Research Back-End technologies

Research and evaluate different back-end technologies that can be used to support the staff access monitoring system. This includes selecting the appropriate server framework, database system and any other tools taken into consideration.

The Back-End is crucial for the correct development of the staff access monitoring system. As shown in the requirements the back end must be responsible for the data processing and implementing the logic of the system. It also needs to centralize the data management and provide a secure environment for the communication between the ESP32 and the database.

As a general overview of the back-end technologies implemented in the project, we will describe the process and then proceed to describe each of the tools required. The ESP32 will communicate with the **EMQX** broker using **MQTT**, which will handle message routing and security. Then **Node.js** will act as the server-side logic, processing incoming data and interfacing with **MySQL** to store the data. MySQL will be the long-term storage solution for all the data generated by the system. Each component works together in order to contribute to the accurate implementation of the project and provide a robust, scalable, and efficient system for recording entries and exits in the Computer Lab.

1. MQTT (Message Queuing Telemetry Transport)

MQTT is the communication protocol that will be used by the ESP32 to send and receive messages to and from the server.

MQTT (Message Queuing Telemetry Transport) is a lightweight, publish-subscribe network protocol that transports messages between devices. We consider MQTT is the framework we need for the project as it excels in the following features that are crucial for the access control system we intend to develop:

a. Efficiency and Lightweight
   - Low Overhead: MQTT is designed to be efficient in terms of both message size and protocol overhead. The header size is only 2 bytes, which makes it suitable for environments with limited bandwidth.
   - Binary Protocol: Unlike text-based protocols like HTTP, MQTT uses a binary format, reducing the size of the messages and the processing required to interpret them.
   - Persistent Sessions: MQTT allows clients to maintain a session, so when they reconnect, they can continue where they left off without needing to resend all data, further reducing bandwidth usage.

b. Device Communication
   - Publish/Subscribe Model: MQTT uses a publish-subscribe pattern instead of the traditional client-server model. Devices (clients) can publish messages to topics and subscribe to topics they are interested in. This decouples the sender and receiver, making communication more flexible and scalable.

- Asynchronous Communication: Devices can send and receive messages asynchronously, meaning they don't need to be connected simultaneously, which is crucial for devices with intermittent connectivity.
- Scalability: The protocol's lightweight nature and publish-subscribe model allow thousands of devices to communicate with a central broker, scaling efficiently with a large number of devices.

c. Quality of Service (QoS): MQTT offers three levels of Quality of Service, allowing developers to choose the appropriate level of assurance for message delivery:
   - QoS 0 - At most once: The message is delivered once with no acknowledgment required. It's the fastest and most lightweight but offers no guarantee of delivery.
   - QoS 1 - At least once: The message is delivered at least once, ensuring delivery by waiting for an acknowledgment. If the acknowledgment is not received, the message is resent.
   - QoS 2 - Exactly once: The message is guaranteed to be delivered exactly once. This level requires the most overhead and ensures that no duplicate messages are received.

d. Security:
   - Username and Password Authentication: MQTT supports username and password authentication to control access to the broker.
   - Access Control: MQTT brokers can implement fine-grained access control, allowing or denying access to specific topics based on client credentials.

<u>Responsibilities in the project:</u>

- Publish-Subscribe Communication: The ESP32 will publish messages (like biometric data) to specific topics, and the server (or other devices) can subscribe to these topics to receive the data.
- Efficient Data Transfer: MQTT ensures that the data is transmitted efficiently, even over low-bandwidth or unreliable networks.
- Quality of Service (QoS): MQTT will manage the delivery of messages based on the QoS level you choose, ensuring reliability in message delivery.

2. EMQX (Erlang MQTT Broker)

EMQX is the MQTT broker that will manage all the MQTT communications between the ESP32 and the server.

A broker is a key component in message-based communication systems, particularly in protocols like MQTT. It acts as an intermediary between devices (clients) that want to exchange messages. It runs by a simple logic:

- Message Routing: The broker receives messages from publishing clients and then distributes them to the appropriate subscribing clients. This allows devices to communicate without needing to know each other's addresses.
- Publish-Subscribe Model: In MQTT, devices don't send messages directly to each other. Instead, they publish messages to a topic on the broker, and any devices subscribed to that topic receive the message. This decouples the sender and receiver, making the system more scalable and flexible.

- Session Management: The broker manages sessions for all connected clients, keeping track of their subscriptions and handling message delivery even when clients temporarily disconnect.

Responsibilities in the project:

- Message Routing: EMQX will route messages published by the ESP32 to the appropriate subscribers (Node.js server).
- Security: EMQX can enforce security policies, such as authenticating clients and ensuring that only authorized devices can publish or subscribe to certain topics.
- Scalability: It will support the scaling of your system, handling multiple devices that can be added in the future with high performance.

3. Node.js

Node.js is the runtime environment that will be used to build the back-end server, which will process the data sent by the ESP32 and interact with the database.

Node.js is a runtime environment that allows you to run JavaScript code outside of a web browser. It's built on Chrome's V8 JavaScript engine and is used to build scalable, high-performance server-side applications.

Responsibilities in the project:

- MQTT Client: Node.js will act as an MQTT client, subscribing to topics on the EMQX broker to receive data from the ESP32.
- Data Processing: It will process the incoming data, such as validating the data, converting it if necessary, and preparing it for storage in the database.
- API Development: You can create RESTful APIs or WebSocket endpoints using Node.js to allow web or mobile applications to interact with your system.
- Database Interaction: Node.js will handle the connection to the MySQL database, executing queries to store, retrieve, or update data.

4. MySQL

MySQL is the relational database management system that will store the data collected from the ESP32. This will allow us to have all the information registered in one centralized place, providing an organized environment.

Responsibilities in the project:

- Data Storage: MySQL will store the biometric data, timestamps, and any other relevant information sent by the ESP32.
- Data Retrieval: It will allow the back-end (Node.js) to query and retrieve stored data for analysis, reporting, or further processing.
- Scalability: MySQL can handle a large volume of data and can be scaled to accommodate growing data storage needs.