Time-series caching with Redis and structured data via PostgreSQL

Abhilasha Katkiya[1,†]

[1] *OTH Amberg-Weiden*

Report written on the January 20th, 2025

Abstract

Our designed Smart Home System securely generates real-time sensor data such as Fire/CO detection, Leak/moisture detection, Window & door open and close, Smart thermostat, Motion sensors, and a Smart garage door. These transmit data via MQTT and Redis, and stored for immediate retrieval by a subscriber. We have successfully integrated Redis, PostgreSQL using a backend designed by using FastAPI to simulate and verify these sensor data. The structured data, including users, houses, and appliances are managed using PostgreSQL. The system achieves secure communication with encryption, API authentication, and a scalable architecture, ensuring robust performance for future expansion.

Keywords: *redis, postgresql, mqtt, sensor data, fastAPI*

1. Introduction

In our current modern society, there are many tasks that are increasingly becoming redundant that there is a need for automation. With increase in technologies used by our society, our homes are becoming smarter as well, along with our phones, appliances and ourselves!

This rapid growth has transformed how we interact with our environment which includes our offices, homes and other areas. Therefore, making our living and working spaces more efficient, secure and personalized has become important. With increasing number of connected appliances, there is a growing need for systems that can seamlessly manage, process, and visualize data in real-time. Such systems not only improve the comfort and convenience of daily life but also enable proactive monitoring and decision-making, ensuring that resources like energy and water are used efficiently.

This project addresses these challenges by building a system that integrates the growing advanced technologies to simulate, store and visualize data while ensuring security and scalability. Using monitoring of smart devices, our system assists users to optimize their energy usage and expenditure thus helping them to adapt to their growing demands.

2. Design aspects

Our design is focused on enabling the user who has different appliances in his/her one or many houses which carry/send sensor data. In order to work with this, we have made sure of system's scalability, real-time data handling, and secure communication, ensuring it can effectively manage and visualize data in a dynamic smart home environment.

Our system employs an architecture that integrates key technologies like Redis (Remote Dictionary Server), PostgreSQL, and a backend which we have integrated using FastAPI; where each fulfills a specific role within the ecosystem.

For real-time data handling, our publisher-subscriber model serves as the backbone. The data for multiple appliances is generated by the publisher, which can be identified uniquely by the SensorID of that particular appliance, and this data is transmitted via MQTT (Message Queuing Telemetry Transport) topics and Redis streams after encryption. This ensures a reliable and scalable mechanism. In turn, the subscriber receives the aforementioned sensor data, decrypts and validates it, and then stores the latest readings in Redis for quick access. This design allows for seamless integration of additional sensors or appliances without impacting system performance.

For structured data, have used PostgreSQL to manage relationships between users, houses, and appliances. Relational tables such as Users, House, HouseUser and Appliance ensure efficient CRUD (Create, Read, Update, Delete) operations [3] and support dynamic querying for application needs.

Finally, the backend API is built using FastAPI framework, which enables secure interaction with the database, offering endpoints for managing user, house and appliance information while ensuring data integrity through JSON Web Tokens (JWT) [4] authentication. JWT are an open, industry standard RFC 7519 method for representing claims securely between two parties, for example, between client and server. This security is provided in a compact, self-contained format and is commonly used for stateless authentication (that is, without storing any storage of a user's session or authentication state on the server), allowing users to authenticate and access protected resources.

Providing security is our fundamental design objective. To provide this confidentiality, Fernet was used from Cryptography library. It was used to encrypt sensor data before it was published to Redis and MQTT topics. This ensures that the sensor data remains secure during transit and storage. Similarly, Fernet was used to decrypt the received encrypted sensor data from MQTT or Redis. This allows the subscriber to process and use the original data. Thus, Fernet provides an encryption which ensures that sensitive sensor data cannot be read by unauthorized parties during transit or while stored in Redis/MQTT.

3. Use case analysis

Our Smart Home System makes use of relational database tables within PostgreSQL (Users, House, HouseUser, Appliance) and their relationships to manage structured data efficiently. Each of these tables serves a specific role, while joins between them enable complex queries [5] to extract meaningful data. The following use cases demonstrate how the system utilizes the backend to query the data from these tables and joins to address real-world scenarios in a smart home:

User and Role Management: The system is designed so as to allow administrators (admins) to manage users (e.g., add, update, delete) and assign them specific roles such as "Owner" or "Admin". This role-based access control ensures users can only access data and features relevant to their permissions. For example, an admin adds a new user and assigns them as the owner of a specific house.

Multi-House Management: Any user who is associated with multiple houses will be able to view and manage data for their each house independently. If a user does not have multiple houses, for obvious reasons, he/she would only be able to view and manage their single property. For example, a user monitoring energy consumption from both their primary residence, and their vacation home.

Appliance Tracking and Management: It is possible for users to keep track of their appliances in their respective houses, and also be able to manage them. Additionally, the system can store the details of the appliances which are associated with the corresponding houses, and also retrieve this information (using API routes). For example, a

user can get a list of the appliances in one of their houses, and their current statuses.

Real-Time Sensor Monitoring: As necessary, users can monitor the real-time sensor data (e.g., temperature, humidity, pressure) from their smart home appliances. The system retrieves this data from Redis, which is updated by subscribers processing MQTT-published data. As example would be where a user views the latest readings of temperature and humidity of their home.

4. Database concept

The Smart Home System effectively uses relational tables (User, House, HouseUser, Appliance) and their relationships to manage users, houses, and appliances. Each table represents a key entity in the system, and joins between these tables enable meaningful insights by connecting these entities. CRUD (Create, Read, Update, Delete) operations on these tables support robust management and monitoring functionality.

- *User* Table: Stores information about the individuals interacting with the system, such as homeowners, tenants, or admins.
- *House* Table: Represents properties with address and location details of each property of the users.
- *HouseUser* Table: Links users and house tables with additional attributes such as role and tenancy duration.
- *Appliance* Table: Manages appliances installed in houses.

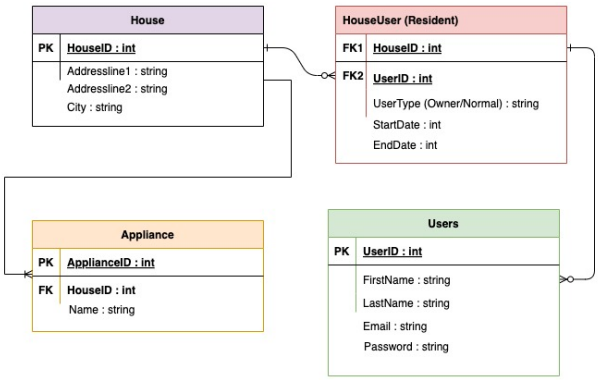

Figure 1. Entity relationship diagram

Each table mentioned above (see Figure [1]) has a unique primary-key, which identifies with the use of the table. That is, User table has a primary-key which represents a unique user. Similarly, we have such an identifier for house-id which identifies a house uniquely. For the table HouseUser, this connects the User and House tables such that user-id and house-id from User and House tables, respectively are foreign-keys in the HouseUser table. Additionally, the Appliance table also has its own primary-key which is the appliance-id and is connected to the House table with house-id being the foreign-key.

5. Implementation aspects

To get the implementation as seamless as possible, we have made use of the below technologies. In what follows, we list them explaining how each one was implemented and connected with other services:

5.1. Docker

Docker has been our main backbone in our implementation and we have used this in our project to containerize and manage various services and applications. This has ensured that our environments are consistent and portable.

Containers are similar to a virtual machine, which has their operating system and everything, except that unlike virtual machines they don't simulate the entire computer, but rather create a sand boxed environment that pretends to be a virtual machine.

Thus, each component/service is containerized into separate Docker containers, which allows them to run independently and yet communicate effectively within the same Docker network hub. The services we have used are backend implemented using FastAPI (to run the API server and handles requests), PostgreSQL (a relational database for structured data), Redis (a cache for real-time inventory which stores data in key-value pairs) [6], MQTT Broker (to handle sensor data publishing and subscribing), Publisher and Subscriber (to simulate and process sensor data) and Grafana (to visualizes real-time and historical data).

In addition to the advantage of running on the same network, Docker's volume management also provides consistent storage for PostgreSQL, ensuring that the data is not lost when the container restarts.

5.2. Redis

Redis and MQTT were implemented to enable real-time communication and data storage for sensor data processing. MQTT is a messaging protocol used for real-time communication between devices, making it ideal for situations where sensor data is published and subscribed to.

The MQTT broker which we have used is Eclipse Mosquitto. It acts as the central message broker to handle secure communication between publishers and subscribers, which is deployed as a Docker container. The Publisher simulates real-time sensor data, which is encrypted. Both publisher and subscriber use the same encryption-key. The data is sent to both the MQTT broker and Redis for further processing and storage. On the other hand, Subscriber subscribes to MQTT topics to receive real-time sensor data. It then decrypts the data, processes it, and saves the latest readings into Redis for visualization and further use.

Redis is an open-source, in-memory key-value data store known for its high performance, and low latency. It is often used for caching, real-time analytics, and pubisher/subscriber messaging systems. Together with MQTT, Rediscts as a buffer for real-time data flowing from MQTT publishers to subscribers.

Redis was deployed as a Docker container. Its service was then configured to expose its port for communication with other containers like the publisher, subscriber and backend (discussed later).

Additionally, we have made sure that the format of this data is such that it can be fed later into InfluxDB for further operations and at the same time due to memory limits and eviction policies of Redis, help in efficient buffering of real-time data, preventing over-flow and reducing write load on InfluxDB to make it scalable and for long-term storage. This we have implemented with a limit on the memory usage to 256MB with a policy to evict the least recently used keys when full. Each key in Redis expires after 1 hour (3600 seconds).

5.3. PostgreSQL

We have used PostgreSQL as our primary relational database. PostgreSQL is a powerful, open-source, an extensible relational database management system (RDBMS). We have made use of this as it is designed for handling structured data with a focus on reliability, robustness, and its compliance with ACID properties (Atomicity, Consistency, Isolation, Durability).

To implement this as part of our smart ecosystem, we used PostgreSQL to store structured data like user profiles, house information, appliances, and their relationships.

PostgreSQL implementation was first started off with its containerization using Docker. Here, we configured our environment variables like a POSTGRES_USER, a POSTGRES_PASSWORD, and POSTGRES_DB for database initialization and setting the database credentials.

Afterwards, database schema which was constructed using an ER-diagram, see Figure [1], was put into place by creating respective

tables and adding the necessary constraints and cascading. Data was later added to these tables using SQL insert statements. This was then integrated to our backend using FastAPI.

5.4. Backend

Using FastAPI framework [1], we have implemented and connected our backend to above listed services. This became our central layer that connected the various components like PostgreSQL, Redis, MQTT, and user authentication. Thus it served as the backbone of the project, handling API endpoints for CRUD operations, real-time data retrieval, and user authentication. *User authentication*: We implemented basic authentication using FastAPI's `HTTPBasic` module [2] to secure the API endpoints. This made sure that each user provides valid credentials (username and password) to access protected routes. This role-based access control ensures that different users (e.g., Admin, Owner) have access to specific functionalities. That is, Admins can perform CRUD operations on all entities, while the Owners can access their assigned houses and related data. Additionally, any Unauthorized access is blocked with an appropriate HTTP error code (401 Unauthorized, 403 Forbidden).

API design: We designed our backend such that it followed a modular architecture, separating concerns into different route files. That is, each file handled a specific functionality like user management, house management, or appliance management. In our case, each such management was put into a separate Python script, which was then routed into the main file with distinct prefixes, namely, `/users`, `/houses`, `/appliances`, for better organization in FastAPI's Swagger UI.

Services: The PostgreSQL integration with the backend was done using `psycopg2` library that handles database connections and executes SQL queries. With this, the FastAPI routes interact with PostgreSQL which enable the CRUD operations and different complex SQL queries. For example, the `/users` endpoint fetches user data by querying the Users table, while the `/houses` endpoint retrieves data about houses.

For Redis, the `redis-py` library was used which provided the necessary connection to the backend. Here, the sensor data is saved in key-value pairs, which result in a quick data retrieval. The FastAPI routes fetch this data to provide real-time update to users. In the case of MQTT, what FastAPI backend does is to access processed data via Redis and reveals it through the endpoints like `/houses/house_id/data`.

Additionally, we also integrated Grafana, a real-time visualization tool, that helps in creating dashboards for visualizing real-time sensor data and historical metrics. This was integrated with both Redis and PostgreSQL as data sources.

6. Testing

Apart from each CRUD operation which was tested to make sure a user, house and appliance can be created, updated and also deleted, we tested the following core sections that needed to be tested to make sure the principles of security and authentication, along with user, house and appliance management are properly tested:

6.1. Security

We tested to make sure the publisher and subscriber were encrypting and decrypting the sensor data in the correct manner. Below is the code [1] that generates a symmetric encryption key. This key is a random 256-bit value encoded in URL-safe Base64 format:

```
from cryptography.fernet import Fernet

# Generate encryption key
key = Fernet.generate_key()

# Save the key to a file
with open("encryption_key_user1.key", "wb") as
    key_file:
    key_file.write(key)
```

Code 1. Generating a symmetric encryption key.

Figure 2. Snapshot of Encrypted and Decrypted sensor data from Publisher and Subscriber, respectively.

From Figure [2], one can see some sample output lines of data that show the encrypted and decrypted sensor data. Here, the encrypted data is clearly cryptic and the decrypted output is shown in a meaningful JSON format.

6.2. User authentication

This part was tested to make sure a user (or admin) is able to login, be authorized and be able to create a user, update a user and be able to delete a user. In turn, a user is able to view only his/her houses and appliances that are available, thus ensuring data protection. See Figure [3] for reference:

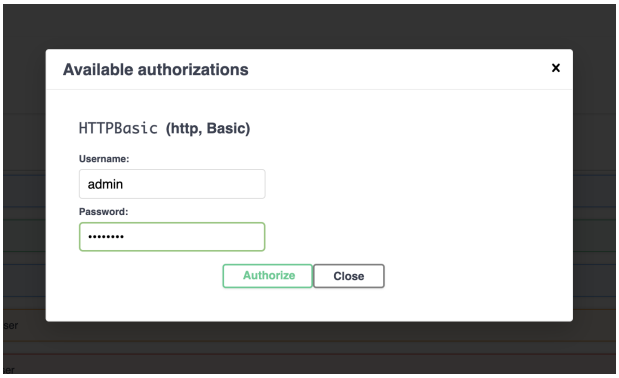

Figure 3. Authentication of an admin user.

6.3. Normalizing PostgreSQL tables

Here [2] we have tested the use case where we retrieve which users have multiple houses and to list those houses showing their addresses.

Result of a sample query for userid=1 is shown in the Figure [4].

```
@router.get("/")

def get_houses_for_user(user_id: int = Query
    (..., description="User ID to filter houses"
    )):
    try:
        conn = get_db_connection()
        cursor = conn.cursor(cursor_factory=
    DictCursor)
        cursor.execute("""
            SELECT
                h.HouseID,
                h.AddressLine1,
                h.City
            FROM
                House h
            JOIN
                HouseUser hu ON h.HouseID = hu.
    HouseID
            WHERE
                hu.UserID = %s;
        """, (user_id,))
```

```
        houses = cursor.fetchall()
        if not houses:
            raise HTTPException(status_code=404,
     detail="No houses found for the specified
    user")
        return [dict(house) for house in houses]
    except Exception as e:
        raise HTTPException(status_code=500,
    detail=f"Database error: {str(e)}")
```

Code 2. Querying for multiple houses belonging to a user.

Figure 4. Result of a sample complex query-1, listing houses belonging to a user.

In query [3], we are retrieving appliances that belong to a particular user. This makes sure that the appliance is selected and displayed for a user to who these houses belong

```
@router.get("/users/{user_id}/appliances")
def get_appliances(user_id: int):
    conn = get_db_connection()
    cursor = conn.cursor(cursor_factory=
    DictCursor)

    # Query to fetch appliances for the given
    user
    cursor.execute("""
        SELECT * FROM Appliance a
        JOIN House h ON a.houseid = h.HouseID
        JOIN HouseUser hu ON hu.HouseID = h.
    HouseID
        WHERE hu.UserID = %s;
    """, (user_id,))
    appliances = cursor.fetchall()
    conn.close()

    if not appliances:
        raise HTTPException(status_code=404,
    detail="No appliances found for the user")

    return [dict(appliance) for appliance in
    appliances]
```

Code 3. Querying for all appliances for a user from their respective houses.

Figure [5] shows the output received by applying the query [3] to a user with userid=1. We can see that all the appliances for this user are retrieved and we can see from the above json structure that some belong to the same house and some belong to a different house. For example, appliances with applianceid=1 and applianceid=9 belong to houses with houseids 1 and 5, respectively.

Figure 5. Result of a sample complex query-2, listing the appliances of a user in different houses.

7. Summary & Outlook

In our system, it is possible for a single user can have multiple houses, enabling seamless management of properties under a single account. Each house can include several appliances, creating a complete mapping of user-owned assets and their associated appliances. The appliances in each house are connected to sensors that generate real-time data. To ensure confidentiality and integrity of this sensor data, encryption is applied before transmission of the data to storage in Redis.

Following this, our next goal is to direct this sensor data into InfluxDB for further processing.

8. Appendix

Please visit the following GitHub page for further details: https://github.com/LoopyPanda/Modern-Databases-NoSQL-MDNE-

9. Acknowledgments

We sincerely thank Prof. Dr. Gerald Pirkl and Prod. Dr. -Ing. Michael Wiehl at OTH-Amberg-Weiden for the various discussions and for his suggestions and support throughout this project.

References

[1] S. Ramírez, *Fastapi*, 2023. [Online]. Available: https://fastapi.tiangolo.com.

[2] S. Ramírez, *Fastapi auth*, 2023. [Online]. Available: https://fastapi.tiangolo.com/advanced/security/http-basic-auth/.

[3] *Crud operations*. [Online]. Available: https://medium.com/@ahmedazier/mastering-crud-operations-a-comprehensive-guide-for-developers-d141f9cf9f50.

[4] *Json web tokens*. [Online]. Available: https://jwt.io/introduction.

[5] *Sql complex queries*. [Online]. Available: https://popsql.com/blog/complex-sql-queries.

[6] G. Pirkl, *Modern databases and nosql-json-redis-key-values-origin of thought [powerpoint slides]*, Amberg, 24. Oktober, 2024.