






ARCHITECTURAL REQUIREMENTS AND DESIGN



Rome was Built
in a Day



HIIT GYM MANAGER GOOD X SOFTWARE

-  Vansh Sood (u23534402)
-  Denis Woolley (u23528860)
-  Jared Hürlimann (u23543932)
-  Jason Mayo (u23587572)
-  Amadeus Fidos (u22526162)

1. 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.

2. Architectural Patterns

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

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

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

3. Architectural Strategies

Quality Requirement	Strategy
Usability (Highest Priority)	<ul style="list-style-type: none">• 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	<ul style="list-style-type: none">• JWT authentication and refresh tokens; bcrypt-hashed passwords.• Role-Based Access Control with middleware.• Input validation and protected routes.• HTTPS/TLS enforced during transport; hashed credentials at rest.
Reliability / Availability	<ul style="list-style-type: none">• /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 multicore operation.• Docker auto-restart and optional Postgres replication.
Portability	<ul style="list-style-type: none">• Shared logic across web/mobile using React Native and Expo (component-based arch)• Axios abstraction to ensure consistent API requests on any platform.• Responsive layouts for multiple screen sizes and browsers
Maintainability	<ul style="list-style-type: none">• 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.

4. Architectural Design and Pattern

4.1. Clean Architecture

The backend adopts a **Clean Architecture** approach (a variation of layered architecture) in which all dependencies point inward toward the domain layer. This style is particularly well-suited for systems with complex business domains, where long-term maintainability and domain-focused development are critical. By keeping the **core domain logic independent** of external frameworks, databases, or UI technologies, the system ensures that business rules remain stable, while infrastructure and presentation layers can evolve or be replaced with minimal impact. This promotes **security** and **maintainability** by clearly separating concerns into distinct layers.

1. **Domain Layer:** Core business entities and contracts
 - Entities: Business objects and data transfer objects
 - Interfaces: Service contracts and repository contracts
 - Dependencies: None (pure business logic)
2. **Repository Layer:** Data access and persistence
 - Repositories: Database operations and data mapping
 - Dependencies: Domain entities and database infrastructure
3. **Service Layer:** Business logic and orchestration
 - Services: Business rules, validation, and workflow orchestration
 - Dependencies: Repositories and infrastructure services
4. **Controller Layer:** HTTP request/response handling
 - Controllers: HTTP concerns, input validation, response formatting
 - Dependencies: Services only
5. **Presentation Layer:** Client-facing UI for end users.
 - Clients: Web (NextJS) and Mobile (React Native).
 - Communication: Interact with the backend through REST APIs (via Axios).
6. **Additional: Infrastructure:** External concerns and technical implementation
 - Auth Services: JWT and password handling
 - Middleware: Authentication middleware
 - Container: Dependency injection container
 - Dependencies: External libraries and services
 - Database: PostgreSQL as the persistence layer.

4.2. Component-Based Architecture

The frontend adopts a **component-based architecture** to ensure **modularity and reusability** across both web (NextJS) and mobile (React Native) clients. This approach ensures better **usability**, **portability** and **maintainability** by a consistent user experience and simplifying development (since both sides follow component based with react).

4.3. Event-Driven Architecture

We employ an **event-driven architecture (EDA)** to handle asynchronous updates such as notifications and leaderboard changes.

- **Notifications:** When a class is scheduled/ updated or when an announcement is made, an event is emitted, which triggers notifications to enrolled users.
- **Leaderboard Updates:** Whenever relevant metrics change (e.g., points, achievements), an event triggers recalculation and broadcasting of the updated leaderboard.

The system uses **Supabase Realtime** for live updates, ensuring that notifications and live class changes propagate instantly to clients. Additionally, **internal event emitters** decouple system components, allowing different modules to respond to events independently and asynchronously. This design promotes **scalability, maintainability, availability and responsiveness (improving usability)**, as each component only reacts to the events it cares about.

5. 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.

6. 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.

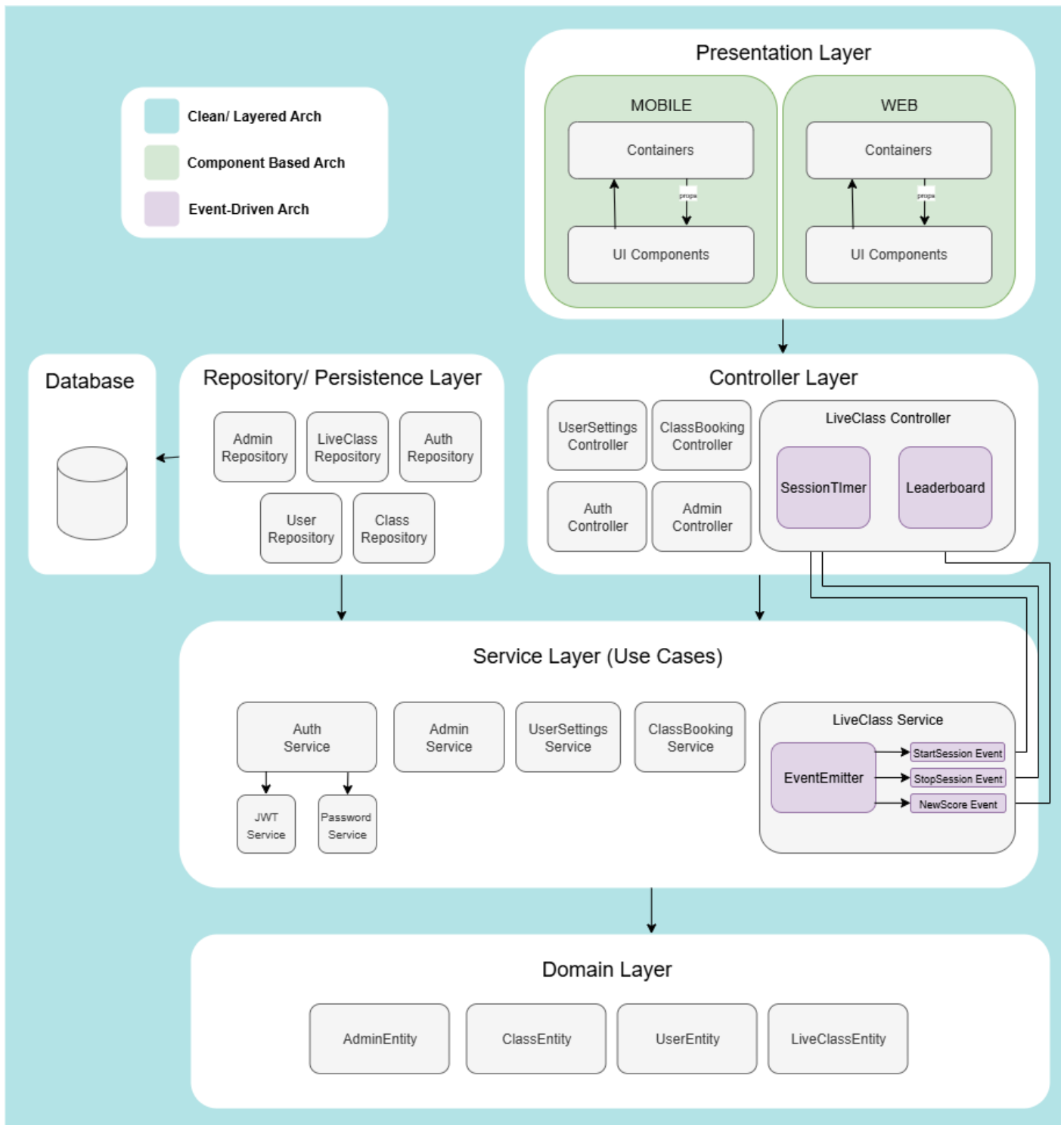
Testing:

- 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.

7. Architectural Diagram



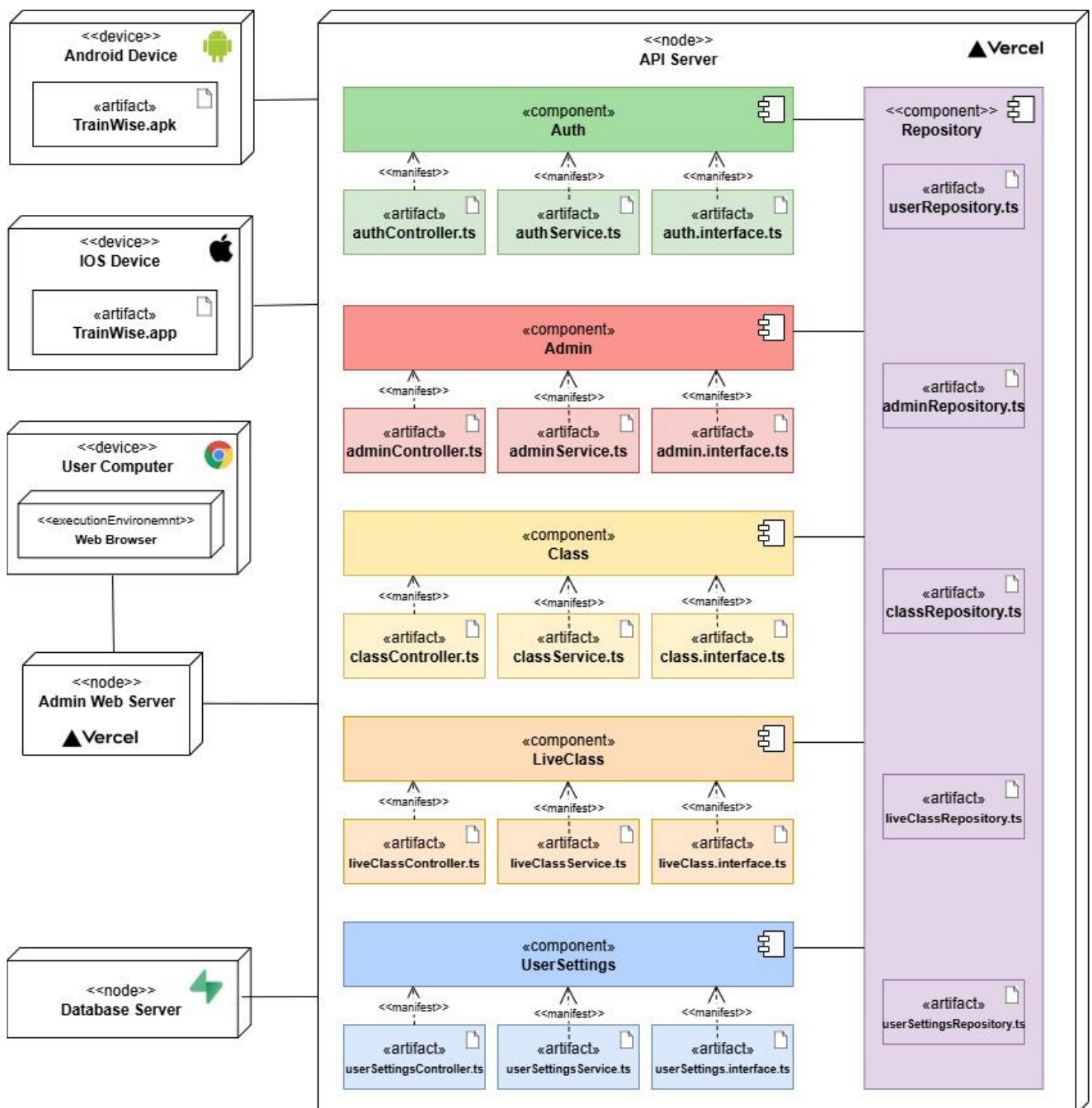
8. Deployment Diagram

8.1. Target Environment

The system is deployed in a **hybrid cloud environment**:

- Mobile clients (iOS and Android) run on user devices.
- Web client (Admin portal) is hosted on **Vercel**.
- Backend API is hosted separately on **Vercel** as a serverless Node.js service.
- Database is hosted on **Supabase**, a managed cloud PostgreSQL service.

This architecture leverages cloud services for scalability and reliability, while mobile apps run natively on user devices.



8.2. Deployment Topology

The system follows a **multi-tier, hybrid-cloud topology**, implemented as a **serverless modular monolith** with **edge CDN delivery** (Vercel) and a **managed cloud database** (Supabase).

Presentation Tier (Clients + CDN)

- The **admin web frontend** is hosted on **Vercel**, which serves static Next.js build artifacts through its **edge CDN** for low-latency delivery to browsers.
- The **mobile apps** run on user devices (iOS via TestFlight/App Store, Android via APK/Play Store).
- This tier leverages edge caching and client execution for scalability and responsiveness.

Application Tier (Business Logic)

- The **API** is deployed on **Vercel** as serverless Node.js functions. These functions auto-scale and are fully managed, eliminating the need for VM or container orchestration.
- Although the system has clear logical modules (Auth, Class, Admin, LiveClass, etc.), they are deployed together as a **single service**, making the backend a **serverless modular monolith** rather than a microservices mesh.

Data Tier (Persistence)

- **Supabase (managed PostgreSQL)** serves as the persistent data store.
- Supabase is a fully managed cloud service, providing database hosting, scaling, and connectivity over TLS.

Network & Connectivity

- Clients ↔ **HTTPS** ↔ Vercel (edge CDN for static assets + serverless functions for API requests).
- Vercel API ↔ **TLS** ↔ Supabase (database connection).
- Static frontend files are delivered from the edge, while dynamic requests are routed to serverless functions.

Overall, the deployment is best described as a **three-tier architecture (Presentation / Application / Data)**, realized using **serverless compute, CDN edge distribution, and managed cloud services**.

The design supports scalability, reliability, and maintainability by combining globally distributed delivery (Vercel Edge) with elastic serverless functions and a robust managed database.

8.2.1. How this topology maps to your quality requirements

- **Scalability:** Vercel auto-scales serverless functions and CDN; Supabase scales DB resources.
- **Reliability:** Edge caching + managed services provide redundancy, low latency, and high uptime.
- **Maintainability:** Separation of frontend (Vercel), backend (Vercel), and DB (Supabase) with modular code improves maintainability.
- **Cost/Effort:** Serverless reduces ops burden and is cost-efficient at smaller loads; Supabase lowers DB ops overhead but watch for cold starts and DB limits.