SUBJECT: **Service Design (IoT23)**

PROJECT BY: **Nnaka Joyce**

TOPIC: **Service Design Case**

**Customer Management and Weather Data Integration**

This project involves the development of a Flask-based API that allows users to manage customer data and retrieve real-time weather information. The primary objective is to create a secure and easy-to-use system that can integrate external APIs (OpenWeatherMap) and perform CRUD operations on customer data. The project focuses on addressing the needs of small business owners and data analysts who require reliable service to automate their data management tasks and integrate external weather data into their decision-making processes.

**Goals and Intended Functionality:**

* **Customer Management**: Provide an API to add, retrieve, and manage customer data in a structured manner.
* **Weather Data Integration**: Allow users to fetch real-time weather data from an external service (OpenWeatherMap).
* **Security**: Implement API key management using environment variables to protect sensitive information.
* **Documentation**: Offer clear and accessible API documentation via an endpoint for developers.
* **Scalability**: Build the foundation for a scalable system with easy expandability in the future.

**Results**

The project successfully met its intended goals by delivering an API that:

* Allows the creation and retrieval of customer data via well-defined RESTful endpoints.
* Integrates with the OpenWeatherMap API to fetch weather data based on city names.
* Implements secure API key handling using a .env file to protect sensitive credentials.
* Provides detailed API documentation accessible at an endpoint for users to quickly understand and use the API.

Users, including both business owners and data analysts, can easily interact with the API through tools like Postman or directly via code, making the service highly accessible and practical for various needs.

**System Description**

**Architecture Overview:**

This project uses a **client-server architecture** where the Flask API acts as the server handling requests and returning responses to the client. The architecture consists of the following components:

* **Flask Web Framework**:
* Acts as the backbone for routing and handling HTTP requests.
* Provides endpoints for customer CRUD operations and external service integration.
* **SQLite Database**:
* A lightweight database is used to store customer data.
* Basic operations such as creating, reading, and adding customers are supported.
* **External Service (OpenWeatherMap)**:
* Integrated to fetch real-time weather data based on city input.
* Data is fetched via API calls and returned in JSON format.
* **API Documentation**:
* YAML-based documentation served at an accessible endpoint.
* Tools like Swagger can be used to enhance this further.

**System Flow:**

* Users interact with the API via tools like Postman or curl.
* The API receives requests, processes them using the SQLite database and external services, and returns responses in JSON format.

**Description of Service Design**

Service Design methods were used to ensure that the API meets user needs efficiently. Key methodologies include **Personas** and a **Customer Journey Map** to visualize user interactions and align the service with user expectations.

**Personas:**

* Two key personas were identified: **John (Business Owner)** and **Sarah (Data Analyst)**.
* John requires a simple interface for adding and retrieving customer data.
* Sarah is more technical and requires structured data for further analysis.

**Customer Journey Map:**

* A Customer Journey Map was created to visualize how users like John and Sarah interact with the API.
* John’s journey highlights the importance of quick-start guides and reliable API performance.
* Sarah’s journey emphasizes the need for data consistency and integration with external services.

**Final Functionality and Service Design Integration**

The final functionality aligns with the Service Design in several ways:

* **Ease of Use**: The API’s user-friendly documentation ensures that even non-technical users like John can easily perform operations such as adding or retrieving customer data.
* **Reliability and Security**: The use of environment variables (.env) ensures that sensitive API keys are not exposed, which is essential for both business users and analysts.
* **Real-Time Data**: Weather data integration allows Sarah to correlate external environmental factors with customer behavior, making the service a valuable tool for data-driven decision-making.

The clear documentation, easy-to-use endpoints, and seamless integration with external services justify how the system works efficiently for different user needs.

**Discussion**

**Summary:**

The Flask API project successfully delivered a scalable, user-friendly service for customer management and weather data integration. By applying service design principles, the project ensures that the final product aligns with the needs of the end-users, offering them both ease of use and flexibility.

**Future Improvements:**

* **Pagination**: Implement pagination for large customer data sets to optimize performance and API response times.
* **OAuth Security**: Introduce OAuth for improved security, ensuring only authenticated users can access certain endpoints.
* **Rate Limiting**: Implement rate-limiting to prevent abuse of the weather data API.
* **Monitoring and Analytics**: Add more detailed logging and analytics to track API usage and performance for further optimizations.

**Appendices**

**API Documentation**

The project includes YAML-based documentation for all the available endpoints, ensuring users can understand the API’s functionality quickly.

**YAML Example**:



**8. References**

* Flask Documentation: <https://flask.palletsprojects.com/en/2.0.x/>
* SQLite Documentation: <https://sqlite.org/docs.html>
* OpenWeatherMap API: <https://openweathermap.org/api>
* Swagger: <https://swagger.io/tools/swagger-ui/>