System Design Document: Personal Journal App

# 1. Architecture Diagram & Explanation

Architecture Overview  
The system follows a three-tier architecture:  
1. Client Layer (Frontend): A web app interacting with the backend via API.  
2. Application Layer (Backend): A Node.js + Express.js server handling authentication, journal entries, and database interactions.  
3. Data Layer (Database): MySQL database managing users, journal entries, and categories.

Technology Stack:  
• Backend: Node.js, Express.js  
• Database: MySQL  
• Authentication: JWT (JSON Web Tokens)  
• Security Enhancements: bcrypt for password hashing, input validation

# 2. Data Model Design & Relationships

Tables & Relationships:  
• t\_users (User information, hashed passwords)  
• entries (Journal entries, linked to users)  
• t\_categories (Categorization of journal entries)

Schema Overview:  
t\_users (userid, username, password\_hash, createdon)  
entries (id, userid, title, content, category, created\_at)  
t\_categories (categoryid, categoryname)

Relationships:  
• One-to-Many: t\_users → entries  
• One-to-Many: t\_categories → entries

# 3. Security Measures Beyond Basic Authentication

• Token Expiry & Refresh Tokens: Implemented access & refresh tokens to improve session security.  
• Rate Limiting: Protect API endpoints from abuse (e.g., login brute-force attacks).  
• SQL Injection Prevention: Used prepared statements for database queries.  
• Data Encryption: Encrypt sensitive user data (e.g., passwords).  
• CORS & Helmet.js: Secure API by limiting cross-origin requests and enforcing security headers.

# 4. Potential Scaling Challenges & Solutions

Challenge Solution  
High Read/Write Load Use database indexing & read replicas  
Increased Authentication Requests Implement token-based authentication & caching  
API Latency Use load balancers & caching (Redis)  
Large Number of Users Deploy a scalable cloud solution (AWS/GCP/Azure)  
Database Bottlenecks Optimize queries, use sharding & partitioning

# 5. Scaling to 1M+ Users

• Database Optimization:  
 o Use horizontal scaling (sharding, read replicas).  
 o Implement connection pooling for MySQL.  
• API Performance:  
 o Use Redis caching for frequently accessed data.  
 o Implement GraphQL or optimized REST API endpoints.  
• Infrastructure & Deployment:  
 o Deploy using Kubernetes/Docker for auto-scaling.  
 o Use CDN for frontend assets.

# 6. Potential Bottlenecks & How to Address Them

Bottleneck Solution  
High DB Load Implement read replicas, indexing, and query optimization  
Slow API Response Use Redis caching & load balancers  
Single Server Failure Deploy on multiple instances using Kubernetes  
Authentication Delays Use JWT with session storage (Redis)

# 7. Components That Might Need Redesign at Scale

• Database Architecture: Move from single MySQL instance to distributed database setup.  
• Authentication Handling: Implement OAuth2.0 for third-party authentication.  
• API Gateway: Introduce GraphQL or gRPC for efficient API interactions.  
• Logging & Monitoring: Implement centralized logging with ELK Stack or Prometheus.

# 8. Key Technical Decisions

## 1. Authentication Method (JWT vs. Sessions)

Problem:  
• Needed a scalable authentication method that supports stateless API interactions.  
Options Considered:  
• Session-based authentication (Stores session on the server).  
• JWT (JSON Web Tokens) (Stateless token-based authentication).

Chosen Approach & Rationale:  
• Used JWT for authentication because it allows scalable stateless authentication without requiring session storage.  
Trade-offs & Consequences:  
• Pro: More scalable than session-based authentication.  
• Con: Requires token expiration and refresh token handling to avoid security risks.

## 2. Database Choice (MySQL vs. NoSQL)

Problem:  
• Needed a structured and relational way to store user and journal entry data.  
Options Considered:  
• MySQL (Relational Database): Provides strong consistency.  
• MongoDB (NoSQL Database): More flexible but lacks relational integrity.

Chosen Approach & Rationale:  
• Used MySQL because relational integrity was essential for linking users and journal entries.  
Trade-offs & Consequences:  
• Pro: Strong data consistency and ACID compliance.  
• Con: May require sharding when scaling to a very high number of users.

## 3. Password Hashing (bcrypt vs. Argon2)

Problem:  
• Needed to securely store passwords.  
Options Considered:  
• bcrypt: Well-tested hashing algorithm.  
• Argon2: More advanced but less widely supported.

Chosen Approach & Rationale:  
• Used bcrypt due to its wide adoption and security features.  
Trade-offs & Consequences:  
• Pro: Strong encryption and well-documented best practices.  
• Con: Slightly slower than Argon2 for hashing.

## 4. API Design (REST vs. GraphQL)

Problem:  
• Needed an API that could efficiently serve client requests.  
Options Considered:  
• REST API: Standard endpoint-based API.  
• GraphQL: More flexible querying capabilities.

Chosen Approach & Rationale:  
• Used REST API because of its simplicity and ease of integration.  
Trade-offs & Consequences:  
• Pro: Easier to implement and widely supported.  
• Con: Less flexible compared to GraphQL in terms of fetching specific fields.

## 5. Rate Limiting (IP-based vs. User-based)

Problem:  
• Needed to prevent brute-force attacks and API abuse.  
Options Considered:  
• IP-based rate limiting: Limits requests per IP.  
• User-based rate limiting: Limits requests per user account.

Chosen Approach & Rationale:  
• Used IP-based rate limiting to prevent DoS attacks while keeping user-based limits as an option.  
Trade-offs & Consequences:  
• Pro: Protects against automated attacks.  
• Con: May block legitimate users behind shared IPs.

# 9. Authentication Strategy

Chosen Approach: JSON Web Tokens (JWT)

How It Works:  
• User logs in with email & password.  
• Server verifies credentials and generates a JWT.  
• JWT is sent to the client and used for subsequent API requests.  
• The backend validates the JWT before processing requests.  
• JWT includes an expiration time and refresh token system for security.

Why JWT?  
• Stateless: No session storage required.  
• Scalable: Works well in distributed architectures.  
• Security: Can be signed and encrypted.

Enhancements for Security:  
• Short-lived access tokens with refresh tokens.  
• Blacklist compromised tokens.  
• Use HTTPS to prevent MITM attacks.

# 9. Instructions on how to run the application

**1**. **Clone the Repository**

**2**. **Install Dependencies by running npm install**

**3. Set Up the Database**

**Ensure MySQL is installed on your local machine**

* **Create a new MySQL database (e.g., journal\_db).**
* **Update the database credentials in the .env file (if it exists) or directly within the configuration file.**

**4. Run the Application**

**Once the dependencies and database are set up, run the application using the following command: npm start**

**5. Access the Application**

**Open your browser and run the link**