Design Software Architecture

A. High-Level Architecture Design

LifeSync utilizes a **Layered (N-Tier) Client-Server Architecture**. This architectural style was selected to separate concerns effectively, ensuring that the presentation logic (React), business logic (Node.js), and data management (PostgreSQL) remain distinct.

Given the constraints of a 10-week academic timeline and the goal of releasing a standalone Minimum Viable Product (MVP), the system avoids over-engineering (e.g., complex microservices) in favor of a robust, monolithic backend that communicates with a decoupled Single Page Application (SPA) frontend via RESTful APIs.

Key Architectural Characteristics:

- **Modular Design:** The system integrates specific Object-Oriented Design (OOD) patterns —specifically **Facade**, **Strategy**, **Observer**, and **Builder**—within the business logic layer to manage complexity.

- **External AI Integration:** The architecture is specifically designed to handle external dependencies (OpenAI GPT-4o) through a dedicated service abstraction layer to prevent API coupling from affecting the core business logic.

- **Security:** The design incorporates a security layer for authentication (hashing, session management) ensuring data protection for user health metrics.

B. Architectural Components and Diagram

The system is structured into four primary logical layers. Below is the structured description of these components followed by an architectural diagram.

1. Presentation Layer (Client Side)

- **Technology:** React (Web Application).
- **Responsibility:** Handles user interactions, displays the dashboard, renders the onboarding survey, and visualizes the weekly plans. It communicates with the backend via HTTP requests.
- **Key Components:** Authentication Forms, Dashboard UI, Survey Interface.

2. Application / Business Logic Layer (Server Side)

- **Technology:** Node.js with Express.js.
- **Responsibility:** This is the core of the system. It processes incoming requests, executes domain logic (e.g., determining fitness levels), and orchestrates data flow.
- **Design Pattern Implementation:**
    - *Notification Engine:* Uses the **Observer Pattern** to trigger time-based reminders.
    - *Level Classifier:* Uses the **Strategy Pattern** to switch algorithms for determining fitness levels (Beginner/Intermediate/Advanced).
    - *Profile Builder:* Uses the **Builder Pattern** to construct complex user profile objects from survey data.

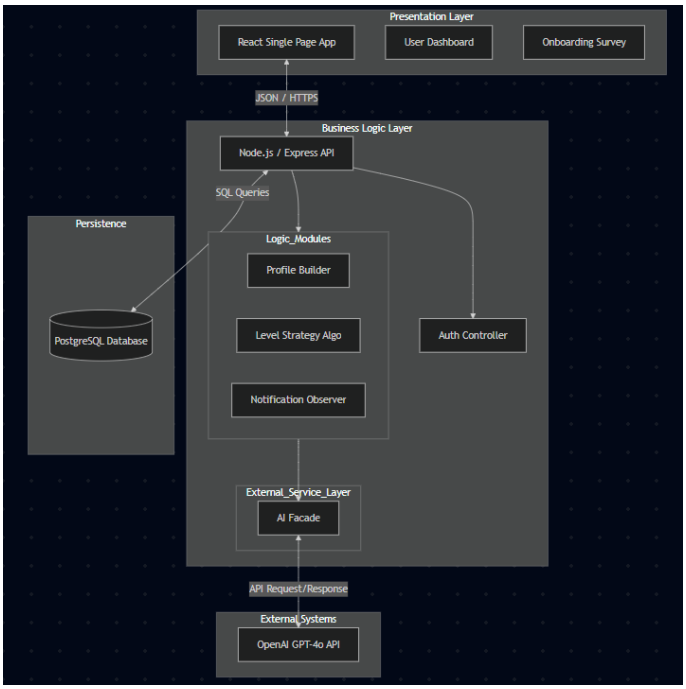
3. Service Integration Layer (Facade)

- **Technology:** OpenAI API Adapter.
- **Responsibility:** Acts as a gateway to the GPT-4o Large Language Model.
- **Design Pattern Implementation:** Uses the **Facade Pattern** to hide the complexity of the OpenAI API (prompt engineering, token management) from the main application logic.

4. Data Persistence Layer

- **Technology:** PostgreSQL.
- **Responsibility:** Stores user profiles, authentication credentials (hashed), generated routines, and progress tracking logs.

System Architecture Diagram:



C. Architecture Facilitation of Use Cases

The proposed architecture directly supports the realization of the defined use cases through specific structural workflows:

1. Facilitating "Complete Survey" & "Determine User Level"

- **Workflow:** The *Presentation Layer* collects data via the survey. The *Business Layer* utilizes the **Builder Pattern** to standardize this data into a User Profile object. Immediately upon submission, the **Strategy Pattern** component analyzes the data to assign a fitness classification (Beginner/Intermediate/Advanced).
- **Architectural Benefit:** This separation allows the developers to modify the logic for determining fitness levels (e.g., changing BMI thresholds) in the backend without requiring changes to the frontend code or the database schema.

2. Facilitating "Generate Personalized Routine"

- **Workflow:** When a plan is requested, the *Business Layer* calls the *Service Layer*. The **Facade Pattern** encapsulates the complex prompt engineering required for GPT-4o. The Facade sends the request, sanitizes the AI response, and passes it back to be stored in the *Data Layer*.
- **Architectural Benefit:** This ensures the "30-second delivery" performance requirement is managed efficiently. If the AI provider changes or the API version updates, only the Facade needs modification, ensuring system stability.

3. Facilitating "Receive Reminders"

- **Workflow:** The *Business Layer* contains a scheduler (Notification Engine) utilizing the **Observer Pattern**. It constantly checks the *Data Layer* for scheduled workout times and triggers notifications to the *Presentation Layer*.
- **Architectural Benefit:** Decoupling the notification logic means the system can scale to add different types of notifications (email, push) later without rewriting the core routine generation logic.

4. Facilitating "Track Progress"

- **Workflow:** The *Presentation Layer* sends completion status updates to the API. The *Data Layer* persists these updates, allowing the Dashboard to re-query and visualize adherence trends over time.
- **Architectural Benefit:** Centralized data management in PostgreSQL ensures data integrity and persistence, allowing users to switch devices (Mobile/Desktop) while maintaining a synchronized view of their progress.