Exalens Coding Challenge Documentation

This is a project for the Recruitment to the Exalens.
Courtesy: Question credits belong to recruiting team @Exalens
Purpose: Simulate the behaviour of sensors, monitor their readings, and provide APIs to retrieve data based on specific criteria.

Table of Contents

Introduction

This project is required to prove to the recruitment team of @Exalens regarding the efficacy of knowledge and the development of this Full Stack Project.

This project serves the purpose of mocking the behavior of different sensors in a typical IoT environment and the devices, with monitoring and logging the climate.Let us consider an apartment like the one below.

Source: Home Designing

Typically we have a the basic structure of a small apartment with

One bedroom
One washroom
A kitchen & dining area
A living space
Here we are targetting that we have a temperature and a corresponding humidity sensor measureing the respective climate in each of the living spaces in the apartment, a climate control system in short.

In IoT (Internet of Things) terms, "climate control" refers to the monitoring and management of environmental conditions, particularly temperature, humidity, and sometimes other factors like air quality, within a specific space or area. Climate control systems are used to create and maintain specific environmental conditions for various

purposes, such as comfort, safety, energy efficiency, and the preservation of sensitive equipment or products.

Key aspects of climate control in IoT include:

Temperature Control: IoT devices and sensors are used to measure the temperature within a controlled environment. These sensors can transmit real-time temperature data to a central system or cloud-based platform for monitoring and control. Based on this data, climate control systems can adjust heating, cooling, or ventilation to maintain a desired temperature range.

Humidity Control: In addition to temperature, humidity levels are crucial for certain applications. IoT sensors can monitor humidity levels, and climate control systems can adjust humidity by controlling humidifiers or dehumidifiers as needed.

Energy Efficiency: IoT-based climate control systems can optimize energy usage by adjusting heating, cooling, and ventilation systems based on real-time data. This helps reduce energy consumption and costs while maintaining comfort or environmental requirements.

Remote Monitoring and Control: IoT enables remote monitoring and control of climate control systems. Users can access and control the system through web interfaces or mobile apps, allowing them to make adjustments and receive alerts or notifications when conditions deviate from desired levels.

Data Analysis: IoT platforms can collect and analyze historical climate data to identify patterns, trends, and potential issues. Machine learning algorithms can be applied to predict future environmental conditions and optimize control strategies.

Applications: Climate control in IoT has various applications, including:

Smart Homes: Thermostats and HVAC systems that can be controlled remotely for energy savings and comfort.
Industrial Environments: Maintaining specific environmental conditions in manufacturing facilities, storage areas, or cleanrooms.
Agriculture: Controlling climate conditions in greenhouses to optimize plant growth.
Healthcare: Monitoring and controlling temperature and humidity in healthcare facilities, laboratories, and pharmaceutical storage areas.
Data Centers: Ensuring temperature and humidity levels are within acceptable ranges to prevent equipment overheating.
Overall, climate control in IoT involves using sensors, data collection, and automated control systems to create and maintain specific environmental conditions, providing benefits such as comfort, energy savings, and environmental stability across various domains and industries.

With these points established we setout to develop on this project.

Coding Challenge questionnaire
Complete problem statement for the Coding Challenge as in the Mail

Challenge Overview
Purpose
Simulate the behaviour of sensors, monitor their readings, and provide APIs to retrieve data based on specific criteria.

- MQTT Broker Setup: Deploy a Mosquitto MQTT broker using Docker.
- MQTT Publisher: Create a Python MQTT client to mimic multiple sensor readings, publishing to topics like sensors/temperature and sensors/humidity. Structure of the JSON payload
    - { "sensor_id": "unique_sensor_id", "value": "<reading_value>", "timestamp": "ISO8601_formatted_date_time" }:

- MQTT Subscriber: Construct a Python MQTT subscriber to store the received messages in a MongoDB collection.
- Data Storage: Initiate a MongoDB instance using Docker and save the incoming MQTT messages.
- In-Memory Data Management: Implement Redis using Docker to store the latest ten sensor readings.
- FastAPI Endpoint: Design an API with the following endpoints:
    - An endpoint that allows users to fetch sensor readings by specifying a start and end range.
    - An endpoint to retrieve the last ten sensor readings for a specific sensor.
- Docker Integration: Integrate all services using Docker Compose.

Deliverables

Repository: Host the entire codebase on GitHub.

Docker Compose: Include a docker-compose.yml file that ensures easy system setup. This file should encompass services for the Python apps (MQTT publisher, subscriber, FastAPI application), Mosquitto, MongoDB, and Redis.

File: Your repository should feature a comprehensive file detailing the following:

- Instructions for setting up and interacting with the system using the docker-compose command.
- A detailed overview of each service is in the docker-compose.yml file.
- Insight into the design choices you made and the rationale behind them.
- A section discussing challenges encountered during the project's development and the solutions you implemented.

Guidelines

- Ensure your code is well-commented and adheres to industry standards.
- If you happen to encounter any ambiguities, please make informed assumptions and document them.

Timeline & Submission

While we understand the complexity and depth of the challenge, three weeks should be sufficient. So, we'd like to ask you to submit your solution by September 18th. Please share the GitHub link of your repository once you've completed the task.

Support
If you have any technical questions or require clarification on the challenge, please make decisions based on your understanding.

Setup
To account for the requirements of this project (check: Coding Problem) all the functionalities of this project are collected into a single docker-compose file which can be used to start-up or shut-down the systems with single commands.

Prerequisites
To run this project to its capabilities the following systems need to be installed first before starting up.

- Docker (check installation commands for Docker here)
- Docker Compose(check installation instructions for Docker-Compose here)
- Any browser of choice

Startup
Once the Docker and Docker-compose is installed and ensured. Clone the Repository at github.com/WolfDev8675/Exalens-CC. This will download a zip file of the Project to your download location.

After the download is done follow the following steps

1. Unzip/Extract the file contents.
2. Open a terminal.
3. From the terminal navigate to the location of the extracted folder.

4. Here execute the command ls -l to list the folder structure. You would see something like this

```
ls -l

total 48
-rw-r--r-- 1 kali kali 2640 Sep 15 08:34 coding_problem.md
-rw-r--r-- 1 kali kali 1674 Sep 16 10:58 docker-compose.yaml
drwxr-xr-x 3 kali kali 4096 Sep 17 01:31 docs
-rw-r--r-- 1 kali kali 1070 Aug 27 13:44 LICENSE
drwxr-xr-x 3 kali kali 4096 Sep 15 12:03 mongodb
drwxr-xr-x 2 kali kali 4096 Aug 29 01:25 mosquitto
drwxr-xr-x 3 kali kali 4096 Aug 30 14:10 publisher
-rw-r--r-- 1 kali kali  594 Sep 16 23:21 README.md
drwxr-xr-x 3 kali kali 4096 Sep 13 03:42 redis
drwxr-xr-x 2 root root 4096 Sep 17 02:29 redis.conf
drwxr-xr-x 3 kali kali 4096 Sep  9 10:59 subscriber
drwxr-xr-x 3 kali kali 4096 Sep  9 10:59 webapp
```

5. If you are starting up for the first time run the command

   docker-compose up -d --build

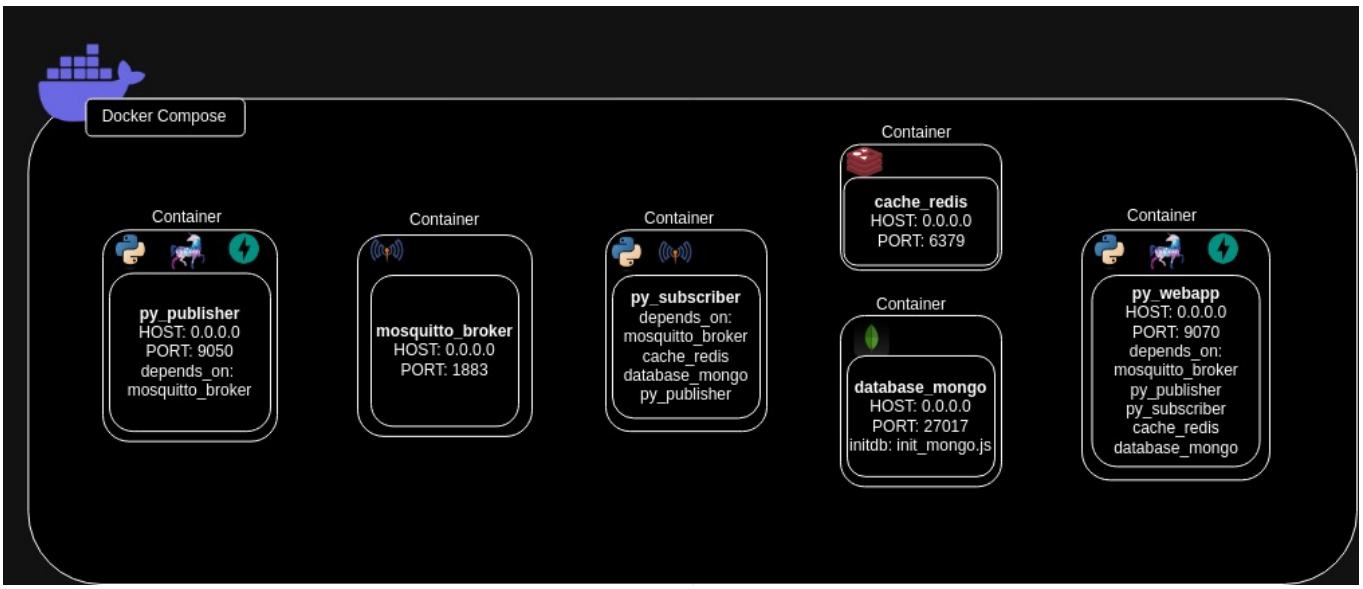
   otherwise a simple docker-compose up -d will work. This will startup the whole system and the suite of services implemented in the docker compose.
6. For confirming the systems are running execute the **docker ps** command in the terminal which will reveal a list of services similar to this.

7. To check the data see the API Documentation open a browser and go to the URL: http://localhost:9070/docs
8. To shutdown the services execute the command **docker-compose down**

Architecture & Design

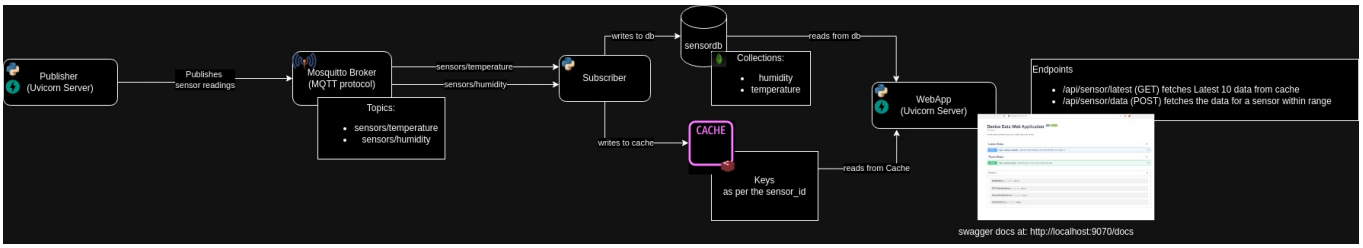
The Project required us to set up various functionalities (check here) in a single docker-compose.yaml file to start all the various systems.

To account for the some design liberties as well as sticking to best practices for a RESTful suite of services the Docker-Compose is structured in a way as shown below



Docker-Compose structure

and the services are created according to the following design.



Service Design

Each component(container) of the Docker-Compose is described as

Mosquitto Broker

- name: mosquitto_broker
- host: 0.0.0.0
- port: 1883

sets up the eclipse mosquitto broker following the MQTT protocols. It depends on a configuration file which defines the bare minimum MQTT broker to be up. For data to be passed around.

Publisher

- name: py_publisher
- host: 0.0.0.0
- port: 9050
- depends on: mosquitto_broker
- language: Python 3.11.4

runs in an ASGI Server Appplication(UVICORN) to send data of the various sensors(total: 8) to the broker at a timed interval of 5 seconds. The service is scheduled to send out a publish message to the broker for all the Temperature sensors together and the Humidity sensors together. This service requires Mosquitto Broker service to be up and running before itself starting up. This could be down otherwise also but thinking in terms of best-practices for coding service was made to run in a server with scheduler rather than a infinite keep-alive loop which can introduce race-round and heap-memory failure.

MongoDB Server

- name: database_mongo
- host: 0.0.0.0
- port: 27017
- initdb: init_mongo.js

starts a mongodb container for a MongoDB Community Server with fixed admin and passwords along with an initialization database and collections for users who are setting up the service for the first time to have a smooth experience. This also potentially removes application crashes in the startup stages.

Redis

- name: cache_redis
- host: 0.0.0.0
- port: 6379

starts up a redis cache with pre-configured details. This is required by the challenge to implement a cache memory for data to be written into available instantly in a short interval without downtime.

Subscriber

- name: py_subscriber
- listens to: monsquitto_broker
- depends on: database_mongo, cache_redis and mosquitto_broker
- language: Python 3.11.4

subscription service with the sole purpose to keep listening to the subscription channels in the broker(Mosquitto Broker), process the data received and send them to be stored in the Cache(Redis) and Database(MongoDB Server).

WebApp(Device Data Web Application)

- name: py_webapp
- host: 0.0.0.0
- port: 9070
- depends on: all other services to be up before starting
- language: Python 3.11.4

an ASGI Web Server(UVICORN) API Application that provides the APIs for interacting with the data stored in the Cache and Database. Please check API Documentation for more information on the API endpoints available and querying them.

All these services coordinate together to bring up the service design to fruition.

Flow of Data

This system is designed to efficiently manage the flow of data from a Publisher to a Subscriber, with data being processed, stored, and made accessible through a web application (API). The flow of data is orchestrated through several interconnected services and components, ensuring data integrity, reliability, and accessibility.

1. Publisher:

The Publisher is the initial source of data in the system. It generates data and pushes it into the system. It communicates with the Broker to publish data to specific topics.

2. Broker:

The Broker acts as an intermediary, receiving data from the Publisher and forwarding it to Subscribers. It is responsible for topic-based message routing and delivery. The Broker maintains a list of active Subscribers for each topic.

3. Subscriber:

Subscribers subscribe to specific topics of interest within the Broker. When data is published to a subscribed topic, the Subscriber receives and processes the data.

In this system, the Subscriber performs two key tasks:

- It stores the received data in a cache for quick access and retrieval.
- It also stores the data in a database for long-term storage and analysis.

4. Cache:

The cache acts as a high-speed, in-memory storage system for recently received data. It enables rapid data retrieval for real-time or frequently accessed information. The cache enhances system performance by reducing the need to access the database for every data request.

5. Database:

The database serves as the long-term data storage solution. It stores historical and persistent data that can be queried and analyzed over time. Data in the database is structured, indexed, and optimized for efficient retrieval.

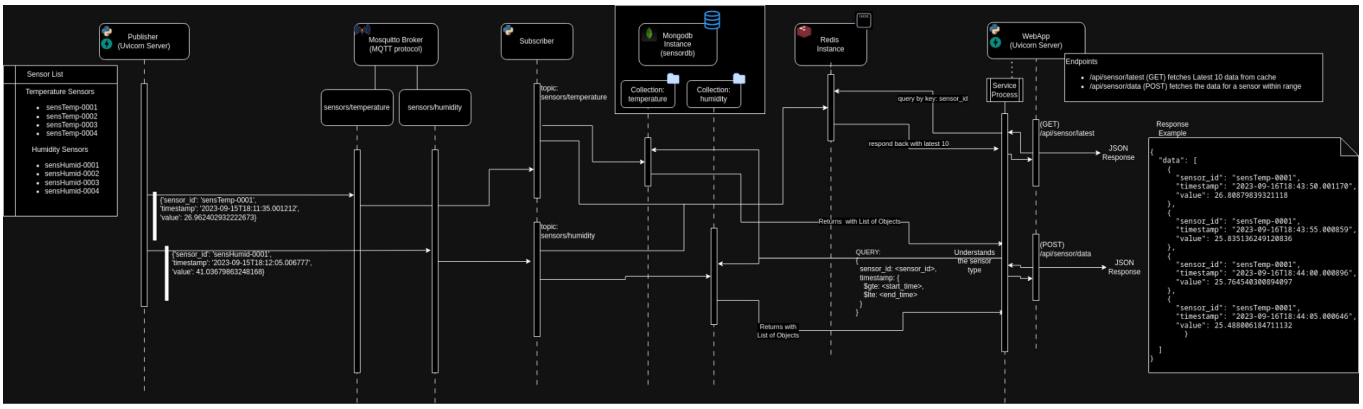
6. Web Application/API:

The web application provides an interface for users to access and query the data stored in the system. It communicates with both the cache and the database to retrieve data based on user requests. Users can access real-time and historical data through the API, making it a central point for interacting with the system.

Data Flow Summary:

Data originates from the Publisher and is channeled through the Broker. Subscribers receive, process, and store data in both the cache and the database. The cache serves as a performance-enhancing layer, while the database stores long-term data. The web application/API connects to both the cache and the database, providing users with seamless access to data.

This orchestrated data flow ensures that data is efficiently collected, stored, and made available for consumption, catering to real-time and historical data needs within the system.



Flow of Data

API Documentation

Device Data Web Application API documentation. This API provides access to sensor data, allowing you to retrieve the latest data entries or data within a specified time range.

Get Latest Sensor Data

Endpoint

GET */api/sensor/latest*

Description

This endpoint allows you to retrieve the latest 10 data entries for a specific sensor.

Parameters

- sensor_id (query parameter) - The unique identifier of the sensor for which you want to retrieve data.

Example

Request:

GET /api/sensor/latest?sensor_id=sensHumid-0001

Response:

```
{
 "data":[
        {
        "timestamp": "2023-09-16T14:57:25.0",
        "sensor_id": "sensHumid-0001",
        "value": 43.5
        },
        // ... (9 more entries)
        ]
}
```


Notes

- The response contains an array of the latest 10 data entries.

Get Sensor Data Within a Time Range

Endpoint

POST */api/sensor/data*

Description

This endpoint allows you to retrieve sensor data within a specified time range.

Request Body


```
{
 "sensor_id": "string",
 "start_time": "string",
 "end_time": "string"
}
```


- sensor_id (string, required) - The unique identifier of the sensor for which you want to retrieve data.
- start_time (string, required) - The start time of the time range in ISO8601 formatted datetime.
- end_time (string, required) - The end time of the time range in ISO8601 formatted datetime.

Example

Request:

POST /api/sensor/data

Request Body:


```
{
 "sensor_id": "sensHumid-0001",
 "start_time": "2023-09-16T14:48:20.0",
 "end_time": "2023-09-16T14:57:25.0"
}
```


Response:


```
{
 "data": [
        {
        "timestamp": "2023-09-16T14:48:20.0",
        "sensor_id": "sensHumid-0001",
        "value": 42.2
        },
        {
        "timestamp": "2023-09-16T14:49:15.0",
        "sensor_id": "sensHumid-0001",
        "value": 41.8
        },
        // ... (additional entries within the specified time range)
        ]
}
```


Notes

- The response contains an array of sensor data entries within the specified time range.

Challenges faced

While looking back at the progress and development of this project, we encountered a series of challenges and difficulties that tested our problem-solving skills and determination. This section provides an overview of the obstacles we confronted, along with the strategies and solutions we employed to overcome them. Our journey through these challenges has not only strengthened our capabilities but also enriched our project's development.

1. Redis & MongoDB Server's own docker file has a docker-entrypoint command.
    - Problem: This problem is quite definitive for building Dockerfile. Since the image pull exercises a docker-entrypoint command the dockerfile in which we are pulling the image couldn't be having a ENTRYPOINT command or any RUN command in such a case. The Image confuses itself which is the starting point and in the end fails to startup successfully with all the required commands.
    - Solution: Run the MongoDB Server and Redis directly in the Docker compose and not go for creating a separate docker file for customized purposes.
2. Mosquitto PAHO MQTT Client - in subscriber mode is always in auto persist condition.
    - Problem: When writing the Subscriber, we tried to follow the same scenario as the Publisher but it was eventually found that paho-MQTT client's loop-forever functionality keeps alive the communication with the broker and adding a server only conflicts with the priority of the code and adds complexity which otherwise is totally avoidable.
    - Solution: Branched code into services but kept the starting point of the function to be handled solely by the paho client.
3. Redis Key:List takes strings and MongoDB documents are fully functional JSON objects.
    - Problem: When pushing data into the Redis we push everything as String data but MongoDB formats everything to objects when passed as a

dictionary from python. Hence MongoDB preserves the data-type as well as data but for Redis it doesn't preserve the data-type.

- Solution: Customized utility function to recognize in each of the Python entities (Publisher, Subscriber, WebApp) how the data is fed into the system and take relative actions accordingly to modify or view as per solution.

License

LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.