# Journaling App - System Design

## 1. Overview

The Personal Journaling App empowers users to securely record their thoughts and experiences, categorize their entries, and gain valuable AI-driven insights into their writing patterns. This application utilizes a full-stack Next.js framework, a robust PostgreSQL database managed through Prisma ORM, secure JWT-based authentication, and leverages Ollama for on-device AI processing. This architecture ensures a scalable, privacy-focused, and efficient journaling experience.

## 2. System Architecture

### 2.1 High-Level Design

```
+----------------------+      +-------------------------+      +------------------
----+
|  User Interface     | ----> |  Backend API (Next.js)   | ----> | PostgreSQL
Database |
+----------------------+      +-------------------------+      +------------------
----+
                                        |
                                        v
                                +----------------+
                                |  Ollama AI API  |
                                +----------------+
```

### 2.2 Why Ollama for AI Processing?

✅ On-Device AI Processing: Eliminates the need for cloud-based AI APIs, significantly reducing latency and operational costs.

✅ Privacy-Focused: Ensures that sensitive journal data remains within the user's local environment, enhancing data security and privacy.

✅ Offline Capability: Enables AI-powered features to function even without an active internet connection.

✅ Fast Response Time: Local execution of AI models by Ollama minimizes API call delays, providing quicker insights.

✅ Open-Source & Customizable: Offers the flexibility to easily fine-tune AI models to better suit individual journaling preferences and needs.

### 2.3 Comparison of AI Model Choices

| AI Model | Pros | Cons |
|---|---|---|
| Ollama (local LLMs) | Private, fast, offline | Requires initial device setup and resource usage |
| OpenAI (GPT-4 API) | Powerful, no local setup required | Costs per request, potential privacy considerations |

|  |  |  |
|---|---|---|
| Hugging Face Inference API | Wide range of models, free tier available | Can be slower than local inference for some models |

## 3. Data Model Design

### 3.1 Journals Table

| Column | Type | Constraints | AI-Powered? | Description |
|---|---|---|---|---|
| id | UUID | Primary Key, Auto-Generated | ❌ | Unique identifier for each journal entry |
| title | String | Required | ❌ | Title of the journal entry |
| contentText | String | Required | ❌ | Main content of the journal entry |
| category | String | ENUM (e.g., Work, Personal, Travel) | ✅ (Auto-categorized) | Category of the journal entry |
| userId | UUID | Foreign Key to User | ❌ | Identifier of the user who created the entry |
| createdAt | DateTime | Default: now() | ❌ | Timestamp when the entry was created |
| updatedAt | DateTime | Auto-updated | ❌ | Timestamp when the entry was last updated |
| summary | String? |  | ✅ (AI-generated) | Concise AI-generated summary of the entry |
| sentiment | String? | "Positive", "Neutral", "Negative" | ✅ | AI-analyzed sentiment of the entry |
| suggestions | String? |  | ✅ | AI-driven writing prompts or suggestions related to the entry |

### 3.2 Users Table

| Column | Type | Constraints |
|---|---|---|
| id | UUID | Primary Key, Auto-Generated |
| name | String? | Nullable |
| email | String | Unique, Required |
| emailVerified | DateTime? | Nullable |
| image | String? | Nullable (Profile Picture) |
| password | String | Required (Hashed) |

| | | |
|---|---|---|
| createdAt | DateTime | Default: `now()` |
| updatedAt | DateTime | Auto-updated |
| accounts | Relation | One-to-Many with `Account` |
| journals | Relation | One-to-Many with `Journal` |

### 3.3 Account Table *(For OAuth login)*

| Column | Type | Constraints |
|---|---|---|
| id | UUID | Primary Key |
| userId | UUID | Foreign Key to `User` |
| provider | String | OAuth Provider (e.g., Google, GitHub) |
| providerAccountId | String | Unique ID from provider |
| access_token | String? | OAuth Access Token |
| refresh_token | String? | OAuth Refresh Token |

### 3.4 Sessions Table

| Column | Type | Constraints |
|---|---|---|
| id | UUID | Primary Key |
| sessionToken | String | Unique |
| userId | UUID | Foreign Key to `User` |
| expires | DateTime | Expiration Date |

---

## 4. API Endpoints

Note: For a comprehensive list of all API endpoints and their specifications, including request and response bodies, please check the Swagger documentation:

📌 VISIT http://localhost:3000/docs

---

### 4.1 Journal Entry CRUD API

- `GET /api/journal` → Retrieve all journal entries for the authenticated user.
- `POST /api/journal` → Create a new journal entry.
- `GET /api/journal/{id}` → Retrieve a specific journal entry by its ID.
- `PUT /api/journal/{id}` → Update an existing journal entry.

- `DELETE /api/journal/{id}` → Delete a journal entry.

📌 Authentication Required: These endpoints require authentication using a Bearer Token (JWT).

### 4.2 AI-Powered Insights API

- `POST /api/ai` → Processes AI-powered insights based on the given journal entry content.
    - `promptType=category` → Suggests relevant categories.
    - `promptType=sentiment` → Returns sentiment analysis (positive, neutral, or negative).
    - `promptType=suggestions` → Provides AI-driven writing prompts or suggestions.

### 4.3 Authentication API (Signup & Login)

- `POST /api/auth/signup` → Register a new user.
    - Request Body: `{ "name": "John Doe", "email": "johndoe@email.com", "password": "#Password123" }`
- `POST /api/auth/login` → Authenticate user and return a JWT.
    - Request Body: `{ "email": "johndoe@email.com", "password": "#Password123" }`
- `GET /api/auth/session` → Get the current authenticated user session.
- `POST /api/auth/logout` → Logout and invalidate the session.

📌 Authentication: Login returns a JWT token, which must be included in the `Authorization` header (`Bearer <token>`) for protected routes.

---

## 5. Ollama AI Implementation

### 5.1 Ollama Setup

Ollama is designed to run locally on the user's device, ensuring privacy and speed for AI processing.

### Installation

Install Ollama on your Mac or Linux system using the following command:

```
curl -fsSL [https://ollama.ai/install.sh](https://ollama.ai/install.sh) | sh
```

### Running Ollama Model (Example: Llama 3 - Consider Phi as an alternative)

To start Ollama with a specific model, for example, phi or llama3 *(I used Phi 2.5B)*:

```
ollama run phi
   # OR
ollama run llama3
```

Ensure the desired model ( `phi` or `llama3`) is pulled by Ollama.

## 6. Authentication & Security

✅ JWT Authentication (Stored in HTTP-only cookies): Provides secure and stateless authentication for API requests after user login. HTTP-only cookies mitigate the risk of client-side JavaScript accessing the token.

✅ Bcrypt for Password Hashing: Securely stores user passwords by using a strong hashing algorithm with salt.

✅ OAuth Support via NextAuth.js: Enables users to authenticate using third-party providers like Google and GitHub, reducing the need to manage passwords directly.

✅ CSRF Protection for API Requests: Protects against cross-site request forgery attacks by using techniques like synchronizer tokens. Next.js and NextAuth.js often provide built-in mechanisms for this.

✅ Rate Limiting for Login Attempts: Prevents brute-force attacks on user accounts by limiting the number of login attempts from a specific IP address within a certain timeframe.

✅ Input Validation: Implemented robust server-side validation on all incoming data to prevent injection attacks Use zod library for schema validation.

✅ HTTPS Enforcement: *(TODO)* Ensure that the application is served over HTTPS to encrypt all communication between the client and the server, protecting sensitive data in transit.

✅ Dependency Management:*(TODO)* Keep all dependencies (Node.js libraries, Prisma, Next.js) up-to-date to benefit from security patches without breaking the application due to mismatch

## 7. Scaling Considerations

### 7.1 Why Ollama is Scalable

- Local AI Processing: Offloads AI computation from the backend servers to individual user devices, eliminating per-request AI costs.
- Horizontal Scaling: As the user base grows, the AI processing scales with the number of users and their devices, rather than requiring more powerful central servers.
- Edge Caching: Next.js Incremental Static Regeneration (ISR) can cache responses that might involve AI computations, reducing the need for repeated processing.

### 7.2 Additional Optimizations

- Indexes: Implementing an index on the `userId` column in the `Journals` table will significantly speed up data retrieval for specific users.
- Partitioning: For very large datasets, consider partitioning the `Journals` table based on the `createdAt` timestamp to improve query performance.
- Read Replicas: Setting up PostgreSQL read replicas can enhance read availability and distribute the load on the primary database.

### 7.3 Scaling to Support 1M+ Users

To support 1M+ active users, the following scaling strategies would be crucial:

- Horizontal Scaling of Backend: Deploy multiple instances of the Next.js backend API behind a load balancer (e.g., AWS ELB, Nginx). Ensure the API is stateless.
- Database Scaling:
    - Read Replicas: Significantly increase the number of read replicas to handle the increased read traffic.
    - Database Partitioning: Implement robust partitioning strategies on the `Journals` table based on `userId` or `createdAt` to distribute data and query load.
    - Consider Sharding: For extreme scale, consider database sharding, where the database is split across multiple independent servers. This adds complexity but can handle massive datasets.
- Caching: Implement a comprehensive caching strategy using Redis for frequently accessed data and computation results. Optimize cache invalidation strategies.
- CDN for Static Assets: Utilize a Content Delivery Network (CDN) like Cloudflare or AWS CloudFront to serve static assets (JavaScript, CSS, images) closer to users, reducing latency and load on the backend servers.
- Asynchronous Task Processing: For non-critical background tasks (e.g., generating historical summaries), use a message queue (e.g., RabbitMQ, Kafka) and worker processes to avoid blocking API requests.
- Monitoring and Alerting: Implement robust monitoring and alerting systems to track key metrics (CPU usage, memory usage, database performance, error rates) and proactively identify and address potential issues.

## 8. Potential Bottlenecks and How to Address Them

- Database Read/Write Performance: With a large number of users and entries, the PostgreSQL database could become a bottleneck.
    - Solution: Implement indexing, partitioning, read replicas, and potentially sharding as described in the scaling section. Optimize database queries.
- Backend API Processing: Handling a large volume of API requests could strain the backend servers.
    - Solution: Horizontal scaling of the backend API behind a load balancer. Optimize API endpoints and code for performance. Implement caching.
- AI Processing on User Devices: While Ollama distributes the AI load, users with older or less powerful devices might experience slower processing times.
    - Solution: Clearly communicate system requirements. Offer the optional cloud-based AI integration for users who need more power or prefer not to run AI locally.
- Network Latency: Users geographically distant from the backend servers might experience higher latency.
    - Solution: Utilize a CDN to serve static assets closer to users. Deploy backend API instances in multiple geographic regions if necessary.

- Cache Invalidation: Incorrect cache invalidation can lead to stale data.
    - Solution: Implement effective cache invalidation strategies based on data changes.

## 9. Components That Might Need Redesign at Scale

- Database Schema: As the application evolves and data requirements change at scale, the database schema might need to be redesigned for better performance and scalability (e.g., further normalization or denormalization, different data types).
- API Endpoints: Some API endpoints might become inefficient at scale. They might need to be redesigned to support pagination, filtering, and more efficient data retrieval. Consider using GraphQL for more flexible data fetching.
- Real-time Features: If real-time features (e.g., collaborative journaling) are added, the architecture would need to incorporate technologies like WebSockets and potentially a dedicated real-time communication service.
- Search Functionality: Basic database queries might become too slow for searching through a large number of journal entries. A dedicated search engine like Elasticsearch might be necessary.

## 10. Technical Decision Log

| Decision | Options Considered | Chosen Approach | Rationale |
|---|---|---|---|
| Backend | Express.js, Next.js API | Next.js API Routes | Simpler full-stack integration, server-side rendering. |
| Authentication | JWT, Session-based | JWT | Scalable, stateless, suitable for API-driven applications. |
| Database | MySQL, PostgreSQL | PostgreSQL | Better suited for structured journaling data, advanced features. |
| ORM | TypeORM, Prisma | Prisma | Type-safe, easy migrations, excellent developer experience. |
| AI Processing | OpenAI, Ollama, Hugging Face | Ollama | Prioritizes privacy, speed, and offline capabilities. |

### Decision 1: AI Processing Location

- Problem: How to integrate AI capabilities while prioritizing user privacy and minimizing backend costs.
- Options Considered:
    - Cloud-based AI APIs (e.g., OpenAI, Google Cloud AI): Powerful AI models, easy to integrate initially.
    - On-device AI processing (Ollama): Enhanced privacy, lower latency, no per-request costs.

    - Hybrid approach: Use cloud AI for some tasks and local AI for others.
- Chosen Approach: On-device AI processing with Ollama as the primary solution.
- Rationale: Prioritizes user privacy by keeping journal data local. Reduces backend costs associated with AI API calls. Offers offline functionality.
- Trade-offs and Consequences: Requires users to install and run Ollama locally, which might be a barrier for some. Local AI models might have limitations compared to the most advanced cloud models. Initial setup might be slightly more involved for the user.

### Decision 2: Backend Framework

- Problem: Choosing a backend framework that supports full-stack development, efficient API creation, and server-side rendering.
- Options Considered:
    - Express.js: Minimalist framework, highly flexible.
    - Next.js: Full-stack framework with built-in support for routing, server-side rendering, API routes, and more.
    - NestJS: Progressive Node.js framework with a structured architecture (Angular-like).
- Chosen Approach: Next.js.
- Rationale: Enables rapid development with its full-stack capabilities. Server-side rendering improves initial load times and SEO. API routes simplify backend development. Large and active community.
- Trade-offs and Consequences: Can have a steeper learning curve compared to basic Express.js for simple APIs. The "magic" of Next.js can sometimes make debugging more complex.

### Decision 3: Database ORM

- Problem: Selecting an Object-Relational Mapper (ORM) to interact with the PostgreSQL database.
- Options Considered:
    - Sequelize: Mature and widely used ORM.
    - TypeORM: TypeScript-first ORM with decorator-based syntax.
    - Prisma: Modern ORM with a focus on developer experience, type safety, and database migrations.
- Chosen Approach: Prisma.
- Rationale: Provides excellent type safety, auto-generated database client, and a smooth migration process. Developer-friendly schema definition and querying.
- Trade-offs and Consequences: Relatively newer compared to Sequelize, but has gained significant traction. The Prisma schema is specific to Prisma.

## Decision 4: Authentication Strategy

- Problem: Implementing a secure and scalable authentication mechanism.
- Options Considered:
    - Session-based authentication: Simple to implement initially but can be challenging to scale for stateless APIs.
    - JWT (JSON Web Tokens): Stateless, scalable, suitable for microservices.
- Chosen Approach: JWT Authentication stored in HTTP-only cookies, combined with OAuth via NextAuth.js.
- Rationale: JWT provides a stateless and scalable authentication mechanism. HTTP-only cookies enhance security by preventing client-side JavaScript access to the token. OAuth simplifies user onboarding by allowing login with existing accounts.
- Trade-offs and Consequences: JWT requires careful management of the secret key. Revoking tokens can be more complex than invalidating sessions (though short-lived tokens and refresh tokens mitigate this).

## Code Quality Focus Areas

- Test Coverage: Implement comprehensive test suites including:
    - Unit Tests (Jest): Test individual functions and components in isolation to ensure they behave as expected. Aim for high code coverage in critical areas.
    - Integration Tests (Supertest/Playwright): Test the interaction between different parts of the system, such as API endpoints interacting with the database.
    - End-to-End Tests (Playwright): Simulate real user scenarios to verify the complete application flow, covering UI interactions and backend logic.
- Error Handling Strategies:
    - Implement robust error handling in backend API routes to catch exceptions gracefully.
    - Return meaningful error messages to the client, including appropriate HTTP status codes.
    - Log errors on the server for debugging and monitoring.
    - Implement client-side error handling to provide informative feedback to the user.
    - Use try-catch blocks and specific error type handling where appropriate.
- Performance Optimization Techniques:
    - Implement database query optimization (indexing, efficient queries).
    - Utilize caching (Redis, Next.js ISR) to reduce database load and improve response times.
    - Optimize frontend code for rendering performance (code splitting, lazy loading).
    - Compress assets (images, CSS, JavaScript).

    - Monitor application performance and identify bottlenecks using tools like Datadog or New Relic.
- Code Architecture, Documentation, and Organization:
    - Follow a clear and consistent code style (e.g., using ESLint and Prettier).
    - Organize code into logical modules and directories based on functionality (e.g., `pages/api`, `lib`, `components`).
    - Write clear and concise comments to explain complex logic.
    - Document API endpoints (ideally with Swagger/OpenAPI).
    - Maintain a well-structured project with clear separation of concerns (e.g., data access layer, business logic layer, presentation layer).
    - Use meaningful variable and function names.
    - Keep functions and components small and focused.
    - Follow SOLID principles where applicable.

## 8. Testing Strategy

- Unit Tests (Jest): Focus on testing the core logic of individual components and utility functions.
- Integration Tests (Supertest): Verify the correct interaction between different parts of the system, especially database interactions.
- End-to-End Tests (Playwright): Ensure the entire user flow works as expected, simulating real user scenarios.

## 9. Deployment Plan

| Component | Deployment Service |
|---|---|
| Frontend | Vercel |
| Backend | Vercel / AWS Lambda |
| Database | Supabase / AWS RDS (PostgreSQL) |
| AI Processing | Ollama (Local - User's Device) |

## 10. Setup Instructions

*(Please Check ReadMe.md also)*

### Prerequisites

Ensure you have the following installed on your system:

- Node.js (LTS version recommended)
- PostgreSQL (Ensure it's running on your system, default port is 5432)

- [Ollama AI](Ollama AI) (For AI-powered journaling features)

## Clone the Repository

```
git clone
[https://github.com/ortupik/journal.git](https://github.com/ortupik/journal.git)
cd journal
```

## Install Dependencies

```
npm install
```

## Setup Environment Variables

Create a `.env` file in the root directory and add the following configuration:

```
# Database Configuration
DATABASE_URL="postgresql://postgres:postgres@localhost:5432/journaldb?schema=public"
SHADOW_DATABASE_URL="postgresql://postgres:postgres@localhost:5432/journaldb?schema=p
ublic"

# NextAuth Configurations (OAuth Authentication)
GOOGLE_CLIENT_ID="your-google-client-id"
GOOGLE_CLIENT_SECRET="your-google-client-secret"
GITHUB_CLIENT_ID="your-github-client-id"
GITHUB_CLIENT_SECRET="your-github-client-secret"
NEXTAUTH_SECRET="your-random-secret-key"

# AI - OLLAMA Configuration
OLLAMA_API_URL="http://localhost:11434/api/generate"
OLLAMA_MODEL="phi:latest" # Or your preferred local model
```

Replace the placeholder values with your actual credentials.

## Setting Up PostgreSQL Database

1. Start PostgreSQL and ensure it's running on port 5432 (default).
2. Create the database manually:

   ```
   psql -U postgres -h localhost -p 5432
   CREATE DATABASE journaldb;
   ```

3. Generate Prisma client:

   ```
   npx prisma generate
   ```

4. Run Prisma Migrations:
   To create and update your database schema based on your Prisma schema, use the following commands:

   ```
   npx prisma migrate deploy                    # Apply pending migrations to
   the database (production)
   npx prisma migrate reset                     # Reset your database and re-
   apply migrations (development)
   npx prisma migrate status                    # Check the status of your
   migrations
   ```

5. Seed the Database :
   To populate your database with initial data, you can use the Prisma seed functionality. Create a `prisma/seed.ts` file and define your seeding logic. Then run:

```
npx prisma db seed
```

6. Default Login Credentials:
   After seeding the database, you can log in using the following default credentials (if seeding is implemented):

   - Email: `johndoe@email.com`
   - Password: `#Password123`

## Running the Application

To start the development server, run:

```
npm run dev
```

The application will be available at `http://localhost:3000/`

## Running Ollama AI (Phi Model)

7. Install Ollama:

```
curl -fsSL [https://ollama.ai/install.sh](https://ollama.ai/install.sh) | sh
```

8. Start Ollama with the Phi model (or the model specified in your `.env`):

```
ollama run phi
```

9. Ensure Ollama is running and accessible at `http://localhost:11434`.

## Building for Production

To build the application for production:

```
npm run build
```

To start the production server:

```
npm start
```

## Linting and Formatting

Ensure code quality by running:

```
npm run lint
npm run format
```

## Deployment

For deploying the application, follow these steps:

1. Set up environment variables on your server environment.
2. Run database migrations in production:

```
npx prisma migrate deploy
```

3. Build and start the application:

```
npm run build && npm start
```

## Conclusion

This comprehensive documentation outlines the system design and provides detailed setup instructions for the Personal Journaling App. By leveraging Next.js, PostgreSQL, Prisma, and Ollama, this application delivers a secure, scalable, and AI-enhanced journaling experience focused on user privacy. 🚀