MunchMates
System Architecture

Team 17 - February 8, 2026

Landon Bever | Olivia Blankenship | Hale Coffman | Aidan Ingram | Aryan Kevat | Sam Suggs

Synopsis

MunchMates helps users provide efficient meal planning through integrating the Spoonacular API, a PostgreSQL database with Prisma, and Keycloak authentication with a Next.js 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 Next.js (App Router) frontend with React 19 and leverages Keycloak for authentication, PostgreSQL for persistent data storage, and TensorFlow.js for client-side 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 route layer, and database persistence.
- Allow users to search for recipes, explore nutritional information, create custom recipes, and plan meals without dealing with raw API complexities.
- Offer personalized experiences through account management, saved recipes, custom recipes, shared collections, pantry tracking, and grocery list management.
- Ensure scalability and maintainability by leveraging modular architecture components and well-defined service boundaries.
- Support future enhancements such as advanced caching, real-time collaborative meal planning, and expanded social features.

System Architecture Overview

The architecture for MunchMates follows a client–API route–service pattern. The Next.js frontend acts as the main interaction point, Next.js API routes serve as the application services layer abstracting all external API and database interactions, and the Spoonacular API provides recipe and ingredient data. User data is persisted in a PostgreSQL database accessed through Prisma ORM. Authentication is handled by a self-hosted Keycloak instance using JWT Bearer tokens. 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: Docker-based backend components enable horizontal scaling and real-time data synchronization if expanded.
- Security and Privacy: Keycloak authentication with JWT verification and compliance with API usage rules (e.g., caching restrictions) protect user and provider data.

Major Components

Database Layer

A PostgreSQL database stores all user-specific data, accessed through the Prisma ORM (v7). The database runs in Docker alongside Keycloak, sharing the same Postgres container:

The database stores:

User Data:

- Base user records linked to Keycloak via the sub claim, along with user profiles containing cuisine preferences, dietary restrictions, and intolerances.
- Recipe Data: Saved (bookmarked) Spoonacular recipes and user-created custom recipes (IDs starting at 100000 to avoid collisions with Spoonacular IDs).
- Meal Planning Data: Weekly meal plans with individual meal entries assigned to specific dates and meal types (breakfast, lunch, dinner).
- Inventory Data: Pantry items with quantity and expiry tracking, grocery list items with completion status, and user-defined grocery categories.

- Social Data: Shared recipe collections with role-based membership (owner, editor, viewer) and per-recipe attribution.

The database also supports multi-user interaction features, enabling recipe sharing between friends or households through the shared collections system.

Authentication Service

Authentication is implemented through a self-hosted Keycloak instance (v26.4). This component handles:

- User registration and login with password-based authentication.
- Password reset and account recovery flows via Mailpit (development email testing).
- JWT-based session management — Bearer tokens are passed on each API request and verified server-side using the verifyBearer() function in lib/verifyToken.ts.
- Token validation via JWKS endpoint, checking JWT signature, issuer, audience, and clock skew tolerance.
- Client-side token management through keycloak-js with PKCE (S256) and silent SSO checks.
- The authedFetch helper (lib/authedFetch.ts) wraps the Fetch API to automatically inject the Keycloak Bearer token into all authenticated requests.

Spoonacular Service

This service is a thin abstraction layer that communicates with the Spoonacular API through Next.js API routes under /api/spoonacular/. Responsibilities include:

- Constructing and sending API requests for recipe search, recipe details, cooking instructions, and ingredient data.
- Parsing and error handling of responses, including API rate limit detection (HTTP 402).
- Returning well-structured JSON objects to the frontend.
- Proxying requests server-side to keep the Spoonacular API key secure (never exposed to the client).

Ingredient Identification Service

This component leverages TensorFlow.js with the MobileNet model for client-side image classification. The implementation in components/image-classification-dialog.tsx enables:

- Direct ingredient detection from uploaded images or device camera capture.
- Dynamic model loading (lazy-loaded when the dialog opens to reduce initial bundle size).
- Classification results mapped to ingredient label suggestions.
- User override and confirmation for ambiguous detections.

- Fully client-side — no server-side ML infrastructure required.

User Interface (Frontend)

The user interface is implemented using Next.js 16 with the App Router and React 19. Core principles:

- Responsiveness: Adaptive layouts across mobile, tablet, and desktop using Tailwind CSS 4.
- Reusability: Components for meal plans, grocery lists, recipe previews, and shared collections are modular and composable, built with shadcn/ui (Radix UI primitives) and Lucide React icons.
- User Experience: Clear navigation via a collapsible sidebar, loading states, error handling, and intercepting routes for slideover recipe previews.
- Drag and Drop: Meal planner, pantry, and grocery list support reordering via dnd-kit.

Pantry Intelligence Layer (new component)

The pantry intelligence layer is a domain service responsible for ingredient normalization, pantry-aware recipe search, and inventory updates. It defines canonical ingredient names, applies normalization rules to pantry items and recipe ingredients, and exposes pantry-intersection queries used by the “What Can I Make?” feature. This layer also supports pantry auto-deduct flows by mapping recipe ingredients to normalized pantry entries and updating quantities when a user marks a recipe as cooked

Platform Hardening Layer (new component)

The platform hardening layer provides standardized API error responses, frontend error presentation patterns, API protection (rate limiting), and system documentation (OpenAPI). It also defines the testing strategy for core services and supports automated builds/tests via CI/CD to ensure consistent deployments and regression prevention

Interactions and Data Flow

The architecture follows a structured data flow:

- Users interact with the Next.js frontend to search for recipes, create custom recipes, or manage their meal plans.
- The frontend calls Next.js API routes, which serve as the service layer.
- API routes communicate with the Spoonacular API for external recipe data or with PostgreSQL (via Prisma) for user-specific data.

- For custom recipes (IDs >= 100000), the detail view fetches from the local database API instead of Spoonacular.
- User actions (e.g., saving recipes, updating pantry) trigger database writes through authenticated API routes.
- Data is rendered back to the user via updated UI components.

The data flow ensures a clear separation of responsibilities between UI, API routes, external APIs, and the database.

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

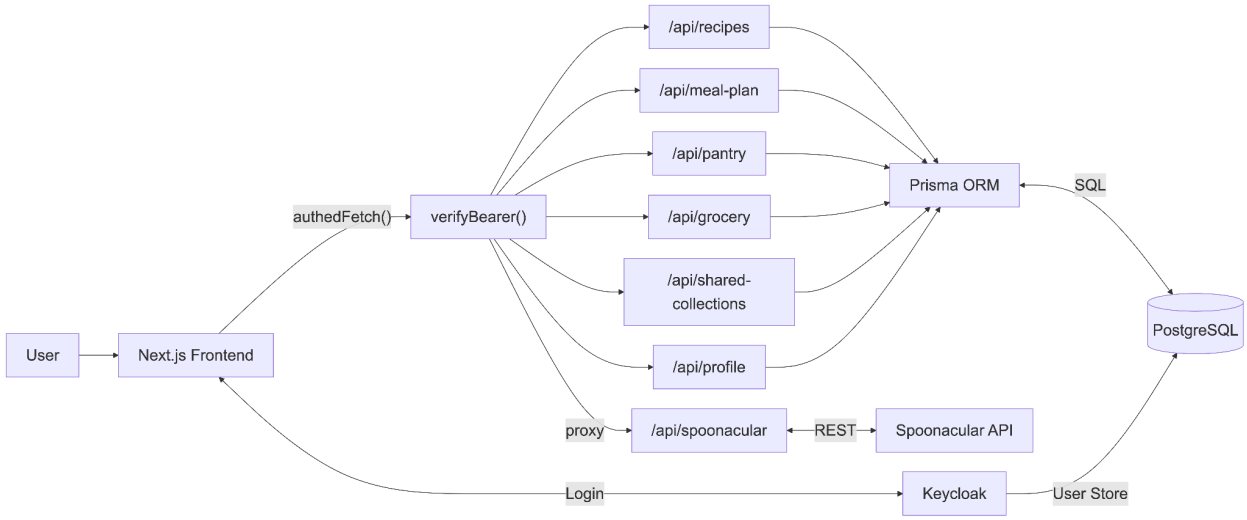

Figure 1: The overall MunchMates system architecture and the flow of data through the application.

- The user interacts with the Next.js frontend, which sends authenticated requests using authedFetch() and verifies access through verifyBearer().
- Once authenticated, requests are routed to backend API endpoints. These endpoints communicate with PostgreSQL through Prisma ORM for persistent data storage.
- External recipe searches are handled through the /api/spoonacular proxy endpoint, which forwards REST requests to the Spoonacular API.
- User authentication and login are managed through Keycloak, which acts as the centralized user store.

UML Diagrams

Database Relation Diagram

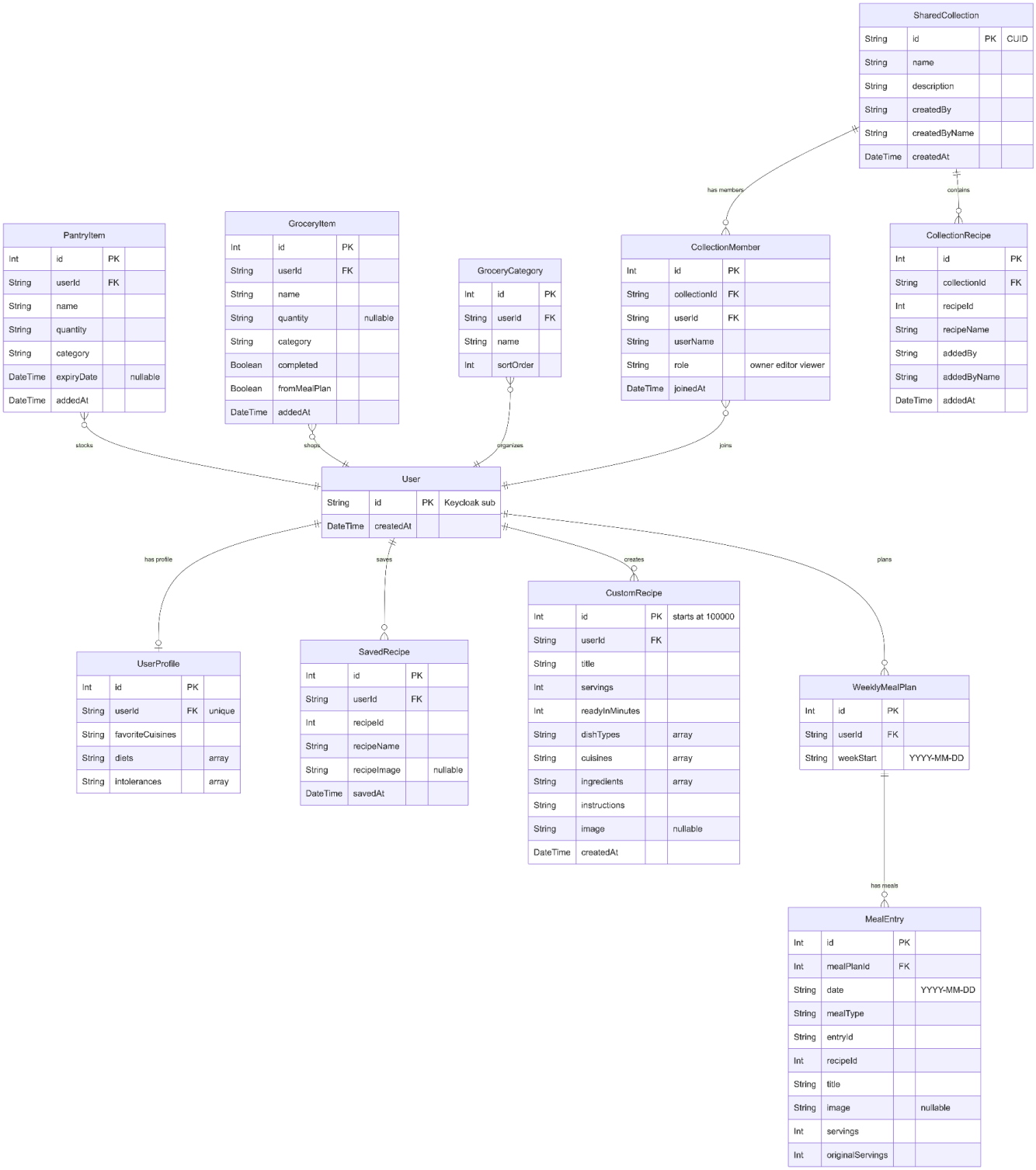

Figure 2: This diagram represents the MunchMates database schema and the relationships between core application entities.

- The User table acts as the central hub, linking to saved recipes, custom recipes, pantry items, grocery items/categories, meal planning data, and user profile preferences.
- Shared recipe collections are supported through the SharedCollection, CollectionMember, and CollectionRecipe tables, allowing multiple users to collaborate with role-based access.
- The schema is designed to ensure persistent storage, organized grocery/pantry tracking, and structured weekly meal planning while supporting both personal and shared recipe management.

Use Case Diagram

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

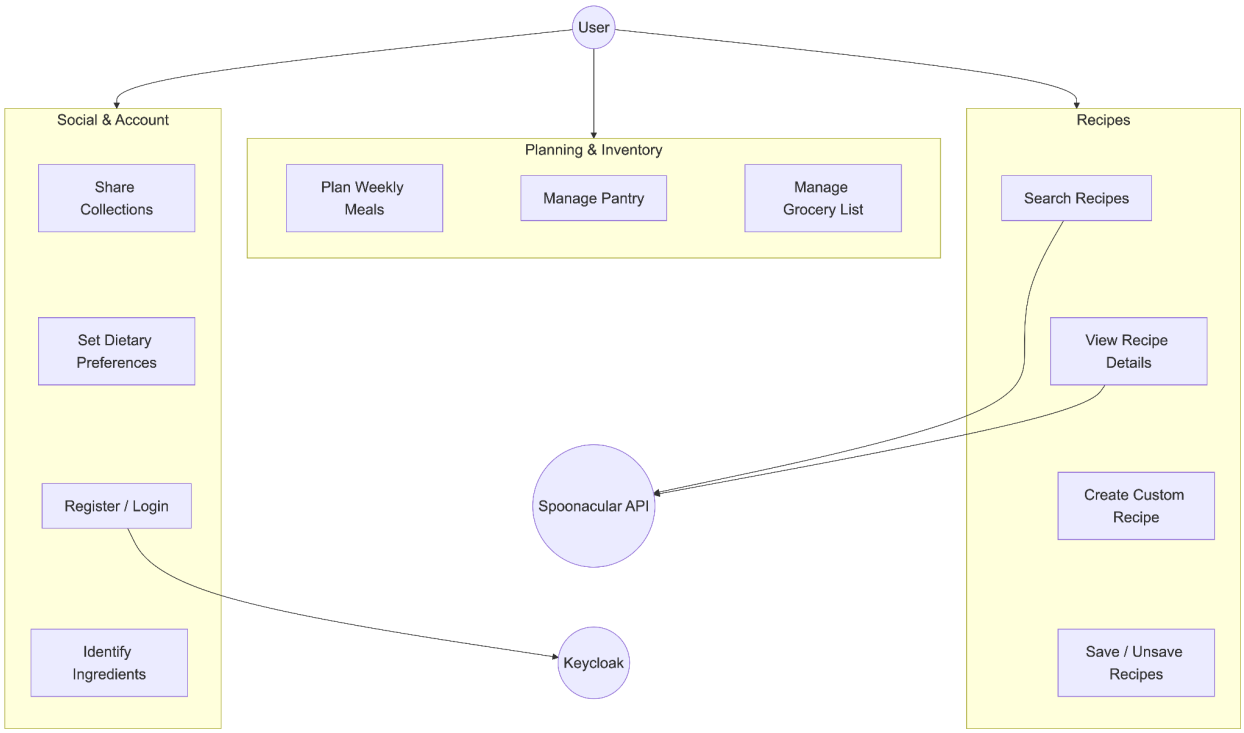

Figure 3: Use Case Diagram. This diagram presents key user interactions with the system, including authentication, ingredient management, recipe search and saving, meal planning, and preferences. External services (e.g., Keycloak Auth, Spoonacular AP) are also represented to show integration points.

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, in addition to Spoonacular.
- Real-time Collaboration: Live updates for shared collections and meal planning for groups.

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 database layers can handle concurrent requests efficiently.