Problem Statement – AlgoHire Webhook Event Relay System

Problem Summary

AlgoHire is a multi-service recruitment platform where multiple internal modules such as Jobs, Candidates, Interviews, and Assessments generate various activity events. These internal actions often need to be communicated to external systems such as client HRMS or CRM platforms in real-time to ensure synchronization of hiring data and process visibility.

At present, there is no unified or secure mechanism to handle this flow. Each partner integration must be developed separately, leading to redundant work, inconsistent data handling, and maintenance overheads.

This challenge requires you to build a **Webhook Event Relay System** that acts as a secure, scalable, and observable event gateway for AlgoHire. The system should allow external systems to **subscribe to specific event types** and receive **real-time notifications** when relevant actions occur inside AlgoHire.

The relay should ensure **reliability, idempotency, and security**, and should be supported by a **dashboard interface** to configure, view, and manage subscriptions and event delivery logs.

Objective

Build a **full-stack Webhook Event Relay System** that performs the following core functions:

1. Receives and stores events generated by internal AlgoHire modules.
2. Allows external clients to register webhook endpoints for specific event types.
3. Pushes corresponding event payloads to those webhooks reliably and securely.
4. Ensures message delivery integrity with retry logic, caching, and idempotency.
5. Provides an internal dashboard for managing webhook configurations, viewing delivery reports, and monitoring system health.
6. Uses a proper database and caching mechanism for optimized performance and scalability.
7. Implements authentication, validation, and HMAC signature verification for secure delivery.

Basic System Schematic

Inputs

- Internal event requests representing actions such as job creation or candidate status updates.
- External webhook registration requests from clients specifying which events they want to subscribe to.
- Administrative actions from internal users for monitoring, retrying, or disabling webhook subscriptions.

Processing

- Events are validated, logged, and stored in the database.
- Subscribers for the event type are retrieved and queued for dispatch.

- Cached data such as active webhooks or recent delivery logs is served through Redis to reduce database load.
- Background workers handle asynchronous delivery with retry mechanisms.

Outputs

- Structured event payloads sent to external systems via HTTP webhooks, with secure headers for authenticity.
- Delivery logs and audit records visible through the management dashboard.
- Performance metrics and activity statistics to observe system behavior.

Implementation Requirements

Participants must build this system with the following engineering expectations:

Event Management and Idempotency
Each event should be uniquely identified and processed exactly once. Duplicate submissions or replays should not result in redundant deliveries.

Security and Verification
Every outgoing webhook must be signed using an HMAC signature to validate that it originated from AlgoHire. All incoming inputs must be authenticated and validated properly.

Database and Cache Layer
Use a relational database to store events, subscriptions, and delivery logs. Implement a caching layer for temporary data, such as event queues, active subscriptions, and retry tracking.

Dashboard and Observability
A web interface should allow internal administrators to view all webhook configurations, inspect delivery histories, retry failed deliveries, and observe system status.

Optimization and Maintainability
The application should be designed for long-term scalability and operational reliability. Focus on separation of concerns, reusability of modules, and efficiency in database and cache interactions.

Documentation and Explanation
Provide both functional and API documentation explaining the flow of data, the reasoning behind your architecture, and how another developer could extend it.

Technology Expectations

The following technology choices are **mandatory** and form part of the assessment criteria:

- **Backend:** Node.js
  - Use frameworks such as Express
  - Focus on modular code organization, async handling, and structured logging.
- **Frontend:** React with **ShadCN UI** or similar modern component library
  - Implement a functional dashboard with pages for managing webhooks, viewing delivery logs, and retrying failed events.
  - Ensure responsive and clean design principles.
- **Database:** PostgreSQL

    - Use it for persisting events, webhook subscriptions, and delivery logs.
    - Define proper relationships and indexing for efficient query execution.
- **Cache Layer:** Redis
    - Use Redis for caching subscriptions, managing retry queues, and optimizing repeated database lookups.
    - Demonstrate proper key naming and expiry handling.

Participants are free to integrate other supporting technologies or services that complement these core requirements.
Examples include:

- Background processing libraries (like BullMQ, Agenda, or custom workers).
- Docker for containerization.
- Nginx for routing and load simulation.
- Testing frameworks such as Jest or Mocha.

The provided stack ensures the solution reflects real-world engineering expectations at AlgoHire.

Expected Output

At the end of the hackathon, participants must submit the following deliverables:

1. A **fully working Webhook Event Relay System** implementing the described functionality.
2. A **backend service** built with Node.js and PostgreSQL, integrated with Redis for caching.
3. A **frontend dashboard** built using React and ShadCN UI for monitoring, control, and reporting.
4. **Functional documentation** explaining the data flow, architecture, and how the system ensures idempotency and security.
5. **API documentation** describing endpoints, parameters, and response structures for integrations.
6. A **short video submission** (3–5 minutes) explaining:
    - What was built and how it works.
    - Why specific tools and technologies were chosen.
    - How reliability, scalability, and optimization were approached.
    - What challenges were encountered during development.
    - What future improvements or enhancements could make it fully production-ready.

Functional Expectations

- Events generated internally must flow through the system to relevant external subscribers.
- The entire delivery lifecycle should be visible on the dashboard, including retries and failures.
- Duplicate events must be avoided through idempotent design.
- Redis should reduce redundant database queries and improve overall responsiveness.

- Security should be enforced through HMAC verification.
- Documentation should allow another engineer to quickly understand and extend the project.

Evaluation Considerations

Submissions will be evaluated based on:

- Completeness of the end-to-end functionality.
- Quality of architecture and adherence to Node.js and React best practices.
- Correct and efficient use of PostgreSQL and Redis.
- Design, clarity, and usability of the frontend dashboard.
- Robustness in error handling, retry mechanisms, and observability.
- Clarity and completeness of documentation.
- Technical explanation and presentation quality in the video submission.
- Forward-thinking about production readiness and future optimization.

Problem Essence

This challenge is designed to test real engineering capability — not just coding proficiency.
It involves building a system that represents modern SaaS architecture, where asynchronous event handling, reliability, and transparency are critical.
Participants are expected to think like engineers building a scalable product, not just a prototype.
They should demonstrate an understanding of backend data flow, frontend clarity, system security, and performance optimization.
The solution should reflect production-level thoughtfulness: clean design, modularity, maintainability, and clear observability.