

# ZeroBeta Assessment - Software Engineer

## Problem Overview

You have been asked to build a small platform that ingests **realtime purchase/order events** from two partners, validates them, converts them into a single internal schema, and makes them available to a dashboard.

This platform will be used to compute basic sales statistics and show recent orders. The platform must be designed so it can be deployed in the cloud (AWS preferable), but we do **not** require you to actually deploy it.

## Functional Requirements

- Receive order events from two partners (**Partner A** and **Partner B**).
- Validate received events (data types + missing fields). See Validations section below.
- If valid:
  - Convert to an internal schema (**OrderEvent**)
  - Assign a **sequenceNumber** per partner (starts at 1 per partner)
  - Publish to a **valid\_orders** stream (can be a queue/topic abstraction)
- If invalid:
  - Publish to an **error\_orders** stream with error reason
- Persist processed valid orders for querying.
- Expose REST APIs to:
  - Fetch orders in a time range
  - Fetch monthly summary totals
- Provide a simple **React UI** to view orders and summary.

Note: You can implement the “stream/topic” with an in-memory queue for local dev. In your design doc, explain what you would use in AWS.

## Validations

1. **Data type validations**
2. **Missing required fields**

3. Required fields **cannot be null/empty**
4. Numeric fields must be valid numbers (reject "abc", "12..3")

If an event fails validation, it must go to `error_orders` with:

- `partnerId`
- `raw payload`
- `receivedTime`
- `validationErrors[]` (list of strings)

## Incoming Data Formats

Assume two partners: A and B.

### Partner A

Format: JSON

Keys:

1. `skuId` – string (**required**)
2. `transactionTimeMs` – long (epoch milliseconds) (**required**)
3. `amount` – decimal (**required**)

Example:

```
JSON
{
  "skuId": "SKU-1001",
  "transactionTimeMs": 1733059200123,
  "amount": 25.50
}
```

### Partner B

Format: JSON

Keys:

1. `itemCode` – string (**required**)
2. `purchaseTime` – string timestamp (YYYY-MM-DD HH:mm:ss) (**required**)
3. `total` – decimal (**required**)
4. `discount` – decimal (**optional**)

Example:

```
JSON
{
  "itemCode": "IT-900",
  "purchaseTime": "2026-01-28 10:12:30",
  "total": 100.00,
  "discount": 10.00
}
```

## Internal Schema (after validation + transform)

### POJO / Model: **OrderEvent**

Parameters:

1. **productCode** — string (from **skuId** or **itemCode**)
2. **eventTime** — timestamp (converted to ISO 8601 UTC)
3. **grossAmount** — decimal (from **amount** or **total**)
4. **discount** — decimal (optional; default 0 if missing)
5. **netAmount** — decimal (**grossAmount** - **discount**)
6. **partnerId** — string (A or B)
7. **sequenceNumber** — long (generated per partner, starts at 1)
8. **receivedTime** — timestamp (when your system received it)
9. **streamOffset** — long (optional; if you simulate a queue/topic you can set 0)
10. **processedTime** — timestamp (when transform completed)

## Components to Implement

Summarizing the platform modules (keep these as separate packages/classes/services):

1. **Partner Onboarding (simplified)**
  - Hardcode partner IDs **A** and **B** (no full signup/login required).
  - (Optional) basic API key per partner.
2. **Feed Handler**
  - Accept order events (HTTP endpoint is OK).
  - Identify partner (**A** or **B**) and parse payload.

- Validate and transform to `OrderEvent`.
  - Generate `sequenceNumber` per partner.
3. **Error Processor**
    - Receives invalid events and stores them for later viewing (in-memory or DB).
  4. **Order Processor**
    - Consumes valid `OrderEvent` and persists it.
    - Must be safe to retry (no duplicates if same event is reprocessed — describe your approach).
  5. **REST API Module**
    - Expose query endpoints (see below).
  6. **React Dashboard**
    - Show recent orders + monthly summary.

## Streams / Topics (Abstract)

Use names (even if implemented as in-memory queues locally):

- `valid_orders`
- `error_orders`

If you use Kafka locally, great. If not, an in-memory queue is fine. In the cloud design section, explain what you'd replace it with (e.g., SQS/SNS/EventBridge/Kinesis).

## REST APIs

All APIs should be accessible in an “authenticated context” (you can keep it simple: a single header token or no auth, but document what you'd do in AWS).

### Orders

1. **Fetch orders between two times**
  - `GET /api/orders?partnerId=A&from=...&to=...`
2. **Fetch monthly sales summary**
  - `GET /api/orders/summary/monthly?partnerId=A&month=MM-YYYY`

- Response: totalGross, totalDiscount, totalNet, orderCount

## Errors (optional but recommended)

### 3. Fetch validation errors

- `GET /api/errors?partnerId=A&from=...&to=...`

## React Frontend Requirements

Create a small React app with 2–3 views/panels:

### 1. Recent Orders

- filter by partnerId (A/B)
- show last N orders (timestamp, productCode, netAmount, partnerId, sequenceNumber)

### 2. Monthly Summary

- partnerId + month selector (MM-YYYY)
- show totals

### 3. (Optional) Errors View

- show latest invalid events and validation reasons

Frontend expectations:

- clean component structure
- loading & error states
- basic input validation
- reasonable UI (no fancy design required)

## Cloud / Deployment Design (Not required to implement)

Explain how you would deploy this platform on AWS:

### 1. Compute

- Would you run it as Lambda or containers (ECS/Fargate)? Why?

### 2. Streaming

- What would replace `valid_orders` / `error_orders` (SQS, EventBridge, Kinesis, MSK)?

### 3. Storage

- What would you use for orders and errors (DynamoDB vs RDS)? Why?
- 4. **Frontend hosting**
  - How to host the React app (S3 + CloudFront recommended)
- 5. **Configuration & secrets**
  - env vars, parameter store, secrets manager
- 6. **Observability**
  - logs/metrics/alarms you'd set up
- 7. **CI/CD**
  - pipeline stages (lint/test/build/package/deploy), and how you'd roll out safely

You do **not** need to actually create AWS resources. This is about reasoning and deployability.

## Things to Consider While Implementing

1. Handling retries / reprocessing safely (idempotency)
2. Efficient persistence mechanism (even if in-memory locally, design for real DB)
3. Clear validation errors (actionable messages)
4. Separation of core logic from transport (HTTP vs queue)
5. Performance basics (batch processing optional)

## Deliverables

1. Source code repository
  - Backend + tests
  - Frontend (React) + basic tests (optional)
2. README with:
  - how to run backend + frontend locally
  - how to run tests
  - assumptions/tradeoffs
3. Design doc (includes cloud/deployment thinking)
4. (Optional) Dockerfile(s) + docker-compose to run locally

## Evaluation Criteria

1. **Correctness** (validation, transform, sequence numbers, summaries)
2. **Code quality** (structure, readability, maintainability)
3. **Testing** (unit tests)
4. **Frontend quality** (clean React patterns, error/loading states)
5. **Cloud deployment reasoning** (practical AWS plan)
6. **Observability mindset** (useful logs/metrics)