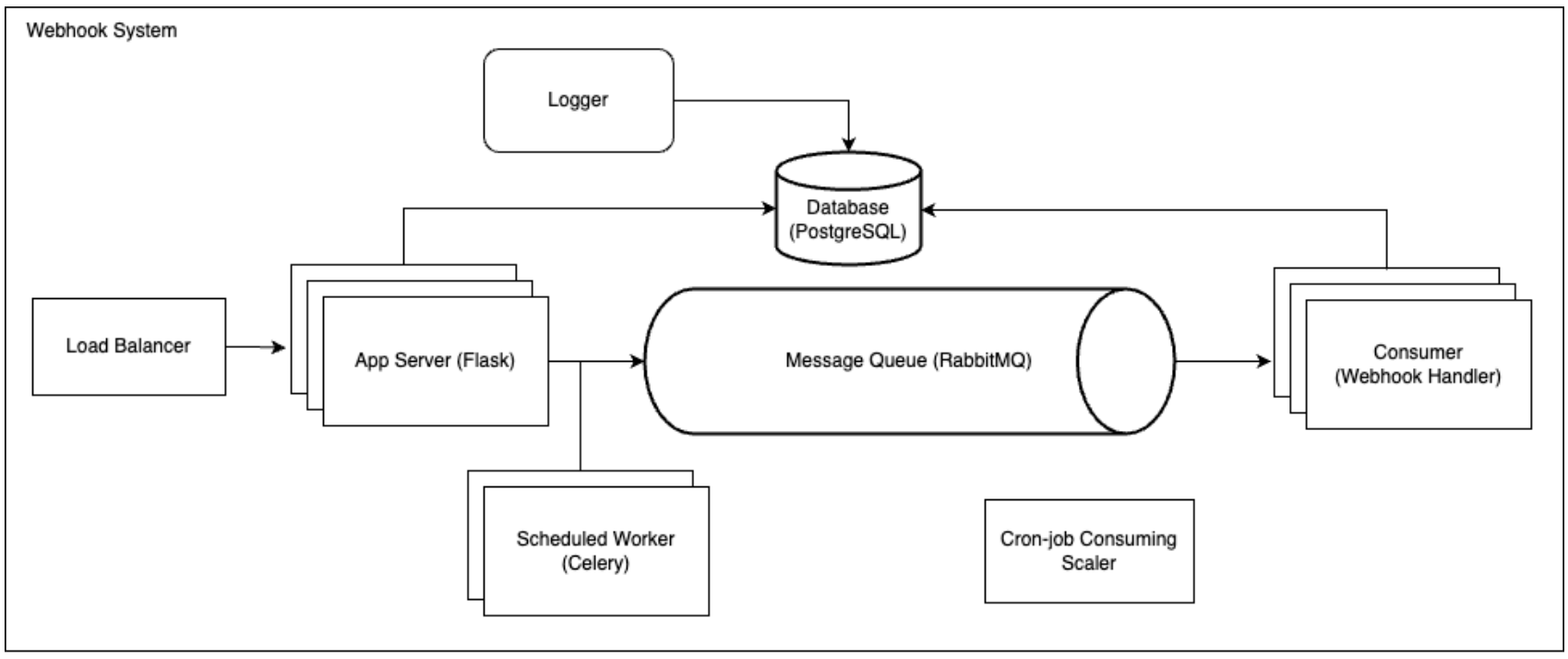
Objective

- Think & create appropriate **architecture** of the solution;
- Develop a webhook system that allows users to **register** and **configure** webhooks;
- Ensure the system can handle, queue, and execute webhooks **efficiently** and **reliably**;
- Implement **error handling** and retry mechanisms

Architecture



Load Balancer: Distribute user requests to App Server.

App Server: The server receives user requests, processes, and allows users to **register** and **configure** webhooks

Scheduled Worker (Celery): Process the event data of scheduled tasks and then put them into the Message Queue. Tasks are registered through the App Server and through error handling at the Consumer.

Message Queue (RabbitMQ): Used to temporarily store event data that will be delivered to the user. These events will be processed by consumers. Using Message Queue will help increase **reliability** and **scalability**.

Cron-job Consuming Scaler: A cron job is used to scale consumers based on the size of the queue. The idea is to coordinate the number of consumers every 60 seconds based on the queue size so that the event delivery rate can be steady to avoid queue high traffic.

Consumer (Webhook Handler): Used to handle delivery events to users, events will be delivered using the POST method.

- Build authentication mechanisms. An algorithm can be used to validate the response for example HMAC-256.
- If the error occurs, the **error handling** mechanism uses retry strategies to set the scheduling time. The retry strategy may include: Fibonacci, repeat after 5 minutes, exponential backoff, ... to calculate the next scheduling time. After that register with Celery.
- Acknowledge the event after it has been processed.

Database (PostgreSQL): PostgreSQL is the choice. The purpose is to take advantage of the parallel writing capabilities of the Database.

Logger: Third-party applications are used to monitor the health of PostgreSQL, App Server, Redis, and RabbitMQ such as Grafana and the scheduled tasks such as Flower.

Technologies

Framework/libraries: Flask, Celery, RabbitMQ, Redis, Docker, Docker Compose.

Database: PostgreSQL.

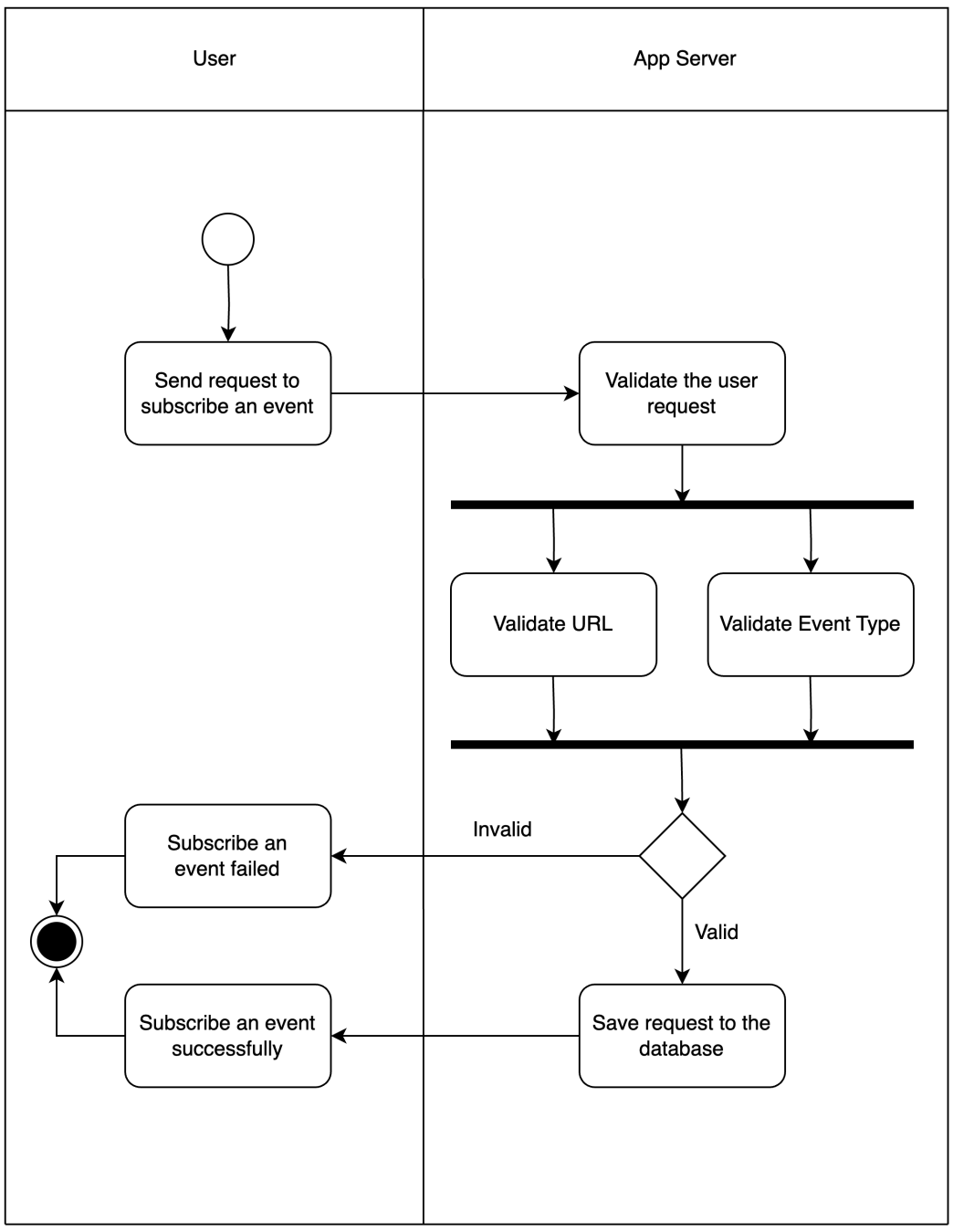
Logger: Grafana.

Tools: ngrok (https tunnel), apache benchmark (for performance testing), Postman (API Testing), flower (monitoring celery task)

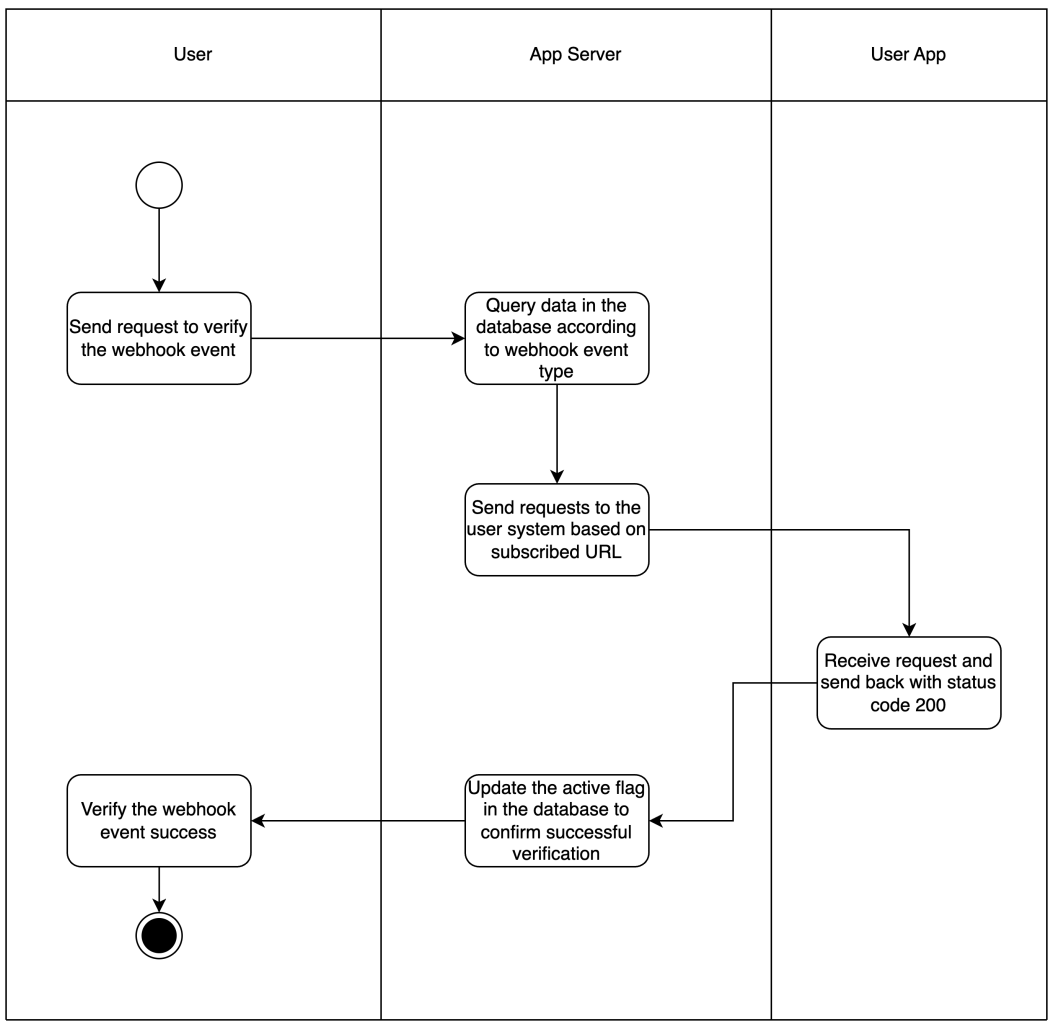
Activity Diagram

We will go into the main activities as the foundation of the system. The system will describe how it operates and the interactions between components in the system using Activity Diagram.

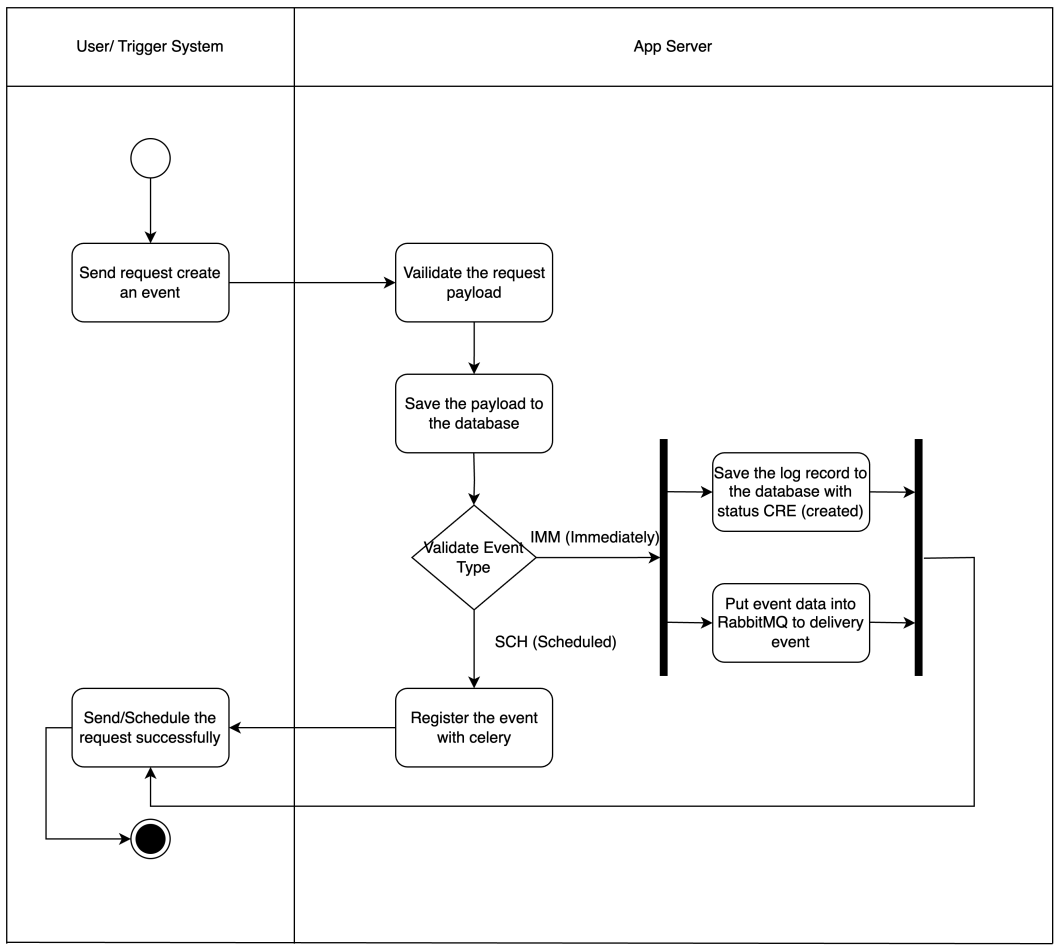
Subscribe to a webhook event



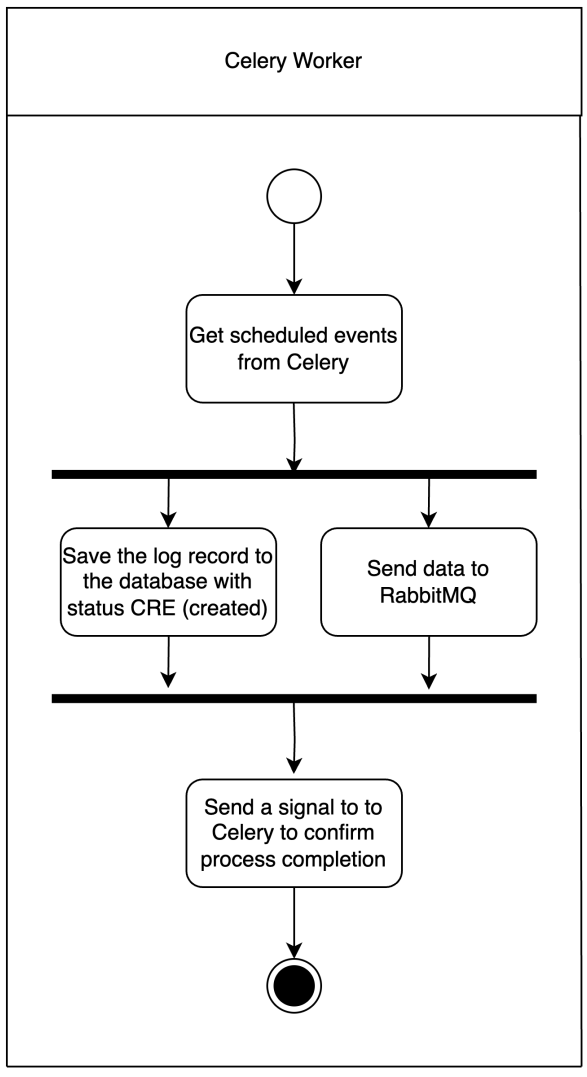
Verify a webhook event



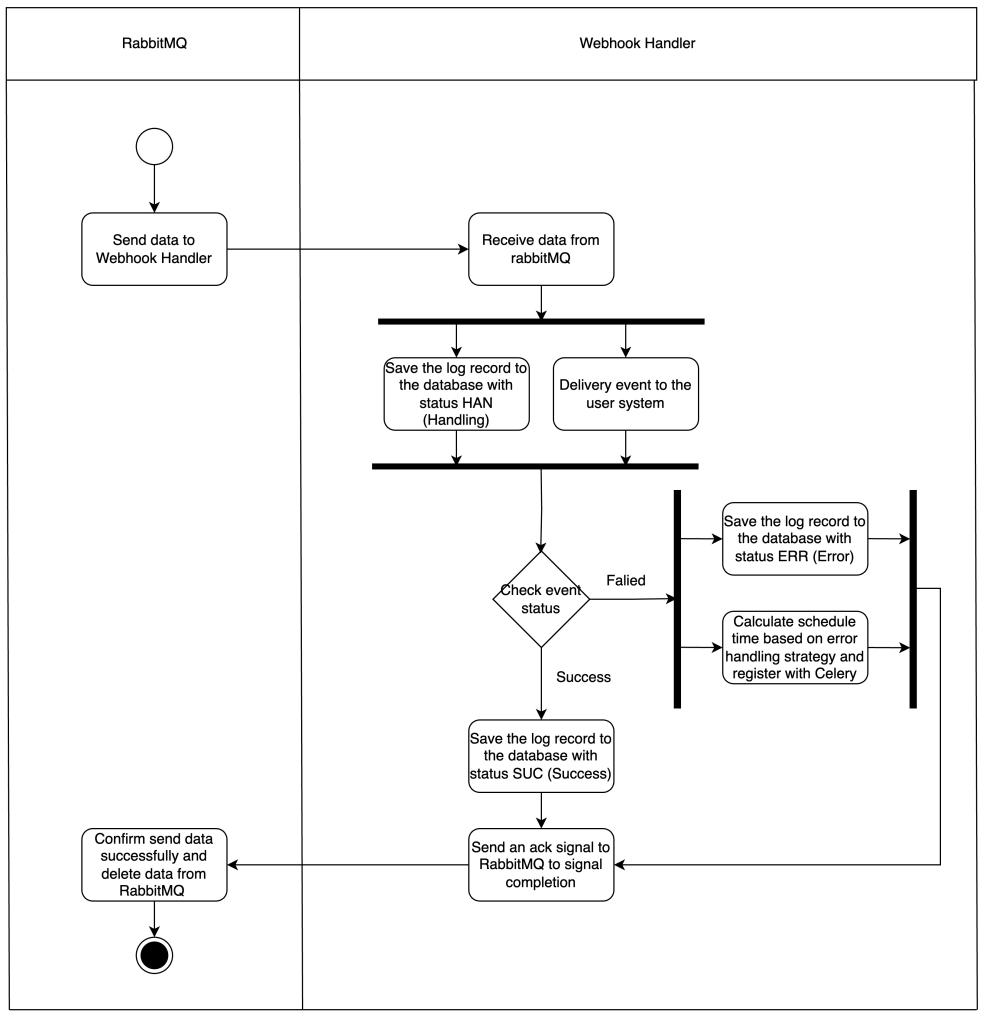
Trigger a webhook event



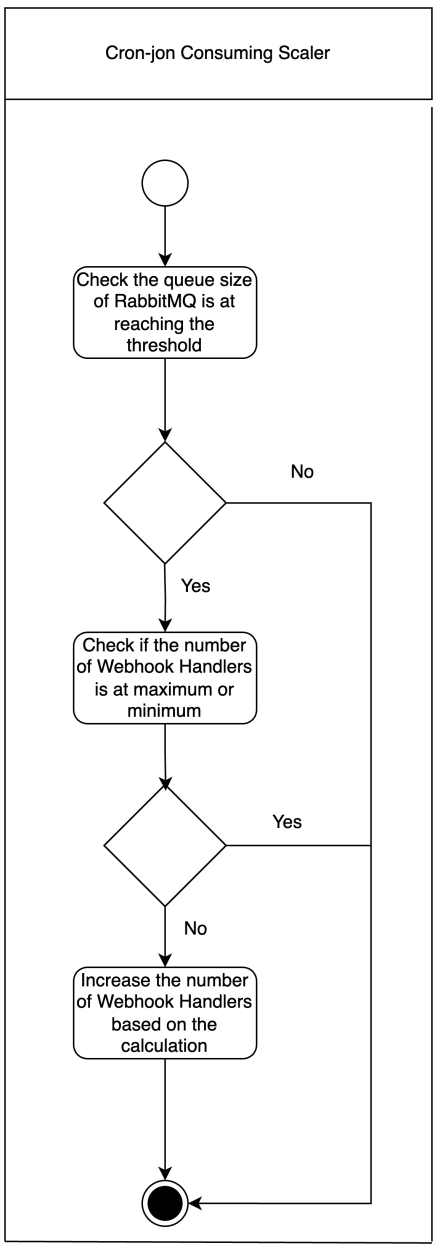
Trigger a scheduled webhook event



Delivery a webhook event and error handling



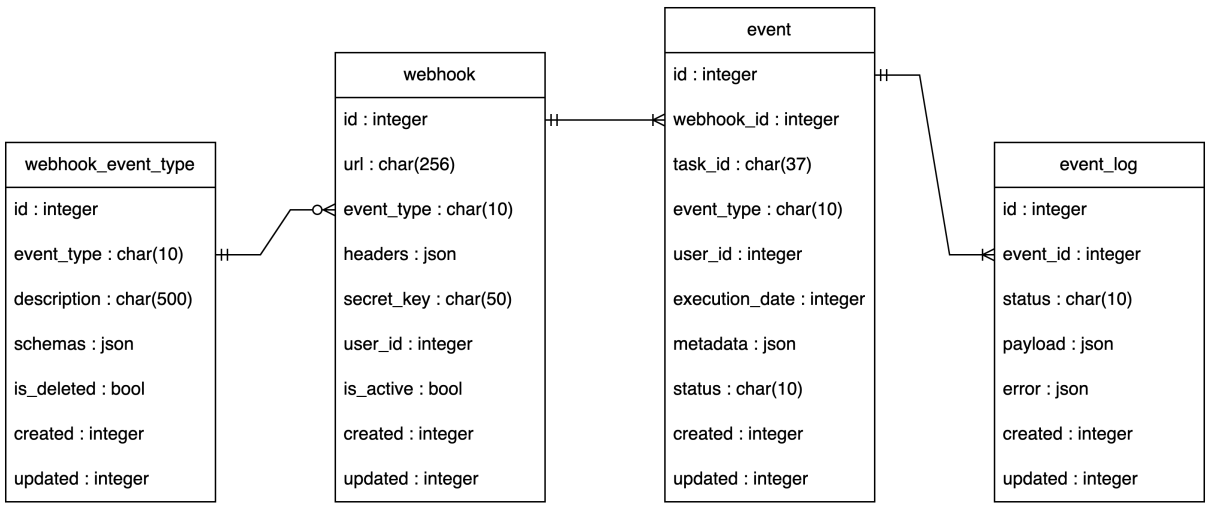
How cron-job Consuming works



Database Design

- For me, the number of write operations is more than the number of read operations in this system. Therefore, to achieve the best performance for the database, we should avoid creating unnecessary constraints to increase the writing speed of the database.
- Use **Composite Index** to index multiple columns to help the database locate and retrieve data without scanning the entire table.
    - In the webhook table, we need to verify whether the event is active and subscribed or not. The query needs 3 fields (event_type, user_id, is_active), so we will do **Composite Index** for these 3 fields to speed up the query.
- Use **Partition** to divide large datasets into smaller, more manageable partitions or subsets to reduce pressure on the database and increase scalability.

    - In the event_log table, we can **Partition** by month and year and then build reports by month and year to help query faster.
- Use **Materialized View** to separate Write and Read independently without real-time updates. In this table, we can transform some data to help reduce the number of records.
    - In the event_log table, every delivered event will create more than 3 event_log records (status: created, running, error, success). So we create a **Materialized View** to transform these records into 1 and refresh this table every 3 seconds(because 3 seconds is enough for an event to go through the entire status).
- Tables:
    - Table: webhook_event_type ~ 1.522kb/record → 1.6gb/1m record
    - Table: webhook ~ 1.176kb/record → 1.2gb/1m record
    - Table: event ~ 1.171kb/record → 1.2gb/1m record
    - Table: event_log ~ 2.026kb/record → 2.1gb/1m record



API Design

- Subscribeable events: list of subscribeable events.
    - Url: /ws/v1/webhook/
    - Method: GET
    - Input:
        - user_id: int
    - Output:
        - List of registered webhooks
- **Subscribe Webhook: Subscribe to events based on event type**
    - Url: /ws/v1/webhook/
    - Method: POST
    - Input:
        - url : string
        - event_type: string
        - headers: object
        - user_id: int
        - metadata: object
    - Output:
        - webhook_id : array
- Unsubscribe Webhook: Unsubscribe to events based on webhook id
    - Url: /ws/v1/webhook/:webhook_id
    - Method: DELETE
    - Input:
        - user_id: int
        - webhook_id: int
    - Output:
        - webhook_id: int
- **Trigger Webhook: Create and trigger a webhook event**
    - Url: /ws/v1/event
    - Method: POST
    - Input:
        - event_type: string

        - trigger_type: string - Now, Scheduled
        - execution_date: int
        - user_id: int
        - payload: object
        - metadata: object
    - Output:
        - event_id
- List of Webhook Events:
    - Url: /ws/v1/event
    - Method: GET
    - Input:
        - user_id: int
    - Output:
        - list event
- Log Webhook Events: the log of webhook events
    - Url: /ws/v1/event/:event_id/log
    - Method: GET
    - Input:
        - user_id: int
    - Output:
        - list event logs