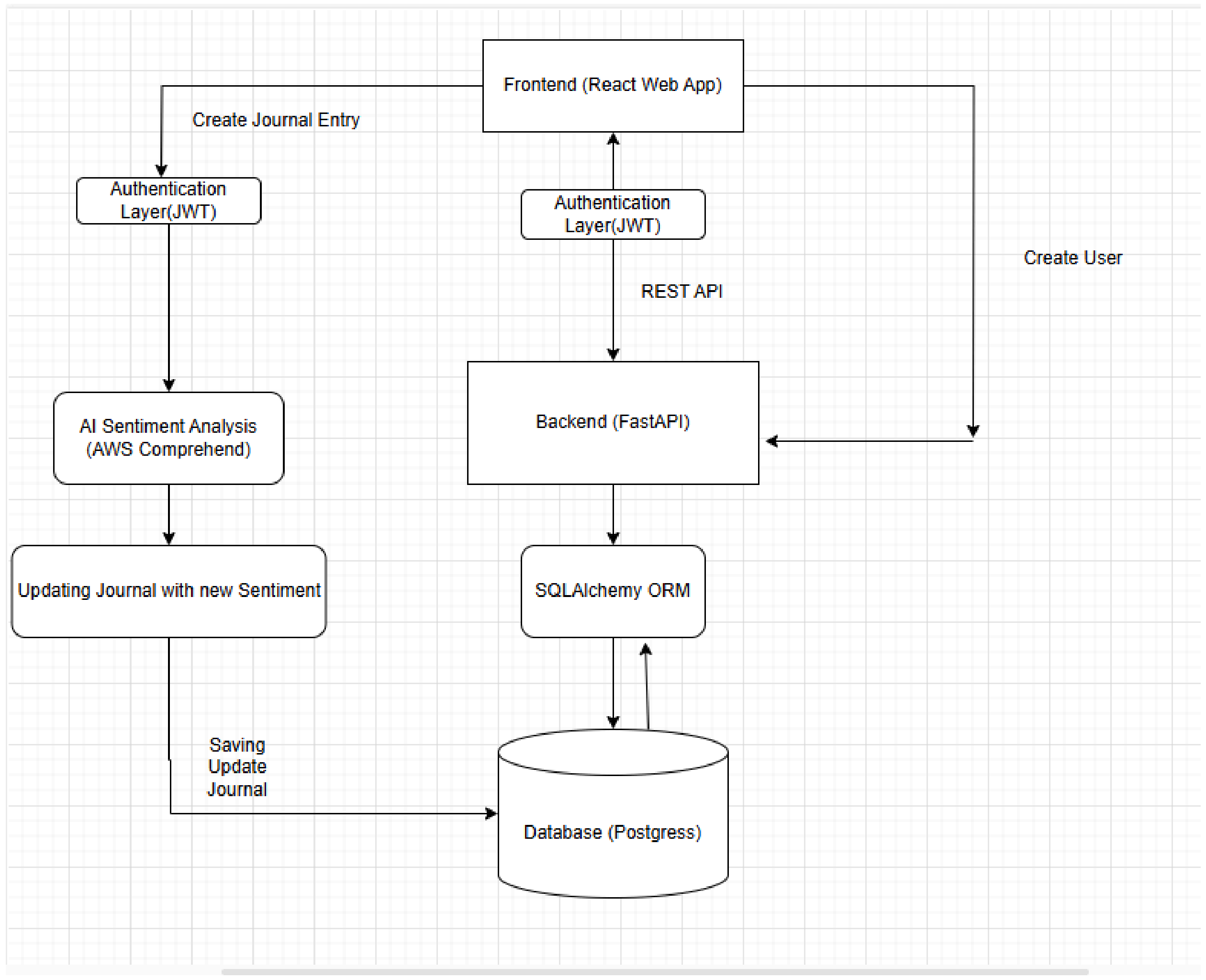
System Design and Implementation



The flow diagram illustrates the architecture of the journal application, detailing how user authentication, journal entry creation, and AI-powered sentiment analysis are integrated**.**

User Authentication

- Users authenticate via JWT (JSON Web Token) before accessing any functionalities.
- The frontend (React Web App) communicates with the authentication layer, which verifies the user's credentials**.**

Journal Entry Creation & Sentiment Analysis

- Authenticated users create journal entries via the frontend.
-
- The journal entry is processed by AWS Comprehend for sentiment analysis.
-
- The analyzed sentiment is then updated in the journal entry.
-
- Backend Processing (FastAPI & PostgreSQL)

The FastAPI backend handles all API requests and interacts with the SQLAlchemy ORM for database operations.

Once sentiment analysis is completed, the journal entry is updated with the new sentiment score and stored in the PostgreSQL databas**e.**

Data Storage & Retrieval

- **The PostgreSQL database stores both user and journal entry data.**
-
- **The frontend fetches journal entries via API calls, ensuring a seamless user experience.**

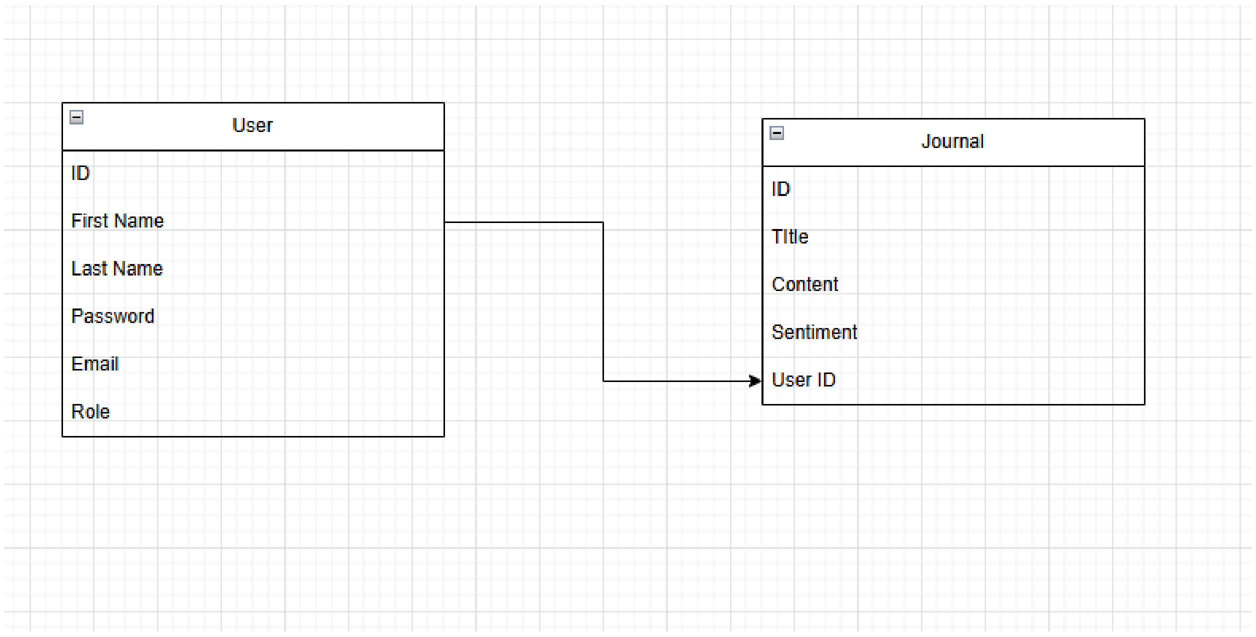
This architecture ensures secure authentication, seamless journal management, and automated AI-driven sentiment analysis, enhancing the overall user experience.

Data model design and relationships

In the design of this database, the relationship between the User and Journal tables follows a one-to-many (1:M) relationship. This means that a single user can create multiple journal entries, but each journal entry belongs to only one user.

Entity Relationship Breakdown

1. **User Table**
   - Represents users who have access to the journal system.
   - Each user has a unique ID to identify them.
2. **Journal Table**
   - Represents individual journal entries written by users.
   - Each entry includes a foreign key (User ID) linking it to a specific user.



Role-based access control architecture

We have role based access control which includes ADMIN and USER. Just to demo , try using these credentials on the Journal Web App you will be directed to an admin only page

email:admin@gmal.com

Password: admin

Security Measures

Q: How does your application implement comprehensive security beyond basic authentication?

A: My application employs a **comprehensive security strategy** that extends beyond basic authentication mechanisms like simple username-password validation. The key security implementations include:

1️⃣ JWT-Based Authentication

- We use **JSON Web Tokens (JWTs)** for authentication, ensuring a **stateless and scalable** approach.
- JWTs include the **user ID and other claims**, which the backend verifies on every request.

2️⃣ Secure Token Storage Using `sessionStorage`

- Instead of using `localStorage` (which is vulnerable to **XSS attacks** and retains tokens indefinitely), we store tokens in `sessionStorage`.
- This ensures that **tokens are cleared automatically** when the user closes their tab, reducing session hijacking risks.

3️⃣ Extracting User ID from the JWT Instead of Passing It in Requests

- We do **not** pass user IDs via **URL parameters or query strings**, reducing the risk of **ID tampering**.

- The backend extracts and verifies the **user ID directly from the JWT**, ensuring **data integrity and security**.

4️⃣ **Token Verification on Every Request**

- API endpoints require the **JWT to be included in the request headers**.
- The server **decodes and validates the token** before allowing access to protected resources.

5️⃣ **Additional Security Best Practices**

- **Rate limiting** to prevent brute-force attacks.
- **Strict CORS policies** to prevent unauthorized cross-origin access.
- **Password hashing (e.g., bcrypt)** to protect user credentials.
- **Role-based access control (RBAC)** to restrict access to sensitive actions.

This **multi-layered security approach** ensures that authentication is **not just basic auth** but rather a **robust, well-secured system** designed to protect user data. 🚀

- **Potential scaling challenges and solutions**
- **Database Bottlenecks**
  - **Challenge:** As the number of users grows, read and write operations to PostgreSQL may slow down.
  - **Solution:** Implement **read replicas** and **horizontal partitioning (sharding)** to distribute the load. Use **caching (Redis)** to reduce database queries for frequently accessed data.
- **Authentication Load**

    - **Challenge:** JWT authentication requires validation, which can become a bottleneck at high loads.
    - **Solution:** Offload authentication to a **dedicated authentication service** like AWS Cognito, Auth0, or Firebase Auth.
- **Backend API Performance**
    - **Challenge:** FastAPI's synchronous tasks (like sentiment analysis requests) may block other requests.
    - **Solution:** Use **asynchronous processing (Celery + Redis)** for background tasks like AI sentiment analysis, reducing API response time.
- **Scaling AI Sentiment Analysis**
    - **Challenge:** AWS Comprehend may introduce **latency** at scale.
    - **Solution:** Implement **batch processing** or **queue-based processing** (e.g., SQS or Kafka) to handle large-scale sentiment analysis.
- **Network Latency & Load Balancing**
    - **Challenge:** Increased API traffic can cause slow response times and failures.
    - **Solution:** Deploy **load balancers (AWS ALB, Nginx)** and use **CDN (Cloudflare, AWS CloudFront)** to optimize API and frontend requests.

Scaling to Support 1M+ Users

To handle 1M+ users efficiently, the architecture needs to be **modular and horizontally scalable**:

1. **Database Scaling**
    - Use **database sharding** to distribute user data across multiple PostgreSQL instances.
    - Add **read replicas** to handle increased read queries.
2. **Backend Scalability**

    - Deploy FastAPI with **Kubernetes (K8s)** for auto-scaling.
    - Use **serverless computing (AWS Lambda)** for specific tasks like sentiment analysis.
3. **Event-Driven Architecture**
    - Instead of synchronous API processing, adopt **event-driven microservices** (Kafka, RabbitMQ).
    - Separate journal creation, AI processing, and updates into different microservices.
4. **Edge Caching & CDN**
    - Store frequently accessed data (e.g., recent journal entries) in **Redis or Memcached**.
    - Use **CDN** for static assets and frontend files.

Technical Decision Log

FastAPI

1 The Technical Decision Log for choosing FastAPI as a Backend includes:

- Problem Solved : Need for a high-performance backend with robust data validation.
- Options Considered : Flask, Django, FastAPI, Node.js.
- Chosen Approach : FastAPI for its speed, async support, and Pydantic integration.
- Trade-offs : Smaller community and async learning curve balanced by performance and scalability.
- This decision ensures a modern, efficient backend foundation for the app.

AWS Comprehend For Sentiment Analysis

The Technical Decision Log for sentiment analysis includes:

- Problem Solved : Analyzing journal entry sentiment to track user well-being.
- Options Considered : Custom model, open-source libraries, AWS Comprehend, other cloud providers.
- Chosen Approach : AWS Comprehend for its scalability, accuracy, and AWS integration.
- Trade-offs : Vendor lock-in and costs balanced by ease of use and performance.
- This decision ensures accurate, scalable sentiment analysis while aligning with the app's architecture.

MVC(Model View Controller Architecture

The Technical Decision Log for choosing MVC includes:

Problem Solved : Need for a maintainable, scalable architecture with clear separation of concerns.

- Options Considered : Monolithic, MVC, MVVM, Microservices.
- Chosen Approach : MVC for its balance of simplicity, scalability, and separation of concerns.
- Trade-offs : Initial complexity balanced by long-term maintainability and scalability.
- This decision ensures a structured, future-proof foundation for the app.