**COMSATS UNIVERSITY ISLAMABAD, ABBOTTABAD CAMPUS**



**GROUP MEMBERS:**

**QURATULAIN**

**ALEENA FAROOQ**

**UM-E-AIMEN**

**(BSE-6C)**

**SOFTWARE DESIGN AND ARCHITECTURE**

**ASSIGNMENT: 02**

**ARCHITECTURE: -**

**EVEN-DRIVEN ARCHITECTURE**

**DATE:**

**June 27, 2024**

# **EVENT-DRIVEN ARCHITECTURE**

## BACKGROUND

The concept of event-driven programming dates back to the 1960s, when computer systems were designed to respond to user interactions, such as keyboard presses and mouse clicks.

In the 1980s, event-driven programming became popular with the rise of graphical user interfaces (GUIs) and the introduction of event-driven programming languages like Smalltalk.

* ***Message-Oriented Middleware (MOM):***

In the 1990s, Message-Oriented Middleware (MOM) emerged as a way to enable asynchronous communication between distributed systems.

MOM systems, such as IBM's MQSeries (now IBM MQ) and TIBCO Rendezvous, allowed applications to send and receive messages, which laid the foundation for event-driven architecture.

* ***Enterprise Integration Patterns:***

In the early 2000s, Gregor Hohpe and Bobby Woolf published "Enterprise Integration Patterns," a book that introduced a set of patterns for integrating distributed systems.

One of the patterns, the "Event-Driven Architecture" pattern, described a design approach where applications communicate with each other by publishing and subscribing to events.

* ***Microservices and Cloud Computing:***

The rise of microservices and cloud computing in the mid-2000s created a need for more scalable and flexible integration approaches.

Event-driven architecture became a popular choice for integrating microservices, as it allowed for loose coupling, scalability, and fault tolerance.

* ***Modern EDA:***

Today, event-driven architecture is a widely adopted design approach in modern software systems.

The rise of cloud-native technologies, such as serverless computing, event-driven functions, and event streaming platforms (e.g., Apache Kafka, Amazon Kinesis), has further accelerated the adoption of EDA.

Modern EDA is characterized by the use of event sourcing, event streaming, and event-driven microservices, which enable real-time data processing, analytics, and machine learning.

## INTRODUCTION:

Event-driven architecture (EDA) is a software design pattern or architectural style that focuses on the flow of events within a system or between different systems, often in the form of messages or notifications. Various components of a system communicate and react to events asynchronously, rather than through direct, synchronous method calls. Events can represent various occurrences, changes in state, or triggers within a system.

Event-driven architecture is commonly used in various applications, including:

* ***Real-time data processing***
* ***Microservices-based systems***
* ***IoT (Internet of Things) applications***
* ***Systems that require high levels of concurrency and responsiveness***

## WORKING

Event-Driven Architecture (EDA) works by facilitating communication between components of a system through events.

Here’s a detailed look at how EDA functions:

### **Core components**

* ***Events:***

An event is a**significant occurrence within the system or from an external source that triggers a specific action or response**. Events can be categorized as either external events (e.g., user input, sensor data) or internal events (e.g., system notifications, timer events).

* ***Event Producers:***

Components or services that generate events when a specific action occurs or a state change.

***Examples***: User actions (e.g., clicking a button, submitting a form), system changes (e.g., a file being uploaded), external system triggers (e.g., receiving a message from another service).

* ***Event Consumers:***

Components or services that listen for events and react to them.

***Examples:*** Services that update databases, trigger other events, send notifications, or process data.

* ***Event Channels/Brokers:***

Middleware that transports events from producers to consumers. They decouple producers from consumers, ensuring loose coupling.

***Examples:*** Message queues (e.g., RabbitMQ), event streaming platforms (e.g., Apache Kafka), pub/sub systems (e.g., AWS SNS).

### Event Flow: -

* ***Event Creation:***

An event is generated by an event producer when a significant action or change occurs.

***Example:*** A user submits an order on an e-commerce website.

* ***Event Publication:***

The event producer publishes the event to an event channel or broker.

***Example:*** The order service publishes an "OrderCreated" event to a message broker.

* ***Event Transmission:***

The event channel or broker transports the event to one or more event consumers.

***Example:*** The message broker ensures the "OrderCreated" event is available to interested consumers.

* ***Event Consumption:***

Event consumers subscribe to the event channel or broker and react to the events they receive.

***Example:*** An inventory service subscribes to the "OrderCreated" event, updates stock levels, and publishes an "InventoryUpdated" event.

* ***Event Processing:***

Consumers process the event, which can include performing business logic, updating databases, triggering further events, or sending notifications.

***Example:*** The inventory service processes the "OrderCreated" event, updates the stock levels in the database, and triggers a reorder if stock is low.

## PROS AND CONS

### PROS:

1. ***Loose Coupling****:*
   * Components interact through events rather than direct calls, which reduces dependencies between them.
   * This enhances the flexibility to modify or replace individual components without affecting others.
2. ***Scalability****:*
   * Components can scale independently based on the number of events they produce or consume.
   * Systems can handle varying loads efficiently, improving overall performance and reliability.
3. ***Real-Time Processing****:*
   * EDA allows for real-time or near-real-time processing of events.
   * Systems can respond quickly to changes, providing timely updates and actions.
4. ***Flexibility****:*
   * New event consumers can be added without altering existing event producers.
   * This makes it easier to extend system functionality and integrate new features.

### Cons

1. ***Complexity****:*
   * Designing and managing an event-driven system can be complex due to the asynchronous nature of event handling.
   * This can increase development time and require a higher level of expertise.
2. ***Debugging and Monitoring****:*
   * Asynchronous events make it harder to trace and debug issues, as the flow of events may not be straightforward.
   * Effective monitoring and logging become crucial but can be difficult to implement.
3. ***Event Storming****:*
   * High volumes of events can overwhelm the system if not managed properly.
   * This requires robust event handling and processing infrastructure to avoid bottlenecks.
4. ***Data Consistency****:*
   * Ensuring data consistency across distributed components can be difficult.
   * Techniques like eventual consistency may need to be employed, which can complicate the system.

## EXAMPLE

### **LinkedIn's Real-Time Analytics Platform**

LinkedIn, the professional networking platform, uses an event-driven architecture to power its real-time analytics system.

### Reasons for Success

1. ***Scalability****:*
   * LinkedIn's real-time analytics system processes billions of events per day. This massive scale is achievable due to the loosely coupled nature of event-driven systems.
   * The architecture allows LinkedIn to scale its infrastructure horizontally, adding more processing nodes to handle increased load without significant changes to the overall system.
2. ***Real-Time Processing****:*
   * Using Apache Kafka as the event streaming platform, LinkedIn can process events in real-time. Events generated by user actions, such as profile views, connection requests, and job applications, are immediately available for analysis.
   * This real-time capability enables LinkedIn to provide timely and relevant insights to users, such as real-time profile views and updates on network activities.
3. ***Flexibility and Extensibility****:*
   * The architecture allows LinkedIn to easily add new event consumers. For instance, a new analytics service can be added to the system by simply subscribing to the relevant event topics in Kafka.
   * This flexibility means LinkedIn can quickly adapt to new business requirements and innovate without disrupting existing services.
4. ***Decoupling of Components****:*
   * Each service within the analytics platform operates independently, publishing and consuming events without direct dependencies on other services.
   * This decoupling enhances the resilience of the system, as the failure of one service does not directly impact others. It also simplifies maintenance and allows for independent updates and scaling of services.
5. ***Resilience and Fault Tolerance****:*
   * Kafka's distributed nature and robust replication mechanisms ensure high availability and fault tolerance. Events are replicated across multiple nodes, and if one node fails, others can take over without data loss.
   * LinkedIn's analytics platform remains reliable and operational even in the face of hardware failures or network issues, ensuring continuous data processing and analytics.

## HOW EDA IS BETTER THAN OTHERS?

### COMPARISON:

| **Aspect** | **Monolithic Architecture** | **Layered Architecture** | **Microservices Architecture** | **Event-Driven Architecture (EDA)** |
| --- | --- | --- | --- | --- |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Structure** | Single, unified codebase | Organized into layers (presentation, business, data) | Small, independently deployable services | Components communicate asynchronously via events |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Coupling** | Tight coupling and interdependencies | Clear separation of concerns but can be tightly coupled between layers | Loose coupling but can be synchronous or asynchronous | Loose coupling, reducing dependencies |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Scalability** | Entire application must be scaled | Layers can be scaled but often not independently | Independent scaling of services | Independent scaling of components |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Flexibility** | Difficult to modify or extend | Separation of concerns improves flexibility but still interconnected | High flexibility, easier to modify and extend services | High flexibility, easy to add new components |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Resilience** | Failure in one part can impact the entire application | Failure in one layer can affect others | High resilience, services are independent | High resilience, failure in one component does not directly impact others |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Real-Time Processing** | Limited real-time capabilities | Generally synchronous, less suited for real-time processing | Can support real-time, depends on implementation | Ideal for real-time or near-real-time applications |

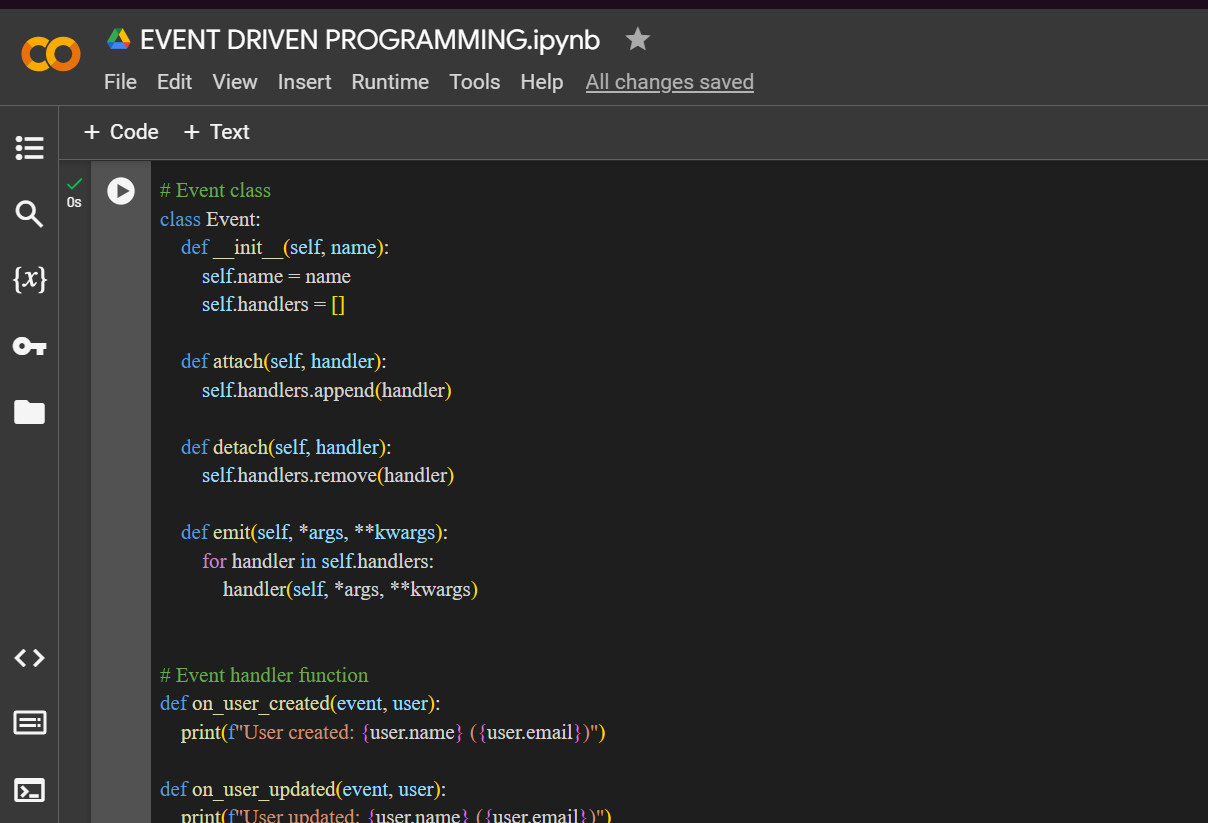
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Complexity** | Simpler to develop initially but complex to maintain | Moderate complexity, clear separation helps but inter-layer dependencies | Moderate complexity, easier to manage and extend | High complexity in design and management, but easier to scale and extend |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Debugging** | Easier to debug in a single codebase | Easier to debug within a layer, but inter-layer issues can be tricky | Easier to debug individual services, but tracing issues can be complex | Harder to debug due to asynchronous nature |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Data Consistency** | Easier to ensure consistency within a single database | Consistency within layers, challenging across layers | Ensuring consistency can be complex, especially in distributed systems | Ensuring consistency can be challenging, often uses eventual consistency |

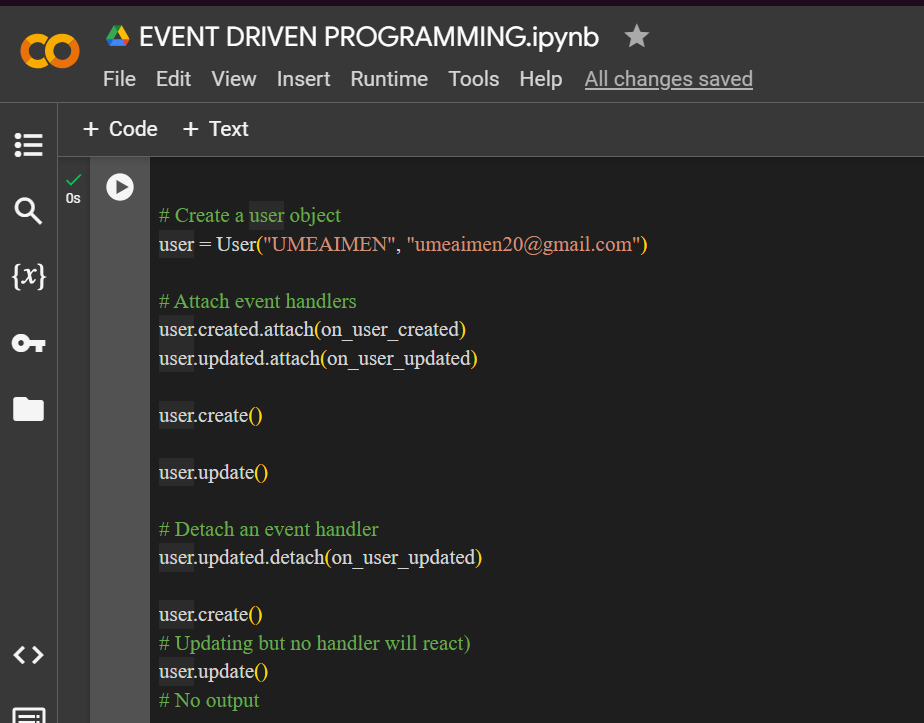
## Code Example:

Here’s a python code’s example to explain the basic implementation of Event Driven Architecture:



A screenshot of a computer program

Description automatically generated



OUTPUT:

A screenshot of a computer

Description automatically generated

## CONCLUSION:

In summary, Event-Driven Architecture is chosen for its scalability, flexibility, real-time capabilities, resilience, support for complex workflows, and alignment with modern technology trends. These factors collectively enable organizations to build robust, responsive, and adaptable systems that meet the demands of today's dynamic business environments.