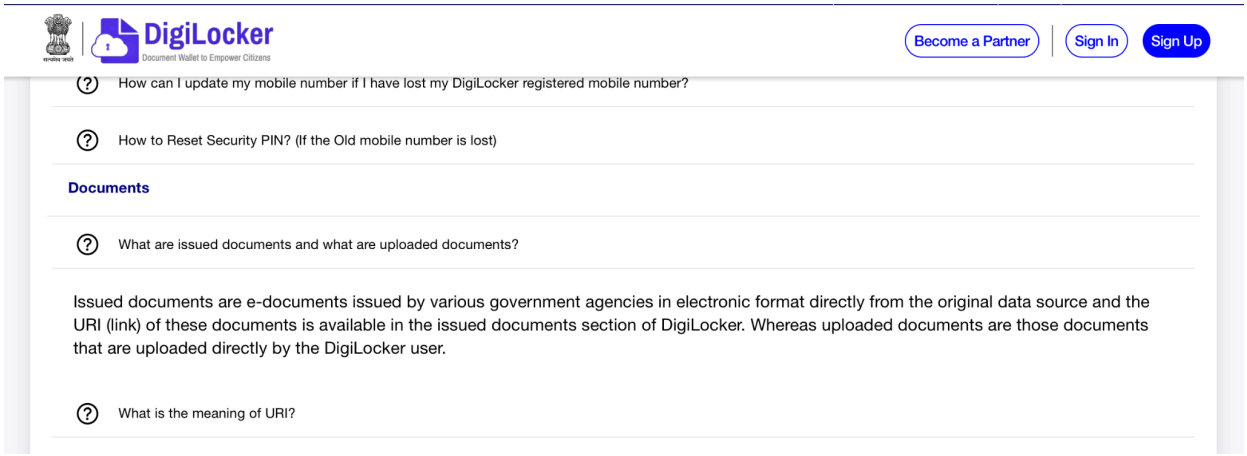
Demonstrating Mediator Topology in Event-Driven Architecture using Apache Kafka for Document Verification

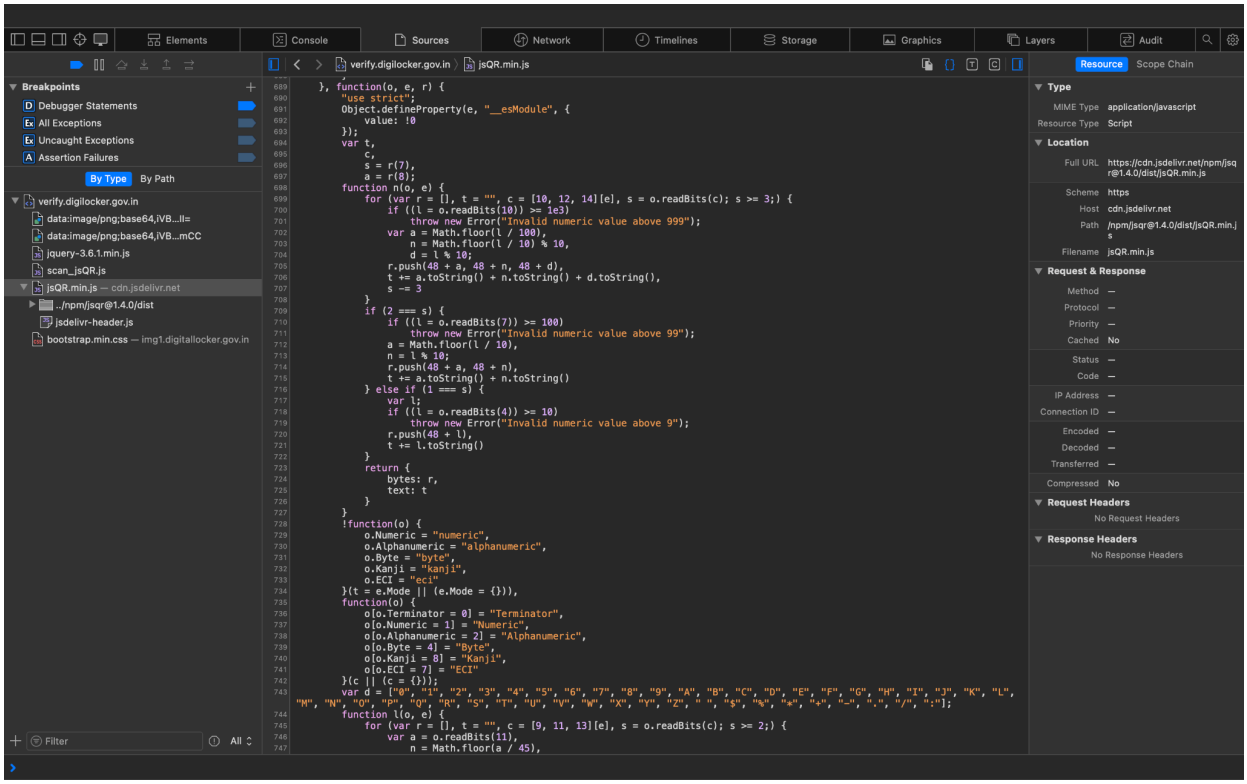
(T1-24-25-CSE 752-Software Architecture and Design Practices)

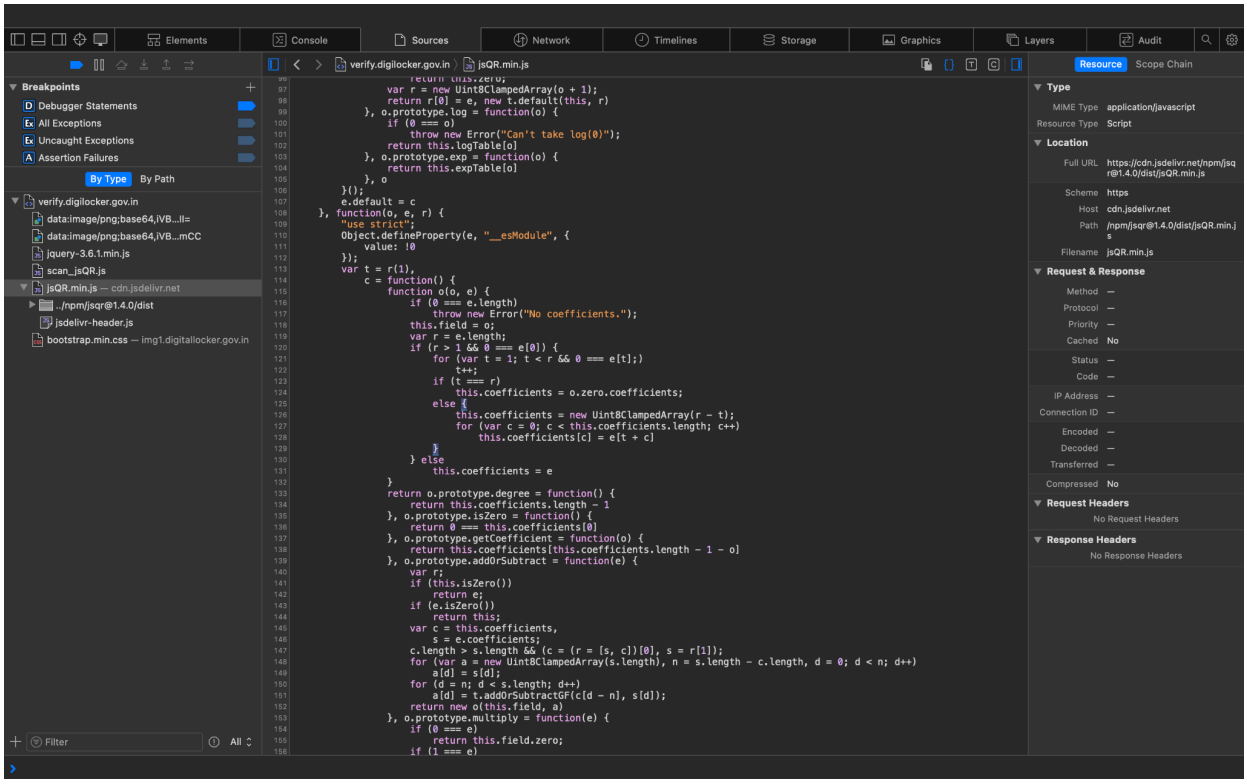
1. Introduction

In this project, we demonstrate an Event-Driven Architecture (EDA) with a focus on the **Mediator Topology**, implemented using **Apache Kafka**. The system revolves around document verification, where documents are processed based on whether they belong to **Education** or **Employment** categories. The architecture consists of multiple components, with each responsible for a specific task in the document verification process. This system processes documents using Kafka topics and directs data between processors (mediator) based on the document type (education or employment).



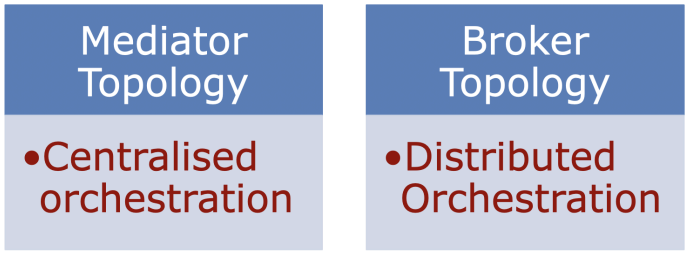
Paperless Verification: Secure, online document verification eliminates the need for physical submissions, enhancing efficiency.


Elements
Console
Sources
Network
Timelines
Storage
Graphics
Layers
Audit
verify.digilocker.gov.in
jsQR.min.js
Breakpoints
Debugger Statements
All Exceptions
Uncaught Exceptions
Assertion Failures
By Type
By Path
verify.digilocker.gov.in
...lI=
...mCC
jquery-3.6.1.min.js
scan_jsQR.js
jsQR.min.js — cdn.jsdelivr.net
../npm/jsqr@1.4.0/dist
jsdelivr-header.js
bootstrap.min.css — img1.digitallocker.gov.in
Filter
All
Resource
Scope Chain
Type
MIME Type application/javascript
Resource Type Script
Location
Full URL https://cdn.jsdelivr.net/npm/jsqr@1.4.0/dist/jsQR.min.js
Scheme https
Host cdn.jsdelivr.net
Path /npm/jsqr@1.4.0/dist/jsQR.min.js
Filename jsQR.min.js
Request & Response
Method —
Protocol —
Priority —
Cached No
Status —
Code —
IP Address —
Connection ID —
Encoded —
Decoded —
Transferred —
Compressed No
Request Headers
No Request Headers
Response Headers
No Response Headers


Elements
Console
Sources
Network
Timelines
Storage
Graphics
Layers
Audit
verify.digilocker.gov.in
jsQR.min.js
Breakpoints
Debugger Statements
All Exceptions
Uncaught Exceptions
Assertion Failures
By Type
By Path
verify.digilocker.gov.in
...lI=
...mCC
jquery-3.6.1.min.js
scan_jsQR.js
jsQR.min.js — cdn.jsdelivr.net
../npm/jsqr@1.4.0/dist
jsdelivr-header.js
bootstrap.min.css — img1.digitallocker.gov.in
Filter
All
Resource
Scope Chain
Type
MIME Type application/javascript
Resource Type Script
Location
Full URL https://cdn.jsdelivr.net/npm/jsqr@1.4.0/dist/jsQR.min.js
Scheme https
Host cdn.jsdelivr.net
Path /npm/jsqr@1.4.0/dist/jsQR.min.js
Filename jsQR.min.js
Request & Response
Method —
Protocol —
Priority —
Cached No
Status —
Code —
IP Address —
Connection ID —
Encoded —
Decoded —
Transferred —
Compressed No
Request Headers
No Request Headers
Response Headers
No Response Headers

Key Concepts:

- **Event-Driven Architecture (EDA)**: EDA is a design pattern where the communication between components happens through events. Components in an event-driven system do not communicate with each other directly. Instead, they produce and consume events that represent state changes or significant occurrences within the system. There are 2 types of event handling topologies, in this project we will implement Mediator topology.

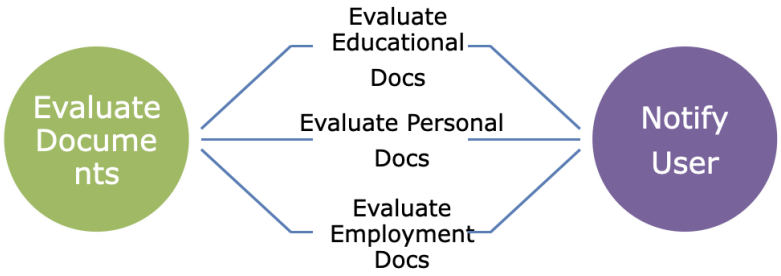


- **Mediator Topology**: In this topology, a mediator component directs messages between different producers and consumers. The mediator helps decouple components by ensuring that producers do not directly send messages to consumers. It enables routing, filtering, and coordination of messages in the system.
- **Kafka Topics**: Kafka is used for message streaming and event handling. Topics in Kafka serve as the communication channels through which events are sent and received.

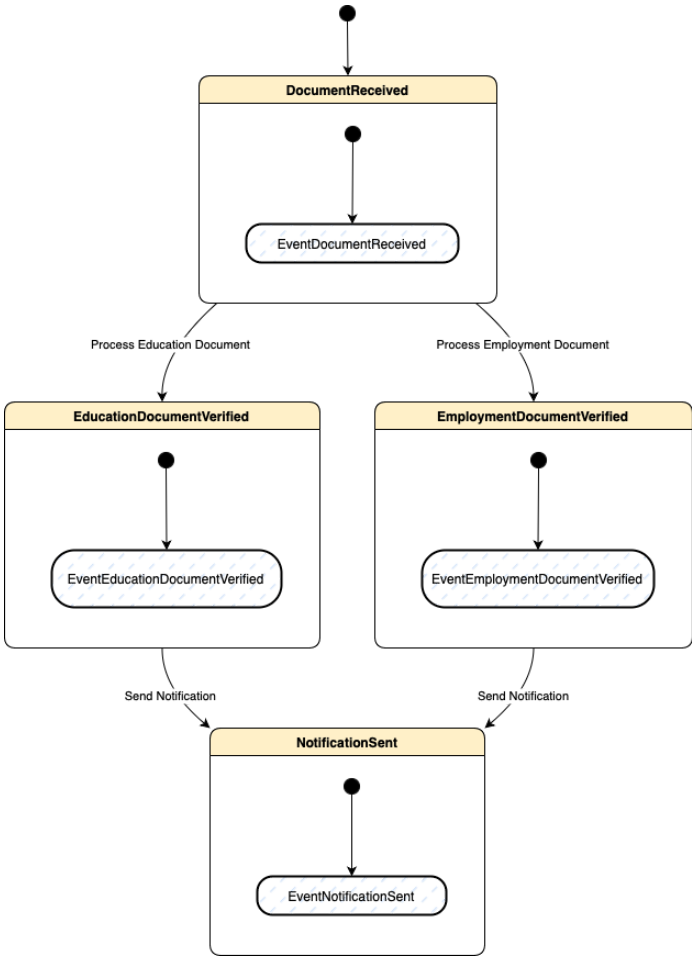
Project Overview:

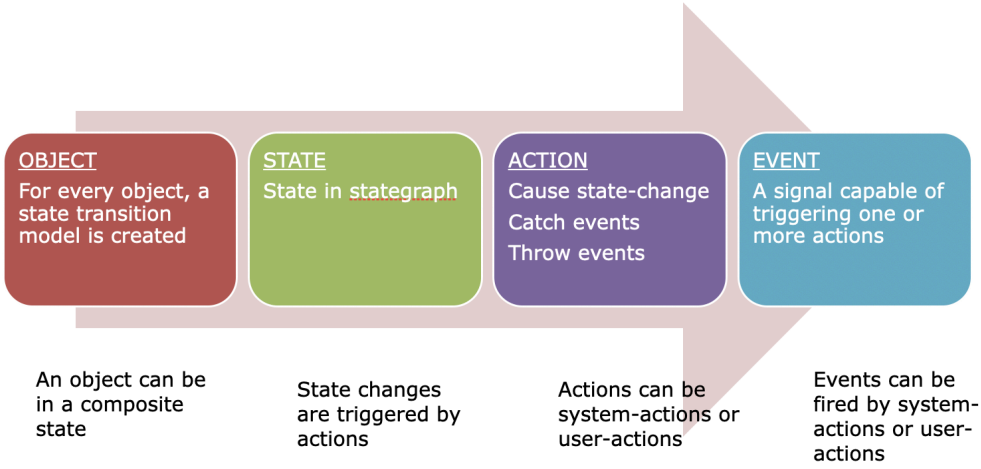
The project focuses on verifying two types of documents:

1. **Education documents**: If the document ID is numeric, it's classified as an education document.
2. **Employment documents**: If the document ID is alphanumeric, it's classified as an employment document.



Based on this classification, the **Mediator** routes the events to the appropriate processor (either **Educational Processor** or **Employment Processor**). Each processor checks the document ID's validity (whether it is 4 digits long), and then sends a status update to a **Consumer Notification** component that displays the result.





1. States:

States represent the current conditions or status of the system or a particular component at any given time. In our project, we can identify the following states:

- **Document Received**: When a document is first received in the system, it's placed in the `event_queue`. This is the initial state before any processing.
- **Education Document Verified**: After the document is routed to the `educational_documents` processor and validated, the state is updated to reflect whether it is valid or invalid based on the document ID.
- **Employment Document Verified**: After the document is routed to the `employment_documents` processor and validated, the state is updated similarly.
- **Notification Sent**: After the verification process is completed, the notification with the verification result is sent to the consumer, indicating the final state.

2. Actions:

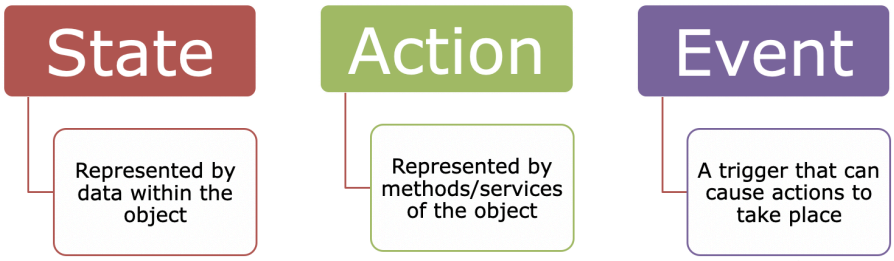
Actions are the tasks or business logic that are executed in response to an event. Actions handle the events and process the business rules but do not trigger other actions directly. In our project, the actions are:

- **Process Document**: This action handles the business logic of checking the document ID, validating it, and determining if the document is valid or invalid. This is done by the processors for both education and employment documents.
- **Send Notification**: This action handles sending the final verification status (valid or invalid) to the consumer notification system.

3. Events:

Events represent significant occurrences or changes in the system. They trigger actions, which in turn may trigger other events. Events can be thought of as signals that inform other components about something that needs to be done. In this project, the events are:

- **Document Received**: The event that occurs when the main producer sends a document to the `event_queue` Kafka topic. This event triggers the mediator action.
- **Document Routed to Education Processor**: The mediator routes the document to the `educational_documents`topic based on the document type, triggering the educational processor action.
- **Document Routed to Employment Processor**: The mediator routes the document to the `employment_documents`topic based on the document type, triggering the employment processor action.
- **Verification Completed**: After the document is processed and verified by the relevant processor, a final event is triggered to indicate that the verification has been completed. This event triggers the notification action.



TRIGGER (ACTIONS)	OBJECT STATE	EVENT	REACTION (ACTIONS)
Document is added to the event_queue	Document Received	Document Received	produce_event(name, document_id)
Mediator routes document to a topic.	Document Routed	Document Routed to Education Processor	producer.produce(`educational_documents')
Mediator routes document to a topic.	Document Routed	Document Routed to Employment Processor	producer.produce(`employment_documents')
Processor validates the document ID.	Document Verified	Verification Completed	validate_document(message)
Verification result is published.	Notification Sent	Verification Completed	consume_verification_status()

Tools & Technologies Used:

- **Apache Kafka**: Used for the event streaming platform to handle communication between components.
- **SQLite**: Used to store processed documents with their respective status.
- **Python**: Used for writing the main logic for event production, consumption, and routing.

2. Theory Behind Event-Driven Architecture

Event-Driven Architecture (EDA)

EDA is a software architecture pattern that promotes the flow of events to communicate between decoupled components. An event in this context is any significant change in the state of the system, such as the arrival of a document, the completion of a verification, or any other business logic-triggered event.

Key Characteristics:

1. **Loose Coupling**: Components do not directly depend on each other. Instead, they communicate through events.
2. **Asynchronous**: Events are handled asynchronously, allowing the system to scale and remain responsive under load.
3. **Event Brokers**: Centralized event brokers, like Kafka, facilitate the routing of events from producers to consumers.
4. **Scalability**: EDA systems are highly scalable because events can be consumed by multiple consumers and producers can push events without waiting for immediate acknowledgment.

Components in EDA:

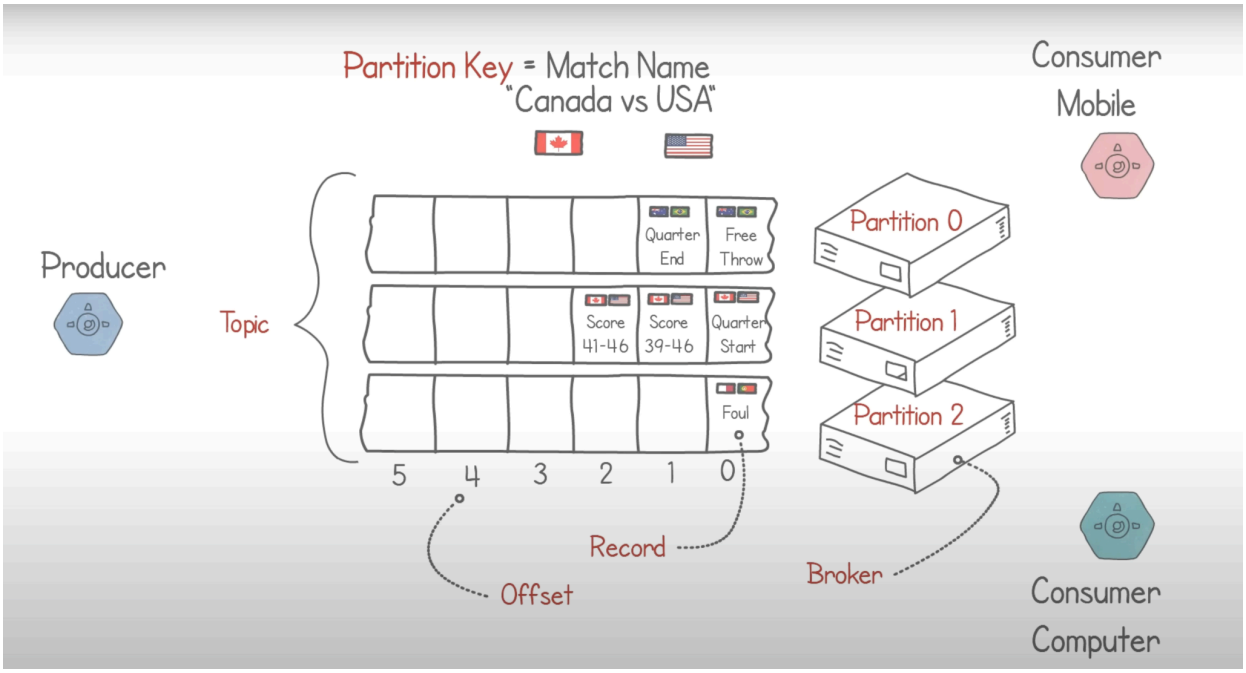
- **Event Producers**: Components that generate and emit events.
- **Event Consumers**: Components that consume events and process them.
- **Event Broker**: A messaging system (Kafka in this case) that acts as the intermediary between producers and consumers.
- **Mediator**: In the mediator topology, the mediator component ensures proper routing of events between producers and consumers.

Advantages of EDA:

- **Loose coupling**: No direct dependency between components.
- **Scalability and Fault Tolerance**: Components can scale independently, and the system remains robust in case of failures.
- **Flexibility**: Easier to add new components to the system as they can simply produce or consume events.

3. Implementation Details

The system simulates document verification using Kafka for event-based communication. Below is a detailed explanation of how each component works and how the system was implemented.



3.1 Kafka Topics Setup

Run the following command to extract the contents of the tar archive:

```
tar -xvf kafka_2.13-3.8.0.tar
```

After extraction, a directory named `kafka_2.13-3.8.0` should appear. Change into that directory:

```
cd kafka_2.13-3.8.0
```

Kafka requires Zookeeper to manage distributed coordination. Start the Zookeeper service:

```
bin/zookeeper-server-start.sh config/zookeeper.properties
```

In a new terminal window or tab (keep Zookeeper running), start the Kafka broker:

```
bin/kafka-server-start.sh config/server.properties
```

Now we will create the following Kafka topics:

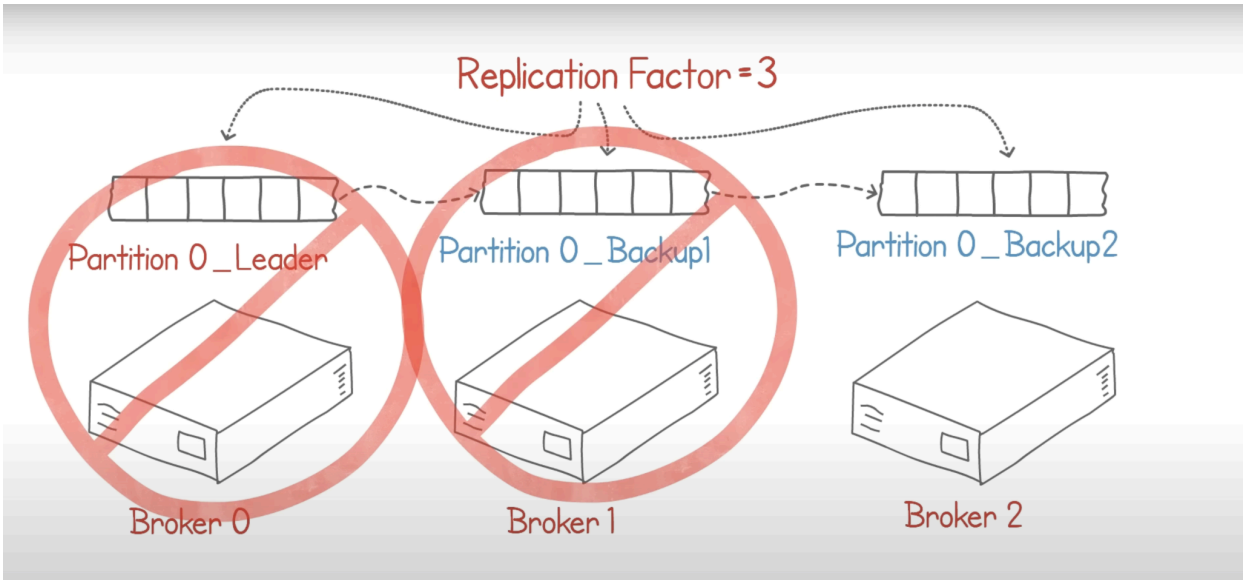
- **event_queue**: The main queue where events are pushed. Producers publish events here, such as new document arrival.
- **educational_documents**: This topic is used for events related to education documents.

- **employment_documents**: This topic is used for events related to employment documents.
- **verification_completed**: A topic for sending verification results (valid/invalid status).

To create these topics, we used Kafka's command-line tool:

QueueList
event_queue
educational_documents
employment_documents
verification_completed

```
kafka-topics.sh --create --topic event_queue --bootstrap-server
localhost:9092 --partitions 1 --replication-factor 1
kafka-topics.sh --create --topic educational_documents
--bootstrap-server localhost:9092 --partitions 1 --replication-factor
1
kafka-topics.sh --create --topic employment_documents
--bootstrap-server localhost:9092 --partitions 1 --replication-factor
1
kafka-topics.sh --create --topic verification_completed
--bootstrap-server localhost:9092 --partitions 1 --replication-factor
1
```



Verify the topic was created:

`bin/kafka-topics.sh --list --bootstrap-server localhost:9092`

```
(venv) nisharathod@Nishas-MacBook-Air kafka_2.13-3.8.0 % bin/kafka-topics.sh --l
ist --bootstrap-server localhost:9092

__consumer_offsets
consumer_channel
document_channel
educational_channel
educational_documents
employment_channel
employment_documents
event_queue
mediator_channel
status_channel
verification_completed
verification_results
```

3.2 Document Verification Logic in Python

The main logic is implemented in Python using the `kafka-python` library to produce and consume events. The flow of the system is as follows:

```
.
└── document-verification/
    ├── main.py
    ├── mediator.py
    ├── educational_processor.py
    ├── employment_processor.py
    ├── consumer_notification.py
    ├── setup_database.py
    └── documents.db
```

1. **Main Producer (`main.py`)**:
    - The main producer accepts inputs for `name` and `document_id` from the user. Based on the `document_id`, it determines whether the document is for education or employment.
    - The producer sends the document details to the `event_queue` Kafka topic.

```
(venv) nisharathod@Nishas-MacBook-Air document-verification % python3 main.py
Enter the name: Nisha
Enter the document_id: 1234
Produced: {'name': 'Nisha', 'document_id': '1234', 'document_type': 'education'}
(venv) nisharathod@Nishas-MacBook-Air document-verification % python3 main.py
Enter the name: John
Enter the document_id: abcs
Produced: {'name': 'John', 'document_id': 'abcs', 'document_type': 'employment'}
```

2. **Mediator (`mediator.py`)**:
    - The mediator listens to the `event_queue` for new events.
    - It checks the document type based on the `document_id`. If it is numeric, it directs the event to the `educational_documents` topic. If it is alphanumeric, it directs the event to the `employment_documents` topic.

```
(venv) nisharathod@Nishas-MacBook-Air document-verification % python3 mediator.p
y

Mediator is listening for events...
Consumed: {'name': 'Nisha', 'document_id': '1234', 'document_type': 'education'}
Consumed: {'name': 'John', 'document_id': 'abcs', 'document_type': 'employment'}
```

3. **Processors**:
    - **Educational Processor (`educational_processor.py`)**: This processor checks if the `document_id` is 4 digits long. If valid, it updates the status to "Valid"; otherwise, it sets it to "Invalid".
    - **Employment Processor (`employment_processor.py`)**: Similar to the educational processor, this processor checks if the `document_id` is 4 digits long for employment documents.

```
(venv) nisharathod@Nishas-MacBook-Air document-verification % python3 educational_processor.py

Educational Processor is listening...
Consumed Educational Document: {'name': 'Nisha', 'document_id': '1234', 'document_type': 'education'}
Processed Educational Document: {'name': 'Nisha', 'document_id': '1234', 'document_type': 'education', 'status': 'Valid'}
```

```
cBook-Air document-(venv) nisharathod@Nishas-MacBook-Air document-verification % python3 employment_processor.py

Employment Processor is listening...
Consumed Employment Document: {'name': 'John', 'document_id': 'abcs', 'document_type': 'employment'}
Processed Employment Document: {'name': 'John', 'document_id': 'abcs', 'document_type': 'employment', 'status': 'Valid'}
```

4. **Consumer Notification (`consumer_notification.py`)**:
    - Once the processor completes the verification, it sends the result (valid or invalid) to the `verification_completed` topic.
    - The consumer listens to this topic and displays a message to the user, such as: `"Name's document_id is Valid/Invalid"`.

```
(venv) nisharathod@Nishas-MacBook-Air document-verification % python3 consumer_notification.py
Notifier is listening...
Consumed Status: {'name': 'Nisha', 'document_id': '1234', 'document_type': 'education', 'status': 'Valid'}
Nisha's 1234 is Valid
Consumed Status: {'name': 'John', 'document_id': 'abcs', 'document_type': 'employment', 'status': 'Valid'}
John's abcs is Valid
```

5. **SQLite Database**:
    - The status of each document is saved in an SQLite database, including the `name`, `document_id`, and `status`.
    - This is achieved using the `sqlite3` library in Python.

```
(venv) nisharathod@Nishas-MacBook-Air document-verification % sqlite3 documents.db

SQLite version 3.43.2 2023-10-10 13:08:14
Enter ".help" for usage hints.
sqlite> SELECT * FROM documents;
Nisha|1234|Valid
John|abcs|Valid
sqlite>
```

4. Conclusion

In this project, we successfully demonstrated an event-driven architecture using Kafka, specifically the **Mediator Topology**, where events are routed and processed based on their type. The system decouples producers, mediators, and consumers, allowing flexibility, scalability, and asynchronous communication between components.

This architecture is well-suited for distributed systems where components need to work independently but in coordination with each other. The Kafka event broker ensures efficient message routing, and the use of SQLite for storing document verification status provides an easy way to track the status of each document.