Project Athena

Done by Jakob Neuhauser, Ivonne Simic and Paul Pavlis

1. Table of contents

2.Executive Summary

The Project Athena can be found at following link in the github repository:
https://github.com/h1v3r/Athena

Background

This project is the final version of a university lecture, from the university "FH-Technikum Wien" called "Big Data Infrastructure". The goal of this project is to gain a deeper knowledge of some of the most commonly used Big Data technologies.

Problem and objective

In many new greenhouses with lots of different plants and floras, there are a lot of sensor needed, to optimise the growth of the plants and efficiently plan out the overall structure. This sensor data needs to be measured, processed, and analysed in a near Real-Time environment. In addition to that, it is mandatory to save all measurements for later diagnostics in a persistent way. Furthermore, if critical sensor data occurs, they need to be addressed and directly send to the hardware and manging part of the greenhouse to correct the problems on the fly.

Our solution

To fulfil these Use-Cases we built a construct of different programs that communicate with one another and directly address the problems. To achieve complete modularity, we assembled our whole application in different containers with a single start-up file using the docker technology.

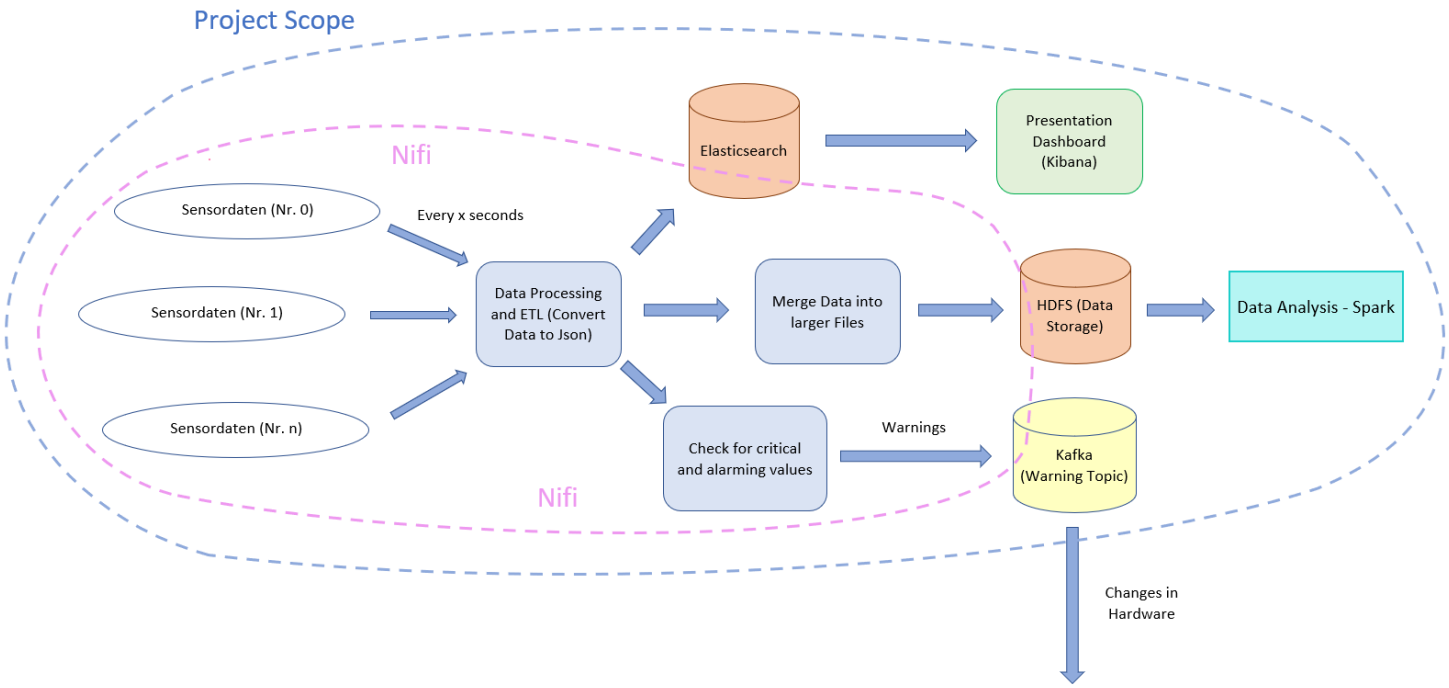
In the first step, we receive the data from the sensors directly via an API HTTP-Request. The data is transferred in the JSON file format and is then checked for critical data. If such data occurs, it is sent into a message queue where it can be read and assessed by hardware.
In a second step, the sensor data is temporarily saved in a search and analytics engine to analyse the data in a near Real-Time graphical dashboard style.

In the last step, the data is saved in a data storage and format that is optimized for long term storage and compression. This data can later be analysed with an analytics engine to generate important reports of the overall efficiency and quality.

The whole solution is designed in a way, that allows easy horizontal (and vertical) scalability for all used technologies.

3.System Architecture

Diagram



Procedure & Technologies

Summary of the overall procedure

To start the whole process, we need to receive the sensors data via messages, so that we can transform and use them for the following steps. This is being achieved with Apache NiFi. The Program listens on a specific port and each sensor data entry that is sent there, will be converted into a usable file (flowfile in NiFi). These flowfiles then undergo a check and are transformed to the right json format for later use.

We use these sensor data sets for three different workflows:

- (Near) Real-Time Analysis and Dashboard
  - The data is transported (with Apache NiFi) to Elasticsearch, where it will be stored at an index (index can basically be associated with a table in SQL). From there the data will be accessed with Kibana and mapped onto different Graphs. These Graphs show the flow of the current (and near past) sensor data and the different values for e.g. the temperature or soil moisture.
- Long-term storage and queries on the dataset
  - First, the data from the different NiFi flowfiles are merged into fewer larger files, so that the following technology is more efficient in handling large amounts of data. The data is then stored in HDFS (Hadoop distributed file system), which basically just stores the data for later use. Then a Query

Engine (Apache Spark) is used to make queries on the whole dataset to get e.g. the average temperature of the year 2019. The data from the analysis can be used to correct different aspects of the greenhouse.

- Check for critical values and changing the hardware accordingly
  - In this pipeline, the values of the sensor data are checked (in NiFi) against predefined maximal and minimal values. If the value is above or below the threshold, the file will be sent to a message broker (Apache Kafka). From there the critical values can be extracted and sent to the hardware to change these values on e.g. the flowerpots. (The Hardware aspect is out of scope for this project. The data will only be provided, not used)

Apache NiFi

Description

Before I will explain what NiFi actually is, I first want to show the problem that NiFi is solving. The problem is communication. In informatics and especially in Data Science you have numerous tools for completing different tasks. However, the problem is that these tools need to interact with one another. This can be really hard since different software has different mechanisms to get data or to export data. This is where NiFi shines and makes life easier. It is a program with graphical interface where you can let different tools speak with one another. It is something like the glue between everything.

NiFi consists of two main parts. First there are processors. For instance, if you have tool A and want to put data into it, you are using a NiFi processor. But you are also using one if you want to get the data from tool A back into NiFi. In the end there are numerous other ways you can use processors, but that would be too much for now.

The second part are queues. These are used to put data from one NiFi processor to another. You could for instance get data from tool A with one NiFi processor. Send that date with a queue to another processor which sends the data to tool B.

NiFi in the project

In a nutshell we are using NiFi to get the date from our API, then check the data and send it to Kafka. We are also using it to send all the data to Elasticsearch and HDFS. In the following I will explain every processor, that we are using. Of course, I can just scratch the surface, so if you want more information, just visit the corresponding documentation.

ListenHttp-Processor

This processor is used for getting the data from the API. The processor is listening for data on a specific port. Our API is sending the data to the same port. The processor is able to get this data and puts it into a queue.

EvaluateJsonPath-Processor

We get all our data in JSON-Format. The "EvaluateJsonPath" processor is mapping the variables from JSON to variables understandable for NiFi. The processor also checks if all the variables we want are in the JSON-File.

RouteOnAttribute-Processor

Because we want to check on critical data direct at the start, we want to do that inside NiFi. With the "RouteOnAttribute" processor we are able to check if the data is inside self-defined boundaries. If the data is outside that boundaries the data will be sent into a queue for further operations (e.g. Kafka).

PublishKafka_2_0-Processor

This processor is used to write Data into a Kafka topic. What Kafka is exactly is explained in another part of the documentation.

PutElasticSearchHttp-Processor

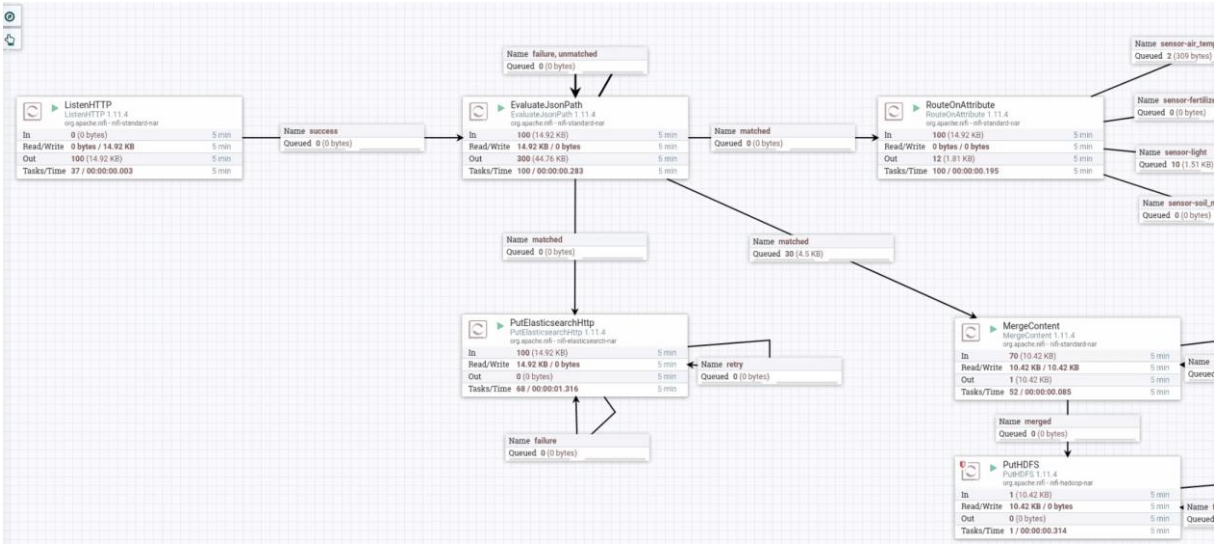
Here it is also quite simple. The processor simply gets data from a queue and sends that to Elasticsearch. Again, Elasticsearch is explained in more detail at another part of the documentation.

MergeContent-Processor

This processor we need for putting data into HDFS. Of course, strictly speaking wo do not really need it. However, HDFS is built for large data junks it is not very performant to put one JSON-File per one into HDFS. So, we need to merge these files to larger files. This is exactly what this processor is doing. It gets the data from one queue, merges the Files and puts them back into another queue.

PutHDFS-Processor

Lastly, we are using the "PutHDFS" processor to put the merged JSON-Files into HDFS which is than saving the data in a persistent way. Once again HDFS is also explained somewhere else in the documentation.



Why we are using NiFi

First of all, we are using it because it is a very simple way to let different programs and tools speak with one another. This not only makes it easier to build the project but also prevents mistakes. Also, we think that it is far more performant to do it that way, instead of writing own software.

Elastic Stack (Elasticsearch & Kibana)

Description

The Elastic Stack consists of Elasticsearch, Logstash, Kibana & Beats. Out of those four technologies, we have used Elasticsearch and Kibana. The other two technologies will not be discussed here further.

Before learning for what the Elastic Stack is used in our project, we will look at the specific problem, that it will fill:
Traditionally, data that is being stored inside a database for longevity is not the best for (near) Real-Time analysis. This is because the amount of data that is stored there plus the technology does not really support analysis on the fly. What we need, is a technology, that will store the most recent data for some time and that lets us make (near) Real-Time analysis while the process is running.

Elasticsearch is a distributed (which means, that it supports multiple instances of nodes) search and analytics engine. It allows to store JSON documents for all types of data (textual, numerical, geospatial, structure and unstructured).

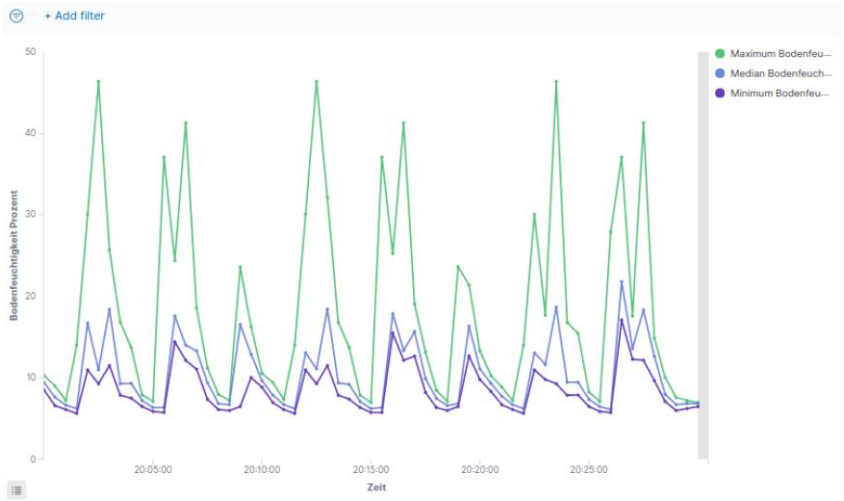
Kibana is an open user interface that allows you to visualize the data, that you stored in Elasticsearch. It allows the creation of bar, line and scatter plots, or pie charts and maps on top of large volumes of data. The different graphs can than be plotted onto a dashboard which can be exported to a different website, where users can interact with it and e.g. change various aspects like the names of sensors or a range for numbers.

Elastic Stack in the project

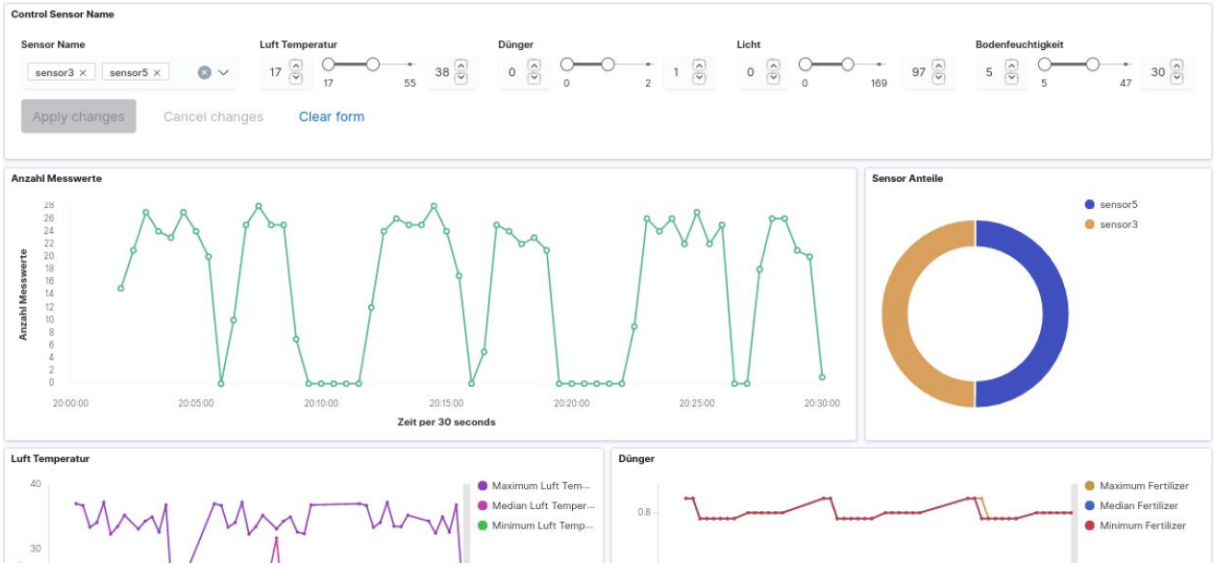
Our main use case for the elastic stack is the (near) Real-Time analysis, visualisation and the ability to filter and search for specific sensors or sensor values.

First, the data is saved from Apache Nifi into an Elasticsearch index. An index is comparable to a table in a relational database. Elasticsearch than saves the received sensor data and provides further functionality for the next step.

After that, the data will be accessed via Kibana. Here you can select the name of the indexes that you are looking to analyse. After choosing the previous index, we created various different visualisations like a histogram that shows the minimum, median and maximum of the measured air temperature.



This is done by splitting the data in various buckets and then plotting different metrics on the y axis. After creating these graphs, we made a dashboard, that sums up all the information's, that are needed for controlling the greenhouse and where it is possible to search for specific sensors or values.



Why we are using the Elastic Stack

We are using the Elastic Stack, because it provides us a (near) Real-Time analysis of the data and the added visualisation is also a particularly good feature to have (for management for example). We need some kind of aggregation and searching on the fly and the Elastic Stack provides these capabilities and many more.

At the start we thought about using Apache Flink as an aggregation service but then decided, that it is not a good fit for the ideas and Use Cases that we have. In addition to that, Apache Flink does not provide a strong bond to a virtualisation tool like elasticsearch has to kibana.

Hadoop HDFS & Apache Spark

Description

In our greenhouse, a lot of sensor data will accumulate over the course of weeks, months and years. These data sets will all go through the Real-Time analysis pipeline (Elastic Stack), but they will not be saved there for long. What we need is a technology, that lets us store massive amounts of data and the possibility to query that data to allow analysis over the course of years.

HDFS, which stands for “Hadoop Distributed File System”. It is a distributed (not only on one node/server) file system and a data storage system used by Hadoop applications. It is used as a long-term storage system. A single cluster consists of a single NameNode, which acts as a master server that manages the file system namespace and regulates access to files by clients. Normally, there are several DataNodes which store the data. In most cases the data will be replicated across many DataNodes to achieve resilience.

To access and query the data stored in HDFS, we chose Apache Spark. Apache Spark is an analytics engine for big data processing with SQL capabilities. Spark works nicely together with HDFS and it allows to query over the HDFS NameNodes and analyse and aggregate the long-term storage data.

HDFS & Spark in the project

First, we merge the various datafiles in Apache Nifi to larger files. These larger files will then be stored with Apache Nifi in HDFS. The reason for merging is that HDFS functions better with fewer larger files compared to many smaller files. That step improves the efficiency of our solution.

After the data is stored, Spark fetches the data from the Node. After that, we create a a table (or better a view on the data) and store the fetched data in it. On this "table" we can then query SQL (Structured Query Language) statements and fetch the wanted data. Here we do some things like group all the data from a specific day, month or year and aggregate the average of each sensor value. This can than be used for long-term analysis.

Why we are using HDFS & Spark

Our main use case for HDFS and Apache Spark is to store all sensor data for long-term use in a format, that allows data compression (we used json, another file format like parquet would be more efficient for columnar compression, which we learned after the final presentation).

Also, we do not need to fetch single entries of that file, because they are basically of no use for us. We need the engine to query for example all data sets from the last year. This is the reason we chose HDFS and not some other kind of data storage option or database like HBase or Cassandra (which in comparison to HDFS excel in fast random access but falls behind in large data analytics).

Apache Spark is than used for its simplicity, scalability and already functioning integration with HDFS. This allows us to query the dataset in a language, that is widely known and accepted.

Apache Kafka

Description

Again, we will start with explaining the Problem. Kafka, like NiFi, is used to let applications speak with one another. More or less Kafka provides queues (also called topics) where programs can write into and other programs can acquire the data from that queues.

Kafka in the project

We are using Kafka to save critical data. The data gets evaluated inside of NiFi and if the data is outside some boundaries, NiFi will send that data to a processor which send it to a Kafka queue. Then for instance programs responsible for regulating the hardware can get the data for quickly reacting on these critical values.

Why we are using Kafka

Many think, that Kafka and NiFi are the same. However, they may have similarities, but they are not used for the same things. On one hand NiFi has problems when one of its queues is full. If that happens NiFi will crash. In Kafka this will not happen. It stores the data and everyone who needs it can get it from there.

On the other hand, in NiFi you need a processor to get the data outside a queue. In Kafka everyone can get the data as long as he can connect to the corresponding Kafka queue. So, if we want more software that needs the data, this is simply possible by letting the software connect to Kafka.

Greenhouse Data

Data Picture

capture_datetim...	fertilizer_level	light	soil_moisture_pe...	air_temperature_...
3Jul18 – 1Aug18	0.78 – 1.11	0.1 – 169	5.62 – 46.4	17.6 – 54.9
2018-07-03 14:49:23	1.02	17.97	14.01	31.47
2018-07-03 15:04:23	1.02	18.56	13.88	31.73
2018-07-03 15:19:23	1.02	19.08	13.74	32.18
2018-07-03 15:34:23	1.02	19.4	13.61	31.96
2018-07-03 15:49:23	1.02	19.01	13.48	31.2
2018-07-03 16:04:23	1.02	18.1	13.48	30.98
2018-07-03 16:19:23	1.02	16.6	13.61	30.67
2018-07-03 16:34:23	1.02	15.0	13.48	30.19
2018-07-03 16:49:23	1.02	13.66	13.22	29.35
2018-07-03 17:04:23	1.02	11.99	13.09	28.69

Description

This part explains the data that is being used for the whole project. You can see a snippet of the data in the previous paragraph. Each of those rows represents one entry of a sensor from the greenhouse. It includes a timestamp and the fields “fertilizer_level”, “light”, “soil_moisture_percent” and “air_temperature_celsius”.

At the start we were looking for an open-API, which would send us sensor data as a stream of data, but we could not find one for free. To compensate for that, we chose to write our own program, that sends data as a stream to a specific internet address and port. The data that is being used for that purpose, is a real data set from kaggle: https://www.kaggle.com/ludwa6/soil-sensor

Field Description

The following description is for the specific dataset, that we are sending from our script to the address to simulate real sensor data. Another field has been added so that a distinction between different sensors is possible. Example data entry from a sensor in json format:

{"name": "sensor1", "time_measured": "2020-06-12T22:34:22Z", "fertilizer": 1.02, "light": 8.83, "soil_moisture_percent": 12.02, "air_temperature": 24.25}

- Field: name
  - This field includes the name of the sensor that recorded and send the data. In our example we are just using one kind of sensor and the sensor are named with "sensor" + [number].
  - Type: String
- Field: time_measured
  - This field describes the exact time, where the sensor recorded the data. Additionally, the data is then formatted to conform to ISO 8601, which is a universally accepted time format.
  - Type: Timestamp
- Field: fertilizer
  - This field describes the level, at which the fertilizer is (e.g. in a flowerpot).
  - Type: Decimal / Float
- Field: light
  - This field describes the amount of light that the plant (and sensor) is getting. This varies heavily if the light is turned of or if a cloud is blocking the sun.
  - Type: Decimal / Float
- Field: soil_moisture_percent
  - This field describes the percent of moisture (wetness) in the soil/ground. This variable degrades lightly over time and then makes a jump to high number again (when it is watered)
  - Type: Decimal / Float
- Field: air_temperature
  - This field describes the air temperature that the plant (and sensor) is getting. If the value is too high or too low, rapid adjustments must be made.
  - Type: Decimal / Float

4.Steps to reproduce (from readme.txt)

Preamble

The following paragraph was written with GitHub Syntax and is better viewed there. You can choose to view it here as well, but for the correct highlighting you should consider going to: https://github.com/h1v3r/Athena/

Github is a place where you can store and get projects from people that are using the technology git. Git is a version control system, which helps you and your team to manage the project so that there is not a conflict with different version of the same document.

Procedure

Athena

How to start up Athena

Prerequirements

You need to install [**docker**](https://www.docker.com/) as well as [**docker-compose**](https://docs.docker.com/compose/). On an [**Arch Linux**](https://i.redd.it/tcdhu46p4y451.jpg) System this can be done with:

`sudo pacman -S docker docker-compose`

Clone from GitHub

First you need to download the project from GitHub.

`git clone https://github.com/h1v3r/Athena.git`

After that you change into the git folder you just downloaded.

`cd Athena`

Start the containers

Before you can start everything up, you need to change the permissions for two scripts. You will need root privileges for that.

`sudo chmod 645 init_athena.sh rm_athena.sh`

Now you can start the "init_athena" script. You need to do this as root too.

`sudo ./init_athena.sh`

Now everything is starting up. If there is a problem with Elasticsearch and permissions, try:

`sudo chmod 777 ../Athena_Data`

in the git folder. It may solve the problem.

Set up NiFi

Next you need to implement the template for NiFi. NiFi will be available at:

`localhost:8080/nifi/`

It may take a bit till it is up and running. On the NiFi web interface you need to navigate you curse over the grid. Then right click on it and select “Upload template”. A window will pop up where you can select the template you want to upload. Click on the icon with the magnifying glass. If you have done that, a window will open up where you can select the template you want to upload. Simply navigate to your git folder and select “NiFi-Template.xml”. After this you must click “upload” and the template will be uploaded.

Now navigate to the icon with the name “Template” in the top bar and drag and drop it into the grid. Now you can choose what template you want to use. Select “NiFi-Template” and click “ADD”. The Template should appear on the grid.

First you need to click somewhere on the grid to dis-select everything. The template consists of many processors (large and rectangular). At every processor you can find a red square which indicates that the processor is turned off. To turn a processor on you right click on it and select start. You need to do this for every processor.

Start the API

After you have set up NiFi you can start the API with:

`python python_SendData_simultan.py`

To accept the default settings of the script click enter.

Pause the Program

The following steps "Pause the Program", "Remove Athena" and "Portainer" are optional and not needed for the workflow. It provides additional information’s to maximize your docker experience or stop and remove the whole application when you are done.

You can bring all containers down by executing

`sudo docker-compose down`

inside the git folder. Use

`sudo docker-compose up -d`

to bring them up again. However, you need to setup NiFi again.

Remove Athena

First you want to stop the API. Simply click into the shell where you have started the API and press "Ctrl-C".

Next you simply have to execute the "rm_athena.sh" script.

`sudo ./rm_athena.sh`

This will delete all data related to the project except the git repo and date in relation with docker.

Portainer

The "docker-compose" includes a Portainer container which can be accessed at:

`localhost:9000`

Analyse the data with Kibana

Kibana will be available at:

`localhost:5601`

After you have entered the Kibana web interface you need to click on “Dashboard” at the menu on the left. At “Dashboards” select “Greenhouse_Dashboard”.

If you are not able to see the dashboard and you get the message that you need to define an index first, then click on "Management" at the menu on the left. Under the header "Kibana" you will find the point "Saved Objects". After clicking that, there should be an option to import a kibana file which is located in your directory at "./zz-Archive/Kibana-save-1.ndjson

Now you should see a few rows added to the list including the index "testindex2" where our data is stored and "Greenhouse_Dashboard" where the Dashboard is located. Either click on

it or go the menu point "Dashboards" at the menu on the left and choose the correct Dashboard.

Now you are at the Dashboard. On the top you will find a console where you can group by “Sensor Name” or select ranges for the parameters. Below that you can find diagrams for the count of measurements per time interval and a pie chart where you can see the shares of each sensor. Under those diagrams you can find four more, each representing one parameter (Air Temperature, Fertilizer, Light and Soil Moisture (in percent)). Each graph displays the maximum, minimum and median per time interval for the corresponding reading.

If you get the message "No data available" at the dashboard screen, try changing the time window at the top right of the window to a different interval (e.g. today).

Analyse HDFS with Spark (example)

To open the spark shell you need to access the Spark container with:

`sudo docker exec -it Athena-spark-master-1 /bin/bash`

Then you can open the spark shell.

`./spark/bin/spark-shell`

On the spark shell you first need to create a data frame.

`val sensorDF = spark.read.json("hdfs://namenode:9000/GreenhouseArchiveTest1")`

With this data frame you can create a view:

`sensorDF.createOrReplaceTempView("sensors")`

You can select from this view with sql statements and print the result.

`val all_data = spark.sql("select * from sensors")`

`all_data.show()`

You can make an example display of all data in a certain time interval (in this example all data on "2020-06-14"),

`val max_for_time = spark.sql("select max(air_temperature), max(fertilizer), max(light), max(soil_moisture_percent) from sensors where time_measured like '2020-06-14%'")`

`max_for_time.show()`

or group by sensor.

`val avg_per_sensor = spark.sql("select name, avg(air_temperature), avg(fertilizer), avg(light), avg(soil_moisture_percent) from sensors group by name order by name")`

`avg_per_sensor.show()`

If you want to export the data you collected you can do that by saving the data at the "/spark-data" directory. This directory is mounted at "../Athena_Data/spark-data" at you host machine.

`val op= avg_per_sensor.rdd.map(_.toString().replace("[","").replace("]", "")).saveAsTextFile("/spark-data/test")`