Shamiri Interview Project Documentation

Introduction

This document outlines the design and implementation strategy for a personal journaling application.The primary goal is to provide a secure, user-friendly platform where individuals can write, categorize, and analyze their journal entries. The document is intended for software architects, developers, and technical stakeholders involved in the project.

System overview

The journaling application provides a secure, intuitive platform for users to create, manage, and analyze their personal journal entries. The system allows users to:

Register and Log In: Securely create accounts and authenticate using robust mechanisms.
Manage Journal Entries: Write, edit, and delete journal entries, each tagged with a title, content, category, and date.
Categorize and Tag: Organize entries by predefined categories (e.g., Personal, Work, Travel) or custom tags.
View Journals: Browse and search through past entries with ease.
Analyze Data: Access analytical summaries and visual dashboards to understand writing patterns, category distributions, word counts, and more.

Architecture Design

Key Components

The application is built with a modern microservices architecture to ensure scalability, maintainability, and performance:

Frontend (React):

Developed using React to provide a dynamic, responsive user interface.
Handles user interactions, data visualization, and integrates seamlessly with backend APIs.

Backend (NestJS, TypeORM, Postgres):

User Microservice: Manages user registration, authentication, and profile management.
Journal Microservice: Handles CRUD operations for journal entries, including creation, updates, deletions, and retrieval of entries.
Analysis Microservice: Processes data for the dashboard, generating analytical summaries and visual insights from journal entries.

Database (Postgres):

Centralized relational database managed via TypeORM.
Stores user information, journal entries, and related metadata ensuring data integrity and efficient querying.

Reverse Proxy (Caddy):

Routes incoming requests to the appropriate microservice.
Provides load balancing, TLS termination, and improved security for the system.

Message Broker (RabbitMQ):

Facilitates asynchronous communication between microservices.
Ensures reliable message delivery for tasks such as updating dashboards and processing background jobs.

Data flow

The application is structured to manage user authentication, journal operations, and data analysis independently. Data flows seamlessly between these services, ensuring that each component fulfills its designated role while maintaining overall system coherence.

1. User Service
    a. Purpose :
        i. The User Service is responsible for managing user authentication and account creation. This service verifies user credentials, handles registration, and ensures secure access to the system.
    b. Data Flow:
        i. Upon registration or login, the User Service processes and validates the user's credentials.

    c. Successful authentication allows the user to access other services within the application, specifically enabling journal creation and management.
2. Journals Service
    a. Purpose:
        i. The Journals Service handles all CRUD (Create, Read, Update, Delete) operations for journal entries. It is the core component for managing user-generated content related to journals.
    b. Data Flow:
        i. Users create and manage their journal entries through this service.
        ii. Each creation event triggers the Journals Service to generate an event, signaling that a new journal entry has been added to the system.
        iii. This event acts as a notification for downstream processes, ensuring that the new data is immediately available for further analysis.
3. Analysis Service
    a. Purpose:
        i. The Analysis Service is designed to perform detailed analysis on the journal entries. Its primary function is to process and analyze the content of journals without interfering with the operational data stored by the Journals Service.
    b. Data Flow:
        i. The Analysis Service continuously monitors for events generated by the Journals Service.
        ii. When a new journal entry is created, the Analysis Service receives the event.
        iii. Upon receipt, it creates a copy of the journal entry. This copy is used exclusively for analysis, thereby isolating analytical processes from the primary data store.
        iv. This separation of concerns ensures that analytical operations do not affect the performance or integrity of the Journals Service.

Relevant documents:
Data Flow Diagrams
Entity Relationship Diagram

Security Measures

JWT Token Whitelisting

JWT token whitelisting involves maintaining an approved list of valid tokens. When a user authenticates successfully, a JWT is issued and added to this whitelist. Every subsequent request is then verified against the list, ensuring that only tokens that are explicitly approved are permitted access.

Key Processes

Token Verification:
Each JWT presented by a client is cross-checked against the whitelist. Only tokens found on the list are considered valid, providing an extra layer of security beyond basic token validation.

Revocation and Expiration Management:
The system can revoke any token by removing it from the whitelist, irrespective of its expiration time. This allows for rapid response in the event of token compromise or suspicious activity.

Monitoring and Auditing:
The whitelist serves as a centralized point for monitoring token usage. Any anomalies, such as tokens not present on the list, trigger alerts for further investigation.

JWT token whitelisting significantly enhances the security of an application by ensuring that only approved tokens are allowed access. This measure not only prevents unauthorized use—by blocking intercepted or duplicated tokens—but also acts as an effective safeguard against token forgery and replay attacks. Additionally, the ability to swiftly revoke compromised or unnecessary tokens provides administrators with granular control over token management, which is crucial in dynamic environments where rapid responses to potential threats are essential.

Potential scaling challenges and solutions

The design is engineered to scale efficiently and support over one million users by decoupling critical components within the application. By separating the analysis functionality from the Journals service, the system ensures that user interactions remain seamless and responsive, regardless of the workload.

This decoupled architecture not only guarantees a smooth user experience but also provides the flexibility to allocate additional resources to the Analysis service. Consequently, the Analysis component can undertake more computationally intensive tasks, such as machine learning, without impacting the performance of core journal operations. This modular approach enhances overall system scalability and reliability as the user base grows.

Potential bottlenecks and how you would address them

As the system scales to support a large user base, identifying potential performance bottlenecks is critical to ensuring a seamless user experience. While RabbitMQ effectively manages communication between services, one potential bottleneck remains in the area of background task management.

Currently, RabbitMQ handles inter-service communication well; however, the absence of a dedicated background task management system—such as Celery—could create performance issues. Without a robust mechanism to handle asynchronous background jobs, resource-intensive tasks may block the main application flow. This bottleneck is particularly evident during peak loads or when executing tasks like data analysis or machine learning operations, where synchronous processing could lead to delays and reduced responsiveness.
To address this bottleneck, the following strategies are recommended:

Implement a Background Task Manager:
Integrate a system like Celery to complement RabbitMQ by offloading long-running tasks from the main application thread. This ensures that time-consuming operations do not interfere with user-facing functionalities.

Adopt Asynchronous Processing:
With a dedicated background task manager in place, asynchronous task queues can be utilized to process jobs in parallel, reducing latency and maintaining system responsiveness.

Task Scheduling and Prioritization:
Deploy a scheduling mechanism to manage the prioritization of tasks. Critical tasks can be executed with higher priority while less urgent tasks are queued, optimizing overall resource utilization.

What components might need to be redesigned at scale?

The loosely decoupled architecture, though flexible, can lead to data consistency challenges across services. To address this, implementing mechanisms such as eventual consistency models, distributed transactions (like the Saga pattern), or automated conflict resolution strategies becomes crucial. Additionally, enhanced monitoring and observability through centralized logging and distributed tracing will support rapid identification and resolution of issues, ensuring the system remains robust and reliable as it scales.