Event-Driven Architecture (EDA)

It is a software architecture pattern that revolves around the production, detection, and reaction to events within a system. An "event" represents a change of state or an occurrence that might be important to the system or business process. In this architecture, components of the system communicate by producing and reacting to events, typically asynchronously.

Example system: https://github.com/thangchung/go-coffeeshop

Salient Properties of Event-Driven Architecture:

1. **Asynchronous Communication:**
   - The system's components communicate with each other via events, and this communication doesn't require them to be tightly coupled or synchronous. Components do not wait for a response after sending an event.
2. **Decoupling:**
   - The event producer and consumer are decoupled, meaning they don't need to know about each other. They only rely on the event or the event message broker to facilitate communication.
3. **Event Stream:**
   - Events are often published to a message broker (like Kafka or RabbitMQ), which acts as an intermediary to deliver the events to the appropriate consumers.
4. **Reactive Nature:**
   - Components are "reactive" because they respond to incoming events. They act only when an event occurs, rather than polling or continuously checking for updates.
5. **Scalability:**
   - Because components are decoupled and communication is asynchronous, it's easier to scale parts of the system independently based on demand.
6. **Real-time Processing:**
   - EDA often supports real-time or near-real-time processing of events, where actions can be triggered immediately as events occur.

How is Event-Driven Architecture Implemented?

Event-Driven Architecture can be implemented using various tools and techniques:

1. **Event Producers:**
   - These are components or services that generate events when certain actions or state changes occur. For example, a user registration system may generate an event when a new user is created.
2. **Event Channel:**

    - The event is published to an event channel or message broker (such as Kafka, RabbitMQ, or AWS SNS/SQS). This helps decouple event producers from consumers.
3. **Event Consumers:**
    - These are services or components that listen for and react to specific events. They process the event and take action based on the event's content.
4. **Event Processing:**
    - Event consumers may trigger specific workflows, processes, or actions based on the event, such as updating a database, notifying a user, or initiating another system call.
5. **Event Store (Optional):**
    - In some cases, events might be stored for auditing, replaying, or monitoring purposes. This is called event sourcing, which is a form of event-driven design.
6. **Event-Driven Frameworks:**
    - Frameworks such as **Apache Kafka**, **RabbitMQ**, **AWS Lambda**, **Google Cloud Pub/Sub**, or **Azure Event Grid** can be used to facilitate the event handling, message queuing, and processing.

Software Quality Attributes Helped by EDA:

1. **Scalability:**
    - Because components are decoupled and communication is asynchronous, it's easier to scale parts of the system independently. When more events are generated, the event consumers can be scaled out without needing to modify producers.
2. **Resilience:**
    - EDA improves fault tolerance. If a component fails to process an event, it can be retried or stored until the failure is resolved. Also, since producers and consumers are decoupled, a failure in one does not affect the others directly.
3. **Flexibility and Extensibility:**
    - Adding new consumers or changing existing consumers is easier since components are loosely coupled. As new event types are added, you can integrate new consumers without modifying the entire system.
4. **Real-Time Data Processing:**
    - EDA can help deliver real-time responses and updates, which is crucial for use cases like fraud detection, IoT data processing, or real-time analytics.
5. **Maintainability:**
    - Because systems are decoupled and each component focuses on a specific event-driven task, it becomes easier to maintain and extend the system over time. New features can be added by simply adding new consumers without touching other components.
6. **Separation of Concerns:**
    - By decoupling producers and consumers, the architecture encourages separation of concerns, making it easier to manage and test components independently.

When to Use Event-Driven Architecture:

1. **When You Need Asynchronous Processing:**
   - If your system needs to handle many events concurrently without blocking or waiting for responses from other parts of the system, EDA is a great fit.
2. **Real-Time or Near Real-Time Systems:**
   - Applications like fraud detection, stock trading systems, or real-time recommendation engines benefit from EDA's ability to respond immediately to changes or events.
3. **When You Have Scalable, Independent Components:**
   - If different parts of the system should scale independently, or if they can operate in isolation, EDA allows them to grow without impacting each other.
4. **When You Need to Process High Volumes of Events:**
   - Systems with high throughput demands (such as IoT platforms or user activity tracking) benefit from event-driven systems because they allow for distributed, parallel processing of events.

When NOT to Use Event-Driven Architecture:

1. **Simple Applications with Limited Events:**
   - For small, monolithic applications where there are few interactions or state changes, EDA might introduce unnecessary complexity. Traditional request-response models might be simpler.
2. **When Transactional Integrity is Critical:**
   - If your system requires strong ACID (Atomicity, Consistency, Isolation, Durability) transactions across multiple components, EDA might be challenging to implement correctly, as events may be processed out of order or result in eventual consistency.
3. **When Immediate Responses Are Not Required:**
   - If the system doesn't need to process events in real-time or near-real-time, EDA may add unnecessary complexity.
4. **When Debugging and Monitoring Are Complex:**
   - In an event-driven system, the flow of events can be difficult to trace, making debugging harder. If your system requires detailed tracing of every action and its immediate result, EDA might not be ideal.
5. **In Systems with Complex Dependencies:**
   - If there are many interdependencies between different parts of the system that need synchronous coordination, EDA might not be the right choice. An event-driven system may struggle to maintain these dependencies without complex workarounds.