Mekelle University

Department of Software Engineering

Software design document(SDD) for

Tibeb: skill marketplace

Submitted to:
Instructor G.Michael

Submitted by:
Nahom Mulualem
Samuel Embaye
Teame G/her

Contents

1. Introduction

1.1 Purpose

The purpose of this Software Design Document (SDD) is to provide a comprehensive architectural and low-level design specification for the Tibeb: Skill Marketplace platform. This document translates the functional and non-functional requirements defined in the Software Requirements Specification (SRS) into a concrete technical implementation plan.

It is intended for the following primary audiences:

- **Software Engineers** : To understand the system decomposition, API contracts, and data structures required for implementation.
- **System Architects:** To validate design patterns, scalability strategies, and security protocols.
- **Quality Assurance (QA) Team**: To design integration test plans based on component interfaces and data flows.

1.2 Scope

System Overview:
Tibeb is a centralized digital intermediary platform designed to connect service seekers (Clients) with service providers (Freelancers) in the Ethiopian market. The system functions as a Modular Monolith application, orchestrated via a RESTful API and a real-time event bus.

Design Boundaries:

- **Architectural Pattern:** Implementation of the Model-View-Controller (MVC) pattern within a Layered Architecture.
- **Subsystems**: Detailed design of Identity Management (RBAC), Job Lifecycle Engine, Proposal Transaction Logic, and Real-Time Messaging.
- **Data Design**: Third Normal Form database schema design using PostgreSQL.

1.3 Definitions, Acronyms, and Abbreviations

- API: Application Programming Interface.
- DTO: Data Transfer Object; simple objects used to transfer data between subsystems without business logic.
- ERD: Entity-Relationship Diagram.
- IEEE 1016: IEEE standard for software design descriptions
- JWT: JSON Web Token; RFC 7519 standard for stateless authentication.
- MVC: Model-View-Controller; a software design pattern separating internal representation from information presentation.

- ORM: Object-Relational Mapping; specifically Prisma, used to bridge the Node.js runtime and PostgreSQL database.
- RBAC: Role-Based Access Control.
- SDD: Software Design Document
- SSR: Server-Side Rendering (Next.js strategy).
- WebSocket: Communication protocol providing full-duplex communication channels over a single TCP connection.

1.4 References

The following documents and standards serve as the foundation for this design:

1. IEEE Std 1016-2009, IEEE Standard for Information Technology—Systems Design—Software Design Descriptions.
2. Prisma, Prisma Schema Reference and Migration Guide.
3. OWASP Foundation, Secure Coding Practices Quick Reference Guide.

1.5 Overview

The remainder of this document is organized as follows:

- Section 2 identifies the design stakeholders and key architectural concerns (Security, Performance).
- Section 3 provides the high-level system context and external constraints.
- Section 4 details the high-level architectural design, including patterns and decomposition.
- Section 5 dives into detailed component design, interface specifications, and algorithm logic.
- Section 6 describes dynamic behavior using sequence and state diagrams.
- Section 7 outlines the physical deployment architecture.
- Sections 8-14 cover operational concerns including logging, configuration, quality attributes, compliance, and appendices.

2. Software Design Stakeholders and Concerns

2.1 Stakeholders

The architecture is designed to address the specific needs of the following stakeholder groups:

1. **End Users (Clients & Freelancers):**
    - **Role**: The primary operators of the system.
    - **Interest**: High availability, intuitive User Interface (UI), data privacy, and reliable message delivery.
2. **Development Team:**
    - **Role**: Implementers of the design and maintain the system.
3. **Academic Advisor:**
    - **Role***:* Project evaluator and standard bearer.
    - **Interest***:* Compliance to IEEE software engineering standards, architectural correctness, and feasibility of the MVP scope within the semester timeline.

2.2 Design Concerns

The following quality attributes represent the Architectural Drivers that have influenced the selection of the Modular Monolith pattern and the specific technology stack

- **Security (Critical):**
    - **Concern**: The platform handles user identity and simulates financial contracts.
    - **Design Response**: Implementation of a Defense in Depth strategy, including Role-Based Access Control (RBAC) at the middleware level.
- **Maintainability & Modularity (High):**
    - **Concern**: As a student project with multiple contributors, the codebase must be easy to understand and modify without causing regression errors.
    - **Design Response**: Adoption of a Layered Architecture (Controller-Service-Repository) to decouple business logic from HTTP transport and database access layers.
- **Performance (High):**
    - **Concern**: Real-time messaging and job feed rendering require low latency.
    - **Design Response**: Utilization of WebSockets (Socket.io) for instant message delivery and Server-Side Rendering (SSR) in Next.js to ensure fast First Contentful Paint for job listings.
- **Scalability (Medium):**
    - **Concern**: The system must support concurrent users during the pilot phase and allow for future growth.
    - **Design Response**: Design of Stateless Backend Services (REST API) using JWTs

3. System Overview

3.1 System Context

The Tibeb: Skill Marketplace operates as a centralized Digital Service Intermediary. It functions within a distributed web environment, acting as the orchestration layer between end-user devices and data persistence services.

The system interacts with the following external entities:

- **User Agents (Browsers)**: The primary entry point for Clients and Freelancers, communicating via HTTPS and Secure WebSockets (WSS).
- **Email Service Provider**: An external relay used for transactional notifications (Welcome emails, Password Resets).

System Boundary:
The system boundary encapsulates the Presentation Layer (Next.js), Application Layer (Node.js API), and Persistence Layer (PostgreSQL). All business logic, data validation, and session state management occur strictly within this boundary.

3.2 System Constraints

The architecture is bounded by the following immutable constraints derived from the SRS and the deployment environment:

1. **Software Constraints**:
   - Runtime Environment: The backend relies strictly on the Node.js asynchronous runtime.
   - Database Strictness: The system requires PostgreSQL 15+ to support specific JSONB indexing and UUID generation features used in the schema.
   - Payload Limits: To prevent Denial of Service attacks on the application layer, the API Gateway enforces a hard limit of 100KB for JSON payloads and 5MB for file uploads.
2. **Hardware/Infrastructure Constraints**:
   - Compute Resources: The design must operate efficiently within a small container environment for the MVP pilot.
   - Network Latency: The architecture assumes a standard 4G network connection; however, aggressive caching strategies are implemented to mitigate intermittent connectivity common in the target deployment region.

3.3 Assumptions and Dependencies

Assumptions:

- Client Capabilities: It is assumed that 95% of the user base accesses the platform via modern, JavaScript-enabled web browsers.

- Connectivity: The system assumes an Always-On connectivity model, no offline synchronization logic is implemented for the backend API.

Dependencies:

- Package Registry: The build pipeline depends on the availability of the public NPM (Node Package Manager) registry for dependency resolution.
- Cloud Infrastructure: The MVP deployment relies on Platform-as-a-Service providers (e.g., Vercel for frontend, Render/Railway for backend).
- Version Control: The Continuous Integration (CI) workflow is dependent on GitHub Actions availability.

4. Architectural Design

4.1 Architectural Overview

The Tibeb Skill Marketplace is architected as a Three-Tier Web Application following the Client-Server model. The system is physically distributed across three distinct logic tiers:

1. Presentation Tier (Client): A Single Page Application responsible for user interaction, state management, and view rendering. It runs in the user's browser and on the Edge for initial loads.
2. Logic Tier (Server): An Application Server acting as the centralized API Gateway and business logic processor. It is stateless and handles all data processing, authentication, and authorization.
3. Data Tier (Persistence): A Relational Database Management System that stores all persistent data.

4.2 Architectural Patterns and Styles

The system implements the following architectural patterns:

4.2.1 Modular Monolith:

The backend is deployed as a single executable unit, but the internal code structure is strictly divided into isolated domain modules.

- Application: This approach simplifies the DevOps pipeline (single deployment) while preventing spaghetti code, allowing for future extraction into Microservices if specific domains (e.g., Chat) require independent scaling.

4.2.2 Layered Architecture:

The backend codebase is organized into three horizontal layers:

1. Controller Layer: Handles HTTP request parsing, input validation, and response formatting.
2. Service Layer: Contains the core business logic, rules, and transaction boundaries.
3. Data Access Layer (DAL): Abstractions using Prisma ORM to interact with the database, ensuring business logic is not coupled to raw SQL.

4.2.3 Role-Based Access Control(RBAC)

- User role determines accessible use cases and operations which improves security.

4.3 System Decomposition

The system is decomposed into the following high-level subsystems:

1. Authentication and authorization:
   - Responsibility: Manages User Registration, JWT Issuance/Verification, and RBAC Policy Enforcement.
2. Job Orchestration:
   - Responsibility: Handles the lifecycle of Job entities (Create, Update, Close) and executes the Search/Filtering logic.
3. Proposal & Transaction:
   - Responsibility: Manages Bids, enforces Idempotency (one bid per job), and handles the Logical Escrow state machine.
4. Real-Time Communication:
   - Responsibility: Manages WebSocket connections, message routing, and chat history persistence.

4.4 Design Rationale

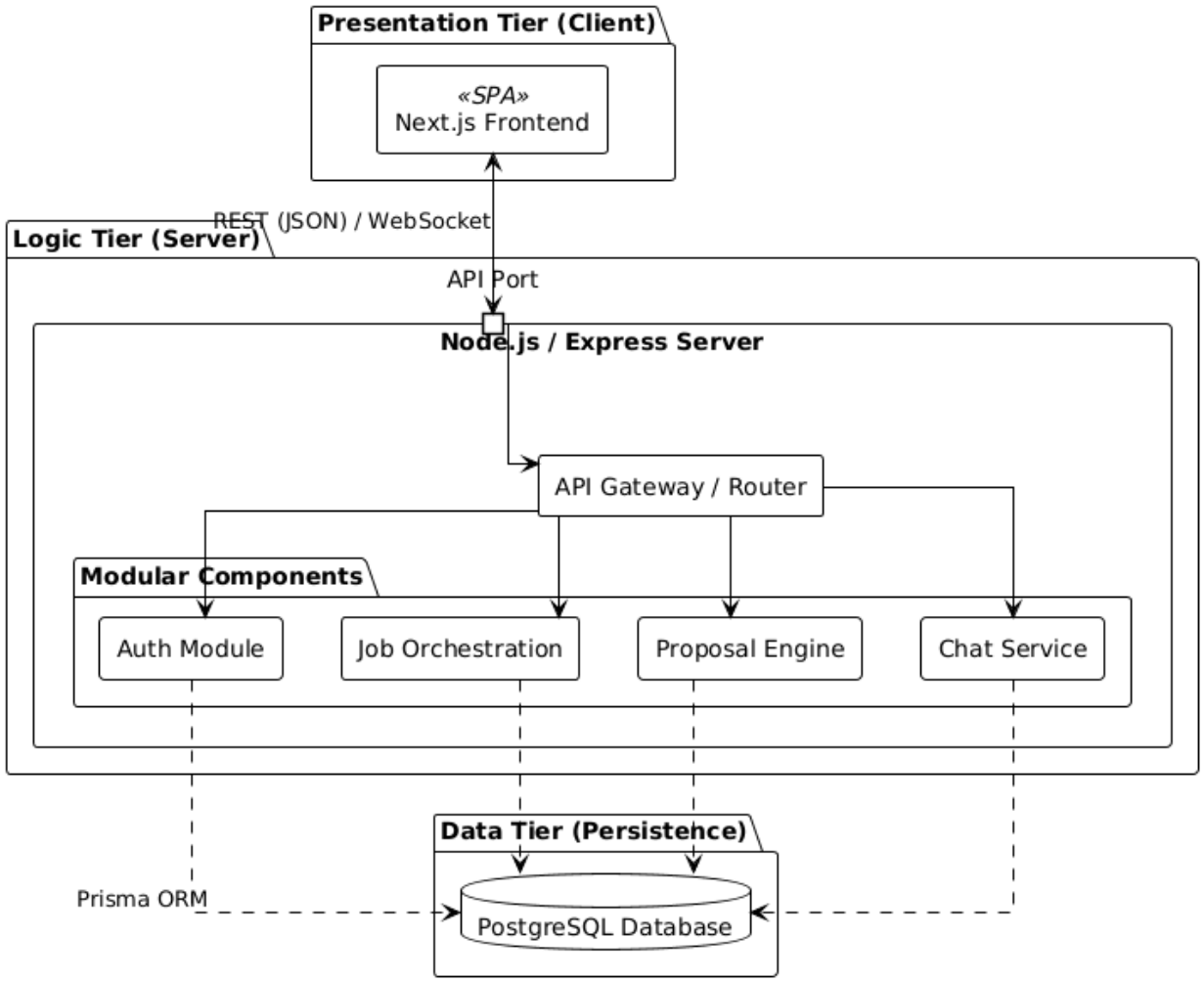
Why Node.js & Express?

- Rationale: Node.js utilizes an event-driven, non-blocking I/O model. This is architecturally superior for the Tibeb platform, which is I/O heavy (handling simultaneous chat messages, frequent database reads for job feeds) rather than CPU intensive.

Why PostgreSQL (Relational) over MongoDB (NoSQL)?

- Rationale: The core domain (Marketplace) relies heavily on structured relationships (Clients have Jobs; Jobs have Proposals) and strict data integrity. PostgreSQL provides ACID compliance, ensuring that a Proposal is never orphaned if a Job is deleted (Cascading Deletes) and that financial simulations are transactionally safe.

Figure 1 System architecture diagram



5. Detailed Software Design

5.1 Component Design

The system's backend is structured into modular components following the Controller-Service-Repository pattern. Below is the detailed design for the critical subsystems.

5.1.1 Identity Management Component

- Name: AuthModule
- Responsibilities: Handles incoming login/registration requests, verifies credentials, issues JWTs, and manages user context.
- Dependencies: Bcrypt (Hashing), JsonWebToken (Signing), PrismaClient (Database).
- Interfaces:
    - POST /auth/register: Accepts RegisterDTO, returns UserResponse.
    - POST /auth/login: Accepts LoginDTO, returns AuthToken.
- Internal Logic: custom middleware strategy to extract the Authorization header, verify the JWT signature, and attach the user object to the Request context.

5.1.2 Job Orchestration Component

- Name: JobModule
- Responsibilities: Manages the lifecycle of Job entities and executes search queries.
- Dependencies: JobRepository (Data Access), AuthMiddleware (RBAC).
- Interfaces:
    - JobController: Handles HTTP requests.
    - JobService: Implements business rules (e.g., "Only Clients can post").
- Internal Logic: The JobService validates that the budget_max >= budget_min and that the deadline is in the future before invoking the Repository to persist data.

5.1.3 Real-Time Messaging Component

- Name: ChatGateway
- Responsibilities: Manages WebSocket connections, handles room subscriptions (based on proposal_id or user_id), and persists chat history.
- Dependencies: Socket.io (Transport), MessageRepository.
- Interfaces:

    - Event: connection: Handshake and Authentication.
    - Event: join_room: Subscribes a socket to a specific chat context.
    - Event: send_message: Broadcasts payload to room members.

5.2 Data Design

The Data Design follows a Relational Model optimized for transactional integrity.

5.2.1 Database Schema (Physical Data Model)

The PostgreSQL database is managed via the Prisma Schema definition:

- Table: Users
    - Columns: id (UUID, PK), email (Unique Index), password_hash, role (Enum), created_at.
- Table: Jobs
    - Columns: id (UUID, PK), client_id (FK -> Users), title, description, budget, status (Enum: OPEN, CLOSED), search_vector (TSVECTOR for full-text search).
- Table: Proposals
    - Columns: id (UUID, PK), job_id (FK), freelancer_id (FK), bid_amount, cover_letter.
    - Constraint: @@unique([job_id, freelancer_id]) Enforces that a freelancer can only bid once per job.

5.2.2 Data Structures (In-Memory)

- DTOs (Data Transfer Objects): Strictly typed TypeScript interfaces (e.g., CreateJobInput) used to validate API payloads using the Zod library before processing.

5.3 Interface Design

5.3.1 API Design (REST)

The Application Programming Interface (API) follows REST Level 2 maturity:

- Resource Oriented: URLs represent resources (e.g., /jobs, /proposals).
- HTTP Verbs: GET (Read), POST (Create), PATCH (Update), DELETE (Remove).
- Status Codes: Uses standard codes (200 OK, 201 Created, 400 Bad Request, 401 Unauthorized, 403 Forbidden, 404 Not Found, 500 Internal Error).

5.3.2 Communication Protocols

- HTTPS: All REST traffic is encrypted via TLS 1.2+.

- WSS (Secure WebSockets): Real-time events use the WebSocket protocol over TLS. The server performs a handshake verification using the JWT token passed in the query string or headers.

5.4 Algorithm Design

5.4.1 Composite Search & Filtering Algorithm

- Problem: Efficiently finding jobs based on multiple optional criteria.
- Algorithm Logic:

1. Initialize a base database query builder.
2. If keyword is present: Add a full-text search condition (plainto_tsquery) to the WHERE clause.
3. If category is present: Add an exact match condition.
4. If min_budget is present: Add a >= condition.
5. Apply ORDER BY created_at DESC.
6. Apply LIMIT and OFFSET for pagination.
7. Execute query.

5.4.2 Proposal Idempotency Check

- Problem: Preventing race conditions where a freelancer clicks "Submit" twice rapidly.
- Algorithm Logic:

1. Receive POST request.
2. Check Database for existing proposal with job_id AND freelancer_id.
3. If exists -> Return Conflict immediately.
4. If null -> Proceed to create transaction.

6. Behavioral Design

This section describes how the software components interact to realize the functional requirements and how the system manages state changes over time.

6.1 Use Case Realizations

This subsection maps the high-level Use Cases (defined in the SRS) to the concrete software classes and components defined in Section 5.

UC-02: Post New Job

- Participating Objects: Client (Actor), JobController, JobService, JobModel (Prisma).
- Flow:

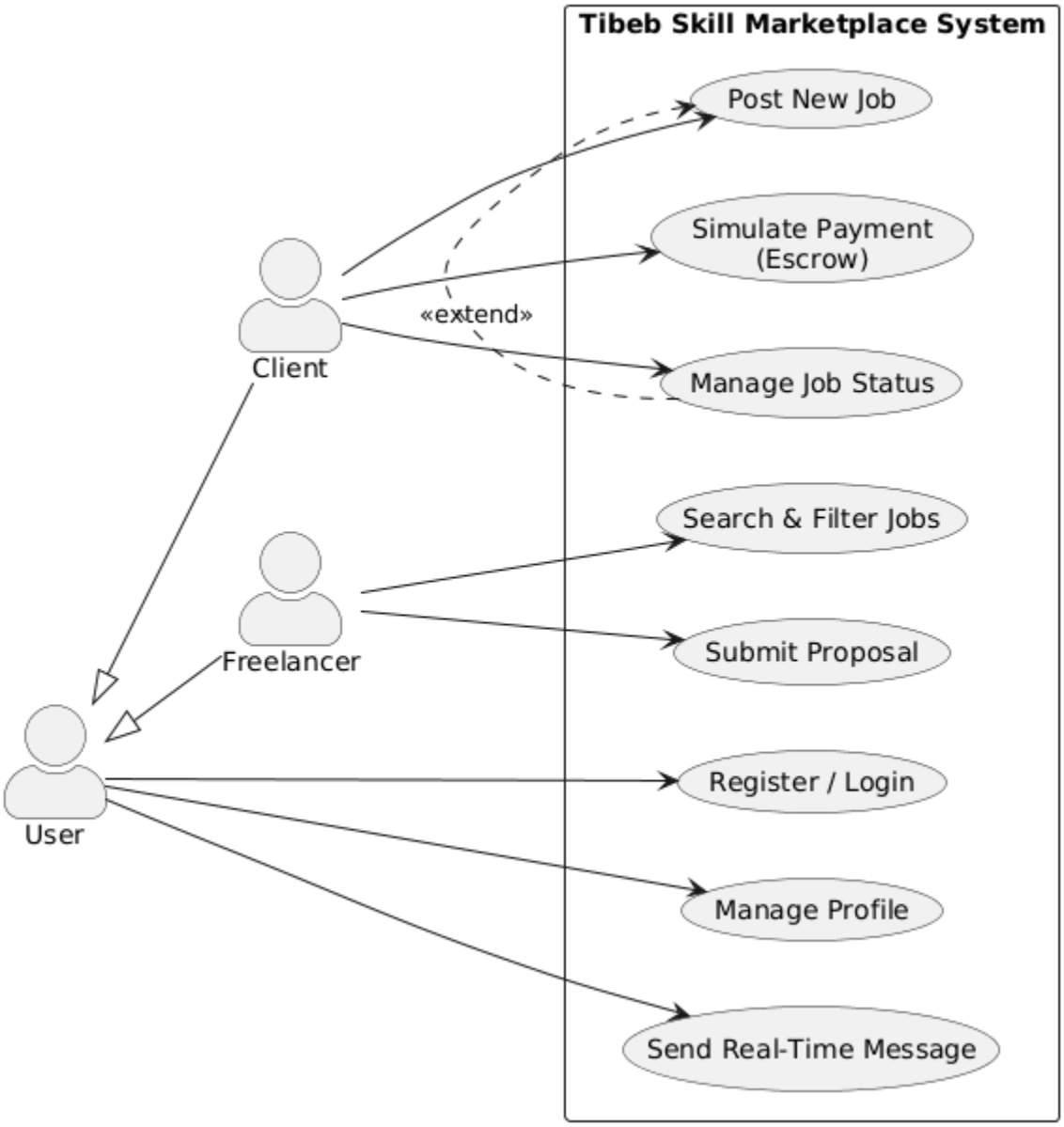
1. The Client sends a POST request.
2. JobController instantiates a CreateJobDTO and validates input structure.
3. JobController delegates execution to JobService.
4. JobService enforces business rules (e.g., checks User Role = CLIENT).
5. JobService invokes JobModel.create() to persist data.
6. The system returns the created Job Object to the Client.

UC-04: Secure Messaging

- Participating Objects: User (Actor), SocketController, ChatService, MessageModel.
- Flow:

1. The User emits a send_message event via WebSocket.
2. SocketController intercepts the event and extracts the JWT from the handshake.
3. ChatService validates that the sender is a participant in the target conversation.
4. ChatService persists the payload via MessageModel.
5. SocketController broadcasts the message to the specific Room ID.

Figure 2 Use case diagram



6.2 Sequence Diagrams

The following descriptions detail the runtime interaction logic.

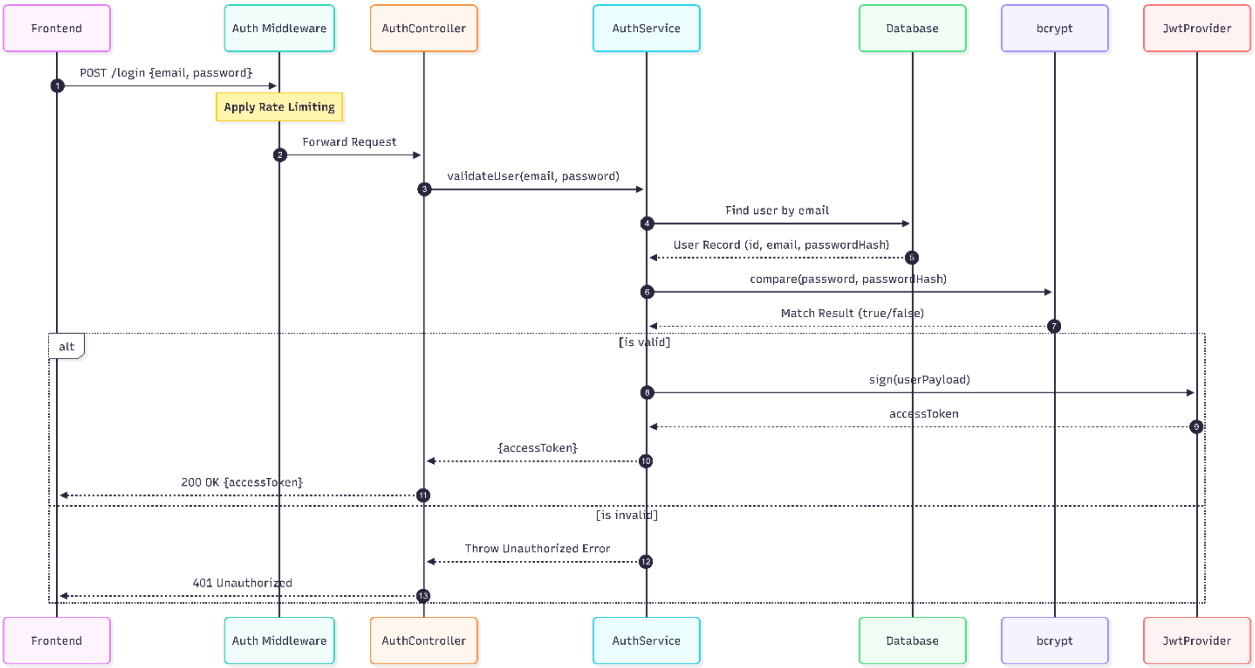
6.2.1 Sequence: User Authentication (Login)

This flow describes the synchronous HTTP interaction for session creation.

1. Frontend sends POST /login with {email, password}.

2. Auth Middleware applies rate limiting to prevent brute force.
3. AuthController calls AuthService.validateUser().
4. AuthService retrieves the user record from the Database.
5. AuthService utilizes bcrypt.compare() to verify the password hash.
6. If valid, AuthService calls JwtProvider.sign() to generate a token.
7. AuthController returns 200 OK with the accessToken.

Figure 3 Sequence diagram for user authentication

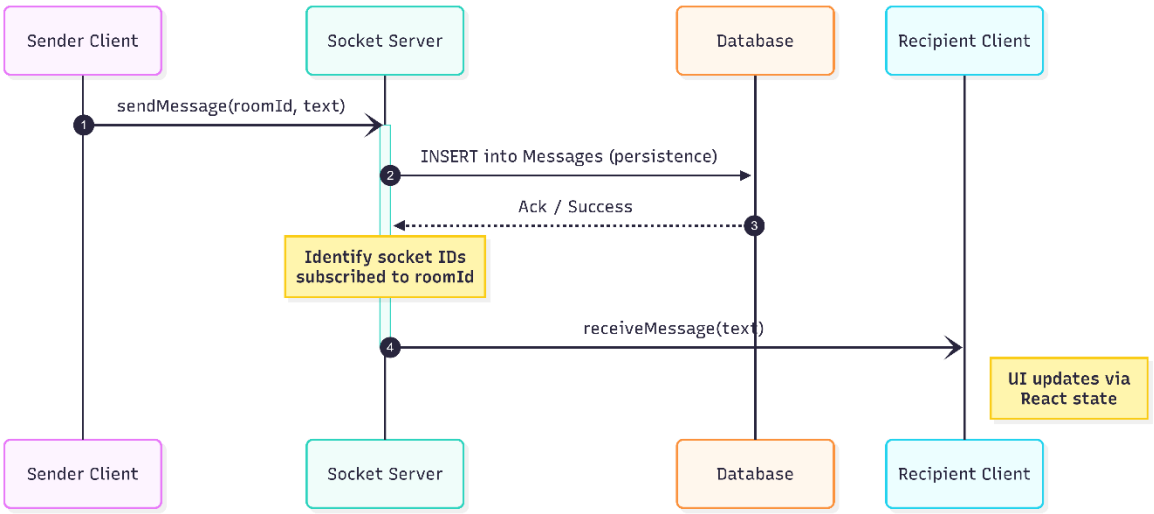


6.2.2 Sequence: Real-Time Message Delivery

This flow describes the asynchronous WebSocket interaction.

1. Sender Client emits sendMessage(roomId, text).
2. Socket Server receives the event.
3. Database insert is executed (Persistence).
4. Socket Server identifies connected socket IDs subscribed to roomId.
5. Socket Server emits receiveMessage(text) to the Recipient Client.
6. Recipient Client UI updates immediately via React state.

Figure 4 Sequence diagram for Real-Time Message Delivery



6.3 State Design

The Job entity acts as a finite state machine. The system enforces strict transition rules to ensure data integrity.

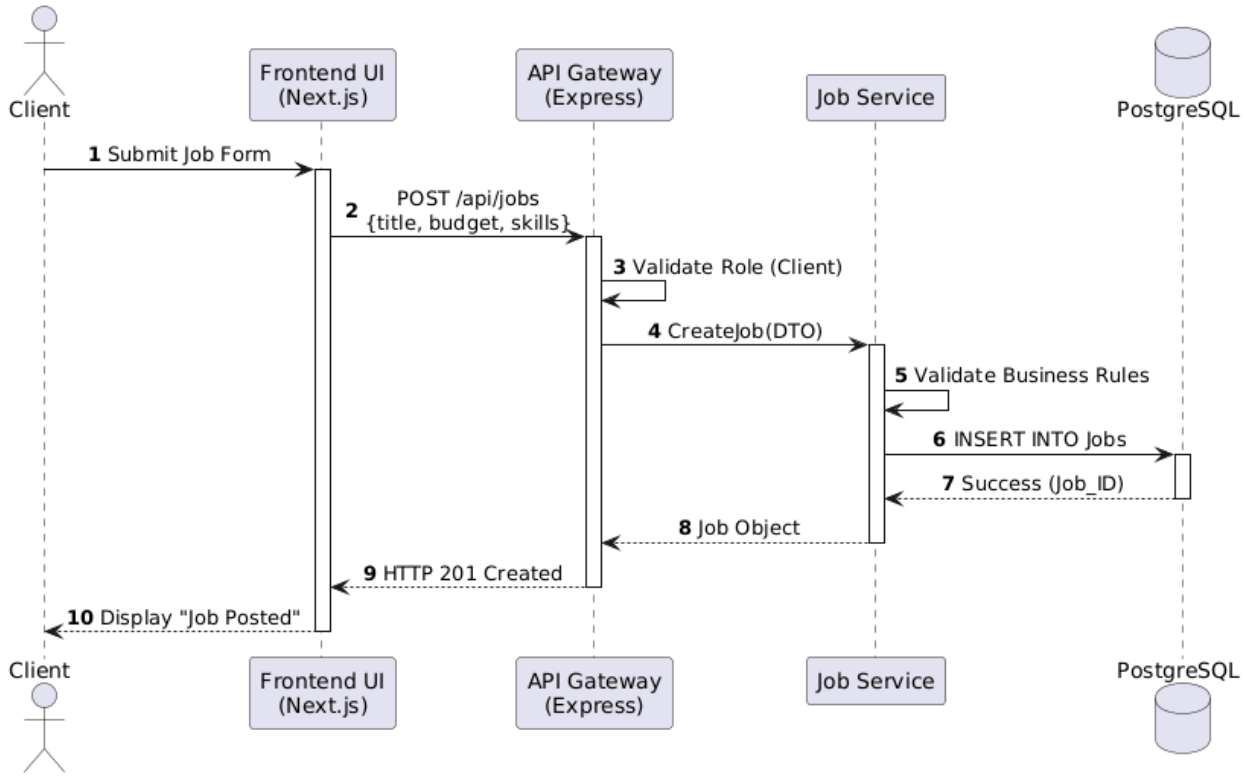
Entity: Job

- States: OPEN, CONTRACTED, CLOSED.

Transition Logic:

1. Initial State: OPEN (Upon creation).
2. Transition: HIRE_FREELANCER (OPEN -> CONTRACTED)
   - Trigger: Client accepts a Proposal.
   - Constraint: Job must not be expired.
   - Side Effect: All other proposals for this job are marked REJECTED.
3. Transition: COMPLETE_WORK (CONTRACTED -> CLOSED)
   - Trigger: Client confirms work completion / Releases funds.
   - Constraint: Must be in CONTRACTED state.
4. Transition: CANCEL_JOB (OPEN -> CLOSED)
   - Trigger: Client manually closes the listing.

Figure 5 Job posting



7. Deployment Design

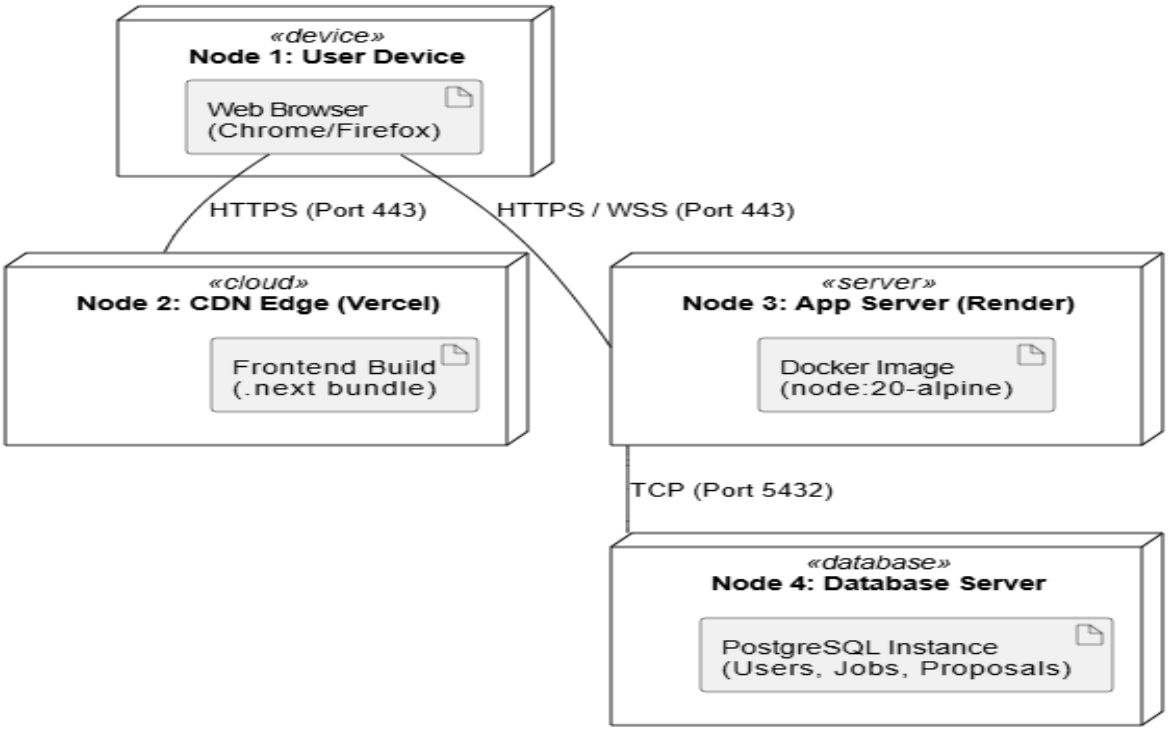
7.1 Deployment Architecture

The Tibeb platform utilizes PaaS (platform as a service) architecture. This approach abstracts the underlying infrastructure management, allowing the engineering team to focus on application logic.

The deployment is split into three decoupled environments:

1. Frontend / Edge Tier:
   - Host: specialized for Next.js.
   - Function: Hosts the static assets (HTML/CSS/JS) on a Global CDN and executes Server-Side Rendering (SSR) functions at the network edge for low latency.
   - Benefits: Automatic DDoS protection, and global caching.
2. Application Tier:
   - Host: Container Platform.
   - Function: Runs the Node.js/Express backend as a stateless Docker Container.
   - Process Management: Utilizes a process manager within the container to handle auto-restarts on failure.
3. Data Persistence Tier:
   - Host: Managed PostgreSQL.
   - Function: Provides a reliable, backed-up relational database service accessible only via secure connection strings.

7.2 Deployment Diagrams



7.3 Environment Description

To ensure stability and testing rigor, the system implements a strict Multi-Environment Strategy.

7.3.1 Development Environment

- Infrastructure: Local developer machines.
- Database: Local PostgreSQL instance.
- Configuration: NODE_ENV=development.
- Purpose: Rapid iteration, coding, and unit testing.

7.3.2 Staging / Acceptance Environment

- Infrastructure: Free Tier PaaS.
- Database: Cloud Hosted Development DB.
- Configuration: NODE_ENV=test.
- Data: Seeded with Synthetic Data (Mock users and jobs).
- Purpose: User Acceptance Testing (UAT), Instructor Demonstration, and Integration Testing.

8. Logging and Monitoring Design

8.1 Logging Strategy

To ensure system observability and rapid root-cause analysis, the application utilizes Structured Logging. Instead of unstructured text, logs are emitted as JSON objects, making them machine-parsable for future aggregation tools.

- Logging Library: The backend utilizes tools configured to output to stdout (Standard Output), which is captured by the container runtime.
- Log Levels:
    1. ERROR: System failures requiring immediate attention (e.g., Database connection lost, Uncaught Exception).
    2. WARN: Runtime abnormalities that are not fatal (e.g., 401 Unauthorized spikes, API rate limit breaches).
    3. INFO: High-level operational events (e.g., Server startup, Job posted, Proposal submitted).
    4. DEBUG: Detailed flow information (enabled only in Development/Staging environments).

8.2 Monitoring and Alerting

The system implements a proactive monitoring strategy to detect outages before they impact end-users.

- Health Checks:
    - Endpoint: The API exposes a lightweight GET /health endpoint.
    - Logic: It verifies connectivity to the PostgreSQL database and returns 200 OK with { status: "UP", timestamp: ... } if healthy, or 503 Service Unavailable if the DB is unreachable.
- Key Metrics:
    - Latency: 95th percentile response time for API requests.
    - Error Rate: Percentage of 5xx HTTP responses.
    - Resource Usage: CPU and Memory utilization of the Node.js container.

8.3 Auditing

To satisfy the system's security and simulated financial constraints, a specialized Audit Trail is implemented for sensitive operations. unlike standard application logs, audit logs are persistent business records.

- Audit Scope:
    - Authentication Events (Login, Logout, Failed Attempts).

    - Financial Simulation Events (Escrow Deposit, Funds Release).
    - Contractual Events (Job Created, Proposal Accepted).
- Storage: Audit records are stored in a dedicated PostgreSQL table (AuditLogs) separate from the application logic logs.
- Immutability: The Audit table is "Append-Only." The API provides no endpoint to update or delete audit records, ensuring integrity for dispute resolution.

9. Configuration Management

9.1 Configuration Strategy

The system adopts the methodology, separating configuration from code. Configuration is managed via Environment Variables, ensuring that the same Docker image or source code can be deployed to Development, Staging, and Production environments simply by changing the runtime environment config.

- Externalized Configuration:
  - Secrets: Sensitive credentials are injected at runtime.
- Loading Mechanism:
  - The Node.js application utilizes the dotenv library for local development (reading from a .env file).
  - In production, variables are read directly from process.env.
  - Validation: On application startup, a configuration schema (using Zod or Joi) validates that all required variables exist.

9.2Version Control

- System: Git is used for all source code versioning.
- Hosting: The repository is hosted on GitHub.

9.3 Build and Release Management

Automation is leveraged to ensure consistent and reliable deployments.

- CI/CD Pipeline:
  - Integration: GitHub Actions triggers on every push to main. It runs the test suite (npm test) and the linter.
  - Deployment**:** Upon successful build, the PaaS providers (Vercel/Render) automatically pull the latest code
- Release Versioning:
  - Database migrations are executed automatically as a pre-deploy step during the release process to ensure the schema matches the code.

10. Quality Attributes

10.1 Performance

To meet the latency budgets defined, the architecture incorporates the following optimization strategies:

- Database Indexing: Composite B-Tree indexes are applied to high-cardinality columns used in filtering to ensure query execution time remains.
- Connection Pooling: The Prisma Client is configured with a connection pool.
- Edge Caching: The Next.js frontend utilizes stale-while-revalidate caching headers. Static assets and public job feeds are.
- Payload Compression: All text-based HTTP responses are compressed.

10.2 Security

The system implements a "Secure by Design" philosophy:

- Authentication & Authorization:
    - Stateless Session: Usage of JWTs signed with HS256 prevents server-side session fixation attacks.
    - RBAC Middleware: A dedicated middleware function intercepts every protected route, decoding the JWT role claim and comparing it against the required route privileges (e.g., ensuring role === 'CLIENT' for POST /jobs).
- Data Protection:
    - Passwords are hashed.

1.3 Reliability and Availability

- Fault Tolerance: The Node.js application implements a Global Error Handler. Uncaught exceptions or unhandled promise rejections are caught, logged.
- Data Consistency: Critical workflows utilize Interactive Transactions. If any operation within the sequence fails, the database rolls back to the previous stable state.

10.4 Maintainability and Extensibility

- Type Safety: The entire backend is written in TypeScript. Shared DTOs and Interfaces ensure that a change in the Database Schema triggers compile-time errors in the Application Layer, catching regressions before runtime.
- Modular Monolith Structure: Code is organized by Domain (Auth, Job, Chat) rather than technical layer. This high cohesion allows future developers to extract a specific domain into a separate Microservice with minimal refactoring.

11. Traceability

11.1 Requirements Traceability

The following matrix maps the Functional Requirements (FR) defined in the SRS to the specific Software Design Elements (Modules, Components, and Database Entities) described in this SDD. This ensures that every requirement has a corresponding implementation strategy.

SRS Req ID	Requirement Name	SDD Component / Module	Implementation Element
FR 3.2.1	Authentication & Session	AuthModule	AuthController, AuthService, JwtStrategy Middleware.
FR 3.2.2	Job Lifecycle Management	JobModule	JobController (Endpoints), JobService (Business Logic), Jobs Table.
FR 3.2.3	Advanced Search & Filtering	JobModule	JobService.search() Algorithm, PostgreSQL TSVECTOR Index.
FR 3.2.4	Proposal Submission Engine	ProposalModule	ProposalService.submit() (Transactional), Proposals Table.
FR 3.2.5	Real-Time Messaging	ChatGateway	Socket.io Server, MessageRepository, Messages Table.
FR 3.2.6	Transaction Simulation	JobModule	JobService.updateStatus() (State Machine), Logical Ledger Logic.
BR 01	Role Isolation (RBAC)	Middleware	RolesGuard (Interceptor), User.role Enum.
BR 02	Proposal Validity	ProposalModule	ProposalService (Pre-check logic: if job.status != OPEN throw error).
BR 04	Escrow Simulation	JobModule	JobStatus Enum (IN_PROGRESS state flag).

12. Design Constraints and Compliance

12.1 Standards Compliance

The software design adheres to the following international and industry-standard specifications to ensure interoperability, security, and quality:

1. IEEE 1016-2009: The structure of this Software Design Document (SDD) follows the IEEE Standard for Information Technology—Systems Design—Software Design Descriptions.

12.2 Legal and Regulatory Requirements

1. Data Privacy and Protection:
    - Constraint: Alignment with the principles of the Ethiopian Personal Data Protection Proclamation (No. 1321/2024) regarding the processing of personal data.
    - Compliance Implementation:
        - Data Minimization: The schema is designed to store only essential data required for service delivery.
        - Security of Processing: Application of Bcrypt hashing for passwords and TLS encryption for data in transit ensures that user data is protected against unauthorized access.
        - Right to Erasure: The database design supports "Soft Delete" mechanisms that can be converted to "Hard Delete" (scrubbing PII) upon user request to satisfy the "Right to be Forgotten".

13. Risks and Mitigations

This section identifies potential architectural, technical, and operational risks associated with the chosen design and outlines the specific mitigation strategies embedded in the system architecture.

13.1 Technical and Architectural Risks

Risk 1: Node.js Event Loop Blocking

- Description: Node.js is single-threaded. CPU-intensive tasks (e.g., complex image processing for portfolio uploads or heavy search aggregation) could block the main event loop, causing the API to become unresponsive for all users.
- Impact: High latency or Service Unavailable (503) errors during peak load.
- Mitigation Strategy:
    - Asynchronous Design: The architecture relies strictly on non-blocking, asynchronous I/O (async/await) for all database and network operations.

Risk 2: WebSocket Connection Limits

- Description: Maintaining thousands of persistent WebSocket connections (for Chat) consumes significant server memory and file descriptors.
- Impact: Server crash (OOM - Out of Memory) if concurrent users exceed the MVP infrastructure capacity (512MB RAM).
- Mitigation Strategy:
    - Horizontal Scaling: The architecture allows for the Chat Service to be deployed on separate nodes behind a Redis Adapter (for socket synchronization) if the user.

13.2 Security Risks

Risk 3: JWT Token Theft (Session Hijacking)

- Description: If a malicious actor steals a valid JWT, they can impersonate the user until the token expires.
- Impact: Unauthorized access to user funds (simulated) or data.
- Mitigation Strategy:
    - Short Lived Tokens: Access Tokens are configured with a short expiration (e.g., 15 minutes).
    - TLS Enforcement: The design mandates HTTPS for all traffic, preventing network sniffing attacks.

13.3 Implementation Risks

Risk 4: Database Schema Rigidity

- Description: Relational databases (PostgreSQL) are rigid. Frequent schema changes during agile development can break existing data or code.
- Impact: Development delays due to migration conflicts.
- Mitigation Strategy:
    - Prisma Migrations: The project utilizes Prisma's automated migration tool. All schema changes are version-controlled (migration.sql files), allowing the team to roll back schema changes safely and ensure the database stays in sync with the codebase.

14. Appendices

Appendix A: List of figures

The following diagrams illustrate the architectural decisions described