MunchMates

System Architecture

Team 17

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

February 8, 2026

Synopsis

MunchMates helps users efficiently plan meals by integrating the Spoonacular API, a PostgreSQL database with Prisma ORM, and Keycloak authentication within a Next.js frontend.

1 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 users who want to streamline their cooking, dieting, and shopping experiences.

The primary goals of the architecture are to provide a clean separation of concerns between frontend UI, API route layer, and database persistence; to allow users to search recipes, explore nutritional information, create custom recipes, and plan meals without dealing with raw API complexities; and to ensure scalability by leveraging modular components and well-defined service boundaries.

2 System Architecture Overview

The architecture 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, illustrated in Figure 1, is structured to minimize coupling between components, allowing rapid iteration and future extension.

- **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 if expanded.
- **Security** — Keycloak authentication with JWT verification and server-side API key management protect user and provider data.

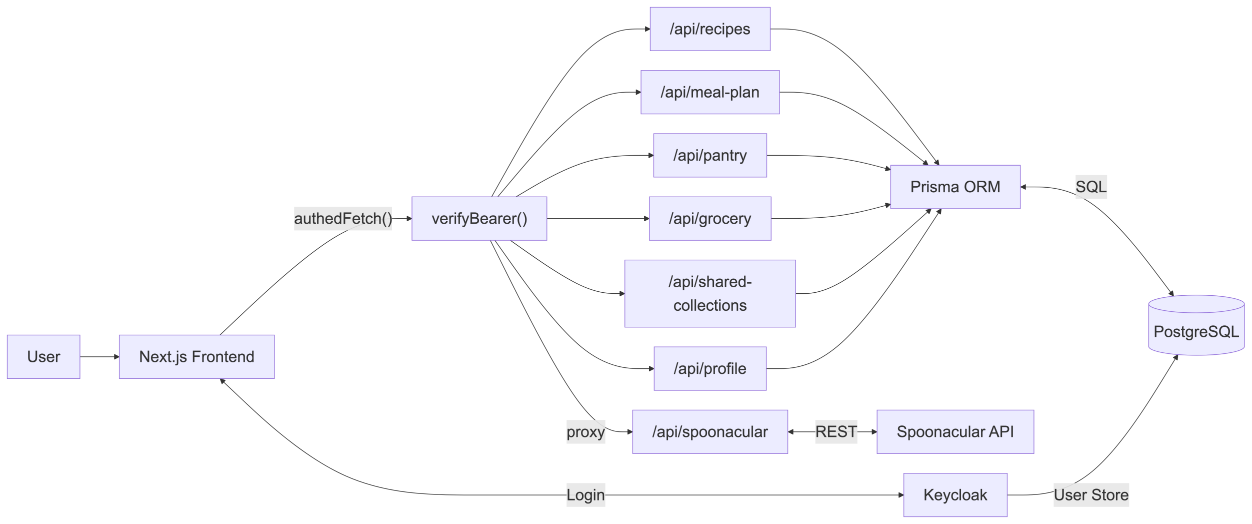

Figure 1: Overall system architecture and data flow. The user interacts with the Next.js frontend, which sends authenticated requests via `authedFetch()` and verifies access through `verifyBearer()`. Backend API routes communicate with PostgreSQL through Prisma ORM for persistent storage. External recipe data is proxied through `/api/spoonacular`. Authentication is managed through Keycloak as the centralized user store.

3 Major Components

3.1 Database Layer

A **PostgreSQL** database stores all user-specific data, accessed through **Prisma ORM** (v7). The database runs in Docker alongside Keycloak, sharing the same Postgres container on port 5433 but using a separate database and user. The Prisma client singleton uses the `@prisma/adapter-pg` driver adapter and prevents connection pool exhaustion during Next.js hot-reload.

The database stores user records linked to Keycloak via the `sub` claim; user profiles with cuisine preferences, dietary restrictions, and intolerances; saved and custom recipes; weekly meal plans with individual entries per date and meal type; pantry items with quantity and expiry tracking; grocery items with completion status and user-defined categories; and shared recipe collections with role-based membership. Figure 2 shows the complete entity relationship model.

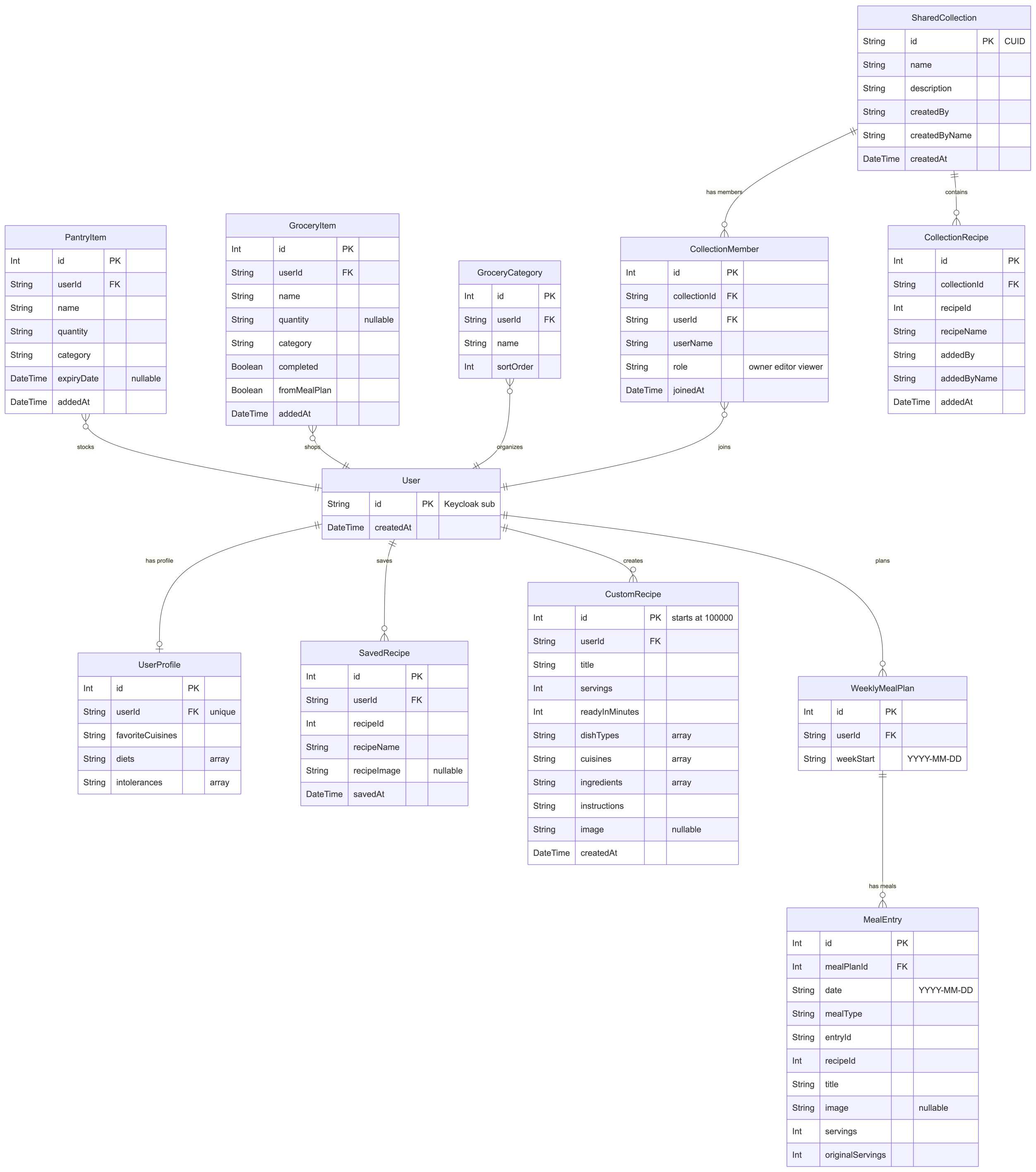

Figure 2: Database entity relationship diagram. The User table acts as the central hub, linking to saved recipes, custom recipes, pantry items, grocery items/categories, meal planning data, and user profile preferences. Shared collections support multi-user collaboration with role-based access (owner, editor, viewer).

3.2 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); and JWT-based session management where Bearer tokens are passed on each API request and verified server-side using `verifyBearer()` in `lib/verifyToken.ts`. Token validation checks JWT signature, issuer, audience, and clock skew tolerance via the JWKS endpoint. Client-side token management uses `keycloak-js` with PKCE (S256) and silent SSO checks. The `authedFetch` helper wraps the Fetch API to automatically inject the Bearer token into all authenticated requests.

3.3 Spoonacular Service

This service is a thin abstraction layer that communicates with the **Spoonacular API** through Next.js API routes under `/api/spoonacular/`. It constructs and sends requests for recipe search, recipe details, cooking instructions, and ingredient data; parses responses with error handling including API rate limit detection (HTTP 402); and returns well-structured JSON to the frontend. All requests are proxied server-side to keep the Spoonacular API key secure. A **recipe caching layer** provides in-memory caching for frequently accessed recipe results and details by ID, reducing redundant API calls and improving response times. Integrated endpoints include:

Endpoint	**Purpose**
`/recipes/complexSearch`	Search by query, cuisine, dish type, diet
`/recipes/findByIngredients`	Search by ingredient list
`/recipes/{id}/information`	Full recipe details (ingredients, nutrition)
`/recipes/{id}/analyzedInstructions`	Step-by-step cooking instructions

3.4 Ingredient Identification Service

This component leverages **TensorFlow.js** with the **MobileNet** model for client-side image classification. The implementation enables direct ingredient detection from uploaded images or device camera capture, with the model lazy-loaded when the dialog opens to reduce initial bundle size. Classification results are mapped to ingredient label suggestions with user override for ambiguous detections. The entire process runs client-side, requiring no server-side ML infrastructure.

3.5 Pantry Intelligence Layer

The pantry intelligence layer is a domain service responsible for ingredient normalization, pantry-aware recipe search, and inventory updates. The ingredient aggregator defines canonical names and applies normalization rules across volume, weight, and count units, enabling cross-recipe ingredient consolidation for grocery list generation and duplicate merging.

This layer exposes pantry-intersection queries used by the **"What Can I Make?"** feature, which filters the recipe catalog to those matching on-hand pantry ingredients through a one-click UI action. It also supports a **pantry auto-deduct** flow that maps recipe ingredients to normalized pantry entries and updates quantities when a user marks a recipe as cooked, keeping inventory accurate without manual adjustment.

3.6 Platform Hardening Layer

The platform hardening layer provides standardized API error responses with consistent status codes and structured JSON error bodies, paired with frontend error presentation patterns for user-facing feedback. API protection is enforced through rate limiting on public-facing routes. System documentation is generated via an **OpenAPI/Swagger** specification for the API route layer.

This layer also defines the testing strategy: unit tests for core domain logic (pantry normalization, grocery aggregation), integration tests for API routes and database operations, and UI tests for accessibility and responsiveness. A **GitHub Actions CI/CD pipeline** automates builds, linting, and test execution to ensure consistent deployments and regression prevention.

3.7 User Interface

The user interface is implemented using **Next.js 16** with the App Router and **React 19**. Layouts are responsive across mobile, tablet, and desktop using **Tailwind CSS 4**. Components are modular and composable, built with **shadcn/ui** (Radix UI primitives) and **Lucide React** icons. Navigation uses a collapsible sidebar with loading states, error handling, and intercepting routes for slideover recipe previews. Drag-and-drop reordering via **dnd-kit** is supported in the meal planner, pantry, and grocery list views.

Additional frontend capabilities include **recipe serving scaling** with dynamic ingredient amount adjustment and a serving selector UI; **print-friendly recipe views**; **nutrition display** showing macros and allergen highlights on recipe pages; and a **daily nutrition summary** that aggregates meal plan data with dietary goals tracking.

3.8 API Routes

All data operations go through **Next.js API routes** at `/api/`. These routes act as the service layer between the frontend and the database/external APIs. All endpoints (except Spoonacular proxies) require a valid Keycloak Bearer token, verified by `verifyBearer()`.

Endpoint	**Methods**	**Purpose**
`/api/profile`	GET, POST	User profile (cuisines, diets, intolerances)
`/api/recipes/saved`	GET, POST, DELETE	Saved recipes management
`/api/recipes/create`	POST, GET	Create and retrieve custom recipes
`/api/meal-plan`	GET, POST	Weekly meal plan CRUD
`/api/pantry`	GET, POST, PUT, DEL	Pantry items management
`/api/grocery`	GET, POST, PUT, DEL	Grocery list management
`/api/grocery/categories`	GET, POST, DELETE	Grocery categories
`/api/grocery/import`	POST	Import items from meal plan
`/api/shared-collections`	GET, POST, DELETE	Collection management
`/api/shared-collections/[id]/members`	GET, POST, DEL	Collection membership
`/api/spoonacular/recipes/*`	GET	Proxied Spoonacular API calls

4 Interactions and Data Flow

The architecture follows a structured data flow:

1. Users interact with the Next.js frontend to search for recipes, create custom recipes, or manage their meal plans and pantry.
2. The frontend calls Next.js API routes, which serve as the service layer.
3. API routes communicate with the Spoonacular API for external recipe data or with PostgreSQL (via Prisma) for user-specific data.
4. For custom recipes (IDs $\geq 100\,000$), the detail view fetches from the local database API instead of Spoonacular.
5. User actions (saving recipes, updating pantry, marking recipes cooked) trigger database writes through authenticated API routes.
6. Data is rendered back to the user via updated UI components.

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

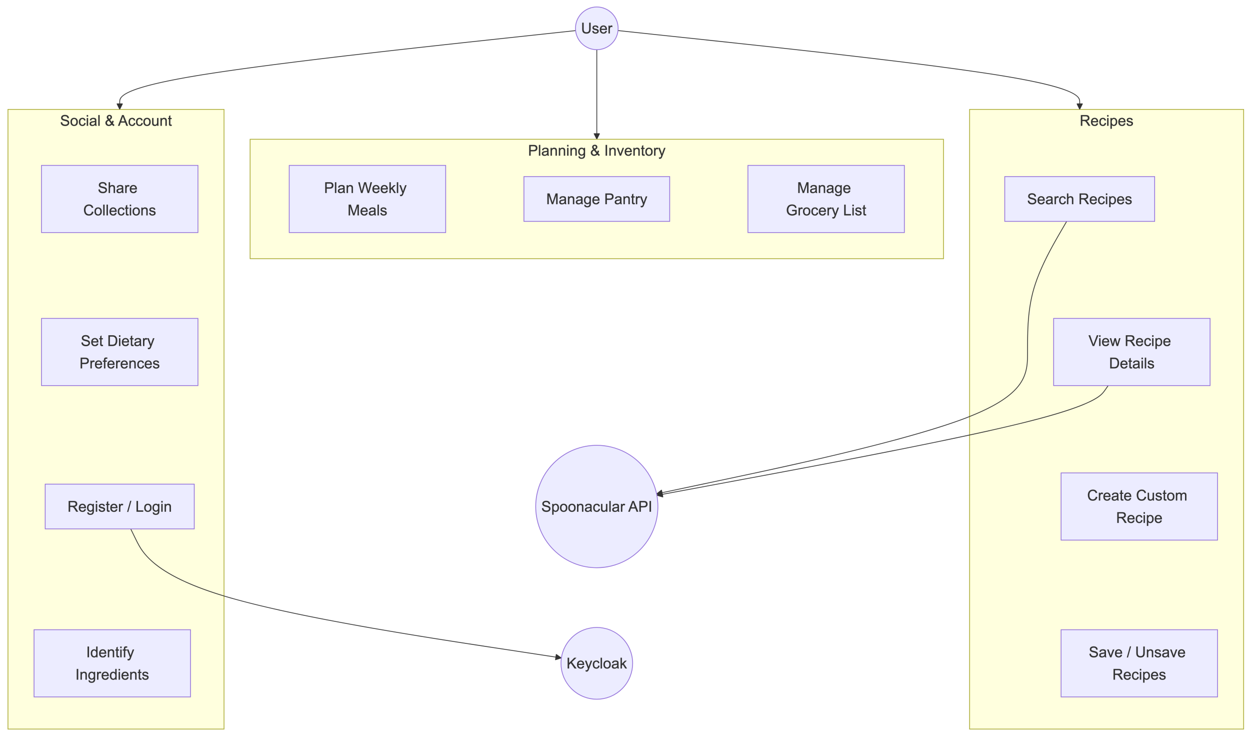

Figure 3: Use case diagram showing key user interactions: authentication, recipe search and management, meal planning, pantry and grocery tracking, shared collections, dietary preferences, and ingredient identification. External services (Keycloak, Spoonacular API) are represented at integration points.

5 Infrastructure and Development Environment

The development environment uses **Docker Compose** to orchestrate three services:

Service	**Version**	**Port**	**Purpose**
PostgreSQL	16	5433	Database for Keycloak and MunchMates
Keycloak	26.4.2	8080	Authentication and user management
Mailpit	—	8025 (web), 1025 (SMTP)	Development email testing

Key configuration files include `prisma/schema.prisma` (database schema), `prisma.config.ts` (Prisma CLI configuration), `lib/verifyToken.ts` (JWT verification), `lib/prisma.ts` (Prisma client singleton), `lib/authedFetch.ts` (authenticated fetch helper), and `lib/keycloak.ts` (Keycloak client setup). Environment variables (database URL, API keys, Keycloak config) are managed through `.env.local`.

6 Future Considerations

- **Offline Mode** — Service workers for offline recipe browsing.
- **Social Features** — Following other meal planners and sharing curated lists.
- **API Extensions** — Supporting alternative food data sources beyond Spoonacular.
- **Real-time Collaboration** — Live updates for shared collections and group meal planning.