Backend Intern Assignment: Web Analytics Event Service

Goal: Build a robust backend service to collect, store, and provide aggregated analytics for user interaction events (view, click, location).

Time Allotment: 48 hours (estimated concentrated work time)

Overview: This assignment involves creating a backend API that serves as an analytics collector. It will receive various user interaction events from a client (e.g., a web page), store them in a database, and then provide an API for querying these events, including specific filtering and aggregation for reporting. This emphasizes database design, API robustness, and query optimization for analytical purposes.

Key Learning Outcomes:

- Designing a flexible database schema for diverse event data.
- Implementing asynchronous data ingestion via a RESTful API.
- Handling geolocation data.
- Performing complex database queries with filtering and aggregation.
- Implementing robust error handling and data validation.
- Understanding the lifecycle of analytics data from client to insights.

Part 1: Understanding the Data and Requirements

1. Event Data Structure (to be received by Backend):

Each event will have the following properties. Note that `event_id`, `timestamp`, and potentially `user_id` will be generated/managed by the backend to ensure data integrity.

- `user_id`: A string identifier for the user (e.g., a session ID, or a logged-in user's ID).
- `event_type`: A string describing the activity. Allowed values: "view", "click", "location".
- `payload`: A flexible JSON object containing additional context specific to the `event_type`:
  - **For `event_type: "view"`:**
    - `url`: The URL of the page viewed (string).
    - `title`: The title of the page (string, optional).
  - **For `event_type: "click"`:**
    - `element_id`: The ID of the clicked HTML element (string, optional).
    - `text`: The text content of the clicked element (string, optional).
    - `xpath`: A simplified XPath or CSS selector to locate the element (string, optional).
  - **For `event_type: "location"`:**
    - `latitude`: User's latitude (float).
    - `longitude`: User's longitude (float).
    - `accuracy`: Accuracy of the location in meters (float, optional).

2. Backend Core API Endpoints:

You will need to implement the following RESTful API endpoints:

- `POST /events`
  - **Purpose:** To ingest a new user activity event from the client.

    - **Request Body:** A JSON object representing a single event, conforming to the `Event Data Structure` (without `event_id`, `timestamp`).
    - **Request Headers:** Consider `Content-Type: application/json`.
    - **Response:**
        - `202 Accepted`: If the event is successfully received and queued for storage (asynchronous processing is good here, but for 48h, direct storage is fine).
        - `400 Bad Request`: If the request body is invalid or missing required fields.
        - `500 Internal Server Error`: For server-side issues during processing or storage.
- `GET /analytics/event-counts`
    - **Purpose:** Retrieve the total count of events, with optional filtering.
    - **Query Parameters (Optional):**
        - `event_type`: Filter by a specific event type ("view", "click", "location").
        - `start_date`: Filter events occurring *on or after* this date (ISO 8601 date, e.g., "2025-05-28").
        - `end_date`: Filter events occurring *on or before* this date (ISO 8601 date, e.g., "2025-05-29").
    - **Response:**
        - `200 OK` with a JSON object containing the total count.

          ```
          JSON
          ```

          ```
          {
            "total_events": 12345
          }

          ```
        - `400 Bad Request`: If query parameters are invalid (e.g., malformed date).
- `GET /analytics/event-counts-by-type`
    - **Purpose:** Retrieve the count of events grouped by `event_type`, with optional filtering.
    - **Query Parameters (Optional):**
        - `start_date`: Start date for aggregation (ISO 8601 date, e.g., "2025-05-28").
        - `end_date`: End date for aggregation (ISO 8601 date, e.g., "2025-05-29").
    - **Response:**
        - `200 OK` with a JSON object where keys are `event_type` and values are their counts.

          ```
          JSON
          ```

          ```
          {
            "view": 8000,
            "click": 3000,
            "location": 1345
          }

          ```
        - `400 Bad Request`: If query parameters are invalid.
        - `200 OK` with empty object `{}` if no events match criteria.

3. Data Generation (Crucial for "Data-Backed"):

- You *must* generate a sample dataset of at least **1,000 to 5,000 events** distributed across various `user_id`s, with a realistic mix of "view", "click", and "location" events.
- Timestamps should span a few weeks (e.g., from `2025-05-01` to `2025-05-29`).
- `payload` data should be representative (e.g., varying URLs, element IDs, and plausible lat/long coordinates).

- This data should be used to pre-populate your database for testing the analytics endpoints. You can use a script (e.g., Python with `Faker`) to achieve this.

Part 2: Technical Design and Implementation

1. Choose Your Stack:

- **Language:** Python, Node.js (Focus on familiarity for speed).
- **Framework:** Flask/FastAPI (Python), Express.js (Node.js)
- **Database:** Any (*Ignore the mentions of PostgreSQL in this doc.*)

2. Steps to Success:

Step 2.1: Project Setup & Database Schema

- Initialize your project.
- Install necessary dependencies (web framework, database driver/ORM).
- **Database Schema Design (`events` table):**
    - `event_id`: UUID (Primary Key).
    - `user_id`: TEXT/VARCHAR (Indexed, for quick lookups).
    - `event_type`: TEXT/VARCHAR (Indexed, `ENUM` if your DB supports it, like "view", "click", "location").
    - `timestamp`: TIMESTAMP WITH TIME ZONE (Indexed, crucial for date-based queries).
    - `payload`: JSONB (PostgreSQL) or TEXT (SQLite, storing JSON string). This stores the event-specific details.
    - Justify your choice of data types and indexes in your `README.md`.

Step 2.2: Data Generation and Initial Population

- Write a Python script (`generate_events.py`) using `Faker` to generate realistic sample data conforming to the `Event Data Structure`.
- The script should connect to your database and insert these 1,000-5,000 events.
- **Verification:** Run simple SQL queries to confirm data population and correct structure.

Step 2.3: Implement `POST /events` Endpoint

- Set up your web server and define the `/events` route.
- Implement the `POST /events` endpoint:
    - **Request Parsing:** Parse the incoming JSON body.
    - **Validation:**
        - Ensure `user_id` and `event_type` are present and non-empty.
        - Validate `event_type` is one of "view", "click", "location".
        - Validate `payload` structure based on `event_type` (e.g., if `event_type` is "view", `payload` must have `url`).
        - Validate latitude/longitude are floats within valid ranges for "location" events.
    - **Data Enrichment:**
        - Generate a unique `event_id` (UUID).
        - Generate a `timestamp` (UTC current time) on the backend.
    - **Database Insertion:** Store the validated and enriched event in your `events` table.
    - **Response:** Send `202 Accepted` on success, `400 Bad Request` with meaningful error messages for validation failures, and `500 Internal Server Error` for database or server issues.

Step 2.4: Implement `GET /analytics/event-counts` Endpoint

- Implement the `/analytics/event-counts` endpoint.
- Parse `event_type`, `start_date`, and `end_date` query parameters.

- Construct a database query to count all events:
  - Add `WHERE event_type = :event_type` if `event_type` query parameter is provided.
  - Add `WHERE timestamp >= :start_date AND timestamp <= :end_date` if date parameters are provided.
- Return the total count in the specified JSON format.
- Handle `400 Bad Request` for invalid date formats or other query parameter issues.

Step 2.5: Implement `GET /analytics/event-counts-by-type` Endpoint

- Implement the `/analytics/event-counts-by-type` endpoint.
- Parse `start_date` and `end_date` query parameters.
- Construct a database query to count events grouped by `event_type`:
  - Example SQL (PostgreSQL): `SELECT event_type, COUNT(*) FROM events WHERE timestamp BETWEEN :start_date AND :end_date GROUP BY event_type;`
  - If no date filters, just `SELECT event_type, COUNT(*) FROM events GROUP BY event_type;`
- Format the results into the required JSON object structure (`{"view": 123, "click": 45}`).
- Handle `400 Bad Request` for invalid query parameters.

Step 2.6: Basic Error Handling and Logging

- Implement a global error handler for unhandled exceptions (e.g., database connection issues).
- Add basic logging (e.g., to console or file) for incoming requests, successful operations, and all errors.

Step 2.7: Testing (Postman/cURL) and Client-Side Example

- Manually test all your API endpoints using Postman, Insomnia, or cURL.
- Test edge cases:
  - Invalid request bodies for `POST` (missing fields, wrong data types, invalid `event_type`).
  - Date ranges with no events for `GET` endpoints.
  - Missing/invalid query parameters.
  - **Crucially:** Test the filters for `GET /analytics/event-counts`.

Bonus (Optional - if time permits):

- **Service Worker and Frontend Integration**
  - **Create `index.html`:** A simple HTML file to demonstrate the client. It should:
    - Register the `service-worker.js` script.
    - Have a `DOMContentLoaded` event listener to send a "view" event.
    - Have a button with `id="click-me"` and an event listener to send a "click" event when clicked.
    - Have a button with `id="get-location"` and an event listener to send a "location" event (using `navigator.geolocation`) when clicked.
    - **Crucially**: Your backend must be running locally (e.g., `http://localhost:5000`) for the service worker to send events to.
  - **Create `service-worker.js`:** This file will be registered by `index.html`.
    - **Registration Listener:** `self.addEventListener('install', ...)` and `self.addEventListener('activate', ...)` to ensure it activates.
    - **Message Listener:** `self.addEventListener('message', (event) => { ... })`
      - The main page (via `postMessage`) will send event data to the Service Worker.
      - Upon receiving an event, the Service Worker should use `fetch()` to send the event data **asynchronously** to your backend's `POST /events` endpoint.
      - **Important for 48h:** Do **not** implement complex caching, network interception, or IndexedDB for offline queueing within the service worker. Focus solely on receiving the message and `fetch()`ing it to the backend. The "asynchronous" part is handled by `fetch` itself.

Part 3: Deliverables

By the end of the 48 hours, you should provide:

1. **A Git Repository Link:**
   - Cleanly structured code.
   - A `README.md` file (see next point).
   - Database schemas
   - Your data generation script (`generate_events.py`).
   - (Optional) The simple client-side HTML/JS example.
2. `README.md` **File:**
   - **Setup Instructions:** Clear, step-by-step instructions on how to set up the project locally (prerequisites, dependencies, database setup, how to run the data generation script, how to start the backend service).
   - **API Documentation:** For each implemented endpoint:
     - HTTP Method & Path
     - Purpose
     - Request Body Example (for `POST`)
     - Query Parameters (for `GET` endpoints)
     - Success Response Example (JSON)
     - Error Response Examples (JSON & HTTP Status Codes)
   - **Chosen Technologies:** List the language, framework, database, and any other significant libraries used, with a brief justification for your choices (e.g., "Chose Flask for its simplicity and Python familiarity").
   - **Database Schema Explanation:** Briefly explain your `events` table design and why you chose certain data types and indexes.
   - **Challenges Faced & Solutions:** Describe any significant technical challenges you encountered (e.g., complex SQL queries, specific validation logic) and how you overcame them.
   - **Future Improvements:** Ideas for how the service could be extended or improved (e.g., user authentication, data aggregation caching, more sophisticated analytics, real-time dashboards, using a message queue for async processing, proper spatial indexing for location data).

Evaluation Criteria:

- **Functionality:** Do all endpoints work as specified? Are the filters and aggregations correct?
- **Correctness:** Are data validations correctly implemented? Is data stored and retrieved accurately?
- **Code Quality:** Readability, modularity, appropriate use of chosen framework/language features, consistent coding style, meaningful variable/function names.
- **Database Design:** Appropriateness of schema, indexing for performance, handling of `payload` JSON data.
- **Error Handling:** Robustness of error responses and logging.
- **Documentation:** Clarity, completeness, and accuracy of the `README.md`.
- **Adherence to Requirements:** All specified features implemented.
- **Time Management:** Demonstrating ability to deliver a functional product within constraints.

Good luck! This assignment will test your ability to build a practical, data-driven backend service.