

Technical Project Documentation

Unstructured and Streaming Data Engineering

Scuola di Ingegneria Industriale e dell'Informazione,
Computer Science Engineering

&

EIT Digital Master School, Fintech Double Degree
Politecnico di Milano – Rennes1



POLITECNICO
MILANO 1863

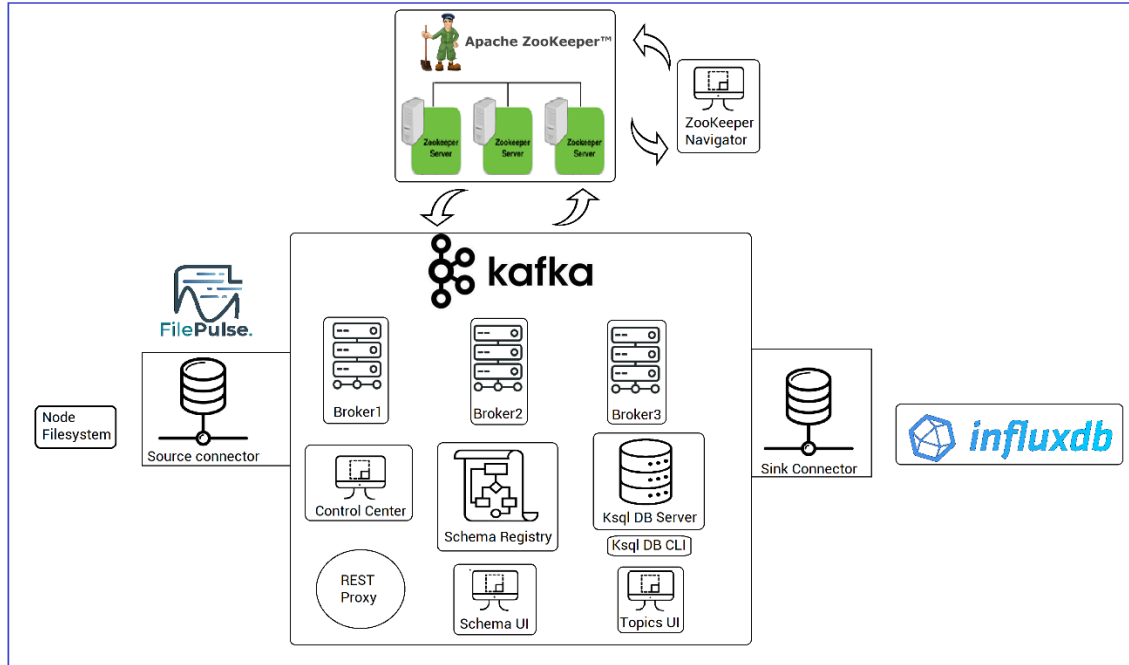


Digital
MASTER SCHOOL

*Creation of a data pipeline for the ingestion, elaboration, and
presentation of streaming data with Apache Kafka and InfluxDB*

*Omar Abdrabou,
Codice Persona: 10573522,
Matricola: 963500*

1. System Architecture



The system is composed of four major blocks:

1. The files are loaded into the kafka-connect filesystem, each in its own directory.
2. A Kafka ecosystem as the main component of this architecture. It will ingest the data coming from the source, elaborate it with queries and deliver it to the target database.
3. Three instances of ZooKeeper to help manage the Kafka ecosystem.
As the Hadoop instance is single node and does not implement HA, they do not communicate in any way.
4. InfluxDB that acts as the target system for the aggregated data. It will store the results in buckets and visualize them by making use of dashboards.

The hypothesis is that an external system will collect the data from the sensors, insert it into .csv files and send it into a predetermined directory.

The raw data will be then stored for a set amount of time, specified by the business rules of the case.

After the files have arrived on the destination directory, the ingestion process will start by using the Filepulse Source Connector, the data inside each of the .csv files will be sent to their respective topic inside Kafka.

Hypothetically, this process could be automatized by implementing a job that checks for the files' presence: if all the raw data has arrived, then the ingestion process can start.

After the data is ingested into the Kafka topics, a set of predetermined KSQL queries will be run on these topics to elaborate the data. The results will be then sent to new topics.

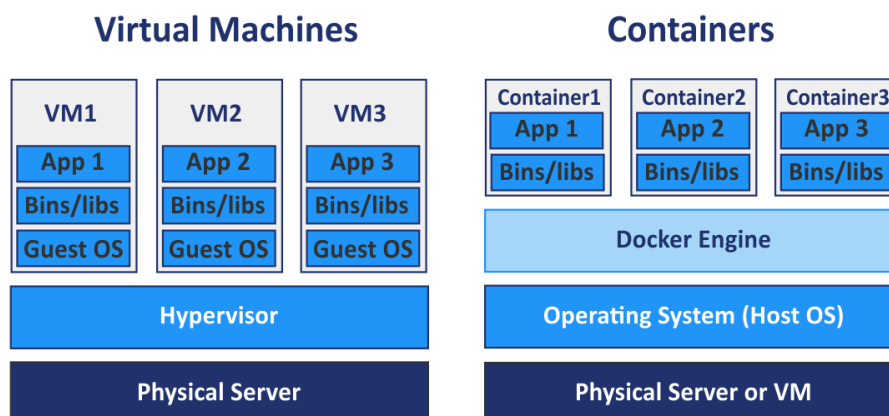
Once the data processing step is complete, the aggregated data will be sent to the target system, in this case InfluxDB, by means of the Sink Connector, to be then organized and visualized in dashboards.

- This process can be managed by means of a scheduler, like Tivoli, to precisely control when and how the pipeline will operate to generate the results that might be used for a multitude of purposes.
- Also, in real case, a better source store could be implemented like **HDFS**.

1.1. Docker



Docker is an open-source platform for building, deploying, and managing containerized applications. A Docker container is a lightweight, standalone, executable unit of software that includes everything needed to run an application, including libraries, system tools, code, and runtime.

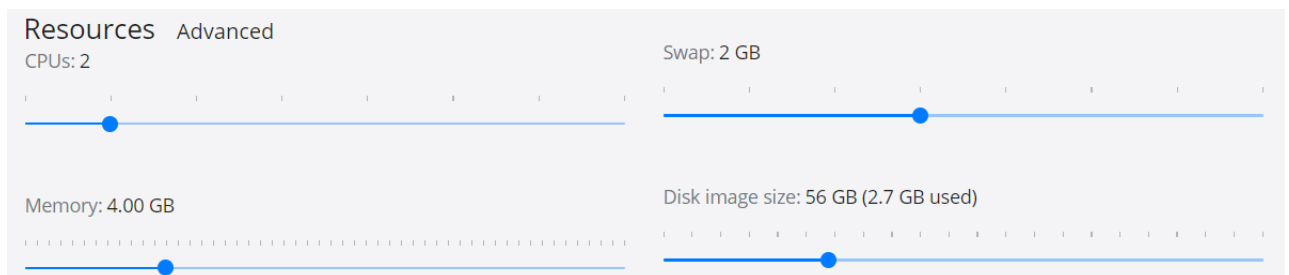


Containers are made possible by operating system (OS) process isolation and virtualization, which enable multiple application components to share the resources of a single instance of an OS kernel the same way that machine virtualization enables multiple virtual machines (VMs) to share the resources of a single hardware server.

1.1.1. Docker Configuration

All of the applications involved in this project are deployed inside Docker Desktop containers and implements Docker engine version 20.10.2.

The used version of Docker does not use on WSL2 based engine, but it still relies on the Hyper-V backend to emulate Linux native containers.



There are three .yml files stored locally that compose the system, each of them representing an important piece of the architecture and carrying out different tasks.

1. *Kafka_cluster.yml*
 - This file contains the instances of the ZooKeeper service, Kafka brokers and the other components that make up the Kafka ecosystem.
2. *Hadoop_cluster.yml*
 - This file contains a simple instance of a Hadoop cluster, including an History server and the Resource Manager.
 - The environment variables for the Hadoop cluster are not written directly in the yaml file but loaded from an external file located in the same directory, called *hadoop.env*.
3. *Influxdb.yml*
 - This file is related to the instance of Influx DB that is being used to visualize the aggregated data.

1.1.2. Docker Containers

Container Image Name	Version
----------------------	---------

zookeeper	3.4.9
zoonavigator	1.1.0
confluentinc/cp-server (Kafka Broker)	6.0.1
confluentinc /cp-schema-registry (Kafka Schema Registry)	6.0.1
landoop/schema-registry-ui	0.9.5
confluentinc/cp-kafka-rest	6.0.1
landoop/kafka-topics-ui	0.9.4
confluentinc/ksqldb-server	0.15.0
confluentinc/ksqldb-cli	6.0.1
confluentinc/cp-kafka-connect	6.0.1
confluentinc/cp-enterprise-control-center	6.0.1
quay.io/influxdb	2.0.2

1.1.3. Docker Commands

The system is started and stopped by running the following commands:

1. *docker-compose -f file_name.yml up*
 - To start all the services contained in that yaml file.
2. *docker-compose -f file_name.yml down*
 - To stop all the services contained in that yaml file.

To access the filesystem of a container, you can go from the Docker Desktop interface or execute from the command prompt the command:

- *docker-compose -f file_name.yml exec container_name bash*

To copy a file from the local filesystem to the container's filesystem, execute this command from the prompt:

- *docker cp path_to_file/file_name container_name:container_absolute_path*

To start the ksqldb CLI, execute this command when the ksqldb Server is up and running:

- *docker-compose -f file_name.yml exec ksqldb-cli ksql http://ksqldb-server:8088*

1.2 ZooKeeper



Apache ZooKeeper™

ZooKeeper is a distributed system configuration management service, that offers core functionalities to handle the coordination of distributed processes and shared resources, through a shared hierarchical name space of data registers called zNodes, similar to a file system.

Regarding the Apache Kafka ecosystem, ZooKeeper implements the following features:

- Brokers registration
 - Implemented with the heartbeat mechanism to keep an updated list of active brokers
- List of created topics
 - With their configuration (partitions, replication factors, additional configuration, ecc...)
- The list of ISRs (in-synch replicas) for each partition
- Leader elections, in case some brokers go down
 - Implemented with a voting mechanism, based on strict majority vote
- Storing the cluster ID
 - Randomly generated at the first startup of the cluster
- ACLs (access control lists), if the security is enabled
 - Related to topics, consumer groups or users
- Quotas configuration, if enabled
 - Related to limits about the consumption/production of data of nodes, to protect against the monopolization of the cluster's resources

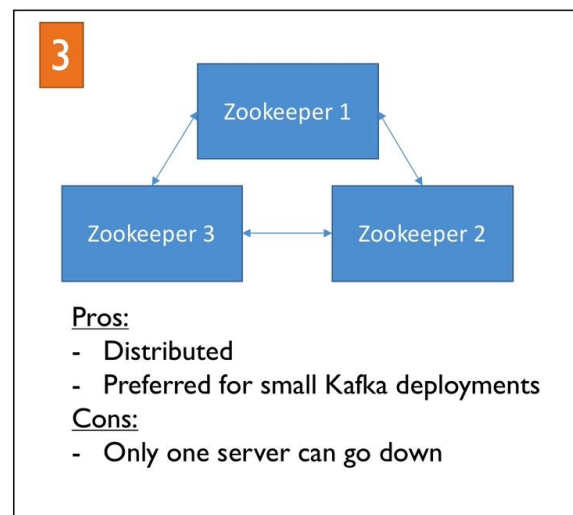
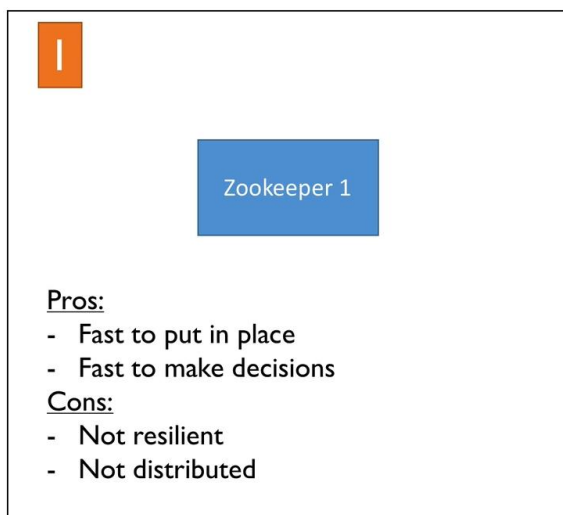
1.2.1 ZooKeeper Servers

The chosen ZooKeeper image is Apache's official image, version 3.4.9, due to its well-proven stability.

The presented solution is composed of three ZooKeeper servers, with the objective of implementing an actual distributed system 'locally'.

All servers participate in the quorum: no *Observer* has been implemented.

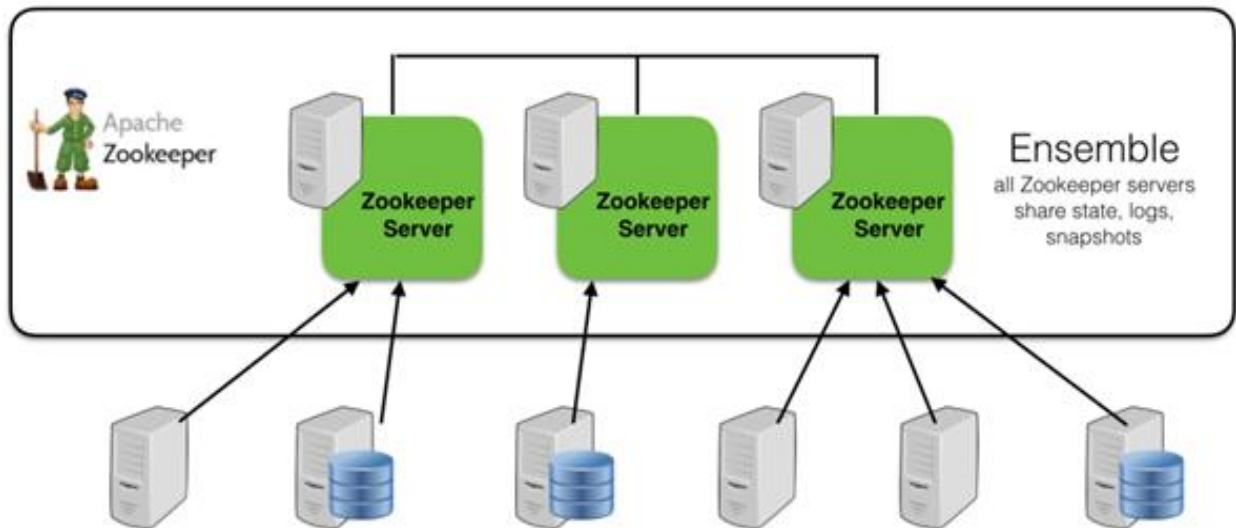
- *Considering the number of resources available and the eventual network overhead generated from having three servers, only one server would have been a more optimal solution.*
By implementing three servers, we can easily see how the leader election happens and how the servers react when one becomes unavailable: for these reasons, this was the chosen approach.



1.2.2 ZooKeeper Configuration

All four letters command words have been whitelisted with the parameter `ZOO_4LW_COMMANDS_WHITELIST`, to enable checking.

The parameter `ZOO_STANDALONE_ENABLED` explicitly to false because it allows to dynamically add or remove other ZooKeeper servers, giving more freedom in the management of the ensemble.



To persist the data and configuration of the ZooKeeper system, two directories have been set:

- `./USDE-project/zoox/data: /data`
 - To store the in-memory database snapshots
- `./USDE-project/zoox/datalog: datalog`
 - To store the transaction log of the updates of the database

1. The internal port **2181** was mapped to the port **2181**, for other services to connect to the first ZooKeeper server.
2. The internal port **2182** was mapped to the port **2182**, for other services to connect to the second ZooKeeper server.
3. The internal port **2183** was mapped to the port **2183**, for other services to connect to the third ZooKeeper server.

On each server:

- The port **2888** was designated as the port for Followers to connect to the Leader.
- The port **3888** was designated as the port for the Leader election.

The *tickTime* property (ZOO_TICK_TIME) was set to 3000 ms, as a standard unit for actions related to the ZooKeeper servers.

The *initLimit* property (ZOO_INIT_LIMIT) was set to 5, corresponding to the amount of ticks the followers have to connect and synch to the current elected leader.

The *synchLimit* property (ZOO_SYNC_LIMIT) was set to 3, corresponding to the amount of ticks the followers have to just synch with the current elected leader.

ZooKeeper Configuration	
ZooKeeper Version	3.4.9
Restart Option	on-failure
Hostname	zoo1 zoo2 zoo3
Port Mapping	2181:2181 2182:2182 2183:2183
Follower Connection Port	2888
Leader Election Port	3888
Tick Time	3000
Init Limit	5
Synch Limit	3
Data Directory	./USDE-project/zoo1/data:/data ./USDE-project/zoo2/data:/data ./USDE-project/zoo3/data:/data
Datalog Directory	./USDE-project/zoo1/datalog:/datalog ./USDE-project/zoo2/datalog:/datalog ./USDE-project/zoo3/datalog:/datalog

Service Dependencies
//

To avoid issues and to assure that the cluster's components can connect to the servers only when it is ready to accept requests, *an healthcheck* option was added to the ZooKeeper containers:

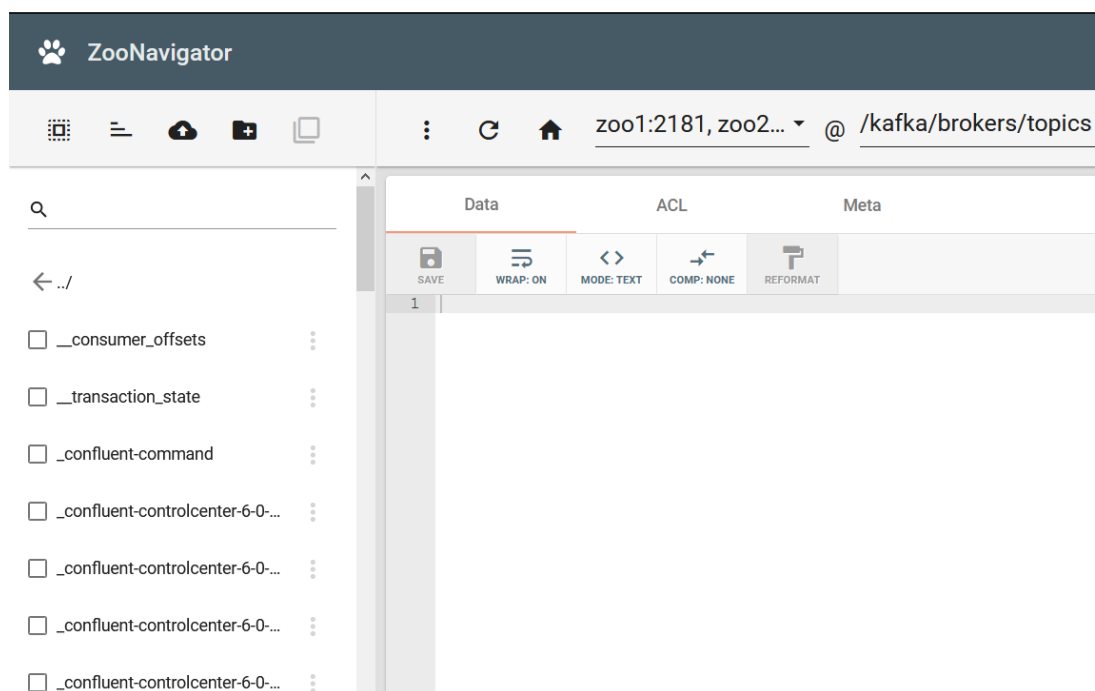
- The condition for the service to be up and ready to accept requests is 'exit 0'.
- Tests start running after 10 seconds from the container startup.
- If the result of the test is negative, Docker will wait 15 seconds to run another one, until the servers are ready.

Healthcheck Parameters	
Test Condition	Exit 0
Start Period Value	10s
Interval Value	15s

1.3 ZooNavigator

As a support for ZooKeeper, the Zoonavigator image from elkozmon was also implemented into the system architecture.

- *ZooNavigator is a web-based, open-source ZooKeeper UI and editor/browser: it enables quick and easy access of the zNodes and their content, offering useful troubleshooting and managing capabilities.*



ZooNavigator Configuration	
Hostname	zoonavigator
Ports Mapping	8004:8000
HTTP Port	8000

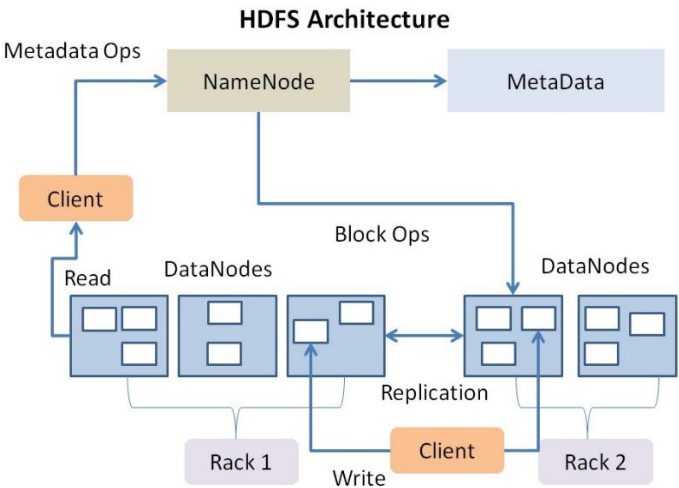
Service Dependencies
zoo1 – Service Healthy

zoo2 – Service Healthy
zoo3 – Service Healthy

1.4 (Optional) Hadoop

A Hadoop Distributed Filesystem (HDFS) is cluster that consists of a single active NameNode, a master server that manages the file system namespace and regulates access to files by clients, and a number of DataNodes, which manage storage attached to the nodes that they run on.

HDFS exposes a file system namespace and allows user data to be stored in files, which are internally split into one or more blocks that are stored in a set of DataNodes. The NameNode executes file system namespace operations like opening, closing, and renaming files and directories. It also determines the mapping of blocks to DataNodes. The DataNodes are responsible for serving read and write requests from the file system’s clients.



1.4.1 Hadoop Configuration

All the environment variable used in the setup of this section of the system are stored in an external file *hadoop.env*, stored in same position of the related yaml file.

- As the first implementation of the system included HDFS as a raw data landing zone, but was substituted with later iterations on the architecture, I decided to quote it into this document.

CORE_CONF_fs_defaultFS	hdfs://namenode:9000
CORE_CONF_hadoop_http_staticuser_user	root
CORE_CONF_hadoop_proxyuser_hue_hosts	*

CORE_CONF_hadoop_proxyuser_hue_groups	*
CORE_CONF_io_compression_codecs	org.apache.hadoop.io.compress.SnappyCodec
HDFS_CONF_dfs_webhdfs_enabled	TRUE
HDFS_CONF_dfs_permissions_enabled	FALSE
HDFS_CONF_dfs_namenode_datanode_registration_ip__hostname__check	FALSE
YARN_CONF_yarn_log__aggregation__enable	TRUE
YARN_CONF_yarn_log_server_url	http://historyserver:8188/applicationhistory/logs/
YARN_CONF_yarn_resourcemanager_recovery_enabled	TRUE
YARN_CONF_yarn_resourcemanager_store_class	org.apache.hadoop.yarn.server.resourcemanager.recovery.FileSystemRMStateStore
YARN_CONF_yarn_resourcemanager_scheduler_class	org.apache.hadoop.yarn.server.resourcemanager.scheduler.capacity.CapacityScheduler
YARN_CONF_yarn_scheduler_capacity_root_default_maximum__allocation__mb	8192
YARN_CONF_yarn_scheduler_capacity_root_default_maximum__allocation__vcores	4
YARN_CONF_yarn_resourcemanager_fs_state__store_uri	/rmstate
YARN_CONF_yarn_resourcemanager_system__metrics__publisher_enabled	TRUE
YARN_CONF_yarn_resourcemanager_hostname	resourcemanager
YARN_CONF_yarn_resourcemanager_address	resourcemanager:8032
YARN_CONF_yarn_resourcemanager_scheduler_address	resourcemanager:8030
YARN_CONF_yarn_resourcemanager_resource__tracker_addresses	resourcemanager:8031
YARN_CONF_yarn_timeline__service_enabled	TRUE
YARN_CONF_yarn_timeline__service_generic__application__history_enabled	TRUE
YARN_CONF_yarn_timeline__service_hostname	historyserver
YARN_CONF_mapreduce_map_output_compress	TRUE
YARN_CONF_mapred_map_output_compress_codec	org.apache.hadoop.io.compress.SnappyCodec
YARN_CONF_yarn_nodemanager_resource_memory__mb	16384
YARN_CONF_yarn_nodemanager_resource_cpu__vcores	8
YARN_CONF_yarn_nodemanager_disk__health__checker_max__disk__utilization__per__disk__percentage	98.5
YARN_CONF_yarn_nodemanager_remote__app__log__dir	/app-logs
YARN_CONF_yarn_nodemanager_aux__services	mapreduce_shuffle
MAPRED_CONF_mapreduce_framework_name	yarn
MAPRED_CONF_mapred_child_java_opts	-Xmx4096m
MAPRED_CONF_mapreduce_map_memory_mb	4096
MAPRED_CONF_mapreduce_reduce_memory_mb	8192
MAPRED_CONF_mapreduce_map_java_opts	-Xmx3072m
MAPRED_CONF_mapreduce_reduce_java_opts	-Xmx6144m
MAPRED_CONF_yarn_app_mapreduce_am_env	HADOOP_MAPRED_HOME = /opt/hadoop-3.2.1/
MAPRED_CONF_mapreduce_map_env	HADOOP_MAPRED_HOME = /opt/hadoop-3.2.1/
MAPRED_CONF_mapreduce_reduce_env	HADOOP_MAPRED_HOME = /opt/hadoop-3.2.1/

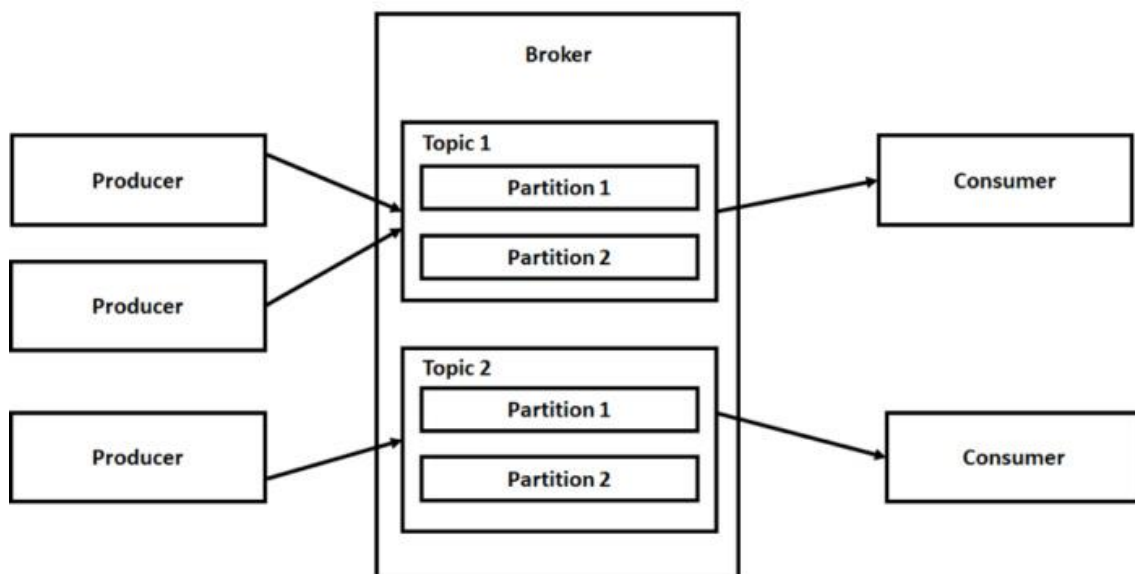
1.5 Apache Kafka



Apache Kafka is a distributed data store optimized for ingesting and processing streaming data in real-time, which is continuously generated by thousands of different data sources, which typically send the records in simultaneously.

Kafka provides three main functions to its users:

- Publish and subscribe to streams of records
- Effectively store streams of records in the order in which records were generated
- Process streams of records in real time

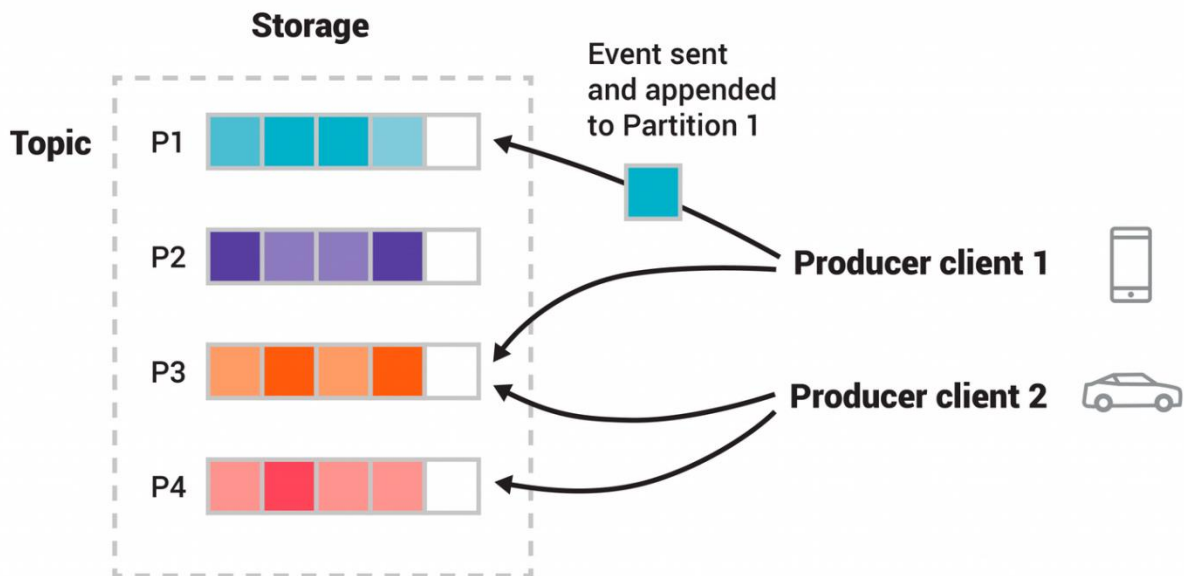


Producers are those client applications that publish (write) events to Kafka, and **consumers** are those that subscribe to (read and process) these events and are fully decoupled with each other, while brokers are the actual node of the cluster, that store and manage these events.

Events are organized and stored in **topics that are stored inside brokers as logs in a key-value format**. Topics in Kafka are always multi-producer and multi-subscriber, as they can have zero or many producers that write events to it, as well as zero or many consumers that subscribe to these events.

Events in a topic can be **read as often as needed** and are not deleted on consumption: their handling policy can be specified on the topic creation and can be based on time passed or total size reached.

It is also possible to set topics **replication factor**, specifying how many copies each event message of that topic will have, and how many **partitions** it should have, specifying the number of storages on different brokers.



1.5.1 Kafka Brokers

A Kafka cluster is composed of multiple 'server-like' nodes, called Brokers, that storage the messages that are sent by Producers and manage them.

In this system architecture, three brokers we instantiated: this is considered the bare minimum to have a decent fault-tolerance to nodes failures and observe how the cluster balances itself in case the controller becomes unavailable and how the nodes recover the lost time and messages after coming back up.

1.5.1.2 Kafka Brokers Configuration

To persists the data and configuration of the Kafka brokers, the following directory has been set:

- `./USDE-project/kafkax/data: /var/lib/kafka/data`
 - To store the logs and the topics' messages
- The internal port **9092** was mapped to the port **9092**, for services external to Docker and **19092** for internal services to connect to the first Kafka Broker.

- The internal port **9093** was mapped to the port **9094**, for services external to Docker and **19093** for internal services to connect to the second Kafka Broker.
- The internal port **9094** was mapped to the port **9094** for services external to Docker and **19094** for internal services to connect to the third Kafka Broker.

On each server:

- The minimum number of replicas that need to send confirmation of a Produced write, KAFKA_MIN_INSYNC_REPLICAS, is set to 2.
- The log retention time, KAFKA_LOG_RETENTION_HOURS, is set to 700.
- The maximum size of a single log file, KAFKA_LOG_SEGMENT_BYTES, is set to 1073741824, equaling 1 GB.
- The log policy is set to delete.
- To enable the sending of brokers' metrics to the Control Center for monitoring, KAFKA_METRIC_REPORTERS was set.

Also, a **chroot /kafka** was set to keep tidy the information related to Kafka on the zNodes, allowing an easy retrieval of the generated data.

Kafka Brokers Configuration	
Brokers Version	6.0.1
Restart Option	on-failure
Hostname	kafka1 kafka2 kafka3
Data Directory	./USDE-project/kafka1/data: /var/lib/kafka/data ./USDE-project/kafka2/data: /var/lib/kafka/data ./USDE-project/kafka3/data: /var/lib/kafka/data
Port Mapping	Internal1: 19092 – External1: 9092:9092 Internal2: 19093 – External2: 9093:9093 Internal3: 19094 – External3: 9094:9094
Advertised Listeners	LISTENER_DOCKER_INTERNAL://kafkax:1909x, LISTENER_DOCKER_EXTERNAL://localhost:909x
Listener Security Protocol Map	LISTENER_DOCKER_INTERNAL:PLAINTEXT, LISTENER_DOCKER_EXTERNAL:PLAINTEXT
Broker ID	kafka1: 1 kafka2: 2 kafka3: 3
ZooKeeper Connect	zoo1:2181,zoo2:2182, zoo3:2183/kafka
Auto Leader Rebalance Enable	true
Leader Imbalance Check Interval	180

Offsets Topic Replication Factor	2
Min Insync Replicas	2
ZooKeeper Session Timeout ms	30000
Request Timeout ms	60000
Auto Create Topic Enable	true
Delete Topic Enable	true
Compression Type	Producer
Log Retention Hours	700
Log Segment Byte	1073741824 (1 GB)
Transaction State Log Min Isr	2
Transaction State Log Replication Factor	2
Log Cleanup Policy	delete
Log Retention Bytes	134217728 (~134 MB)
Confluent Metrics Reporter Bootstrap Servers	kafka1:19092,kafka2:19093,kafka3:19094
Confluent Metrics Reporter Topic Create	true
Confluent Metrics Reporter Max Request Size	134217728 (~134 MB)

Service Dependencies
zoo1 – Service Healthy
zoo2 – Service Healthy
zoo3 – Service Healthy

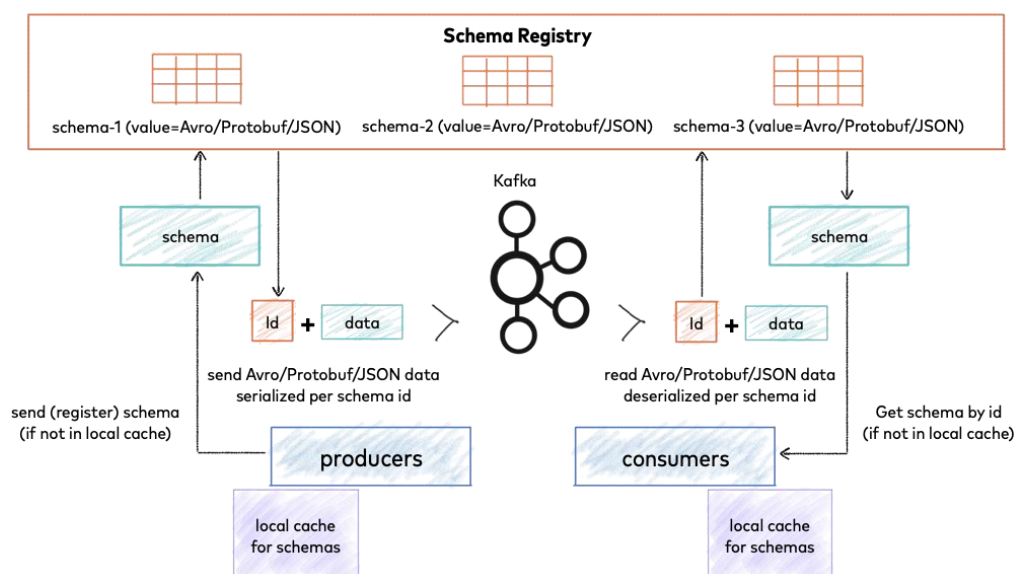
Being the core of the cluster and having multiple components and services communicate with the Brokers, an healthcheck option was added to the brokers, to ensure a clean general startup.

Healthcheck Parameters	
Test Condition	Exit 0
Start Period Value	30s
Interval Value	30s

1.6 Kafka Schema Registry

As Producers write data to topics and Consumers read data from them, there is an implicit “contract”: the written data must have a schema that can be read by consumers, even as producers and consumers evolve their schemas.

The Schema Registry helps ensure that this property is valid through compatibility checks and centralized schema management.



1.6.1 Schema Registry Configuration

Schema Registry Configuration	
Schema Registry Version	6.0.1
Restart Option	on-failure
Hostname	kafka-schema-registry
Registry Kafkastore Bootstrap Servers	PLAINTEXT://kafka1:19092, PLAINTEXT://kafka2:19093, PLAINTEXT://kafka3:19094
Port Mapping	8081:8081
Registry Listeners	http://0.0.0.0:8081
Schema Compatibility Level	full
Debug	true

Service Dependencies
zoo1 – Service Healthy
zoo2 – Service Healthy
zoo3 – Service Healthy
kafka1 – Service Healthy
kafka2 – Service Healthy
kafka3 – Service Healthy

Just as the brokers, an healthcheck was implemented to control the service status during startup.

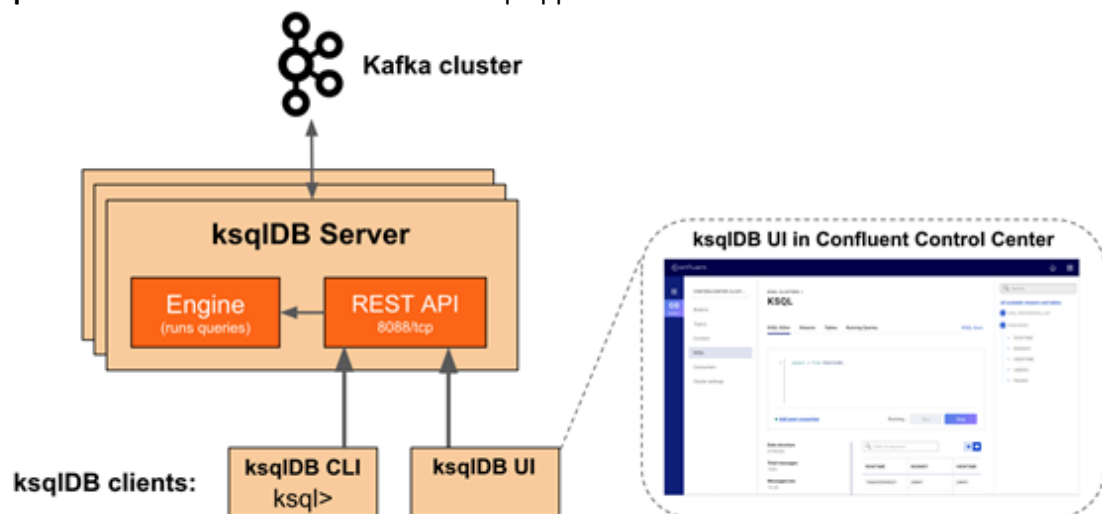
Healthcheck Parameters	
Test Condition	Exit 0
Start Period Value	10s
Interval Value	10s

1.7 Kafka KsqlDB

KsqlDB is a component of the Kafka ecosystem that allows to query, read, write, and process data in Apache Kafka in real-time and at scale using only a lightweight SQL-like syntax.

The main components of KsqlDB are:

- **ksqlDB engine** to allow the processing of the kql statements and queries
- **REST interface** to enable client access to the engine.
- **ksqlDB CLI** to interact with the engine by using a console of familiar format in respect to other famous commercial products.
- **ksqlDB UI** to offer a user interface to develop applications in the Control Center



1.7.1 KsqlDB Configuration

Schema Registry Configuration	
KsqlDB Version	6.0.1
Restart Option	on-failure
Hostname	ksqldb-server
Bootstrap Servers	PLAINTEXT://kafka1:19092, PLAINTEXT://kafka2:19093, PLAINTEXT://kafka3:19094
Port Mapping	8088:8088
Ksql Listeners	http://0.0.0.0:8088
Schema Registry URL	http://0.0.0.0:8081
Producer Interceptor Classes	io.confluent.monitoring.clients.interceptor.MonitoringProducerInterceptor
Consumer Interceptor Classes	io.confluent.monitoring.clients.interceptor.MonitoringConsumerInterceptor
Logging Processing Topic Replication Factor	1
Logging Processing Topic Auto Create	true
Logging Processing Stream Auto Create	true

Service Dependencies
kafka1 – <i>Service Healthy</i>
kafka2 – <i>Service Healthy</i>
kafka3 – <i>Service Healthy</i>
kafka-schema-registry – <i>Service Healthy</i>

Just as the brokers, an healthcheck was implemented to control the service status during startup.

Healthcheck Parameters	
Test Condition	Exit 0
Start Period Value	15s
Interval Value	20s

1.8 KsqlDB CLI

The client interface to communicate with the ksqldb server was also implemented. This way, users can run queries or create streams without accessing the control center, only using the command line tool.

```
OpenJDK 64-Bit Server VM warning: Option UseConcMarkSweepGC was deprecated in version 9.0 and will likely be removed in a future release.

=====
=                               =
=  KSQL                         =
=                               =
= Streaming SQL Engine for Apache Kafka® =
=====

Copyright 2017-2018 Confluent Inc.

CLI v5.3.2, Server v5.3.2 located at http://localhost:8088

Having trouble? Type 'help' (case-insensitive) for a rundown of how things work!

ksql>
```

1.8.1 KsqlDB CLI Configuration

KSQLDB CLI Configuration	
KsqlDB CLI Version	6.0.1
Restart Option	on-failure
Hostname	ksqldb-cli
entrypoint	/bin/sh
tty	true

Service Dependencies
ksqldb-server – <i>Service Healthy</i>

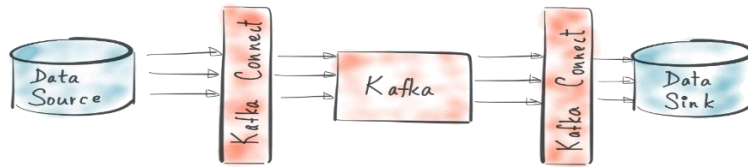
1.9 Kafka Connect

Kafka Connect is an open-source tool that acts as a centralized data hub for simple data integration between databases, key-value stores, search indexes, and file systems and Apache Kafka by making use of Connector from large variety to ingest or deliver the data.

Kafka Connect includes two types of connectors:

- **Source connectors** to ingest entire databases and streams table updates to Kafka topics.
- **Sink connectors** to deliver data from Kafka topics into secondary indexes such as Elasticsearch, or batch systems such as Hadoop for offline analysis.

KAFKA CONNECT



1.9.1 Kafka Connect Configuration

Schema Registry Configuration	
Connect Version	6.0.1
Restart Option	on-failure
Hostname	kafka-connect
Bootstrap Servers	PLAINTEXT://kafka1:19092, PLAINTEXT://kafka2:19093, PLAINTEXT://kafka3:19094
Port Mapping	8083:8083
REST Advertised Hostname	kafka-connect
Config Storage Topic	__docker-connect-configs
Offset Storage Topic	__docker-connect-offsets
Status Storage Topic	__docker-connect-status
Key Converter Schema Registry URL	http://0.0.0.8081
Key Converter	io.confluent.connect.avro.AvroConverter
Value Converter Schema Registry URL	http://0.0.0.8081
Value Converter	io.confluent.connect.avro.AvroConverter
Internal Key Converter	org.apache.kafka.connect.json.JsonConverter
Internal Value Converter	org.apache.kafka.connect.json.JsonConverter
Config Storage Replication Factor	2
Config Offset Replication Factor	2

Config Status Replication Factor	2
----------------------------------	---

Service Dependencies
zoo1 – Service Healthy
zoo2 – Service Healthy
zoo3 – Service Healthy
kafka1 – Service Healthy
kafka2 – Service Healthy
kafka3 – Service Healthy
kafka-schema-registry – Service Healthy

Healthcheck Parameters	
Test Condition	Exit 0
Start Period Value	10s
Interval Value	15s

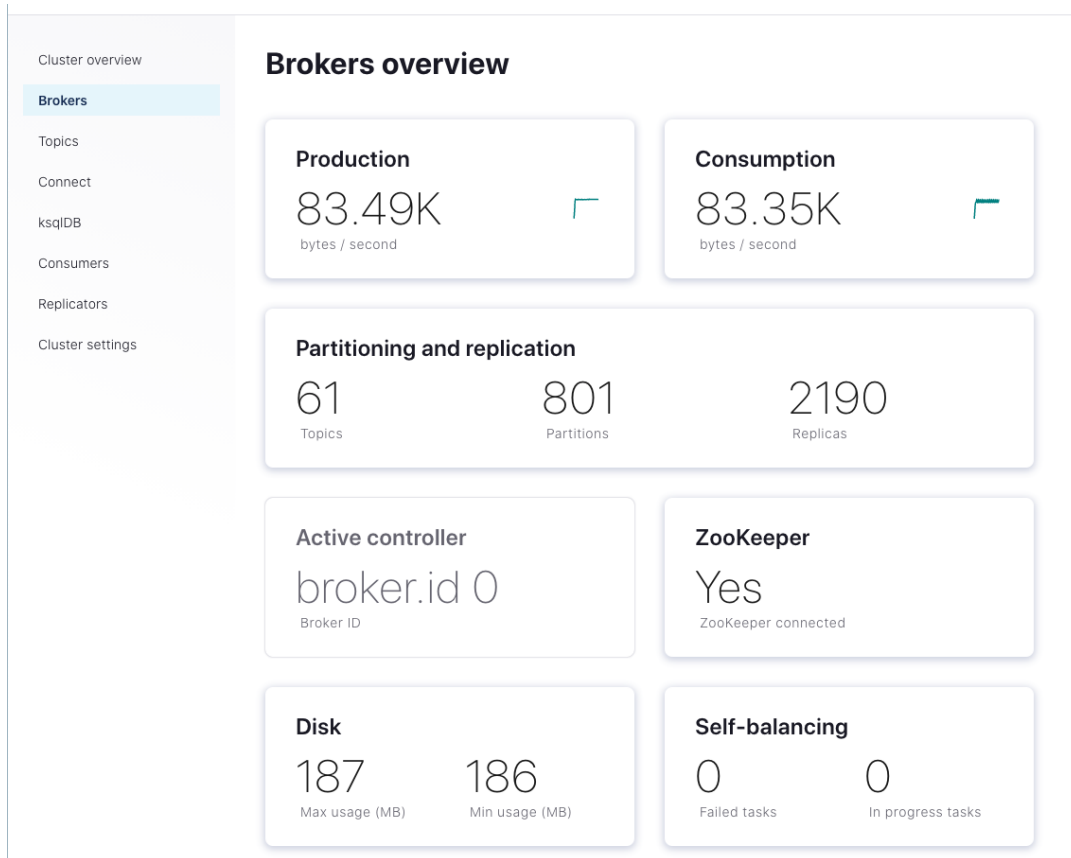
1.10 Kafka Control Center

Confluent Control Center is a web-based tool for managing and monitoring Apache Kafka, by providing a user interface that allows developers and operators to get a quick overview of cluster health, observe and control messages, topics, and Schema Registry, and to develop and run ksqlDB queries.

Control Center is comprised of three fundamental parts:

- Metrics interceptors that collect metric data on clients (producers and consumers).
- Kafka to move metric data.
- The Control Center application server for analyzing stream metrics.

The interceptors work by collecting metrics on messages produced or consumed on each client and sending these to Control Center for analysis and reporting. Interceptors use Kafka message timestamps to group messages.



1.10.1 Control Center Configuration

Schema Registry Configuration	
Control Center Version	6.0.1
Restart Option	on-failure
Hostname	control-center
Bootstrap Servers	PLAINTEXT://kafka1:19092, PLAINTEXT://kafka2:19093, PLAINTEXT://kafka3:19094
Port Mapping	9021:9021
Connect Cluster	kafka-connect:8083
KsqlDB URL	http://localhost:8088
KsqlDB Advertised URL	http://localhost:8088
Schema Registry URL	http://kafka-schema-registry:8081
Replication Factor	2

Internal Topics Partitions	1
Monitoring Interceptor Topic Partitions	1
Metrics Topic Replication	1
Metrics Topic Partitions	1

Service Dependencies
kafka1 – Service Healthy
kafka2 – Service Healthy
kafka3 – Service Healthy
kafka-schema-registry – Service Healthy
kafka-connect – Service Healthy

1.11 InfluxDB

A time series database written in Go and optimized for fast, high-availability storage and retrieval of time series data in fields such as operations monitoring, application metrics.

Aside the image for InfluxDB, a supporting image by emanuelefalzone for an easier setup of InfluxDB configuration, has also been implemented.

InfluxDB Configuration	
InfluxDB Version	2.0.2
Restart Option	on-failure
Hostname	influxdb
Port	8086:8086

InfluxDB Sidecar Configuration	
InfluxDB Sidecar Version	1.0.1
Restart Option	on-failure

Hostname	influxdb-sidecar-setup
Influx Retantion Period	0
Influx Port	8086
Influx Username	admin
Influx Password	quantia-analytics
Influx Token	d2VsY29tZQ==
Influx Organization	http://kafka-schema-registry:8081
Influx Bucket	usde_project

1.12 Additional Components

As the Control Center is an enterprise component, meaning a license has to be bought to be used for a long period of time, I decided to incorporate into my system a few ‘free options’ as a substitute for some of the Control Center’s functions.

1.12.1 Schema Registry UI

As a way to get a very simple user interface to manage the Kafka’s schema registry, I included landoop’s Schema Registry UI.

It offers the possibility of easily creating new schemas from scratch, importing from external sources or exporting them.

The screenshot displays the Schema Registry UI interface. On the left, a sidebar shows '0 Schemas' and a search bar. The main area is titled 'New Subject' and contains a form for creating a new schema. The 'Subject Name' field is populated with 'my-new-topic-value'. Below this, the 'Schema' field is active, showing a sample Avro schema for a record named 'evolution' in the 'com.landoop' namespace. The schema includes three fields: 'name' (string), 'number1' (int), and 'number2' (float). A 'VALIDATE' button is located at the bottom right of the schema editor.

```

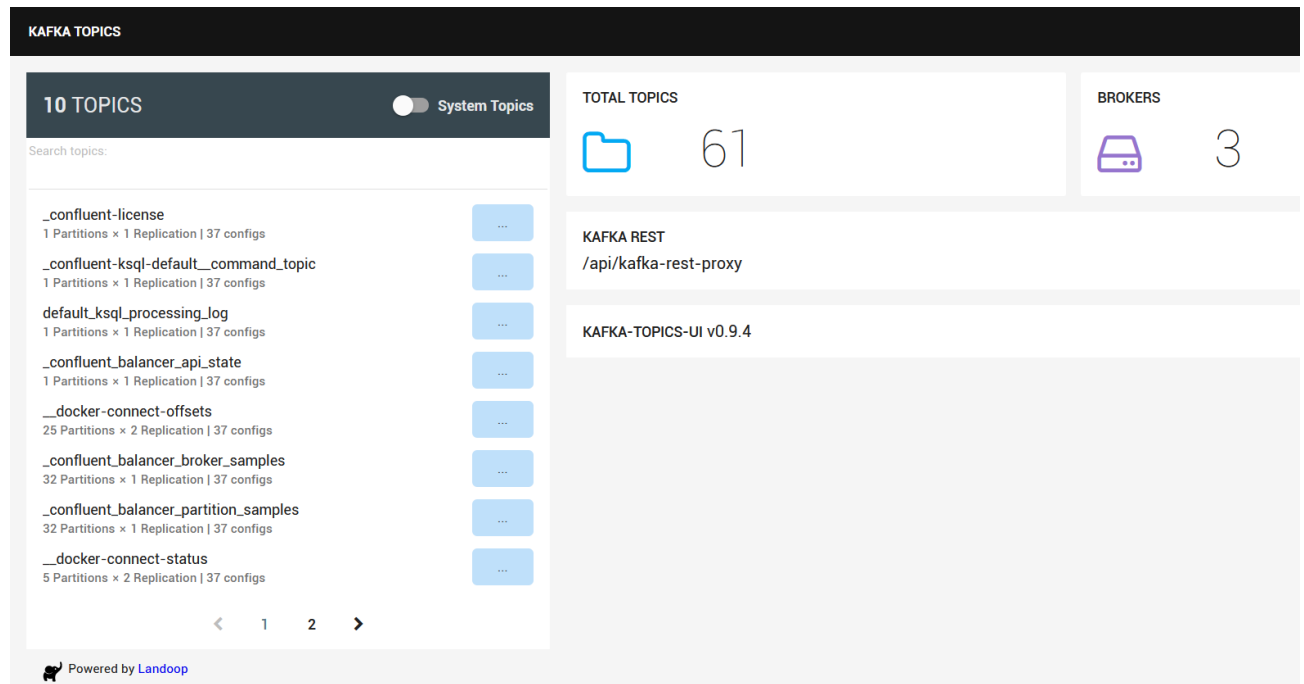
1 {
2   "type": "record",
3   "name": "evolution",
4   "doc": "This is a sample Avro schema to get you started. Please edit!",
5   "namespace": "com.landoop",
6   "fields": [
7     {
8       "name": "name",
9       "type": "string"
10    },
11    {
12      "name": "number1",
13      "type": "int"
14    },
15    {
16      "name": "number2",
17      "type": "float"
18    }
19  ]
20 }

```

1.12.2 REST Proxy & Topics UI

As a way to get a very simple user interface to manage the Kafka's topics I included landoop's Topics UI and Kafka's REST Proxy that it uses.

It offers the possibility of easily creating new topics and inspecting the ones already present.



2. System startup process

2.2 Load input files

The first step consists of transferring the six input files into the correct folders for the connectors to be able to ingest them.

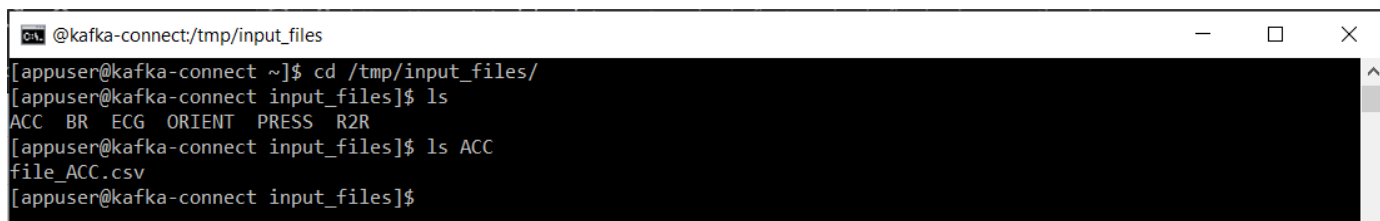
This command can be executed from the command prompt and will copy the files directly into the kafka-connect Docker container by using its container name as the connection point:

- `docker cp file_name.csv kafka-connect:/tmp/input_files/name/file_name.csv`

After having run the command for every input file, we can then enter the bash of the container with the command:

- `docker-compose -f kafka_cluster.yml exec kafka-connect bash`

If the previous commands ended correctly, you should end up with seeing the files locally:



```
@kafka-connect/tmp/input_files
[appuser@kafka-connect ~]$ cd /tmp/input_files/
[appuser@kafka-connect input_files]$ ls
ACC BR ECG ORIENT PRESS R2R
[appuser@kafka-connect input_files]$ ls ACC
file_ACC.csv
[appuser@kafka-connect input_files]$
```

2.3 Startup Kafka Cluster

To start the Kafka ecosystem services, after positioning at the correct directory, execute this command:

- `docker-compose -f kafka_cluster.yml up`

All the services listed inside the Yaml file will start with the order and dependencies specified, from the servers to the user interfaces.

2.3.1 ZooKeeper Servers

The first thing to check is the state of the ZooKeeper servers, as their job is crucial in managing the Kafka cluster.

One way to do this is to use the additional tool **netcat**: this tool enables the usage of four letters words, commands for monitoring and other purposes.

- Remember to add the tool's path to your PATH environment variable, so to invoke the tool commands from wherever in your system.
 - e.g: `C:\Users\omara\Downloads\netcat-win32-1.11\netcat-1.11`

If we open the command prompt and, for each ZooKeeper server, we invoke the command:

- `echo stat | nc address port_number`

We will be able to see the statistics of each server, as well as the state (Leader/Follower).

This is an example of what will be shown:

```

C:\Users\omara\OneDrive\Desktop\USDE_PROJECT>echo stat | nc localhost 2181
Zookeeper version: 3.4.9-1757313, built on 08/23/2016 06:50 GMT
Clients:
  /172.31.0.1:55654[0](queued=0,recved=1,sent=0)
  /172.31.0.6:41750[1](queued=0,recved=6905,sent=6927)
  /172.31.0.6:42124[1](queued=0,recved=376,sent=376)

Latency min/avg/max: 0/1/134
Received: 9261
Sent: 9298
Connections: 3
Outstanding: 0
Zxid: 0x20000048e
Mode: follower
Node count: 781

C:\Users\omara\OneDrive\Desktop\USDE_PROJECT>echo stat | nc localhost 2182
Zookeeper version: 3.4.9-1757313, built on 08/23/2016 06:50 GMT
Clients:
  /172.31.0.6:60188[1](queued=0,recved=312,sent=313)
  /172.31.0.1:58246[0](queued=0,recved=1,sent=0)

Latency min/avg/max: 0/0/21
Received: 327
Sent: 327
Connections: 2
Outstanding: 0
Zxid: 0x20000048e
Mode: follower
Node count: 781

C:\Users\omara\OneDrive\Desktop\USDE_PROJECT>echo stat | nc localhost 2183
Zookeeper version: 3.4.9-1757313, built on 08/23/2016 06:50 GMT
Clients:
  /172.31.0.7:52630[1](queued=0,recved=1122,sent=1122)
  /172.31.0.8:42332[1](queued=0,recved=6230,sent=6246)
  /172.31.0.1:59418[0](queued=0,recved=1,sent=0)

Latency min/avg/max: 0/0/24
Received: 7432
Sent: 7447
Connections: 3
Outstanding: 0
Zxid: 0x20000048e
Mode: leader
Node count: 781

```

As you can see, the current Leader node is the third instance of Zookeeper, zoo3, instantiated on port 2183.

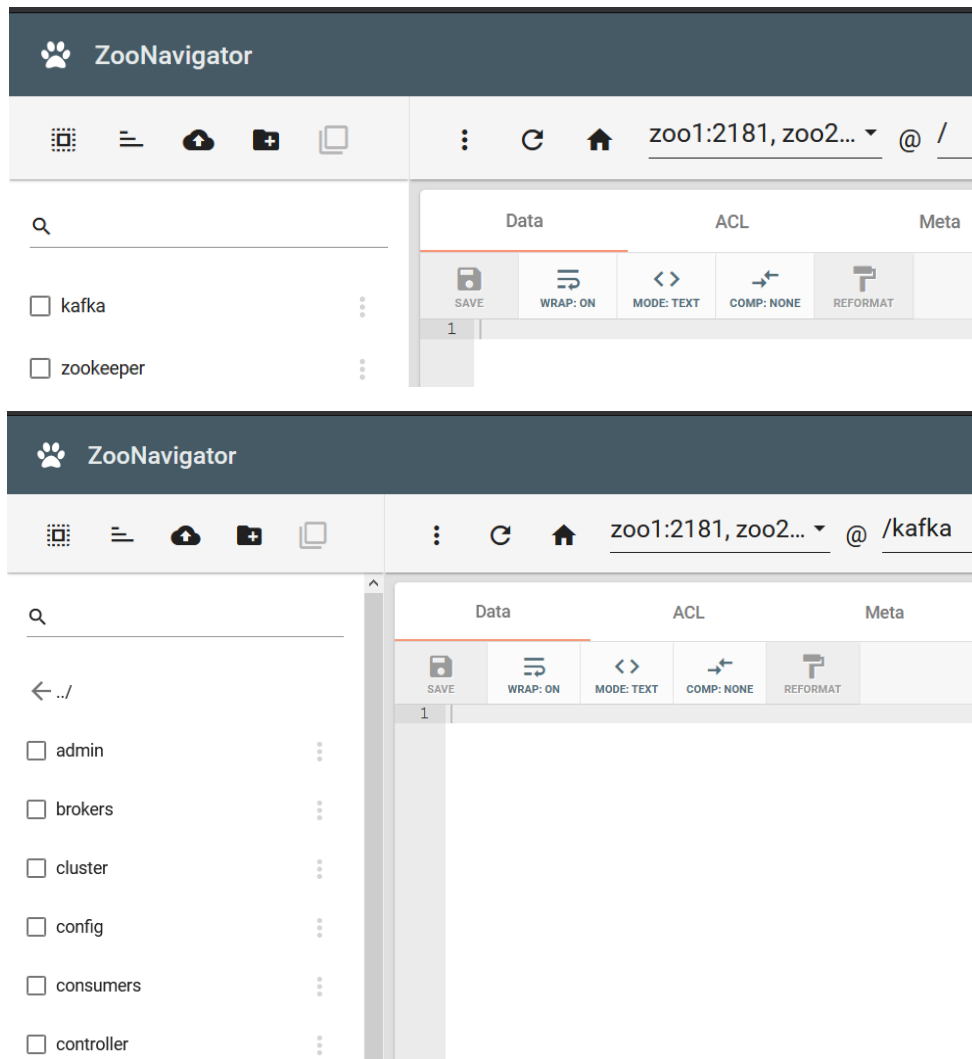
2.3.2 ZooNavigator

What we now want to check is the structure of zNodes of the ZooKeepers servers, to be sure that all the folders have been created properly.

We can access the service with this address:

- <http://localhost:8004>

As we set a chroot `‘/kafka’` into the Kafka brokers configurations, you should be able to see a situation like this:



2.3.3 Topics UI & Schema UI

The basic UIs of the brokers' topics and on the schema registry can be accessed with the following addresses:

- Topics UI
 - `http://localhost:8000`
- Schema Registry UI
 - `http://localhost:8001`

2.3.4 KsqlDB CLI

To test if the CLI to the ksqldb server instance can connect correctly, open the command prompt and run the following command:

- `docker-compose -f kafka_cluster.yml exec ksqldb-cli ksql http://ksqldb-server:8088`

If both the server and the CLI started properly, this is the screen that should appear:

```
C:\Users\omara\OneDrive\Desktop\USDE_PROJECT>docker-compose -f kafka_cluster.yml exec ksqldb-cli ksql http://ksqldb-server:8088
OpenJDK 64-Bit Server VM warning: Option UseConcMarkSweepGC was deprecated in version 9.0 and will likely be removed in a future release.

=====
=                                     =
=  [K] [E] [A] [F] [K] [A] [A] [A]  =
=  [A] [A] [A] [A] [A] [A] [A] [A]  =
=                                     =
=  Event Streaming Database purpose-built  =
=    for stream processing apps          =
=====

Copyright 2017-2020 Confluent Inc.

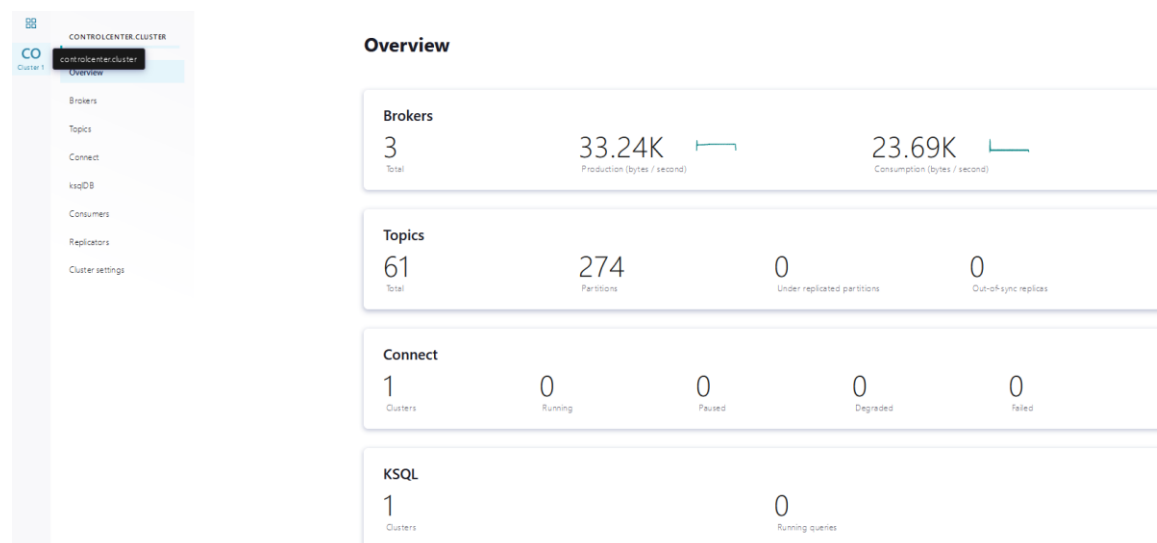
CLI v0.15.0, Server v6.0.1 located at http://ksqldb-server:8088
Server Status: <unknown>

Having trouble? Type 'help' (case-insensitive) for a rundown of how things work!

ksql>
```

2.3.5 Control Center

As we implemented a central, monitoring station to monitor the entire Kafka cluster, we can easily see, at any point, how all the components are faring.



In this screen is represented the situation of our cluster. The brokers, KSQLDB server and Connect are up and running and no topics seems to be under-replicated.

2.4 InfluxDB

Accessing the address:

- localhost:8086

The credentials to enter are the environment variables specified in the influx_sidecar image: it set the credentials, admin token and created a bucket with the specified name.

3. Connectors Installation

The Kafka Connect image that we have pulled down and downloaded does not natively implement the connectors that we need in our data pipeline.

For this reason, we have to download the connectors directly from the Confluent Hub and then add them to the container.

We need two connectors:

- A FilePulse Source Connector, to ingest the data
 - <https://www.confluent.io/hub/streamthoughts/kafka-connect-file-pulse>
- An InfluxDB Sink Connector, to deliver the aggregated data
 - <https://www.confluent.io/hub/confluentinc/kafka-connect-influxdb>

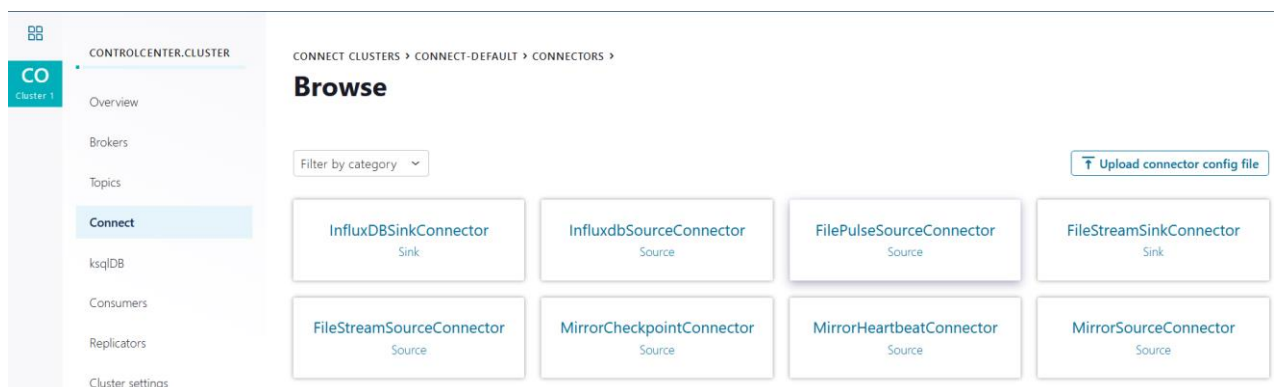
After having downloaded the two .zip files, we need to extract them and then copy to the kafka-connect container.

Run the following command:

- `docker cp filepulse_connector kafka-connect:/etc/kafka-connect/jars/filepulse_connector`
- `docker cp influxdb_connector kafka-connect:/etc/kafka-connect/jars/influxdb_connector`

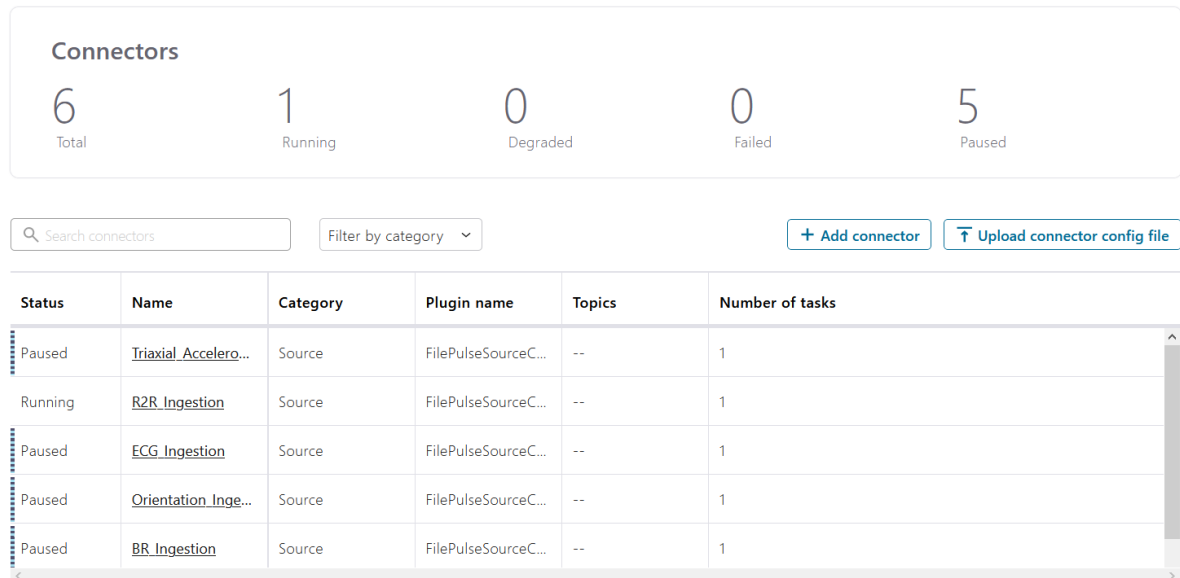
And then restart Kafka Connect to permit to the worker to receive the change.

If the process was successful, you should see the connectors listed when opening the Kafka Connect section in the Control Center.



The connectors configurations are included inside .properties files stored into the folder *Connectors_conf_files*.

They can be easily directly imported to create the instances, all configured and ready.



4. Data Exploration

The data that was made available comes from IoT wearables that provide information about the user health status and position by measuring his vitals and spatial position. In details, we have **seven data sources**, each represented by a distinct **.csv** file with its own schema.

File Name	Data source Meaning	Number of rows (with header)
<i>file_ECG</i>	Electrocardiogram values	107393
<i>file_R2R</i>	Hearth rate values	1894
<i>file_PRESS</i>	Respiratory signal values	10973
<i>file_BR</i>	Respiratory frequency values	317
<i>file_ACC</i>	Triaxial Accelerometer values	7425
<i>file_ORIENT</i>	User position in the space values	465

file_ECG schema

timestamp	inx	sampleNum	HeartRate [bpm]	SensorStatus	RRlenght [ms]
-----------	-----	-----------	-----------------	--------------	---------------

- **timestamp:** It describes the exact time the data packet was ingested from the sensor. The ingestion rate is 1 second and it spans over fourteen minutes and four seconds.
- **inx:** Incremental index identifying a single data row of each data packet. Each data packet contains 128 different values.
- **sampleNum:** It identifies the ingested data with a number that grows by one with each new data packet associated to the event.
- **HeartRate:** Number of heartbeats measured per minute.
- **SensorStatus:** Its value describes if the sensor is operational or not. As a numerical value, '255' indicates that the sensor was fully functional and is the standard.
- **RRlenght:** It expresses the amount of time between successive heartbeats, measured in milliseconds.

file_R2R schema

timestamp	sampleNum	HeartRate [bpm]	rrLen [ms]
-----------	-----------	-----------------	------------

- **timestamp** It describes the exact time the data packet was ingested from the sensor. The ingestion rate is 1 second and it spans over fourteen minutes and four seconds.
- **sampleNum:** It identifies the ingested data with a number that grows by one with each new data packet associated to the event.
- **HeartRate:** It describes the value of the *heart rate*, measured in beats per minute.
- **rrLen:** It expresses the amount of time between successive heartbeats, measured in milliseconds.

file_ORIENT schema

timestamp	orientation
-----------	-------------

- **timestamp:** It describes the exact time the data packet was ingested from the sensor. The ingestion rate is 2 second and it spans over fourteen minutes and three seconds.
- **orientation:** It expresses with a non-negative value the position of the user in the space.

file_ACC schema

timestamp	inx	sampleNum	Samples X	Sample Y	Sample Z
-----------	-----	-----------	-----------	----------	----------

- **timestamp:** It describes the exact time the data was ingested from the sensor. The ingestion rate is 2 second and it spans over fourteen minutes and three seconds.
- **inx:** Incremental index identifying a single data row of each data packet. Each data packet contains 16 different values.
- **sampleNum:** It identifies the ingested data with a number that grows by one with each new data packet associated to the event.
- **Samples X/Y/Z:** Measured acceleration in respect of the relative axes.

file_PRESS schema

timestamp	inx	sampleNum	breathRate [breaths/min]	filterType	dataType
-----------	-----	-----------	-----------------------------	------------	----------

- **timestamp:** It describes the exact time the data packet was ingested from the sensor. The ingestion rate is 2 second and it spans over fourteen minutes and four seconds.
- **inx:** Incremental index identifying a single data row of each data packet. Each data packet contains 13 different values.
- **sampleNum:** It identifies the ingested data with a number that grows by one with each new data packet associated to the event.
- **breathRate:** Its value identifies the breath rate, described in *breaths per minute*.
- **filterType:** Integer value that identifies the type of filter used in the measurement.
- **dataType:** Integer value that identifies the type of data that was measured.

file_BR schema

timestamp	sampleNum	brInterval [samples]	breathRate [breaths/min]
-----------	-----------	-------------------------	-----------------------------

- **timestamp:** It describes the exact time the data was ingested from the sensor. The value is fixed at 1580000000.
- **sampleNum:** It identifies the ingested data with a number that grows by one with each new data packet associated to the event.
- **brInterval:** It identifies the interval of a breath, described in *number of samples*.
- **breathRate:** Its value identifies the breath rate, described in *breaths per minute*.

4.1. Data Issues

There are three characteristics of the data sources that immediately jump out:

1. Some of the column names have spaces in them, this might cause problems during the ingestion phase.
2. The amount of data of between them is vastly different in quantity.
3. Even in a single file, is hard to tell if a measurement belongs to a certain individual across the timestamps.
4. There is no evident relationship between the data sources that would enable us to perform an analysis over multiple files at the same time.

We lack information about how the measurements have been conducted and how specific options, like the filters used in the analysis or the sampling frequency, impacted the data collection process.

We also do not have available any information on the quality of the ECG signals (P wave, T wave, QRS complex) to assess if the data is reliable or not.

- As we are missing any reference to the age of the subjects, we will take as reference the parameters of an average, not particularly trained, adult.

4.2. Data Insight

An interesting characteristic in the data that can be further analyzed is the **RRLength**, present in both the ECG and R2R data sources.

From this value, we can approximately measure the **Heart Rate Variability (HRV)** of a subject, an accurate, non-invasive measure of the Autonomic Nervous System, which corresponds to personal parameters such as fitness and stress levels.

By analyzing the raw data in our system and sending the results to, possibly, a Machine Learning algorithm, we have the possibility of learning what is the 'normality' for each user and spot anomalous patterns beforehand.

The **Root Mean Squared of the Successive Differences (rMSSD)** is one the useful parameters, in the domain of time, that can be calculated to assess the users' wellbeing.

$$rMSSD = \sqrt{\sum_{i=2}^{N-1} \frac{1}{N-1} (RR_{i+1}^2 - RR_i^2)}$$

A good estimate should be done on values over twenty-four hours, but we lack de amount of data to do it, so we will run our estimation over ten seconds intervals.

As the data is coming from IoT healthcare wearables, the data was most likely collected from the subjects during 'free-living' conditions, meaning that the subjects were carrying

out ordinary everyday activities.

Thus, quality assessment algorithms can also take into consideration the data coming from the accelerometer, besides ECG signal, recognizing particular activities like running that could have an impact on the values collected.

- We consider normal a heart rate in the interval 60-100 [bpm] in resting condition, with the range 90-100 [bpm] being a possible warning.
- We instead consider acceptable a heart rate of 120 – 150 [bpm] during physically demanding activities, like exercising.

By analyzing the value of the breath rate, we can possibly spot abnormal or even dangerous conditions like *hyperventilation* or *pneumonia* and generate a warning to the related subject.

- A normal breathing rate will be considered in the range of 12 – 20 [breaths/min] in resting conditions
- During exercises, depending on the intensity, we consider acceptable a breath rate in the interval 30 – 50 [breaths/min].

If we had a reference in data sources to match a single subject, we could also monitor the **ratio between heart rate and breath rate**: for a healthy subject, the ratio should stay close to 4:1, during both moments of exercise and relaxation.

5. Data Ingestion

After having configured the connectors, we now have to create the topics that will receive the data ingested.

There are six topics to create, one for each file:

- *Topic list*
 - *file_ACC*
 - *file_BR*
 - *file_ECG*
 - *file_ORIENT*
 - *file_PRESS*
 - *file_R2R*

The topics can be created directly into the control center or by executing the command on the brokers:

- `bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor X --partitions Y --topic topic-name`

Now that the topics are created and the connectors are instantiated, if we start them, the topics will get the relative messages.

Editor Flow Streams Tables Running queries

1 PRINT 'file_BR' FROM BEGINNING;

[Add query properties](#) Running... Stop Run query

Data structure
--
Total messages
--
Messages/sec
--
Total message bytes
--
Message fields

▶ ⏸

record

rowtime: 2021/02/23 17:26:14.007 Z, key: <null>, value: {"timestamp": 1580000000, "sampleNum": "10963", "brInterval": ...}

rowtime: 2021/02/23 17:26:14.007 Z, key: <null>, value: {"timestamp": 1580000000, "sampleNum": "10929", "brInterval": ...}

rowtime: 2021/02/23 17:26:14.007 Z, key: <null>, value: {"timestamp": 1580000000, "sampleNum": "10898", "brInterval": ...}

Queried topic from the Control Center

```
ksql> PRINT 'file_R2R' FROM BEGINNING;
Key format: "\(\B )/" - no data processed
Value format: AVRO
rowtime: 2021/02/23 17:30:41.243 Z, key: <null>, value: {"timestamp": 1579604534, "sampleNum": "452", "heartRate": 108, "rrLen": 554}
rowtime: 2021/02/23 17:30:41.243 Z, key: <null>, value: {"timestamp": 1579604535, "sampleNum": "453", "heartRate": 103, "rrLen": 578}
rowtime: 2021/02/23 17:30:41.243 Z, key: <null>, value: {"timestamp": 1579604535, "sampleNum": "454", "heartRate": 101, "rrLen": 593}
rowtime: 2021/02/23 17:30:41.243 Z, key: <null>, value: {"timestamp": 1579604536, "sampleNum": "455", "heartRate": 106, "rrLen": 562}
rowtime: 2021/02/23 17:30:41.243 Z, key: <null>, value: {"timestamp": 1579604536, "sampleNum": "456", "heartRate": 114, "rrLen": 523}
rowtime: 2021/02/23 17:30:41.243 Z, key: <null>, value: {"timestamp": 1579604537, "sampleNum": "457", "heartRate": 116, "rrLen": 515}
rowtime: 2021/02/23 17:30:41.243 Z, key: <null>, value: {"timestamp": 1579604537, "sampleNum": "458", "heartRate": 114, "rrLen": 523}
rowtime: 2021/02/23 17:30:41.243 Z, key: <null>, value: {"timestamp": 1579604538, "sampleNum": "459", "heartRate": 112, "rrLen": 531}
rowtime: 2021/02/23 17:30:41.243 Z, key: <null>, value: {"timestamp": 1579604539, "sampleNum": "460", "heartRate": 111, "rrLen": 539}
rowtime: 2021/02/23 17:30:41.243 Z, key: <null>, value: {"timestamp": 1579604539, "sampleNum": "461", "heartRate": 108, "rrLen": 554}
rowtime: 2021/02/23 17:30:41.243 Z, key: <null>, value: {"timestamp": 1579604540, "sampleNum": "462", "heartRate": 108, "rrLen": 554}
rowtime: 2021/02/23 17:30:41.243 Z, key: <null>, value: {"timestamp": 1579604540, "sampleNum": "463", "heartRate": 106, "rrLen": 562}
rowtime: 2021/02/23 17:30:41.243 Z, key: <null>, value: {"timestamp": 1579604541, "sampleNum": "464", "heartRate": 108, "rrLen": 554}
rowtime: 2021/02/23 17:30:41.243 Z, key: <null>, value: {"timestamp": 1579604541, "sampleNum": "465", "heartRate": 102, "rrLen": 585}
rowtime: 2021/02/23 17:30:41.243 Z, key: <null>, value: {"timestamp": 1579604542, "sampleNum": "466", "heartRate": 98, "rrLen": 609}
rowtime: 2021/02/23 17:30:41.243 Z, key: <null>, value: {"timestamp": 1579604543, "sampleNum": "467", "heartRate": 92, "rrLen": 648}
rowtime: 2021/02/23 17:30:41.243 Z, key: <null>, value: {"timestamp": 1579604543, "sampleNum": "468", "heartRate": 97, "rrLen": 617}
rowtime: 2021/02/23 17:30:41.243 Z, key: <null>, value: {"timestamp": 1579604544, "sampleNum": "469", "heartRate": 105, "rrLen": 570}
rowtime: 2021/02/23 17:30:41.243 Z, key: <null>, value: {"timestamp": 1579604544, "sampleNum": "470", "heartRate": 109, "rrLen": 546}
rowtime: 2021/02/23 17:30:41.243 Z, key: <null>, value: {"timestamp": 1579604545, "sampleNum": "471", "heartRate": 111, "rrLen": 539}
rowtime: 2021/02/23 17:30:41.243 Z, key: <null>, value: {"timestamp": 1579604545, "sampleNum": "472", "heartRate": 112, "rrLen": 531}
rowtime: 2021/02/23 17:30:41.243 Z, key: <null>, value: {"timestamp": 1579604546, "sampleNum": "473", "heartRate": 112, "rrLen": 531}
rowtime: 2021/02/23 17:30:41.243 Z, key: <null>, value: {"timestamp": 1579604546, "sampleNum": "474", "heartRate": 116, "rrLen": 515}
rowtime: 2021/02/23 17:30:41.243 Z, key: <null>, value: {"timestamp": 1579604547, "sampleNum": "475", "heartRate": 120, "rrLen": 500}
rowtime: 2021/02/23 17:30:41.243 Z, key: <null>, value: {"timestamp": 1579604547, "sampleNum": "476", "heartRate": 120, "rrLen": 500}
rowtime: 2021/02/23 17:30:41.243 Z, key: <null>, value: {"timestamp": 1579604548, "sampleNum": "477", "heartRate": 116, "rrLen": 515}
rowtime: 2021/02/23 17:30:41.243 Z, key: <null>, value: {"timestamp": 1579604549, "sampleNum": "478", "heartRate": 106, "rrLen": 562}
rowtime: 2021/02/23 17:30:41.243 Z, key: <null>, value: {"timestamp": 1579604549, "sampleNum": "479", "heartRate": 90, "rrLen": 664}
rowtime: 2021/02/23 17:30:41.243 Z, key: <null>, value: {"timestamp": 1579604550, "sampleNum": "480", "heartRate": 81, "rrLen": 734}
rowtime: 2021/02/23 17:30:41.243 Z, key: <null>, value: {"timestamp": 1579604551, "sampleNum": "481", "heartRate": 79, "rrLen": 757}
rowtime: 2021/02/23 17:30:41.243 Z, key: <null>, value: {"timestamp": 1579604551, "sampleNum": "482", "heartRate": 80, "rrLen": 742}
rowtime: 2021/02/23 17:30:41.243 Z, key: <null>, value: {"timestamp": 1579604552, "sampleNum": "483", "heartRate": 88, "rrLen": 679}
rowtime: 2021/02/23 17:30:41.243 Z, key: <null>, value: {"timestamp": 1579604553, "sampleNum": "484", "heartRate": 99, "rrLen": 601}
```

Queried topic from the KSQLDB CLI

6. KSQL Queries

Before running the KSQL queries, we need to create Streams out of the topics.

We can do that directly from the Control Center or from the Ksqldb CLI with the command:

- `CREATE STREAM stream_name WITH (KAFKA_TOPIC='topic_name', VALUE FORMAT='AVRO');`

We can see the Streams that we have created by running from the Ksqldb CLI the command:

- `SHOW STREAMS;`

If everything went alright, six streams should be displayed, one for each source topic.

```
ksql> SHOW STREAMS;
```

Stream Name	Kafka Topic	Key Format	Value Format	Windowed
ACC_BR	file_BR	KAFKA	AVRO	false
ECG_STREAM	file_ECG	KAFKA	AVRO	false
KSQL_PROCESSING_LOG	default_ksql_processing_log	KAFKA	JSON	false

```
ksql>
```

From the Control Center, they would look like this:

Topic

Topic name	file_ECG
Replication	3
Partitions	1
Serialization	AVRO

Schema

Name	Type
TIMESTAMP	BIGINT
INX	STRING
SAMPLENUM	STRING
FREQUENCY	INTEGER
HEARTRATE	INTEGER
SENSORSTATUS	STRING
RRLNGTH	INTEGER
SAMPLES	STRING

A simple query to show the inside of our streams:

- *SELECT * FROM stream_name EMIT CHANGES LIMIT 10;*

```
ksql> show topics;
```

Kafka Topic	Partitions	Partition Replicas
__docker-connect-configs	1	3
__docker-connect-offsets	25	3
__docker-connect-status	5	3
connect-file-pulse-status	1	1
default_ksql_processing_log	1	1
file_ACC	1	3
file_BR	1	3
file_ECG	1	3
file_ORIENT	1	3
file_PRESS	1	3
file_R2R	1	1
file_R2R_copy	1	1
test	1	1

```
ksql> show streams;
```

Stream Name	Kafka Topic	Key Format	Value Format	Windowed
ACC_STREAM	file_ACC	KAFKA	AVRO	false
BR_STREAM	file_BR	KAFKA	AVRO	false
ECG_STREAM	file_ECG	KAFKA	AVRO	false
ORIENT_STREAM	file_ORIENT	KAFKA	AVRO	false
PRESS_STREAM	file_PRESS	KAFKA	AVRO	false
R2R_STREAM	file_R2R_copy	KAFKA	AVRO	false

```
ksql> SET 'auto.offset.reset' = 'earliest';
Successfully changed local property 'auto.offset.reset' from 'earliest' to 'earliest'.
ksql> SELECT * FROM ECG_STREAM EMIT CHANGES LIMIT 10;
```

TIMESTAMP	INX	SAMPLENUM	FREQUENCY	HEARTRATE	SENSORSTATUS	RRLenght	SAMPLES
1579604534	20	715	128	108	255	554	67
1579604534	24	715	128	108	255	554	134
1579604534	28	715	128	108	255	554	214
1579604534	32	715	128	108	255	554	315
1579604534	36	715	128	108	255	554	449
1579604534	40	715	128	108	255	554	562
1579604534	44	715	128	108	255	554	648
1579604534	48	715	128	108	255	554	716
1579604534	52	715	128	108	255	554	711
1579604534	56	715	128	108	255	554	608

```
Limit Reached
Query terminated
ksql>
```

Before executing any new query, run the command:

SET 'auto.offset.reset' = 'earliest';

This will ensure that we are at the correct offset when reading the messages.

1) Insert into a new stream the average rrlenght for each value heart rate received and how many times that heart rate appears.

```
CREATE STREAM heart_rate_information AS
SELECT HEARTRATE, COUNT(*) AS how_common, AVG(RRLENGHT) as avg_rrlenght
FROM ECG_STREAM
GROUP_BY HEARTRATE
EMIT CHANGES;
```

2) Making the hypothesis that we can infer if the subject is resting, insert into a new stream all the heart rates that are higher than 90 [bpm].

Insert then into a table based on this stream entries where the heart rate is higher than 120, to highlight potentially critical conditions.

```
CREATE STREAM high_heart_rate_when_resting AS
SELECT TIMESTAMP as detected_time, HEARTRATE, [USER_CODE]
FROM ECG_STREAM
WHERE HEARTRATE >= 90
EMIT CHANGES;
```

```
CREATE TABLE critical_heart_rate_resting_condition AS
SELECT DETECTED_TIME, HEARTRATE, [USER_CODE], COUNT(*)
FROM HIGH_RATE_WHEN_RESTING
WINDOW TUMBLING (size 15 second)
GROUP BY DETECTED_TIME, HEARTRATE, [USER_CODE]
WHERE HEARTRATE >= 120
EMIT CHANGES;
```

****USER CODE is just the hypothetical code of each subject, that would allow us to track the users' activities in all the data streams.**

3) Making the hypothesis that we can infer if the subject is exercising, insert into a new stream all the heart rates that are higher than 130 [bpm].

Insert then into a table based on this stream entries where the heart rate is higher than 160, to highlight potentially critical conditions.

```
CREATE STREAM high_heart_rate_when_exercising AS
SELECT TIMESTAMP as detected_time, HEARTRATE, [USER_CODE]
FROM ECG_STREAM
WHERE HEARTRATE >= 130
EMIT CHANGES;
```

```
CREATE TABLE critical_heart_rate_exercising_condition AS
SELECT DETECTED_TIME, HEARTRATE, [USER_CODE], COUNT(*)
FROM HIGH_RATE_WHEN_EXERCISING
WINDOW TUMBLING (size 15 second)
GROUP BY DETECTED_TIME, HEARTRATE, [USER_CODE]
WHERE HEARTRATE >= 160
EMIT CHANGES;
```

****USER CODE is just the hypothetical code of each subject, that would allow us to track the users' activities in all the data streams.**

4) For example, is we had a user code to identify each subject, we could make an hypothesis on the value detected by the triaxial accelerometer to know if the subject was exercising or not.


```
CREATE STREAM high_heart_rate_when_exercising_second AS
SELECT e.TIMESTAMP AS detected_time, e.HEARTRATE, e.USER_CODE, a.SAMPLES_X, a.SAMPLE_Y,
a_SAMPLES_Z
FROM ECG_STREAM e INNER JOIN ACC_STREAM a
WITHIN 2 SECONDS
ON e.USER_CODE = a.USER_CODE
WHERE ABS(a.SAMPLES_X) > 1000 OR ABS(a.SAMPLES_Y) > 1000 OR ABS(a.SAMPLES_Z) > 1000 AND
a.HEARTRATE >= 130
EMIT CHANGES;
```

- By making a join between the two streams, we say that the the subject had too high a heart rate during a physically demanding activity, by putting a condition on the heart rate value and triaxial values and also the time difference between those events.

5) We could take a similar approach with the breath rate.

```
CREATE STREAM high_breathing_rate_when_exercising AS
SELECT b.TIMESTAMP AS detected_time, b.HEARTRATE, b.USER_CODE, a.SAMPLES_X, a.SAMPLE_Y,
a_SAMPLES_Z
FROM BR_STREAM b INNER JOIN ACC_STREAM a
WITHIN 2 SECONDS
ON b.USER_CODE = a.USER_CODE
WHERE ABS(a.SAMPLES_X) > 1000 OR ABS(a.SAMPLES_Y) > 1000 OR ABS(a.SAMPLES_Z) > 1000 AND
b.BREATHRATE > 20
EMIT CHANGES;
```

The queries that we execute can be of either type **PUSH** or **PULL**.

With PUSH queries, messages keep being queried from the Streams/Topic of origin, until manually terminated or until a set LIMIT condition is exceeded.

PUSH queries are characterized by the syntax:

- *EMIT CHANGES;*

To end their execution, run this command:

- *terminate query_id;*

Where the query_id is the identifier of the specific query to stop, and we can get it by running the command:

- *show queries;*

Which will show us all the active queries, their id and syntax.