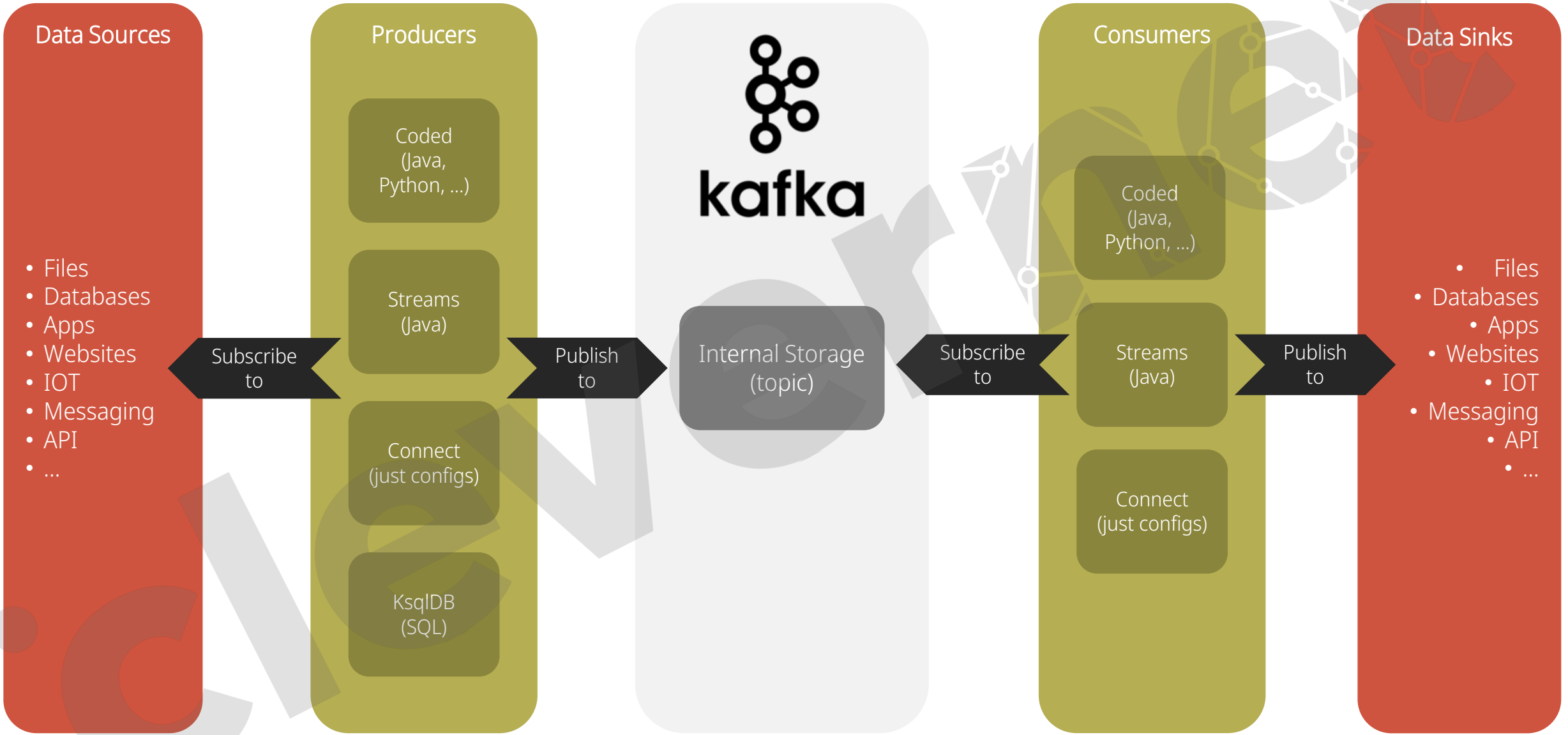
kafka fundamentals

course overview



kafka fundamentals

installation & setup

installation & setup

Virtualbox VM

- regular user: kafkauser / kafkauser
- root user: root / kafkaroot
- software:
  - java 11
  - confluent platform
  - rsync
  - python
  - mysql
    - user: root / kafkaroot
- port forwarding:
  - 2222 - SSH
  - 8081 - Schema Registry
  - 8082 - Kafka REST API
  - 8083 - Connect REST API
  - 9021 - Control Center
  - 9092 - Kafka Broker
- connections:
  - SSH - ssh kafkauser@127.0.0.1 -p 2222 (password: kafkauser)
  - RSYNC - rsync --rsh='ssh -p2222' test.py kafkauser@127.0.0.1:/home/kafkauser (password: kafkauser)

installation & setup

Local machine

- java client
  - gradle
  - java 11
- python client
  - C client *librdkafka*: https://docs.confluent.io/3.1.1/installation.html#c-c
  - python client: pip install confluent-kafka

Common SSH commands

- start/stop kafka services
  - confluent local services start / stop
- services status
  - confluent local services status
- start/stop a specific service
  - confluent local services <service-name> start/stop
- stop and cleanup installation
  - confluent local destroy

Browser access

- Control Center : http://localhost:9021

kafka fundamentals

what is Apache Kafka - an overview

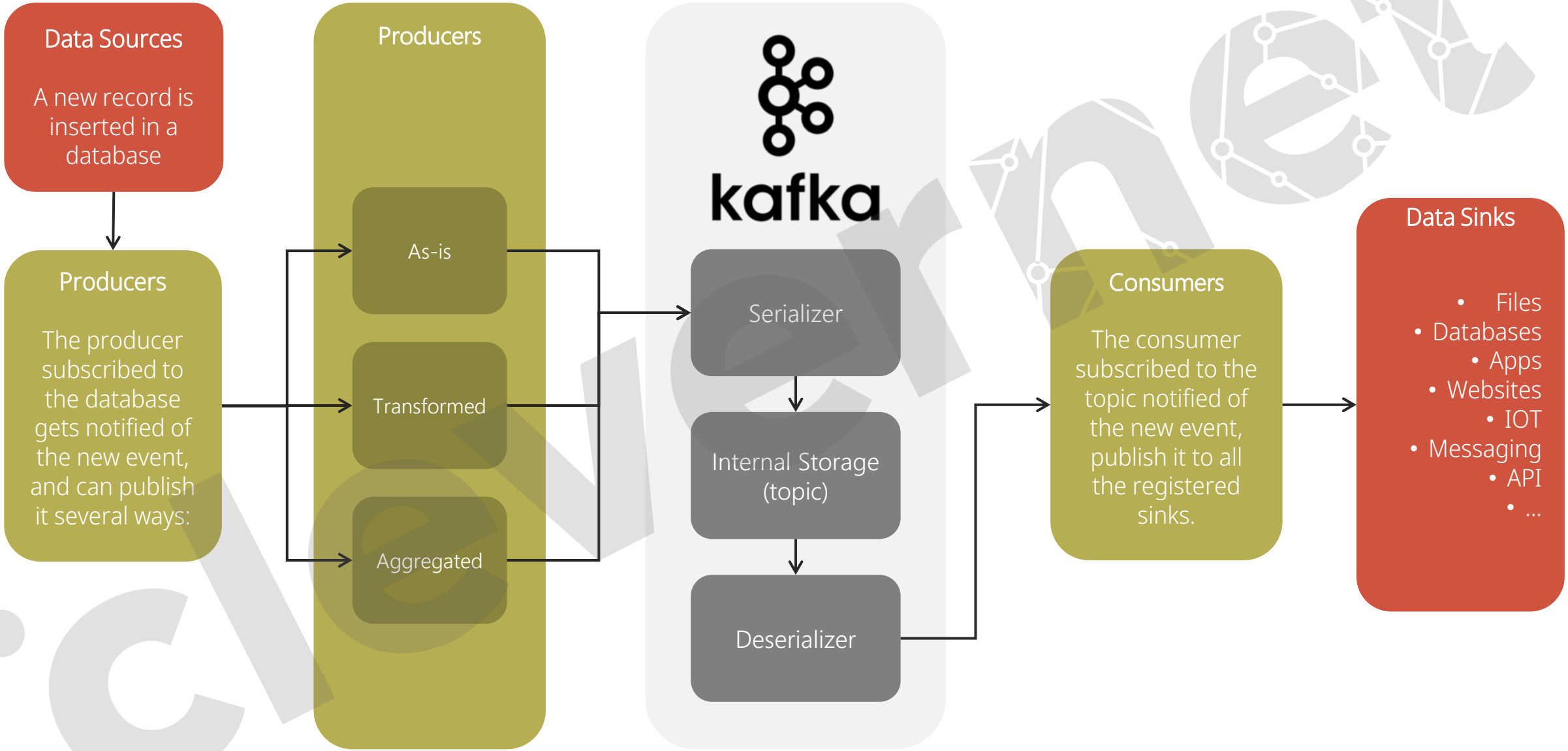
what is apache kafka

what is

- Apache Kafka is an event driven platform used to collect, store and process real time data streams, at scale.
- What is an event? Is a thing that has happened, combined with a description of what happened.
- A combination between Notification + State
- State is usually less than a megabyte, usually serialized in some standard format like JSON or AVRO
- Kafka has a data model for an event, a key/value pair. The value is going to be the state, while the key is usually an identifier (not a unique identifier like a primary key), like a user or an order.

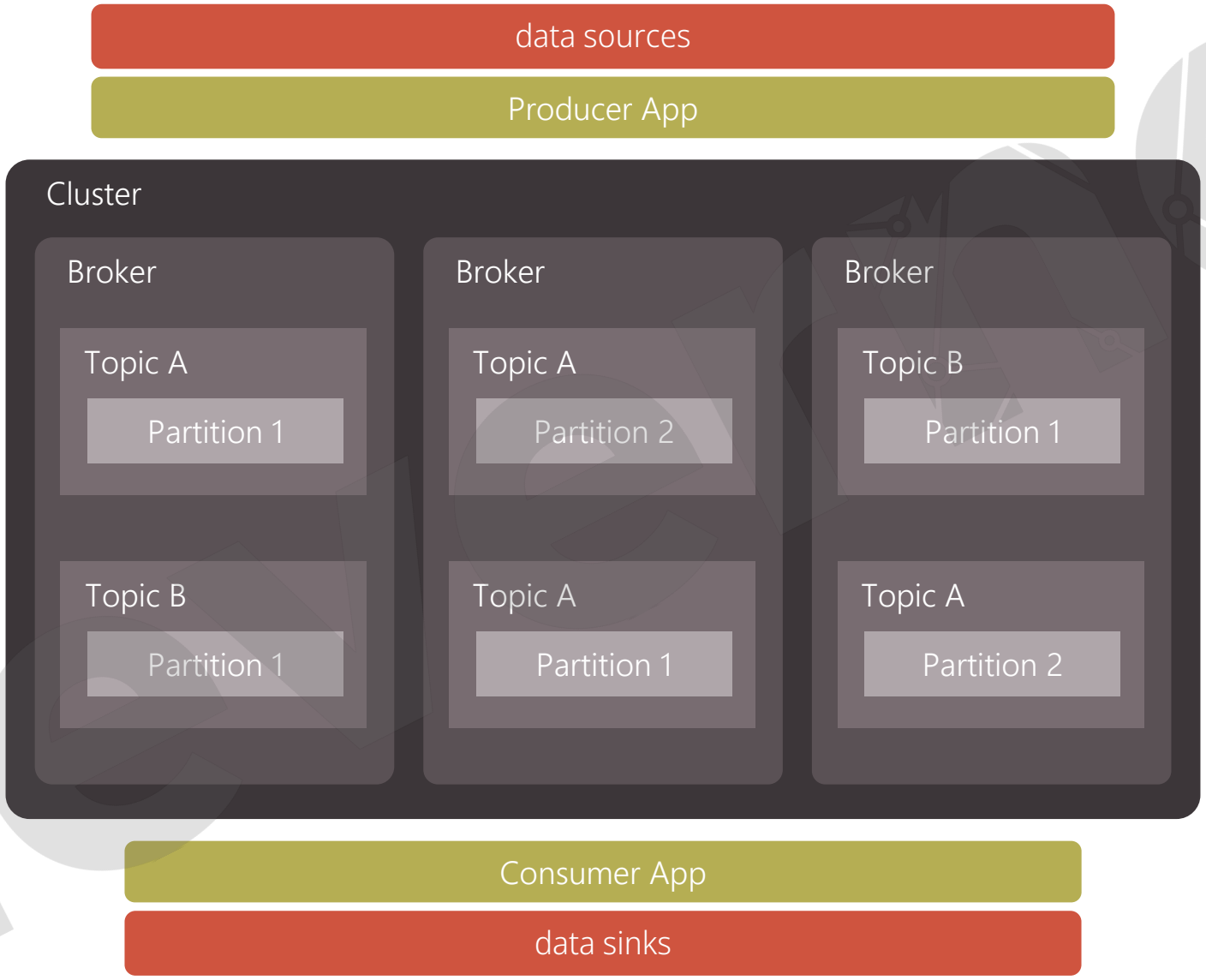
what is apache kafka

event streaming workflow



what is apache kafka

As new data is produced in the source, the producer fetches and places it in one or more topics.



Each machine in the cluster is called a broker.

A broker can have one or more topics.

A topic can have one or more partitions, which can live on different brokers (scalable).

A broker contains backups of other brokers (fault-tolerant).

As new data arrives to the topic, kafka sends it to all the consumers that subscribed to it, which in turn pass the data to the sink.

what is apache kafka

brokers

- The name given to the computer, instance or container running a kafka process.
- Host some set of the partitions, handles incoming events to write to those partitions and read events from them.
- Manage replication of partitions.

topics

- Primary component for storage in kafka.
- You can think of it as a database table, a container that holds similar data.
- They are append-only
- Can only seek by offset (go to line #10, and read from there), not indexed (I want lines 3, 7, 12 and 20)
- It's immutable (you can't change it after it happened)
- Topics can contain duplicated data from other topics: imagine a topic that contains all the messages from a log, and another that only contains the error messages in that log. This second topic is a filtered version of the first.
- Messages inside topics can expire by age or by the topic's size.
- Topic's are files stored on disk.

what is apache kafka

partitions

- Topics are broken down into smaller components called partitions.
- When a message is written to a topic, it's actually stored in one of the topic's partitions.
- The partition where the message is routed to is based on the key of that message. The nice thing about this is that it allows kafka to guarantee that messages having the same key always land in the same partition and therefore are always in order.
- If a message has no key, it's distributed round-robin across all partitions.

replications

- Brokers are susceptible to failure, so their partition data must be replicated to other brokers. These are called 'follower replicas', while the main is called 'leader replica'. So every partition has one leader and n+1 followers.
- Usually, when reading/writing data from/to a partition, you're talking with the leader. After that the leader and the followers work together to get replication done.
- Most of the time, you don't have to think about this as a developer.
- Every component in the kafka platform that is not a broker is either a producer, a consumer, or both. Producing and consuming is how you interface with the cluster.

kafka

what is apache kafka

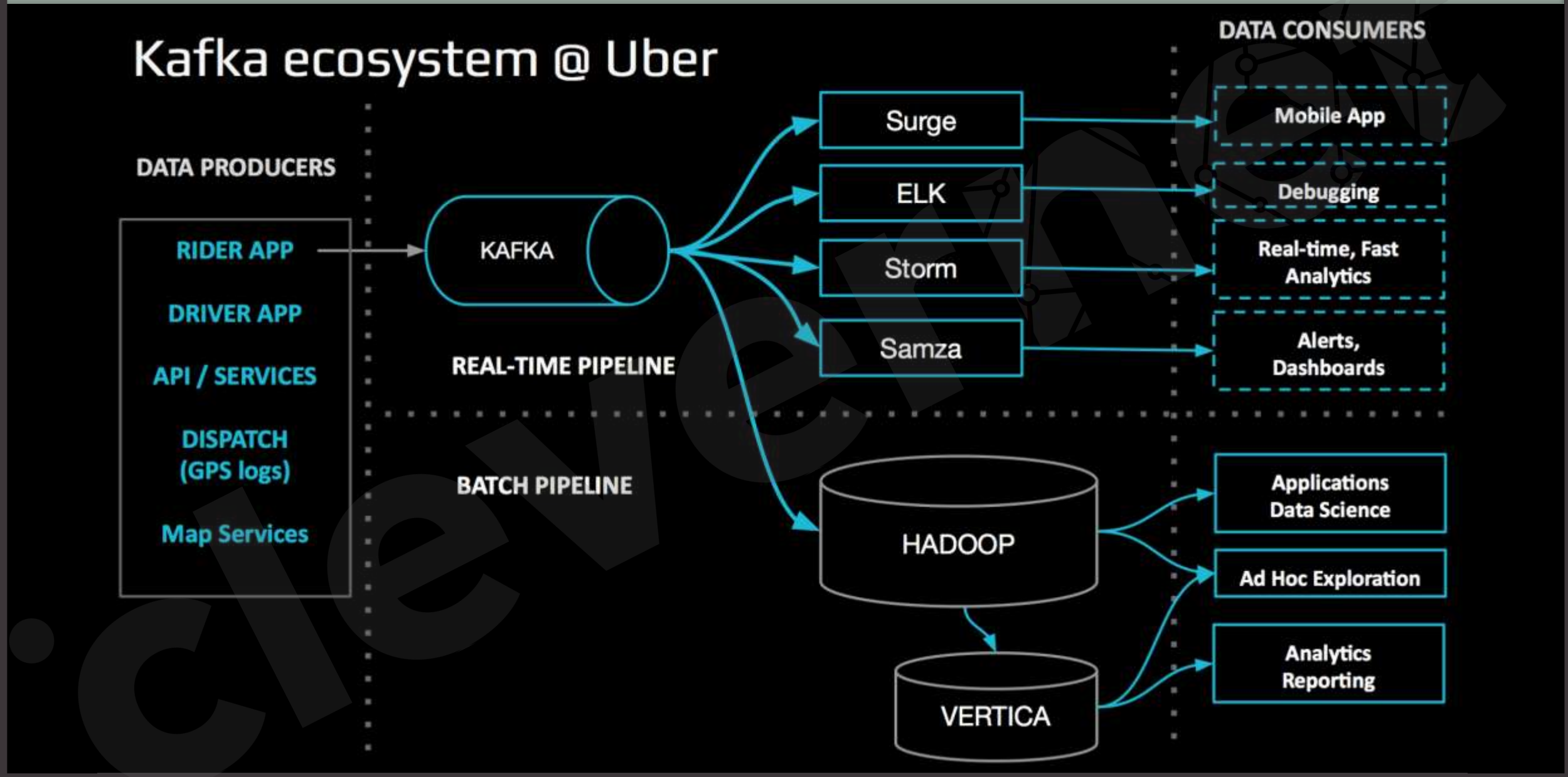
producers

- In Java, there's a class called KafkaProducer that you can use to connect to the cluster.
- There's another class called ProducerRecord that holds the key/value pair that you want to send to the cluster (the message).
- It's the producer that decides to which partition to send the message.

consumers

- Similar to producers, there's a class called KafkaConsumer to connect to the cluster.
- That connection is then used to subscribe to one or more topics. When there are messages available, they comeback as a collection called ConsumerRecords, where each element (the key/value pair) is a ConsumerRecord.
- Consumers can be scaled. If you have the same consumer program running in two different machines, that will automatically trigger in Kafka a rebalancing process to attempt to fairly distribute partitions between the two machines. This is a Consumer Group.
- Example: If you have a topic with 2 partitions and two consumers running, each consumer will get the data from a single partition. If you deploy a third consumer it will be idle because there's no partition left to assign.

what is apache kafka



kafka fundamentals

quick practical examples

Let's Play?

python client

Check and run the code in the *python/kafka_course/getting_started* folder

Keep in mind that you'll need the *librdkafka* library and the *confluent-kafka* python package (see slide 5).

- Run the producer with: *python producer.py getting_started.ini*
- Run the consumer with: *python consumer.py getting_started.ini*

Let's Play?

java client

Check and run the code in the *java/kafka/getting-started* folder

Keep in mind that you'll need to have *gradle* and Java 11 installed.

If you make changes to the code:

1. Run *gradle build*
2. Run *gradle shadowJar*

- Run the producer with: *java -cp build/libs/getting-started-0.0.1.jar examples.ProducerExample getting-started.properties*
- Run the consumer with: *java -cp build/libs/getting-started-0.0.1.jar examples.ConsumerExample getting-started.properties*

Let's Play?

console producer

let's play?

what we'll do

- create a topic
- publish data into it

1. Create topic named *john*

1. Boot up the VM
2. Log in using Putty/Terminal
3. Type *confluent local services start*
4. Open *http://localhost:9021* in the browser
5. Type *cd confluent-7.1.0/bin/*
6. Type *./kafka-topics --create --bootstrap-server localhost:9092 --replication-factor 1 --partitions 1 --topic john*
7. Make sure it's been created: *./kafka-topics --list --bootstrap-server localhost:9092*

2. Start producer that will stream data written in the console

1. Type *./kafka-console-producer --bootstrap-server localhost:9092 --topic john*

3. Open another console to consume the data

1. Type *cd confluent-7.1.0/bin/*
2. Type *./kafka-console-consumer --bootstrap-server localhost:9092 --topic john --from-beginning*

4. Type whatever you want on the producer console and see it appear on the consumer console

5. Press *control+c* on both consoles when you're done

Let's Play?

file producer

let's play?

what we'll do

- a more real life example
- use a built-in connector to monitor files in a folder and output those changes to another file
- use a consumer to see it happen in real time

1. Install a connector and create files

1. Type *confluent-hub install jcustenborder/kafka-connect-spooldir:2.0.64*
2. Create two folders: *inputcsv* and *outputcsv*
3. Inside the *inputcsv* folder, create a *users.csv* file and add some content
4. In Control Center, go to Connect and click the *Upload connector config file*. Use the *SpoolDirCsvSourceConnector_config.properties* file.
5. If the connector is successfully started, go to Topics and you should see a *csv* topic with the contents of the file.

2. On a second console, create a consumer for the topic

1. Type *cd confluent-7.1.0/bin*
2. Type *./kafka-console-consumer --bootstrap-server localhost:9092 --topic csv --from-beginning*

3. Create a new csv file and place it in the *inputcsv* folder

4. Watch the contents of the file appear in the consumer console

kafka fundamentals

Kafka Streams

what is

- Functional Java API that give access to filtering, grouping, aggregating, joining and more tools.
- It's mostly used on top of Consumers, ex: we want to group all the messages from the past hour and filter them.
- It's scalable and fault tolerant. The scale (and infrastructure) will be the same as the Consumer Group you're *streaming*. Streams is a library that sits on top of it, not besides it.
- It's specially useful for state management: the state of your operations lives in memory, is persisted to disk and also to the topic you're working, inside the kafka cluster.

kafka streams

quick example

code to only consume certain messages based on a filter (without using Streams)

```
public static void Main(String[] args) {
  try (Consumer<String, Widget> consumer = new KafkaConsumer<>(consumerProperties()); \
Producer<String, Widget> producer = new KafkaProducer(producerProperties())) {
    consumer.subscribe(Collections.singletonList("widgets"));
    while (true) {
      ConsumerRecords<String, Widget> records = consumer.poll(Duration.ofSeconds(5));
      for (ConsumerRecord<String, Widget> record: records) {
        Widget widget = record.value();
        if (widget.getColour().equals("red") {
          ProducerRecord<String, Widget> producerRecord = new ProducerRecord<>("widgets-red", record.key(), widget);
          producer.send(producerRecord, (metadata, exception)-> { ... });
        }
      }
    }
  }
  ...
}
```

code to only consume certain messages based on a filter (using Streams)

```
final StreamsBuilder builder = new StreamBuilder();

builder.stream("widgets", Consumed.with(stringSerde, widgetsSerde))
       .filter((key, widget) -> widget.getColour.equals("red"))
       .to("widgets-red", Produced.with(stringSerde, widgetSerde));
```

kafka streams

kstream topology

- The topology is a DAG (directed acyclic graph)
- Defining a kafka stream topology:

```
StreamBuilder builder = new StreamBuilder();
KStream<String, String> firstStream = builder.stream(
     inputTopic,
     Consumed.with(Serdes.String(), Serdes.String())
);
```

datatype of the key and value

- Applying transformations (typical functional programming):

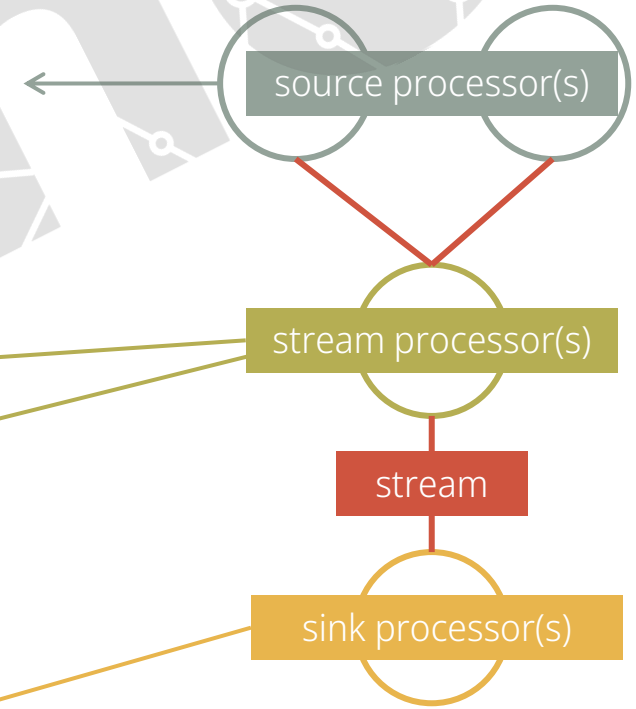
```
mapValues(value -> value.substring(5))
map((key, value) -> ...)
```

whenever possible, mapValues is preferred since it's more optimized

```
filter((key, value) -> Long.parseLong(value) > 1000)
```

- Writing transformed events:

```
to(outputTopic, Produced.with(Serdes.String(), Serdes.String()));
```



Let's Play?

- The topology is a DAG (directed acyclic graph)
- Defining a ktable topology:

```
StreamBuilder builder = new StreamBuilder();
KTable<String, String> firstKTable = builder.table(
     inputTopic,
     Materialized.with(Serdes.String(), Serdes.String())
);
```

datatype of the key and value

- Applying transformations (typical functional programming):

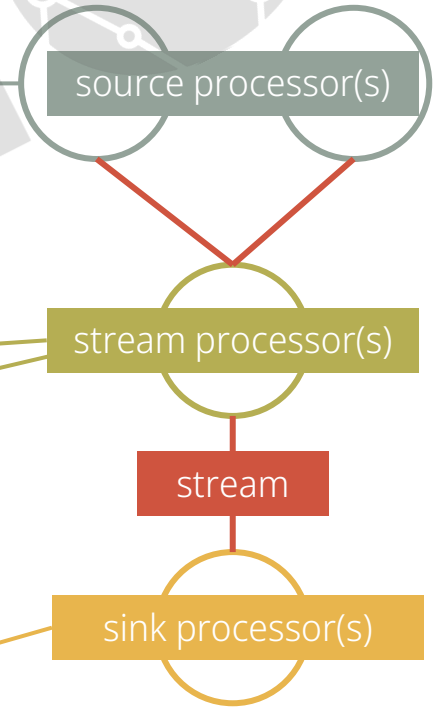
```
mapValues(value -> value.substring(5))
map((key, value) -> ...)
```

whenever possible, mapValues is preferred since it's more optimized

```
filter((key, value) -> Long.parseLong(value) > 1000)
```

- Writing transformed events:

```
to(outputTopic, Produced.with(Serdes.String(), Serdes.String()));
```



Let's Play?

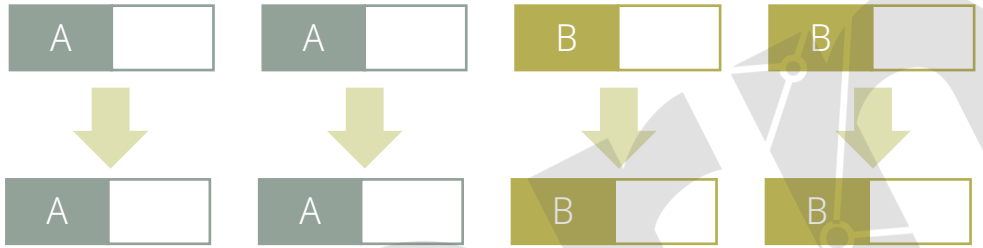
kafka streams

difference between kstream and ktable

event handling

- Kstream:

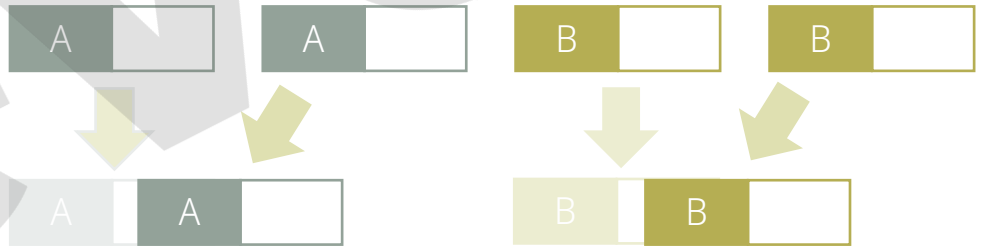
input: four events with duplicated keys



output: four events with duplicated keys

- Ktable:

input: four events with duplicated keys



output: if the event coming in has a key that already exists in the table, the new event will update the existing event.

unlike Kstreams, Ktables need what is called a state store to keep record of previous keys

about

- Kafka brokers only deal with bytes, so they have no idea about it's event's datatype
- So when data gets in and out of kafka, it has to be serialized (in) and deserialized (out). We've already seen some examples:

IN: `Consumed.with(Serdes.String(), Serdes.String())`

IN: `Materialized.with(Serdes.String(), Serdes.String())`

OUT: `to(outputTopic, Produced.with(Serdes.String(), Serdes.String()));`

- Therefore, we need to specify which serializer to use.
- When data is going to be serialized/deserialized to the same datatype, we can use a Serdes:

`Serializerdeserializer`

- We can create custom Serdes:

```
Serde<T> serde = Serdes.serdeFrom(new CustomSerializer<T>, new CustomDeserializer<T>);
```

- Or use built-in ones:

`String`, `Integer`, `Double`, `Long`, `Float`, `Bytes`, `ByteArray`, `ByteBuffer`, `UUID` and `Void`

From Schema Registry:

- Avro: `SpecificAvroSerde`, `GenericAvroSerde`
- Protobuf: `KafkaProtobufSerde`
- JSONSchema: `KafkaJsonSchemaSerde`

kafka streams

join operations

- Kstream-Kstream joins
  - Combine two even streams into a new event stream
  - Join is key based
  - Records arrive within a defined window of time
  - Possible to compute a new value type
  - Keys are available in read-only mode and can be used in computing the new value
- Kstream-Ktable joins
  - Returns a new Kstream
  - Not windowed
- Ktable-Ktable joins
  - Returns a new Ktable
  - Not windowed

join types

- Kstream-Kstream joins
  - Inner: Left + Right match or null
  - Outer
    - Left + Right
    - Left + Null
    - Null + Right
  - Left outer
    - Left + Right
    - Left + Null
- Kstream-Ktable joins
  - Inner: Left + Right match or null
  - Left outer
    - Left + Right
    - Left + Null

example

```
KStream<String, String> leftStream = builder.stream(“topic-A”);
KStream<String, String> rightStream = builder.stream(“topic-B”);

ValueJoiner<String, String, String> valueJoiner = (leftValue, rightValue) -> {
     return leftValue + rightValue;
}

KStream<String, String> joinedStream = leftStream.join(
                                        rightStream,
                                        valueJoiner,
                                        JoinWindows.of(Duration.ofSeconds(10))
                                        StreamJoined.with(Serdes.String(),
                                   Serdes.String())
                                   );
```

Let's Play?

about

- So far we've talked about stateless operations (ex: map and filter), but sometimes you also need stateful operations, such as:
  - How many times has a user logged in?
  - What's the total sum of sold products?
- For all stateful operations you're required to first group by the key
- Stateful operations always return a Ktable
- Stateful operations: count, reduce, aggregate
- Reduce example:

```
KStream<String, Long> myStream = builder.stream("topic-A");

Reducer<Long> reducer = (longValueOne, longValueTwo) -> longValueOne + longValueTwo;

myStream.groupByKey().reduce(
                        reducer,
                        Materialized.with(Serdes.String(), Serdes.Long())
                    ).toStream()
                     .to("output-topic");
```

- Aggregate example:

```
KStream<String, Long> myStream = builder.stream(“topic-A”);

Aggregator<String, String, Long> characterCountAgg = (key, value, charCount) -> value.length + charCount;

myStream.groupByKey().aggregate(
                            () -> 0, characterCountAgg, Materialized.with(Serdes.String(), Serdes.Long())
                     ).toStream()
                      .to(“output-topic”);
```

- Stateful operations don’t emit results immediately, since internal caching buffers results
- Factors controlling when cache emits records:
  - Cache is full (10mb by default)
  - Commit interval (30sec by default)

Let's Play?

about

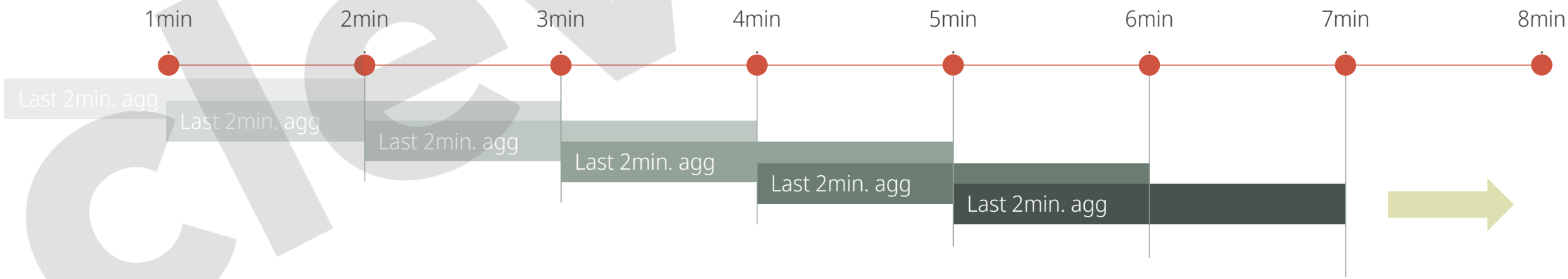
- The previous aggregate example will continue to run indefinitely so it will contain the total of all time, but if you only want the total of a given month you'll have to resort to windowing to get a snapshot in time of what you want.
- Window types:
  - Hopping
  - Tumbling
  - Session
  - Sliding

- Hopping

```
KStream<String, Long> myStream = builder.stream("topic-A");

Duration windowSize = Duration.ofMinutes(2);
Duration advanceSize = Duration.ofMinutes(1);
TimeWindows hoppingWindow = TimeWindows.of(windowSize).advanceBy(advanceSize);

myStream.groupByKey().windowedBy(hoppingWindow)
                      .aggregate(
                          () -> 0, characterCountAgg, Materialized.with(Serdes.String(), Serdes.Long())
                     ).toStream()
                      .to("output-topic");
```

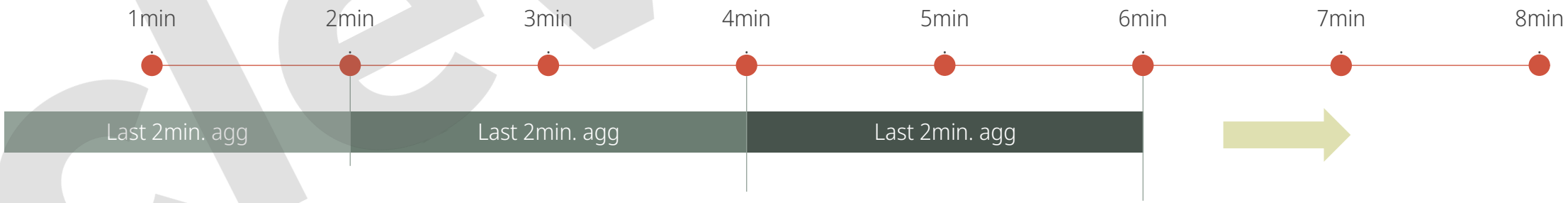


- Tumbling

```
KStream<String, Long> myStream = builder.stream("topic-A");

Duration windowSize = Duration.ofMinutes(2);
TimeWindows tumblingWindow = TimeWindows.of(windowSize);

myStream.groupByKey().windowedBy(tumblingWindow)
                      .aggregate(
                          () -> 0, characterCountAgg, Materialized.with(Serdes.String(), Serdes.Long())
                     ).toStream()
                      .to("output-topic");
```

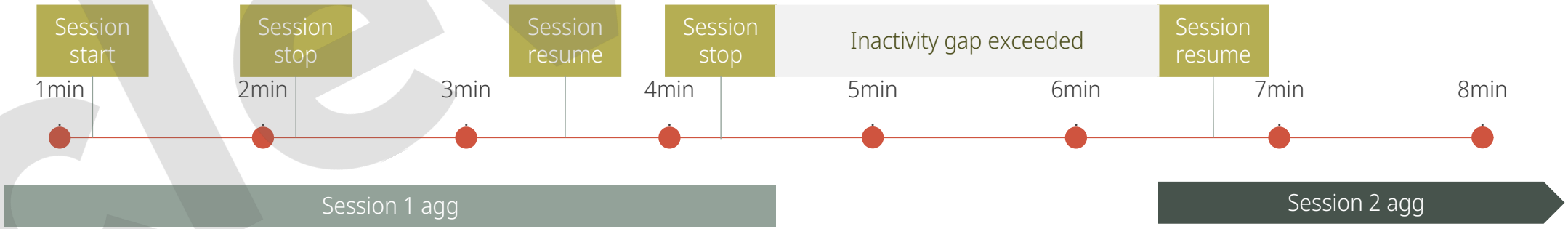


- Session

```
KStream<String, Long> myStream = builder.stream(“topic-A”);

Duration inactivityGap = Duration.ofMinutes(2);

myStream.groupByKey().windowedBy(SessionWindows.with(inactivityGap))
                        .aggregate(
                            () -> 0, characterCountAgg, Materialized.with(Serdes.String(), Serdes.Long())
                       ).toStream()
                        .to(“output-topic”);
```



- Sliding example
  - Sliding Windows are defined based on a record's timestamp, the window size based on the given maximum time difference (inclusive) between records in the same window, and the given window grace period. While the window is sliding over the input data stream, a new window is created each time a record enters the sliding window or a record drops out of the sliding window.
  - For example, if we have a time difference of 5000ms and the following data arrives:

key	value	time
A	1	8000
A	2	9200
A	3	12400

We'd have the following 5 windows:

- window [3000;8000] contains [1] (created when first record enters the window)
- window [4200;9200] contains [1,2] (created when second record enters the window)
- window [7400;12400] contains [1,2,3] (created when third record enters the window)
- window [8001;13001] contains [2,3] (created when the first record drops out of the window)
- window [9201;14201] contains [3] (created when the second record drops out of the window)

- Sliding

```
KStream<String, Long> myStream = builder.stream(“topic-A”);

Duration timeDifference = Duration.ofMillis(5000);
Duration gracePeriod = Duration.ofMillis(500);

myStream.groupByKey().windowedBy(SlidingWindows.withTimeDifferenceAndGrace(timeDifference, gracePeriod))
                      .aggregate(
                          () -> 0, characterCountAgg, Materialized.with(Serdes.String(), Serdes.Long())
                     ).toStream()
                      .to(“output-topic”);
```

Let's Play?

about

- Timestamps are a critical component of Kafka
- The kafka message format has a dedicated timestamp field
- Event-time: a producer automatically sets this timestamp if the user does not
  - Is the current time of the producer environment when the event is created
- Ingestion-time: a broker can be configured to set this timestamp when an event is appended (stored in) the topic
  - Is the current time of the broker environment
- Kafka streams uses the `TimestampExtractor` interface to get the timestamp
  - Default behaviour is to use the event timestamp (set by either the producer or the broker)
  - Default extractor is the `FailedOnInvalidTimestamp`
- If it's desired to use a timestamp embedded in the payload (the event key or value), provide a custom `TimestampExtractor`
- Time moves forward in Kafka streams by these timestamps
- For windowing operations this means that the timestamps govern the opening and closing of windows
  - How long a window remains open depend exclusively on timestamps; it has nothing to do with current server time

about

- Kafka streams has a concept of Stream Time:
  - Largest timestamp seen so far
  - Only moves forward, never backward
  - If an out-of-order event arrives, stream time remains where it is, but event is processed
  - An out-of-order event, if not inside the window anymore, it's considered a late event and is ignored

about

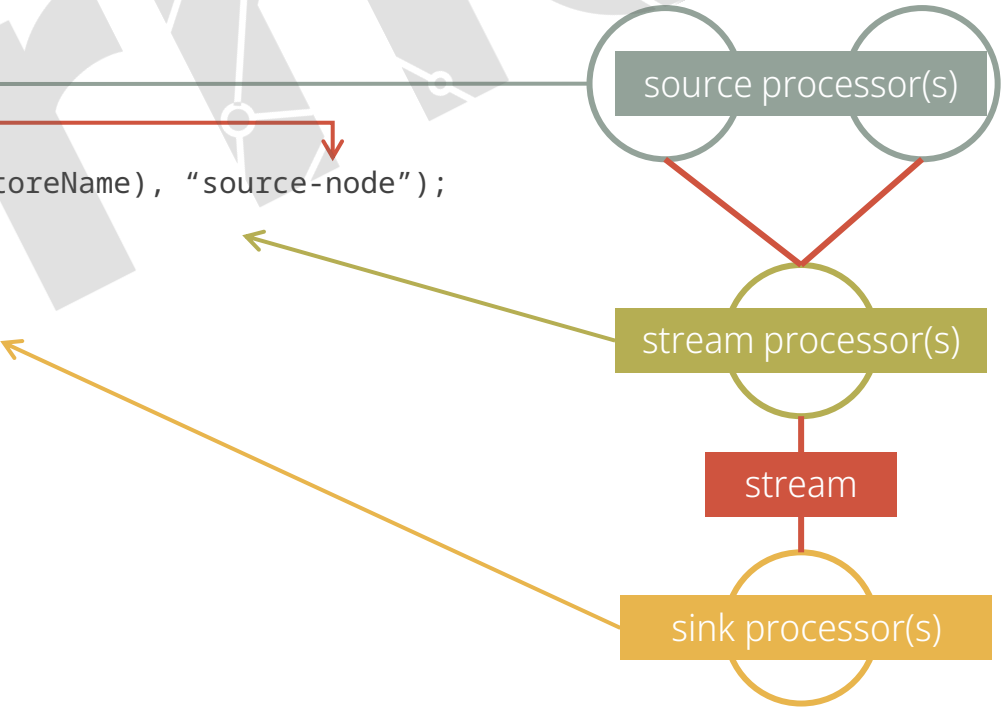
- So far we've been using the `StreamBuilder` to create the topologies, but if you need to create your own with full control over what happens, you can use the processor API. Here's a simplified skeleton of a topology using the processor API:

```
Topology topology = new Topology();

topology.addSource("source-node", "topicA", "topicB");

topology.addProcessor("custom-processor", new CustomProcessorSupplier(storeName), "source-node");

toplogy.addSink("sink-node", "output-topic", "custom-processor");
```



- Keep in mind that it's a lower level API, meaning that you'll have to take care of all the aspects of the topology, many of which are abstracted to you when using the **StreamBuilder**.

about

- Three broad areas where errors can occur:
  - Entry - consuming records - network and serialization errors
  - Processing of records - not in expected format
  - Exit - producing records - network and serialization records
- There are handlers in place for these errors to help developers decide:
  - In most cases the best approach is to aknowledge and continue
  - Sometimes there's no alternative but to shut down and restart

consuming

- Streams provide the `DeserializationExceptionHandler`
  - Default configuration is the `LogAndFailExceptionHandler`
  - Other option is the `LogAndContinueExceptionHandler`
  - You can provide a custom implementation and provide the class name via the configuration

processing

- Streams provide the `StreamsUncaughtExceptionHandler`
  - Works for exceptions not handled by Streams
  - The implementation has three options:
    - Replace the thread
    - Shutdown the individual Streams instance
    - Shutdown all Streams instances

producing

- Streams provide the `ProductionExceptionHandler`
  - Can continue processing or fail
  - Default configuration is the `DefaultProductionExceptionHandler`
    - The default option will always fail
    - For any other option you'll need your own implementation

kafka fundamentals

Kafka Connect

kafka connect

what is

- Made to connect external storage with Kafka, allowing data from those systems to be imported into topics and data from topics to be exported to those systems.
- It's a data integration system (client application that runs outside the cluster) and ecosystem (pluggable connectors).
- It's scalable and fault tolerant.
- Source connectors act as producers, sink connectors as consumers.
- If a specific connector is needed and doesn't exist in the market, one can easily build one by coding against the Connector API.
- It’s a declarative framework (you shouldn’t have to write actual code, just declare it’s configuration, in JSON format).
- Example configuration:

```
{
  “connector_class”: “io.confluent.connect.jdbc.JDBCSourceConnector”,
  “connection.url”: “jdbc:mysql://localhost:3306/demo”,
  “table.whitelist”: “sales, orders, customers”
}
```

kafka connect

sources

connect

kafka

kafka connect

kafka

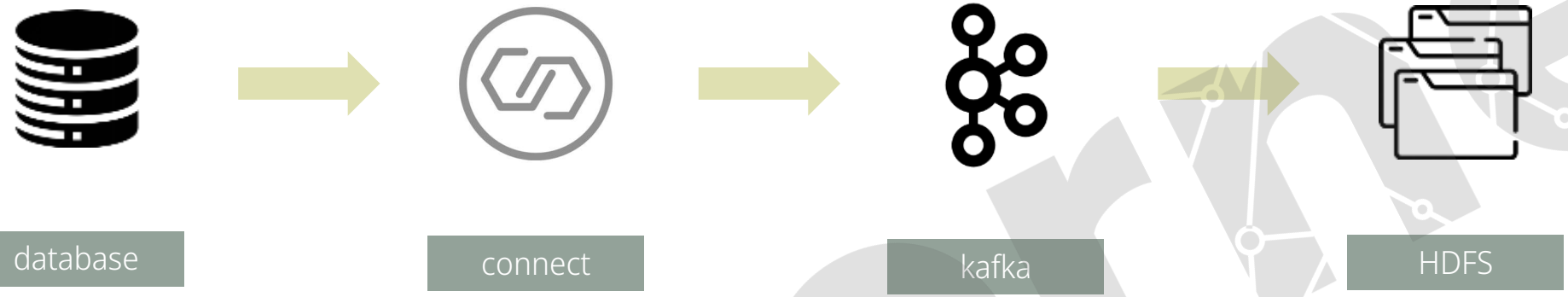
connect

sinks

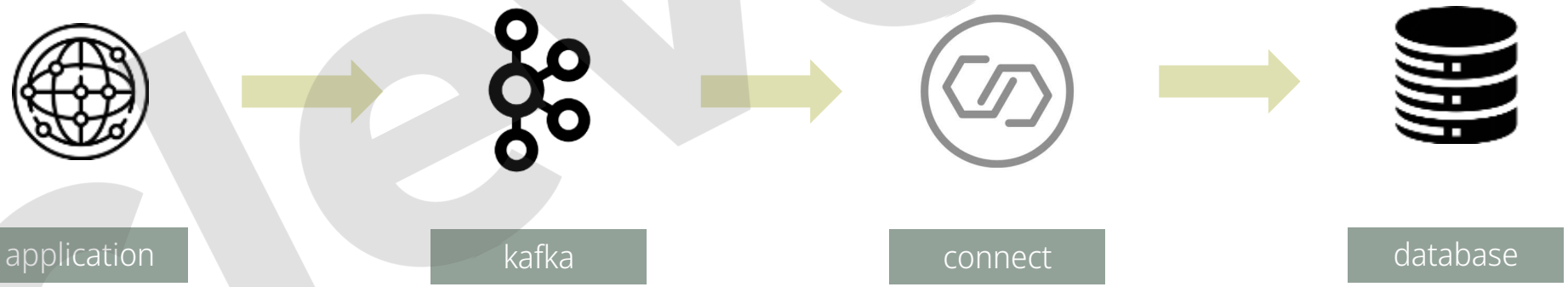
kafka connect

implementation examples

streaming pipelines



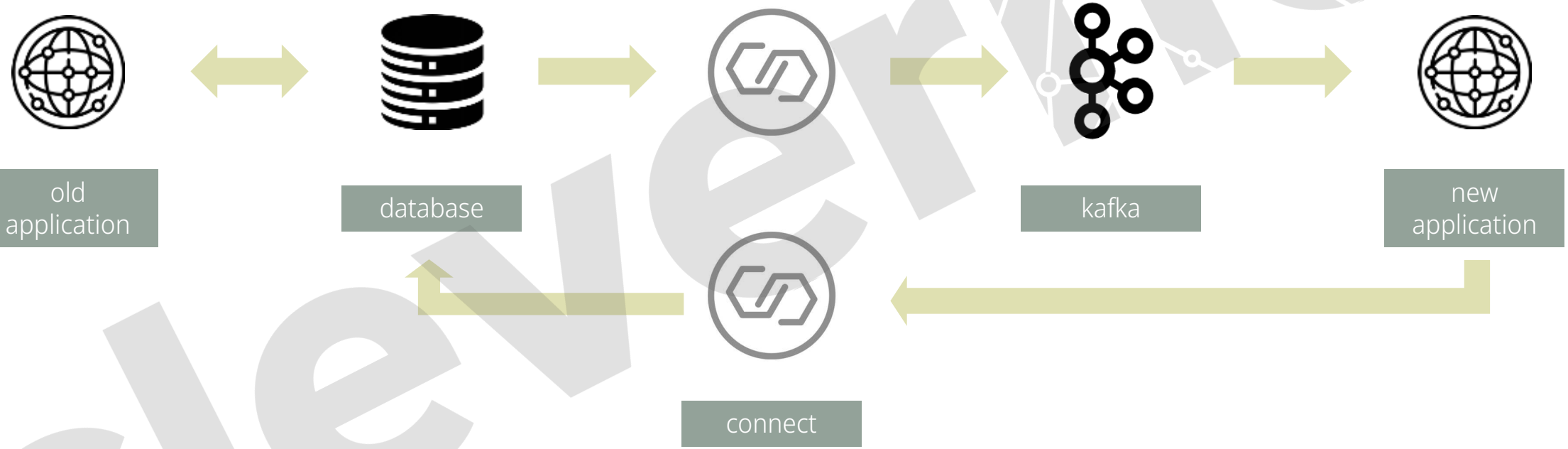
persisting application data



kafka connect

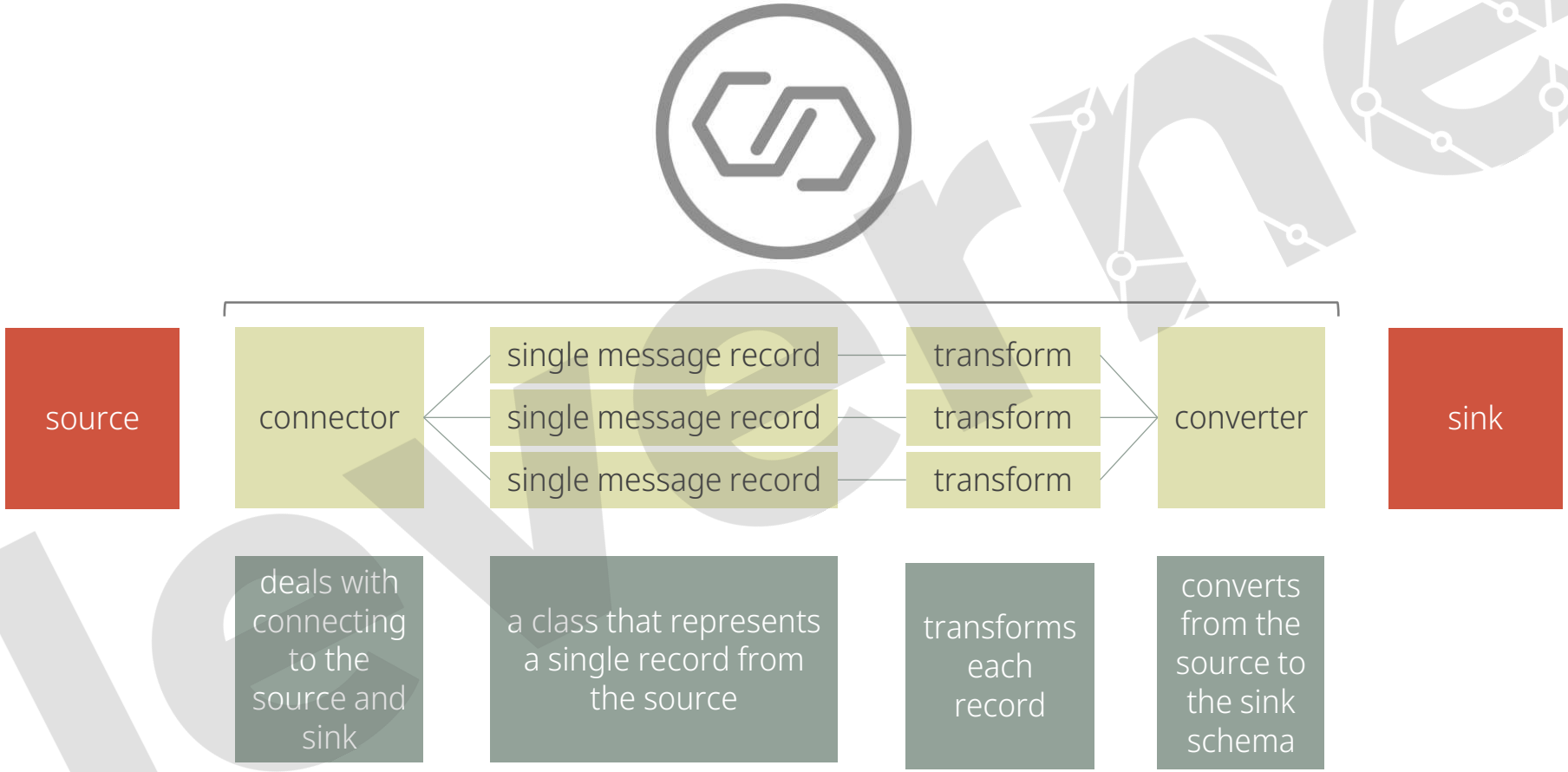
implementation examples

migrating to new systems



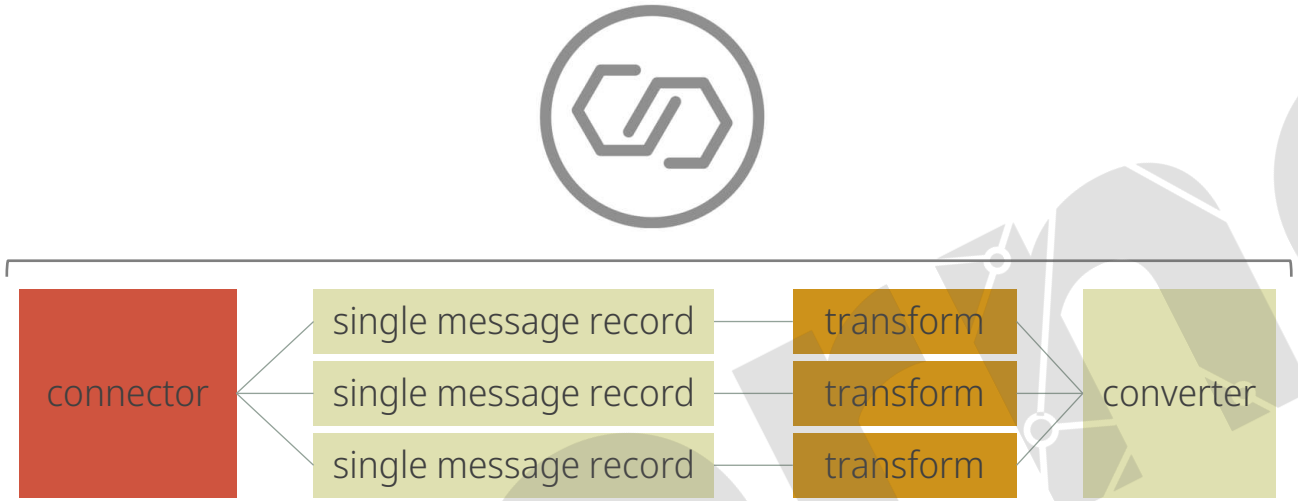
kafka connect

what's inside a connector



kafka connect

configuring a connector




```
{
  “connector_class”: “io.confluent.connect.jdbc.JDBCSourceConnector”,
  “connection.url”: “jdbc:mysql://localhost:3306/demo”,
  “table.whitelist”: “sales, orders, customers”,
  “transforms”: “addDateToTopic, labelColumn”,
  “transforms.addDateToTopic.type”: “org.apache.kafka.connect.transforms.TimeStampRouter”,
  “transforms.addDateToTopic.topic.format”: “${topic}-${timestamp}”,
  “transforms.addDateToTopic.timestamp.format”: “YYYYMM”,
  “transforms.labelColumn.type”: “org.apache.kafka.connect.transforms.ReplaceField$Value”,
  “transforms.labelColumn.renames”: “delivery_address:shipping_address”
}
```

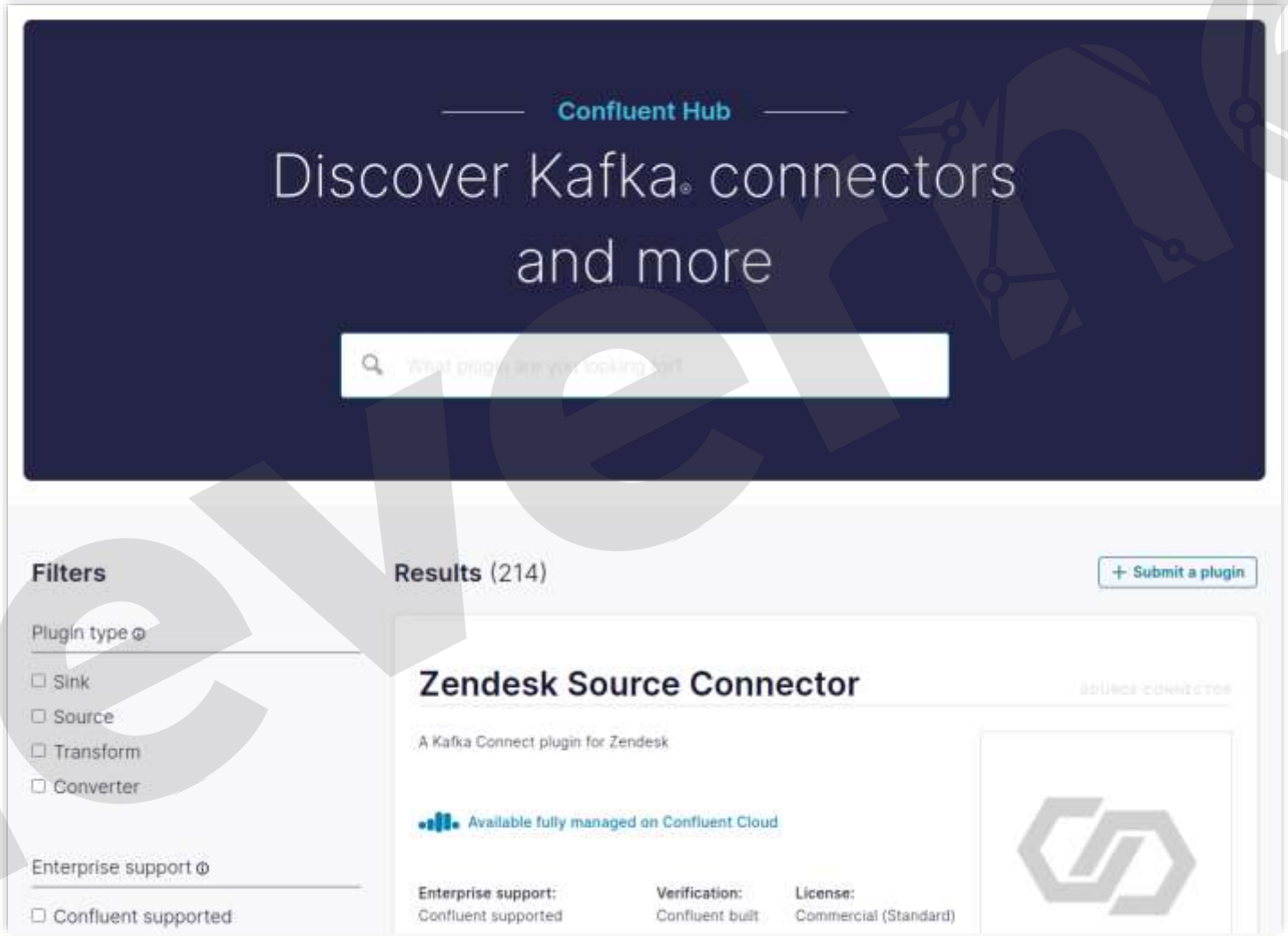
transforms list

transform 1 configs

transform 2 configs

kafka connect

confluent hub



kafka connect

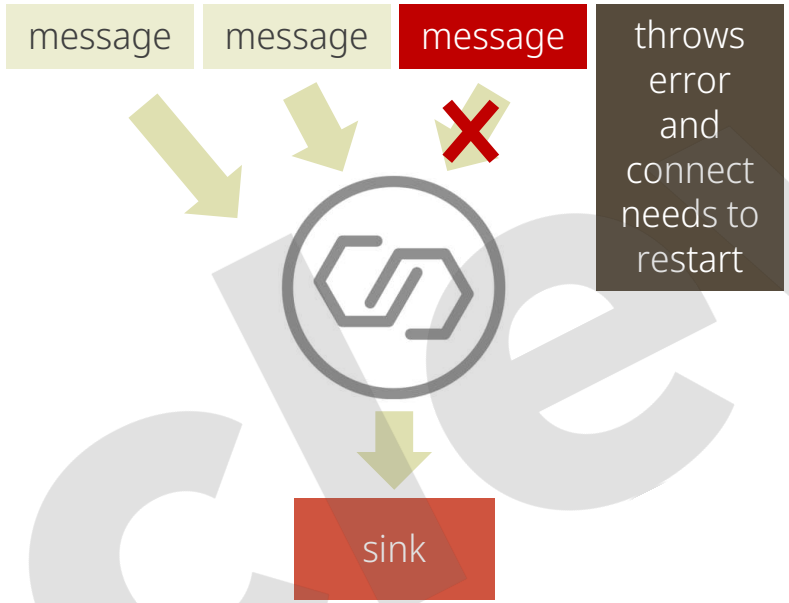
troubleshooting

source of truth: the log file

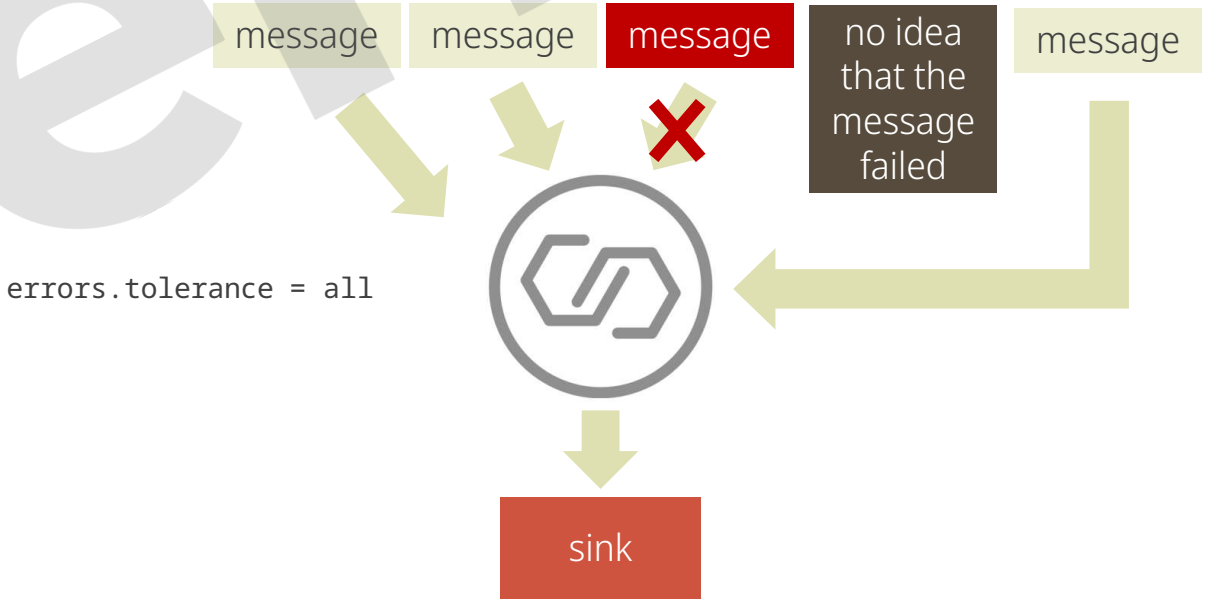
```
confluent local services connect log
```

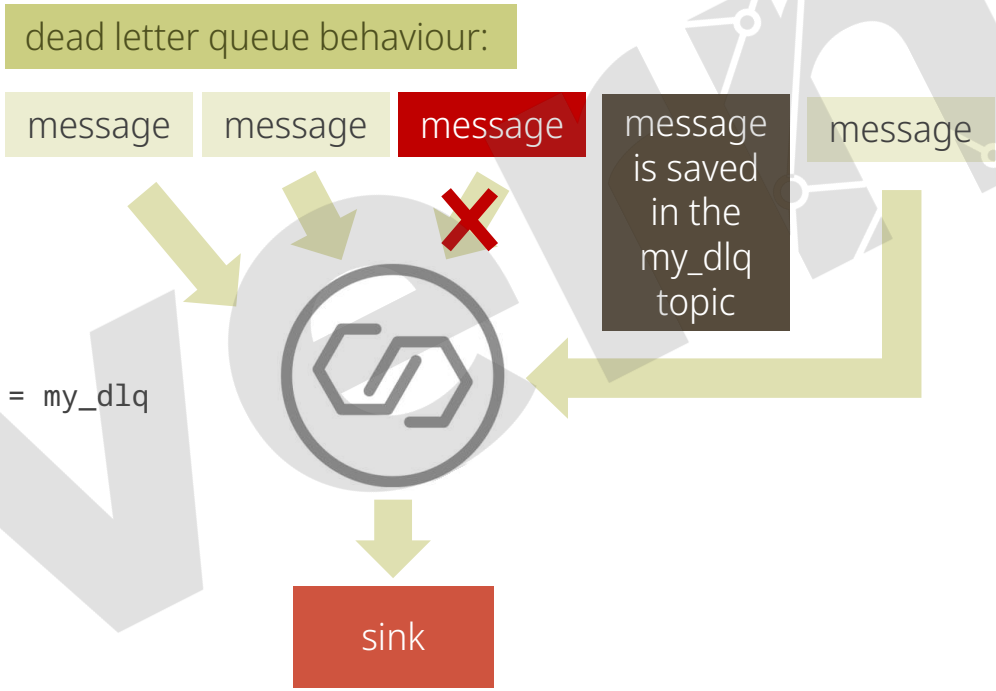
org.apache.kafka.common.errors.SerializationException: Unknown magic byte!

default behaviour:



bypass errors behaviour:






```
errors.tolerance = all
errors.deadletterqueue.topic.name = my_dlq
```

kafka connect

REST api

server info:

```
curl http://localhost:8083/
```

list of installed plugins:

```
curl http://localhost:8083/connector-plugins
```

create/update a connector:

```
curl -i -XPUT -H “Content-Type:application/json” \
http://localhost:8083/connectors/connector-name/config \
-d ‘{
  “connector_class”: “io.confluent.connect.jdbc.JDBCSourceConnector”,
  “connection.url”: “jdbc:mysql://localhost:3306/demo”,
  “table.whitelist”: “sales, orders, customers”,
  “transforms”: “addDateToTopic, labelColumn”,
  “transforms.addDateToTopic.type”: “org.apache.kafka.connect.transforms.TimeStampRouter”,
  “transforms.addDateToTopic.topic.format”: “${topic}-${timestamp}”,
  “transforms.addDateToTopic.timestamp.format”: “YYYYMM”,
  “transforms.labelColumn.type”: “org.apache.kafka.connect.transforms.ReplaceField$Value”,
  “transforms.labelColumn.renames”: “delivery_address:shipping_address”
}’
```

list of active connectors:

```
curl http://localhost:8083/connectors
curl http://localhost:8083/connectors?expand=info&expand=status
curl http://localhost:8083/connectors/connector-name/status
```

delete/pause a connector:

```
curl -XDELETE http://localhost:8083/connectors/connector-name
curl -XPUT http://localhost:8083/connectors/connector-name/pause
```

restart connector/task:

```
curl -XPOST http://localhost:8083/connector-name/restart
curl -XPOST http://localhost:8083/connector-name/task-name/restart
```

Off-topic: install the **jq** package to pretty print json in the command line

kafka connect

delete a connector:

```
curl -XDELETE http://localhost:8083/connectors/connector-name
```

restart connector/task:

```
curl -XPOST http://localhost:8083/connector-name/restart
curl -XPOST http://localhost:8083/connector-name/task-name/restart
```

pause/resume a connector:

```
curl -XPUT http://localhost:8083/connectors/connector-name/pause
curl -XPUT http://localhost:8083/connectors/connector-name/resume
```

info about tasks of a connector:

```
curl -XGET http://localhost:8083/connectors/connector-name/tasks
```

info about topics of a connector:

```
curl -XGET http://localhost:8083/connectors/connector-name/topics
```

Off-topic: install the **jq** package to pretty print json in the command line

Let's Play?

kafka connect

the Datagen connector

- We'll be using Datagen, a connector provided by confluent to automatically generate mock data into Kafka.

```
curl -i -X POST -H "Accept:application/json" \
    -H  "Content-Type:application/json" http://localhost:8083/connectors/ \
    -d '{
      "name": "datagen-credit_cards",
      "config": {
          "connector.class": "io.confluent.kafka.connect.datagen.DatagenConnector",
          "kafka.topic": "credit_cards",
          "quickstart": "credit_cards",
          "key.converter": "org.apache.kafka.connect.storage.StringConverter",
          "value.converter": "org.apache.kafka.connect.json.JsonConverter",
          "value.converter.schemas.enable": "false",
          "max.interval": 1000,
          "iterations": 20,
          "tasks.max": "1"
       }
   }'
```

- In *Control Center*, go to *Connect* , click the cluster name and then the *Upload connector config file* button
- Select the *connector_datagen-connector_config.properties* file.
- Click *Launch*
- Go to *Topics*, see the new *credit_cards* topic and browse it's messages.
- Create a consumer in the console as an alternative:
  - *cd confluent-7.1.0/bin*
  - *./kafka-console-consumer --topic credit_cards --bootstrap-server localhost:9092 --from-beginning*

Let's Play?

kafka connect

installing, configuring and using a MySQL connector

- We'll be using Debezium, a connector that allows to read/write data to/from Kafka using MySQL, MongoDB, PostgreSQL, Oracle DB, MS SQL Server, DB2 or Cassandra

```
curl -i -X POST -H "Accept:application/json" \
    -H  "Content-Type:application/json" http://localhost:8083/connectors/ \
    -d '{
      "name": "mysql-connector",
      "config": {
            "connector.class": "io.debezium.connector.mysql.MySqlConnector",
            "database.hostname": "localhost",
            "database.port": "3306",
            "database.user": "debezium",
            "database.password": "dbz",
            "database.server.id": "42",
            "database.server.name": "demo",
              "database.include.list": "my_database",
            "table.include.list": "my_users_table",
            "column.exclude.list": "password",
            "database.history.kafka.bootstrap.servers": "localhost:9092",
            "database.history.kafka.topic": "dbhistory.demo" ,
            "include.schema.changes": "true"
      }
    }'
```

- Follow the same steps as the previous exercise, but now select the *connector_mysql-connector_config.properties* file.

kafka fundamentals

Kafka ksqlDB

kafka ksqlDB

what is

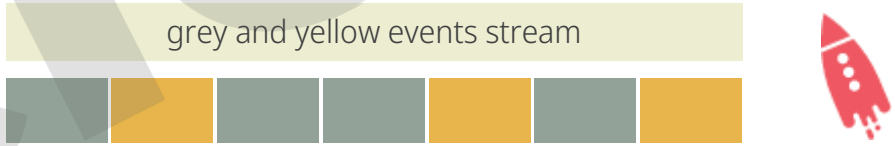
- Can be seen as an alternative to Streams
- It's a database optimized to stream processing applications.
- Runs on it's own cluster, adjacent to the Kafka cluster.
- Programs are written in a SQL-like language to query the data from the database.
- Has multiple interfaces to interact: through a REST Api, Java library, CLI or web interface.
- Can integrate with Connect, allowing to bring other datasources into ksqlDB.
- The results from the query can be pushed back to a new topic in the Kafka cluster.
- Handles joins, aggregations, filters, push and pull queries and user-defined functions.

kafka ksqlDB

filter

join

aggregate



color	count
grey	4
yellow	3

kafka ksqlDB

REST API	`http://localhost:8088`
CLI	type `ksql` in the terminal
web	available in Control Center
java	https://docs.ksqldb.io/en/latest/developer-guide/ksqldb-clients/java-client/

```
KsqlObject row = new KsqlObject()
      .put("PERSON", "John")
      .put("LOCATION", "Lisbon");

// insert into stream
client.insertInto("MOVEMENTS", row).get();

// read from stream
client.streamQuery("SELECT * FROM MOVEMENTS EMIT CHANGES;")
      .thenAccept(streamedQueryResult -> {
            RowSubscriber subscriber = new RowSubscriber();
            streamedQueryResult.subscribe(subscriber);
      }).exceptionally(e -> {
            System.out.println("Request failed: " + e);
            return null;
      });
```

Note: there are also Python, .Net and Go interfaces available.

Let's Play?

kafka ksqlDB

interfaces quick start

web

1. In Control Center, click on *ksqlDB*.
2. In the editor, type:
   ```
   CREATE STREAM MOVEMENTS (PERSON VARCHAR KEY, LOCATION VARCHAR)
     WITH (VALUE_FORMAT='JSON', PARTITIONS=1, KAFKA_TOPIC='movements');
   ```
3. The *movements* stream will show up on the right-hand sidebar.
4. Clear the editor and type:
   ```
   INSERT INTO MOVEMENTS VALUES ('John', 'Lisbon');
   INSERT INTO MOVEMENTS VALUES ('Ann', 'Sintra');
   INSERT INTO MOVEMENTS VALUES ('Peter', 'São Paulo');
   INSERT INTO MOVEMENTS VALUES ('Sophie', 'Fortaleza');
   ```
5. Look at the data in the stream by clicking the *Flow* tab, and then click on *Movements.*

CLI

1. Login via SSH/Putty
2. Type *ksql*
3. Type *show streams;* (you should see the *movements* streams we created)
4. Type:
   ```
   INSERT INTO MOVEMENTS VALUES ('John', 'Lisbon');
   INSERT INTO MOVEMENTS VALUES ('Ann', 'Sintra');
   INSERT INTO MOVEMENTS VALUES ('Peter', 'São Paulo');
   ```
5. Because we want to see all of the data that's already in the stream, and not just new messages as they arrive, we need to type *SET 'auto.offset.reset' = 'earliest';*

CLI (cont.)

6. Type *SELECT * FROM MOVEMENTS EMIT CHANGES;*
7. Insert a new row in the web and see it appear in the CLI.
8. Create a new table, type:

```
CREATE TABLE PERSON_STATS WITH (VALUE_FORMAT='AVRO') AS
  SELECT PERSON,
    LATEST_BY_OFFSET(LOCATION) AS LATEST_LOCATION,
    COUNT(*) AS LOCATION_CHANGES,
    COUNT_DISTINCT(LOCATION) AS UNIQUE_LOCATIONS
  FROM MOVEMENTS
GROUP BY PERSON
EMIT CHANGES;
```

9. Check creation with *show tables;*

REST API

1. We'll be using Postman (postman.com), a tool to quickly experiment with API calls.
2. Import the *ksqlDB.postman_collection.json* file into the program to replicate the requests.

- So far we've been manually creating the data, but ksqlDB can read/write to existing data repositories

reading from / writing to existing kafka topics

- The advantage here is that any new event produced to the topic will automatically be streamed into ksqlDB, and vice-versa.

```
CREATE STREAM people WITH(
 KAFKA_TOPIC='topic1',
 VALUE_FORMAT='AVRO'
);
```

```
CREATE TABLE departments WITH(
 KAFKA_TOPIC='topic1',
 PARTITIONS=3,
 VALUE_FORMAT='AVRO'
);
```

using connectors from kafka Connect

- Source and Sink connectors can be used in ksqlDB in the same way as with topics.

```
CREATE SOURCE CONNECTOR mysql WITH(
 'connector.class' = 'MySqlConnector',
 'database.hostname' = 'localhost',
 'table.whitelist' = 'departments');
```

```
CREATE SINK CONNECTOR mysql WITH(
 'connector.class' = 'MySqlConnector',
 'database.hostname' = 'localhost',
 'topics' = 'departments');
```

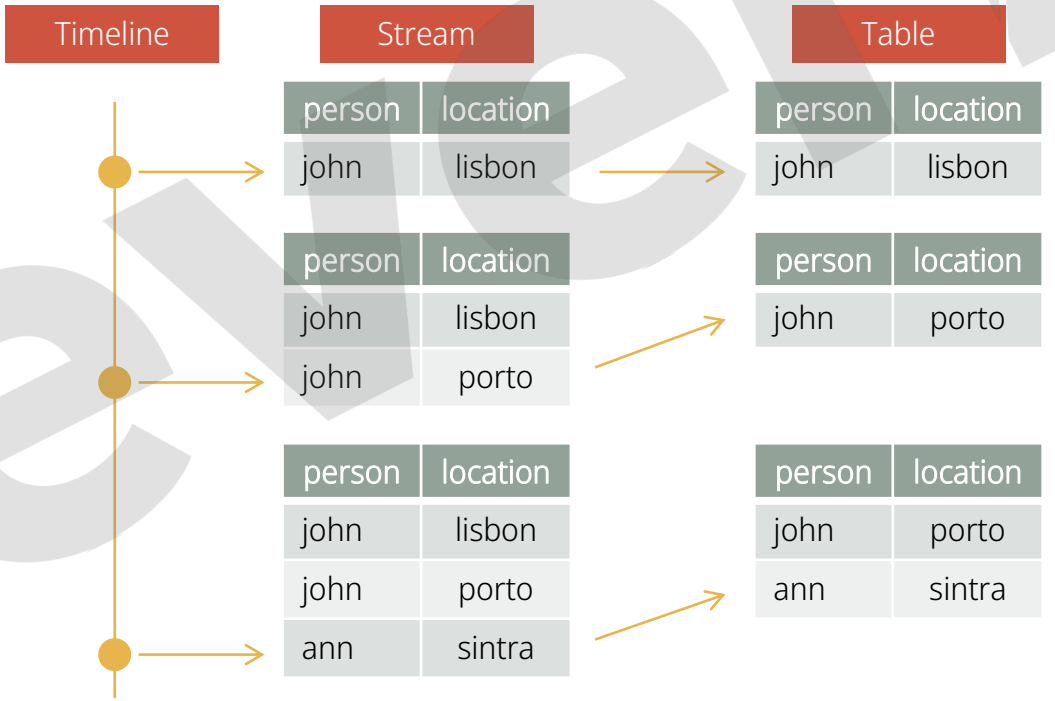
Let's Play?

about

- The following exercises are examples of typical operations made in ksqlDB.
- We'll be using the web interface, but we could use any of the other interfaces.
- Refer to the *ksqlDB-hands_on.sql* file for all the commands.
- Here's the list of exercises:
  - Filtering
  - Lookups and Joins
  - Transformations
  - Flatten nested records
  - Converting data formats
  - Merging two streams
  - Splitting two streams

about

- Whilst ksqlDB reads and writes to kafka topics, it can model how it works with that data in different ways.
- Streams will store all events, while tables will only store the latest value of a given key
- Therefore, tables require a key, streams do not.



about

- Besides filters and joins, we can also performs aggregations.
- Aggregations always return a table.
- To persist these aggregations, we can save the query inside a create table statement.
- That's what materialized views are.

Perform an aggregation

```
SELECT person, COUNT(*)
FROM movements
GROUP BY person;
```

Persist an aggregation

```
CREATE TABLE person_movements AS
 SELECT person, COUNT(*)
 FROM movements
 GROUP BY person
EMIT CHANGES;
```

Let's Play?

about

- Pull queries run once and die after outputting results.
- Push queries run forever, until they are shut down.
- The difference is the use of **EMIT CHANGES** in push queries.
- Pull queries are often used while developing, and push queries while in production.

Pull queries

```
SELECT person, COUNT(*)
FROM movements
GROUP BY person;
```

Push queries

```
SELECT person, COUNT(*)
FROM movements
GROUP BY person
EMIT CHANGES;
```

Let's Play?

kafka ksqlDB

lambda functions

about

- Used to perform operations on arrays or map (i.e. `{key:value}` pair objects) fields inside events.
- Three functions available: `TRANSFORM`, `FILTER` and `REDUCE`
- Same principles as in functional programming: map, filter and reduce.

Transform

```
CREATE STREAM transformed
  AS SELECT id,
  TRANSFORM(
    map_field,(k,v)=> UCASE(k), (k, v)=> (v * 2)
  )
FROM stream1 EMIT CHANGES;
```

For a MAP field, two lambda expressions are required: one for the key and one for the value. If no change is required for the key or the value, the lambda can just return the existing data.

Filter

```
CREATE STREAM filtered
  AS SELECT id,
  FILTER(
    numbers,x => (x%2 = 0)
  ) AS even_numbers
FROM stream2 EMIT CHANGES;
```

The filter function takes an ARRAY or MAP and returns the same type, but filled with only the elements that meet the filter's criteria, which are given in the lambda expression.

Reduce

```
CREATE STREAM reduced
  AS SELECT name,
  REDUCE(
    scores,0,(s,x)=> (s+x)
  ) AS total
FROM stream3 EMIT CHANGES;
```

For an ARRAY, the lambda takes two arguments: the ARRAY element and an accumulator. For a MAP, the lambda takes three arguments: the key, the value, and an accumulator.

Let's Play?

kafka fundamentals

Kafka Schema Registry

kafka schema registry

why the need for a schema

- Kafka takes bytes as an input and publishes them
- No data verification

- What if the producer sends bad data?
- What if a field gets renamed?
- What if the data format changes?

→ Consumers fail

kafka schema registry

- We need data to be self describable
- We need to be able to evolve without breaking consumers

in case you're wondering...

- *What if kafka brokers were verifying the messages they receive?*
- It would break what makes kafka so good:
  - Kafka doesn't parse or even read your data (no CPU usage)
  - Kafka distributes bytes, meaning that it takes bytes as an input and sends them straight to the consumer (zero copy)

- Enter the Schema Registry, that will solve this for you:
  - must agree on a common data format
  - must be able to reject bad data
  - must support a schema
  - must support schema evolution
  - must be lightweight

kafka schema registry

what is

- Standalone server process that runs on a machine external to the kafka cluster.
- It maintains a database of schemas of the data ingested into the topics (this database will actually be a kafka topic in the cluster).
- Currently supports:
  - AVRO (recommended and what we'll use) - avro.apache.org
  - JSON - json-schema.org
  - ProtoBuff (from Google) - developers.google.com/protocol-buffers/

kafka fundamentals

Kafka Schema Registry

AVRO

kafka schema registry - AVRO

an evolution of data formats

CSV

Advantages

- Easy to parse
- Easy to read
- Easy to make sense of

Disadvantages

- Data types have to be inferred and are not guaranteed
- Parsing may become tricky
- Headers may or may not exist

Relational databases

Advantages

- Data is fully typed
- Data fits in a table

Disadvantages

- Data has to be flat
- Data definitions will be different for each database

JSON

Advantages

- Data can take any form
- Widely accepted format on the web
- Can be read by any language
- Easily shared across the network (it's just text)

Disadvantages

- No schema enforcing
- Can be quite big because of repeated keys
- No comments, metadata, documentation

kafka schema registry - AVRO

AVRO

- Defined by a schema (schema is written in JSON)
- Also used in Hadoop based technologies, like Hive
- The only supported data format by the Confluent Schema Registry

Advantages

- Data is fully typed
- Data is compressed automatically
- Schema comes with the data
- Documentation is embedded in the schema
- Schema can evolve over time, in a safe manner

Disadvantages

- Not all languages support it
- Can't print the data without using avro tools (it's compressed and serialized)

```
{
  "type": "record",
  "name": "userinfo",
  "namespace": "app.user",
  "fields": [{
      "name": "username",
      "type": "string",
      "default": "None"
    },
    {
      "name": "age",
      "type": "int",
      "default": -1
    },
    {
      "name": "address",
      "type": {
        "type": "record",
        "name": "mailing_address",
        "fields": [{
            "name": "username",
            "type": "string",
            "default": "None"
          },
          {
            "name": "age",
            "type": "int",
            "default": -1
          }]
      }
    }
  ]
}
```

kafka schema registry - AVRO

schema definition

primitive types

- null
- boolean - true/false
- int - 32bit integers
- long - 64bit integers
- float - 32bit decimal numbers
- double - 64bit decimal numbers
- bytes
- string

record schema common fields

- name
- namespace - equivalent of package in java
- doc - documentation about the schema
- aliases - other optional names for your schema
- fields
  - name
  - doc - documentation about the field
  - type - can be one of the primitives
  - default - the field's default value

```
{
  "type": "record",
  "name": "userinfo",
  "namespace": "app.user",
  "fields": [{
      "name": "username",
      "type": "string",
      "default": "None"
    },
    {
      "name": "age",
      "type": "int",
      "default": -1
    },
    {
      "name": "address",
      "type": {
        "type": "record",
        "name": "mailing_address",
        "fields": [{
            "name": "username",
            "type": "string",
            "default": "None"
          },
          {
            "name": "age",
            "type": "int",
            "default": -1
          }]
        }
      }
  ]
}
```

Let's Play?

kafka schema registry - AVRO

let's play

try to write a schema definition for the following

- It's a customer with these fields:
  - First Name
  - Last Name
  - Age
  - Height (in cms)
  - Weight (in kgs)
  - Automated email turned on (Boolean, default true)
- You can use this as a starter boilerplate:

```
{
     "type": "record",
     "namespace": "com.example",
     "name": "",
     "doc": "",
     "fields": [
       { "name": "", "type": "", "doc": "" },
       { "name": "", "type": "", "doc": "" }
     ]
}
```

Solution in the *schema-definition* folder: *1-customer-solution.avsc*

kafka schema registry - AVRO

complex types

- Enums - the value of the field must belong to a list of possible values
  - Ex: customer status can only be Bronze, Silver or Gold

```
{ "type": "enum", "name": "customer_status", "symbols": ["Bronze", "Silver", "Gold"] }
```

- Arrays - a list of items with the same schema
  - Ex: multiple customer emails

```
{ "type": "array", "name": "customer_emails", "items": "string" }
```

- Maps - a list of key-value pairs, where the keys are strings
  - Ex: pasword recovery secret questions

```
{ "type": "map", "name": "recovery_question", "values": "string" }
```

- Unions - allow a field to have multiple types
  - Ex: customer middle name might be null or a string
  - If a default value is defined, it must be of the first type

```
{ "type": "union", "name": "middle_name", "type": ["null", "string"], "default": null }
```

Let's Play?

kafka schema registry - AVRO

let's play

try to write a complex schema definition for the following

- It's a customer with these fields:
  - First Name
  - Middle Name (optional)
  - Last Name
  - Age
  - Automated email (Boolean, default true)
  - Customer Emails (array)
  - CustomerAddress
    - Address
    - City
    - Postcode (number or string)
    - Type (Enum: PO BOX, RESIDENTIAL, ENTERPRISE)
- You can use this as a starter boilerplate:

Solution in the *schema-definition* folder: *2-customer-upgraded-solution.avsc*

- You can use this as a starter boilerplate:

```
[{
        "type": "record",
        "namespace": "com.example",
        "name": "CustomerAddress",
        "fields": [
          { "name": "", "type": "" }
        ]
    },
    {
        "type": "record",
        "namespace": "com.example",
        "name": "Customer",
        "fields": [
          { "name": "first_name", "type": "string" },
          { "name": "last_name", "type": "string" },
          { "name": "age", "type": "int" },
          { "name": "automated_email", "type": "boolean", "default": true }
        ]
}]
```

kafka schema registry - AVRO

logical types

- Used to give more meaning to the primitive types
- Some of them:
  - Decimals (bytes)
  - Date (int) - number of days since unix epoch (Jan 1st 1970)
  - time-millis (long) - number of milliseconds after midnight
  - timestamp-millis (long) - number of milliseconds since unix epoch
- Ex: customer signup timestamp

```
{ "type": "long", "name": "signup_ts", "logicalType": "timestamp-millis" }
```

View example in the *schema-definition* folder: *3-customer-timestamp.avsc*

kafka schema registry - AVRO

generic record

- Used to create an avro object from a schema
- The schema might be in a file or specified as a string
- It's not the recommended way of creating avro objects, but it is the most simple way.

specific record

- Used to create an avro object from a schema
- Obtained using a code generation tool like Maven

---

avro tools

- Used to quickly inspect avro objects, like the records above
- Used as a CLI

Let's Play?

kafka schema registry - AVRO

let's play

what we'll do

- Generic record:
  - Create schema from a string
  - Write the schema and data to a file
  - Read data from a file
- Specific record
  - Automatically generate a java class for our schema
  - Write the schema and data to a file
  - Read data from a file
- Avro tools
  - Use it to retrieve data and schema from the objects we created above

Project used: *avro-kafka-project*

kafka schema registry - AVRO

schema evolution

about

- Avro enables us to evolve our schema over time, to adapt with the business changes.
- Ex: today we need the first and last name of a customer, and that's our V1 schema, but tomorrow we may also need their phone number. That will be our V2 schema.
- We want to be able to make the schema evolve without breaking the programs that consume our data.
- There are four kinds of schema evolution:
  - Backward - when a new schema can also be used to read old data
  - Forward - when an old schema can be used to read new data
  - Full - is both backward and forward
  - Breaking - is none of those

backward

```
{ "name": "customer", "version": "1",
  "fields": [
    {"name": "firstName", "type": "string"},
    {"name": "lastName", "type": "string"}
  ]
}
```

```
{ "name": "customer", "version": "2",
  "fields": [
    {"name": "firstName", "type": "string"},
    {"name": "lastName", "type": "string"},
    {"name": "phoneNumber", "type": "string", "default": "000-000-000"}
  ]
}
```

- We can read both new and old data thanks to the default value in our new field.

kafka schema registry - AVRO

schema evolution

forward

```
{ “name”: “customer”, “version”: “1”,
  “fields”: [
    {“name”: “firstName”, “type”: “string”},
    {“name”: “lastName”, “type”: “string”}
  ]
}
```



```
{ “name”: “customer”, “version”: “2”,
  “fields”: [
    {“name”: “firstName”, “type”: “string”},
    {“name”: “lastName”, “type”: “string”},
    {“name”: “phoneNumber”, “type”: “string”, “default”: “000-000-000”}
  ]
}
```

- We can still read new data with the old schema, avro will just ignore the new fields, but old data will not work with the new schema.

full

```
{ “name”: “customer”, “version”: “1”,
  “fields”: [
    {“name”: “firstName”, “type”: “string”},
    {“name”: “lastName”, “type”: “string”}
  ]
}
```

```
{ “name”: “customer”, “version”: “2”,
  “fields”: [
    {“name”: “firstName”, “type”: “string”},
    {“name”: “lastName”, “type”: “string”},
    {“name”: “phoneNumber”, “type”: “string”, “default”: “000-000-000”}
  ]
}
```

- The recommended type. Has only one rule: specify a default value everytime a field is added or removed.

kafka schema registry - AVRO

breaking

- Examples of changes that break compatibility:
  - Adding/Removing elements from an Enum
  - Change a field's type (ex: string => int)
  - Rename a required field (a field without a default)

advices when writing a schema

- Make your primary key required
- Give default values to all the fields that could be removed in the future
- Be very careful when using Enums since they can't evolve
- Don't rename fields, use aliases instead
- When evolving a schema, always provide default values
- When evolving a schema, never delete a required field

Let's Play?

kafka schema registry - AVRO

let's play

what we'll do

- Backward compatible change
  - Write with old schema -> read with new schema
- Forward compatible change
  - Write with new schema -> read with old schema

Project used: *avro-kafka-project*

kafka fundamentals

Kafka Schema Registry

Hands-on

kafka schema registry

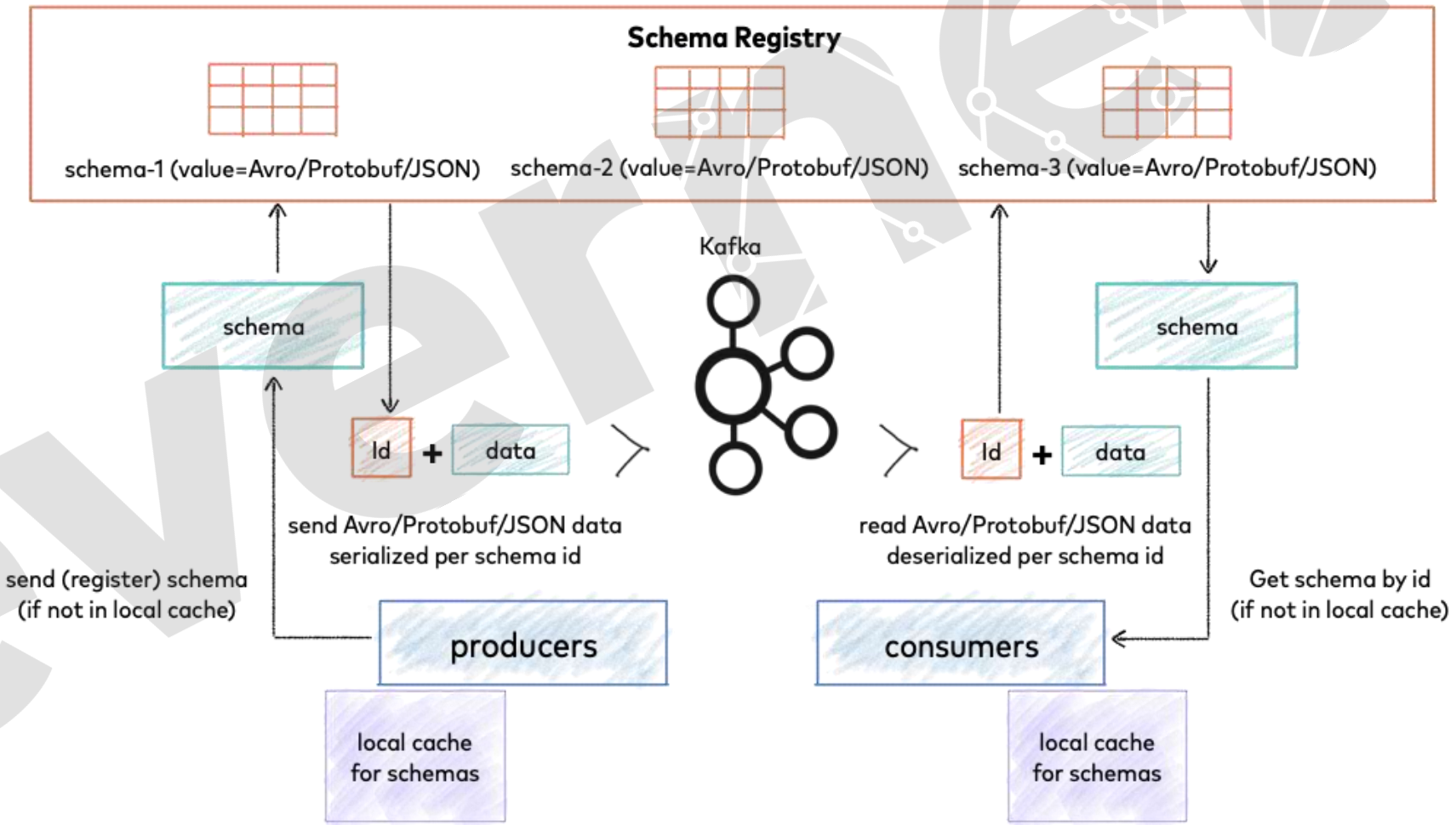
how it works

what is does

- Stores and retrieves schemas for Producers / Consumers
- Enforce backward / forward / full compatibility on topics
- Decrease the size of the payload of data sent to Kafka

operations

- Add schema
- Retrieve Schema
- Update Schema
- Delete Schema
- Operations done through REST API



Let's Play?

kafka schema registry - hands-on

let's play

what we'll do

- Write a kafka producer in Java
- Write a kafka consumer in Java
- Leverage the knowledge from before using the SpecificRecord as the way to create Avro objects

Project used: *avro-kafka-project-v1*

Let's Play?

kafka schema registry - hands-on

let's play

what we'll do

- Backward compatible change
  - Write with old schema -> read with new schema
- Forward compatible change
  - Write with new schema -> read with old schema

Project used: *avro-kafka-project-v2*

kafka fundamentals

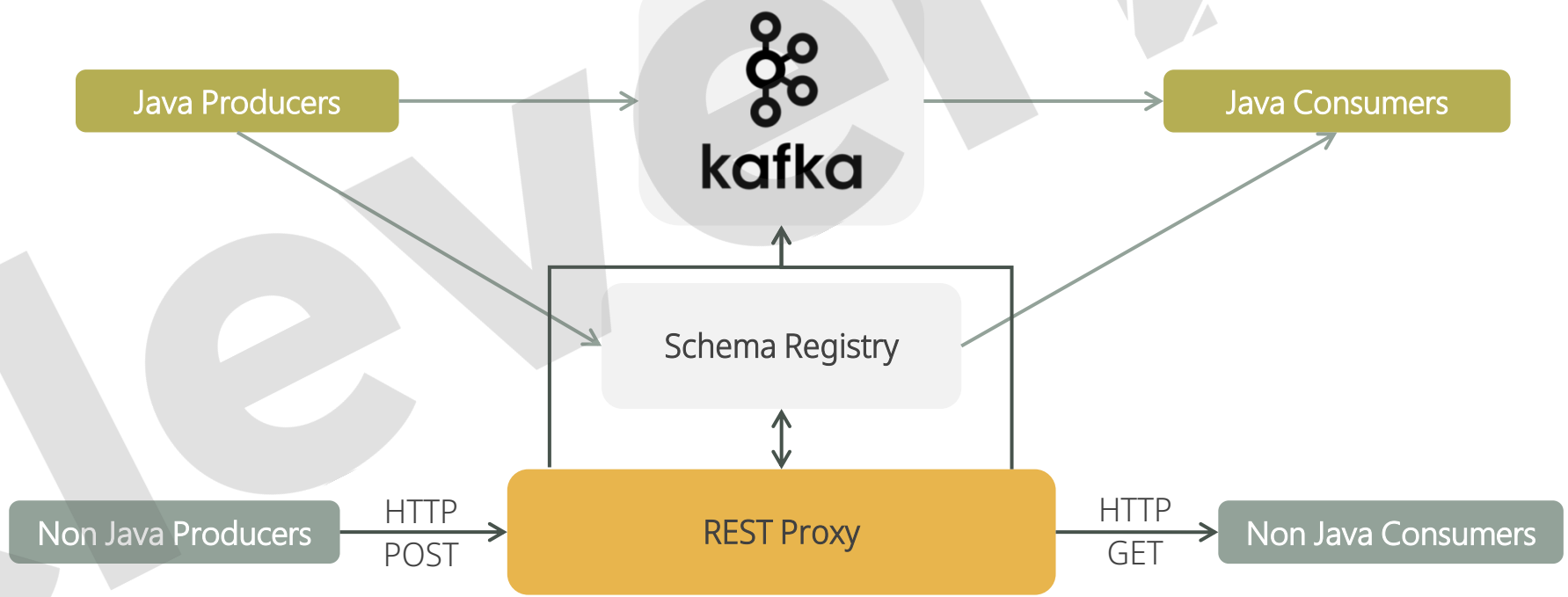
Kafka REST Proxy

kafka REST proxy

what is

- Open source project created by Confluent
- Addresses the need of programming producers and/or consumers in languages other than Java, that don't support Avro.
- Many languages don't support Avro, but all of them support JSON and HTTP requests.

There's a performance hit to using HTTP instead of the native protocol. It's estimated that the throughput is 3-4x slower when using the REST proxy.



kafka REST proxy

what is

- Any request made to the REST proxy needs a specific Content-Type header...

```
Content-Type: application/vnd.kafka.[embedded_format].[api_version]+[serialization_format]
```

[embedded_format]	[api_version]	[serialization_format]
json, binary or avro	always v2	always json

- ... and an Accept header. Example:

```
Content-Type: application/vnd.kafka.avro.v2+json
Accept: application/vnd.kafka.v2+json
```

- Topics cannot be created using the REST proxy, you can only list them or get their details

Let's Play?

kafka schema registry - AVRO

let's play

what we'll do

- Topics
  - List and get details
- Producers
  - Binary, JSON
- Consumers
  - Binary, JSON

Create the topics in the CLI using the commands in *REST-Proxy.create_topics_commands.txt* .

We'll be using Postman next. Import the *REST-Proxy.postman_collection.json* file into the program to replicate the requests.

Let's Play?

kafka schema registry - AVRO

let's play

what we'll do

- See how can we use the REST proxy with AVRO schemas and the Schema Registry
- AVRO Producer
  - To work with AVRO, you need to send the schema in JSON (stringified) and the data encoded in JSON
  - After the first produce call, you can get a schema id to re-use in the next requests to make them smaller
- AVRO Consumer

We'll continue to use Postman.

kafka fundamentals

Kafka Security

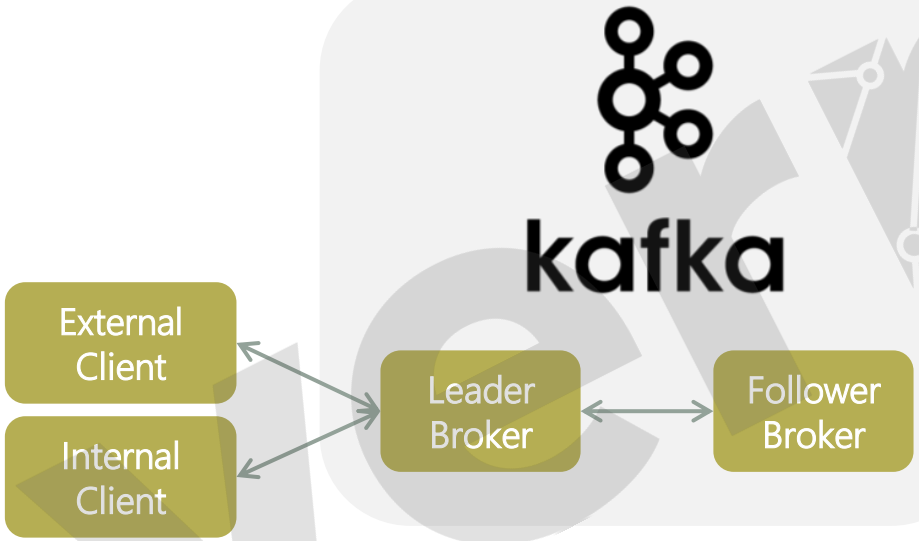
kafka security

authentication

All communications between a client (ex: a producer) and the kafka cluster need to authenticate through a KafkaPrincipal (a username).

The principal can only do what he is authorized to do and all its actions are logged.

If authentication isn't enabled, everyone's principal is *Anonymous*.



Different hosts, ports and security protocols can be used depending on the connection type: external, internal (same network) or between brokers

Available security protocols:

plaintext - unsecure
sasl_plaintext - unsecure
ssl - secure
sasl_ssl - secure

example configuration

```
listeners=EXTERNAL://:9092,INTERNAL://10.0.0.2:9093,BROKER://10.0.0.2:9094
advertised.listeners=EXTERNAL://broker1.example.com:9092,INTERNAL:// broker1.local:9093,BROKER://broker1.local:9094
listener.security.protocol.map=EXTERNAL:SASL_SSL,INTERNAL:SSL,BROKER:SSL
inter.broker.listener.name=BROKER
```

authorization

- Determines what an entity can do on a system once it has been authenticated.
- Kafka uses access control lists (ACLs) to specify which users are allowed to perform which operations on specific resources or groups of resources.

```
The user:Test has allow permission to write to topic:customer
```

principal | permission type | operation | resource

example ACL creation

- Use the **kafka-acls** command-line tool to create ACLs.

```
kafka-acls --bootstrap-server localhost:9092 \
  --command-config adminclient-configs.conf  \
  --add \
  --allow-principal User:alice \
  --allow-principal User:fred \
  --operation read
  --operation write \
  --topic finance`
```

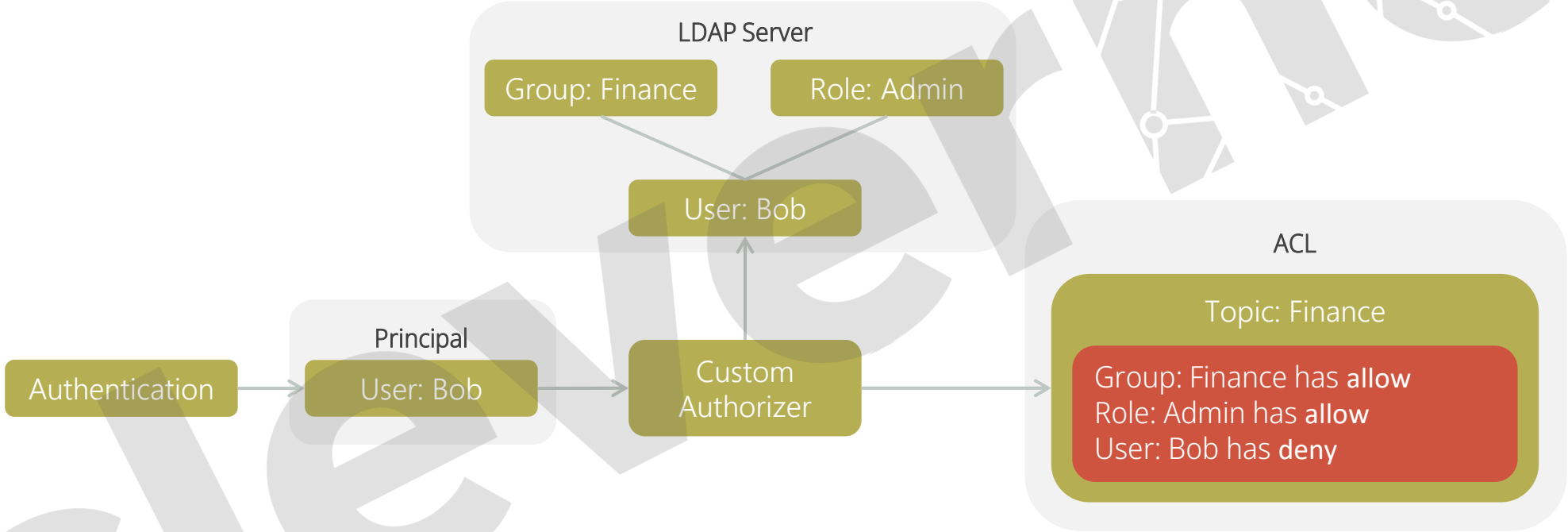
how ACL's are applied

- ACLs created with `kafka-acls` are stored in ZooKeeper, then cached in memory by every broker to enable fast lookups when authorizing requests.
- The default authorizer (for ZooKeeper Kafka) is *AclAuthorizer*, which you specify in each broker's configuration: `authorizer.class.name=kafka.security.authorizer.AclAuthorizer`.
- However, if you are using Kafka's native consensus implementation based on KRaft then you'll use a new built-in `StandardAuthorizer` that doesn't depend on ZooKeeper. *StandardAuthorizer* accomplishes all of the same things that AclAuthorizer does for ZooKeeper-dependent clusters, and it stores its ACLs in the `__cluster_metadata` metadata topic.

customize access control

- If you work in a large organization or have a large cluster topology, you might find it inefficient to specify ACLs for each individual user principal. Instead, you might wish to assign users to groups or differentiate them based on roles.
- You can accomplish this with Kafka, but you will need to do several things:
  - you'll need an external system that allows you to associate individuals with roles and/or groups, something like an LDAP store;
  - you'll need to apply ACLs to resources based not only on users but also on roles and groups;
  - finally, you'll need to implement a custom authorizer that can call your external system to find the roles and groups for a given principal.

custom implementation example



considerations

- No matter which type of authorizer you use, whether default or custom, you should **clearly distinguish between user and service accounts**. You might be inclined to reuse a person's user details to authenticate a service or application to Kafka but you shouldn't do this as people often change teams or roles (or even leave companies altogether).

- Keep in mind that ACLs require careful management. If you are working in a development or test environment, it may be tempting to use anonymous principles, or Kafka's notion of the superuser, or its `allow.everyone.if.no.acl.found` setting. But it's easy to accidentally promote these settings to a production environment so it's far better to develop good habits; from the outset, you should automatically assign proper credentials and set ACLs strictly according to need.

- If a user is compromised, you will need to remove that user from the system as soon as possible, but you will also need to check for any existing connections associated with that user since a principal is assigned to a resource upon connection. (You can set how often users need to reconnect with `connections.max.reauth.ms` but you should be careful not to frustrate your users with too frequent reconnections.) Note that if a connection persists for a long time, **removing a user will only take effect when the connection is closed** and reconnection is attempted. Thus the **best solution** for blocking access to a compromised user **is to implement a "Deny ACL"** to prevent actions on any existing connections since ACLs get propagated quickly and are checked with every request.

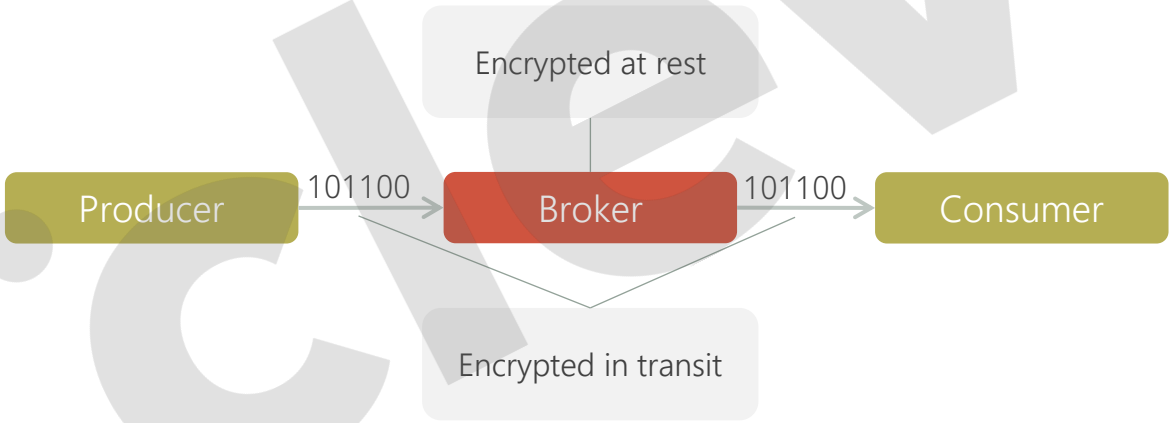
kafka security

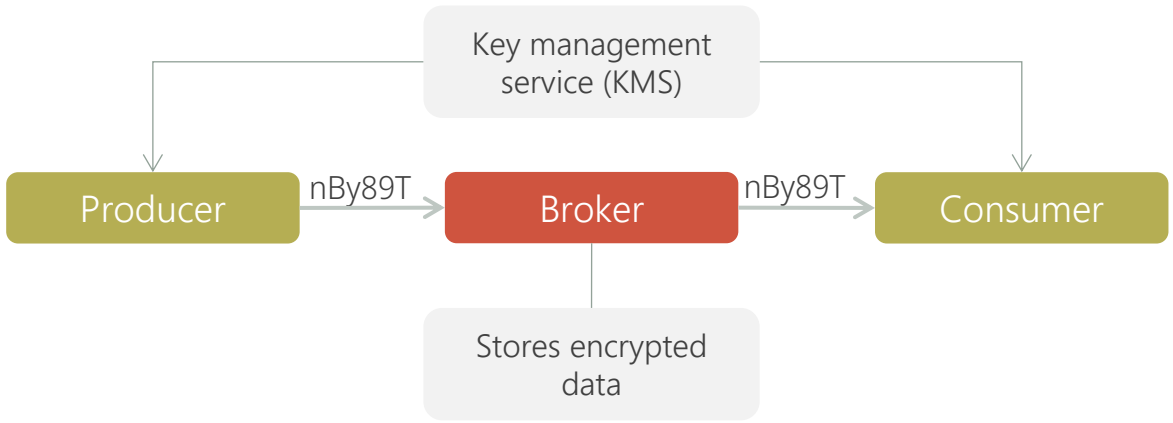
in transit and at rest (volume encryption)

- Not used by default (all data flows across the network in plain-text).
- To encrypt the communication between a client and a broker, one must configure each broker with **ssl.client.auth=required** and use an SSL certificate.
- To encrypt inter-broker communication, one must configure each broker with **security.inter.broker.protocol=SSL**.
- Encryption can cause a overhead in the system due to the cpu overload in encrypting and decrypting the data.
- Kafka doesn't have any capability to encrypt data at rest, but all the cloud providers have solutions for volume encryption.

end-to-end encryption

- Instead of the stategies above, you can implement an end-to-end encryption. See the comparison:





what is provided

- Insight – They provide insight into situations such as trying to determine if a particular group of users was successful in authenticating and getting access to the correct broker resources after a new ACL was added.
- Security – They enhance security by letting you identify anomalies and unauthorized operations in the historical record so that you can take action as quickly as possible.
- Impact – They let you see who, as well as which services, have been impacted by unusual activities, so that you can communicate with stakeholders as the situation progresses.
- Compliance – They enable you to generate audit reports according to internal policies and external regulations, and also provide an official record in the event of a security breach.

Apache Kafka doesn't provide audit logging out of the box. It does, however, support comprehensive audit logging with Log4j, which can be configured to deliver events from separate logger instances to separate destinations (typically files). By configuring separate appenders for the **log4j.properties** file on each broker, you can capture detailed information for auditing, monitoring, and debugging purposes.

kafka authorizer logger

- `kafka.authorizer.logger` is used for authorization logging and generates INFO entries for events where access was denied and DEBUG entries for those where access was allowed. Each entry includes the principal, client host, the attempted operation, and the accessed resource (e.g., a topic):
  - `log4j.properties`:

```
log4j.logger.kafka.authorizer.logger=DEBUG
```

  - sample output:

```
DEBUG Principal = User:Alice is Allowed Operation = Write from host = 127.0.0.1 on resource =
Topic:LITERAL:customerOrders for request = Produce with resourceRefCount = 1
(kafka.authorizer.logger)


INFO Principal = User:Mallory is Denied Operation = Describe from host = 10.0.0.13 on resource =
Topic:LITERAL:customerOrders for request = Metadata with resourceRefCount = 1
(kafka.authorizer.logger)
```

kafka request logger

- `kafka.request.logger` logs the principal and client host at the DEBUG level as well as full details of the request when logging at the TRACE level:
  - `log4j.properties`:

```
log4j.logger.kafka.request.logger=DEBUG
```

  - sample output:

```
DEBUG Completed request:RequestHeader(apiKey=PRODUCE, apiVersion=8, clientId=producer-1,
correlationId=6) -- {acks=-1,timeout=30000,partitionSizes=[customerOrders-0=15514]},response:
{responses=[{topic=customerOrders,partition_responses=[{partition=0,error_code=0
,base_offset=13,log_append_time=-1,log_start_offset=0,record_errors=[],error_mes
sage=null}]}],throttle_time_ms=0} from connection 127.0.0.1:9094-127.0.0.1:61040-
0;totalTime:2.42,requestQueueTime:0.112,local-
Time:2.15,remoteTime:0.0,throttleTime:0,responseQueueTime:0.04,sendTime:
0.118,securityProtocol:SASL_SSL,principal:User:Alice,listener:SASL_SSL,clientInf
ormation:ClientInformation(softwareName=apache-kafka-java, softwareVersion=2.7.0-SNAPSHOT)
(kafka.request.logger)
```

considerations

- Kafka's logs are bound to be useful to you, whether you use them for compliance, debugging, or anomaly detection. However, there are a few things to keep in mind:
  - Make sure there is enough disk space if you are capturing logs for each broker and set an **appropriate Log4j retention policy so as not to fill the disks**. The request logger is particularly volumous so you may only want to use it for debugging or within small retention windows.
  - Since logs are per broker, you will **need to consolidate all of the broker logs from the cluster** to get a complete picture of your system.
  - Kafka doesn't provide log analysis or visualization so we recommend that you **use something like the "ELK" stack** (Elasticsearch, Logstash, and Kibana) for these purposes. You might also **direct your log output to a Kafka topic** on another cluster and then perform a streaming analysis of your audit events.

kafka fundamentals

Data Pipeline Example

data pipeline

what we'll build

- An end-to-end example of a data pipeline.
- Using kafka Connect:
  - Two datasources: mock data from the datagen connector (ratings) and a MySQL table (customers).
- Using KsqlDB:
  - Enrich the ratings events with information from the customers table
- Using kafka Connect:
  - Stream data to a data sink (Elasticsearch)
- Build a simple dashboard in Elasticsearch to inspect the data

Let's Play?

data pipeline

first datasource

- Create a new topic called *ratings*
- Use the *RatingsSourceConnector_config.properties* file to add a new connector in kafka Connect.
- Check if the ratings topic gets populated with data

second datasource

- Import the *customers.sql* into Mysql:
  - Login to mysql with *mysql -uroot -p* (type *kafkaroot as password)*
  - Type *source <path-to-file>/customers.sql*
  - Inspect the data with *select * from CUSTOMERS;*
- Use the *CustomersSourceConnector_config.properties* file to add a new connector in kafka Connect.
- Check if the *db.demo.CUSTOMERS* topic exists and has data

data pipeline

filtering unwanted data

- Go to ksqlDB in Control Center
- Create stream :

```
CREATE STREAM RATINGS
  WITH (KAFKA_TOPIC='ratings',VALUE_FORMAT='AVRO');
```

- Inspect data in streams (don't forget to set `auto.offset.reset` to *Earliest*):

```
SELECT USER_ID, STARS, CHANNEL, MESSAGE
  FROM RATINGS EMIT CHANGES;
```

- We need to filter out all the events that have 'test' in the *channel* column. Let's create a new stream with only these values:

```
CREATE STREAM RATINGS_LIVE
  AS SELECT * FROM RATINGS
     WHERE LCASE(CHANNEL) NOT LIKE '%test%'
EMIT CHANGES;
```

data pipeline

enriching data

- We're going to use the customer information that we are pulling in from an external MySQL database to enrich each rating as it arrives (as well as all the existing ratings that we have already received and are stored in the Kafka topic).
- To do this, we need to first model the customer data held in the Kafka topic in such a way that ksqlDB can use it to join to the ratings events. We'll store the model in a ksqlDB table, but first we need to import the data to a stream:

```
CREATE STREAM CUSTOMERS_S
    WITH (KAFKA_TOPIC ='db.demo.CUSTOMERS'
    VALUE_FORMAT='AVRO');


SELECT * FROM CUSTOMERS_S;
```

- Now we'll use the stream to build the model:

```
CREATE TABLE CUSTOMERS WITH (FORMAT='AVRO') AS
    SELECT after->id                           AS customer_id,
           LATEST_BY_OFFSET(after->first_name)  AS first_name,
           LATEST_BY_OFFSET(after->last_name)   AS last_name,
           LATEST_BY_OFFSET(after->email)       AS email,
           LATEST_BY_OFFSET(after->club_status) AS club_status
    FROM   CUSTOMERS_S
    GROUP BY after->id;

SELECT * FROM CUSTOMERS;
```

data pipeline

enriching data

- Perform a join between the stream of ratings and the table of customer details.

```
CREATE STREAM RATINGS_WITH_CUSTOMER_DATA
        WITH (KAFKA_TOPIC='ratings-enriched') AS
  SELECT C.CUSTOMER_ID,
         C.FIRST_NAME + ' ' + C.LAST_NAME AS FULL_NAME,
         C.CLUB_STATUS,
         C.EMAIL,
         R.RATING_ID,
         R.MESSAGE,
         R.STARS,
         R.CHANNEL,
         TIMESTAMPTOSTRING(R.ROWTIME,'yyyy-MM-dd''T''HH:mm:ss.SSSZ') AS RATING_TS
FROM   RATINGS_LIVE R
        INNER JOIN CUSTOMERS C
          ON R.USER_ID = C.CUSTOMER_ID
EMIT CHANGES;

select * from RATINGS_WITH_CUSTOMER_DATA;
```

data pipeline

quick test

- To show the power of streaming changes directly from the database, we'll make a change to the customer data and observe how it is reflected in the enriched ratings data.
- Run the query below to show current ratings from customer ID 1. Because we only want current ratings, set the `auto.offset.reset` to *latest*. Note the value of CLUB_STATUS shown for each rating.

```
SELECT CUSTOMER_ID, CLUB_STATUS
  FROM  RATINGS_WITH_CUSTOMER_DATA
  WHERE CUSTOMER_ID = 1
  EMIT CHANGES;
```

- In MySQL, update the club_status of the customer:

```
UPDATE demo.CUSTOMERS
   SET CLUB_STATUS='platinum'
 WHERE ID=1;
```

- Notice that the *db.demo.CUSTOMERS* topic will pick up the change as well as the CUSTOMERS topic (it's our CUSTOMERS table in Ksql)

data pipeline

setup

- For this phase you'll need a working elasticsearch instance.
- To install the elasticsearch sink connector using confluent hub type: *confluent-hub install confluentinc/kafka-connect-elasticsearch:13.1.2*
- Use the *connector_elasticsearch-connector_config.properties* file to add a new connector in kafka Connect. The following options might need some tweaking, depending on the instance settings:

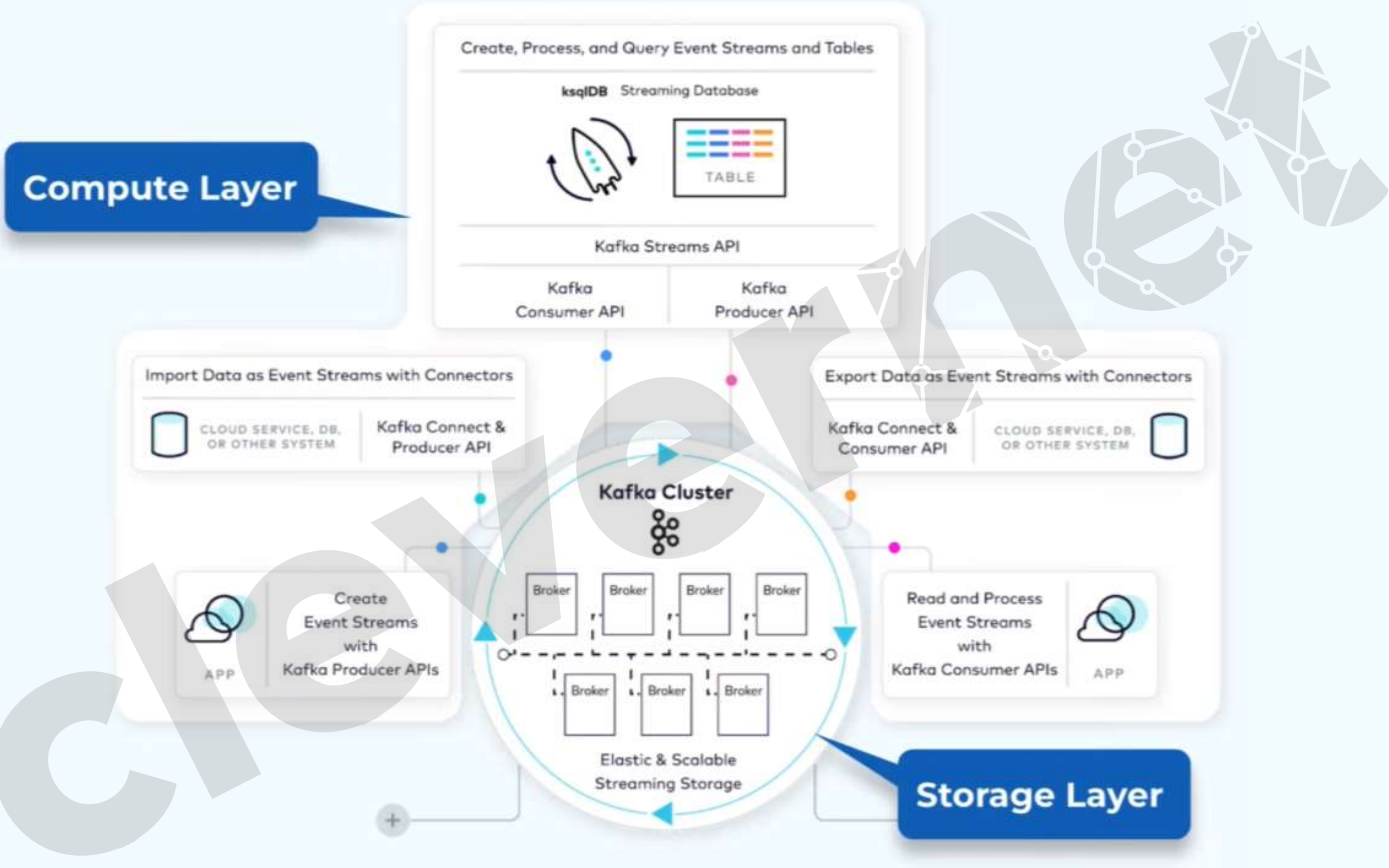
```
connection.url=http://localhost:9200
connection.username=elastic
connection.password=elastic
```

- In Elasticsearch, check that data has been received in the index. You can do this using the REST API, Kibana or using curl:

```
curl -u $ES_USER:$ES_PASSWORD $ES_URL/_cat/indices/ratings\*\?v=true
```

kafka fundamentals

That's a wrap!


Compute Layer
Create, Process, and Query Event Streams and Tables
ksqlDB Streaming Database
TABLE
Kafka Streams API
Kafka Consumer API
Kafka Producer API
Import Data as Event Streams with Connectors
CLOUD SERVICE, DB, OR OTHER SYSTEM
Kafka Connect & Producer API
Export Data as Event Streams with Connectors
Kafka Connect & Consumer API
CLOUD SERVICE, DB, OR OTHER SYSTEM
Kafka Cluster
Broker
Broker
Broker
Broker
Broker
Broker
Broker
APP
Create Event Streams with Kafka Producer APIs
Read and Process Event Streams with Kafka Consumer APIs
APP
Elastic & Scalable Streaming Storage
Storage Layer

Obrigado :)

rferreira@clevernet.pt
967 270 033
linkedin.com/in/rferreirapt