Design Document: Lead Management System using FastAPI

Overview

This document outlines the design decisions and rationale behind the architecture and implementation of the Lead Management System FastAPI application for managing leads and attorneys. I have tried to follow best practices for building secure, scalable, and maintainable web applications.

Requirements

The key requirements driving the design are:

1. **Lead Management**: Leads submit their information (name, email, resume) through a public form. Notify attorney(s) when a new lead is created.
2. **Attorney Access**: Internal UI (API) for attorneys to view and manage leads. Attorneys should be able to update the status of a lead after reaching out.
3. **Authentication**: Implement secure authentication for attorneys to access the internal UI (API).
4. **Data Persistence**: Store and retrieve lead data, attorney data, and lead statuses from a database.

Architecture Overview

The application follows a microservices architecture built with FastAPI. It consists of the following components:

1. **API Gateway**: The main entry point that handles incoming requests, routing, authentication, and response handling.
2. **Lead Service**: Responsible for managing leads, including creating new leads and retrieving lead data.
3. **User Service**: Handles user (attorney) authentication, authorization, and management of attorney accounts.
4. **Database**: A SQLite3 database for persisting lead data, user data, and lead statuses.

Design Choices

API Design

The application exposes a RESTful API following the OpenAPI specification. The API endpoints are designed around resources (leads, users, lead_status, attorney availability) and support standard HTTP methods (GET, POST, PUT) for performing CRUD operations.

The OpenAPI specification is used to generate interactive documentation (Swagger UI) and client libraries, promoting API discoverability and ease of integration.

Authentication and Authorization

Authentication and authorization are implemented using FastAPI's JSON Web Tokens (JWT) and the OAuth2 protocol. The application uses the `python-jose` library for JWT handling.

Attorneys authenticate by providing their email and password, which are verified against the database. Upon successful authentication, a JWT access token is issued with a configurable expiration time (30 mins default).

The internal API endpoints are protected and require a valid JWT in the `Authorization` header for access. Role-based access control (RBAC) can be implemented to restrict certain operations (e.g., updating lead status) to attorneys only. Other roles can be added subsequently.

Database Design

The application uses SQLite3 as the database engine for simplicity and portability during development and testing. The database schema consists of the following tables:

1. **`users`**: Stores attorney information (email, hashed password, role).
2. **`leads`**: Stores lead data (name, email, resume file).
3. **`lead_statuses`**: Tracks the status of each lead (**`PENDING`**, **`REACHED_OUT`**) and the associated attorney.

SQLAlchemy is used as the Python SQL toolkit and Object-Relational Mapping (ORM) library for interacting with the database.

Email Notifications

When a new lead is created, the application sends email notifications to the prospect (acknowledgment) and an available attorney (new lead notification). The smtplib module is used for sending emails, and email configurations (SMTP server, credentials) are loaded from environment variables.

Here the user will need to specify their credentials to send email.

Separation of Concerns

The application follows the principle of separation of concerns by dividing the codebase into separate modules:

1. `` `**main.py**` ``: Contains the FastAPI application instance and route definitions.
2. `` `**crud.py**` ``: Implements the Create, Read, Update, and Delete (CRUD) operations for interacting with the database.
3. `` `**models.py**` ``: Defines the SQLAlchemy models representing the database tables.
4. `` `**schemas.py**` ``: Defines the Pydantic models for request/response validation and serialization.
5. `` `**utils.py**` ``: Contains utility functions, such as sending emails.

This separation promotes code reusability, testability, and maintainability.

Scalability and Performance

While the current implementation uses SQLite for simplicity, it can be switched out for any other database. By leveraging SQLAlchemy, the database can be swapped with a more scalable solution (e.g., PostgreSQL, MySQL) for production deployments.

Caching mechanisms can be implemented to improve performance for frequently accessed data, such as lead lists or attorney information.

The microservices architecture allows for horizontal scaling of individual services based on demand, promoting better resource utilization and fault isolation.

Security

The application follows security best practices, including:

1. Password hashing using bcrypt for secure storage of attorney passwords.
2. Protection against common web vulnerabilities (SQL injection, XSS, CSRF) through input validation and sanitization.
3. Use of environment variables for storing sensitive configurations (e.g., database credentials, SMTP credentials).
4. Secure handling of file uploads (resumes) to prevent arbitrary file writes.

Regular security audits and penetration testing should be performed to identify and mitigate potential vulnerabilities.

Future Enhancements

While the current implementation fulfills the core requirements, the following enhancements can be considered for future iterations:

1. **Logging and Monitoring**: Implement centralized logging and monitoring solutions for better visibility into application health and performance.
2. **Testing**: Develop comprehensive unit and integration tests to ensure code quality and catch regressions early.
3. **Containerization**: Package the application and its dependencies into Docker containers for consistent deployments across environments.
4. **CI/CD Pipeline**: Set up a Continuous Integration and Continuous Deployment (CI/CD) pipeline for automated builds, testing, and deployments.
5. **Microservices** Communication: Implement inter-service communication patterns (e.g., message queues, event-driven architecture) for better decoupling and scalability.
6. **Caching**: Implement caching mechanisms (e.g., Redis) for frequently accessed data to improve performance.
7. **Database Migration**: Implement database migration tools (e.g., Alembic) for managing schema changes and data migrations.

This design document outlines the key architectural decisions and rationale behind the FastAPI application Lead Management System Version 1.0.0.