
ONEWAY
SOLUTION

One Way Solution

The Ingestion

Data Engineering – [Day 2]

LUAN MORENO

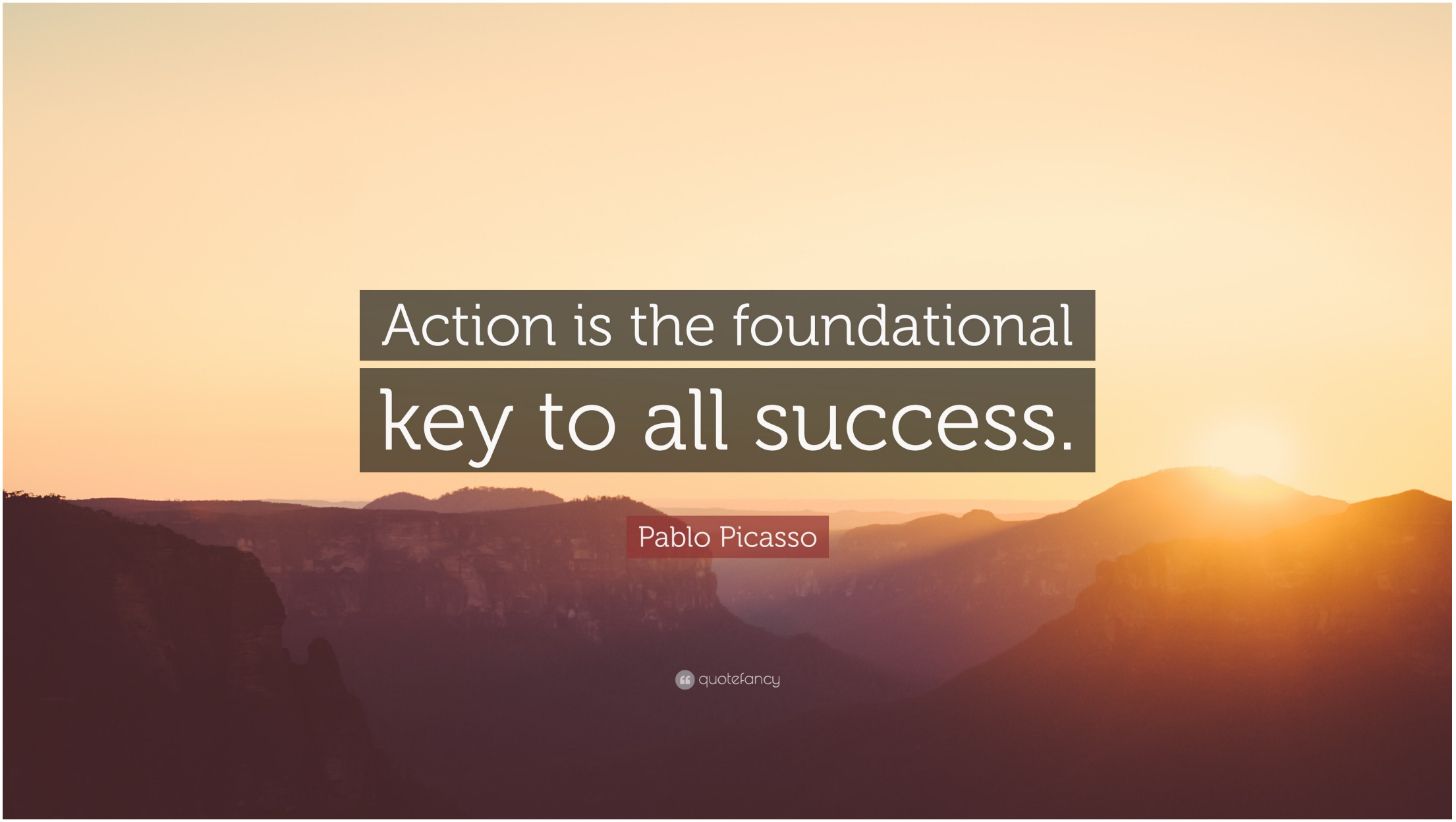
CEO & Data Architect

Data Engineer & MVP

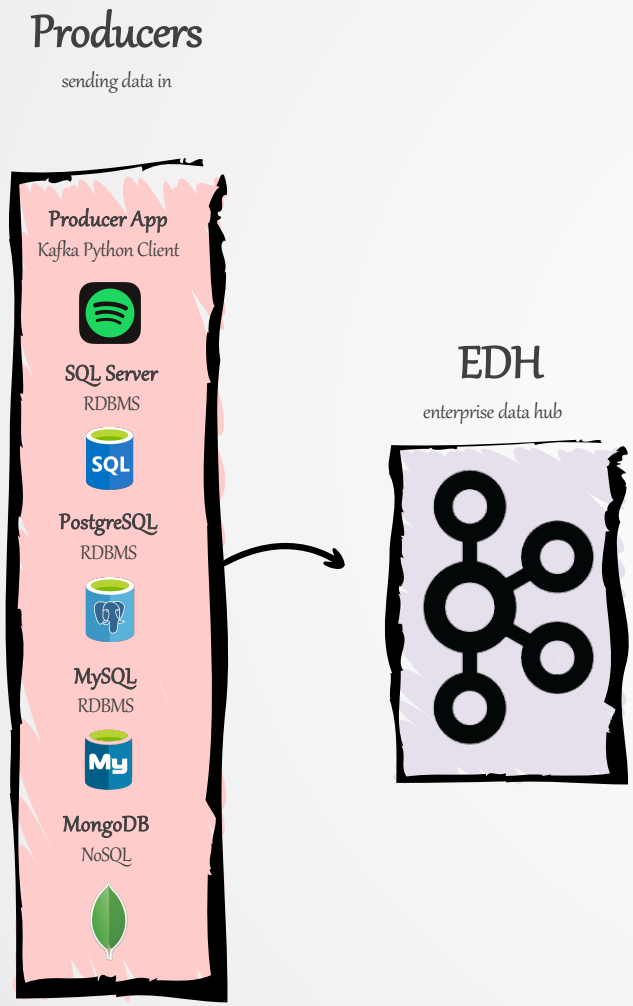
MATEUS OLIVEIRA

Big Data Architect

Data In-Motion Specialist


Action is the foundational
key to all success.
Pablo Picasso
quotefancy

Use Case ~ [Data Ingestion]



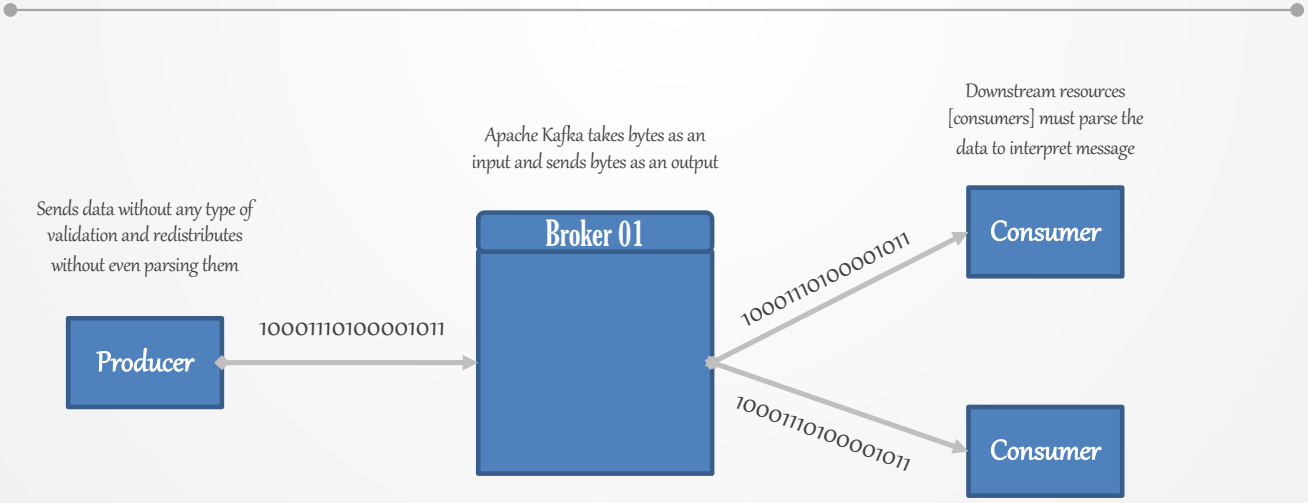
The Data Formats [Data Streaming]

Worst data format for streaming. Data types are inferred, parsing & figuring out header are time consuming. No safety on this format

Heavyweight and CPU intensive, XSD is a nightmare to handle and most of cases XML schema needs to be embedded on payload

There is a schema support and data validation as first-class citizen. Unfortunately since it's not a data format data is converted to JSON

The most used data format which all modern app uses. There is a gain due the flexibility but since there is not schema support adds complexity



Apache Avro & Apache Kafka

Apache Avro

- Open-Source Data Serialization System
- Supported by Different Programming Languages
- Defines a Binary Format for Data
- JSON Data Model for Schema Specification

Features

- Primitive Types
- Embedded Documentation
- Defined using JSON
- Object = Schema & Data
- Schema Evolution
- Efficiency on Data Space
- Compression [Snappy]

```
{
  "time": 1424849130111,
  "customer_id": 1234,
  "product_id": 5678,
  "quantity":3,
  "payment_type": "mastercard"
}
```

This is a stream of record. Remember that once this event reach the kafka broker it will be stored in a binary format.

Apache Kafka

- Kafka Producers & Consumers
- Confluent Platform
- Apache Avro, Protobuf, Json ~ Schema Registry
- Schema Registry [Open-Source]

```
{
  "type": "record",
  "doc":"this event records the sale of a product",
  "name": "ProductSaleEvent",
  "fields" : [
    {"name":"time", "type":"long", "doc":"The time of the purchase"},
    {"name":"customer_id", "type":"long", "doc":"The customer"},
    {"name":"product_id", "type":"long", "doc":"The product"},
    {"name":"quantity", "type":"int"},
    {"name":"payment", "type":{"type":"enum", "name":"payment_types", "symbols":["cash","mastercard","visa"]},
     "doc":"The method of payment"}
  ]
}
```

This is a schema notation for Apache Avro. It creates a better way to manage schemas. In this case, this schema defines the stream of records from the message above.

Protobuf = *method of serializing structured data. Works well with nested structure and it's available for* **Producers** *and* **Consumers**.

Confluent Schema Registry [Apache Avro]

Features

- Apache Avro, JSON Schema & Protobuf
- RESTful Interface for Developers
- Define Standard Schemas
- Schema Evolution

event that will be stored in a topic

```
{
  "id":1,
  "name":"luan moreno medeiros maciel",
  "company":"one way solution"
}
```

Terminologies

- Topic = Key-Value Pair [Events]
- Serialization = Key & Value [Avro, Json, Protobuf]
- Schema = Data Format Structure
- Subject = Scope of Topic

JSON Schema Definition

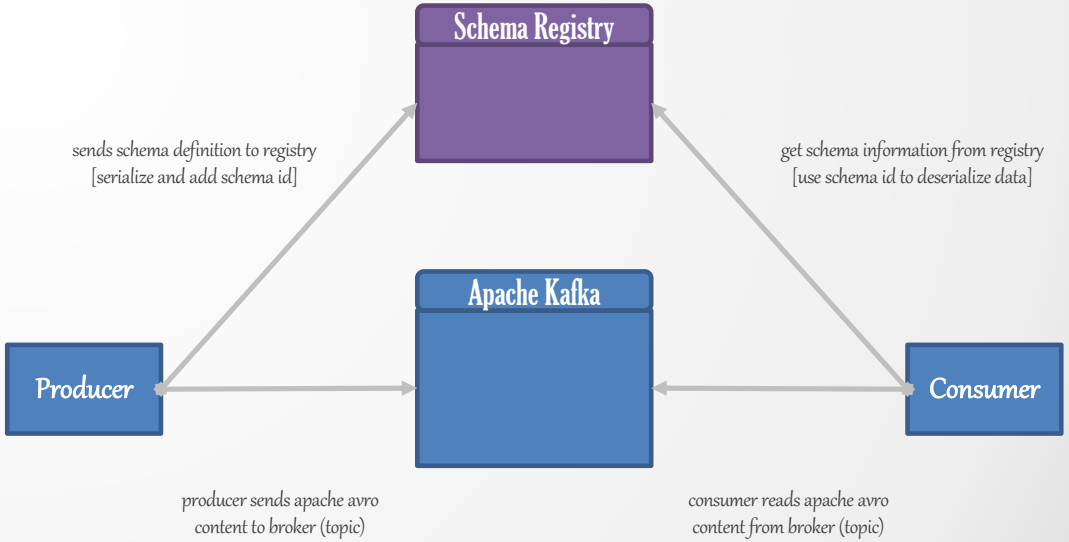
```
{
  "type": "object",
  "properties": {
    "id": {
      "type": "integer"
    },
    "name": {
      "type": "string"
    },
    "company": {
      "type": "string"
    }
  },
  "required": [
    "id",
    "name",
    "company"
  ]
}
```

Apache Avro Schema Definition

```
{
    "type": "record",
    "namespace": "com.onewaysolution",
    "name": "Users",
    "fields": [
      { "name": "name", "type": "int" },
      { "name": "name", "type": "string" },
      { "name": "company", "type": "string" }
    ]
}
```

Protobuf Schema Definition

```
message Users
{
  required int32 id = 1;
  required string name = 2;
  optional string company = 3;
}
```




Success is doing ordinary things extraordinarily well.
Jim Rohn
quotefancy

Kafka Producer [Basics]

Producer API [101]

- **Producers** = Publishers or Writes in a Traditional Messaging Systems
- Writes Data as Messages [Events] in a Topic(s)
- Message Key & Partitioner to Write into a Specific Partition

- REST Server for Unsupported Languages
- Command-Line Producer Tool

Programming Languages [Kafka Clients]

Librdkafka is a C library of the Apache Kafka Protocol, providing Producer, Consumer & Admin Clients. Designed with Message Delivery Reliability and High Performance in mind.

Producers = 1 Million of Messages per Second
Consumers = 3 Million of Messages per Second

Confluent

- C, C++, .Net, Python, Go

Open-Source Community

- Erlang, Groovy, Haskell, Kotlin, Lua, Node.JS, OCaml, PHP, Ruby, Rust, TCL, Swift

Kafka Producer [librdkafka]

Kafka Improvement Proposals [KIP]

- Apache Kafka [Confluence]
- LIBRDKAFKA [GitHub]

Apache Kafka C/C++ Client Library

- Full Exactly-Once-Semantics (EOS) Support
- High-Level Producer with Idempotent & Transactional Producers
- High-Level Balanced Kafka Consumer
- Simple (Legacy) Consumer
- Admin Client
- Compression = SNAPPY, GZIP, LZ4, ZSTD
- SSL Support

Confluent's Python Client for Apache Kafka

confluent-kafka-python provides a high-level producer, consumer & adminclient.

writes the program in python using the library – confluent-kafka-python

librdkafka (compiler) run behind the scenes to transport bytes to kafka broker

data is written to a kafka topic inside of a partition

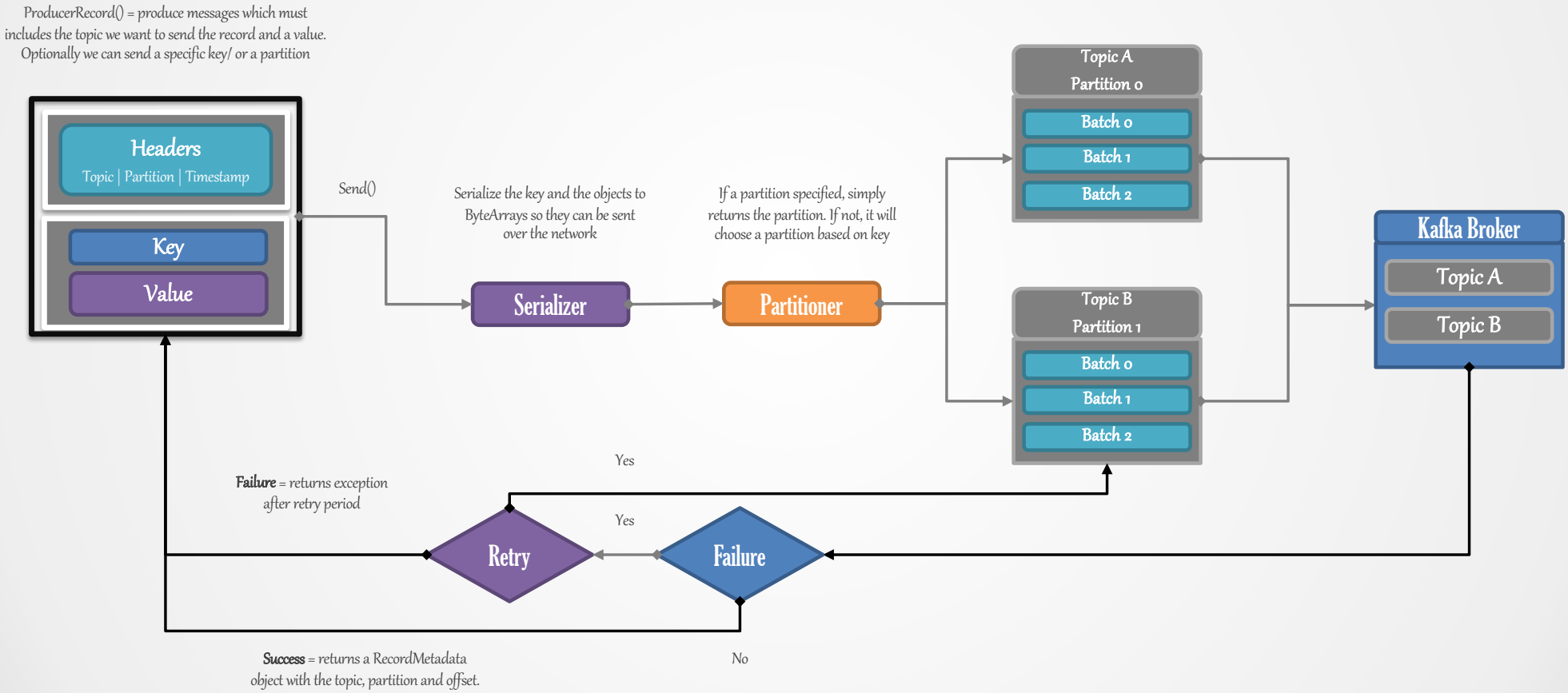
MacOS = Homebrew

Linux = Confluent APT

Linux = Confluent YUM

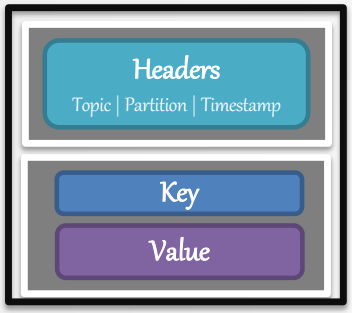
Windows = NuGet

Kafka Producer [Design]



Kafka Producer [Core Configuration]

ProducerRecord() = produce messages which must includes the topic we want to send the record and a value. Optionally we can send a specific key/ or a partition



message that will be written into a topic

Producer

Apache Kafka

serialization is the process of converting an object into a stream of bytes that are used for transmission

deserialization is the process to convert bytes of arrays into the desired data type

Serializers

- Serialization = ByteArray
- ByteArray, ByteBuffer, Bytes, Double, Integer, Long, String
- Custom Serializer

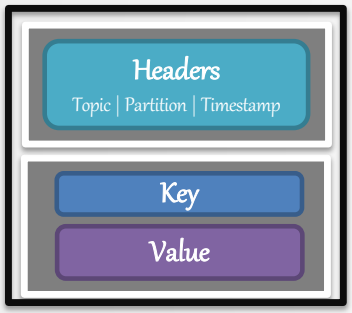
Initial Configs

- **bootstrap.servers** = list of host:pairs of brokers, include at least two
- **client.id** = to easily correlate requests on broker with the client instance

- **key.serializer** = serialize the key, name of the class that implements interface
- **value.serializer** = name of the class to serialize values of records

Kafka Producer [Message Durability]

ProducerRecord() = produce messages which must includes the topic we want to send the record and a value. Optionally we can send a specific key/ or a partition



Send()

ACKS	Batch Size (Bytes)	Throughput (MB/s)
0	10,000	70.63
1	10,000	55.93
ALL	10,000	26.78

Message Methods

Fire-and-Forget = send() a message to the server and don't really care if it arrives successfully or not. Most of the time, it will arrive successfully however, some messages will get lost using this method.

Synchronous Send = the send() method returns a Future object, and we use get() to wait on the future and see if the send() was successfully or not.

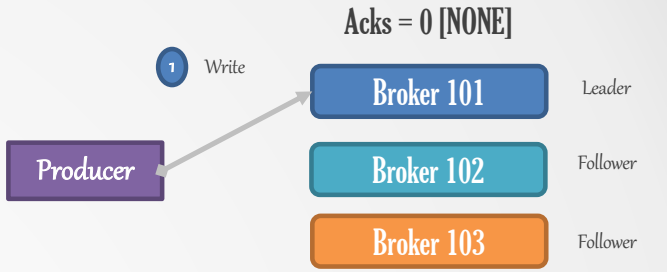
Asynchronous Send = the send() method with a callback functions, which gets triggered when it receives a response from the Kafka broker.

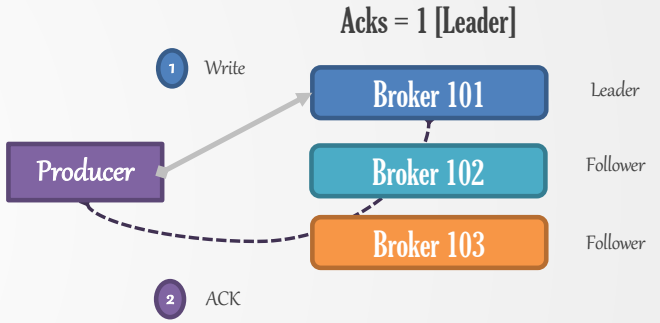
Acknowledgment = [Acks]

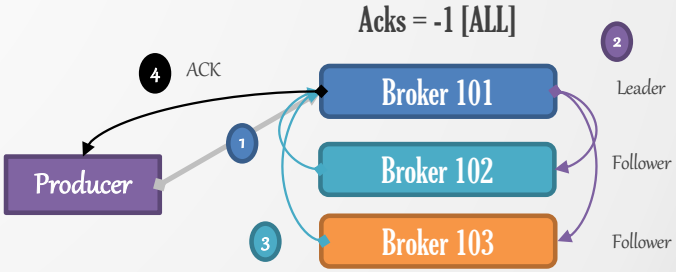
Acks [0] = the producer will not wait for a reply from the broker before assuming the message was sent successfully. Used for **High Throughput**

Acks [1] = the producer will receive a success response from the broker the moment the leader replica receives a message, if message can't be written the producer will get an error response and will retry. Throughput depends on Sync or Async, if use callback latency will be hidden and send will be limited to in-flight messages. User for **Message Guarantee and High Throughput**

Acks [ALL] = the producer will receive a success response from broker once all in-sync replicas received the message. Safest mode since will guarantee that message will remain after a crash. Used for **High Availability**



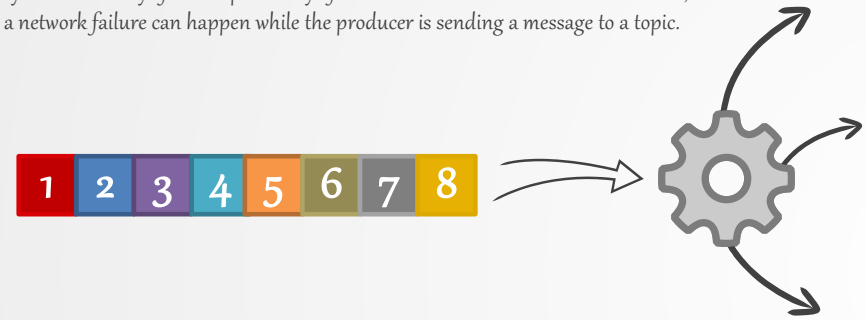




Kafka Producer [Idempotent Producers & EOS]

Messaging System Semantics

in a distributed publish-subscribe messaging system, the computers that make up the system can always fail independently of one another. Individual broker can crash, or a network failure can happen while the producer is sending a message to a topic.



At [Most] Once

if producer does not retry when an ack times out or returns an error, the message might end up not being written to a kafka topic, and hence not delivered to consumer.

ups, lost some...

At [Least] Once

producer receives an (ack) from broker with acks=ALL, meaning it was written once to the kafka. however, if an error arise, it might retry sending the same message, assuming that the message was not written.

hmm, got some duplicates...

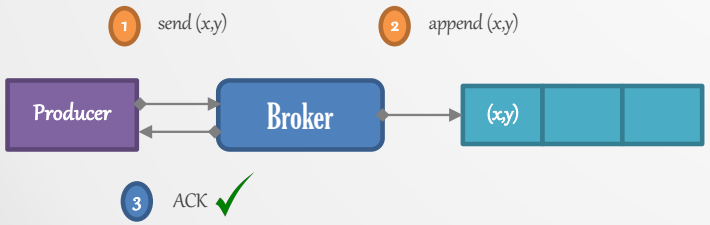
[Exactly] Once Semantics [EOS]

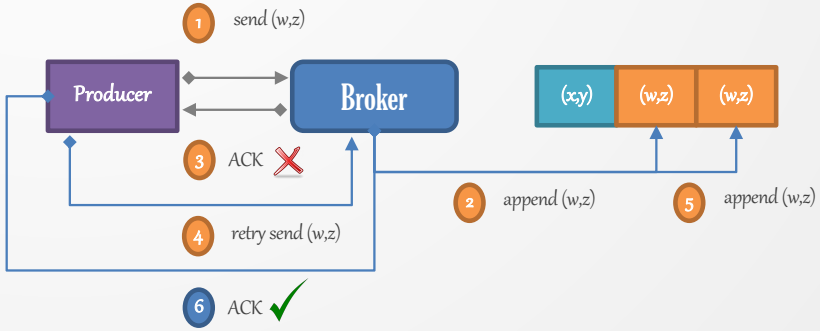
even if a producer retries sending a message, it leads to the message being delivered exactly once to the end consumer. requires cooperation between the systems.

this is exactly what i was waiting for...

Idempotent Producers

can be performed many times without causing a different effect than only being performed once. if retries occurs, same message won't be written again. each message will contain a sequence number which broker will use to dedup any duplicate send.





EOS [Exactly-Once Semantics]

KIP-98

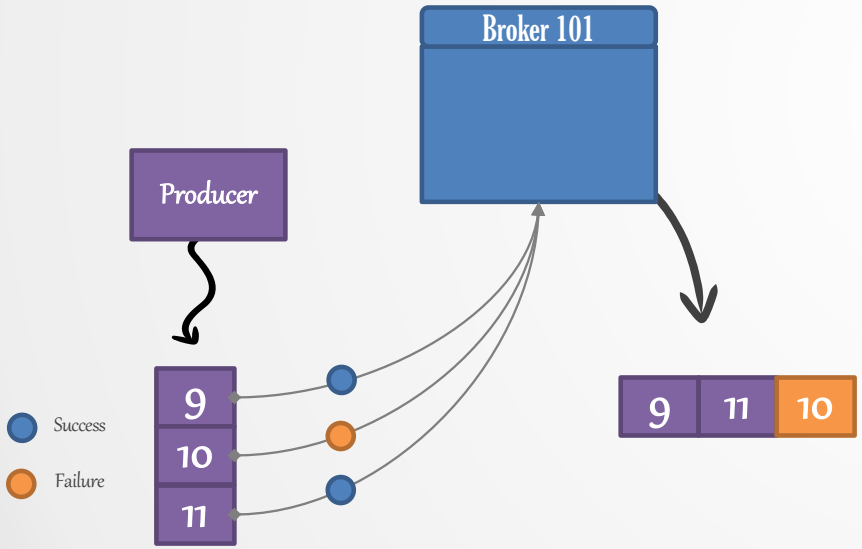
Apache Kafka Release = 0.11 [June 28, 2017]
Confluent Platform = 3.3 [August 01, 2017]

Producer = Enable.Idempotence = TRUE

Kafka Producer [Message Ordering]

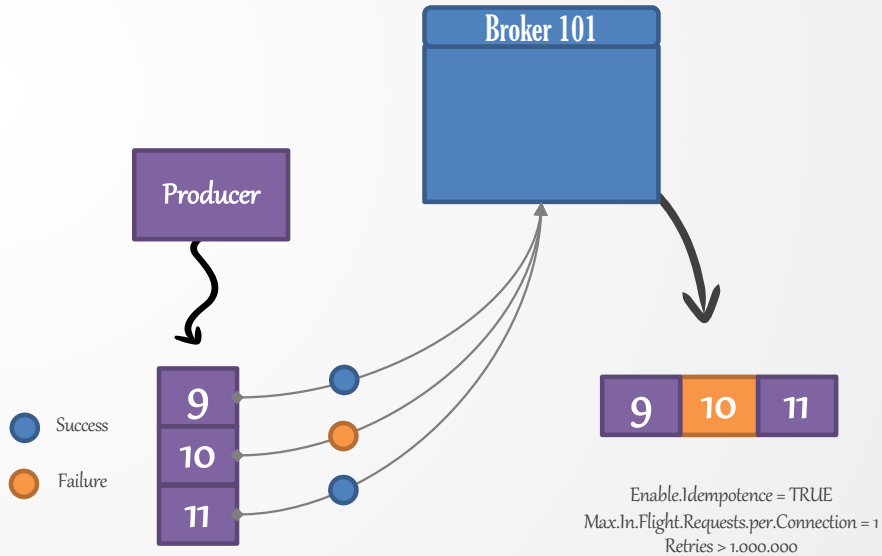
Retries **with** Message Reordering

messages are written to the broker in the same order that they are received by the producer client. however, if you enable **retries > 0 (default),** then message reordering may occur since the retry may occur after a following write succeeded.



Retries **without** Message Reordering

to enable retries without message reordering set **max.in.flight.requests.per.connection** = 1 to ensure that only one request can be sent to broker at a time.

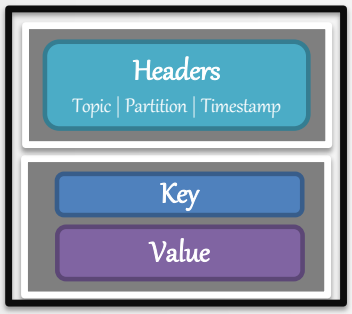


Enable.Idempotence = TRUE
Max.In.Flight.Requests.per.Connection = 1
Retries > 1.000.000
ACKS = ALL

Used for High Durability & Availability

Kafka Producer [Batching & Compression]

ProducerRecord() = produce messages which must includes the topic we want to send the record and a value. Optionally we can send a specific key/ or a partition



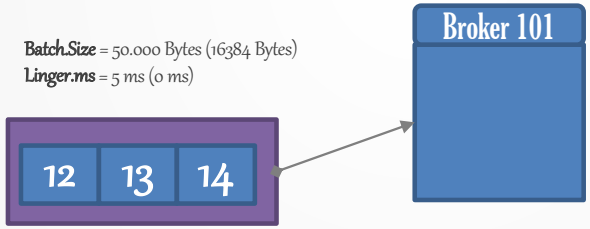
Send()

Batching

producers attempt to collect sent messages into batches to improve throughput.

- **batch.size** when multiple records are sent to the same partition, the producer will batch them together, this control the amount of memory in bytes (not messages). when batch is full, all messages will be sent. Setting size too small add some overhead.
- **linger.ms** controls the amount of time to wait for additional messages before sending the current batch. By default producer will send messages as soon as there is a send thread available. Setting linger > 0 increases latency but also increase throughput.

Batch.Size = 50.000 Bytes (16384 Bytes)
Linger.ms = 5 ms (0 ms)



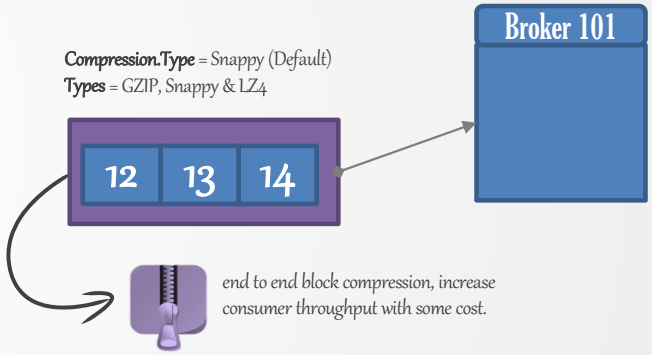
Batch Size (Bytes)	Linger (ms)	Throughput (MB/s)
1,000	60	8.69
50,000	60	140.72
500,000	60	189.71

Compression

Compression covers full message batches, so larger batches will typically mean a higher compression ratio. By enabling compression, you reduce network utilization and storage, which is often a bottleneck when sending messages to Kafka.

Snappy compression was invented by Google to provide decent compression ratios with low CPU overhead and good performance, so it is recommended in cases where both performance and bandwidth are a concern. **Gzip** compression will typically use more CPU and time but result in better compression ratios, so it recommended in cases where network bandwidth is more restricted.

Compression.Type = Snappy (Default)
Types = GZIP, Snappy & LZ4



end to end block compression, increase consumer throughput with some cost.

Batch Size (Bytes)	Linger (ms)	Compression Type	Throughput (MB/s)
10,000	60	None	55.93
10,000	60	GZIP	131.35
10,000	60	LZ4	293.38
10,000	60	SNAPPY	208.68

Kafka Producer [Sticky Partitioner]

Sticky Partitioning Strategy

sticky partitioner addresses the problem of spreading out records without keys into smaller batches by picking a single partition to send all non-keyed records. Once the batch at that partition is filled or otherwise completed, the sticky partitioner randomly chooses and "sticks" to a new partition. That way, over a larger period of time, records are about evenly distributed among all the partitions while getting the added benefit of larger batch sizes.

Sticky Partitioner

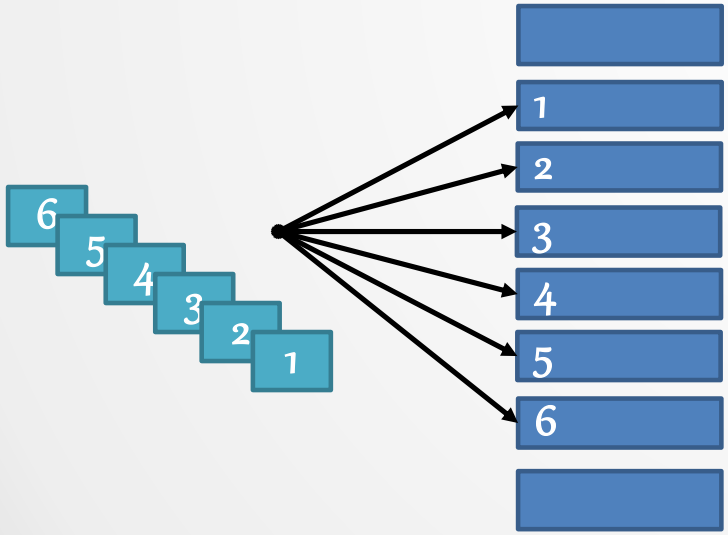
KIP-480 [16/12/2019]
onNewBatch

Messages = 1,000 (ms)
Batch.Size = 16,384 (bytes)
Linger.MS = 1,000 (ms)
KEY = NULL

Default Partitioner = 500 ms
Sticky Partitioner = 125 ms

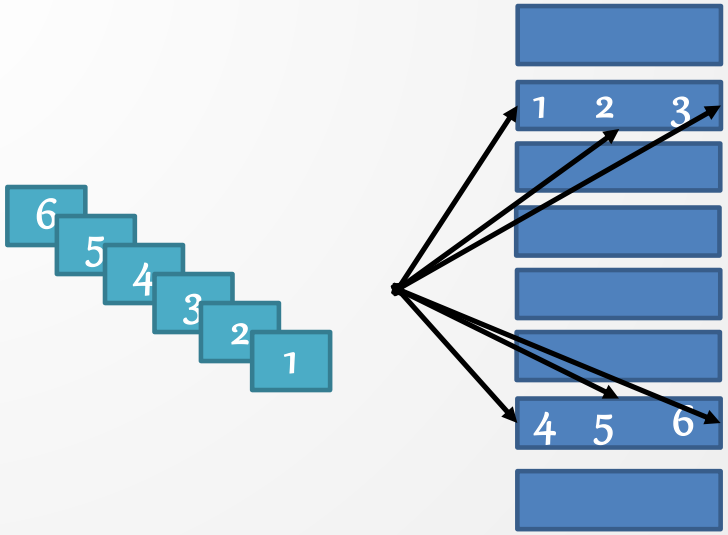
Round-Robin Partitioning Strategy

each partition receives a batch with a single record



Sticky Partitioning Strategy

stick to a partition until the batch is closed; each partition receives a full batch with three records.



Building a Kafka Producer Application using Python [confluent-kafka-python]

1. Producer Configuration Settings
2. Kafka Producer API
3. Schema Registry Integration

Kafka Skin Producer using an API Gateway

in most of places we have producer write directly ingest events into kafka, but not everybody can leverage the best of apache kafka so what we see is the creation of we call kafka skin

Kong is an api gateway open-source that can act as gatekeeper of your applications, we can setup authentication, routing and many other services

in this scenario we give the access to producer connects into kafka and write into topics, each developer choose the way to write, manually we can lose pattern and best practices

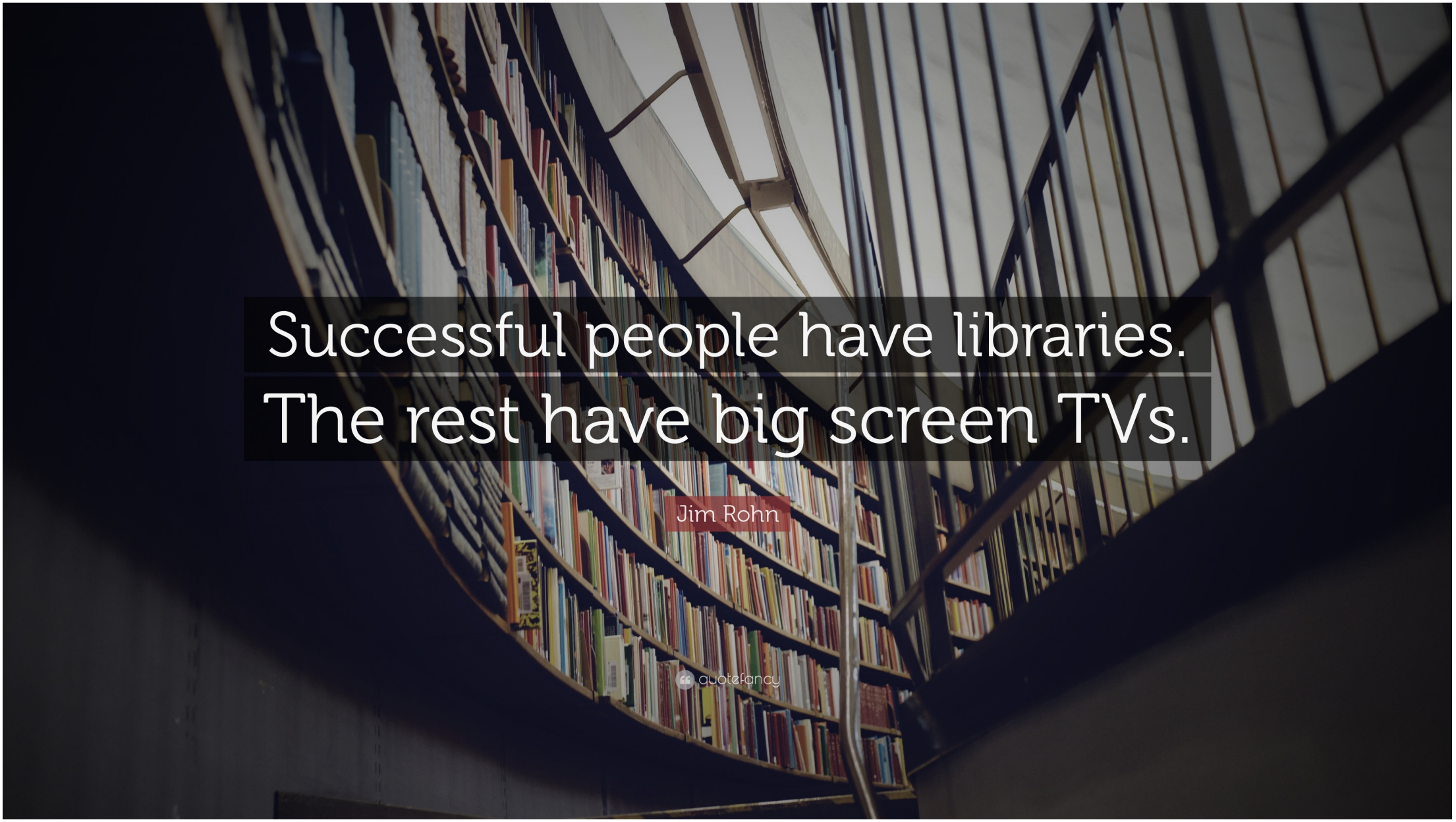
kafka cluster receive the producer request and write into the topics as producer configuration demands, we can ingest different types of events and the same topic without, and we need to build a safe net in the development side

now with an api sending post request to the api gateway, the developer doesn't need to know apache kafka access and only need to send the post request to produce data and the api gateway + kafka api will be responsible to deliver for apache kafka cluster

API Gateway is responsible to route the request and send to the skin, we can configure authentication and request rate limits, it will route the post request into kafka skin api that will write into apache kafka cluster

kafka skin writes into kafka topic, only the skin access the kafka topic from the producer side

Kafka Skin API will receive post request from api, the kafka skin will have all the configurations for a producer include call back, the idea is thought about what we need to write into kafka and let the skin enable with all correct configurations


Successful people have libraries.
The rest have big screen TVs.
Jim Rohn
quotefancy

Kafka Connect [Basics]

Kafka Connect [101]

- Open-Source Framework Component for Apache Kafka
- Connect with External Systems

- Source Connector = Ingest Data from Data Sources [Databases, Key-Value Stores]
- Sink Connector = Delivers Data to Data Sources [Search Indexes, File Systems]

Benefits

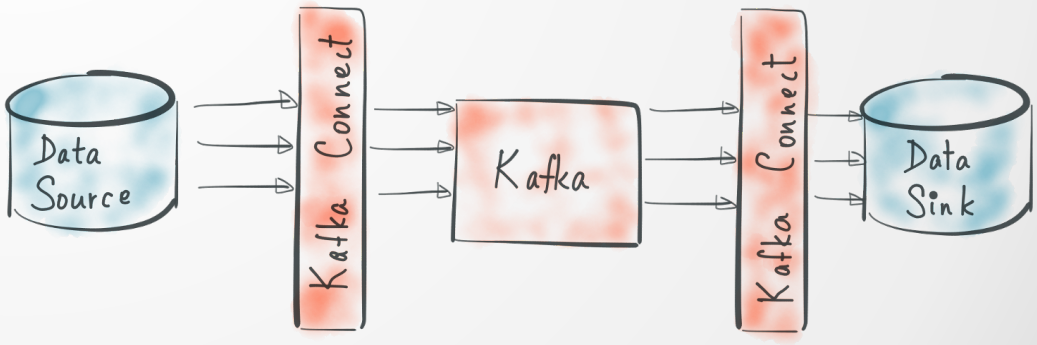
- Data Centric Pipeline = Meaningful Data Abstractions to Pull Data
- Flexibility & Scalability = Runs with Streaming & Batch-Oriented Systems
- Reusability & Extensibility = Lower Time to Production

- Standalone vs. Distributed Mode
- REST Interface

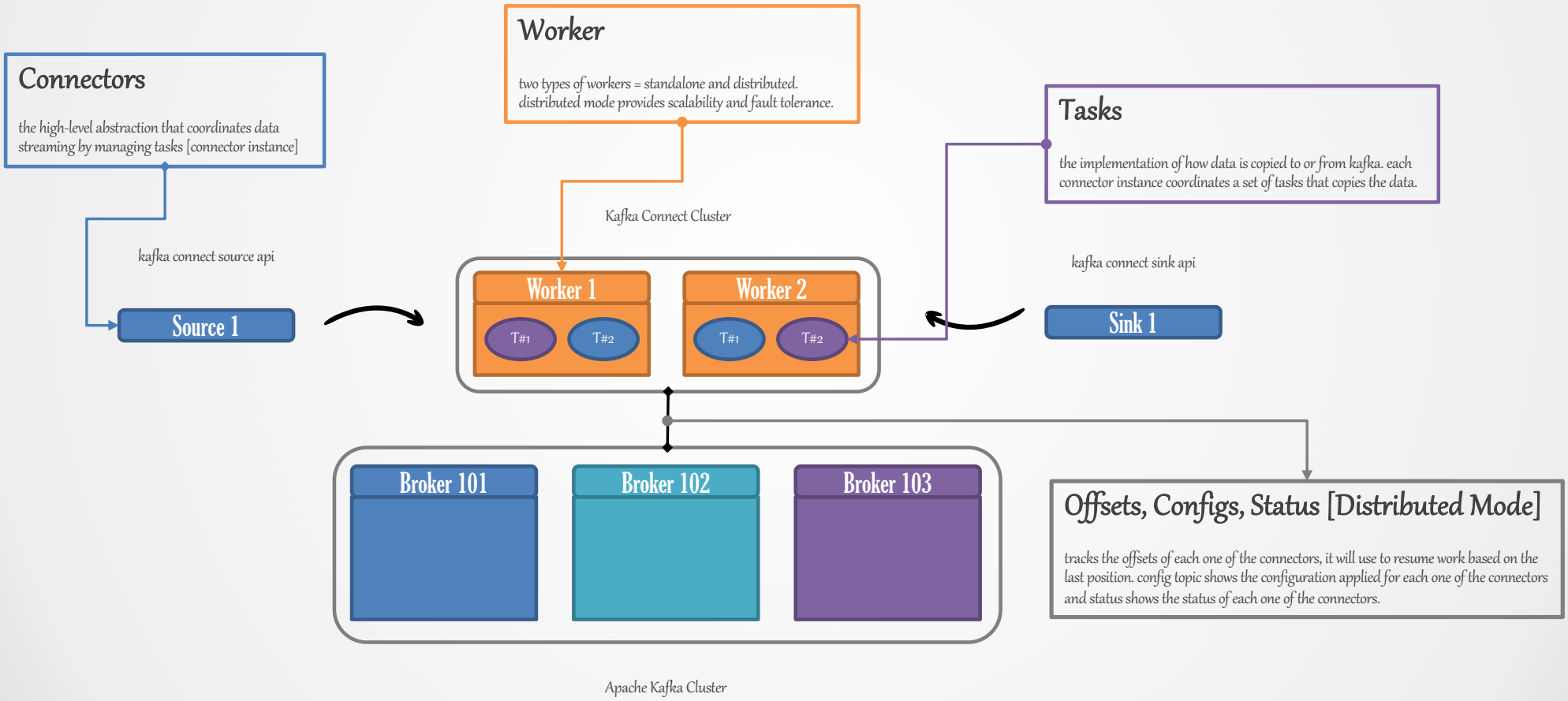
Motivation

- Serve as a Data Integration Hub [Confluent Hub]

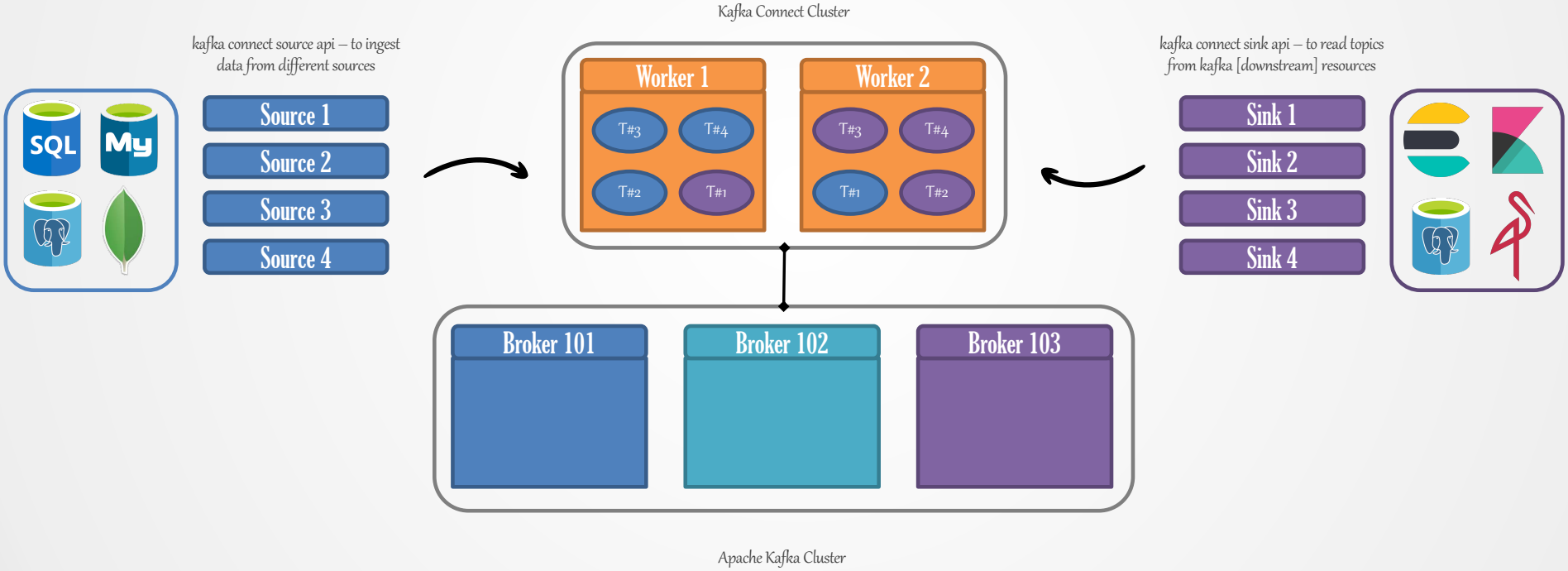
- Use-Case [1] = Log & Metric Collection, Processing & Aggregation
- Use-Case [2] = ETL for Data Warehousing
- Use-Case [3] = Data Pipelines Management [Apache Nifi]



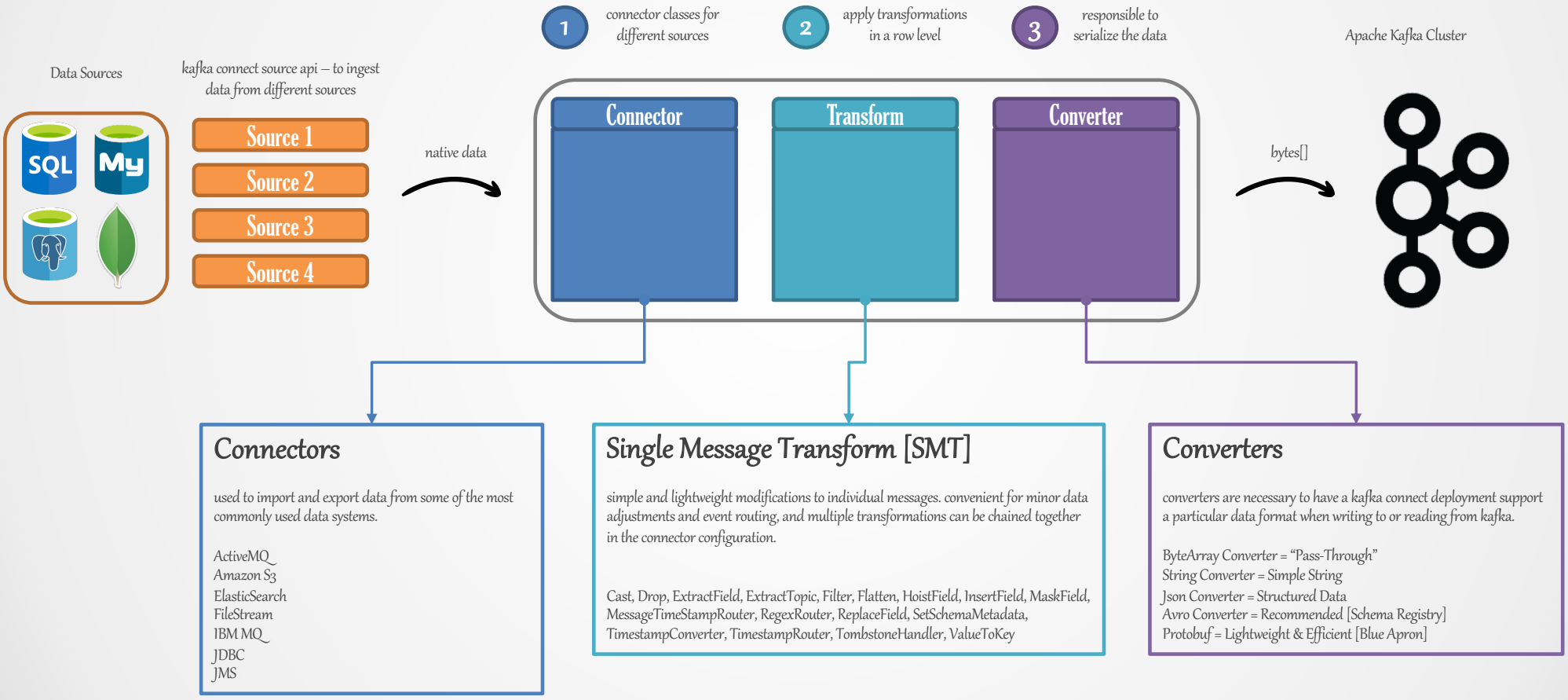
Kafka Connect [Core Concepts]



Kafka Connect [Sources & Sinks]



Kafka Connect [Data Movement]



Connectors

used to import and export data from some of the most commonly used data systems.

ActiveMQ
Amazon S3
ElasticSearch
FileStream
IBM MQ
JDBC
JMS

Single Message Transform [SMT]

simple and lightweight modifications to individual messages. convenient for minor data adjustments and event routing, and multiple transformations can be chained together in the connector configuration.

Cast, Drop, ExtractField, ExtractTopic, Filter, Flatten, HoistField, InsertField, MaskField, MessageTimeStampRouter, RegexRouter, ReplaceField, SetSchemaMetadata, TimestampConverter, TimestampRouter, TombstoneHandler, ValueToKey

Converters

converters are necessary to have a kafka connect deployment support a particular data format when writing to or reading from kafka.

ByteArray Converter = "Pass-Through"
String Converter = Simple String
Json Converter = Structured Data
Avro Converter = Recommended [Schema Registry]
Protobuf = Lightweight & Efficient [Blue Apron]

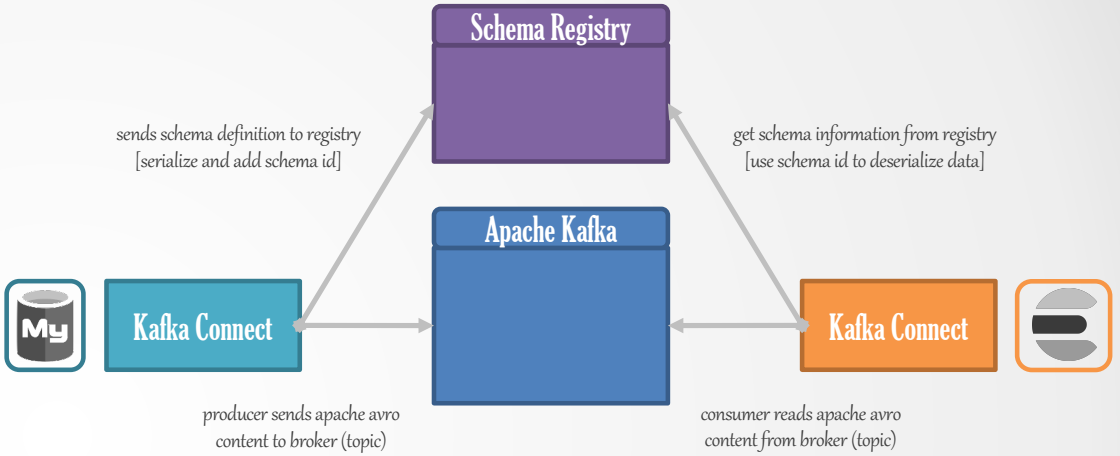
Kafka Connect & Schema Registry

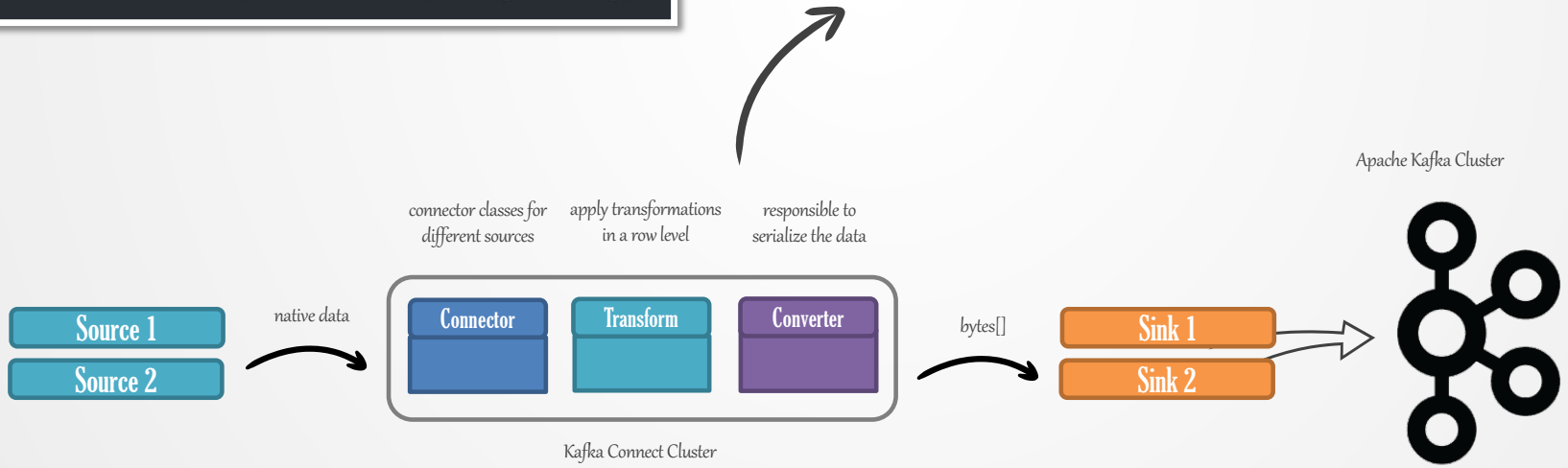
deploying a connector with schema registry

```
curl -X POST -H "Content-Type: application/json" --data '{"name": "ingest-src-mssql-user-names-avro-1", "config":
{
    "connector.class":"io.confluent.connect.jdbc.JdbcSourceConnector",
    "key.converter":"io.confluent.connect.avro.AvroConverter",
    "key.converter.schema.registry.url":"http://localhost:8081",
    "value.converter":"io.confluent.connect.avro.AvroConverter",
    "value.converter.schema.registry.url":"http://localhost:8081"
  }
}' http://localhost:8083/connectors
```

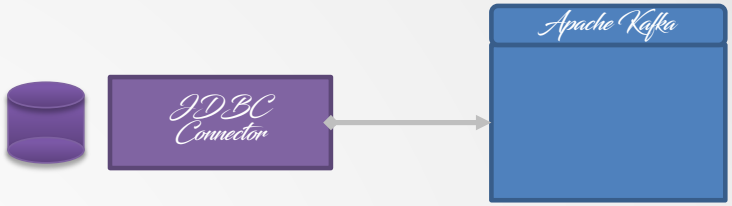
retrieve info from schema registry

```
curl --silent -X GET http://localhost:8081/subjects/ | jq
```



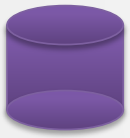


Kafka Connect [JDBC Source Connector]



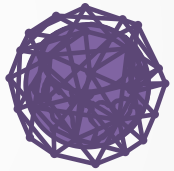
JDBC Connector [Source & Sink] for Confluent Platform

you can use the kafka connect jdbc source connector to import data from any relational database with a jdbc driver into apache kafka® topics. you can use the jdbc sink connector to export data from kafka topics to any relational database with a jdbc driver. the jdbc connector supports a wide variety of databases without requiring custom code for each one.



Relational Database Systems [RDBMS]

- Microsoft SQL Server
- PostgreSQL
- Oracle Database
- IBM DB2
- MySQL Server
- SAP HANA
- SQLite Embedded Database



Features

Incremental Query Modes

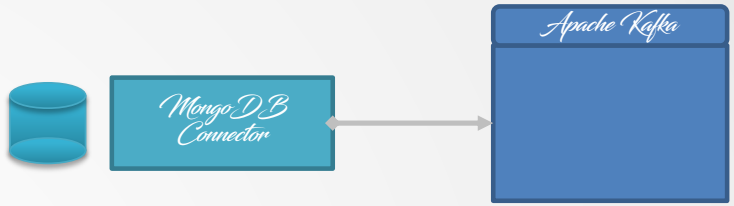
- Incrementing Column
- Timestamp Column
- Timestamp + Incrementing Columns
- Custom Query
- Bulk

Message Keys = SMT [Single Message Transform]

- ValueToKey
- ExtractField$Key

Schema Evolution [Apache Avro ~ Confluent Schema Registry]

Kafka Connect [MongoDB Source Connector]

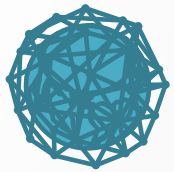


MongoDB Kafka Connector

is a confluent-verified connector that persists data from kafka topics as a data sink into mongodb as well as publishes changes from mongodb into kafka topics as a data source. this guide provides information on available configuration options and examples to help you complete your implementation.

NoSQL ~ MongoDB

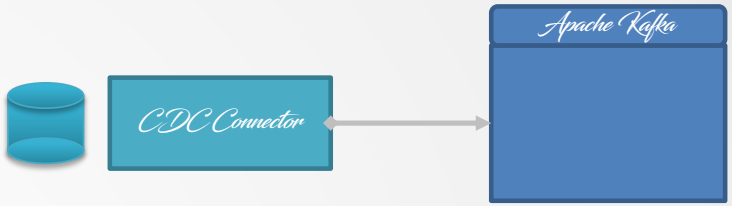
- CAP Base Theorem
- Document Database [Json]
- Fast Development Time
- High Performance
- Rich Query Language
- High Availability
- Horizontal Scalability
- Storage Engines – WiredTiger & In-Memory



Features

- Change Streams Documents [Real-Time] Data Changes
- Collection, Database & Deployment
- Publish Full Document Only
- Change Stream Full Document
- Copy Existing

Kafka Connect [Debezium CDC Connector] for Event Sourcing

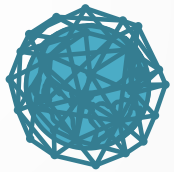


Debezium CDC Connectors

Debezium is an Open-Source Distributed Platform for Change Data Capture, Inside Debezium.io ~ Collection of Connector Build Using CDC as Base for Event Sourcing

Supported Data Sources

- MySQL
- MongoDB
- PostgreSQL
- Oracle
- SQL Server
- DB2
- Cassandra



Features

- Capture Based in CDC [Insert, Update and Delete]
- decimal.handling.mode (SQL)
- tombstones.on.delete
- Custom SMTs

Kafka Connect Source [Deploying Connectors]

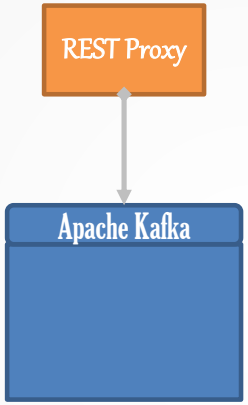
1. Meet Data Sources
2. Deploying RDBMS & NoSQL
3. Deploying Event Sourcing Connectors


Success usually comes to those who are too busy to be looking for it.
Henry David Thoreau
quotefancy

Confluent REST Proxy [Basics]

REST Proxy [101]

- RESTful Interface to Kafka Cluster
- Used to Produce and Consume Messages
- View State of Kafka Cluster and Perform Administrative Actions



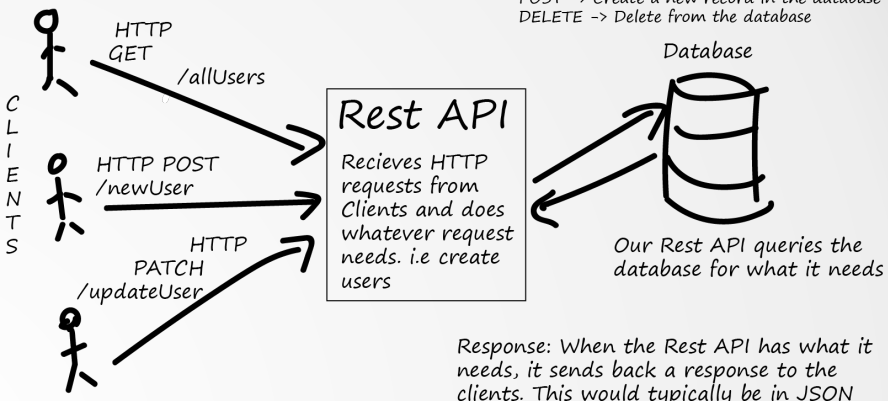
Features

Current Supported Actions for REST Proxy

- Metadata = metadata information
- Producers = accepts produce requests targeted at specific topics or partitions
- Consumers = limited to one thread per consumer, use multiple for throughput
- Data Formats = Json, Raw Bytes Encoded with Base64 and Avro
- REST Proxy Clusters & Load Balancing = supports multiple instances

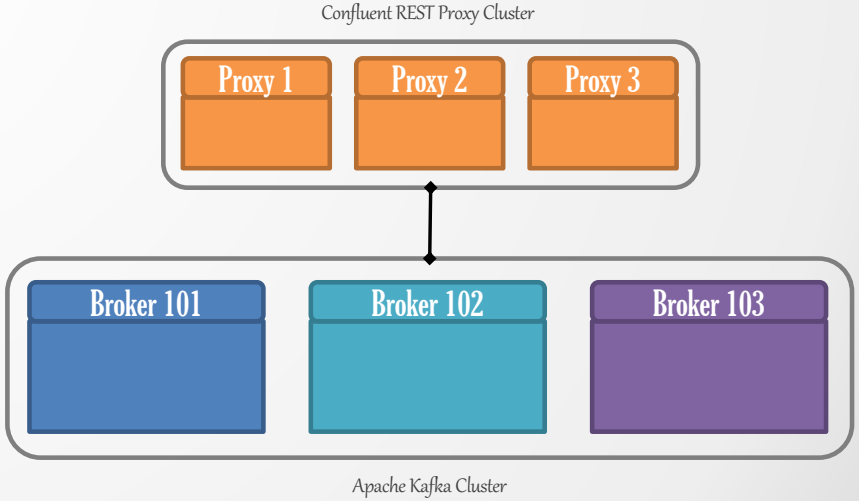
Rest API Basics

Typical HTTP Verbs:
GET -> Read from Database
PUT -> Update/Replace row in Database
PATCH -> Update/Modify row in Database
POST -> Create a new record in the database
DELETE -> Delete from the database

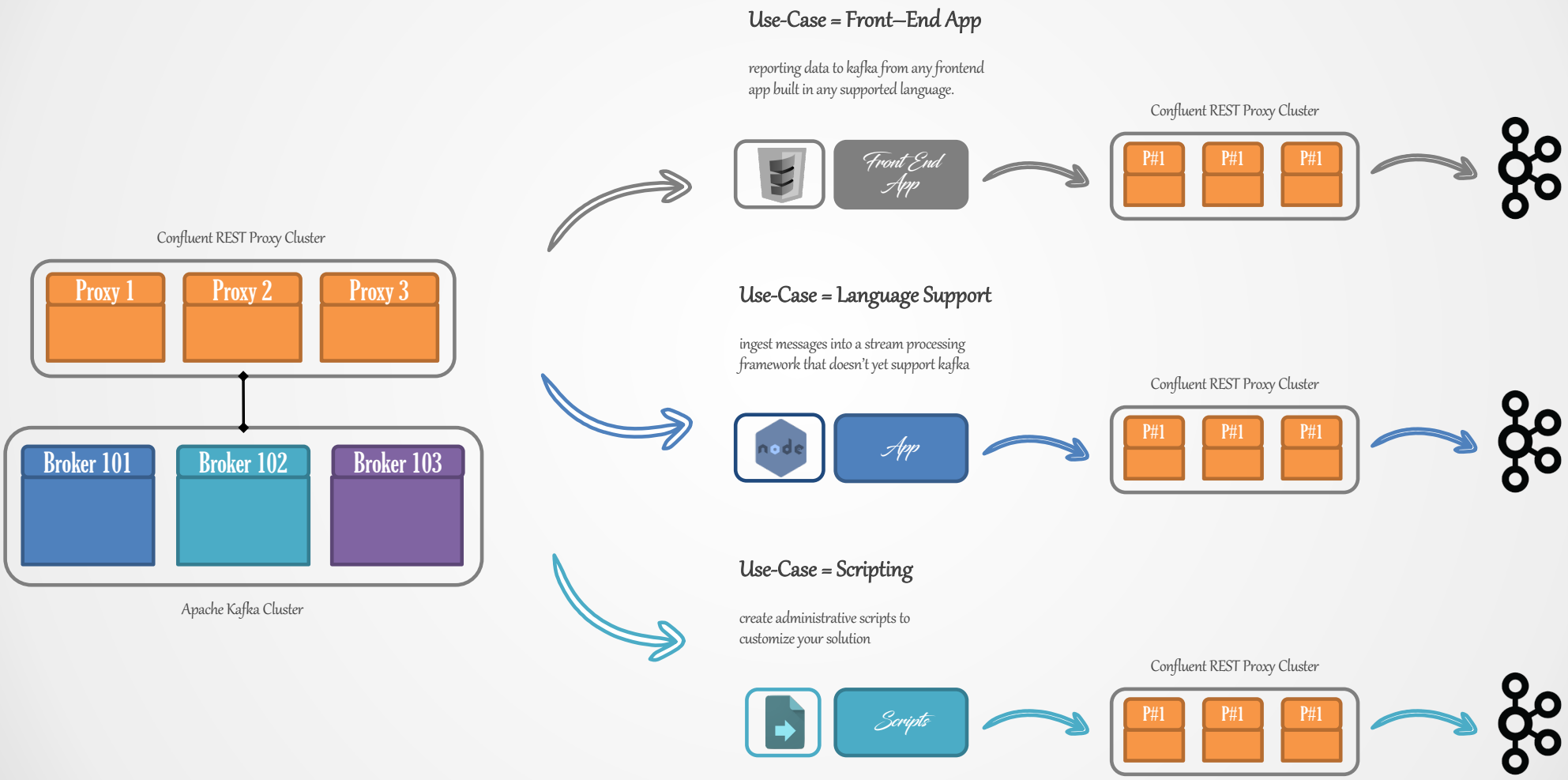


Our Clients, send HTTP Requests and wait for responses

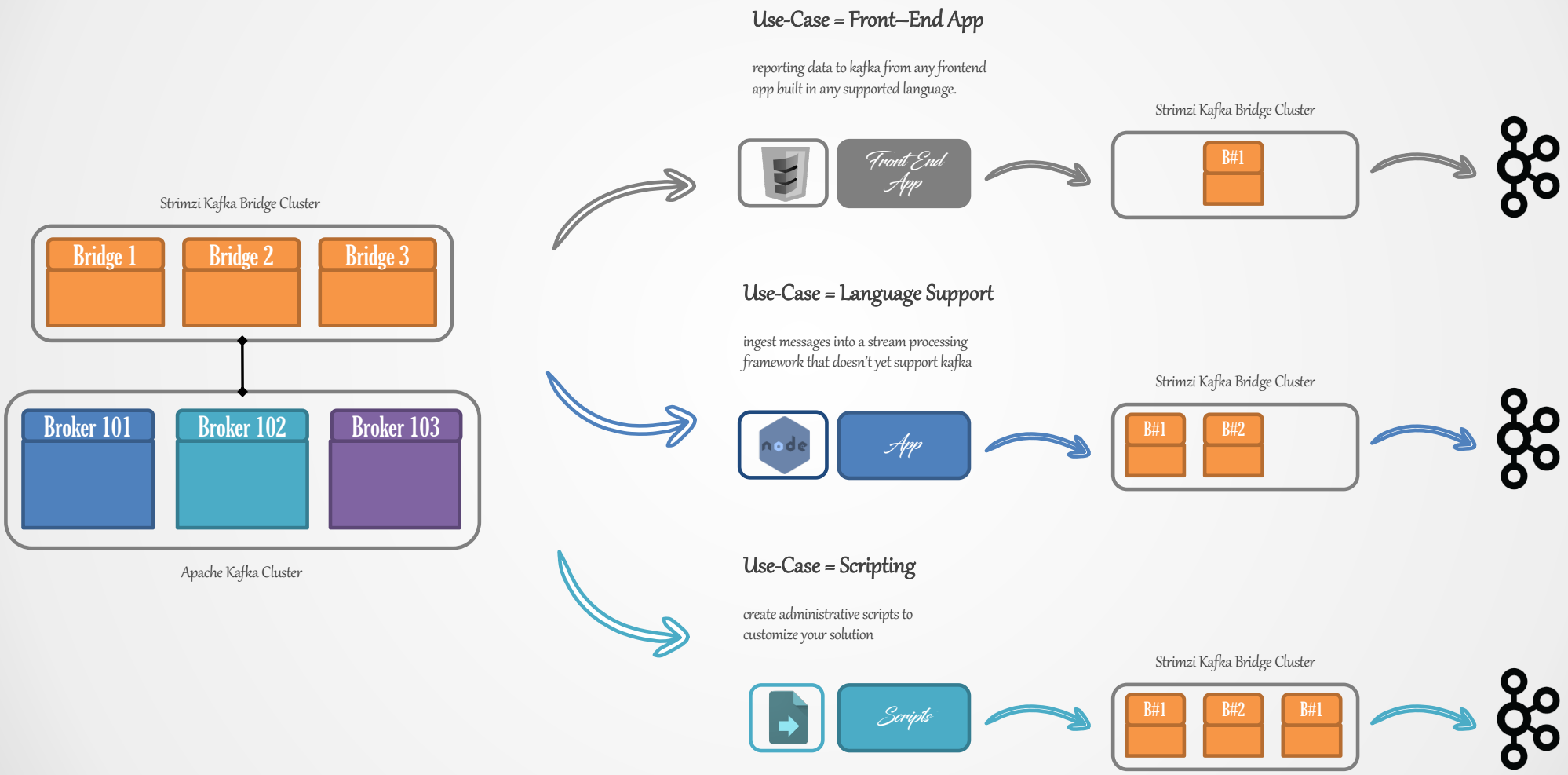
Response: When the Rest API has what it needs, it sends back a response to the clients. This would typically be in JSON or XML format.



Confluent REST Proxy [Use-Cases]

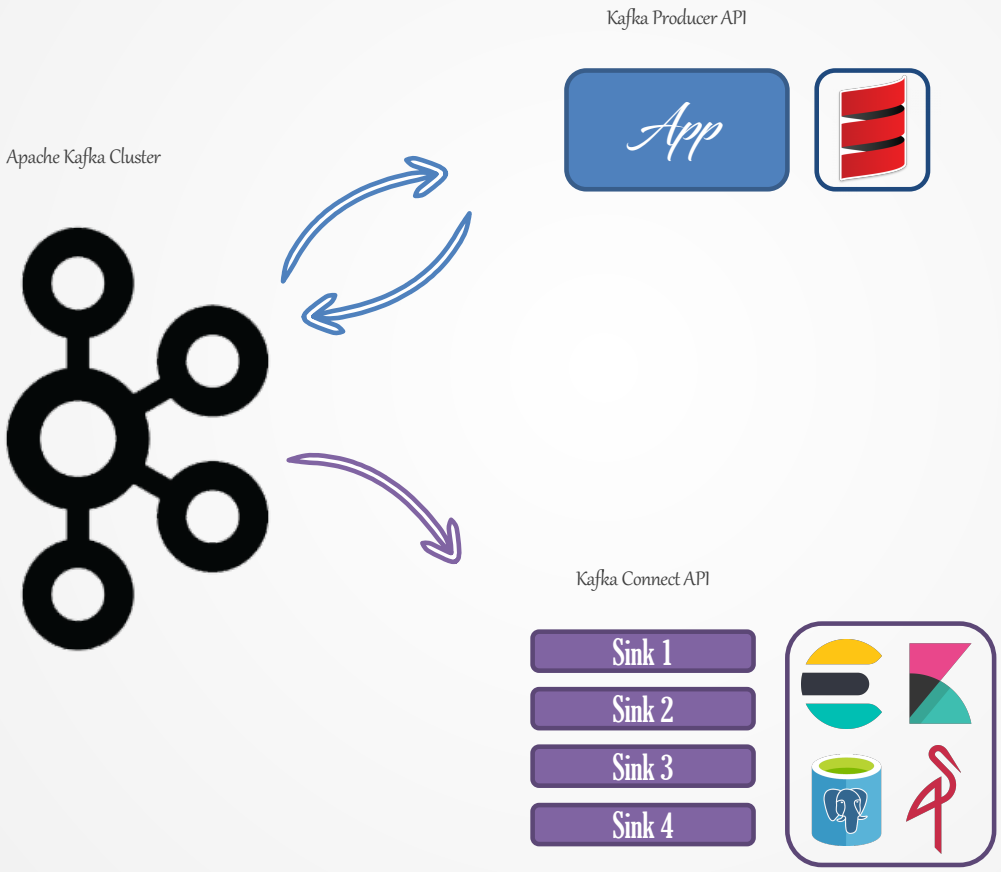


Strimzi Kafka Bridge

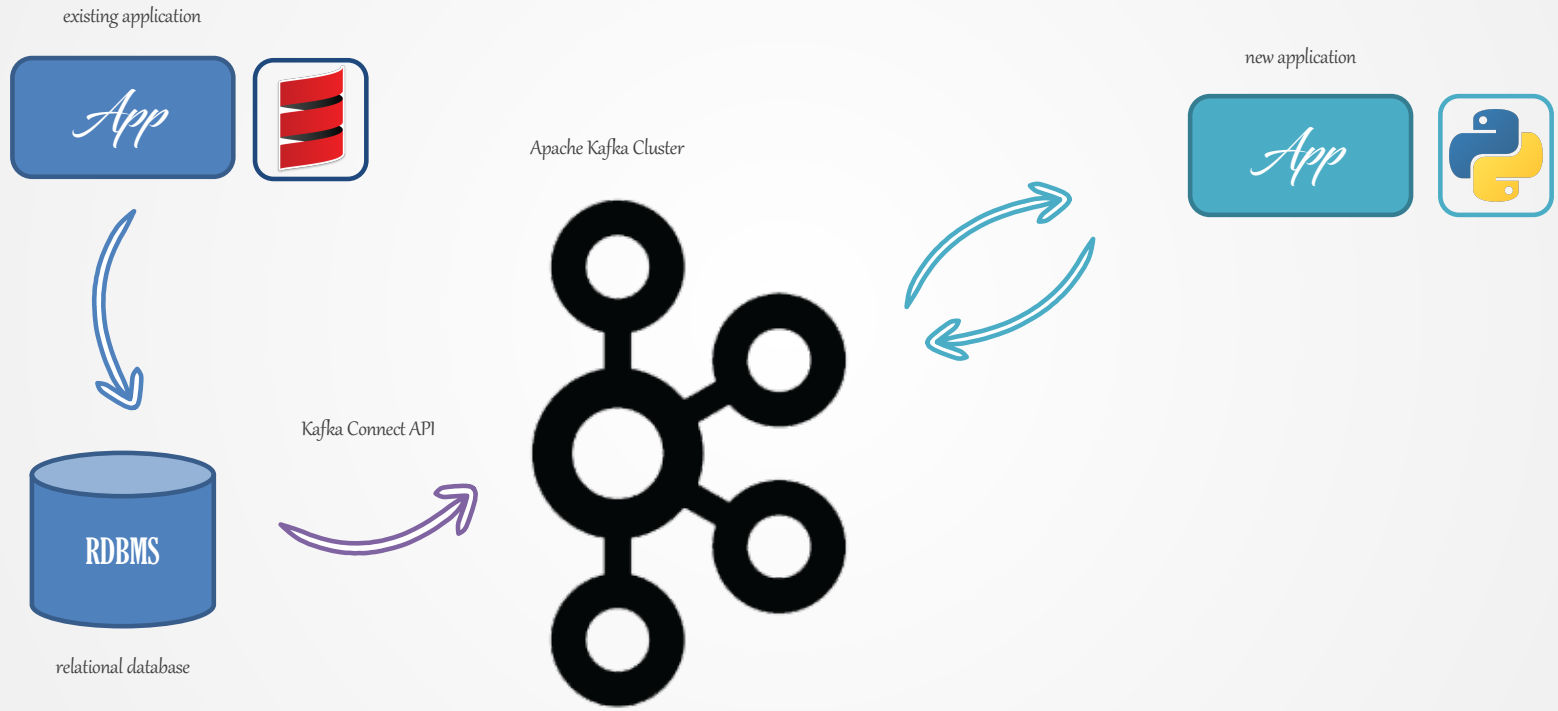



You only live once, but if you do it right, once is enough.
Mae West
quotefancy

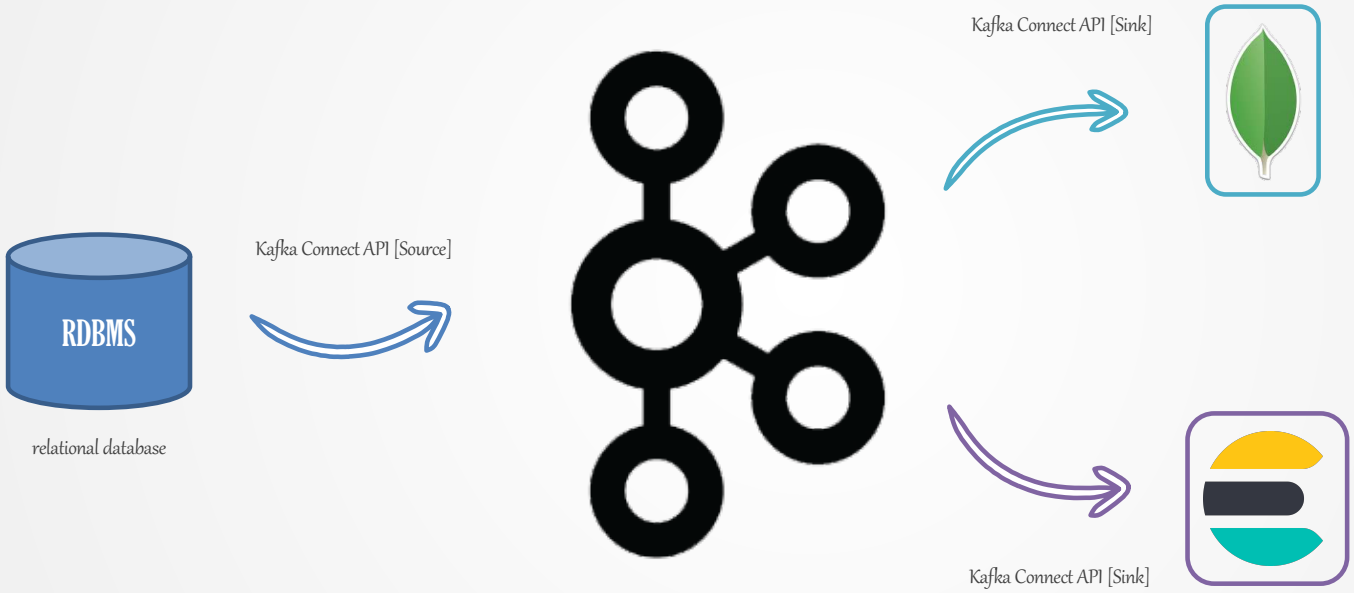
Kafka Connect [Use-Case] = Writing into Data Stores



Kafka Connect [Use-Case] = Application Communication



Kafka Connect [Use-Case] = Data Streaming Pipeline




ONEWAY
SOLUTION