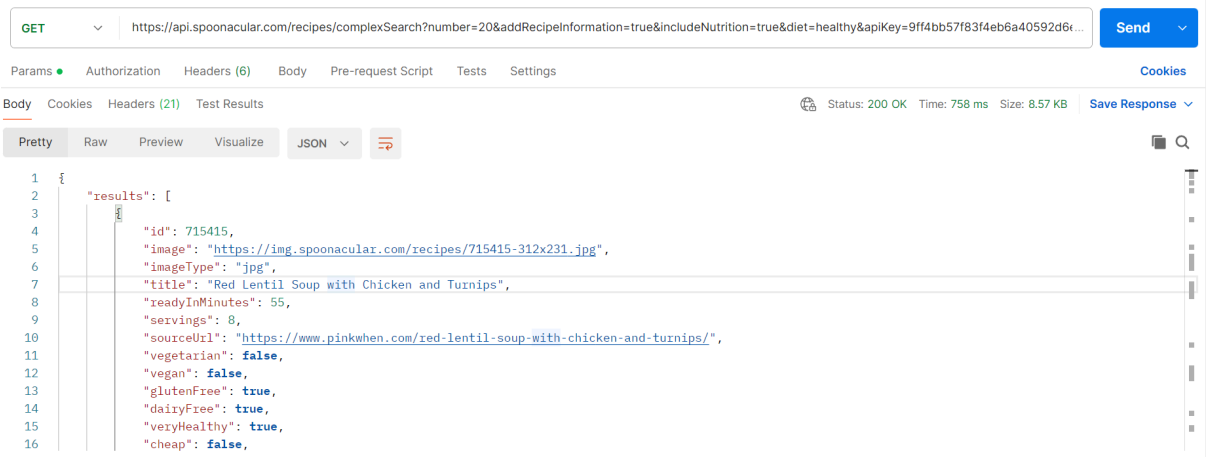
- The Spoonacular free tier limits the number of requests per day, so recipe data is cached locally and refreshed only once every 24 hours in this app, which reduces real-time updates and limits dynamic exploration of new recipes.
- The OpenFoodFacts API relies on user-contributed data, which can sometimes be incomplete or inconsistent, affecting the accuracy of nutritional information.
- Both APIs require an internet connection for data retrieval, which limits the app's offline functionality.
- Features like personalized recommendations and filtering are constrained by the available endpoints and data provided by the APIs.

Despite these limitations, using these APIs greatly improves the app's usability and provides a complete nutrition tracking experience. Future improvements could include upgrading to paid tiers, combining multiple data sources, or adding a backend to manage cached data, which would improve responsiveness and expand functionality.

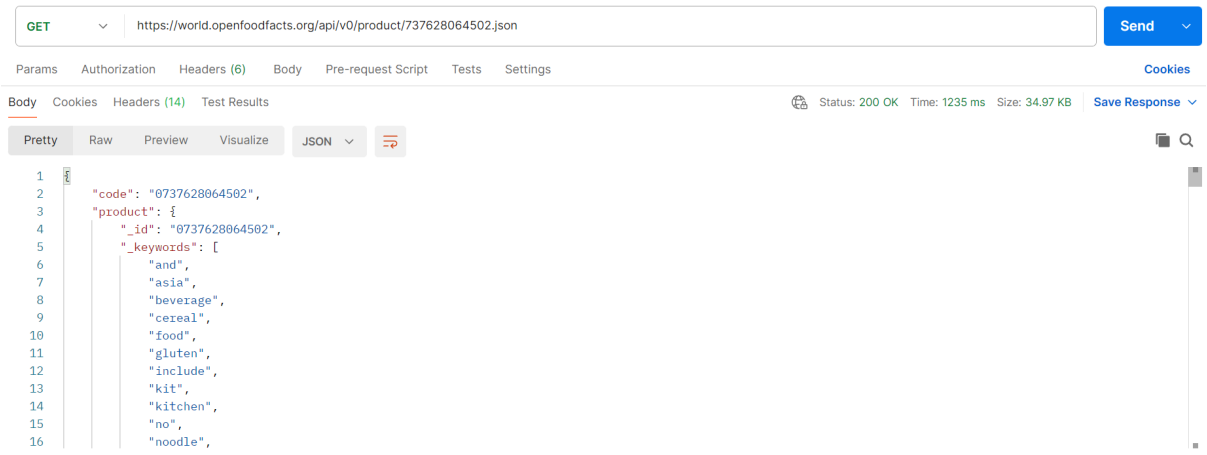
Testing

API Testing

To verify the reliability of external data sources, GET requests were tested for both the Spoonacular and OpenFoodFacts APIs. For Spoonacular, recipe searches were executed to ensure that responses included accurate recipe IDs, images, titles, preparation times, servings, dietary tags, and nutrition information.



For OpenFoodFacts, barcode lookups were tested to confirm that product details, nutrient content, and serving information were correctly retrieved. These tests also verified that the app could handle edge cases such as missing fields or invalid product IDs. The caching mechanism was checked to ensure that API responses were stored locally and refreshed once every 24 hours, maintaining functionality even under the free-tier request limits. The API testing confirmed that data could be consistently normalized and integrated into components such as RecipeCard, NutritionCard, and the daily intake logs.



Unit Testing

During development, I planned to implement unit testing using the Jest framework to verify that key functions, such as nutrient logging, profile management, and storage operations, were working as intended. Unit testing would have helped ensure reliability, catch bugs early, and improve maintainability of the code.

However, due to compatibility issues with the Expo environment and the asynchronous nature of certain functions such as AsyncStorage calls, I was unable to successfully execute Jest tests during this project. Despite this, I manually verified the functionality of critical components through careful debugging and iterative testing within the app itself.

For future work, unit testing can be implemented after resolving these environment constraints, potentially by mocking asynchronous storage and API calls, to provide automated verification of all key app functions.

Evaluation

Challenges and Reflection

During the development of Nouri, several technical challenges were encountered. Integrating multiple APIs proved complex because each source represented nutrient data differently. For example, the Spoonacular API and OpenFoodFacts API used different naming conventions and measurement units for nutrients, requiring careful normalization to ensure consistency across the app. Additionally, some APIs provided incomplete data, such as missing nutrient values or serving sizes, which required implementing fallback logic and validation to maintain accuracy in the daily intake calculations.

Managing offline functionality also presented challenges, as recipe browsing and barcode scanning require an internet connection, while daily intake data must remain accessible offline. Achieving a responsive and visually consistent interface across different device sizes, handling asynchronous data fetching, and ensuring reliable notifications further added to the project's complexity. These challenges required iterative testing, careful data handling, and thoughtful UI design to create a seamless and accurate user experience.

Limitations and Future Improvements

Nouri demonstrates its core functionality effectively, but there are some limitations. Using the free tier of the Spoonacular API restricts requests, so recipe data is cached and refreshed only once every 24 hours, limiting real-time updates. Features like recipe browsing and barcode scanning require an internet connection, reducing offline usability. The app currently lacks advanced personalization, and the user interface could be improved for better accessibility and navigation. Additionally, all user data is stored locally, which limits synchronization across devices and does not provide robust backup or security.

Future improvements could include upgrading to a paid API or adding multiple data sources, implementing a backend for user data and profiles to better manage storage and synchronization, adding a search function for the Explore screen, and enabling more personalization through integration with third-party apps. Additional enhancements such as push notifications for daily goals, improved UI design, accessibility, tutorials, and richer data visualizations would further increase the app's usability and engagement.

Conclusion

This project demonstrates the development of a mobile nutrition tracking app designed to make logging meals and monitoring nutrient intake simple and engaging. Nouri combines barcode scanning, manual food entry, and recipe exploration to offer users a flexible and comprehensive tool for managing daily nutrition. Through iterative design, including low, mid, and high-fidelity prototypes, the app was refined based on user feedback to ensure intuitive navigation, clear visual design, and effective data presentation. Key features such as NutritionRings, the skeuomorphic NutritionLabel, and personalized daily goals enhance usability and engagement. While limitations remain, including API request constraints and offline functionality, Nouri shows potential for expansion through backend integration, improved personalization, and enhanced visual design. Overall, it provides a user-friendly platform that supports healthier eating habits and encourages consistent nutrition tracking.

References

1. U.S. Food and Drug Administration (FDA), 2022. Daily Value on the Nutrition and Supplement Facts Labels. [online] Available at: https://www.fda.gov/food/nutrition-facts-label/daily-value-nutrition-and-supplement-facts-labels [Accessed 3 August 2025].