**Distributed Event Streaming Platform Components**

Welcome to Distributed Event Streaming Platform Components.

After watching this video, you will be able to:

Describe what an Event is.

List the common Event formats.

Describe what an Event Stream Platform (ESP) is.

List the main components of an ESP.

And, list the popular ESPs.

An event normally means something worth noticing is happening.

In the context of event streaming, an event is a type of data which describes the entity’s

observable state updates over time.

For example,

The GPS coordinates of a moving car.

The temperature of a room.

Blood pressure measurements of a patient,

or a RAM usage of a running application.

An Event, as a special type for data, has different formats.

Let’s have a look at the three most common formats:

It can be as a primitive type such as a plain text, number, or date,

or an event may be in key-value format, and its value can be a primitive data type, or

complex data type like list, tuple, JSON, XML, or even bytes.

For example, the coordinates tuple of a car with car\_id\_1.

Also, very often, an event can be associated with a timestamp to make it time-sensitive.

Next, we will have a closer look at event streaming.

Suppose we have one event source such as a group of sensors, a monitoring device, a database,

or a running application.

This event source may continuously generate a large event volume at a short time interval

or nearly real-time.

Those real-time events need to be properly transported to an event destination such as

a file system, another external database, or an application.

The continuous event transportation between an event source and an event destination is

called event streaming.

After all you have learned about ETL so far, you may think that to implement such an ETL

process between one event source to one event destination should be straightforward.

However, what if we have multiple different event sources and destinations?

In a real-world scenario, event streaming can be really complicated with multiple distributed

event sources and destinations, as data transfer pipelines may be based on different communication

protocols such as:

FTP: File Transfer Protocol

HTTP: Hypertext Transfer Protocol

JDBC: Java Database Connectivity

SCP: Secure copy.

And so on.

An event destination can also be an event source simultaneously.

For example, one application could receive an event stream and process it, then transport

the processed results as an event stream to another destination.

To overcome such a challenge of handling different event sources and destinations, we will need

to employ the Event Stream Platform, or ESP.

An ESP acts as a middle layer among various event sources and destinations and provides

a unified interface for handling event-based ETL.

As such, all event sources only need to send events to an ESP instead of sending them to

the individual event destination.

On the other side, event destinations only need to subscribe to an ESP,

and just consume the events sent from the ESP instead of the individual event source.

Different ESPs may have different architectures and components.

Here we show you some common components included in most ESP systems.

The first and foremost component is the Event broker, which is designed to receive and consume

events.

Since it is the core component of an ESP, we will explain it in more detail in the next

slide.

The second common component of an ESP is Event Storage, which is used for storing events

being received from event sources.

Accordingly, event destinations do not need to synchronize with event sources, and stored

events can be retrieved at will.

The third common component is the Analytic and Query Engine which is used for querying

and analyzing the stored events.

Let’s have a look at the Event broker, which is the core component of an ESP.

It normally contains three sub-components: ingester, processor, and consumption.

The ingester is designed to efficiently receive events from various event sources.

The processor performs operations on data such as serializing and deserializing; compressing

and decompressing; encryption and decryption; and so on.

The consumption component retrieves events from event storage and efficiently distributes

them to subscribed event destinations.

There are many ESP solutions, including:

Apache Kafka

Amazon Kinesis

Apache Flink

Apache Spark

Apache Storm

and more such as Logstash in Elastic Stack, and so on.

Each has its unique features and application scenarios.

Among these ESPs, Apache Kafka is probably THE most popular one.

In this video, you learned that:

An event stream represents entities’ status updates over time.

Common event formats include primitive data types, key-value, and key-value with a timestamp.

An ESP is needed especially when there are multiple event sources and destinations.

The main components of an ESP are Event broker, Event storage, Analytic and Query Engine.

Apache Kafka is the most popular open source ESP.

And finally,

popular ESPs include Apache Kafka, Amazon Kinesis, Apache Flink, Apache Spark, Apache

Storm, and more.

What is an Event

A screenshot of a web page

Description automatically generated

Common Event Formats

A screenshot of a computer program

Description automatically generated

One source to one destination

A screenshot of a website

Description automatically generated

A diagram of a source

Description automatically generated with medium confidence

Event streaming

Many Sources to many destinations

Example

A diagram of a flowchart

Description automatically generated

Event Streaming Platform (ESP)

A diagram of a streaming platform

Description automatically generated

Common Coponents of an Esp

A diagram of a event

Description automatically generated with medium confidence

The first and foremost component is the Event broker, which is designed to receive and consume

events.

Since it is the core component of an ESP, we will explain it in more detail in the next

slide.

The second common component of an ESP is Event Storage, which is used for storing events

being received from event sources.

Accordingly, event destinations do not need to synchronize with event sources, and stored

events can be retrieved at will.

The third common component is the Analytic and Query Engine which is used for querying

and analyzing the stored events.

Event Broker

A diagram of a company

Description automatically generated

It normally contains three sub-components: ingester, processor, and consumption.

The ingester is designed to efficiently receive events from various event sources.

The processor performs operations on data such as serializing and deserializing; compressing

and decompressing; encryption and decryption; and so on.

The consumption component retrieves events from event storage and efficiently distributes

them to subscribed event destinations.

A screenshot of a computer

Description automatically generated

## Apache Kafka Overview

Welcome to Apache Kafka Overview. After watching this video, you will be able to: Recognize Apache Kafka as an

Event Streaming Platform (ESP). Describe the architecture of Apache Kafka. List common use cases for Apache Kafka. Summarize the main features and

benefits of Apache Kafka, and list popular Kafka-based

ESP-as-a-Service providers. By now you should be able to describe what an

Event Streaming Platform and its core components are, and may think that it must be extremely

difficult to implement an ESP from scratch. That is true, but we already

have many great open source and commercial ESP solutions available

that come with many built-in capabilities. Among these, Apache Kafka, an open source project, has probably become THE most

popular event streaming platform. Kafka is a comprehensive platform and can

be used in many application scenarios. Let's look at some common ones here: Kafka was originally used to track user

activities such as keyboard strokes, mouse clicks, page views, searches,

gestures, screen time, and so on. But now it is also suitable for all kinds

of metric-streaming such as sensor readings, GPS, as well as hardware and software monitoring. For enterprise applications and

infrastructure with a huge number of logs, Kafka can be used to collect and integrate

them into a centralized repository. For banks, insurance, or FinTech companies, Kafka

is also widely used for payments and transactions. These scenarios are just the tip of the iceberg. You basically can use Kafka in scenarios

when you want high throughput and reliable data transportation services among

various event sources and destinations. All these events will be ingested

in Kafka, and become available for subscriptions and consumption including: Further data storage and movement to other

online or offline databases and backups. Real-time processing and analytics, including dashboard, machine

learning, AI algorithms, and so on. Generating notifications such as email,

text messages, and instant messages, or data governance and auditing to

make sure sensitive data such as bank transactions are complying with regulations. Kafka has a distributed

client-server architecture: For the server side, Kafka is a

cluster with many associated servers called brokers, acting as

the event broker to receive, store, and distribute events. All those brokers

are managed by another distributed system called ZooKeeper to ensure all brokers work

in an efficient and collaborative way. Kafka uses a TCP (Transmission

Control Protocol) based network communication protocol to exchange

data between clients and servers. For the client side, Kafka provides

different types of clients, such as Kafka CLI (Command Line Interface) which is a collection of shell scripts to

communicate with the Kafka server, many high-level programming

APIs such as Java, Scala, REST APIs, and some specific third party clients

made by the Kafka community. Accordingly, you could choose different

clients based on your requirements. Now that you have a basic understanding of Kafka, let’s see why Apache Kafka

has become such a popular ESP: Kafka has the following main features: It is a distribution system which makes it highly scalable to handle high data

throughput and concurrency. A Kafka cluster normally has multiple event

brokers which can handle event streaming in parallel. As such, Kafka is

very fast and highly scalable. Kafka also divides event storage into

multiple partitions and replications which makes it fault-tolerant and highly reliable. Kafka stores the events permanently. As such, event consumption could be done whenever

suitable for consumers without a deadline. And, Kafka is open source, meaning

that you can use it for free, and even customize it based

on your specific requirements. Even though Kafka is open source

and well-documented, it is still challenging to configure and deploy a

Kafka without professional assistance. Deploying a Kafka cluster requires

extensive efforts for tuning infrastructure and consistently adjusting the configurations,

especially for enterprise-level deployments. Fortunately, several commercial service providers offer an on-demand ESP-as-a-Service

to meet your streaming requirements. Many of them are built on top of Kafka

and provide added value for customers. Some well-known ESP providers include: Confluent Cloud, which provides customers

with fully managed Kafka services, either on-premises or on-cloud. IBM Event Streams, which is also based on

Kafka and provides many add-on services such as enterprise-grade security, disaster

recovery, and 24-7 cluster monitoring. Amazon Managed Streaming for Apache Kafka, which is also a fully managed service to

facilitate the build and deployment of Kafka. In this video, you learned that: Apache Kafka is one of the

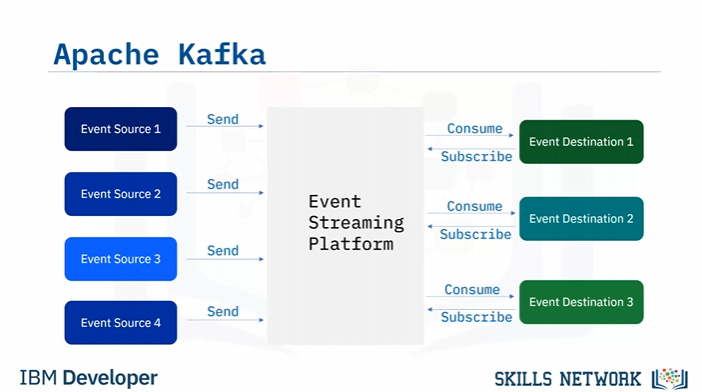
most popular open source ESPs. Common Kafka use cases include

user-activity tracking, metrics and logs integrations, and

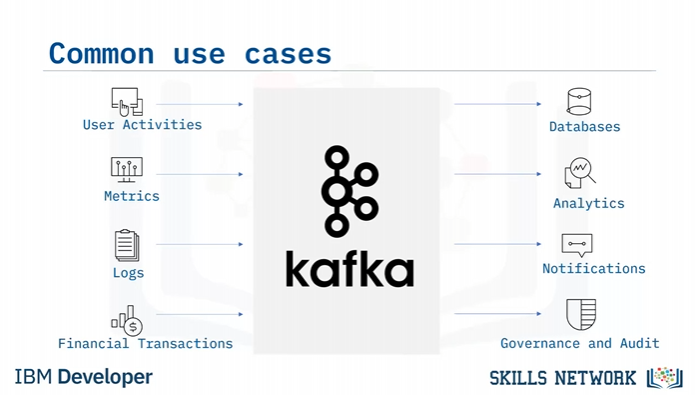
financial transaction processing. Apache Kafka is a highly scalable and reliable

platform that stores events permanently. And finally, popular Kafka-based ESP

service providers include Confluent Cloud, IBM Event Streams, and Amazon Managed Streaming.



Apache Kafka Common Use Cases



Kafka is a comprehensive platform and can be used in many application scenarios.

Let's look at some common ones here:

Kafka was originally used to track user activities such as keyboard strokes,

mouse clicks, page views, searches, gestures, screen time, and so on.

But now it is also suitable for all kinds of metric-streaming such as sensor readings,

GPS, as well as hardware and software monitoring.

For enterprise applications and infrastructure with a huge number of logs,

Kafka can be used to collect and integrate them into a centralized repository.

For banks, insurance, or FinTech companies, Kafka is also widely used for payments and transactions.

These scenarios are just the tip of the iceberg.

You basically can use Kafka in scenarios when you want high throughput and reliable

data transportation services among various event sources and destinations.

All these events will be ingested in Kafka, and become available for

subscriptions and consumption including:

Further data storage and movement to other online or offline databases and backups.

Real-time processing and analytics,

including dashboard, machine learning, AI algorithms, and so on.

Generating notifications such as email, text messages, and instant messages, or

data governance and auditing to make sure sensitive data such as

bank transactions are complying with regulations.

Kafka Architecture

A screen shot of a computer

Description automatically generated

Kafka has a distributed client-server architecture:

For the server side, Kafka is a cluster with many associated servers

called brokers, acting as the event broker to receive,

store, and distribute events. All those brokers are managed by another distributed system called

ZooKeeper to ensure all brokers work in an efficient and collaborative way.

Kafka uses a TCP (Transmission Control Protocol) based network

communication protocol to exchange data between clients and servers.

For the client side, Kafka provides different types of clients, such as

Kafka CLI (Command Line Interface) which is

a collection of shell scripts to communicate with the Kafka server,

many high-level programming APIs such as Java, Scala,

REST APIs, and

some specific third party clients made by the Kafka community.

Accordingly, you could choose different clients based on your requirements.

A close-up of a computer

Description automatically generated

Event Streaming as a Service

Hard to tune

A diagram of a service

Description automatically generated with medium confidence

## Building Event Streaming Pipelines using Kafka

Welcome to Building Event

Streaming Pipelines Using Kafka. After watching this video, you will be able to: Describe the core components of Kafka. Use Kafka to publish (write) and

subscribe to (read) streams of events. Use Kafka to consume events, either

as they occur or retrospectively, and Describe an end-to-end event

streaming pipeline example. A Kafka cluster contains one or many

brokers. You may think of a Kafka broker as a dedicated server to receive,

store, process, and distribute events. Brokers are synchronized and managed by

another dedicated server called ZooKeeper. For example, here we have A log topic and a transaction topic in broker 0. A payment topic and a GPS topic in broker 1. and, a user click topic and user

search topic in broker 2. Each broker contains one or many topics. You

can think of a topic as a database to store specific types of events, such as logs,

transactions, and metrics, for example. Brokers manage to save published

events into topics and distribute the events to subscribed consumers. Like many other distribution systems, Kafka implements the concepts of

partitioning and replicating. It uses topic partitions and replications

to increase fault-tolerance and throughput so that event publication and consumption can

be done in parallel with multiple brokers. In addition, even if some brokers

are down, Kafka clients are still able to work with the target topics

replicated in other working brokers. For example: A log topic has been separated

into two partitions: 0, 1, and a user topic has been separated

into two partitions: 0, 1. And each topic partition is duplicated into two

replications and stored in different brokers. The Kafka CLI, or command line interface

client provides a collection of powerful script files for users to build

an event streaming pipeline: The Kafka-topics script is the

one you probably will be using often to manage topics in a Kafka

cluster. It is straightforward. Let’s have a look at some common usages: This first one is to create a topic. Here we are trying to create

a topic called ‘log\_topic’ with two partitions and two replications. One important note here is that many Kafka

commands, like kaf-topics, require users to refer to a running Kafka cluster with a host and

a port, such as a localhost with the port 9092. After you have created some topics, you can check all created topics in

the cluster using the ‘list option.’ And, if you want to check more details of a topic, such as partitions and replications,

you can use the ‘describe option’. And you can delete a topic

using the ‘delete option.’ Next, you will find out more about

publishing events using Kafka producers. Features of Kafka producer: They are client applications that

publish events to topic partitions according to the same order as they are published. When publishing an event in Kafka producer, an

event can be optionally associated with a key. Events associated with the same key will

be published to the same topic partition. Events not associated with any key will be

published to topic partitions in rotation. Let’s see how you can publish events to topic

partitions using the following example: Suppose you have an event source 1,

which generates various log entries, and an event source 2, which generates

user-activity tracking records. Then, you can create a Kafka producer to

publish log records to log topic partitions, and a user producer to publish user-activity

events to user topic partitions, respectively. When you publish events in producers, you

can choose to associate events with a key, for example, an application name or a user ID. Similar to the Kafka topic CLI, Kafka provides the Kafka producer

CLI for users to manage producers. The most important aspect is starting a

producer to write or publish events to a topic: Here you start a producer and point it

to the log\_topic, then you can type some messages in the console to start publishing

events. For example, log1, log2, and log3. You can provide keys to events to make sure the events with the same key

will go to the same partition. Here you are starting a producer to user\_topic,

with the parse.key option to be true, and you also specify the

key.separator to be comma. Then you can write messages as follows:

key: ‘user1’, value: ‘login website’. Key: ‘user1’, value: ‘click the top item’. And, key: ‘user1’, value: ‘logout

website’. Accordingly, all events about user1 will be saved in the same partition

to facilitate the reading for consumers. Once events are published and

properly stored in topic partitions, you can create consumers to read them. Consumers are client applications that

can subscribe to topics and read the stored events. Then event destinations can

further read events from Kafka consumers. Consumers read data from topic partitions

in the same order as they are published. Consumers also store an offset for each

topic partition as the last read position. With the offset, consumers are

guaranteed to read events as they occur. A playback is also possible by

resetting the offset to zero. This way the consumer can read all events in

the topic partition from the beginning. In Kafka, producers and

consumers are fully decoupled. As such, producers don’t need to synchronize with

consumers, and after events are stored in topics, consumers can have independent

schedules to consume them. To read published log and

user events from topic partitions, you will need to create

log and user consumers, and make them subscribe to corresponding topics. Then Kafka will push the events

to those subscribed consumers. Then, the consumers will further

send to event destinations. To start a consumer is also easy,

using the Kafka consumer script. Let’s read events from the log\_topic. You just need to run the

Kafka-console-consumer script and specify a Kafka cluster

and the topic to subscribe to. Here, you can subscribe to and read

events from the topic log\_topic. Then the started consumer will read only the new

events, starting from the last partition offset. After those events are consumed, the

partition offset for the consumer will also be updated and committed back to Kafka. Very often a user wants to read

all events from the beginning, as a playback of all historical events. To do so, you just need to add

the ‘from beginning’ option. Now, you can read all events

starting from offset 0. Let’s have a look at a more

concrete example to help you understand how to build an event

streaming pipeline end-to-end. Suppose you want to collect and analyze weather

and Twitter event streams, so that you can correlate how people talk about extreme weather

on Twitter. Here you can use two event sources: IBM Weather API to obtain real-time and

forecasted weather data in JSON format. And Twitter API to obtain real-time tweets

and mentions, also in JSON format. To receive weather and twitter JSON data

in Kafka, you then create a weather topic and a Twitter topic in a Kafka cluster,

with some partitions and replications. To publish weather and Twitter

JSON data to the two topics, you need to create a weather

producer and a Twitter producer. The event’s JSON data will be serialized

into bytes and saved in Kafka topics. To read events from the two topics,

you need to create a weather consumer and a Twitter consumer. The bytes stored in Kafka

topics will be deserialized into event JSON data. If you now want to transport the weather and Twitter event JSON data from the

consumers to a relational database, you will use a DB writer to parse those

JSON files and create database records. And then you can write those records into

a database using SQL insert statements. Finally, you can query the database

records from the relational database and visualize and analyze them in a dashboard

to complete the end-to-end pipeline. In this video, you learned that: The core components of Kafka are: Brokers: The dedicated server to receive,

store, process, and distribute events. Topics: The containers or databases of events. Partitions: Divide topics into different brokers. Replications: Duplicate

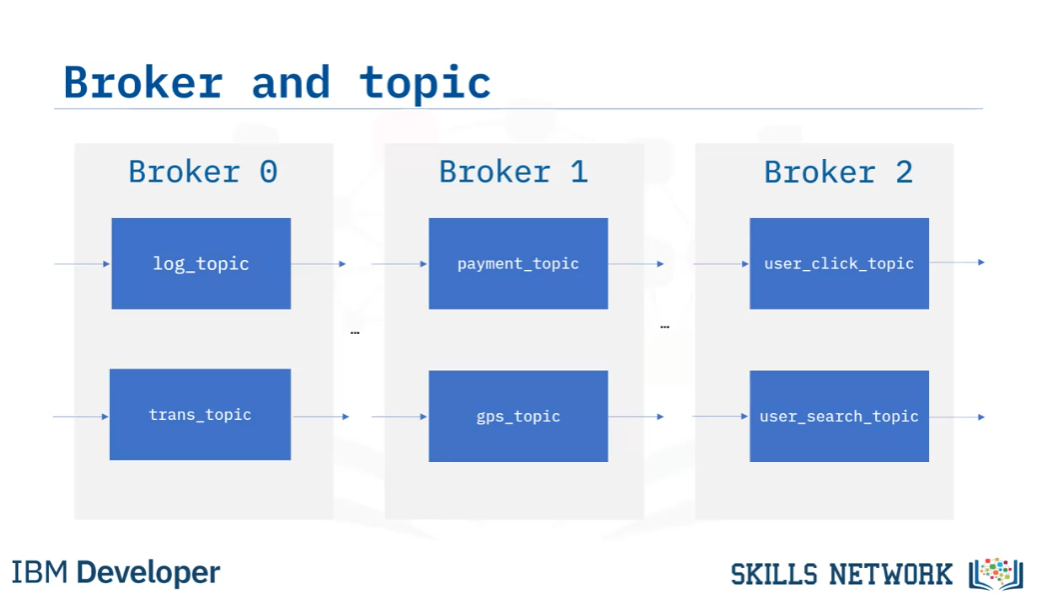
partitions into different brokers. Producers: Kafka client applications

to publish events into topics. And, consumers: Kafka client

applications subscribed to topics and read events from them. You also learned the Kafka-topics CLI manages topics. The Kafka-console-producer CLI manages producers. And finally, the Kafka-console-consumer

manages consumers.

Broker and Topic



A Kafka cluster contains one or many brokers. You may think of a Kafka

broker as a dedicated server to receive, store, process, and distribute events.

Brokers are synchronized and managed by another dedicated server called ZooKeeper.

For example, here we have

A log topic and a transaction topic in broker 0.

A payment topic and a GPS topic in broker 1.

and, a user click topic and user search topic in broker 2.

Each broker contains one or many topics. You can think of a topic as a database to store

specific types of events, such as logs, transactions, and metrics, for example.

Brokers manage to save published events into topics and distribute

the events to subscribed consumers.

Partition and Replications

A screenshot of a computer

Description automatically generated

Like many other distribution systems,

Kafka implements the concepts of partitioning and replicating.

It uses topic partitions and replications to increase fault-tolerance and throughput

so that event publication and consumption can be done in parallel with multiple brokers.

In addition, even if some brokers are down, Kafka clients are

still able to work with the target topics replicated in other working brokers.

For example:

A log topic has been separated into two partitions: 0, 1, and

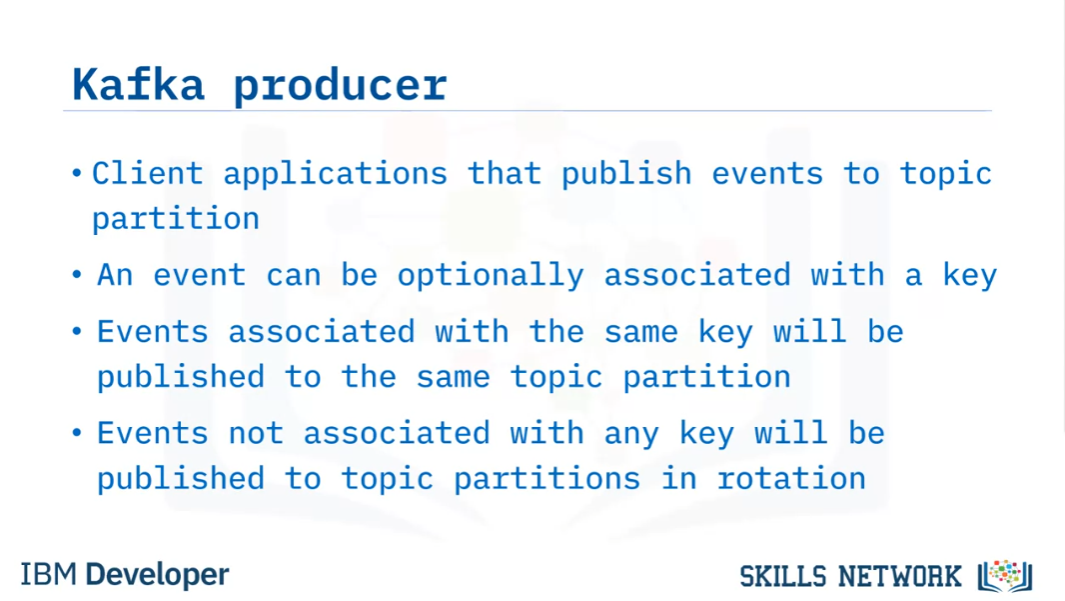
a user topic has been separated into two partitions: 0, 1.

And each topic partition is duplicated into two replications and stored in different brokers.

Kafka Topic CLI



Kafka Producer



A diagram of a production process

Description automatically generated

Let’s see how you can publish events to topic partitions using the following example:

Suppose you have an event source 1, which generates various log entries, and

an event source 2, which generates user-activity tracking records.

Then, you can create a Kafka producer to publish log records to log topic partitions,

and a user producer to publish user-activity events to user topic partitions, respectively.

When you publish events in producers, you can choose to associate events with a key,

for example, an application name or a user ID.

Kafka Producer CLI

A screenshot of a computer program

Description automatically generated

Kafka Consumer

A screenshot of a computer

Description automatically generated

Kafka Consumer in Action

A diagram of a consumer

Description automatically generated

Kafka Consumer CLI

A screenshot of a computer

Description automatically generated

A more concrete Example

A diagram of a weather pipeline

Description automatically generated

Let’s have a look at a more concrete example to help you

understand how to build an event streaming pipeline end-to-end.

Suppose you want to collect and analyze weather and Twitter event streams, so that you can

correlate how people talk about extreme weather on Twitter. Here you can use two event sources:

IBM Weather API to obtain real-time and forecasted weather data in JSON format.

And Twitter API to obtain real-time tweets and mentions, also in JSON format.

To receive weather and twitter JSON data in Kafka, you then create a weather topic

and a Twitter topic in a Kafka cluster, with some partitions and replications.

To publish weather and Twitter JSON data to the two topics,

you need to create a weather producer and a Twitter producer.

The event’s JSON data will be serialized into bytes and saved in Kafka topics.

To read events from the two topics, you need to create a weather consumer

and a Twitter consumer. The bytes stored in Kafka topics will be deserialized into event JSON data.

If you now want to transport the weather

and Twitter event JSON data from the consumers to a relational database,

you will use a DB writer to parse those JSON files and create database records.

And then you can write those records into a database using SQL insert statements.

Finally, you can query the database records from the relational database

and visualize and analyze them in a dashboard to complete the end-to-end pipeline.

## Kafka Streaming Process

Welcome to Kafka Streaming Process. After watching this video, you will be able

to: Describe what the Kafka Streams API is and

its main benefits. Describe what the Kafka Stream processing

topology is. In event streaming, in addition to transporting

data, data engineers also need to process data through, for example, data filtering,

aggregation, and enhancement. Any applications developed to process streams

are called stream processing applications. For stream processing applications based on

Kafka, a straightforward way is to implement an ad hoc data processor to read events from

one topic, process them, and publish them to another topic. Let’s look at an example. You first request raw weather JSON data from

a weather API, and you start a weather producer to publish

the raw data into a raw weather topic. Then you start a consumer to read the raw

weather data from the weather topic. Next, you create an ad hoc data processor

to filter the raw weather data to only include extreme weather events, such as very high

temperatures. Such a processor could be a simple script

file or an application which works with Kafka clients to read and write data from Kafka. Afterwards, the processor sends the processed

data to another producer, and it gets published to a processed weather topic. And finally, the processed weather data will

be consumed by a dedicated consumer and sent to a dashboard for visualization. Such ad hoc processors may become complicated

if you have many different topics to be processed. A solution that may solve these challenges

is Kafka. It provides the Streams API to facilitate

stream processing. Kafka Streams API is a simple client library

aiming to facilitate data processing in event-streaming pipelines. It processes and analyzes data stored in Kafka

topics, thus both the input and output of the Streams API are Kafka topics. Additionally, Kafka Streams API ensures that

each record will only be processed once. And finally, it processes only one record

at a time. Kafka Streams API is based on a computational

graph called a stream-processing topology. In this topology, each node is a stream processor

which receives streams from its upstream processor, performs data transformations such as mapping,

filtering, formatting, aggregation, and produces output streams to its downstream stream processors. Thus, the edges of the graph are I/O streams. There are two special types of processors: On the left, you can see the source processor

which has no upstream processors. A source processor acts like a consumer which

consumes streams from Kafka topics and forwards the processed streams to its downstream processors. On the right, you can see the sink processor,

which has no downstream processors. A sink processor acts like a producer which

publishes the received stream to a Kafka topic. Let’s re-design the previous weather stream

processing application with Kafka Streams API. Suppose you have a raw weather topic and a

processed weather topic in Kafka. Now, instead of spending a huge amount of

effort developing an ad hoc processor, you could just plug in the Kafka Streams API here. In the Kafka Streams topology, we have three

stream processors: The source processor that consumes raw weather

streams from the raw weather topic and forwards the weather stream to the stream processor to filter the stream

based on high temperature. Then, the filtered stream will be forwarded

to the sink processor which then publishes the output to the processed weather topic. Concluding, this is a much simpler design

than an ad hoc data processor, especially if you have many different topics to be processed. In this video, you learned that: Kafka Streams API is a simple client library

to help data engineers with data processing in event streaming pipelines. A stream processor receives, transforms, and

forwards the streams. Kafka Streams API is based on a computational

graph called a stream processing topology. And in the topology, each node is a stream

processor, while edges are the I/O streams. And finally, in this topology, we find two special types

of processors: The source processor and the sink processor.

A diagram of weather forecast

Description automatically generated

Let’s look at an example.

You first request raw weather JSON data from a weather API,

and you start a weather producer to publish the raw data into a raw weather topic.

Then you start a consumer to read the raw weather data from the weather topic.

Next, you create an ad hoc data processor to filter the raw weather data to only include

extreme weather events, such as very high temperatures.

Such a processor could be a simple script file or an application which works with Kafka

clients to read and write data from Kafka.

Afterwards, the processor sends the processed data to another producer, and it gets published

to a processed weather topic.

And finally, the processed weather data will be consumed by a dedicated consumer and sent

to a dashboard for visualization.

Kafka Streams API

A screenshot of a computer

Description automatically generated

It provides the Streams API to facilitate stream processing.

Kafka Streams API is a simple client library aiming to facilitate data processing in event-streaming

pipelines.

It processes and analyzes data stored in Kafka topics, thus both the input and output of

the Streams API are Kafka topics.

Additionally, Kafka Streams API ensures that each record will only be processed once.

And finally, it processes only one record at a time.

Stream Processing Topology

A diagram of process processing

Description automatically generated

In this topology, each node is a stream processor which receives streams from its upstream processor,

performs data transformations such as mapping, filtering, formatting, aggregation, and produces

output streams to its downstream stream processors.

Thus, the edges of the graph are I/O streams.

There are two special types of processors:

On the left, you can see the source processor which has no upstream processors.

A source processor acts like a consumer which consumes streams from Kafka topics and forwards

the processed streams to its downstream processors.

On the right, you can see the sink processor, which has no downstream processors.

A sink processor acts like a producer which publishes the received stream to a Kafka topic.

Kafka Weather stream processing

A diagram of a weather stream process

Description automatically generated

Let’s re-design the previous weather stream processing application with Kafka Streams

API.

Suppose you have a raw weather topic and a processed weather topic in Kafka.

Now, instead of spending a huge amount of effort developing an ad hoc processor, you

could just plug in the Kafka Streams API here.

In the Kafka Streams topology, we have three stream processors:

The source processor that consumes raw weather streams from the raw weather topic and forwards

the weather stream to

the stream processor to filter the stream based on high temperature.

Then, the filtered stream will be forwarded to the sink processor which then publishes

the output to the processed weather topic.

Concluding, this is a much simpler design than an ad hoc data processor, especially

if you have many different topics to be processed.