# EECS 581 — Project 3: MunchMates

**System Architecture Document**

Team 26
Aidan Ingram, Landon Bever, Hale Coffman, Aryan Kevat, Olivia Blankenship, Sam Suggs

## Synopsis

MunchMates helps users provide efficient meal planning through integrating the **Spoonacular API** and SDKs with a React frontend.

## Overview

MunchMates is a meal planning and recipe discovery application that allows users to efficiently plan their meals using recipe and ingredient data sourced from the **Spoonacular API**. The application integrates modern web technologies through a React-based frontend and leverages third-party SDKs and cloud services for data storage, authentication, and ingredient identification. By focusing on usability and performance, MunchMates provides an intuitive interface for both casual and advanced users who want to streamline their cooking, dieting, and shopping experiences.

## Purpose

The primary purpose of the MunchMates architecture is to:

- Provide a clean separation of concerns between **frontend UI**, **API service integration**, and **user data storage**.
- Allow users to search for recipes, explore nutritional information, and plan meals without dealing with raw API complexities.
- Offer personalized experiences through account management, saved recipes, and shared collections.
- Ensure scalability and maintainability by leveraging modular architecture components and well-defined service boundaries.

- Support future enhancements such as biometrics authentication, passkey login, advanced caching, and real-time collaborative meal planning.

# System Architecture Overview

The architecture for MunchMates follows a **client–service–API** pattern. The **React frontend** acts as the main interaction point, the **application services layer** abstracts all external API and storage interactions, and the **Spoonacular API** provides recipe and ingredient data. The data flow is structured to minimize coupling between components, allowing rapid iteration and future extension.

## Key Architectural Goals

- **Modularity**: Each service (auth, API, database, ingredient recognition) operates independently.
- **Extensibility**: Additional features (e.g., caching, new APIs) can be added without modifying existing core components.
- **Scalability**: Cloud-based backend components enable horizontal scaling and real-time data synchronization if expanded.
- **Security and Privacy**: Authentication services and compliance with API usage rules (e.g., caching restrictions) protect user and provider data.

# Major Components

## Database Layer

A backend database will be used to store user-specific data including:

- **Authentication Data**: Secure storage of user credentials, MFA configurations, and session tokens.
- **User Preferences**: Dietary restrictions, favorite cuisines, and other customization settings.
- **Application Data**: Saved meal plans, grocery lists, and shared recipe collections.

The database also supports **multi-user interaction** features, enabling recipe sharing between friends or households.

## Authentication Service

Authentication will be implemented through **Firebase Auth** or a custom solution. This component handles:

- User registration and login with password-based authentication.
- Password reset and account recovery flows.
- Future support for multi-factor authentication and passwordless login via passkeys or biometrics.

### Spoonacular Service

This service is a thin abstraction layer that communicates with the Spoonacular API. Responsibilities include:

- Constructing and sending API requests for recipe and ingredient data.
- Parsing and error handling of responses.
- Returning well-structured object types (through the official SDK) to the frontend.
- Implementing short-term caching (subject to Spoonacular's API rules) to reduce redundant network calls.

### Ingredient Identification Service

This component leverages device-specific camera and gallery APIs to identify ingredients through images. Integration with a vision API (e.g., Google Vision) enables:

- Direct ingredient detection from uploaded or captured images.
- Recipe inference based on identified ingredients.
- User override and confirmation for ambiguous detections.

### User Interface (Frontend)

The user interface is implemented using React. Core principles:

- **Responsiveness**: Adaptive layouts across mobile, tablet, and desktop.
- **Reusability**: Components for meal plans, grocery lists, and recipe previews are modular and composable.
- **User Experience**: Clear navigation, error handling, and loading states.

## Interactions and Data Flow

The architecture follows a structured data flow:

1. Users interact with the **React frontend** to search for or capture ingredients.
2. The frontend calls the **Spoonacular Service** or **Ingredient Identification Service**.
3. Identified ingredients or recipes are returned as structured objects.
4. User actions (e.g., saving recipes) trigger database writes through the **Authentication and Database layer**.
5. Data is rendered back to the user via updated UI components.

As illustrated in **Figure 1**, the data flow ensures a clear separation of responsibilities between UI, services, and APIs.

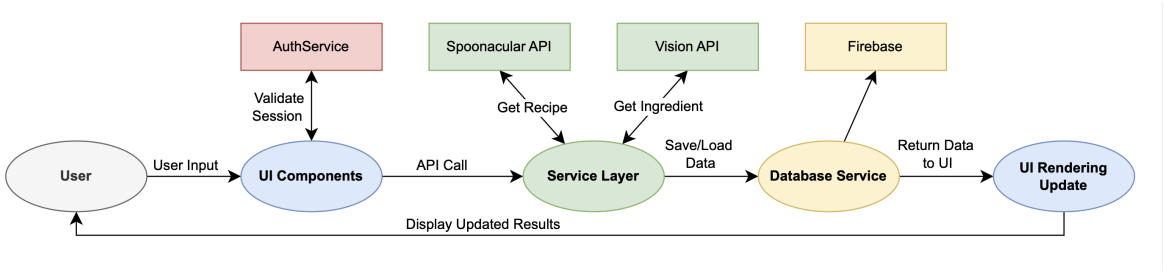


Figure 1: High-Level Data Flow Diagram. This diagram illustrates how user interactions travel through the UI components, service layer, external APIs (Spoonacular and Vision), and database services. It shows how data is retrieved, processed, and returned to the user interface for rendering.

# UML Diagrams

## Class Relation Diagram

The system's internal structure and relationships between key entities are shown in **Figure 2**.

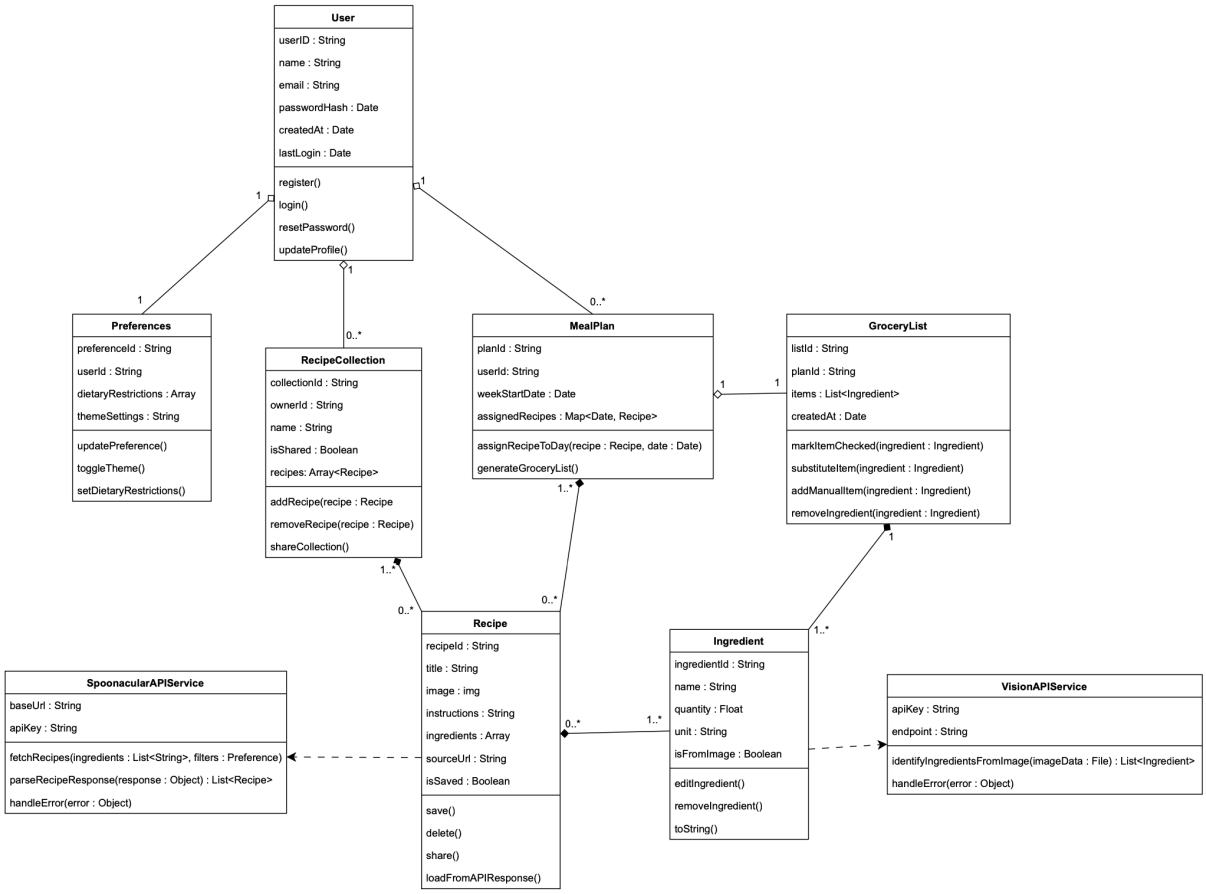


Figure 2: Component Diagram. This diagram depicts the core entities and relationships within the system, including User, RecipeCollection, MealPlan, GroceryList, Ingredient, and API services. It highlights how application logic and data are structured and interconnected.

## Use Case Diagram

**Figure 3** outlines the primary user interactions with the system, including core features and their connections to external services.

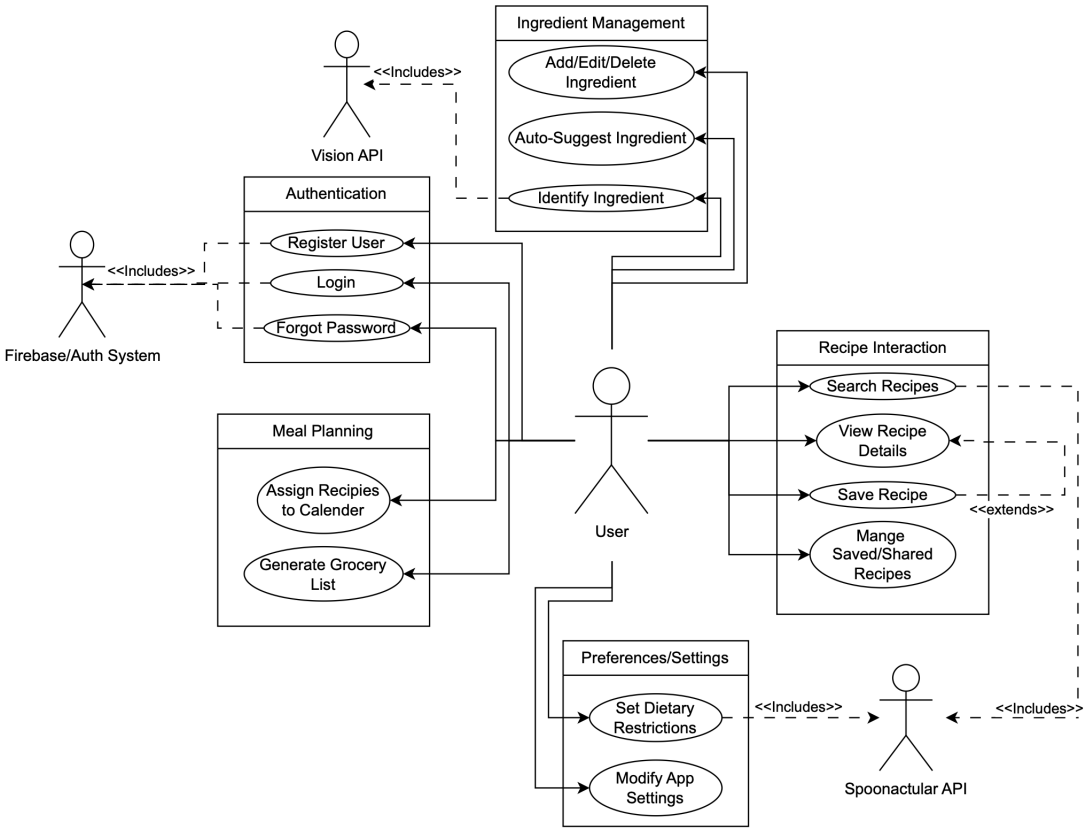


Figure 3: Use Case Diagram. This diagram presents key user interactions with the system, including authentication, ingredient management, recipe search and saving, meal planning, and preferences. External services (e.g., Firebase Auth, Spoonacular API, and Vision API) are also represented to show integration points.

# Future Considerations

- **Offline Mode**: Implementing local caching or service workers for offline recipe browsing.
- **Social Features**: Allowing users to follow other meal planners or share curated lists.
- **Nutrition Goals**: Personalized recommendations and tracking based on stored preferences.
- **API Extensions**: Supporting alternative or supplemental food data sources.

# Testing & Verification Plan

- **Unit Testing**: Component-level testing for service logic, including error handling and data parsing.

- **Integration Testing**: End-to-end testing of API calls, database reads/writes, and frontend rendering.
- **UI Testing**: Ensuring accessibility, responsiveness, and proper state handling.
- **Load Testing**: Ensuring API and caching layers can handle concurrent requests efficiently.