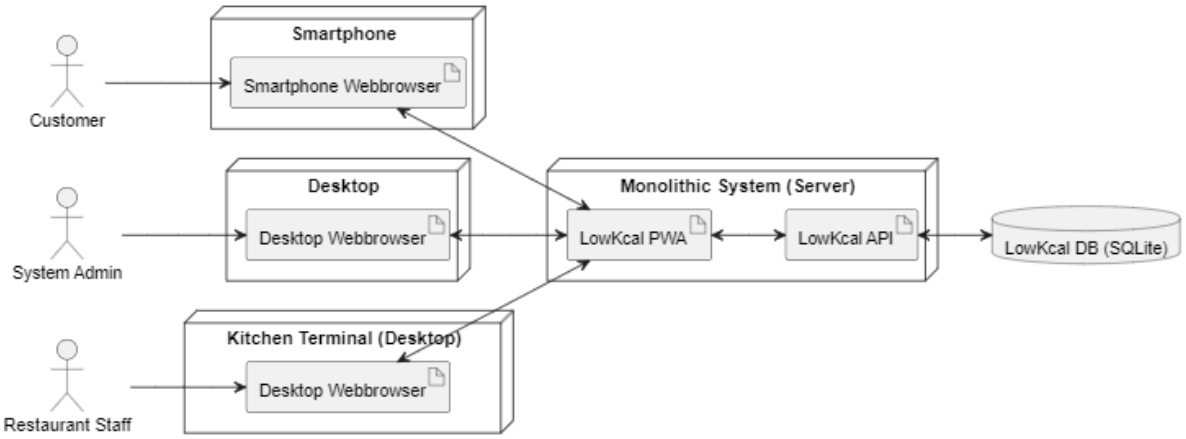
LowKcal System - Architecture and Tecnologies

1. Architecture

a. High-Level Architecture



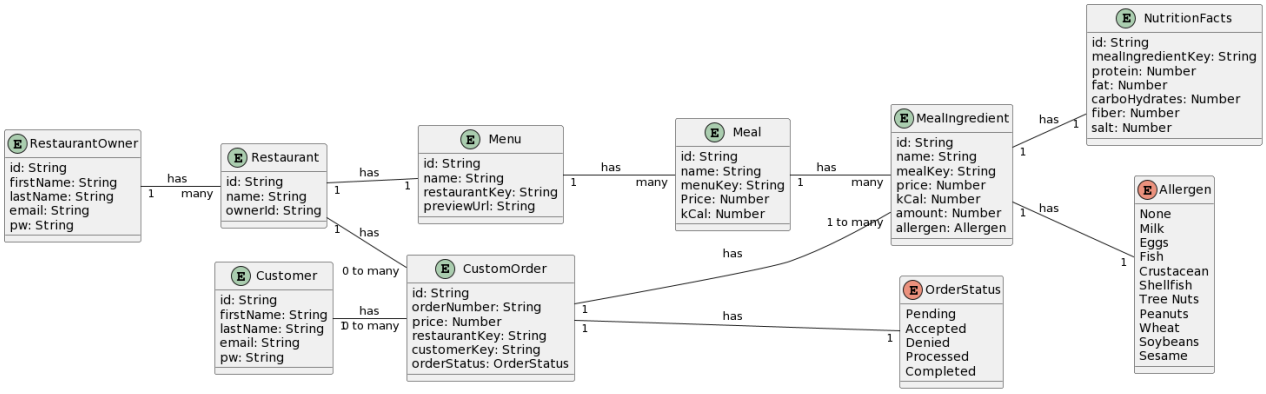
(figure 1 - LowKcal High Level Architecture Diagram)

The architecture diagram outlines the interaction between different components of the LowKcal system. The components of the system are the following:

1. **Users and Interfaces**:
    - **Customer**: Uses a smartphone web browser to access the LowKcal Progressive Web App (PWA).
    - **System Admin**: Uses a desktop web browser to manage the system via the LowKcal PWA.
    - **Restaurant Staff**: Uses a desktop web browser on a kitchen terminal to interact with the LowKcal PWA.
2. **Client-Side Components**:
    - **Smartphone Web Browser**: For customers to browse menus and place orders.
    - **Desktop Web Browser**: For system admins to manage restaurants, menus, and orders.
    - **Kitchen Terminal (Desktop Web Browser)**: For restaurant staff to view and process incoming orders.
3. **Server-Side Components**:
    - **Monolithic System (Server)**: The core backend of the application that includes:
        - **LowKcal PWA (Progressive Web App)**: Serves the user interface for different user roles (customers, admins, and staff).
        - **LowKcal API**: Handles the business logic and acts as a service between the PWA and the database. Handles all the data requests from the PWA.

- **LowKcal DB (SQLite)**: A database that stores all the data for the system(menus, orders, user details, etc.).

b. Entity-Relationship Model



(figure 2 - LowKcal Entity-Relationship Model Diagram)

The database model outlines the structure of the data and the relationships between different entities.

1. **Entities**:
    - **RestaurantOwner**: Represents the owner of one or more restaurants.
        - Attributes: `id`, `firstName`, `lastName`, `email`, `password`
    - **Restaurant**: Represents a restaurant in the system.
        - Attributes: `id`, `name`, `ownerId` (foreign key to `RestaurantOwner`)
    - **Menu**: Represents a menu for a specific restaurant.
        - Attributes: `id`, `name`, `restaurantKey` (foreign key to `Restaurant`), `previewUrl`
    - **Meal**: Represents a meal within a menu.
        - Attributes: `id`, `name`, `menuKey` (foreign key to `Menu`), `price`, `kCal`
    - **Customer**: Represents a customer using the system.
        - Attributes: `id`, `firstName`, `lastName`, `email`, `password`
    - **OrderStatus**: Represents the status of an order.
        - Attributes: `Pending`, `Accepted`, `Delivered`, `Denied`
    - **CustomOrder**: Represents an order placed by a customer.
        - Attributes: `id`, `orderNumber`, `price`, `restaurantKey` (foreign key to `Restaurant`), `customerKey` (foreign key to `Customer`), `orderStatus`
    - **Allergen**: Represents potential allergens in meal ingredients.
        - Attributes: `None`, `Milk`, `Eggs`, `Fish`, `Crustacean`, `Shellfish`, `Tree Nuts`, `Peanuts`, `Wheat`, `Soybeans`, `Sesame`
    - **MealIngredient**: Represents an ingredient in a meal.
        - Attributes: `id`, `name`, `mealKey` (foreign key to `Meal`), `price`, `kCal`, `amount`, `allergen` (foreign key to `Allergen`)

- **NutritionFacts**: Represents the nutritional information for a meal ingredient.
  - Attributes: `id`, `mealIngredientKey` (foreign key to `MealIngredient`), `protein`, `fat`, `carbohydrates`, `fiber`, `salt`

c. System Description

1. **Restaurant Management:**
   - The System Admin and RestaurantOwner use the LowKcal PWA on a desktop to manage restaurants and menus.
   - The System Admin can add new restaurants and the RestaurantOwner can create and update his menus, and input detailed meal information, including ingredients and nutritional facts.
2. **Customer Interaction:**
   - Customers use the LowKcal PWA on their smartphones to browse restaurant menus, customize their meal choices to fit their nutritional goals, and place orders.
   - The orders include selected meals and any customization, and the nutritional information is displayed based on the meal ingredients.
3. **Order Processing:**
   - The Restaurant Staff uses the kitchen terminal to view incoming orders and update their status (pending, accepted, delivered, denied).
   - The status of each order is tracked in the system and visible to both the customer and the restaurant staff.
4. **Data Management:**
   - All data (users, orders, restaurants, menus, meals, ingredients, nutritional facts) is stored in the SQLite database.
   - The LowKcal API handles the CRUD operations (Create, Read, Update, Delete) between the web app and the database, ensuring data consistency and integrity.

This architecture ensures a seamless experience for all users (customers, restaurant owners, system admins, and restaurant staff) while maintaining robust data management and processing capabilities on the backend.

2. Technologies

Frontend/PWA:

- **React:**
    - **Description:** React is a popular JavaScript library for building user interfaces, particularly single-page applications (SPAs). It allows for the creation of reusable UI components.
    - **Usage in LowKcal:** Used to build the Progressive Web App (PWA) that serves the user interface for customers, system admins, and restaurant staff. React's component-based architecture makes it easy to manage and develop the UI.
- **TypeScript:**
    - **Description:** TypeScript is a statically typed superset of JavaScript that adds optional type annotations.
    - **Usage in LowKcal:** Enhances code quality and maintainability by providing type safety and better development tools (e.g., autocompletion, type checking).
- **Vite:**
    - **Description:** Vite is a modern build tool that offers a fast development environment and optimized production builds.
    - **Usage in LowKcal:** Used for building and serving the React application. Vite's performance optimizations and rapid build times enhance the development workflow.
- **Tailwind CSS:**
    - **Description:** Tailwind CSS is a utility-first CSS framework that allows developers to build custom designs quickly.
    - **Usage in LowKcal:** Used for styling the PWA, providing a consistent and responsive design with minimal custom CSS. Tailwind's utility classes make it easy to apply styles directly within the HTML.

Backend:

- **Django:**
    - **Description:** Django is a high-level Python web framework that encourages rapid development and clean, pragmatic design.
    - **Usage in LowKcal:** Django is used to build the LowKcal API, which handles all the business logic and data processing. Django's built-in features like ORM (Object-Relational Mapping), authentication, and admin interface streamline backend development.

Database:

- **SQLite:**
    - **Description:** SQLite is a lightweight, disk-based database that doesn't require a separate server process. It's self-contained, serverless, and zero-configuration.

- **Usage in LowKcal:** SQLite serves as the database for storing all application data, including user information, restaurants, menus, meals, orders, and nutritional facts. Its simplicity and ease of setup make it ideal for the initial phases of the project.

How These Technologies Integrate into the Architecture

1. **Frontend/PWA (React + TypeScript + Vite + Tailwind):**
   - **Components:** The frontend application consists of React components written in TypeScript, styled using Tailwind CSS.
   - **Build and Development:** Vite is used to bundle and serve the application during development and to produce optimized builds for production.
   - **Functionality:** Provides interfaces for customers to browse and order meals, for restaurant staff to manage orders, and for system admins to manage the overall system.
2. **Backend (Django):**
   - **API Development:** Django is used to develop RESTful APIs that the frontend communicates with. These APIs handle CRUD operations for users, restaurants, menus, meals, and orders.
   - **Business Logic:** All business rules and data processing are managed within the Django application.
   - **Security:** Django's robust authentication and authorization mechanisms ensure secure access to the application's resources.
3. **Database (SQLite):**
   - **Data Storage:** All application data is stored in an SQLite database, which is accessed through Django's ORM.
   - **Data Management:** Django migrations are used to manage database schema changes, ensuring data integrity and consistency.

Benefits of This Technology Choice

- **React + TypeScript:**
  - Provides a modern, efficient development environment with type safety.
  - React's component-based architecture and TypeScript's static typing enhance code maintainability and scalability.
- **Vite:**
  - Offers a fast, optimized development experience with minimal configuration.
  - Provides rapid feedback during development, improving developer productivity.
- **Tailwind CSS:**
  - Enables rapid UI development with utility-first styling, ensuring a consistent design system.
  - Reduces the need for custom CSS, making the codebase cleaner and easier to manage.
- **Django:**
  - Simplifies backend development with its powerful features and "batteries-included" philosophy.

    - Django’s ORM abstracts database interactions, reducing the amount of SQL code required.
- **SQLite:**
    - Ideal for lightweight, serverless database management, perfect for the initial stages of the project.
    - Simplifies setup and maintenance, with no need for a separate database server.

Overall, this technology stack is well-suited for the LowKcal project, providing a robust, scalable, and maintainable architecture that supports the application's requirements and goals.

3. Conclusion

The architecture and technology stack chosen for the LowKcal project demonstrate a thoughtful balance between modern, efficient development practices and robust, scalable design. Here’s a comprehensive reflection on the architecture and technology choices:

Strengths of the Architecture

1. **Scalability**:
    - The use of **React** for the frontend allows for the development of a highly interactive and responsive Progressive Web App (PWA). Its component-based architecture makes it easy to extend and scale the user interface as the application grows.
    - **Django** on the backend provides a powerful framework that can handle increasing complexity and user loads, thanks to its modular and extensible nature.
2. **Maintainability**:
    - **TypeScript** enhances the maintainability of the codebase by providing static typing, which helps catch errors early in the development process and makes the code easier to understand and refactor.
    - **Tailwind CSS** facilitates a consistent design system with its utility-first approach, reducing the need for extensive custom styling and simplifying the maintenance of the user interface.
3. **Performance**:
    - **Vite** offers a fast development environment and optimized production builds, ensuring that the development process is efficient and the final application performs well for end users.
    - **SQLite** provides a lightweight, serverless database solution that is easy to set up and requires minimal maintenance, while still being capable of handling the data needs of the application in its early stages.
4. **Development Efficiency**:

- **Django**'s “batteries-included” approach means that many common tasks (like authentication, database migrations, and admin interfaces) are handled out-of-the-box, allowing developers to focus on the unique aspects of the LowKcal project.
- **React** and **Vite** enable rapid development and iteration on the frontend, with hot module replacement and fast refresh features that speed up the feedback loop for developers.

Areas for Future Consideration

1. **Database Scalability**:
   - While **SQLite** is an excellent choice for the initial phases of the project, as the user base and data volume grow, transitioning to a more robust database system (such as PostgreSQL) may be necessary to ensure continued performance and scalability.
2. **Microservices Architecture**:
   - As the application evolves, there might be a need to transition from a monolithic architecture to a microservices architecture. This would involve breaking down the Django monolith into smaller, independently deployable services to improve scalability, fault isolation, and development agility.
3. **Enhanced Security**:
   - Ensuring robust security measures throughout the application will be crucial, especially as it handles sensitive user data and payment information. Regular security audits, penetration testing, and adherence to best security practices will be essential.
4. **Advanced Features and Customizations**:
   - The architecture should allow for the easy addition of advanced features, such as personalized meal recommendations, integration with fitness trackers, or AI-driven nutritional advice, to enhance the value proposition of the LowKcal app.

Overall Reflection

The LowKcal project's architecture and technology choices position it well for success. The use of modern technologies and frameworks provides a solid foundation for building a user-friendly, efficient, and scalable application. By leveraging React, TypeScript, Vite, Tailwind, Django, and SQLite, the development team can ensure a high-quality product that meets user needs while remaining adaptable to future requirements and growth.

This architectural setup not only supports the current goals of enabling restaurants to offer healthy, customizable meals but also lays the groundwork for future enhancements and scalability. With a clear path for potential upgrades and improvements, the LowKcal project is well-prepared to evolve and succeed in the competitive market of health-focused food delivery apps.