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

Apex

Members

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

Executive Summary & Deployment Goals

This project establishes the robust server-side foundation for a distributed web application, designed to deliver dynamic content efficiently to a decoupled frontend. Our strategic architectural choices ensure scalability, performance, and maintainability.

Core Function

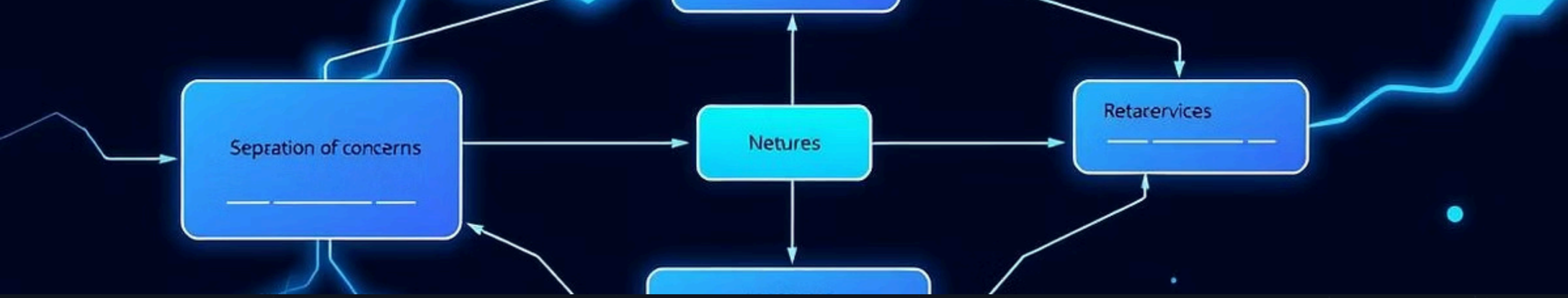
Server-side foundation of a distributed web application, delivering dynamic content to a decoupled frontend.

Architectural Pattern

Leverages a "Backend-for-Frontend" (BFF) pattern, separating UI rendering from complex data processing.

Current Status

Fully functional, publicly accessible via HTTPS, and integrated with a GitOps workflow for continuous delivery.



Decoupled Architecture & Platform Strategy

Our architecture emphasizes a clear separation of concerns, moving away from monolithic designs to enhance flexibility and resilience. This strategy allows each component to scale and evolve independently.

Rationale

Adopts a "Separation of Concerns" principle to avoid tightly coupled dependencies inherent in monolithic applications, fostering independent development and deployment.

Vercel (Frontend)

Dedicated to UI presentation and client routing, benefiting from Vercel's global CDN and edge caching for unparalleled speed and availability.

Render (Backend)

Manages core business logic, complex data aggregation, and persistent database connections, ensuring robust and reliable backend operations.

Technical Advantage

Unlike serverless functions, Render's persistent web services maintain a "warm" state, guaranteeing immediate API responses without cold start delays.

Request Flow & Communication

Understanding the journey of a request from the client to the server and back is crucial for optimizing performance and debugging. Our hub-and-spoke model ensures efficient data exchange.

1

Initial Load

Browser requests application assets directly from Vercel's high-performance CDN, ensuring rapid initial page display.

2

Data Fetching

The client initiates asynchronous HTTP requests (e.g., using fetch or axios) to the Render-hosted backend API for dynamic data.

3

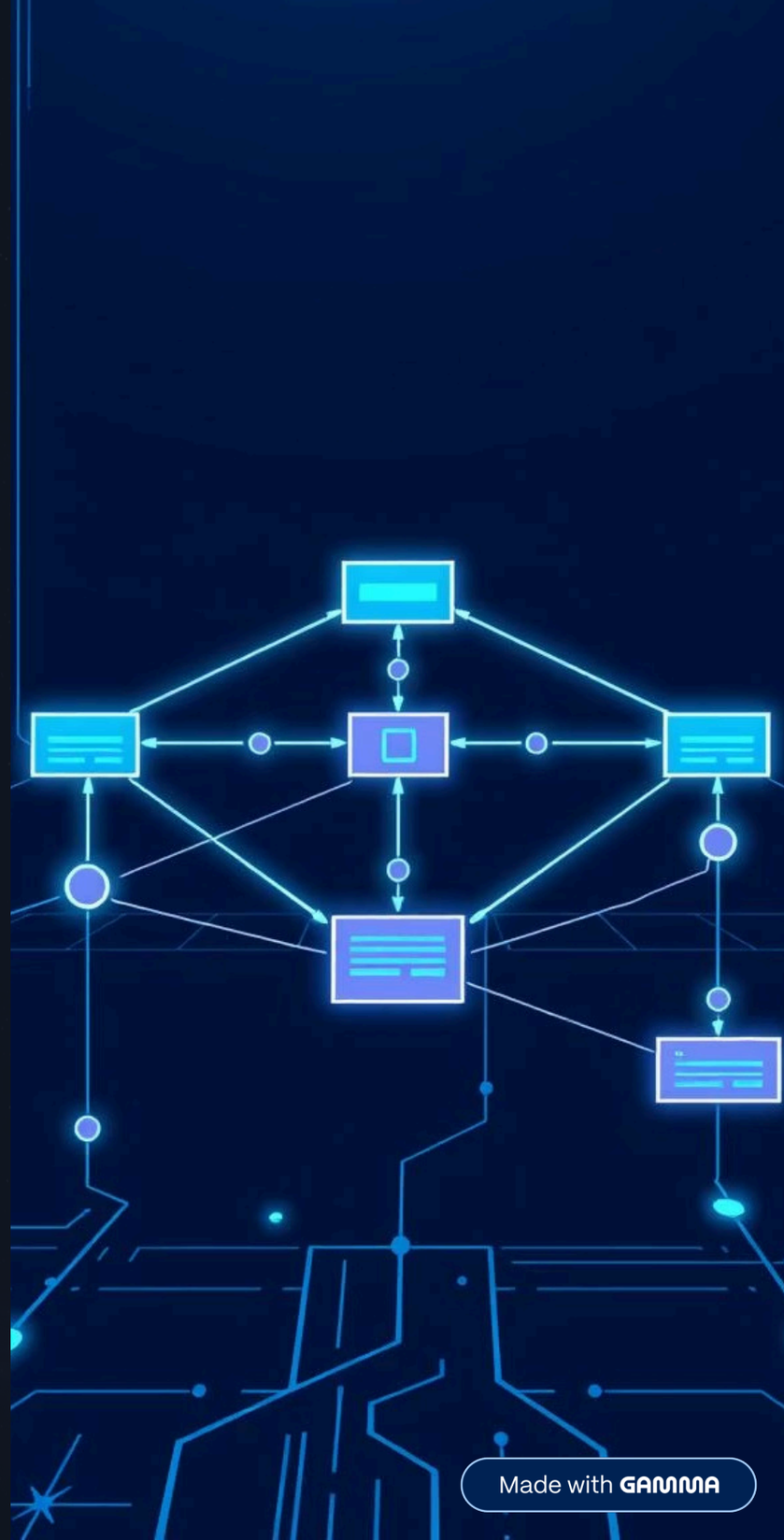
Processing

Render routes the incoming request to the Node.js container, where Express.js efficiently parses, authenticates, and queries MongoDB.

4

Response

The backend then returns a structured JSON response to the frontend, which dynamically updates the DOM without requiring a full page refresh.



Deployment Configuration & Environment Variables

Infrastructure as Code (IaC) is foundational to our deployment strategy, using Render Blueprints to ensure consistency and prevent configuration drift across environments. Essential environment variables secure our application.

Render Blueprints

- Uses `render.yaml` to define infrastructure specifications like runtime and build commands.
- Ensures declarative infrastructure, preventing configuration drift and promoting reproducibility.

Service Configuration

- **Build Command:** `npm install` (ensures dependency consistency via lockfiles).
- **Start Command:** `node src/server.js`.

Critical Environment Variables

- **NODE_ENV:** Set to "production" for performance optimizations.
- **MONGODB_URI:** Secure MongoDB Atlas connection string.
- **MONGO_DB_NAME:** Specifies the target database instance.
- **ALLOWED_ORIGINS:** Frontend URL(s) for Cross-Origin Resource Sharing (CORS) configuration.
- **EMAIL_USER & EMAIL_PASS:** Credentials for email service integration.
- **EMAIL_PORT:** SMTP port for secure email transmission.
- **REDIS_URL:** Redis connection string for caching or session management.

Network Integration & CORS Strategy

Bridging the frontend on Vercel and the backend on Render requires careful management of browser security policies. Our robust CORS strategy ensures secure and seamless communication while adhering to the Same-Origin Policy (SOP).

01

The Challenge

The browser's Same-Origin Policy (SOP) inherently blocks direct requests between distinct origins (Vercel and Render) to prevent malicious cross-site scripting.

02

The Solution (CORS)

We implemented strict Cross-Origin Resource Sharing (CORS) middleware on the backend to explicitly allow secure cross-origin requests from our authorized frontend.

03

Configuration Logic

- **Allowed Origin:** Reads from `ALLOWED_ORIGINS` to whitelist trusted frontend domains.
- **Allowed Methods:** Restricted to standard HTTP verbs (GET, POST, PUT, DELETE) for controlled access.
- **Credentials:** `Access-Control-Allow-Credentials` set to `true` to support secure cookies and JWTs.

Layered Architecture & Middleware Chain

Our backend adopts a clear Model-Route-Controller (MVC) design pattern, separating concerns to enhance modularity and maintainability. A robust middleware pipeline ensures consistent request processing and error handling.

Design Pattern

Follows the Model-View-Controller (MVC) pattern to effectively decouple business rules from HTTP transport mechanisms, improving code organization and testability.

Directory Structure

- `src/models`: Defines Mongoose schemas for data structure and validation.
- `src/controllers`: Contains the core request processing logic and orchestrates data interactions.
- `src/routes`: Maps API endpoint definitions to specific controller functions, defining the API surface.

Middleware Pipeline

1. **Security:** `helmet` for setting secure HTTP headers against common vulnerabilities.
2. **CORS:** Handles cross-origin permissions, as detailed in the previous section.
3. **Parsing:** `express.json` for efficient parsing of incoming JSON payloads.
4. **Logging:** `morgan` for comprehensive request tracking and debugging.
5. **Error Handling:** A global error handler designed to return standardized JSON error responses, improving client-side error management.

UNITE BloodBank: Layered Backend Architecture

Our backend system is meticulously structured into distinct layers, ensuring robust security, maintainability, and scalability from the entry point to data persistence.

Logic & Data Abstraction

A modular design separates Controllers (request handlers) from Services (business logic). MongoDB Schemas (Models) define structured and scalable data storage.

Security & Integrity Layers

A comprehensive Middleware Layer enforces Authentication, Rate Limiting, and CORS policies. JOI validation meticulously ensures input data integrity.

Core Server & Entry Points

The Express Server serves as the central entry, efficiently handling standard HTTP requests via RESTful Routes and real-time events through Socket.IO WebSockets.

RESTful Principles & JWT Strategy

Our API is built on RESTful principles, utilizing standard HTTP methods and status codes for clear communication. Authentication is managed via a stateless JSON Web Token (JWT) strategy, ensuring secure and scalable user access.

REST Design

Uses standard HTTP verbs (GET, POST, PUT, DELETE) and precise status codes (e.g., 201 Created, 401 Unauthorized, 500 Internal Error) for clear communication.

Login

Client sends user credentials (username/password) to the `/api/auth/login` endpoint securely.

Verification

The backend validates the submitted credentials against the stored user collection in MongoDB.

Issuance

Upon successful validation, a JSON Web Token (JWT) is signed and issued, containing the User ID and Role.

Access

All protected routes subsequently require the client to present this token in the `Authorization` header (as `Bearer <token>`) for access.

Database Selection & Configuration

MongoDB Atlas, a managed NoSQL database, was chosen for its flexibility, scalability, and seamless integration with Node.js. Mongoose provides an Object Data Modeling (ODM) layer for schema enforcement and data validation.

Database Choice

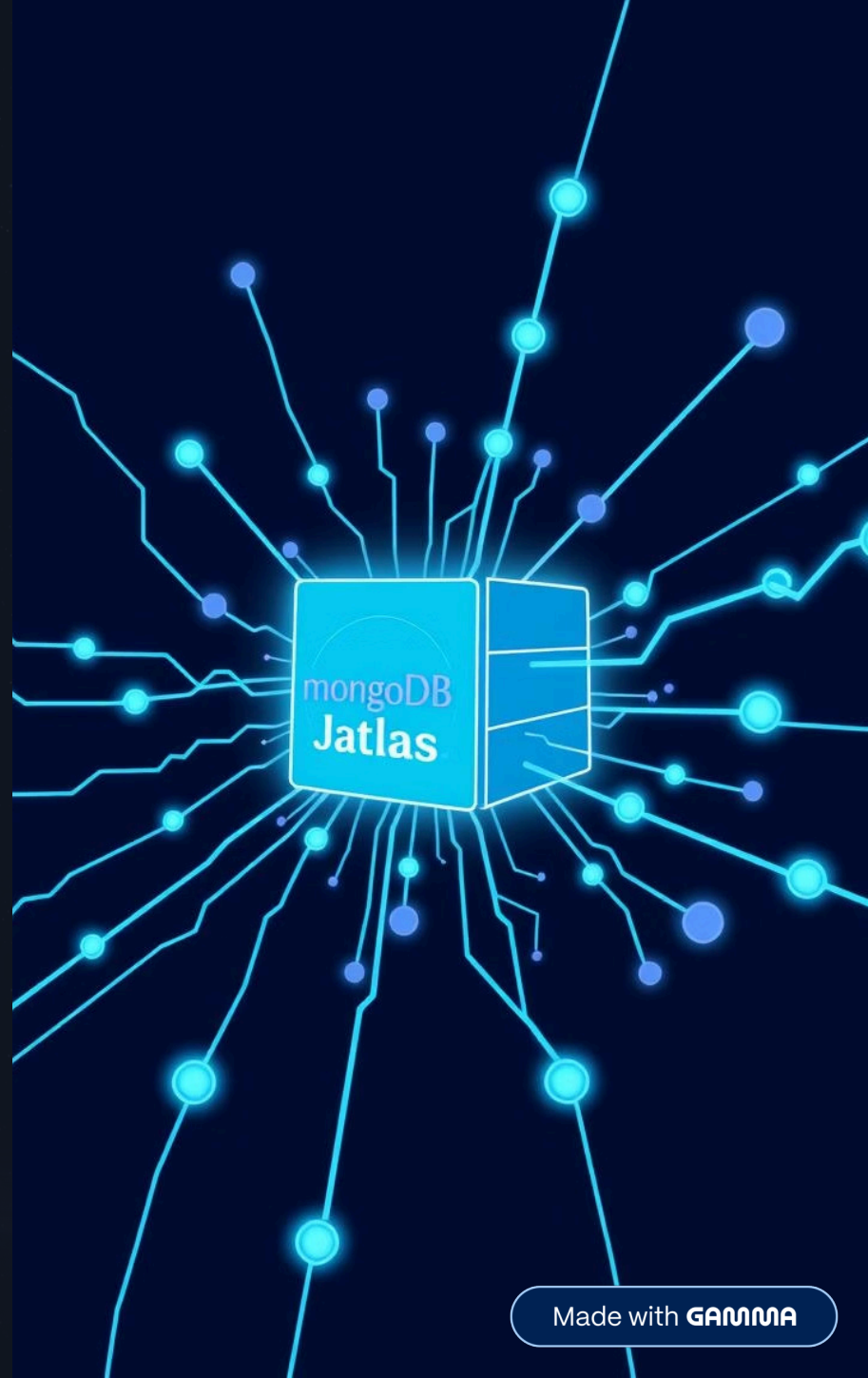
MongoDB Atlas (Managed NoSQL) selected for its robust document-oriented nature, providing unparalleled flexibility and native integration capabilities with Node.js.

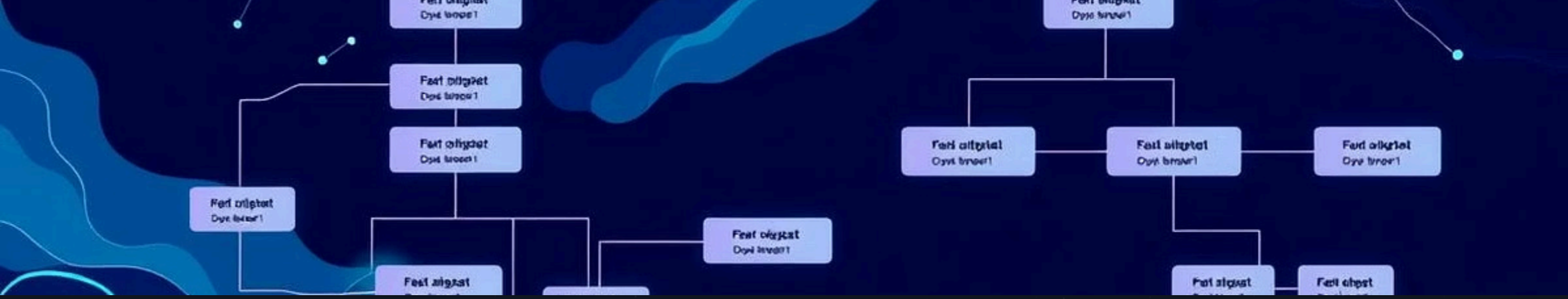
Object Data Modeling

Leverages Mongoose to define strict schemas, enforcing data integrity and applying validation rules directly to MongoDB documents, enhancing data quality.

Security Protocols

Ensures secure connections via standard driver strings (MONGODB_URI) and implements IP whitelisting, configured to restrict access only from authorized Render servers.





Core Entity Relationships

Our application’s data models are intricately designed to manage complex relationships, from geographic hierarchies to staff roles and event flows. These relationships form the backbone of our system's functionality.

Geographic Hierarchy

Organizes location data by linking Provinces, Districts, and Municipalities, enabling precise geographic filtering and management.

Staff Hierarchy

Defines clear roles (System Admin, Coordinator, Stakeholder) for BloodBank Staff, linked via Staff Identification for secure access control.

Event & Request Flow

Tracks the status, history, and approval workflows for various event types, including Blood Drives, Advocacy campaigns, and Training sessions.

Real-Time Chat

Facilitates seamless communication through `Chat_Message`, `Chat_Conversation`, and monitors user `Presence` (online/offline status) for enhanced interaction.

Reliability, Security & Future Scalability

Our commitment to reliability and security is evident in our robust logging and protection mechanisms. The future roadmap includes enhancements for automated testing, performance optimization, and architectural evolution towards microservices.

Observability

Structured JSON logging (INFO, WARN, ERROR) facilitates programmatic querying and real-time monitoring, crucial for rapid incident response.

Security Measures

- **Rate Limiting:** Prevents Denial of Service (DoS) and brute-force attacks.
- **Sanitization:** Inputs are rigorously sanitized to guard against NoSQL injection vulnerabilities.
- **TLS:** Transport Layer Security is managed robustly by Render at the load balancer level, ensuring encrypted communication.

Future Roadmap

- **Short-Term:** Implementation of automated testing (Jest) and comprehensive Swagger API documentation for improved developer experience.
- **Long-Term:** Integration of a Redis caching layer for performance gains and a strategic transition to a microservices architecture for enhanced scalability and resilience.

Thank You

This end the presentation about UNITE's backend architecture and design