IoT Event Ingestion and Alerting System

Assignment Overview

This assignment evaluates your skills in backend development by building a system with two microservices for IoT data ingestion and alerting. It involves working with RESTful APIs, PostgreSQL, Redis, RabbitMQ, and Docker to simulate a real-world IoT backend system.

Objective

Create a system that ingests IoT events, processes them, and generates alerts based on predefined criteria.

Assignment Details

Microservices

1. **Ingestion Service:**
    - Accept real-time IoT event data via a RESTful API.
    - Validate the `device_id` as a valid MAC address.
    - Store events in **PostgreSQL** for persistence.
    - Use **Redis** to cache sensor details for faster validation.
    - Restrict payloads from sensors not in the database.
    - The sensor model should include the following fields:
        - `device_id` (MAC address): Used for validation.
        - `device_type` (e.g., radar, security_camera, access_controller): Defines the type of the sensor.
    - Provide the following endpoints:
        - `POST /events`: Accepts IoT events after validation and saves them.
        - `GET /events`: Retrieves stored events with optional filters for time range, event type, or device type.
2. **Alerting Service:**
    - Subscribe to event streams published by the Ingestion Service via **RabbitMQ**.
    - Process incoming events and generate alerts based on predefined criteria specific to the device type.
    - Store generated alerts in **PostgreSQL**.
    - Provide the following endpoint:
        - `GET /alerts`: Retrieves triggered alerts with optional filters.

Predefined Criteria for Alerts

1. **Unauthorized Access:**
   - Triggered when an access control event includes a `user_id` not in a predefined authorized list.
   - The authorized list should be stored in the database and cached in Redis for quick access.
2. **Speed Violation:**
   - Triggered when a radar event records a `speed_kmh` above 90 km/h.
3. **Intrusion Detection:**
   - Triggered when a motion detection event indicates movement in a restricted area after hours. The event must include a photo encoded as a base64 string in the `photo_base64` field, which should be stored along with the alert.

Simulated Data Examples

Provide the following example payloads to guide implementation, including examples that trigger alerts and those that do not:

Access Control Sensor Event

1. **Triggers Alert:**

```
{
    "device_id": "AA:BB:CC:DD:EE:FF",
    "timestamp": "2024-12-18T14:00:00Z",
    "event_type": "access_attempt",
    "user_id": "unauthorized_user"
}
```

2. **Does Not Trigger Alert:**

```
{
    "device_id": "AA:BB:CC:DD:EE:FF",
    "timestamp": "2024-12-18T14:05:00Z",
    "event_type": "access_attempt",
    "user_id": "authorized_user"
}
```

Radar Speed Event

1. **Triggers Alert:**

```
{
    "device_id": "11:22:33:44:55:66",
    "timestamp": "2024-12-18T14:05:00Z",
```

```
    "event_type": "speed_violation",
    "speed_kmh": 120,
    "location": "Zone A"
}
```

2. **Does Not Trigger Alert:**

```
{
    "device_id": "11:22:33:44:55:66",
    "timestamp": "2024-12-18T14:10:00Z",
    "event_type": "speed_violation",
    "speed_kmh": 70,
    "location": "Zone A"
}
```

Intrusion Detection Event

An intrusion detection event photo example can be found in the `assets` folder.

- The image `intrusion-detection-1-alert.jpg` should be used for events with `zone` -> `Restricted Area`
- The image `intrusion-detection-2-no-alert.jpg` should be used for events with `zone` -> `Open Area`

1. **Triggers Alert:**

```
{
    "device_id": "77:88:99:AA:BB:CC",
    "timestamp": "2024-12-18T14:10:00Z",
    "event_type": "motion_detected",
    "zone": "Restricted Area",
    "confidence": 0.95,
    "photo_base64": "<base64-encoded-string-of-intrusion-photo>"
}
```

2. **Does Not Trigger Alert:**

```
{
    "device_id": "77:88:99:AA:BB:CC",
    "timestamp": "2024-12-18T14:20:00Z",
    "event_type": "motion_detected",
    "zone": "Open Area",
    "confidence": 0.80,
    "photo_base64": "<base64-encoded-string-of-non-intrusion-photo>"
}
```

Technical Requirements

1. **Technologies to Use:**
   - **PostgreSQL** for database storage.
   - **Redis** for caching.
   - **RabbitMQ** for message brokering.
   - **FastAPI** or **Django** for building APIs.
2. **Dockerized Setup:**
   - Provide a `Dockerfile` and `docker-compose.yml` file to containerize the application and its dependencies.
3. **Testing:**
   - Include basic unit tests for critical functionality.
4. **Documentation:**
   - Include a `README.md` with:
     - Setup instructions.
     - API endpoint documentation.
     - Explanation of alert criteria.
5. **Architecture Diagram:**
   - Include an architecture diagram demonstrating how the ingestion service, alerting service, RabbitMQ, PostgreSQL, and Redis interact.

Evaluation Criteria

1. **Technical Implementation:**
   - Adherence to best practices in API design, database schema design, and message brokering.
   - Efficient and clean code.
2. **Functionality:**
   - Correct processing and storage of simulated events.
   - Accurate triggering of alerts based on criteria.
3. **Code Quality:**
   - Readability, modularity, and proper use of version control.
4. **Documentation:**
   - Clear setup instructions and thorough explanation of the system.
   - Inclusion of an architecture diagram.

Submission Guidelines

1. **Code Repository:**
    - Share a Git repository (e.g., GitHub, GitLab) containing the project code.
2. **Documentation:**
    - Include a `README.md` file with clear setup and usage instructions.
3. **Docker Setup:**
    - Ensure the application can be run using the provided Docker setup.
4. **Testing:**
    - Include test cases for key features.
5. **Architecture Diagram:**
    - Submit an architecture diagram as part of the documentation.

Good luck! We look forward to reviewing your submission.