Insight Platform Documentation

> You can fork this repository @ https://github.com/DahlmannIT/UIP_WS19-20_Infrastructure

This documentation describes the infrastructure of a scalable data streaming and processing platform, which gives the possibility to persist and analyze incoming data from various types and sources.

Table of Contents

- 1 Getting started
  - 1.1 Introduction
  - 1.2 Use Cases
  - 1.3 Prerequisites
  - 1.4 Quick start
  - 1.5 Installation Guide
- 2 Frameworks
  - 2.1 Architecture
  - 2.2 Documentation of Frameworks
- 3 How to use
  - 3.1 Reading data
  - 3.2 Make data persistable
  - 3.3 Deploying Flink-Job
  - 3.4 Accessing PostgreSQL
  - 3.5 Exploring data with Zeppelin
  - 3.6 Monitoring container status with Grafana
  - 3.7 Raw monitoring with Prometheus
- 4 Developers
  - 4.1 docker-compose.yml
  - 4.2 Environment Variables
    - 4.2.1 ZooKeeper
    - 4.2.2 Kafka
    - 4.2.3 Kafka-Connect
    - 4.2.4 PostgreSQL
    - 4.2.5 Jobmanager
    - 4.2.6 Taskmanager
    - 4.2.7 Prometheus
    - 4.2.8 Grafana
  - 4.3 Connectors
    - 4.3.1 Source-Connectors
    - 4.3.2 Sink-Connectors
- 5 KeyHashingJob
- 6 Troubleshooting

1 Getting started

1.1 Introduction

This software builds a powerful insight-platform to persist, analyze and work with big-data streams.

The infrastructure consists of the distributed streaming-platform `Apache Kafka` to build a real time data pipeline for multiple data sources to be used. It should be noted that, as a message streaming service, Apache Kafka needs to run `Apache ZooKeeper` for robust synchronization of naming and configurating data as well as keeping track of the Kafka nodes, topics, partitions etc. The data will automatically be cleaned and persisted in a `PostgreSQL` ORDBMS. To stream-process the data, connect `Apache Flink` to the right Kafka-Consumer and deploy your job. Analyzed data and results can also be persisted in PostgreSQL. For a more visual view of your database, please refer to `Apache Zeppelin`.

1.2 Use Cases

- As a data analyst, I want to have as much data as possible to have a good basis for analysis and prognosis.
- As a developer, I want to be able to scale the environment to handle big amounts of incoming data.
- As a developer, I want to stream and persist live data from various data sources into a database to create a sufficient foundation for my colleagues to work on.

1.3 Prerequisites

- JDK8 has to be installed
- Docker has to be installed
- Docker-Compose has to be installed

1.4 Quick start

Perform the following two steps (required only once)

- move a file called `transaction_data.csv` to the `data` directory to be able to deploy a connector (next step)

In your terminal, navigate to `docker-compose.yml` file and start a cluster

- `sudo docker-compose up`
- execute the `deploy-connector.sh` file with
    - `bash deploy-connector.sh` OR `./deploy-connector.sh`

For destroying a cluster, type

- `sudo docker-compose down`

To get this platform started you have to move a file which has to be named `transaction_data.csv` into the `data` directory. This is necessary to deploy a connector. In order to deploy a connector you have to execute the `deploy-connector.sh` file with the terminal command `bash deploy connector.sh`. The command `./deploy-connector.sh` works as well. After you performed these two steps you can start a whole docker cluster in the terminal by navigating to the `docker-compose.yml` file. With the simple command `sudo docker-compose up` all containers will start. If the required docker images aren´t available on your machine, they will automatically be downloaded. With the command `sudo docker-compose down` the cluster will be destroyed.

1.5 Installation Guide

create a user called "kafka_connect" in postgres by accessing postgres with `sudo docker exec -it postgres bash` then `psql postgres postgres`

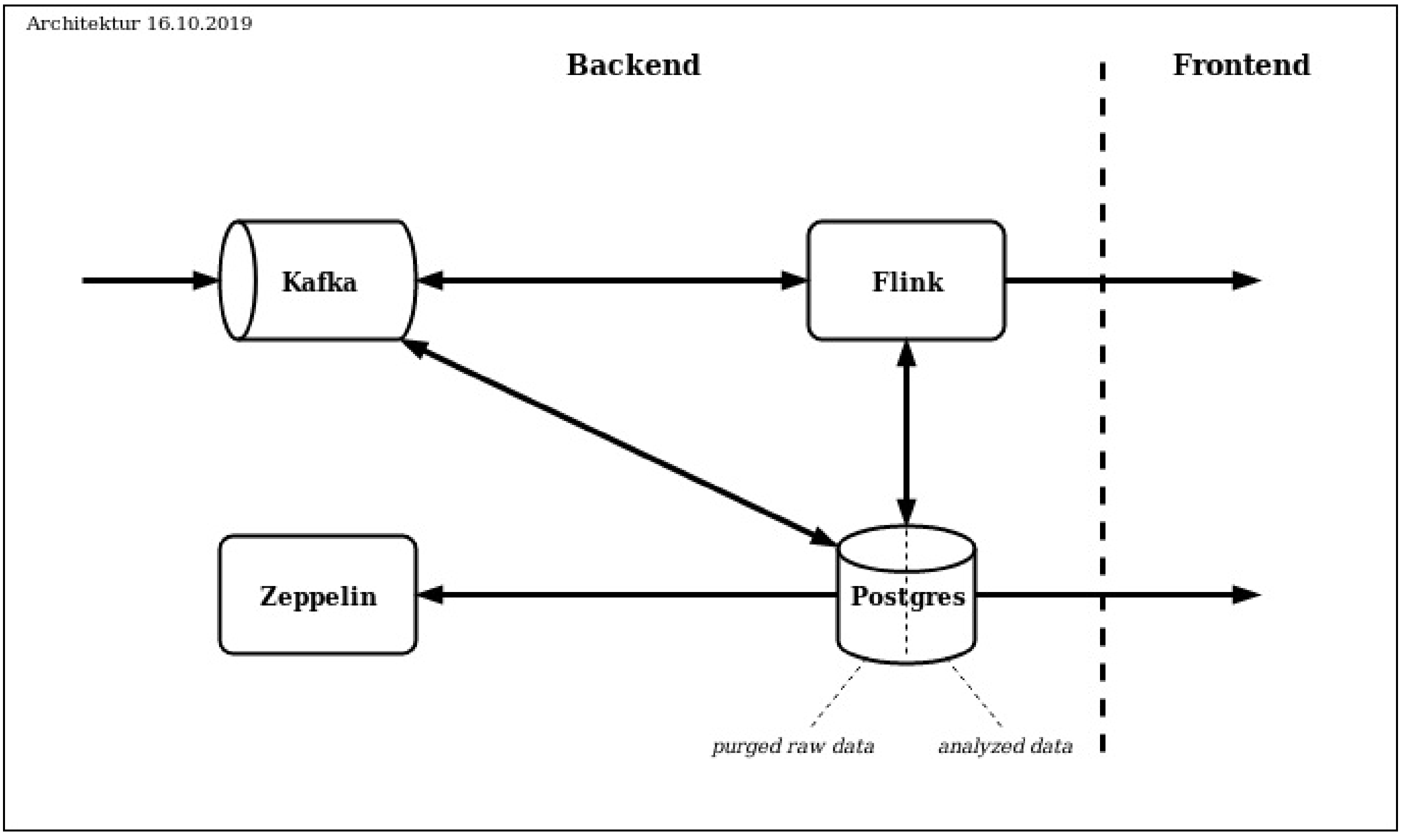
afterwards you create the required user by entering the following commands

```
CREATE USER kafka_connect WITH PASSWORD 'kafka_connect';
CREATE DATABASE kafka_connect;
GRANT ALL PRIVILEGES ON DATABASE kafka_connect TO kafka_connect;
```

and `./deploy-connectors.sh` again!

2 Frameworks

2.1 Architecture



2.2 Documentation of Frameworks

[Apache Kafka]

[PostgreSQL]

[Apache Flink]

[Apache Zeppelin]

3 How to use

3.1 Reading data

To let Kafka read your data, just move it to the `data` folder. Kafka will automatically process your data according to your schemes (in `deploy-connector.sh`).

If the data is parsed correctly, it will be moved to the `data/finish` folder, or else you can find your data in `data/error` directory.

To inspect and adjust the schemes, take a look at the `deploy-connector.sh` file.

For adding a new data source, take the following steps:

- create a new connector in deploy-connector.sh (.csv) scheme
- copy another connector and change "name" to `<anything>_source*` e.g. `transaction_data_source` "input.file.pattern" to a regex matching your files, e.g. `".*?transaction_data.*?\\.csv"` for reading all .csv files consisting "transaction_data" "topic" to `<anything>` e.g. `transaction_data`

For adding a new data source, you have to take the following steps. Initially you have to create a new connector (.csv) scheme. Afterwards a new `InputFilePattern` has to be created. Finally you need to create a new topic inside the connector.

3.2 Make data persistable

- to persist data, we need a `PRIMARY KEY` for your data, therefore we let Flink do its wonders
- lets say, your original data source writes to `Input`-topic in Kafka
- deploy `KeyHashingJob.jar` into Flink, whereas the InputTopic = `Input` and OutputTopic = `Input_persist`
- Flink will generate Primary Keys for your data, so it can be persisted

To persist the data in the database you need a `PRIMARY KEY` for the data. Our stream processing framework Flink is used for this purpose. You need to deploy the existing Flink job `KeyHashingJob.jar` to generate primary keys for all your data in order to persist them. If your original data source writes to the `Input`-topic in Kafka, you have to deploy the `KeyHashingJob.jar` (see next step) in which case the InputTopic should be named `Input` and OutputTopic should be named `Input_persist`. You can also configure those variables in the Flinkjob.

To make your own Flinkjob, check Flink Hashing Job for an example.

3.3 Deploying Flink-Job

To deploy the Flink-Job you have to go to `localhost:8081` where the Flink GUI is accessible. Click the `Submit new Job`-button to upload and start the `KeyHashingJob.jar`. See the Flink examples below for a more detailed view of Flink-Jobs. If you can't access the Flink GUI, you can also deploy a Job in the Flink-Bash (called Jobmanager): `./bin/flink run <directory/name.jar>`.

3.4 Accessing PostgreSQL

Access the postgres-container bash

- `docker exec -it postgres bash

Connect to `psql` with username: kafka_connect - password: kafka_connect

- `psql kafka_connect kafka_connect`

Connect to your preferred database with `\c <database>`

- `\c`

Print all tables

- `\dt` or `\d`

enter any `SQL` -commands, ending with a `;`

- `select * from transaction_data_persist;`

In the terminal you have direct access to the PostgreSQL database within the postgres bash. To get access you have to use the command `docker exec -it postgres bash`. After that you need to connect to `psql` with the username: postgres and the password: postgres. The required command is `psql kafka_connect kafka_connect` (`psql <username> <password>`). To access your preferred database use the command `\c <database>`. In our example just `\c` is used. All tables can be printed with `\dt`. After these commands you can enter any `SQL` -commands e.g. `SELECT * FROM transaction_data_persist;`. The `;` at the end of your SQL-command should by no means be forgotten!

3.5 Explore data with Zeppelin

- Fast Data Exploration on database via multiple interpreters
- Access GUI @ `localhost:8080`
- create new Notebook
- choose Interpreter and start your coding with e.g.

  ```
  %jdbc
  select * from transaction_data_persist;
  ```

You can also explore the database with the included notebook framework called Zeppelin. Go to `localhost:8080` to access the GUI of Apache Zeppelin. To get started a new `notebook` must be created (the name of the notebook doesn´t matter, so feel free to be creative). In the notebook created you have to choose an interpreter. After you have chosen one you can start your coding e.g.

```
%jdbc
SELECT * FROM transaction_data_persist;
```

Again, don´t forget the `;`.

3.6 Monitoring container status with Grafana

- monitoring container-status, based on Prometheus
- go to `localhost:3000`

  ```
  user: admin
  password: password
  ```

- check out Home -> Kafka-Overview

3.7 Raw monitoring with Prometheus

Access Prometheus by visiting

* `localhost:9090`

4 Developers

4.1 docker-compose.yml

```
version: '3'
services:

  zookeeper:
    image: confluentinc/cp-zookeeper:latest
    container_name: zookeeper
    hostname: zookeeper
    environment:
      ZOOKEEPER_CLIENT_PORT: 2181
      ZOOKEEPER_TICK_TIME: 2000
    ports:
     - 2181

  kafka:
    build: ./docker-images/kafka
    container_name: kafka
    hostname: kafka
    environment:
      KAFKA_BROKER_ID: 1
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
      KAFKA_ADVERTISED_LISTENERS:
PLAINTEXT://kafka:29092,PLAINTEXT_HOST://localhost:9092
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP:
PLAINTEXT:PLAINTEXT,PLAINTEXT_HOST:PLAINTEXT
      KAFKA_INTER_BROKER_LISTENER_NAME: PLAINTEXT
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
      KAFKA_OPTS: -
javaagent:/usr/app/jmx_prometheus_javaagent.jar=7071:/usr/app/prom-jmx-agent-
config.yml
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
    ports:
     - 9092:9092
    depends_on:
     - zookeeper

  kafka-connect:
    build: ./docker-images/kafka-connect
    container_name: kafka-connect
    hostname: kafka-connect
    environment:
      CONNECT_BOOTSTRAP_SERVERS: kafka:29092
      CONNECT_REST_ADVERTISED_HOST_NAME: connect
      CONNECT_GROUP_ID: kafka-connect-group
      CONNECT_REST_PORT: 8083
      CONNECT_CONFIG_STORAGE_TOPIC: connect-configs
      CONNECT_OFFSET_FLUSH_INTERVAL_MS: 10000
      CONNECT_OFFSET_STORAGE_TOPIC: connect-offsets
      CONNECT_STATUS_STORAGE_TOPIC: connect-status
      CONNECT_CONFIG_STORAGE_REPLICATION_FACTOR: 1
      CONNECT_OFFSET_STORAGE_REPLICATION_FACTOR: 1
      CONNECT_STATUS_STORAGE_REPLICATION_FACTOR: 1
      CONNECT_KEY_CONVERTER: org.apache.kafka.connect.json.JsonConverter
      CONNECT_VALUE_CONVERTER: org.apache.kafka.connect.json.JsonConverter
      CONNECT_INTERNAL_KEY_CONVERTER: "org.apache.kafka.connect.json.JsonConverter"
```

```yaml
      CONNECT_INTERNAL_VALUE_CONVERTER: "org.apache.kafka.connect.json.JsonConverter"
      CONNECT_ZOOKEEPER_CONNECT: 'zookeeper:2181'
      CONNECT_PLUGIN_PATH: "/usr/share/java,/usr/share/confluent-hub-components"
    depends_on:
      - kafka
      - postgres
      - zookeeper
    volumes:
      - ./data:/home/data
    ports:
      - 8083:8083

  postgres:
    image: postgres
    container_name: postgres
    ports:
      - 5432:5432
    environment:
      POSTGRES_USER: postgres
      POSTRES_PASSWORD: postgres
    volumes:
      - ./.docker-volumes/postgres-data:/var/lib/postgresql/data
      - ./.docker-volumes/postgres-init:/docker-entrypoint-initdb.d/

  zeppelin:
    image: xemuliam/zeppelin
    container_name: zeppelin
    ports:
      - 8080:8080
    volumes:
      - /opt/zeppelin/logs
      - /opt/zeppelin/notebook
      - ./.docker-volumes/zeppelin-conf:/opt/zeppelin/conf

  jobmanager:
    image: ${FLINK_DOCKER_IMAGE_NAME:-flink}
    container_name: jobmanager
    expose:
     - 6123
    ports:
     - 8081:8081
    command: jobmanager
    environment:
     - JOB_MANAGER_RPC_ADDRESS=jobmanager

  taskmanager:
    image: ${FLINK_DOCKER_IMAGE_NAME:-flink}
    container_name: taskmanager
    expose:
     - 6121
     - 6122
    depends_on:
     - jobmanager
    command: taskmanager
    links:
     - jobmanager:jobmanager
    environment:
     - JOB_MANAGER_RPC_ADDRESS=jobmanager

  prometheus:
    image: prom/prometheus:latest
```

```
    container_name: prometheus
    user: root
    ports:
      - 9090:9090/tcp
    volumes:
      - ./.docker-volumes/prometheus:/etc/prometheus
    depends_on:
      - kafka

  grafana:
    image: grafana/grafana:6.1.1
    container_name: grafana
    ports:
      - "3000:3000"
    environment:
      - GF_SECURITY_ADMIN_PASSWORD=password
    volumes:
      - ./.docker-volumes/grafana_data:/var/lib/grafana
    depends_on:
      - "prometheus"
```

4.2 Environment Variables

4.2.1 ZooKeeper

The ZooKeeper image uses variables prefixed with `ZOOKEEPER_` with the variables expressed exactly as they would appear in the `zookeeper.properties` file. As an example: for `clientPort` and `tickTime`, you can use

```
    ZOOKEEPER_CLIENT_PORT: 2181
    ZOOKEEPER_TICK_TIME: 2000
```

Please refer to Confluent Documentation for more details.

4.2.2 Kafka

The Kafka image uses variables prefixed with `KAFKA_`. See Confluent Documentation.

4.2.3 Kafka-Connect

Kafka Connect is used for connecting various datatypes and schemas with Kafka. There are a lot of predefined source and sink connectors available. Please refer to Confluent Kafka Connect Documentation for more details. For a list of used connectors, refer to Chapter 4.3 Connectors.

4.2.4 PostgreSQL

The PostgreSQL database is used to persist analyzed and cleaned raw data and results. Please refer to PostgreSQL Documentation for more details.

4.2.5 Zeppelin

For configuring Zeppelin, make sure to expose port 8080 and overwrite the necessary volumes. See Apache Zeppelin Documentation for a detailed view.

4.2.6 Jobmanager

This is the master-node of Apache Flink. You can deploy Flink-Jobs to this container. See Apache Flink Documentation for more details.

4.2.7 Taskmanager

The Taskmanager is an image of Apache Flink, which use is to process Kafkas incoming data. Please refer to Apache Flink Documentation for more details.

4.2.8 Prometheus

To get a list of possible Prometheus Environment Variables, see Prometheus Documentation.

4.2.9 Grafana

As for Grafana, see Grafana Documentation.

4.3 Connectors

4.3.1 Source-Connectors

As you can see in `deploy-connector.sh`, we use the following **source-connectors** for:

- transaction_data

  ```
  curl -X POST http://localhost:8083/connectors -H "Content-Type: application/json"
  -d '{
        "name": "transaction_data_source",
        "config": {
                "tasks.max": "1",
                "connector.class":
  "com.github.jcustenborder.kafka.connect.spooldir.SpoolDirCsvSourceConnector",
                "input.path": "/home/data",
                "error.path": "/home/data/error",
                "finished.path": "/home/data/finished",
                "halt.on.error": "false",
                "errors.tolerance": "all",
                "errors.deadletterqueue.topic.name": "csv_deadletterqueue",
                "errors.deadletterqueue.topic.replication.factor": "1",
                "empty.poll.wait.ms": "3000",
                "csv.first.row.as.header": "true",
                "schema.generation.enabled": "true",
                "csv.null.field.indicator": "EMPTY_SEPARATORS",
                "csv.separator.char": "59",
                "input.file.pattern": ".*?transaction_data.*?\\.csv",
                "topic": "transaction_data"
                }
        }'
  ```

  which reads CSV-files containing `transaction_data` in its name and are `;` separated. Data will be produced into the `transaction_data` topic of Kafka.

- marketo

  ```
  "name": "marketo_source",
  ```

```
      "config": {
              "tasks.max": "1",
              "connector.class":
"com.github.jcustenborder.kafka.connect.spooldir.SpoolDirCsvSourceConnector",
              "input.path": "/home/data",
              "error.path": "/home/data/error",
              "finished.path": "/home/data/finished",
              "halt.on.error": "false",
              "errors.tolerance": "all",
              "errors.deadletterqueue.topic.name": "csv_deadletterqueue",
              "errors.deadletterqueue.topic.replication.factor": "1",
              "empty.poll.wait.ms": "3000",
              "csv.first.row.as.header": "true",
              "schema.generation.enabled": "true",
              "csv.null.field.indicator": "EMPTY_SEPARATORS",
              "csv.separator.char": "59",
              "input.file.pattern": ".*?marketo.*?\\.csv",
              "topic": "marketo"
              }
```

which is the same as `transaction_data` but reading files containing `marketo` in its name. Data will be produced into the `marketo` topic of Kafka.

For a detailed view of this connector-class' configuration, check SpoolDirConnector Documentation.

4.3.2 Sink-Connectors

For **sink-connectors** we use:

```
  curl -X POST http://localhost:8083/connectors -H "Content-Type: application/json" -
d '{
      "name": "postgres-sink",
      "config": {
              "tasks.max": "1",
              "connector.class": "io.confluent.connect.jdbc.JdbcSinkConnector",
              "connection.url": "jdbc:postgresql://postgres:5432/kafka_connect",
              "connection.user": "kafka_connect",
              "connection.password": "kafka_connect",
              "auto.create": "true",
              "auto.evolve": "true",
              "errors.tolerance": "all",
              "errors.deadletterqueue.topic.name": "jdbc_deadletterqueue",
              "errors.deadletterqueue.topic.replication.factor": "1",
              "pk.mode": "record_value",
              "pk.fields": "postgres_pk",
              "insert.mode": "upsert",
              "topics.regex": ".*?persist"
              }
      }'
```

This connector reads every topic ending with `persist` and writes to the PostgreSQL database using `kafka_connect` as the user. In our case, the Flink-Job `KeyHashingJob` reads data from the source connectors topics and creates a primary key (PK) called `postgres_pk` by hashing the payload of each event. Afterwards the new data will be written to its respective Kafka-topic, adding a "persist" at its end, so the sink-connector can write it to PostgreSQL.

JDBC Sink Connector Documentation can be found here.

5 KeyHashingJob

To get the Flink-Job started we need to give it some various meta data. In our example we use

```
 String jobName = parameterTool.get("job-name", "KeyHashingJob");
        String inputTopic = parameterTool.get("input-topic", "transaction_data");
        String outputTopic = parameterTool.get("output-topic",
"transaction_data_persist");
        String consumerGroup = parameterTool.get("group-id", "KeyHashingGroup");
        String kafkaAddress = parameterTool.get("kafka-address", "kafka:29092");
```

After getting the execution environment and adding a new kafka consumer to it we need to process the datastream according to our calculation. Like so

```
DataStream<String> outputStream = dataStream
                        // parse the json string
                        .map((MapFunction<String, ObjectNode>) value ->
(ObjectNode)objectMapper.readTree(value))
                        // do the calculation and add the hash to the payload as well
as its definition to the schema
                        .map((MapFunction<ObjectNode, ObjectNode>) value -> {
                                ((ObjectNode)
value.get("payload")).put("postgres_pk",
DigestUtils.sha256Hex(objectMapper.writeValueAsBytes(value.get("payload"))));
                                ((ArrayNode)
value.get("schema").get("fields")).add(objectMapper.createObjectNode()

.put("type","string").put("optional",false).put("field","postgres_pk"));
                                return value;
                        })
                        // convert to string again
                        .map((MapFunction<ObjectNode, String>) value ->
objectMapper.writeValueAsString(value));
```

The KeyHashingJob hashes the payload of the current dataset from the input topic and adds the resulting value to the payload. Respectively, the key postgres_pk will be added to the schema.

After that the resulting JSON will be converted to string and written to the output topic `transaction_data_persist` via a new Kafka Producer.

```
//create a new kafka producer
        Properties producerProps = new Properties();
        producerProps.setProperty("bootstrap.servers", kafkaAddress);
        FlinkKafkaProducer<String> flinkKafkaProducer = new FlinkKafkaProducer<>
(outputTopic,
                        new KeyedSerializationSchemaWrapper<>(new
SimpleStringSchema()),
                        producerProps, FlinkKafkaProducer.Semantic.AT_LEAST_ONCE);

        //add the producer to the dataStream as a sink
        outputStream.addSink(flinkKafkaProducer);

        environment.execute(jobName);
```

6 Troubleshooting

- If user `kafka_connect` doesn't exist in PostgreSQL, make sure to check `/.docker-volumes/postgres-init/init.sql` or create an user yourself.
- If there are no tables being created in PostgreSQL, get your Flink-Job running, re-deploy all connectors with a valid schema in the `data` folder and wait for about 5 minutes. (You can delete them by using `curl -X DELETE http://localhost:8083/connectors/<name_connector>`).
- If there is no worker running on your Kafka-Connect node, copy a valid schema in the `data` folder and re-deploy your connectors. (You can delete them by using `curl -X DELETE http://localhost:8083/connectors/<name_connector>`).