

# Urban Oasis App

---

DEVELOPER DOCUMENTATION

BY: FOOD APPLICATION TEAM

HANNA FUGULIN



# Table of Contents

## Contents

Table of Contents .....	2
List of Figures .....	3
Introduction.....	4
Development Environment.....	4
React Native .....	7
Emulation.....	9
Expo GO .....	10
Jest .....	13
Firebase Firestore .....	14
Firebase Cloud Storage.....	15
Google Places API.....	15
Styling .....	16
Wireframe Overview.....	19
Recipe React Pages .....	23
Database Structure.....	29
Recipe Data .....	30
Location Data .....	33
Functionality Not Implemented .....	35
By <b>Food Application Team</b> .....	39

## List of Figures

<b>Figure 1.1</b> – Android Studio AVD Setup and App Launch .....	Pg. 9
<b>Figure 1.2</b> – Expo Development Server Running with QR Code .....	Pg. 11
<b>Figure 2.1</b> – Styled Home Screen with NativeWind Components .....	Pg. 15
<b>Figure 2.2</b> – Grocery Stores Header .....	Pg. 17
<b>Figure 2.3</b> – Store List Screen with Search and Filter UI .....	Pg. 17
<b>Figure 2.4</b> – Store Detail Screen with Store Info and Firebase Integration .....	Pg. 17
<b>Figure 2.5</b> – Store Hours Detail Styling Highlights with Icons and Centered Layout.....	Pg. 18
<b>Figure 2.6</b> – Recipe List and Detail Screens with Styled Cards and Content .....	Pg. 18
<b>Figure 2.7</b> – Bottom Navigation Bar .....	Pg. 18
<b>Figure 2.8 / 2.9</b> – Recipe Styling Highlights with Skeleton Loaders and Grid Layout.....	Pg. 18
<b>Figure 3.1</b> – Wireframes: Home Screen – Figma Draft/Final Draft .....	Pg. 19
<b>Figure 3.2</b> – Wireframes: Recipe List Screen – Figma Draft/Final Draft .....	Pg. 19
<b>Figure 3.3</b> – Wireframes: Recipe Details Screen – Figma Draft/Final Draft .....	Pg. 20
<b>Figure 3.4</b> – Wireframes: Store List Screen – Figma Draft/Final Draft .....	Pg. 20
<b>Figure 3.5</b> – Wireframes: Store Detail Screen – Figma Draft/Final Draft .....	Pg. 21
<b>Figure 4.1</b> – Figma Usage Overview .....	Pg. 22
<b>Figure 5.1</b> – Firebase Realtime Database: Location Node Structure .....	Pg. 29
<b>Figure 5.2</b> – Firebase Realtime Database: Recipes Node Structure .....	Pg. 29
<b>Figure 5.3</b> – Entity-Relationship Diagram (ERD) of Firebase Data Model .....	Pg. 30
<b>Figure 5.4</b> – Firebase Recipe Object Mapping in Code with Nutritional Fields .....	Pg. 31
<b>Figure 5.5</b> – recipeDetails Code Mapping from Firebase with Nutritional Fields .....	Pg. 32
<b>Figure 5.6</b> – Firebase Location Data Retrieval and Mapping Function .....	Pg. 34

## Introduction

This document provides a comprehensive overview of the Urban Oasis application, developed as part of our group project. The application aims to provide users with a user-friendly platform for discovering and managing recipes, as well as locating nearby grocery stores. Throughout this document, we detail the design, development, and testing processes that contributed to the successful creation of Urban Oasis, highlighting the tools and technologies used, including React Native, Firebase, and various third-party integrations.

We begin by outlining the development environment and the specific configuration settings that ensured a smooth collaboration between team members. The document further explores key components of the application, from the user interface to the backend infrastructure, providing insights into how Firebase and the Google Places API were integrated to provide real-time data. Additionally, we discuss the tools employed for testing, including Jest and Expo Go, and review the challenges faced during cross-platform development.

Beyond discussing the current features of the application, we also look into planned future improvements and enhancements to better serve our users. This document serves as a guide to understanding the design decisions, technical approach, and future roadmap of the Urban Oasis project.

## Development Environment

Visual Studio Code is used as the primary integrated development environment (IDE). Visual Studio Code is known to be lightweight, fast, and highly customizable. It was developed by Microsoft, designed specifically for modern JavaScript/TypeScript development. It works exceptionally well for React, React Native, and Node.js.

Below are the **extensions** used for Urban Oasis:

Extension	Use
ESLint	Highlights JavaScript/React linting issues in real time
Prettier- Code Formatter	Ensures consistent code formatting based on rules in .prettierrc
Tailwind CSS Intellisense (NativeWind)	Provides class suggestions and hover previews for Tailwind CSS (NativeWind)
React Native Tools	Adds debugging and simulator tools for React Native

**Settings** used for Urban Oasis in the settings.json file:

```
{
  "workbench.colorTheme": "Monokai++",
  "eslint.codeActionsOnSave.rules": null,
  "editor.wordWrap": "on",
  "git.autofetch": true,
  "git.enableSmartCommit": true,
  "liveServer.settings.donotVerifyTags": true,
  "[html]": {
    "editor.defaultFormatter": "esbenp.prettier-vscode"
  },
  "editor.defaultFormatter": "esbenp.prettier-vscode",
  "css.format.spaceAroundSelectorSeparator": true,
  "less.format.spaceAroundSelectorSeparator": true,
  "scss.format.spaceAroundSelectorSeparator": true,
  "prettier.bracketSameLine": true,
  "editor.detectIndentation": false,
  "prettier.printWidth": 100,
  "prettier.tabWidth": 4,
  "html.format.wrapLineLength": 100,
  "workbench.iconTheme": "material-icon-theme",
  "code-runner.clearPreviousOutput": true,
  "editor.formatOnSave": true,
  "prettier.trailingComma": "none",
  "editor.renderWhitespace": "all",
  "eslint.options": {},
  "eslint.lintTask.enable": true,
  "emmet.showSuggestionsAsSnippets": true,
  "debug.console.clearBeforeReusing": true,
  "tailwindCSS.experimental.configFile": null,
  "colorHelper.formatsOrder": [],
  "vscode-color-picker.languages": [
    "php",
    "html",
    "css",
    "python",
    "jsonc",
    "javascript",
    "javascriptreact",
    "typescript",
    "typescriptreact",
    "vue"
  ],
  "github.copilot.chat.reviewSelection.instructions": [],
  "github.copilot.advanced": {},
  "workbench.editor.enablePreview": false,
  "github.copilot.editor.enableAutoCompletions": false,
  "editor.trimAutoWhitespace": false,
  "tailwindCSS.emmetCompletions": true
}
```

These settings ensure auto-formatting and lint fixes on save, proper file type association, and consistent quote style ( " instead of ' ). The settings are applied globally and there is zero-configuration for collaborators wanting to contribute to the project through our GitHub repository.

### Environment Configuration

All environment variables are stored in a .env file at the root level of the project. This ensures that credentials are stored away from a public facing repository. The .env file is shared amongst developers actively contributing to the project.

A call from the data.js file is made to the .env file in a generic object. The .env file then provides the actual credentials to the firebase database upon project execution. The generic object is displayed below:

```
const firebaseConfig = {
  apiKey: API_KEY,
  authDomain: AUTH_DOMAIN,
  databaseURL: DATABASE_URL,
  projectId: PROJECT_ID,
  storageBucket: STORAGE_BUCKET,
  messagingSenderId: MESSAGING_SENDER_ID,
  appId: APP_ID,
  measurementId: MEASUREMENT_ID
};
```

An import is used to specify that the object will be calling from a .env file. This import is displayed below:

```
import {
  API_KEY,
  AUTH_DOMAIN,
  DATABASE_URL,
  PROJECT_ID,
  STORAGE_BUCKET,
  MESSAGING_SENDER_ID,
  APP_ID,
  MEASUREMENT_ID
} from "@env";
```

## React Native

According to its website, React Native is a “best-in-class” JavaScript library used for building user interfaces bringing the React programming paradigm to both Android and iOS devices as well as web browsers. Developed in 2015 by Meta Platforms, formally Facebook Inc., React Native is almost identical to React except in the way that it handles the Document Object Management (DOM)— by running a background process on the end-device allowing communication. Although React Native’s structure and styling is similar in syntax to HTML and CSS, it does not use either. Instead, it uses JavaScript to create views along with third-party plugins. For example, in the case of the Urban Oasis application, it makes use of the open-source CSS Tailwind framework along with JavaScript created CSS StyleSheet objects. The detailed developer React Native installation details are available on the application’s GitHub in the Readme.md file.

In React Native, the component’s or tag’s functionality must first be imported from the various libraries before they are eligible to be called. These libraries include the React Native library as well as third-party libraries such as React Native Paper which allows for easy-to-use components, such as search boxes, icons, and Moti Skeleton used for showing animated loading states. These are shown in the following code excerpt:

```
import React, { useState, useEffect } from "react";
import { SafeAreaView, StyleSheet, Text, View, Image, ScrollView, } from
"react-native";
import { IconButton } from "react-native-paper";
import { Skeleton } from "moti/skeleton";
```

All screen pages are wrapped in a SafeAreaView component tag which allows content to be rendered within the “safe area” boundaries of a device. This considers a device’s physical limitations, such as rounded screen corners, camera notches, and other sensors present. Each page also includes a top-navigation bar, which currently only contains a stylized image of the app name, and a bottom-navigation bar containing a basket, home, and fork-and-knife icons. Other tags used are typical of any application, including the ScrollView, View, Image, and Pressable tags. Application formatting is handled by a combination of Tailwind and/or JavaScript CSS StyleSheet objects. In either case, the formatting statements are applied inline to the associated component as shown in the following code excerpt:

```
// Recipe List Screen
export const RecipeListScreen = ({ navigation }) => {
  ...
  return (
    <SafeAreaView style={styles.container}>
      { /* Top Navigation Bar */ }
      <View style={styles.topBar}>
        <Image
          source={require("../assets/urban-oasis-text-
only.png")}
          style={styles.logo}
        />
```

```

</View>
{/* Header Section */}
<View style={styles.headerRow}>
  <Text style={styles.headerText}>Recipes</Text>
</View>
{/* Recipe Cards*/}
<ScrollView>
  loading ? Array.from({ length: 5 }).map((_, index) => (
    ...
    recipeList.map((item) => (
      <RecipeCard key={item.id} item={item}
navigation={navigation} />
    ))
  )
</ScrollView>
{/* Bottom Navigation */}
<View style={styles.bottomBar}>
  <IconButton
    icon="basket-outline"
    size={45}
    iconColor="white"
    onPress={() => navigation.navigate("StoresListScreen")}
  />
  <IconButton
    icon="home-outline"
    size={45}
    iconColor="white"
    onPress={() => navigation.navigate("Home")}
  />
  <IconButton
    icon="silverware-fork-knife"
    size={45}
    iconColor="#BCEDC3"
    onPress={() => navigation.navigate("RecipeList")}
  />
</View>
</SafeAreaView>
);});
// Styles
const styles = StyleSheet.create({
  container: {
    flex: 1,
    marginTop: StatusBar.currentHeight
  },
  topBar: {
    flexDirection: "row",
    justifyContent: "center",
    alignItems: "center",
    paddingVertical: 10,
    backgroundColor: "#7FA184",
    borderBottomWidth: 2,
    borderBottomColor: "#5E7147"
  },
  ...
});

```



# Emulation

## What is Android Studio?

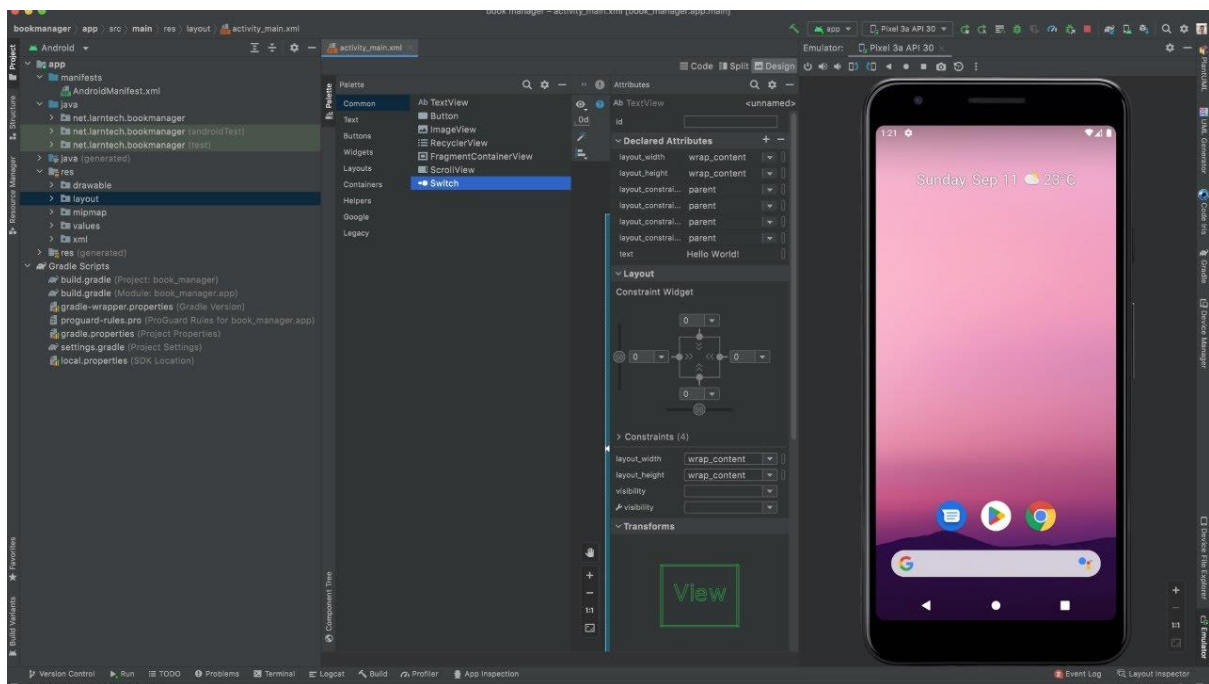
To test our mobile app without needing a physical Android device, we used Android Studio, Google's official IDE for Android development. It includes a built-in emulator that simulates different Android phones and operating systems, making it ideal for testing React Native apps like Urban Oasis.

## How to Set Up Android Studio and Run the App

After downloading Android Studio, we set up an Android Virtual Device (AVD) using the AVD Manager. This allowed us to run the app on a simulated phone directly from our computers.

Here are the steps on how to set it up:

1. Download and install it from [developer.android.com/studio](https://developer.android.com/studio).
2. Open Android Studio
3. Go to **Tools > SDK Manager** and install the latest SDK tools
4. Open **Tools > AVD Manager**, create a new emulator (e.g., Pixel 4 or 6 with Android 12+)
5. Start the emulator from the AVD Manager
6. In your terminal, run "expo start".
7. Press "a" to launch the app in the Android emulator.



## iOS Emulation and Challenges

For Apple devices, we initially tried using Xcode's iOS Simulator, but it caused several issues during setup and integration. As a result, we switched to Visual Studio (with the required iOS setup on Mac systems) for testing the app on Apple devices. This approach allowed us to confirm cross-platform functionality without fully relying on Xcode's simulator.

## Why Android Studio Was Preferred

Android Studio offered better cross-platform compatibility, easier setup, and smoother integration with React Native, which made it a reliable tool for development and testing, making it our preferred choice for consistent testing across team members' systems.

## Expo GO

### What is Expo Go?

Expo Go is a mobile application that allows developers to test and preview React Native apps on physical devices without compiling native code. It works with Expo CLI and the Metro Bundler, enabling fast development, live updates, and real-time testing across iOS and Android platforms. It simplifies early-stage development by eliminating the need for native builds, emulators, or simulators.

### How to Install Expo Go on a Mobile Device

#### Option 1: Use an Actual Android or iOS Device (Recommended)

##### Step 1: Download Expo Go on your phone

- Android users: Expo Go on Google Play
- iOS users: Expo Go in the App Store

##### Step 2: Start the Expo development server


In your project directory, run either of the following commands in the terminal:

```
npm start or expo start
```

This will launch the Metro Bundler in your terminal or browser and display a QR code.

```
> urban-oasis@1.0.0 start
> expo start

env: load .env
env: export GOOGLE_PLACES_API_KEY API_KEY AUTH_DOMAIN DATABASE_URL
MEASUREMENT_ID
Starting project at C:\Users\Maha\Desktop\Urban-Oasis-development
Starting Metro Bundler
The following packages should be updated for best compatibility with
expo@52.0.38 - expected version: ~52.0.46
expo-location@18.0.7 - expected version: ~18.0.10
react-native@0.76.7 - expected version: 0.76.9
Your project may not work correctly until you install the expected
```



### Step 3: Scan the QR code

- Open the Expo Go app on your device.
- Use the built-in QR scanner to scan the QR code shown in the terminal or browser.
- Your React Native app will load and run on your device.

**Note:** Your computer and mobile device must be connected to the same Wi-Fi network for this to work.

### How to Run the App Using Expo Go

Once the project is set up and the Expo development server is running (`expo start`), you can run the app using one of the following options:

- Scan QR Code on Device: Use your phone's camera or Expo Go's scanner to open the app directly.
- Run in Browser: Press `w` in the terminal to open the app in a web browser.
- Run on Android Emulator: Press `a` in the terminal to open the app on an Android emulator.
- Run on iOS Simulator (Mac only): Press `i` to launch the app in the iOS simulator.

## Option 2: Expo Go Setup for Testing Without a Physical Device

### Android Emulator (Windows & Mac)

#### Step 1: Install Android Studio

- Download and install from [developer.android.com/studio](https://developer.android.com/studio).

#### Step 2: Set Up the Emulator

- Open Android Studio.
- Go to Tools > SDK Manager and install the latest SDK tools.
- Open Tools > AVD Manager to create a new emulator (e.g., Pixel 4 or Pixel 6 with Android 12+).
- Start the emulator from the AVD Manager.

#### Step 3: Run the App

In your terminal, run `expo start`

- Press `a` to launch the app in the Android emulator.

### iOS Simulator (Mac Only)

If you are using a Mac and have Xcode installed:

- Run `expo start` in the terminal.
- Press `i` to open the iOS simulator and load the app.

Alternatively, test directly on a real iPhone using Expo Go and the QR code method described earlier, no need for Xcode builds.

## How Expo Go Was Used for Manual Testing

During the development of the application, Expo Go was used for manual testing on real devices. The process included:

- Verifying app layout and UI design across different screen sizes and platforms.
- Testing navigation, user interaction, and feature functionality.
- Identifying and fixing cross-platform issues.
- Allowing team members to test the app individually on their devices without requiring a complex setup.

## Jest

Jest is a JavaScript testing framework maintained by Meta (Facebook). It is widely used for testing JavaScript and React applications (web and mobile). It provides out-of-the-box tools that include:

- Zero-configuration setup
- Built-in assertion library
- Snapshot testing
- Mocking capabilities
- Parallel test execution

In Urban Oasis, we used Jest for unit testing critical data-fetching logic and UI behavior in components built with React Native.

Examples of what was tested:

Function / Component	What Was Tested
getLocations()	Returns mocked store location data from Firebase
StoresList component	Renders the correct store cards, filters by type, filters by open status
getRecipeList()	Returns a list of recipes from Firebase, used to test RecipeListScreen
getRecipeDetail()	Returns individual recipe data and used to verify detail view components

### Project Directory Structure (Testing Specific)

Urban Oasis

```
├── __mocks__/  
│   ├── react-native-star-svg-rating.js: Mock for external rating  
component  
│   └── react-native-svg.js: Mock for SVG support in Jest  
├── __tests__/  
│   ├── StoresList.test.js: Unit tests for store list screen  
│   ├── RecipesList.test.js: Unit tests for recipe list screen  
│   └── RecipesDetail.test.js: Unit tests for recipe detail screen
```

**Mocks:**

The `_mocks_` folder is automatically picked up by Jest and used to override imports during test execution. This is particularly helpful for libraries like `react-native-svg` that don't work out-of-the-box in testing environments.

**Tests:**

The `_tests_` folder houses all our component tests. Each test file indicates what is being tested. We separated the test files from the actual `/src` files to avoid polluting the directory with test logic.

**Running the Test Cases:**

To run the test cases, the developer will type `npm test` which will return the output of the results, such as the one below:

```
PASS  __tests__/RecipesList.test.js

✓ displays the correct recipe data (75 ms)

Test Suites: 1 passed, 1 total
Tests: 1 passed, 1 total
Snapshots: 0 total
Time: 2.34s
```

## Firestore

Firebase Firestore is a no SQL database solution provided by Google. It stores data in documents organized into collections, allowing for efficient and real-time synchronization across client applications. In our recipe application, Firestore was used to store and retrieve recipe and location details.

In order to start with Firebase, we created the project in Google's UI, then followed the instructions to create a project within the free tier.

Creating an app and obtaining an API key begins by selecting "Add app" in your Firebase project and choosing a platform, such as web, Android, or iOS. After following the prompts to register the app, Firebase generates configuration settings including the API key, `authDomain`, `databaseURL`, and `projectId`.

The following is an example of what the API key may look like:

```
const firebaseConfig = {
  apiKey: "[Your API Key]",
  authDomain: "your-app.firebaseio.com",
  projectId: "your-app-id",
  storageBucket: "your-app.appspot.com",
};
```

Security rules are set in a similar block of code, exemplified below:

```
rules_version = '2';
service cloud.firestore {
  match /databases/{database}/documents {
    match /recipes/{recipeId} {
      allow read, write: if request.auth != null;
    }
  }
}
```

And finally, to pull recipe data from Firestore into the app, the code looks as follows:

```
firebase.firestore().collection("recipes").get()
  .then(snapshot => {
    snapshot.forEach(doc => {
      console.log(doc.id, " => ", doc.data());
    });
  });
```

## Firestore Cloud Storage

Firestore Cloud Storage allows the storage and retrieval of files, and for Urban Oasis, it was utilized for images. Files stored in Firestore Storage can be accessed using URLs, with the following templated code:

```
firebase.storage().ref('recipe-images/image-
name.jpg').getDownloadURL().then(url => console.log("Image URL: ",
url));
```

It utilizes the same credentials for security principles that was outlined in the above section.

## Google Places API

Google Places API is a service used in the application, provided by Google, that returns information about places using HTTP requests. This includes real-time data such as names, addresses, photos, and geographic coordinates of millions of places worldwide.

In this application, the Google Places API was integrated to enhance store listings with formatted address information and images when users input or search for locations using either their zip, city, or state. Specifically, the app used Google's Geocoding API to convert a user-input address into latitude and longitude.

Example usage in the app:

```
const url =
`https://maps.googleapis.com/maps/api/geocode/json?address=${encodeURIComponent(
searchQuery)}&key=${API_KEY}`;
const response = await fetch(url);
const data = await response.json();
const location = data.results[0].geometry.location;
```

The API key was securely stored in a .env file shared among endpoint developers.

Example code from HomeScreen.jsx:

```
const fetchCoordinatesFromAddress = async (searchQuery) => {
  const url =
    `https://maps.googleapis.com/maps/api/geocode/json?address=${encodeURIComponent(
      searchQuery
    )}&key=${API_KEY}`;

  const response = await fetch(url);
  const data = await response.json();

  if (data.results.length > 0) {
    return {
      latitude: data.results[0].geometry.location.lat,
      longitude: data.results[0].geometry.location.lng
    };
  }
};
```

## Styling

### Home Screen

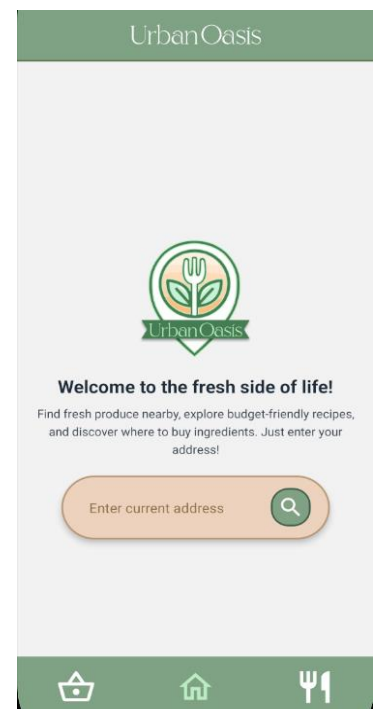
The Home Screen welcomes users to the app and helps them either enter an address or use their current location to find nearby grocery stores. The screen combines branding elements, a descriptive introduction, and a stylized search bar.

#### NativeWind Usage

- Centering elements vertically and horizontally using `className="items-center justify-center"`
- Typography: `text-2xl font-bold text-gray-800 mt-6`, `text-base text-gray-700`
- Responsive layout: `w-3/4` and `mt-6` used to adjust spacing and sizing across devices
- Rounded Search Bar: `rounded-full px-4 py-2`
- Background Color: Tailwind-style hex color inside `className` `#EDD2BD`

#### Styling Highlights

- Shadow and border layering done via StyleSheet for compatibility
- IconButton wrapped in custom border and shadow for visibility
- Mix of StyleSheet for structure + NativeWind for fine-tuning





## Store List Screen

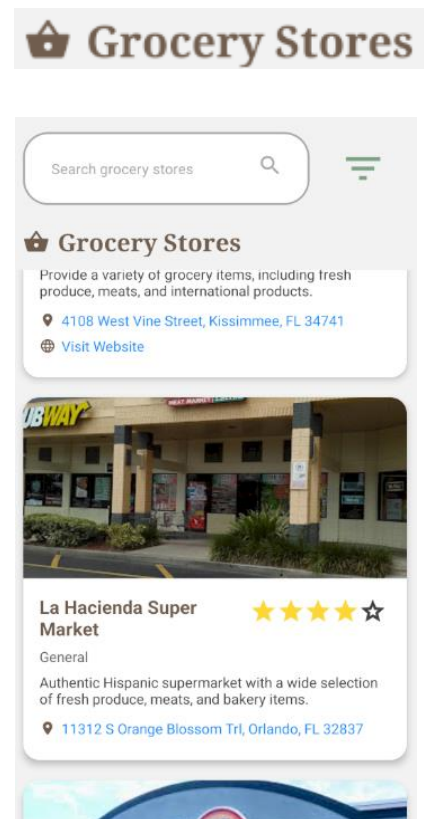
Presents a searchable and scrollable list of grocery stores using StoresList component. Includes filter and search options.

### NativeWind Usage

- className="flex-1 items-center justify-center" for container
- Text input and icon rows styled using NativeWind flex utilities
- Icon headers like the basket icon use consistent margin and color from theme

### Styling Highlights

- NativeWind used to simplify row layouts and text styling
- Filter icon styled via Tailwind and custom color
- Shadow and border-radius are retained in StyleSheet for more control

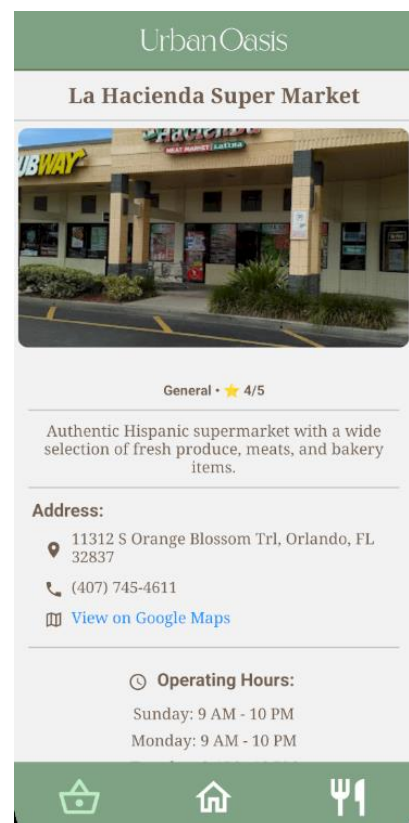


## Store Detail Screen

Displays details about a selected store, including description, address, operating hours, and contact info. Uses both Firebase and external links.

### NativeWind Usage

- NativeWind is minimal on this screen to allow greater structural control
- Alignment and layout are mostly handled via StyleSheet
- Operating hours styled with text-center manually in styles



## Styling Highlights

- Icons (map, phone, web) aligned using flex-row items-center
- All operating hours centered using textAlign: "center"
- Link colors and padding follow branding via StyleSheet

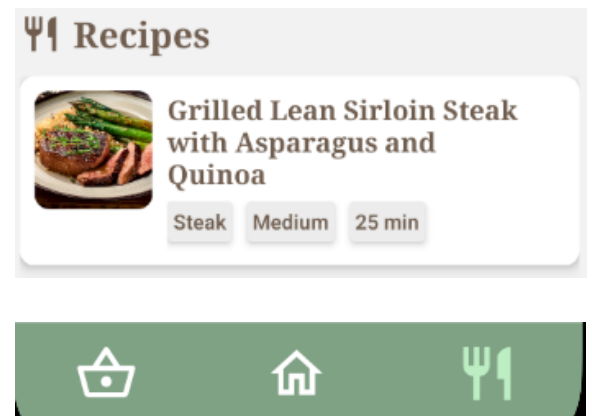


## Recipe List and Detailed Recipes

Recipes are presented in cards on the List screen, and detailed with images, time, ingredients, and steps on the Detail screen.

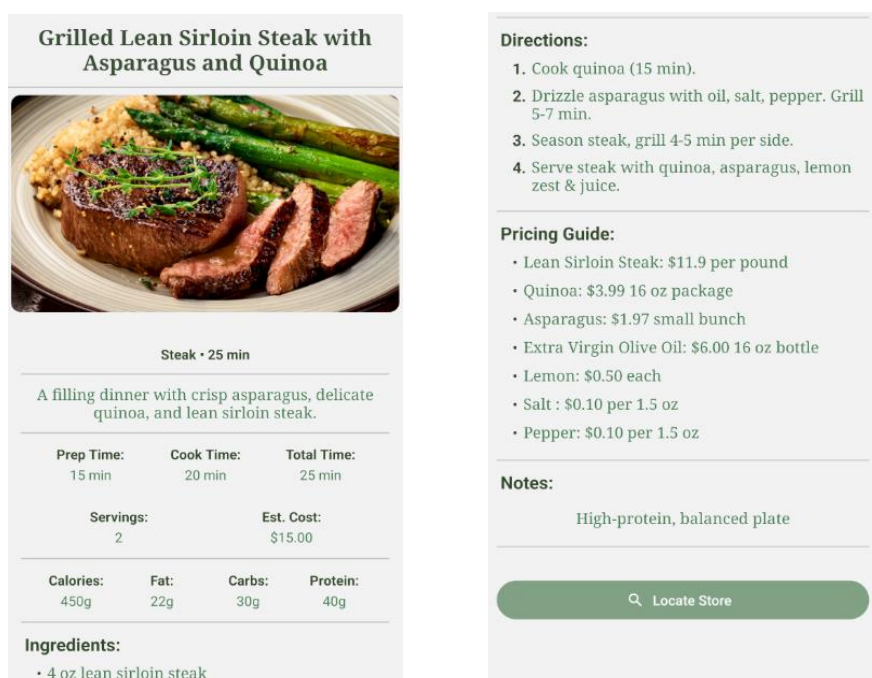
## NativeWind Usage

- Header section: flex-row items-center with fork-and-knife icon
- NativeWind used for shadow, rounded corners, and spacing in cards
- Text classes like text-base text-gray-800 and text-center font-bold



## Styling Highlights

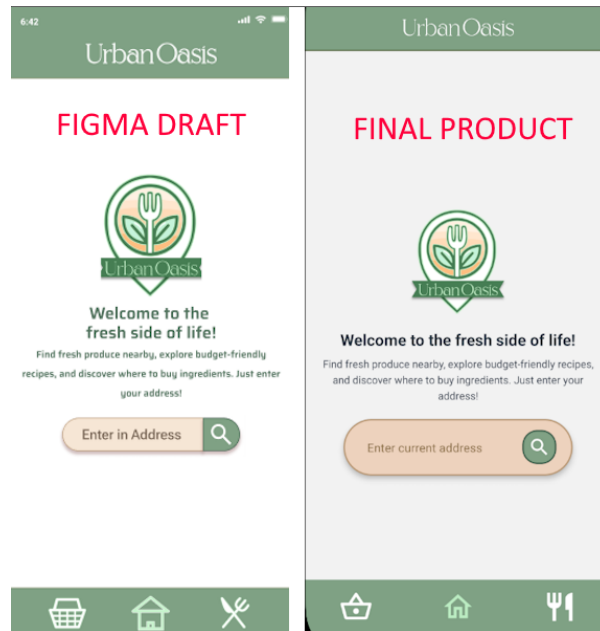
- Skeleton loaders for loading state (non-NativeWind but styled similarly)
- Buttons and icons themed via consistent iconColor
- Ingredients, directions, and pricing all structured with grid layout in StyleSheet



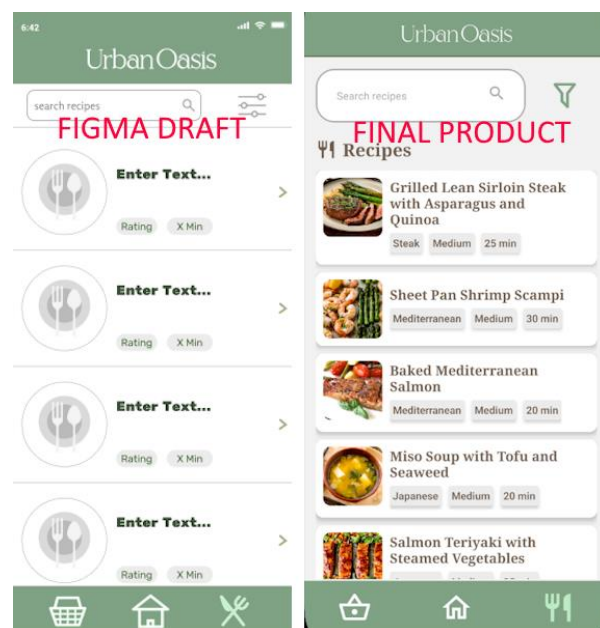
## Wireframe Overview

The wireframe was built to create a clean, user-friendly interface focused on accessibility and intuitive navigation. It consists of the following key screens:

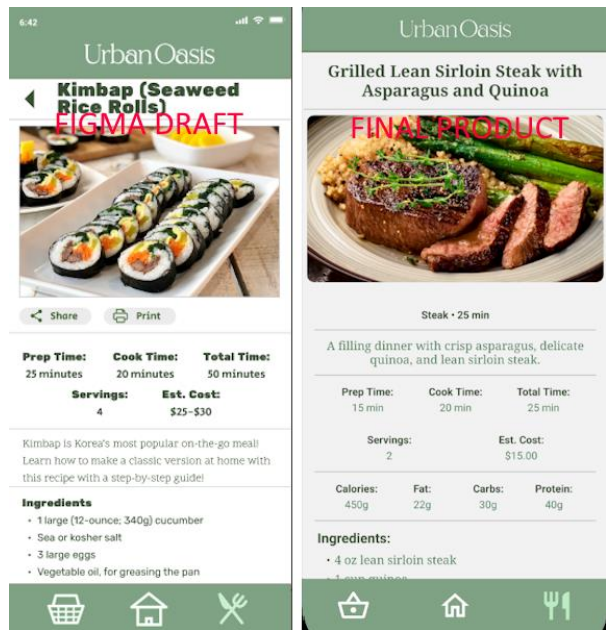
- **Home Screen** – Features brand identity, links to explore recipes and stores.



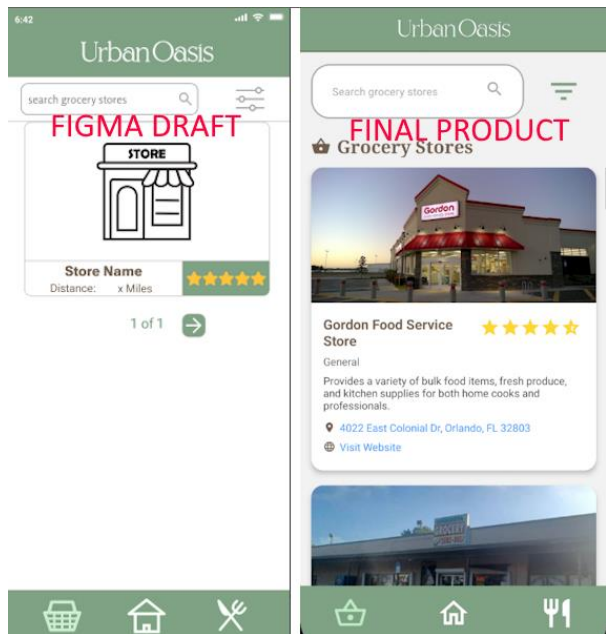
- **Recipe List Screen** – Displays a searchable list of recipes with thumbnails, titles, categories, and preparation times.



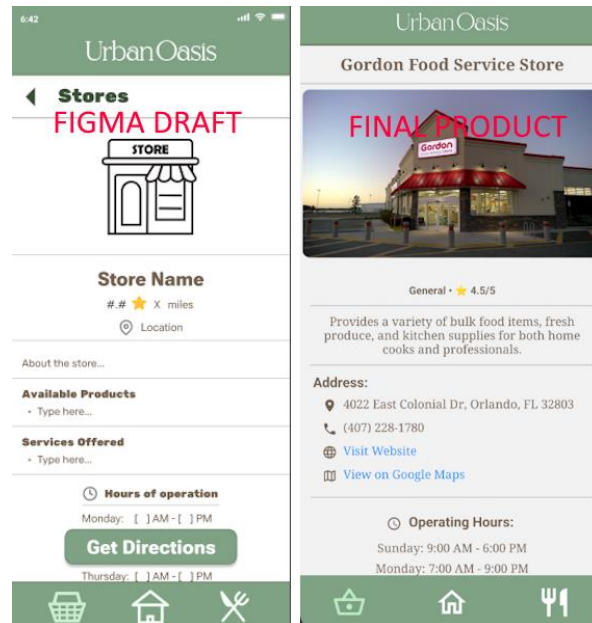
- Recipe Detail Screen** – Shows detailed information such as ingredients, steps, nutrition facts, and pricing.



- Store List Screen** – Lists local grocery stores, which are searchable and rated.



- **Store Detail Screen** – Displays store hours, address with map link, contact info, and description.

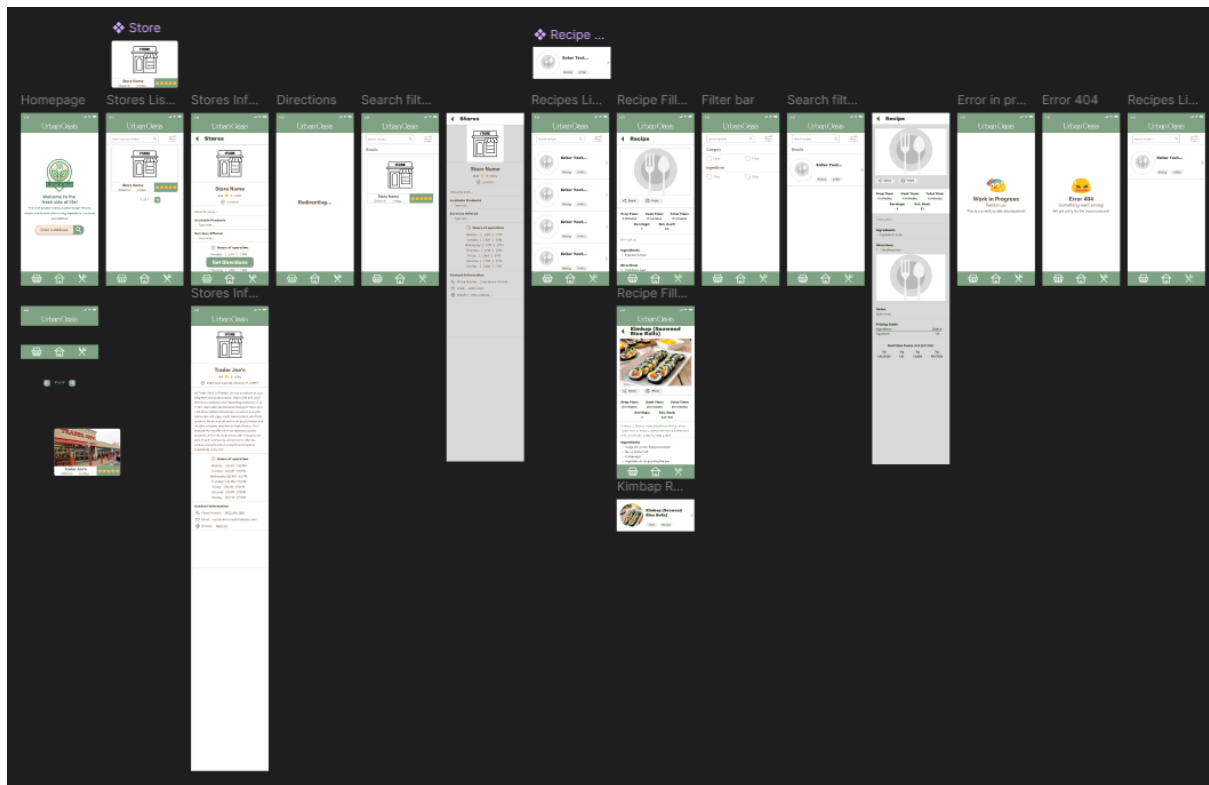


## Figma Usage

Figma was used to plan and visually construct the layout of the application before any development began. The design process included:

- Creating **frames** for each screen (Recipe List, Detail, Store List, etc.)
- Using **components** for reusable UI blocks like navigation bars, buttons, and cards
- Applying **auto layout** to keep spacing uniform across screen sizes
- Leveraging **vector icons** (like fork/knife, maps, and phone icons) for a polished, consistent look
- Using **prototype linking** in Figma to simulate navigation and interactions

The Figma file served as a reference throughout the development process to guide layout, styling, and spacing decisions.



## Intended Styling

The visual identity of the app was inspired by nature and organic themes to promote freshness and clarity. Key design choices include:

### Primary Color Palette:

- **#7FA184** – Earthy green for top bars and key UI accents
- **#467E53** – Deep green for text and emphasis
- **#705E4E** – Warm brown for secondary text and icon accents

### Typography:

- Serif fonts for titles and descriptions to give a traditional, elegant touch
- Clean sans-serif fonts for labels and input fields

**Card-based layout:** All content (recipes and stores) is grouped into shadowed cards with rounded corners and subtle elevation for depth.

**Icons over Emojis:** Instead of emojis, vector icons are used to ensure consistent appearance and color control across devices.

**Animations:** Touch animations (scale-in/scale-out) provide tactile feedback when pressing recipe or store cards.



## Recipe React Pages

### Overview

The Recipe section in the application consists of two broad functionalities. The first is a list of available recipes along with the category, rating, preparation time, and an image of the completed meal. The recipe list is accessed by clicking the fork and knife icon in the bottom navigation pane from any screen. The second is the recipe details which includes information such as a summary, the required ingredients, preparation directions, other helpful notes, and summary statistical information such as number of servings, estimated cost, calories, grams of fat, carbohydrates, and proteins. There is also a pricing guide to assist the user and an image of the completed meal. The recipe details are accessed by clicking the desired recipe from the list screen. Users can return to the recipe list by swiping right or navigate away from the recipe sections altogether by clicking the basket or home icons.

### Implementation

Both the recipe list and details are similarly coded and structured using React Native to handle the functionality. Appearance and presentation are handled using a combination of NativeWind, which allows the application to use Tailwind to style components, and basic JavaScript executed CSS styles.

### Importing Required Functionality

The React Native, custom, and third-party library functionality are initially imported into the application. This functionality is modular in nature consisting of various classes and methods. For example, to include general text or an image, the Text and Image classes must be imported, respectively.

```
import { Text, View, Image, Pressable, ScrollView } from "react-native";  
// Import required native React Native functionality.  
import { getRecipeList } from "../components/data"; // Import custom  
functionality from data.js external file.  
import { Skeleton } from "moti/skeleton"; // Import animation from Moti  
Skeleton plugin.
```

### Retrieving Data

The interface from the application to the database is called from methods in the data.js external file in the components directory. These methods are called from the recipe list and recipe detail screen pages using the useState statement. In the case of the recipe list page, the getRecipeList() method loads the data into the recipeList object which is iterated with the .map command as demonstrated in the following code excerpt:

```
/**  
 * Method in data.js file.  
 * Fetch all recipes from Firestore.  
 */  
export async function getRecipeList() {  
  try {
```

```

    const snapshot = await getDocs(collection(db, "recipes"));
    const recipeList = snapshot.docs.map((docSnap) => ({
      id: docSnap.id,
      title: docSnap.data().title || "Untitled",
      category: docSnap.data().category || "Uncategorized",
      rating: docSnap.data().rating || "Medium",
      totalTime: docSnap.data().totalTime || "30 minutes",
      imageUrl: docSnap.data().imageUrl || "https://placeholder.co/400"
    }));
    return recipeList;
  } catch (error) {
    console.error("Error fetching recipes:", error);
    return [];
  }
}

// Recipe List screen.
const [recipeList, setRecipeList] = useState([]);
...
<ScrollView>
  recipeList.map((item) => (
    <RecipeCard key={item.id} item={item} navigation={navigation} />
  )))
</ScrollView>

```

In the case of the recipe detail page, methods such as the `getRecipeDetail`, `getIngredients`, and `getDirections` methods are called passing in the `recipeId` parameter. The data is then loaded into objects such as `recipeDetail` which are referenced in the code as demonstrated in the following code excerpt:

```

/**
 * Method in data.js file.
 * Fetch recipe details by ID.
 */
export async function getRecipeDetail(recipeId) {
  try {
    const docRef = doc(db, "recipes", recipeId);
    const docSnap = await getDoc(docRef);
    if (docSnap.exists()) {
      return { id: docSnap.id, ...docSnap.data() };
    } else {
      console.log("No recipe found for ID:", recipeId);
      return null;
    }
  } catch (error) {
    console.error("Error fetching recipe detail:", error);
    return null;
  }
}

```



```

    }
  }

  // Recipe List screen.
  const [recipeDetail, setRecipeDetail] = useState(null);
  const [ingredients, setIngredients] = useState([]);
  const [directions, setDirections] = useState([]);
  ...
  <ScrollView>
    <Text style={styles.recipeTitle}>
      {recipeDetail.title || "Untitled Recipe"}
    </Text>
    ...
  </ScrollView>

```

## Navigation

Navigation is handled by the navigation object's navigate functionality. In the case of selecting a recipe from the list to view, the application passes the recipe ID as a parameter to the navigate object. In the case of pressing an icon, the name of the page to be displayed is used. These are demonstrated in the following code excerpt:

```

// Recipe List Navigation to Recipe Detail Page using recipe ID.
const RecipeCard = ({ item, navigation }) => {
  ...
  <Pressable
    onPress={() => navigation.navigate("RecipeDetail", { recipeId: item.id })}>
  </Pressable>

  // Fork and Knife Icon Navigation to Recipe List Page.
  <IconButton
    icon="silverware-fork-knife"
    size={45}
    iconColor="#BCEDC3"
    onPress={() => navigation.navigate("RecipeList")}
  />

```

## Formatting and Presentation

Formatting and presentation are handled by both NativeWind and JavaScript based CSS stylesheets. In both cases, the styles are applied inline to the React Native tags, referencing either the styling library or the custom styles StyleSheet object:

```

<View style={styles.imageContainer}>
  <Skeleton
    width={80}
    height={80}
    radius="round"

```

```

colorMode="light"
boneColor="#dee2e6"
highlightColor="#f7f7f7"
/>
</View>

...
const styles = StyleSheet.create({
  // Style for image container.
  imageContainer: {
    marginRight: 10
  },
});

```

### Functionality Not Implemented

Several intended features are not initially implemented due to the completion deadline, such as the ability to filter the recipe list based on some criteria. The elements and icons for these features were initially presented but are now commented out in the source code. When the time comes to implement these features, a good place for the developer to start would be to uncomment out this code and return these elements to the screen:

```

{/* <View style={styles.searchContainer}>
<TextInput
  placeholder="Search recipes"
  placeholderTextColor="#A7A7A7"
  mode="flat"
  underlineColor="transparent"
  activeUnderlineColor="transparent"
  keyboardType="default"
  cursorColor="#A7A7A7"
  style={styles.recipeSearch}
  editable={true}
/>

<IconButton
  icon="magnify"
  size={25}
  iconColor="#A7A7A7"
  onPress={() => console.log("Recipe-search pressed")}
/>
</View> */}

```

### Future Development

Although the recipe list and recipe detail pages work as expected to the front-end user, there are some desired coding changes to allow for better maintenance and increased functionality. First, both pages contain a considerable amount of duplicated code which if changed on one page might require changes on the other. For example, the bottom navigation bar code is

duplicated on each page. If it were updated on one page, it would not be consistent with the other page.

Another example includes each page having its own styles StyleSheet CSS object. A better practice would be to refactor these pages so that they shared the same object allowing changes to the object to propagate to both pages. Additionally, these pages should be refactored from the functional programming perspective which would allow for better unit testing. Although unit testing to some extent is possible to the current application, it would be desirable to extend the surface area of this testing to be more inclusive of the full range of functionality.

// Bottom Navigation Bar code is duplicated in both the recipe list and detail pages.

```
<View style={styles.bottomBar}>
  <IconButton
    icon="basket-outline"
    size={45}
    iconColor="white"
    onPress={() => navigation.navigate("StoresListScreen")}
  />
  <IconButton
    icon="home-outline"
    size={45}
    iconColor="white"
    onPress={() => navigation.navigate("Home")}
  />
  <IconButton
    icon="silverware-fork-knife"
    size={45}
    iconColor="#BCEDC3"
    onPress={() => navigation.navigate("RecipeList")}
  />
</View>
```

## Location React Pages

The Locations page in this application are responsible for displaying nearby store information, rating, and integration with the map view. This page includes components that fetch data from Firestore and present it with interactive UI elements styled in Tailwind CSS (NativeWind) and React Native Paper.

### Key Components:

1. StoreListScreen.jsx
  - a. Fetches store locations from Firebase using the `getLocations()` function.
  - b. Renders a list of store cards with data such as title, address, and rating.
  - c. Includes a skeleton loader for better user experience during data fetching.

```
useEffect(() => {
  getLocations().then((data) => {
    setStores(data);
    setTimeout(() => setLoading(false), 1000);
  });
}, []);
```

## 2. StoreDetailScreen.jsx

- a. Displays detailed information about a selected store.
- b. Receives store ID via route.params and fetches store details accordingly.
- c. Uses useEffect() to trigger a Firestore query.
- d. Shows skeleton placeholders while waiting for data.

```
useEffect(() => {
  const fetchStore = async () => {
    const data = await getStoreDetail(storeId);
    setStore(data);
    setTimeout(() => setLoading(false), 500);
  };
  fetchStore();
}, [storeId]);
```

## 3. getLocations()

- a. Located in components/data.js.
- b. Queries the locations collection in Firestore.

```
const snapshot = await getDocs(collection(db, "locations"));
```

## Future Development Considerations:

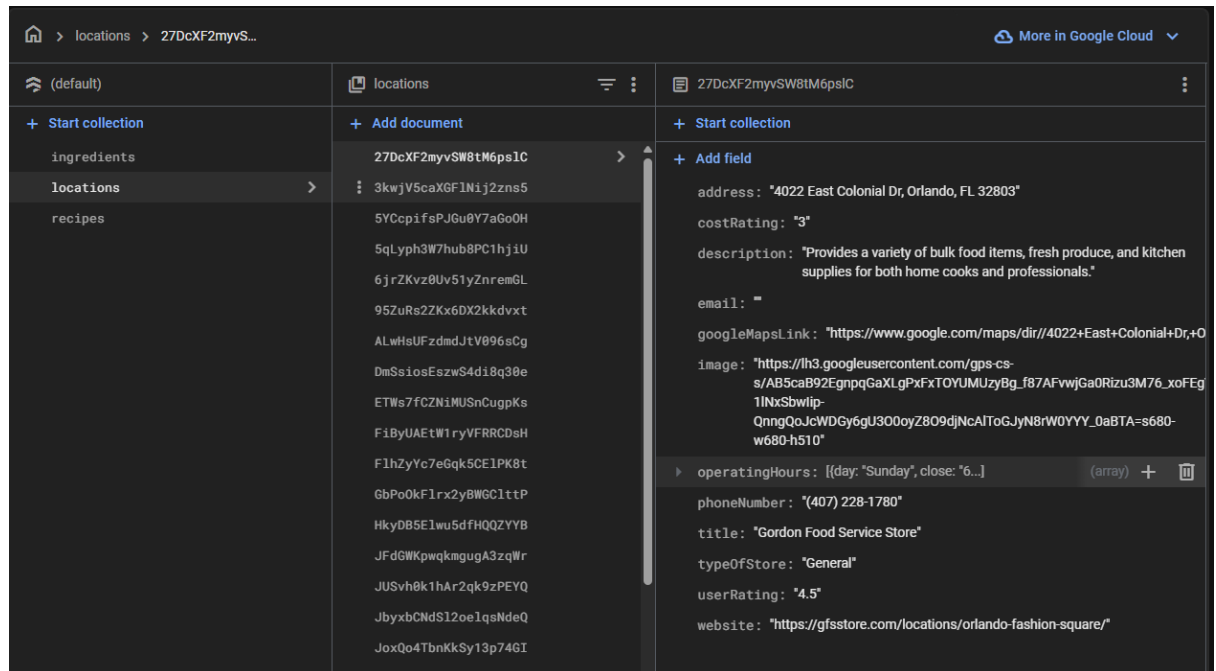
1. Add filter and filter and search capabilities
  - a. Use TextInput and dropdown menus to filter stores by distance, rating, and category.
2. Map integration
  - a. Embed a MapView component to visually display store pin with information.
3. Add Reviews
  - a. Allow users to submit reviews via a modal or screen
  - b. Store reviews in Firestore and render them on the StoreDetailScreen.

## Database Structure

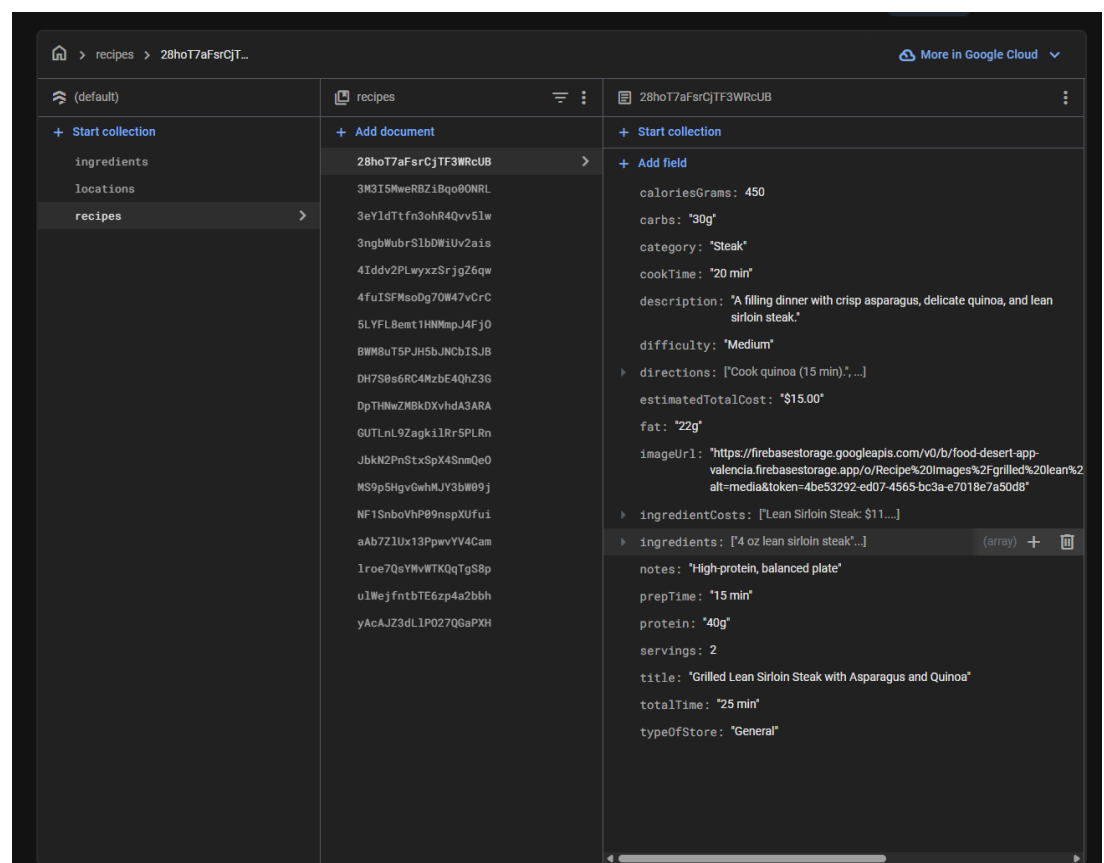
The Firebase Database is organized into two nodes: **Locations** and **Recipes**. Each part stores items using IDs, and each item has its own set of details. This setup is simple and works well with Firebase.

### Firebase Structure:

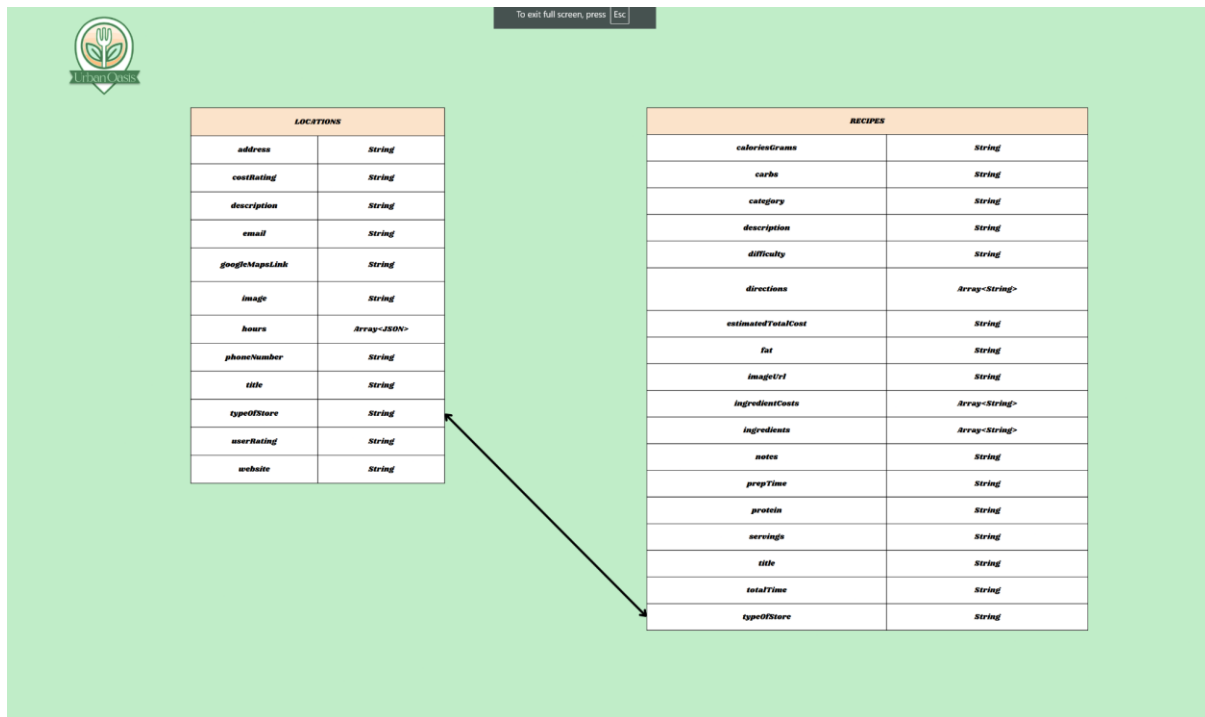
Location:



Recipes:



## ERD:



## Recipe Data

The recipe data object is a structure used to systematically represent all relevant information about a culinary recipe within a computer system. This object consists of various fields that describe the specific characteristics of the recipe. The most common fields are as follows:

**Title:** The name of the recipe.

**Category:** The country or culture of origin for the recipe.

**Total Time:** Estimated time required to complete the entire preparation.

**Servings:** Number of people the recipe serves.

**Estimated Total Cost:** Total estimated cost of all ingredients.

**Description:** A brief overview of the recipe or its purpose.

**Ingredients:** A list of items needed for preparation.

**Directions:** Step-by-step instructions for making the dish.

**Calories (grams):** Caloric content per serving.

**Difficulty:** The skill level required (e.g., Easy, Medium, Hard).

**Type of Store:** Where the ingredients can typically be purchased.

**Ingredient Cost:** Approximate cost per ingredient.

**Prep Time:** Time required to prepare ingredients.

**Cook Time:** Time needed to cook the recipe.

**Notes:** Extra tips, substitutions, or dietary suggestions.

**Fat (grams), Carbs (grams), Protein (grams):** Macronutrient content per serving.

This structure enables efficient storage, management, and reuse of recipes in both the database and the application.

```
--  
61 // --- Firestore Fetching Functions ---  
62 /**  
63  * Fetch all recipes from Firestore.  
64  */  
65 export async function getRecipeList() {  
66   try {  
67     const snapshot = await getDocs(collection(db, "recipes"));  
68     const recipeList = snapshot.docs.map((docSnap) => ({  
69       id: docSnap.id,  
70       title: docSnap.data().title || "Untitled",  
71       category: docSnap.data().category || "Uncategorized",  
72       rating: docSnap.data().rating || "Medium",  
73       totalTime: docSnap.data().totalTime || "30 minutes",  
74       imageUrl: docSnap.data().imageUrl || "https://placeholder.co/400"  
75     }));  
76   }
```

**Above:** Maps firebase information onto the recipe object.

```

12         <ScrollView style={{ padding: 16 }}>
13             {recipes.map((item) => {
14                 <View key={item.id} style={styles.recipeCard}>
15                     <Text style={styles.title}>{item.title}</Text>
16                     <Image source={{ uri: item.imageUrl }} style={styles.image} />
17                     <Text style={styles.subTitle}>
18                         {item.category} • {item.totalTime}
19                     </Text>
20                     <Text style={styles.description}>{item.description}</Text>
21                     <Text>Servings: {item.servings}</Text>
22                     <Text>Cost: {item.estimatedTotalCost}</Text>
23                     <Text>Calories: {item.caloriesGrams}</Text>
24                     <Text>
25                         Prep: {item.prepTime} | Cook: {item.cookTime}
26                     </Text>
27                     <Text>
28                         Fat: {item.fat} | Carbs: {item.carbs} | Protein: {item.protein}
29                     </Text>
30                     <Text style={styles.sectionTitle}>Notes:</Text>
31                     <Text>{item.notes}</Text>
32                     <Text style={styles.sectionTitle}>Ingredients:</Text>
33                     <Text>{item.ingredients}</Text>
34                     <Text style={styles.sectionTitle}>Directions:</Text>
35                     <Text>{item.directions}</Text>
36                     <Text style={styles.sectionTitle}>Pricing Guide:</Text>
37                     <Text>{item.ingredientCosts}</Text>
38                 </View>
39             })}

```

**Above:** recipeDetails

## How Recipe Data Can Be Collected:

### Data Sources:

- Existing recipe databases or cooking APIs
- Digital/physical cookbooks
- User submissions or community entries
- Collection Methods:
- Web Scraping (if allowed)
- Manual Entry by admins or verified contributors
- Chef/Nutritionist Collaborations

### Key Data to Collect:

- Recipe name and description
- Ingredients with quantities
- Step-by-step instructions
- Nutritional info (calories, protein, fat, etc.)
- Estimated local cost
- Images of the final dish and preparation steps
- How Recipe Data Should Be Formatted



### To ensure consistency and readability in the app:

- **Recipe Name:** Clear and concise
- **Description:** Short summary of the dish
- **Ingredients:** Bullet-point list with name, amount, and units
- **Directions:** Numbered or step-by-step format
- **Nutritional Info:** Include calories, fat, carbs, and protein per serving
- **Estimated Cost:** Based on local ingredient prices, in USD
- **Images:** JPEG or PNG format, high quality, with good lighting

## Location Data

The location data is used to store and display information about food-related locations such as food banks, grocery stores, or markets. This object includes fields that help users identify useful services in their area.

### The fields include:

**Title:** The name of the location or business.

**Address:** The full physical address of the location.

**Operating Hours:** Days of the week with corresponding open and close times.

**Description:** A brief summary of what the location offers.

**Cost Rating:** A 1–5 scale indicating how affordable the location is.

**Type of Store:** The category of the location (e.g., Pantry, Market, General).

**Phone Number:** A contact number for the location.

**Email:** Email address for communication.

**Website:** A link to the location's official site or page.

**Google Maps Link:** A direct link to view the location on Google Maps.

**Image:** A URL to a photo or logo of the location.

**User Rating:** Average rating from users based on their experience.

This allows for efficient searching, filtering, and displaying of location data within the app.

```

/**
 * Fetch directions from Firestore.
 */
export async function getDirections(recipeId) {
  try {
    const snapshot = await getDocs(collection(db, "recipes", recipeId, "directions"));
    return snapshot.docs.map((docSnap) => ({ id: docSnap.id, ...docSnap.data() }));
  } catch (error) {
    console.error("Error fetching directions:", error);
    return [];
  }
}

/**
 * Fetching data from the locations collection
 */

export async function getLocations() {
  try {
    const snapshot = await getDocs(collection(db, "locations"));
    const locationsList = snapshot.docs.map((docSnap) => ({
      id: docSnap.id,
      title: docSnap.data().title || "Unknown store :(",
      image: docSnap.data().image || "https://placeholder.co/400",
      address: docSnap.data().address || "Address not available :(",
      typeOfStore: docSnap.data().typeOfStore || "General",
      userRating: Number(docSnap.data().userRating) || 3,
      description: docSnap.data().description || "No description available",
      website: docSnap.data().website || "",
      googleMapsLink: docSnap.data().googleMapsLink || ""
    }));

    return locationsList;
  } catch (error) {
    console.error("Error fetching locations :", error);
    return [];
  }
}

```

**Above:** This function retrieves location data from Firebase and formats it for app's UI

## How Location Data Can Be Collected

### Data Sources:

- Public datasets from city or government open data portals
- Google Maps listings and business directories
- Community partnerships (e.g., local food banks, non-profits)
- Admin dashboard or CMS interface
- User-submitted suggestions

### Collection Methods:

- Manual entry by admins or project team
- Web APIs (e.g., Google Places API, Yelp API)
- Verified user submissions (pending review)
- Community outreach or local surveys

**Key Data to Collect:**

- Location name (store, pantry, center, etc.)
- Full address (street, city, state, ZIP)
- Operating hours by day
- Type of store (e.g., Market, General, Pantry)
- Cost rating (scale of 1 to 5)
- Contact info (phone number, optional email)
- Website and Google Maps link
- Image URL of the store or center
- User rating (if available)
- Short description of services offered

**How Location Data Should Be Formatted**

To ensure consistency and readability in the app:

**Title:** Use title case (e.g., “Gordon Food Service Store”)

**Address:** Include full street address, city, state, and ZIP

**Operating Hours:** Store as an array of objects with open/close times per day

**Description:** Short summary explaining the location’s purpose or services

**Cost Rating:** Whole number from 1 to 5

**Type of Store:** Predefined options like “General,” “Pantry,” or “Market”

**Phone Number:** Standard format like (407) 228-1780

**Email/Website/Google Maps Link:** Valid and complete https:// URLs

**Image:** JPEG or PNG and publicly accessible

**User Rating:** Decimal number (e.g., 4.5) for display as stars

## Functionality Not Implemented

**Overview**

Given constraints on time and team experience with new tools, there were features of the application that could not be implemented. These include in-scope requirements which represent the minimum functionality intended to meet client requirements. Additionally, there were out-of-scope requirements. These represent features meant to enhance the application not directly related to the client’s primary requirements.

## In-Scope Requirements Not Implemented

Requirement	Description
Users can filter locations by distance in miles.	This would allow users to view locations that are close to them by inputting an acceptable distance. A distance of 5 miles would only return a list of locations within a 5-mile radius of the user.
Users can filter locations by the types of ingredients they offer.	This would allow users to search for locations that sell certain ingredients.
Users can filter locations that are currently open.	This would allow users to filter locations that are currently open.
Users can filter recipes by ingredients.	This would allow users to search for recipes that include the ingredients they choose.
Users can exclude recipes that include certain ingredients.	This would allow users to exclude recipes that include certain ingredients such as potential allergens like mushrooms and peanuts.
Users can filter recipes by category.	This would allow users to search for recipes based on a category for that recipe.
Users can view the distance of a location to their address in miles.	This would allow users to visualize how far locations are from them.
Users can rate locations anonymously on a scale from 1 to 5 stars.	Users would be able to rate each location affecting the overall rating for a location. This rating would be an average of all user ratings input so far. Because user authentication is not a part of this application, this rating would be anonymous.
Users will be able to click a button to locate nearby stores that sell all ingredients included in a recipe.	This would be a button included on the recipe details page. It would take the user to the location list page with filters for ingredients based on the recipe the user pressed the button on. This would allow the user to quickly find stores that sell the ingredients needed for a single recipe.
Users can search for specific recipes.	A search bar was to be included in the application. The user would be able to type the name of a recipe they would like to find, and the search bar would show them a list of possible options.

## Out-of-Scope Requirements Not Implemented

Requirement	Description
Users can click a local map perspective image of a location in the location details page and be redirected to Google Maps.	An image would replace the use of a URL or button for redirecting users to Google Maps in the location details page. This image would include a Google Maps view of the location zoomed in to show local roads and nearby locations.
Users can leave anonymous comments along with ratings for each location.	In addition to rating locations, users would be able to leave comments along with their ratings.
Users can rate locations based on affordability, quality, and selection.	Ratings for locations would include categories for affordability, quality, and ingredient selection. Users could rate each category individually to provide a more accurate rating of a location.
Users can rate recipes between 1 and 5.	User ratings would be implemented for recipes.
Users can leave anonymous comments along with ratings for each recipe.	Users would be able to leave comments along with their ratings.
Users can print out recipes.	Users would be able to print out each recipe to a local printer.
Users can share recipes with non-application users on social media.	Users would be able to share their favorite recipes with non-application users across social media.
Users can rate recipes based on ease, taste, and estimated cost.	Ratings for recipes would include categories for ease of making, taste, and estimated cost.
User comments and reviews are to be moderated to prevent spam and inappropriate content.	A moderation system would be implemented to ensure user comments and ratings do not violate the terms of service. Spam, malicious ratings, and inappropriate content would not be tolerated.
Location data will include services offered.	The services provided by each location would be included in the location details page of each location.
Location data will include a list of ingredients sold.	The ingredients sold at a location would be stored in Firebase and be available for users to view within the location details page.
Recipe data will include recommended substitutions.	The recipe details page would include a section for the list of potential substitutions for each recipe.
Recipe data will include a meal type representing categories such as breakfast, lunch, and dinner.	A meal-type representing either breakfast, lunch, or dinner would be included for each recipe.
Users can view the last 10 recipes they have visited.	Local storage would be implemented to save the last 10 recipes the user accessed. These recipes would be viewable in a list format on a page labeled “recently viewed recipes”.
Users can view the last 10 locations they have visited.	Local storage would be implemented to save the last 10 locations the user accessed. These recipes would be viewable in a list format on a page labeled “recently viewed locations”.

## Plans for In-Scope Requirements

Requirement	Plan
Users can filter locations by distance in miles.	This would be implemented using either the Google Places API or the Google Distance Matrix API.
Users can filter locations by the types of ingredients they offer.	This was to be implemented by creating a category for each location called Store Type. The Store Type would define what ingredients are sold at a location. The intention was to allow users to filter based on Store Type instead of by ingredients directly.
Users can filter locations that are currently open.	This was to be implemented using store hours data stored in the Firebase database. The application would evaluate whether a location was open by checking for the current time and comparing it to the store hours stored for each location.
Users can filter recipes by ingredients.	No specific plans.
Users can exclude recipes that include certain ingredients.	No specific plans.
Users can filter recipes by category.	Categories for recipes needed to be defined and specified in a logical way. Currently, formal categories for recipes are not defined.
Users can view the distance of a location to their address in miles.	This would be implemented using either the Google Places API or the Google Distance Matrix API.
Users can rate locations anonymously on a scale from 1 to 5 stars.	No specific plans.
Users will be able to click a button to locate nearby stores that sell all ingredients included in a recipe.	This would be implemented by reusing filtering functionality for locations.
Users can search for specific recipes.	No specific plans.

## Plans for Out-of-Scope Requirements

There were no specific plans for the out-of-scope requirements. These requirements were to be discussed and worked on following the completion of all in-scope requirements.

## Commitment to Ethics

For already implemented work, and any future implementation, ethics and industry guidelines must be considered. Our code and documentation are our own original work, with references cited where applicable. Our application can be easily extended to comply with ADA/WCAG and any other accessibility/inclusion requirements. This application is meant to serve people, and so user-friendliness and equitable access remain a priority.

## By Food Application Team:

---

Elise Kidroske (Team Lead)

Danny Ken

George Mitchell

Gerardo Mota

Hanna Melo Fugulin

Herby Hertilien

Jason Graves

Josue Suazo

Maha Elabbadi

Max Shoemaker

Robiana Labady

April 2025