Design Document for Notion Backend

Overview

This document outlines the design choices and architecture of the Notion Backend project. The backend system is built using FastAPI, SQLAlchemy, and Pydantic. It includes user authentication, lead management, and email notification functionalities. The choices made aim to ensure scalability, maintainability, and ease of development.

Table of Contents

Architecture

The architecture of the Notion Backend follows a standard three-layer design:

1. Presentation Layer: Manages incoming HTTP requests and returns HTTP responses.
2. Application Layer: Contains business logic and application rules.
3. Data Layer: Manages data persistence and retrieval using SQLAlchemy ORM.

Technology Stack

- FastAPI: A modern, fast (high-performance), web framework for building APIs with Python 3.6+ based on standard Python type hints.
- SQLAlchemy: An SQL toolkit and Object-Relational Mapping (ORM) library for Python.
- Pydantic: Data validation and settings management using Python type annotations.
- SQLite: A lightweight database used for development and testing. Can be replaced with any SQL database.
- Redis: An in-memory data structure store used for handling asynchronous tasks such as sending emails.

Design Decisions

1. FastAPI for the Web Framework

Why:

- FastAPI provides high performance, on par with Node.js and Go.
- Automatic interactive API documentation with Swagger UI and ReDoc.
- Asynchronous support for handling many simultaneous connections.

How:

- Used FastAPI to create the main application instance.
- Defined API endpoints using path operations decorators (e.g., @app.post, @app.get).

2. SQLAlchemy for ORM

Why:

- SQLAlchemy is a powerful and flexible ORM for Python.
- Supports multiple database backends, allowing easy migration from SQLite to other databases like PostgreSQL or MySQL.

How:

- Defined database models using SQLAlchemy's `declarative_base`.
- Created a database session using `sessionmaker` and managed connections with dependency injection in FastAPI.

3. Pydantic for Data Validation

Why:

- Pydantic ensures data is validated and parsed using Python type annotations.
- Simplifies the creation of data models for request validation and response serialization.

How:

- Define Pydantic models for request and response schemas.
- Used these models in FastAPI path operations to automatically validate incoming requests.

4. Environment Variables for Configuration

Why:

- Keeps sensitive information and configuration details out of the codebase.
- Facilitates easy changes in configuration without modifying the code.

How:

- Used Pydantic's BaseSettings to load configuration from environment variables.
- Created a .env file to store environment-specific variables and added it to .gitignore to prevent it from being committed.

5. Authentication using JWT

Why:

- JSON Web Tokens (JWT) are a compact, URL-safe means of representing claims to be transferred between two parties.
- Stateless nature of JWT allows for scalable authentication mechanisms.

How:

- Implemented JWT token creation and verification using jose library.
- Defined OAuth2 password flow using OAuth2PasswordBearer and integrated it into FastAPI dependency injection.

6. Email Notifications using Redis

Why:

- Redis provides an efficient way to handle asynchronous tasks.
- Allows for queuing and processing tasks like sending emails without blocking the main application flow.

How:

- Used Redis to queue email tasks.
- Implemented email sending functionality using smtplib for SMTP interactions.

Database Design

Models

Lead:

- id: Integer, Primary Key
- first_name: String
- last_name: String
- email: String
- resume: String
- state: Enum (PENDING, REACHED_OUT)

User:

- id: Integer, Primary Key
- username: String, Unique
- email: String, Unique
- hashed_password: String
- disabled: Boolean

Design Choice:

- Used SQLAlchemy enums for lead states to ensure valid states and simplify queries.
- Indexed email and username fields for faster lookups.

Authentication

Design Choice:

- Used JWT tokens for stateless authentication.
- Defined token expiry to enhance security.
- Implemented helper functions for password hashing and verification using passlib.

Email Notifications

Design Choice:

- Queued emails in Redis to offload the task from the main application thread.
- Configured SMTP settings using environment variables for flexibility.

Configuration Management

Design Choice:

- Loaded configuration from a `.env` file using Pydantic's `BaseSettings`.
- Ensured sensitive information is stored securely and not hardcoded in the codebase.

Error Handling

Design Choice:

- Used FastAPI's built-in exception handling to manage common errors.
- Defined custom HTTPException for specific error cases like invalid credentials.

Deployment Considerations

Design Choice:

- Configured the application to run using Uvicorn for asynchronous support.
- Ensured environment variables are managed securely in production environments.
- Planned for database migrations and backup strategies when moving from SQLite to other databases.

By following these design choices, the Notion Backend ensures a robust, scalable, and maintainable backend system capable of handling user authentication, lead management, and email notifications efficiently.