Architectural Requirements Document

Gym Manager Project - Rome was Built in a day

June 2025

Architectural Design Strategy

Our design strategy is **driven by quality requirements**, aligning core design decisions to explicitly meet the project's goals around usability, security, reliability, portability, and maintainability. This approach ensures that the final architecture satisfies key non-functional requirements essential for long-term user satisfaction, system robustness, and developer productivity.

Architectural Strategies

We adopted a hybrid of modern architectural styles best suited to our problem domain:

- **Layered Architecture** for the backend (Express + Drizzle ORM): Separates concerns between routing, business logic, and persistence.
- **Component-Based Architecture** for frontend (React + React Native): Encourages reuse and modularity.
- **Planned Event-Driven Extensions**: For future real-time features such as leaderboard updates and notifications.

This combination provides flexibility, high maintainability, and scalability for both current and future requirements.

3.6.3 Architectural Quality Requirements

Quality Requirement	**Strategy**
Usability (Highest Priority)	• Intuitive UI and timeline for class booking on both web and mobile. • Cancel/reschedule classes, and switch between roles (coach vs member). • Presets/templates for workouts (coaches) and schedules (managers). • Profile-based suggestions, loading indicators, and validation messages.
Security	• JWT authentication and refresh tokens; bcrypt-hashed passwords. • Role-Based Access Control with requireRole() middleware. • Input validation and protected routes. • HTTPS/TLS enforced during transport; hashed credentials at rest.
Reliability / Availability	• `/healthz` route for database/Node uptime check (used by GitHub Actions). • Global error handler prevents worker crashes and returns friendly errors. • PM2 cluster mode with auto-restarts for fault-tolerant multi-core operation. • Docker auto-restart and optional Postgres replication.

Portability	• Shared logic across web/mobile using React Native and Expo. • Axios abstraction to ensure consistent API requests on any platform. • Responsive layouts for multiple screen sizes and browsers.
Maintainability	• Modular monorepo with apps/, services/, packages/, and infra/ folders. • Feature-sliced and layered backend architecture (routing, logic, DB). • Shared libraries for API and ORM logic. • TypeScript with strict typing and ESLint/Prettier for style. • Swagger for API docs; CI/CD tests to prevent regressions.

Architectural Design and Pattern

The system follows a layered architecture:

- **Presentation Layer**: React and React Native clients communicate via Axios.
- **Application Layer**: Express routes, middleware, and controllers.
- **Domain Layer**: Business logic and Drizzle ORM models.
- **Infrastructure Layer**: PostgreSQL DB

Frontend uses a component-based pattern. Future plans include event-driven patterns for class reminders and leaderboard updates.

Architectural Constraints

- Must support both mobile and web interfaces with shared functionality.
- Must operate in limited-infrastructure environments
- Must be usable by gym members with minimal training (intuitive UX).
- Must enforce role-based access control for coaches, managers, and members.
- Must allow scaling for high amounts of requests per second under moderate load.

Technology Choices

Backend Framework:

- **Express.js (chosen)**: Minimal, fast, flexible with middleware support.
- Nest.js: Heavyweight, more opinionated and learning curve.
- Fastify: Slightly faster, but smaller ecosystem.

Database ORM:

- **Drizzle ORM (chosen)**: Type-safe schema with excellent TS support.
- Prisma: Great developer experience but requires generation steps.
- TypeORM: Older and less type-safe.

Frontend Framework:

- **React + React Native (chosen)**: Cross-platform UI with shared logic.
- Flutter: Strong mobile support, limited web ecosystem.
- Angular: Complex for small teams, heavier initial setup.

Testing:

- **Jest + Supertest (chosen)**: Great integration/unit testing support.
- Vitest: Modern and fast, but not as mature.
- Mocha/Chai: Manual setup and more boilerplate.

CI/CD and Monitoring:

- **GitHub Actions + Uptime badge**: Monitors /healthz and runs automated tests.
- Docker: Ensures consistent environments across dev/prod.