Course: Integrative Programming And Technologies 2

UNITE's backend-unit: Comprehensive Architectural Documentation and Deployment Report

Proponents

Petalio, Joseph Angelo Q.
Sulit, Marc Lester E.
Villamer Patrick Kurt O.

2025

Title Page 0
1. Executive Summary 2
    1.1 Project Overview and Objectives 2
    1.2 Current Deployment Status 2
2. Architectural Paradigms and System Design 2
    2.1 The Decoupled Architecture Rationale 2
    2.2 Network Topology and Request Lifecycle 3
    2.3 Technology Stack Specifications 4
3. Infrastructure as Code (IaC) and Deployment Configuration 4
    3.1 The Role of Render Blueprints 5
    3.2 Service Configuration Details 5
    3.3 Environment Variable Management 5
    3.4 Continuous Deployment Pipeline 6
4. Network Integration: Bridging Vercel and Render 6
    4.1 The Same-Origin Policy and CORS 6
    4.2 Implementation of CORS Middleware 6
5. Backend Software Architecture 7
    5.1 Directory Structure and Modularity 7
    5.2 Middleware Chain and Request Processing 8
    5.3 Health Check Implementation 8
6. API Documentation and Contract 8
    6.1 RESTful Design Principles 8
    6.2 Authentication Strategy 8
7. Data Persistence and Schema Design 9
    7.1 Database Selection: MongoDB Atlas 9
    7.2 Schema Design and Data Model 9
8. Operational Procedures and Reliability 10
    8.1 Logging and Observability 10
    8.2 Performance Monitoring 10
9. Security and Compliance Considerations 11
    9.1 Network Security Measures 11
    9.2 Application Security Best Practices 11
10. Future Roadmap and Scalability 11
    10.1 Short-Term Enhancements 11
    10.2 Long-Term Scalability Goals 11
11. Conclusion 12

1. Executive Summary

1.1 Project Overview and Objectives

The **backend-unit** project represents the server-side foundation of a distributed web application designed to deliver dynamic content and robust business logic processing to a decoupled frontend interface. This documentation serves as the definitive technical reference for the backend system, which has been successfully deployed to a production environment using **Render** as the Platform-as-a-Service (PaaS) provider, integrated with a client-side application hosted on **Vercel**.

The primary objective of this deployment was to establish a scalable, maintainable, and secure architecture that leverages the strengths of modern cloud primitives. By separating the concern of UI rendering (handled by Vercel's edge network) from data processing and persistence, the project achieves a high degree of architectural flexibility. This "Backend-for-Frontend" (BFF) or dedicated API service pattern allows for independent deployment cycles, specialized scaling strategies, and a cleaner codebase separation compared to monolithic alternatives.

This report documents the current state of the system, detailing the successful configuration of cross-origin resource sharing (CORS) to bridge the network gap between Vercel and Render, the implementation of RESTful API endpoints, and the specific integration with **MongoDB Atlas** for data persistence.

1.2 Current Deployment Status

As of the latest operational review, the backend unit is fully functional and publicly accessible via a secure HTTPS endpoint on the onrender.com domain. The integration with the Vercel-hosted frontend has been verified, with data flowing successfully between the two distinct origins. Critical infrastructure components, including the Node.js runtime environment, Express.js framework, and **MongoDB Atlas** database connectivity, have been configured and stabilized. The system utilizes a GitOps workflow, where updates to the central GitHub repository automatically trigger build and deployment pipelines on the Render platform.

2. Architectural Paradigms and System Design

2.1 The Decoupled Architecture Rationale

The decision to adopt a decoupled architecture—hosting the frontend and backend on separate infrastructure providers—is rooted in the "Separation of Concerns" principle. While monolithic applications offer simplicity in early development, they often suffer from tightly coupled dependencies that hinder scalability and independent iteration. In contrast, the split-stack approach utilized here allows the project to leverage the specialized optimizations of distinct platforms.

Vercel is utilized for the frontend due to its sophisticated edge network, which caches static assets (HTML, CSS, JavaScript) geographically close to the user.[7] **Render**, conversely, provides a robust environment for long-running processes, such as the Node.js server required for this backend unit. Unlike Vercel's serverless functions, which have execution time limits and cold-start latencies, Render's persistent web services maintain a "warm" state, allowing for immediate response to complex API requests and maintaining persistent connections to the MongoDB Atlas cluster.

The table below summarizes the strategic allocation of responsibilities across the architecture:

Component	Provider	Primary Responsibility	Key Technical Advantage
Frontend UI	Vercel	Presentation Layer, Client Routing	Global CDN, Edge Caching, Static Site Generation (SSG)
Backend API	Render	Business Logic, Data Aggregation	Long-running processes, Persistent RAM/CPU, Native Node.js support
Database	MongoDB Atlas	Data Persistence	Fully Managed, Multi-cloud, Document Model (NoSQL)
CI/CD	GitHub	Source Control, Workflow Trigger	Version Control, Automated Pipeline Initiation

2.2 Network Topology and Request Lifecycle

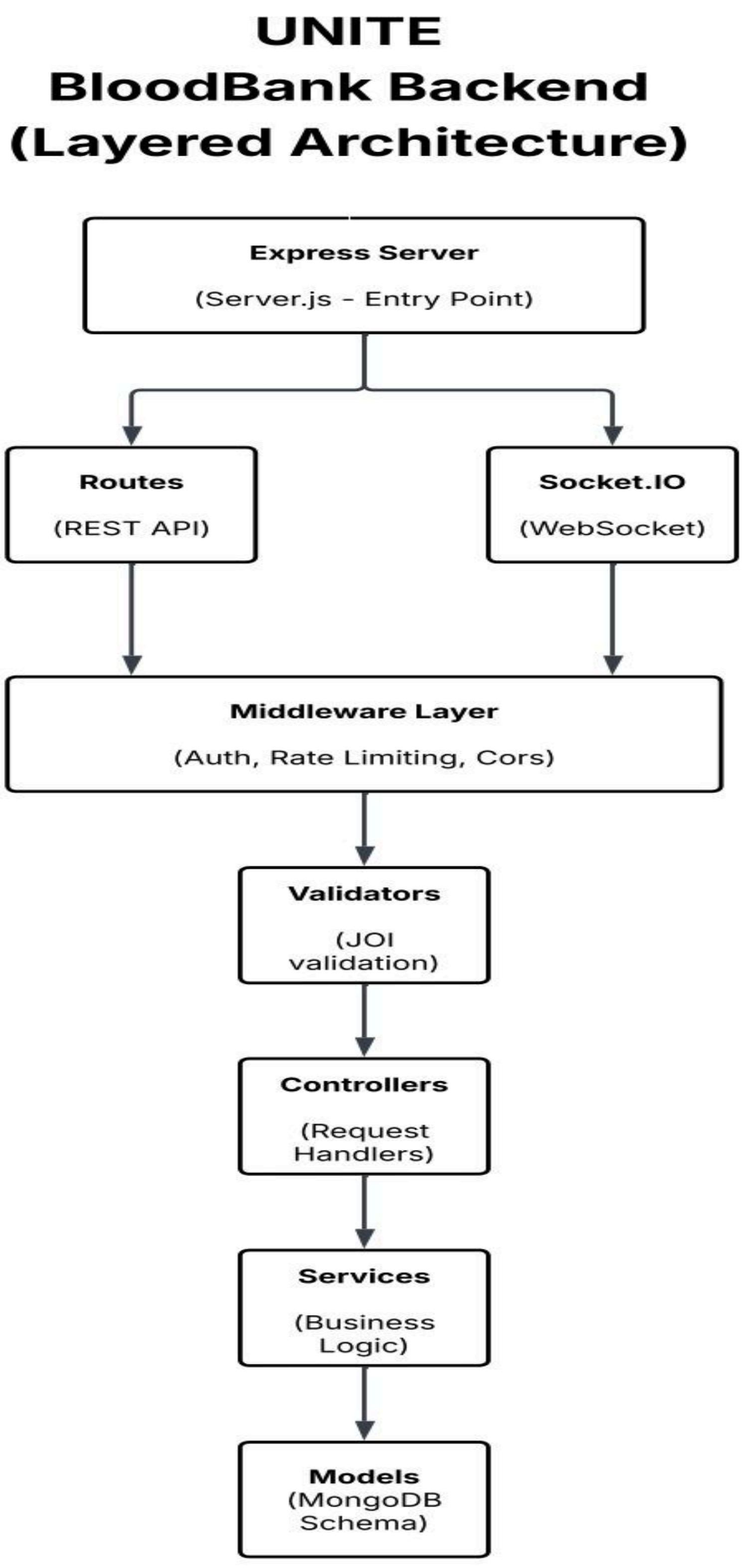
The communication between the frontend and backend occurs over the public internet, necessitating secure transport protocols and strict access controls. The network topology can be conceptualized as a hub-and-spoke model where the Render backend acts as the central hub for data, processing requests from the Vercel frontend spoke.

When a user interacts with the application, the following sequence of events occurs:

1. **Initial Load:** The user's browser requests the application from https://unite-development.vercel.app/. Vercel's CDN serves the compiled React/Next.js assets.
2. **Client-Side Execution:** The React application initializes in the user's browser.
3. **Data Fetching:** When dynamic data is required (e.g., loading a user profile), the client initiates an asynchronous HTTP request (using fetch or axios) to https://my-backend.onrender.com/api/resource.
4. **Network Traversal:** The request travels via HTTPS to Render's load balancer.
5. **Processing:** Render routes the request to the active Node.js container. The Express.js application parses the request, authenticates the user, and queries **MongoDB Atlas**.
6. **Response:** The backend constructs a JSON response and sends it back. The browser receives the data and updates the DOM.

2.3 Layered Application Architecture

To ensure code maintainability and separation of logic, the backend follows a strict layered architecture. This separation ensures that business rules are decoupled from the HTTP transport layer (Routes/Controllers) and the data access layer (Models).



3. Infrastructure as Code (IaC) and Deployment Configuration

3.1 The Role of Render Blueprints

To move beyond manual configuration and "ClickOps"—where infrastructure is set up via GUI clicks, leading to configuration drift—the project adopts Infrastructure as Code (IaC) using Render Blueprints. The configuration is defined in a render.yaml file located at the root of the repository. This file serves as a manifest, declaring the exact specifications of the backend service, ensuring that the production environment can be replicated deterministically.

The render.yaml specification allows for the definition of the service type (web, worker, cron), the runtime environment (node), and the specific build and start commands. This approach couples the infrastructure definition with the application code, meaning that a pull request can theoretically update both the software logic and the server configuration simultaneously, promoting DevOps best practices.[1]

3.2 Service Configuration Details

The specific configuration deployed for the backend unit includes the following parameters, which control how Render builds and runs the application:

- **Service Name:** A unique identifier for the service within the Render ecosystem.
- **Environment:** node - Specifies the managed runtime.
- **Build Command:** npm install (or yarn install) - This command is executed in the build environment to resolve and install dependencies listed in package.json. It is critical that a lockfile (package-lock.json or yarn.lock) is present in the repository to ensure that the exact versions of dependencies used in development are installed in production, preventing "it works on my machine" discrepancies.
- **Start Command:** node src/server.js (or npm start) - This command initiates the Express server process. The server must be configured to listen on the port defined by the PORT environment variable, which Render injects dynamically.

3.3 Environment Variable Management

Secure configuration is achieved through the use of environment variables, separating code from configuration. This "Twelve-Factor App" methodology prevents sensitive credentials from being committed to version control.

The following environment variables are critical for the backend unit's operation:

- NODE_ENV: Set to production. This signals the Express framework to optimize for performance by caching view templates (if used) and generating less verbose error messages.
- PORT: The port number on which the server listens. Render automatically assigns this (typically 10000), but the application code must be written to accept process.env.PORT.
- DATABASE_URL: The MongoDB connection string (SRV record) provided by Atlas. This includes the username, password, and cluster address (e.g., mongodb+srv://<user>:<password>@cluster0.mongodb.net/myFirstDatabase).
- FRONTEND_URL: The specific URL of the Vercel frontend (e.g., https://my-app.vercel.app). This is used to configure Cross-Origin Resource Sharing (CORS) policies dynamically, ensuring that only the authorized frontend can access the API.
- MONGO_URI: The connection string used to link your backend to the MongoDB database. It contains the protocol, host, credentials, and options required for the database driver to establish a secure connection.
- MONGO_DB_NAME: The specific MongoDB database name your application will use. This tells the database driver which logical database to operate on inside the MongoDB cluster.
- EMAIL_USER: The email address or username used by your backend's mail service. This is required so your server can authenticate with your SMTP provider before sending messages.

- EMAIL_PASS: The SMTP password or application-specific password for the email account defined in EMAIL_USER. This credential allows the backend to perform authenticated email sending.
- EMAIL_PORT: The port number used by your SMTP service (commonly 465 or 587). This indicates which secure channel your backend should use when sending outbound mail.
- REDIS_URL: The connection string for your Redis instance. This is used for caching, session storage, rate limiting, or any backend feature that relies on fast in-memory data operations.
- ALLOWED_ORIGINS: A list of authorized frontend URLs permitted to access your backend through CORS. This variable tells the server which domains are allowed to send requests, helping prevent unauthorized websites from interacting with your API.

3.4 Continuous Deployment Pipeline

The project utilizes Render's native integration with GitHub to establish a continuous deployment (CD) pipeline. The workflow operates as follows:

1. **Code Push:** A developer pushes code to the main branch of the GitHub repository.
2. **Webhook Trigger:** GitHub sends a webhook to Render, signaling a change.
3. **Build Phase:** Render allocates a build container, checks out the latest code, and runs the buildCommand.
4. **Health Check:** Upon successful build, Render starts the service. It attempts to connect to the configured Health Check Path (e.g., /health).
5. **Traffic Switch:** If the health check returns a 200 OK status, Render seamlessly routes traffic to the new instance and terminates the old one.

4. Network Integration: Bridging Vercel and Render

4.1 The Same-Origin Policy and CORS

One of the most significant challenges in a split-stack architecture is managing the browser's security model, specifically the Same-Origin Policy (SOP). The SOP restricts how a document or script loaded from one origin (e.g., domain-a.com) can interact with resources from another origin (e.g., domain-b.com). Since the frontend is served from Vercel and the backend from Render, they have distinct origins, and browsers will block API requests by default.

To resolve this, the backend implements **Cross-Origin Resource Sharing (CORS)**. CORS is a mechanism that allows the server to explicitly whitelist certain origins, telling the browser that it is safe to allow the request.

4.2 Implementation of CORS Middleware

The backend utilizes the cors middleware package for Express. The configuration is strict and explicit, avoiding the insecure wildcard (*) setting typically found in development environments.

Configuration Logic:

The cors middleware is configured to read the allowed origin from the FRONTEND_URL environment variable. This ensures that in a production environment, only the production frontend is allowed, while in a development environment, a local URL (e.g., http://localhost:3000) can be used.

Key CORS headers configured include:

- Access-Control-Allow-Origin: Set to the value of FRONTEND_URL.
- Access-Control-Allow-Methods: Restricts the HTTP verbs allowed (e.g., GET, POST, PUT, DELETE, OPTIONS).
- Access-Control-Allow-Headers: Detailed whitelist of headers the client is permitted to send (e.g., Content-Type, Authorization).
- Access-Control-Allow-Credentials: Set to true. This is crucial if the application uses cookies (such as HTTP-only secure cookies for JWT storage) for authentication.[14]

5. Backend Software Architecture

5.1 Directory Structure and Modularity

The codebase is organized according to the Model-View-Controller (MVC) pattern—adapted as Model-Route-Controller for APIs—to ensure scalability and ease of maintenance. This structure separates the definition of data structures (Models), the handling of HTTP requests (Routes/Controllers), and the underlying business logic.

- src/app.js: The entry point. Initializes Express, connects to MongoDB Atlas, and mounts global middleware.
- src/config/: Contains configuration logic, such as logger setups and database connection utilities.
- src/models/: Defines the Mongoose schemas and data interactions.
- src/controllers/: Contains the logic for processing requests, validating input, and sending responses. These functions orchestrate the flow of data.
- src/routes/: Defines the API endpoints and maps them to specific controller functions.
- src/middleware/: Custom middleware functions for authentication verification, error handling, and request logging.

5.2 Middleware Chain and Request Processing

In Express.js, the request processing pipeline is constructed via a series of middleware functions. For this backend unit, the chain is configured as follows:

1. **Security Middleware (helmet):** Sets various HTTP headers (e.g., Strict-Transport-Security, X-Frame-Options) to protect against common attack vectors like clickjacking and protocol downgrading.
2. **CORS Middleware:** Handles cross-origin permissions as described in Section 4.
3. **Body Parsing (express.json):** Parses incoming JSON payloads, making them available in req.body.
4. **Logging (morgan):** Logs details of every request (method, URL, status code, response time) to the standard output, which is captured by Render's logging system.
5. **Route Handlers:** The core application logic.
6. **Global Error Handler:** A final middleware that catches any errors thrown in the routes. It ensures that the server does not crash and returns a standardized JSON error response to the client (e.g., { "error": "Internal Server Error" }) rather than an HTML stack trace.

5.3 Health Check Implementation

A dedicated /health endpoint is implemented to support Render's zero-downtime deployment features. This endpoint returns a simple 200 OK status with a JSON body (e.g., { "status": "UP" }). This check confirms that the Express server is running and capable of accepting connections.

6. API Documentation and Contract

6.1 RESTful Design Principles

The API follows RESTful design principles to ensure predictability and ease of integration for the frontend developers.

- **Resources:** Data entities are exposed as resources (e.g., `/users`, `/posts`).
- **HTTP Verbs:** Standard verbs indicate the action (`GET` for retrieval, `POST` for creation, `PUT` / `PATCH` for updates, `DELETE` for removal).
- **Status Codes:** The API uses precise status codes to communicate outcomes (e.g., `201 Created`, `400 Bad Request`, `401 Unauthorized`, `500 Internal Server Error`).

6.2 Authentication Strategy

The API utilizes **JSON Web Tokens (JWT)** for stateless authentication.

1. **Login Flow:** The client sends credentials (email/password) to `POST /api/auth/login`.
2. **Verification:** The backend validates credentials against the MongoDB user collection.
3. **Token Issuance:** If valid, the backend signs a JWT containing the user's ID and role, and returns it to the client.
4. **Protected Routes:** Subsequent requests to protected endpoints must include the token in the `Authorization` header (`Bearer <token>`).

7. Data Persistence and Schema Design

7.1 Database Selection: MongoDB Atlas

The backend utilizes **MongoDB Atlas**, a managed NoSQL database service, chosen for its flexibility and seamless integration with Node.js via the **Mongoose** Object Data Modeling (ODM) library.

Key Configurations:

- **Connection:** The application connects using the standard MongoDB driver connection string stored in the `DATABASE_URL` environment variable.
- **IP Whitelisting:** To ensure security, the MongoDB Atlas cluster is configured to allow connections from Render's outbound IP addresses (or `0.0.0.0/0` if static IPs are not available, relying on strong password authentication for security).
- **Mongoose ODM:** Mongoose is used to define strict schemas for the application data. This provides a layer of validation over the schema-less nature of MongoDB, ensuring that documents adhere to expected structures.

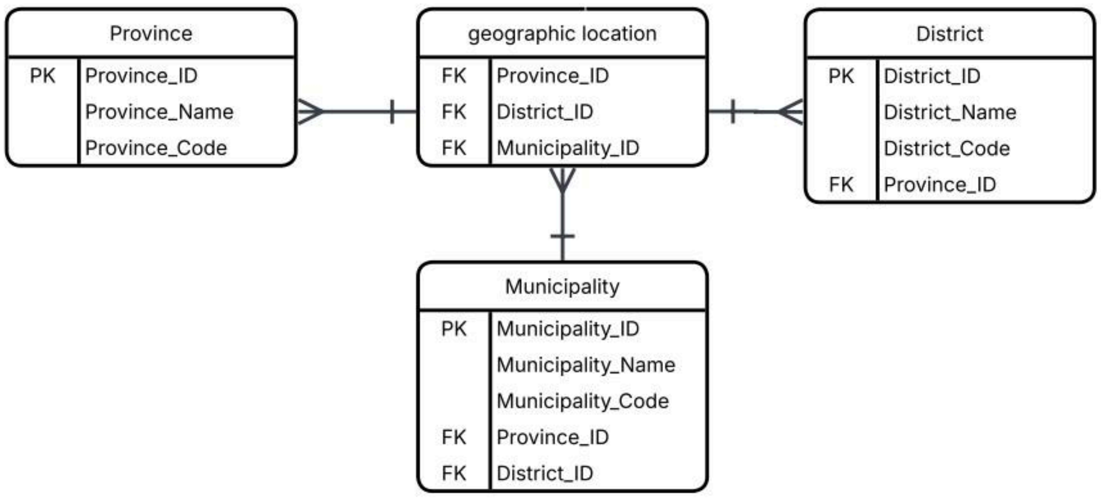
7.2 Entity Relationship Diagrams (ERD)

The following diagrams illustrate the core data structures and relationships within the system, based on the projected documentation for the "UNITE" system.

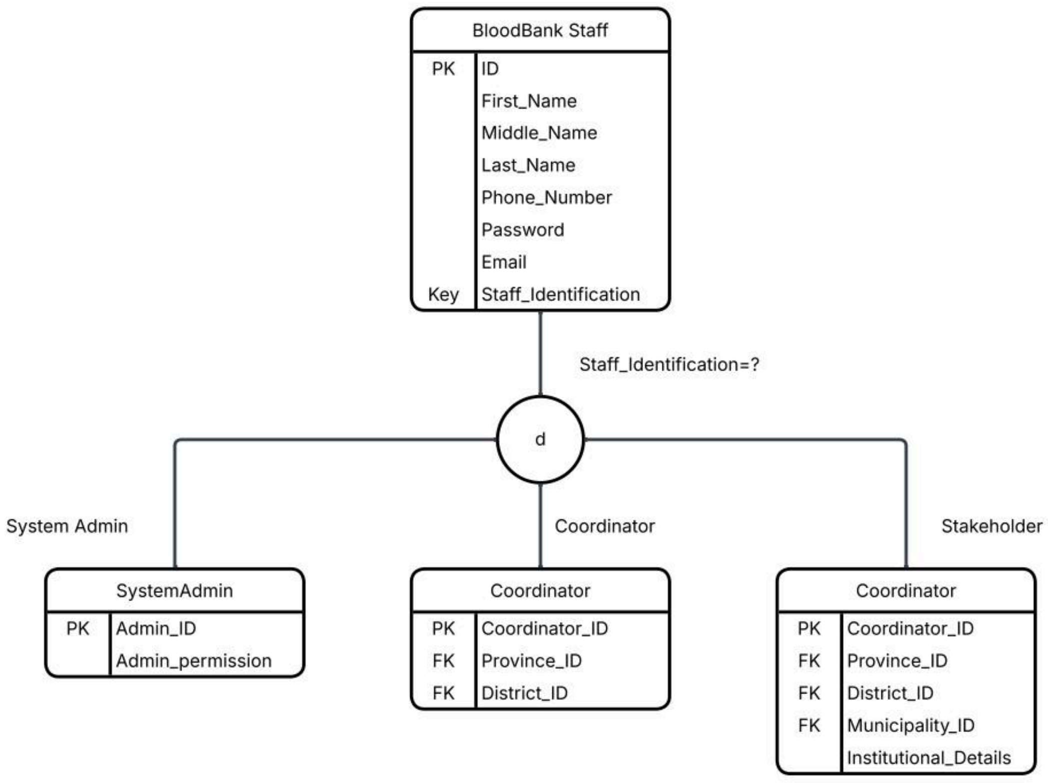
7.2.1 Geographic and Staff Hierarchy

This diagram details the administrative divisions (Province, District, Municipality) and the staff roles associated with them.

UNITE
BloodBank ERD
(Geographic Location)



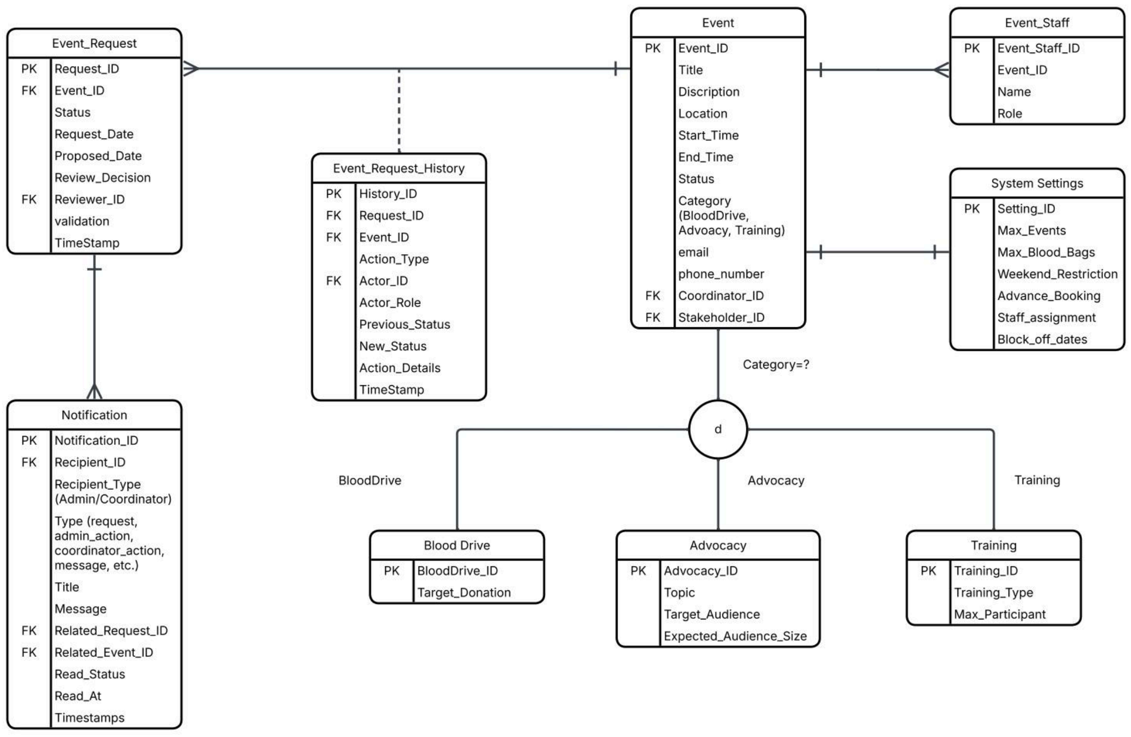
UNITE
BloodBank ERD
(BloodBank Staff)



7.2.2 Event and Request Management System

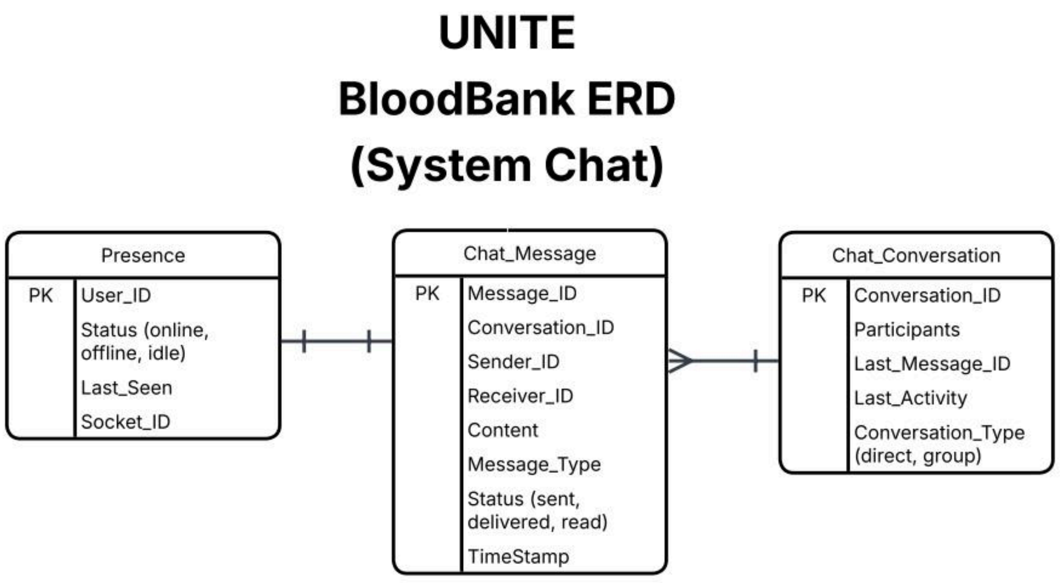
This segment covers the core business logic regarding blood drives, advocacy, training events, and the request/approval flow.

UNITE
BloodBank ERD
(Event-Request Flow)



7.2.3 Real-Time Chat System

The schema supports the socket.io-based communication layer.



8. Operational Procedures and Reliability

8.1 Logging and Observability

Effective logging is the primary tool for debugging production issues in a PaaS environment. The backend utilizes **structured logging**. Instead of printing raw strings (console.log('error happened')), the application logs JSON objects (logger.error({ error: 'db_connection_failed', timestamp: '...' })).

This approach allows logs to be ingested by log aggregation services (like Datadog or Render's native log streams) and queried programmatically. Logs are categorized by severity:

- **INFO:** Normal application flow (e.g., "Server started on port 10000").
- **WARN:** Non-critical issues (e.g., "Rate limit exceeded for IP x").
- **ERROR:** Critical failures requiring attention (e.g., "Database connection lost").

8.2 Performance Monitoring

Render provides basic metrics for CPU and Memory usage. High memory usage in Node.js often indicates memory leaks or inefficient processing of large datasets. If the backend exceeds the memory limits of the Render plan (e.g., 512MB in the Starter plan), the instance will be killed (OOM Kill). Monitoring these metrics is essential for capacity planning and deciding when to scale vertically (upgrade plan) or horizontally (add more instances).[2]

9. Security and Compliance Considerations

9.1 Network Security Measures

While HTTPS encrypts data in transit, further network security measures are implemented or planned:

- **TLS Termination:** Render handles SSL/TLS termination at the load balancer, offloading cryptographic overhead from the application.

- **IP Allowlisting (Future Roadmap):** Currently, the backend is public. A future enhancement involves restricting access to only Vercel's IP addresses. However, since Vercel uses dynamic IPs for its serverless functions, this is non-trivial and may require "Vercel Secure Compute" or a shared secret token mechanism.

9.2 Application Security Best Practices

- **Dependency Auditing:** The project pipeline includes npm audit to identify vulnerabilities in third-party libraries.
- **Rate Limiting:** To prevent Denial of Service (DoS) attacks and brute-force attempts on login endpoints, express-rate-limit is utilized to cap the number of requests a single IP can make within a time window.
- **Sanitization:** All inputs are sanitized to prevent **NoSQL injection** attacks, ensuring that malicious queries cannot manipulate the MongoDB logic.

10. Future Roadmap and Scalability

10.1 Short-Term Enhancements

- **Automated Testing Suite:** Integration of Jest or Mocha for unit and integration testing within the CI pipeline. This ensures that no code is deployed to Render without passing a regression test suite.
- **API Documentation Generator:** Implementation of Swagger/OpenAPI (via swagger-ui-express) to automatically generate interactive API documentation from code comments, ensuring the docs never drift from the implementation.

10.2 Long-Term Scalability Goals

- **Caching Layer:** Introduction of Redis to cache frequently accessed data (e.g., user profiles, static config). This reduces load on the MongoDB cluster and improves API response times.
- **Microservices Transition:** As the application complexity grows, the monolithic backend unit can be decomposed into smaller microservices (e.g., a dedicated Auth Service, a dedicated Reporting Service), each deployed as a separate Render Web Service.

11. Conclusion

The deployed backend unit represents a robust, modern foundation for the application. By leveraging the specific strengths of Render for backend processing, Vercel for frontend delivery, and **MongoDB Atlas** for scalable data persistence, the architecture achieves a balance of performance, maintainability, and developer productivity. The successful implementation of CORS and secure environment configuration ensures reliable communication between the platforms. This documentation provides the necessary context for current maintenance and future expansion, defining the standards for API design, security, and operational management.