Available online www.ejaet.com

European Journal of Advances in Engineering and Technology, 2021, 8(5):53-58

Research Article ISSN: 2394 - 658X

Event Handler Platform for Large-Scale Supply Chain Systems

Gautham Ram Rajendiran

gautham.rajendiran@icloud.com

ABSTRACT

In large-scale supply chain systems that are built with the service oriented architecture pattern [1], event handling plays a critical role in managing operations like inventory updates, order processing and logistics [3]. However, inconsistent event-handling mechanisms across teams dedicated to managing such a large scale system can lead to a lack of holistic view of events that enable proactive identification of supply chain risks [4] in the system and redundant development work to handle events. This paper proposes a centralized event-handling platform that standardizes interfaces, provides a holistic view of system events and simplifies development by providing abstract implementations of error handling, logging, retry mechanism and other common functionality. The platform enhances operational efficiency and other qualitative benefits, offering an efficient solution to event handling for teams managing large scale supply chain systems. Future development can focus on reducing error response times even further by enabling AI driven automation using the centralized event logs to identify patterns and formulate responses based on prior experience.

Keywords: supply chain, event handling, microservices

INTRODUCTION

It is a common practice in software engineering to decompose large systems into smaller micro services when following the Service Oriented Architecture pattern [1]. Large-Scale supply chain systems are a case where adopting such an architecture is beneficial due to the number of components involved. Companies manage the implementation, maintenance and operation of such systems by allocating teams dedicated to the ownership of a group of microservices [2]. In such a distributed environment, it is easy for teams to build their own event handling systems with differing standards for logging, monitoring and interfaces for their services. While this largely works, it may be cumbersome to get a holistic view of all the events processed by the system. Typically the entry point of such actions is through an event, this could be inventory update events, order events, shipping or logistics events etc. If every team maintains their own mechanism to handle such events, it will be cumbersome to trace the lifecycle of events pertaining to fulfilling an order, which would then result in longer cycle times to trace the root cause of risks when they occur, thereby reducing operational efficiency. Hence such a system would benefit by the presence of a mechanism to enforce standardization of interfaces and the logging and monitoring of such events. This paper explores the design and implementation of such a system which was subsequently implemented in a large scale ecommerce company.

PROBLEM STATEMENT

Through the implementation of such a component, we aim to solve the following problems:

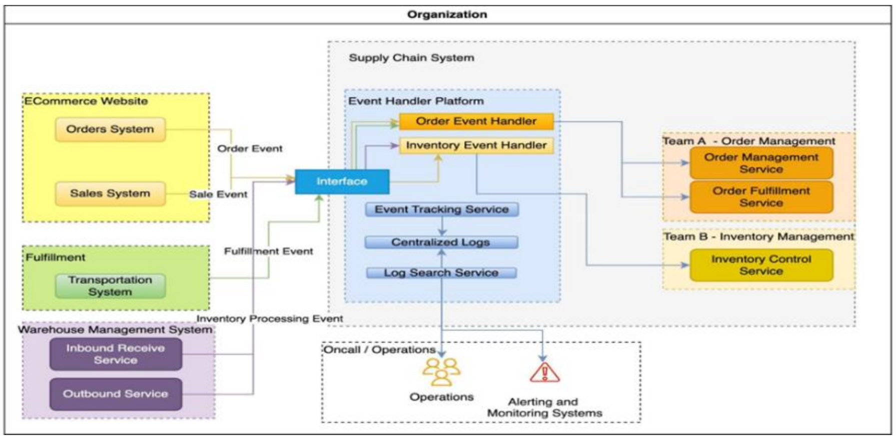
1. Increased time to debug root cause of failures caused due to supply chain risks
2. Inconsistent logging, monitoring standards and retry handling strategies.
3. Redundant development effort since each team needs to build the event handling system from scratch.
4. Inconsistent interfaces to external systems which might create confusion to upstream data providers.

LITERATURE REVIEW

Event-driven architectures (EDA) have become a fundamental component in modern distributed systems [5]. It enhances scalability and avoids tightly coupled architectures. Idempotent event processing is crucial in ensuring that events are handled reliably in cases of retries or failures and remains a key focus in the design of fault-tolerant systems. Research on automated logging, retry mechanisms and monitoring systems has highlighted the importance of centralized observability [6] and error handling to enhance system performance and reduce developer workload.

Event Handler Platform

Architecture



The diagram above demonstrates how an event handler platform integrates into a supply chain system, particularly in organizations managing eCommerce and logistics. Various sub-organizations, such as an eCommerce website, fulfillment platforms, or warehouse management systems generate different types of events (e.g., order events, sales events, fulfillment events, inventory updates etc.). These events need to be processed and acted upon by specific microservices in the supply chain system, which can be owned across different teams.

Each team, such as order management or inventory management, leverages the event handler platform by extending an abstract implementation of the event handler interface to implement their own domain-specific logic. For instance:

- **Order Management Team:** Manages the Order Event Handler, which processes predefined order events (like order creation or status changes). These events are forwarded to either the Order Management Service or the Order Fulfillment Service, ensuring that relevant business logic is applied to each event.
- **Inventory Management Team:** Utilizes the Inventory Event Handler, which listens to inventory-related events originating from the eCommerce website, fulfillment systems, or warehouse systems. These events are routed to the Inventory Control Service for processing and updates.

The operations team ensures the smooth functioning of the supply chain system. With the event handler platform in place, the team has access to a centralized logging system that provides real-time visibility into the flow of events across the entire supply chain. These logs contain crucial information on order events, inventory movements, sales and fulfillment progress. These events can be used to monitor issues caused by supply chain risks.

By leveraging the Log Search Service, the operations team can perform the following tasks:

- **Real-Time Monitoring:** They can monitor the status of orders, fulfillment processes and inventory updates as they happen. For example, when an order is placed, the platform logs the order event, allowing the operations team to track it through various stages—from processing, inventory checks and packaging to shipment and delivery.
- **Alerting and Issue Resolution:** The platform generates automated alerts for exceptional conditions like late-arriving events and failures in order processing. These alerts are triggered based on predefined rules, ensuring the operations team can take prompt action to resolve issues before they escalate. For instance, if an inventory update fails, the operations team can immediately investigate the issue by reviewing event logs and coordinating with relevant teams (e.g., warehouse management).
- **Root Cause Analysis:** In case of supply chain disruptions (e.g., delivery delays, order cancellations, or system failures), the centralized logging system provides a complete audit trail of the events leading up to the issue. This allows the operations team to perform root cause analysis, trace the source of the problem, and coordinate with teams responsible for the affected services (e.g., order fulfillment or inventory control).
- **Performance Reporting:** The centralized logging and event tracking system also allows the operations team to generate performance reports, offering insights into order processing times, fulfillment efficiency and overall system health. These reports help the management make informed decisions to make system optimizations.

The centralized nature of the event handler platform allows the operations team to have a single source of truth for all events occurring within the system. This unified view not only improves operational efficiency but also enhances responsiveness to disruptions, ensuring that issues are resolved quickly with minimal impact on customers and overall business operations.

CORE COMPONENTS

Event Handler Abstract Class (The Interface)

This class acts as a foundational component for handling events in a reliable and traceable way. It enforces idempotency to duplicate processing, guarantee at-least-once [7] delivery and handling retry. By using an abstract class, it enforces certain behaviors across all derived event handlers while leaving the specific event handling logic to be implemented by different teams.

The following are the responsibilities of the interface:

1. Inject Idempotent Event ID: Every event must have an idempotent ID that ensures the event is processed only once, regardless of how many times it is received.

2. Consume the Event: The class is responsible for consuming the event from a source (like a message queue, stream, etc.).

3. Log the Event: Every event must be logged for audit and traceability. This enables tracking the lifecycle of an event through various services.

4. Emit Event Metrics: Emit metrics for each event, such as processing time, success/failure rates, and retry counts, which are useful for monitoring and alerting.

5. Handle Retries and Failures: The class should handle retries for events that fail due to transient issues and implement a backoff strategy to prevent overwhelming the system.

6. Expose the Abstract handleEvent Method: Teams that extend this class will implement their own logic in the handleEvent method.

These functionalities are shown in the pseudocode below:

Algorithm 1: Abstract EventHandler Class
Input: event (the incoming event to be processed)
Output: Processed event or error handling

```
1. Function injectIdempotentID(event):
2.     if event.ID does not exist then
3.         event.ID ← Generate new event ID
4.     else
5.         Use existing event ID
6.     end if
7. Function consumeEvent(event):
8.     Call injectIdempotentID(event)
9.     Call logEvent(event)
10.    try:
11.        Call handleEvent(event)  // Abstract method, to be implemented by subclass
12.        Call emitMetrics(event, success = True)
13.    catch Exception e:
14.        Call handleFailure(event, e)
15.        Call emitMetrics(event, success = False)
16.    end try
17. Abstract Function handleEvent(event):
18.     // To be implemented by other teams, includes event-specific logic
19. Function logEvent(event):
20.    Log event.ID and relevant event details (e.g., timestamp, event type)
21. Function emitMetrics(event, success):
22.    if success == True then
23.        Emit processing time and success metrics
24.    else
25.        Emit failure and retry metrics
26.    end if
27. Function handleFailure(event, exception):
28.    Log the error details with event.ID and exception
29.    if retry attempts available then
30.        Retry event processing with backoff strategy
31.    else
32.        Move event to dead-letter queue or escalate issue
33.    end if
```

EVENT HANDLER

This component provides the domain specific logic to handle events. This is where different teams can define the behavior they need when an event is consumed, like processing orders, updating inventory, or triggering workflows.

Algorithm 2: EventHandler Implementation (Concrete Class)
Input: event (the incoming event to be processed)
Output: Processed event or error handling

```
Class ConcreteEventHandler extends EventHandler:
  // Implement abstract method from EventHandler
  Function handleEvent(event):
    eventType ← event.getType()  // Retrieve the type of event

    switch eventType do:
      case "OrderEvent":
        orderID ← event.getOrderID()
        customerDetails ← event.getCustomerDetails()
        Call processOrder(orderID, customerDetails)
        break

      case "InventoryUpdateEvent":
        itemID ← event.getItemID()
        newStockLevel ← event.getNewStockLevel()
        Call updateInventory(itemID, newStockLevel)
        break

      case "WorkflowTriggerEvent":
        workflowID ← event.getWorkflowID()
        Call triggerWorkflow(workflowID)
        break

      default:
        Log "Unknown event type: " + eventType
        throw Exception("Unrecognized event type")
        break
    end switch
  Function processOrder(orderID, customerDetails):
    // Logic to process an order
    Validate customer details
    Initiate payment processing
    Update order status to "Processed"
    Log successful order processing
  Function updateInventory(itemID, newStockLevel):
    // Logic to update inventory
    Fetch current inventory level for itemID
    Set new inventory level to newStockLevel
    Log inventory update
  Function triggerWorkflow(workflowID):
    // Logic to trigger workflow
    Retrieve workflow definition by workflowID
    Initiate workflow execution
    Log workflow initiation
```

EVENT TRACKING SERVICE

The Event Tracking Service is responsible for collecting and logging metrics from the event handler. It provides observability into how the event processing system is functioning. Metrics include:

- **Missing Events:** Detect if any events are lost or unprocessed.
- **Processing Failures:** Log when events fail to process and track their frequency.
- **Retries**: Keep track of retry attempts for failed events.

The collected metrics can be used to trigger alarms and take proactive measures, ensuring system reliability.

RESULT

This section discusses the qualitative impact and performance metrics derived from the real-world deployment of the event handler system in a high-traffic production environment.
The system offered several qualitative benefits. The modular design of the event handler enabled faster integration of new event types across different teams. The abstract class design allowed for consistent logging, retry mechanisms, and metric collection across the organization, reducing development overhead and improving operational efficiency.

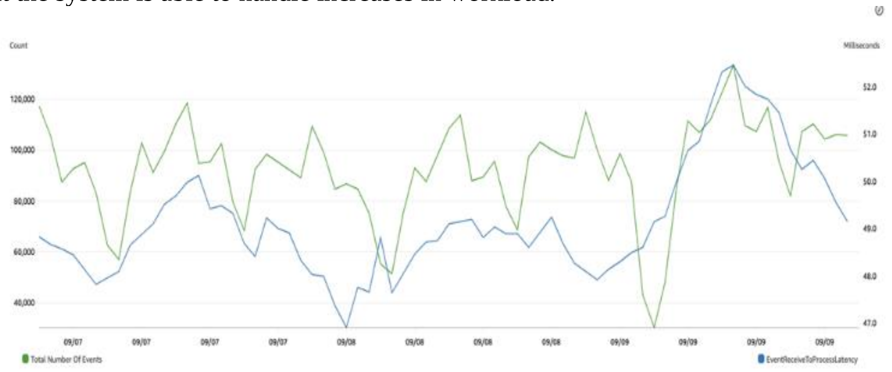
Table 1: Qualitative Benefits of Event Handler Platform

Benefit	Description
Improved Developer Experience	Simplified event handler integration across teams, reducing development time.
Enhanced Observability	Centralized logging made it easier to trace events and identify issues.
Reduced Failures	Retry mechanisms reduced event failures by handling transient issues.
Scalability	The system scaled linearly with increased event volumes, supporting real-time processing.
Easier Maintenance	The modular architecture allowed for easier system updates and maintenance.

Some other noticeable improvements are mentioned below.

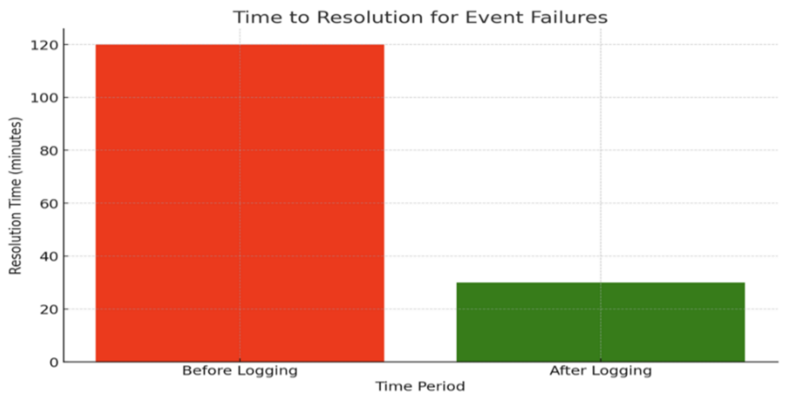
Performance Analysis

The event handler system was tested under varying loads to evaluate its scalability. The platform exhibited a consistent latency as the number of consumers increased and demonstrated high throughput. As shown in the graph below, the EventReceiveToProcessLatency keeps in line with the total number of events without any errors which demonstrates that the system is able to handle increases in workload.



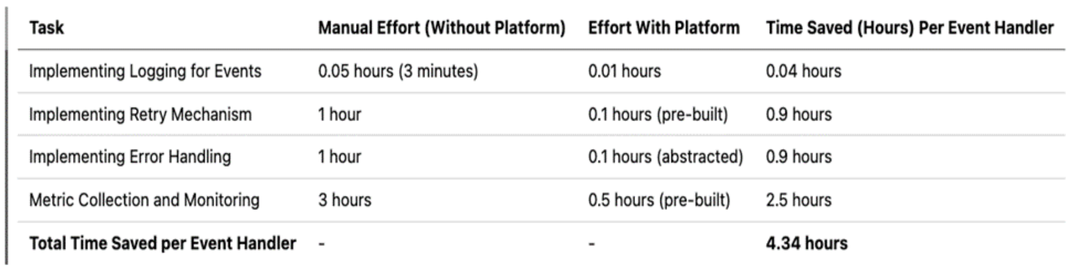
Enhanced Error Resolution Time

The below graph is compiled from issue resolution times of issues caused by supply chain risk events. As shown in the graph, issue resolution took an average of 120 minutes prior to having the centralized logging system which reduced to an average of 22.5 minutes. Which is a 81.25% reduction in resolution time.



Saved Development Time

The table below lists out the task time that was saved due to the implementation of the platform. It saves 4.34 hours of developer time per event handler, and large scale supply chain systems can typically onboard several hundred events in a given quarter which would amount to almost 500 hours of saved developer time. The average hourly salary of a Software Developer is $66.4 according to the Bureau of Labor Statistics [8], which amounts to a total of $33,000 of development time per quarter.



Task	Manual Effort (Without Platform)	Effort With Platform	Time Saved (Hours) Per Event Handler
Implementing Logging for Events	0.05 hours (3 minutes)	0.01 hours	0.04 hours
Implementing Retry Mechanism	1 hour	0.1 hours (pre-built)	0.9 hours
Implementing Error Handling	1 hour	0.1 hours (abstracted)	0.9 hours
Metric Collection and Monitoring	3 hours	0.5 hours (pre-built)	2.5 hours
Total Time Saved per Event Handler	-	-	**4.34 hours**

CONCLUSION

In this paper we discussed the architecture and core components of an event handler platform that can be shared across multiple teams working on the same supply chain system. This system was implemented in high traffic production eCommerce environments and the results were gauged in terms of qualitative benefits like enhanced monitoring, time to root cause analysis and quantitative benefits like saved development time and proactive responses to failures thereby giving way to operational excellence.

REFERENCES

[1]. Gastón Márquez, Mónica M. Villegas, and Hernán Astudillo. 2018. A pattern language for scalable microservices-based systems. In Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings (ECSA '18). Association for Computing Machinery, New York, NY, USA, Article 24, 1–7. https://doi.org/10.1145/3241403.3241429

[2]. Baškarada, S., Nguyen, V., & Koronios, A. (2018). Architecting Microservices: Practical Opportunities and Challenges. Journal of Computer Information Systems, 60(5), 428–436. https://doi.org/10.1080/08874417.2018.1520056

[3]. Rong Liu, Akhil Kumar, Wil van der Aalst,A formal modeling approach for supply chain event management, Decision Support Systems,Volume 43, Issue 3, 2007, https://doi.org/10.1016/j.dss.2006.12.009

[4]. Dani, S. (2009). Predicting and Managing Supply Chain Risks. In: Zsidisin, G.A., Ritchie, B. (eds) Supply Chain Risk. International Series in Operations Research & Management Science, vol 124. Springer, Boston, MA. https://doi.org/10.1007/978-0-387-79934-6_4

[5]. Bellemare, A., 2020. Building event-driven microservices. " O'Reilly Media, Inc."..

[6]. P. Huang, C. Guo, J. R. Lorch, L. Zhou, and Y. Dang, "Capturing and Enhancing In Situ System Observability for Failure Detection," in Proc. 13th USENIX Symp. Operating Systems Design and Implementation (OSDI), Carlsbad, CA, USA, Oct. 2018, pp. 1–16.. Available: https://www.usenix.org/conference/osdi18/presentation/huang

[7]. "Amazon SQS: Standard Queues - At-Least-Once Delivery," AWS Documentation, [Online]. Available: https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/standard-queues-at-least-once-delivery.html

[8]. U.S. Bureau of Labor Statistics, [Online]. Available: https://www.bls.gov/oes/current/oes151252.htm.