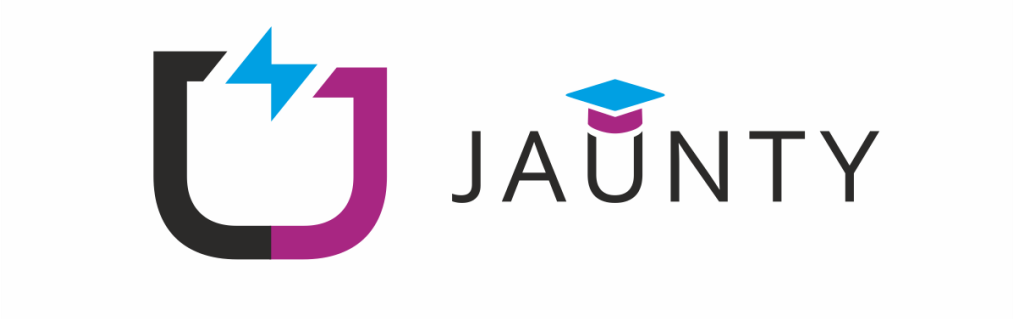
Energy Meter Logging, Storage, and Visualisation with InfluxDB and Grafana



Author: Public Power Corporation S.A

Co-funded by the
Erasmus+ Programme
of the European Union

Copyright

Consisting of

Coordinator:	Technical University of Sofia	Bulgaria
Partners:	University of Western Macedonia	Greece
	International Hellenic University	Greece
	Public Power Corporation S.A.	Greece
	University of Cyprus	Cyprus
	K3Y Ltd	Bulgaria
	Software Company EOOD	Bulgaria

Co-funded by the Erasmus+ Programme of the European Union

Table of Contents

1. Abbreviations

API	Application Programming Interface
RFC	Request For Comments
CLI	Command Line Interface
CSV	Comma Separated Values
HTTP	HyperText Transfer Protocol
GUI	Graphical User Interface
UTC	Universal Time Clock

2. Scope

The scope of this laboratory module is to introduce time series databases and specifically the InfluxDB, regarding the main concepts involved, the format of the data and its uses. The students will learn to use specific tools to write and retrieve data from InfluxDB. Additionally, ways to create dashboards will be explored, where the accumulated data can be presented and visualized will be explored.

2.1 Specific outcomes

Upon completion of this lab, individuals will be able to:

- Understand concepts regarding time series databases.
- Manage and interact with InfluxDB through its own tools and the provided API.
- Create dashboards, visualizing the InfluxDB data.

2.2 General description

This laboratory builds on the work done on previous modules. The measurements retrieved in previous laboratories will be received in this laboratory and stored in a database. Specifically, we will use a time series database named InfluxDB. The data storage will be implemented using Telegraf, which is an InfluxDB tool and using python scripts utilising the InfluxDB API. Finally, the data will be visualized using the built-in tools of InfluxDB along with a separate tool, Grafana.

InfluxDB is a time series database, which means that it is optimized to handle data that are time-stamped. Time series represent measurements or events that evolve over time, for this reason a time-series database should be able to track this relationship, aggregate over time, handle large data sets, etc [1].

In relational databases we have tuples, attributes and tables that are used as abstractions to store and organize data. In time series, as the needs change, so does the abstractions with which we refer to the stored data. More specifically InfluxDB is structured firstly in **Buckets**. A bucket is a database together with a retention policy that describes how long the data will be held before they are deleted. The data a bucket holds are called **Points** and they can be generally described in the following format, which is called **Line Protocol**:

```
measurement[,tag_key=tag_value[, tag_key=tag_value]]
field_key=field_value[, field_key=field_value] timestamp
```

Anything inside square brackets can be omitted. Below is a description for each terminology.

- **Measurement**: a description of the data stored in the associated fields. It is used to associate related points with different field sets and tag sets.

- **Tag set**: the tag key-value pair (dictionary), which provides certain metadata about that point. Tags are indexed, which means that a query on a tag value will be significantly faster. It is a good practice to use commonly queried metadata for tags. However, as they are indexed, it is not a good idea to have tag sets with many possible values.
    - **Tag key**: of string type
    - **Tag value**: of string type
- **Field set**: the field key-value pair (dictionary) that represents the actual data that is being recorded by the point. Fields are not indexed, which means that queries on fields values will be significantly slower as they need to scan all field values to match.
    - **Field key**: of string type
    - **Field value**: of string, float, integer or Boolean type
- **Timestamp**: the date and time associated with a point. Timestamp can be omitted when writing a point and InfluxDB will insert the current time.

For example, based on the measurements retrieved in the previous laboratories we would have:

```
1  rpi,sensor=ina260 voltage=3.5,current=0.004 1630149279
2  rpi,sensor=sht40 humidity=32,temperature=40 1630149279
```

Which would mean that we store all this in the same set of measurements (rpi), as they come from the same source, the raspberry pi. In line 1, we describe that this point belongs to the ina260 tag. While in line 2, we describe that this point belongs to the sht40 sensor. Then we put the measurements as fields and finally a timestamp in RFC 3339 format [2].

Another term that may come up when working with InfluxDB is **Series**. Series are collections of data in the same bucket that share a measurement, and tag set. Therefore, in the above example we would have two series. One for the INA260 sensor and one for the SHT40 sensor.

For further study on the InfluxDB terms, you can check the official glossary and documentation [3].

For this Course we will be using InfluxDB 2.0.

2.3 Lab configuration

The following parameters are provided by the lab instructor:

Property	Value
Workstation IP address	
Workstation username	
Workstation password	
Remote access method for the workstation	VNC / SSH
INFLUXDB_IP	
ORGANISATION (needed in exercise 1)	
TOKEN (needed in exercise 1)	
Web link of InfluxDB installation	
MQTT_BROKER_LOCATION	
MQTT_TOPICS	
Web link of Grafana installation	
Grafana username	
Grafana password	

3. Exercise 1: Introduction

The first exercise will be an introduction to the basic commands of the InfluxDB. The goal is to familiarize yourself with the concept and interact with the database.

Step 1: InfluxDB CLI Configuration

The most basic way in which we can interact with the InfluxDB is through the CLI (Command Line Interface). For this we will use the `influx` command, which is installed by default in the lab. As with every database, we should provide some credentials. InfluxDB uses the concepts of tokens for identifying the appropriate users. The lab instructors should have provided a token to you, along with the location of the database. Now run the following command in the terminal replacing the appropriate fields (check section 2.3 to retrieve the appropriate values for database IP, organisation, and token):

```
1 influx config create -n default \
2 -u http://***INFLUXDB_IP***:8086 \
3 -o ***ORGANIZATION*** \
4 -t ***TOKEN*** \
5 -a
```

The above command creates a new configuration with the following flags:

- -n: Sets the name of the configuration to be “default”.
- -u: Defines the location of the database to connect to.
- -o: Defines the organization the user belongs to.
- -t: Defines the token of the user.
- -a: Sets the newly created configuration to be the active one.

Additionally, the backslashes (\) indicate that the command is multiline. This can be omitted, if the command is written only in one line, however it is more readable this way.

To make sure the configuration has been applied correctly you may run the following command.

```
influx bucket list
```

If the response returns any error, it means that the configuration has not been set right. In such case, you should delete the stored configuration and try again. To do this you may run:

```
influx config list
```

And then

```
influx config delete ***NAME***
```

Where ***NAME*** is the configuration name, the previous command has returned. Then try again. In any case, please reach an instructor to make sure everything is right.

Step 2: Create a Bucket and Store Data

Now that we have configured the influx cli tool to connect to the influxDB database, we can create our first Bucket and store some data in it.

To create a bucket, use the following command

```
1  influx bucket create -n test -o ***ORGANIZATION*** -r 30d
```

The above command creates a new bucket with the following flags:

- -n: Sets the name of the bucket. In our case the name is test
- -o: Defines the organization the user belongs to.
- -r: This sets the retention policy, as it has been explained in the introduction. The "`30d`" here denotes 30 days. Alternatively, "`m`" could be used for minutes, "`h`" for hours, etc.

You can use the CLI built-in help to read more about possible commands and flags. To do this you can run `influx help` or use the `-h` flag for each subcommand. E.g. `influx bucket -h` or `influx bucket create -h`. Additionally you can check the InfluxDB documentation for further explanations, like for example all the different units that can be used in the retention policy flag, etc.

We can check that the bucket has been created, along with every other bucket in the database, you have access to, by running:

```
influx bucket list
```

Now it is time to insert some data to the bucket. We will use again the CLI to achieve that. Use the following command:

```
1  influx write \
2  -b test \
3  -o ***ORGANIZATION*** \
4  'rpi,sensor=ina260 voltage=3.5,current=0.004' \
5  'rpi,sensor=sht40 humidity=32,temperature=40'
```

The `-o` flag again denotes the organization, the `-b` flag denotes the bucket in which we want to write.

You see that the format of the points is the same as in the one discussed in the General Description section, the Line Protocol Format. Additionally, you can see that the timestamp has been omitted, which means that InfluxDB will assign the current time.

Using the CLI, we can enter the various point one after the other, enclosed in single quotes. Of course, this is not practical and is used here for demonstration purposes.

To write multiple points together, we can store them in a file. Each point has to be in a separate line, following the Line Protocol format like so:

```
1  rpi,sensor=ina260 voltage=3.5,current=0.004 1630149279
2  rpi,sensor=sht40 humidity=32,temperature=40 1630149279
3  rpi,sensor=ina260 voltage=3.6,current=0.003 1630149339
4  rpi,sensor=sht40 humidity=30,temperature=42 1630149339
5  rpi,sensor=ina260 voltage=3.4,current=0.005 1630149399
6  rpi,sensor=sht40 humidity=32,temperature=39 1630149399
```

For each record, a timestamp must be provided in Unix epoch time format (i.e., seconds elapse from 00:00:00 UTC on 1 January 1970). Each timestamp must be inside the retention policy range!

Create a file with the above contents and name it measurements.txt. Then by invoking the following format with the `-f` flag, it will read the file and write each individual point to the bucket:

```
1  influx write \
2  -b test \
3  -o ***ORGANIZATION*** \
```

```
4  -f measurements.txt
```

Instead of the Line Protocol, you can also structure your files in a CSV format as described in the InfluxDB documentation. We will come back on the CSV format later this lab.

Finally, you can delete point by using the influx delete command like so:

```
1  influx delete \
2  --start '<start-date>' \
3  --stop '<end-date>' \
```

The format of the <start-date> and <end-date> should be in this form **Y**-**M**-**D**T**h**:**m**:**s**Z, where the bold letters represent the **Y**ear, **M**onth, **D**ate, **h**our, **m**inutes and **s**econds. For example, 2021-08-28T11:28:31Z.

Remember that we are working from the CLI. That means that we can use the bash environmental variables. So, if you want to have the current date you could invoke `$(date +"%Y-%m-%dT%H:%M:%SZ")`, which will return the current timestamp in the appropriate format, to use it as <end-date> for example.

In the next step, we will investigate queries, with which you can get the data you want from a bucket. There we will explain how to structure and use a query in detail. However, to check that the above points have been written in the bucket you can use the following query command to get the contents of the test bucket.

```
influx query 'from(bucket: "test") |> range(start:-30d)'
```

This command will return all the points that you have stored in the bucket in the last 30 days. Remember that we have defined the retention policy to be 30 days, so the returned points will essentially be everything that is stored in the bucket. In our case, they should be the 8 points added above, 2 by the CLI directly and 6 from the file.

The query above is not practical in real-life cases, as the points would be too many to handle. It is shown here only for demonstration purposes.

Before moving on to the next exercise, you are encouraged to experiment with what we have described so far. Insert random points, check that they are written in the bucket, delete them etc. This process will help you better understand the format of the data and how each command works. Remember that you can use the -h flag, to see the help description of each command.

Step 3: Queries

The purpose of databases is not just to store data, but to enable the user to access data in an efficient and easy way. To do that we use queries, which are structured requests that when run by a database would retrieve specific data that match our criteria. InfluxDB uses a powerful query language called **Flux**. As with before, we will use the CLI tool to begin with.

First, we will need a bucket with data, on which we will test our queries. Run the following command.

```
influx bucket list
```

You should see that there is already a bucket called Sample. Along with it you should see the test database of the previous steps and every, if any, buckets you have created yourself.

The Sample bucket has a sample dataset, provided by InfluxDB [4] that contains humidity, temperature and CO2 measurements of eight different sensors. The dataset loaded is from a certain time period, specifically from 2021-08-29T16:48:40Z until 2021-08-29T17:48:40Z.

The dataset contains one measurement, named **"airSensors"** and is structured with one tag, with a key named "**sensor_id**" that can take the following values for each of the eight sensors: {`TLM0100, TLM0101, TLM0103, TLM0200, TLM0201, TLM0202, TLM0203, TLM0300`}. The fields of the dataset are three **"co"**, **"humidity"** and **"temperature"**.

With that in mind, we will build some sample queries. A Flux query comprises of four parts:

1. Data Source: The bucket from which we will retrieve the data.
2. Time Range: The time range in which the data reside.
3. Data Filters: The filters with which we will narrow down our search.
4. Transform Functions: The function that will be used to transform the retrieved data in some way.

From these four parts, only the two first are obligatory. The Data Filters and the Transform Functions could be omitted, if not needed, but in almost all real-life cases you will use them.

Below you can find a simple query:

```
1  from(bucket: "Sample")
2  |> range(start: 2021-08-29T16:48:40Z,
3           stop: 2021-08-29T16:48:50Z)
4  |> filter(fn: (r) => r._measurement=="airSensors")
5  |> filter(fn: (r) => r.sensor_id=="TLM0202")
7  |> yield()
```

In line 1, we describe the Data Source. Specifically, we describe that the data will be retrieved from the bucket named "Sample.

In lines 2 and 3, we describe the Time Range. Specifically, we describe that we want the data in the time frame of 10 seconds.

In lines 4 and 5 we describe the Data Filters. Specifically, we describe that we want data that belong to the measurement of "`airSensors`" and that have the tag of "`TLM0202`".

Finally in line 7 we describe a Transform Function. The `yield()` command is not a transform function, it just returns the data of the query and in general it can be omitted, unless we have multiple queries in the same Flux query. We will later see other Transform Functions to understand their uses.

There are two ways to run the above query. The first is directly from the CLI, as we have done in Step 2. However, that way is not convenient. Instead, you should save the above query in a file named for example "firstQuery.flux" and run the following command

```
influx query -f firstQuery.flux
```

The retrieved data should be three points, for co2, humidity and temperature.

Before moving on, let's elaborate on this query a little more. As you can see, each command in the query is connected to the next one with this symbol `|>`. This is called a pipe-forward operator and is used to pipe data from the previous command to the next.

The first command we use is the `range()` function. This receives at most two parameters. A start time and a stop time. The stop time can be omitted, implying current time. The two parameters could be absolute as in our example, but they can also be relative, following the same format as in the retention policy. For example, the following range means from 2 hours before current time, until 5 seconds before current time:

```
range(start: -2h, stop: -5s)
```

Next we use the filter() function. This receives another function as a parameter which narrows down the data it has received. In most cases we would use what is called anonymous function. These are set as in our example in the following format, where we declare an anonymous function with `fn`, with one parameter "`r`":

```
fn (r) => *do stuff*
```

In our example we use two different filter functions. However we could use only one with the "`and`" relational operator as below. In general, the following notation is better and is the one that should be used in most times:

```
1 |> filter(fn: (r) =>
2         r._measurement=="airSensors" and
3         r.sensor_id=="TLM0202"
4 )
```

In most cases, we would use what is known as dot notation to access the fields of a row. Like in the example above in "`r.measurement`" and "`r.sensor_id`". The following notation is also acceptable:

```
|> filter(fn: (r) =>
        r[_measurement]=="airSensors" and
        r[sensor_id]=="TLM0202"
)
```

Finally regarding the Transform Functions. In our example we used just yield(), which is not a real Transform Function. Let's consider the following example, that uses Transform Functions:

```
from(bucket: "Sample")
|> range(start: 2021-08-29T16:50:00Z,
        stop: 2021-08-29T17:50:00Z)
|> filter(fn: (r) =>
        r._measurement=="airSensors" and
        r.sensor_id=="TLM0202" and
        r._field=="co"
)
|> window(every: 10m)
|> mean()
```

Before reading the explanation, study the above query to see if you can understand it by yourself.

The above query retrieves data in the time range of 1 hour that belong to the "`airSensors`" measurements, of the "`TLM0202`" sensor and specifically the field "`co`".

Next it runs two Transform Functions, the `window()` function, which groups the retrieved data, based on the parameter defined. In our case, it collects data points in each time range of 10 minutes. So we will have 6 groups in the end (1 hour has 60 minutes). Additionally, the `mean()` function computes the mean value for each group. There are many more Transform Function. You can refer the InfluxDB documentation for them.

Save the above query to a file, for example "secondQuery.flux" and run it. The response should be six rows, with each one corresponding to the appropriate 10-minute window and with value the computed mean.

Step 4: Web Interface

By now, you should have grasped the basics of the InfluxDB. How data is represented, how it is stored and how it is retrieved. We have used extensively the CLI tool, to familiarize yourselves with the abstract ideas. However, InfluxDB also has a useful and well-made Web Interface, which presents similar functionalities with the CLI tool together with some virtualization of the queried data.

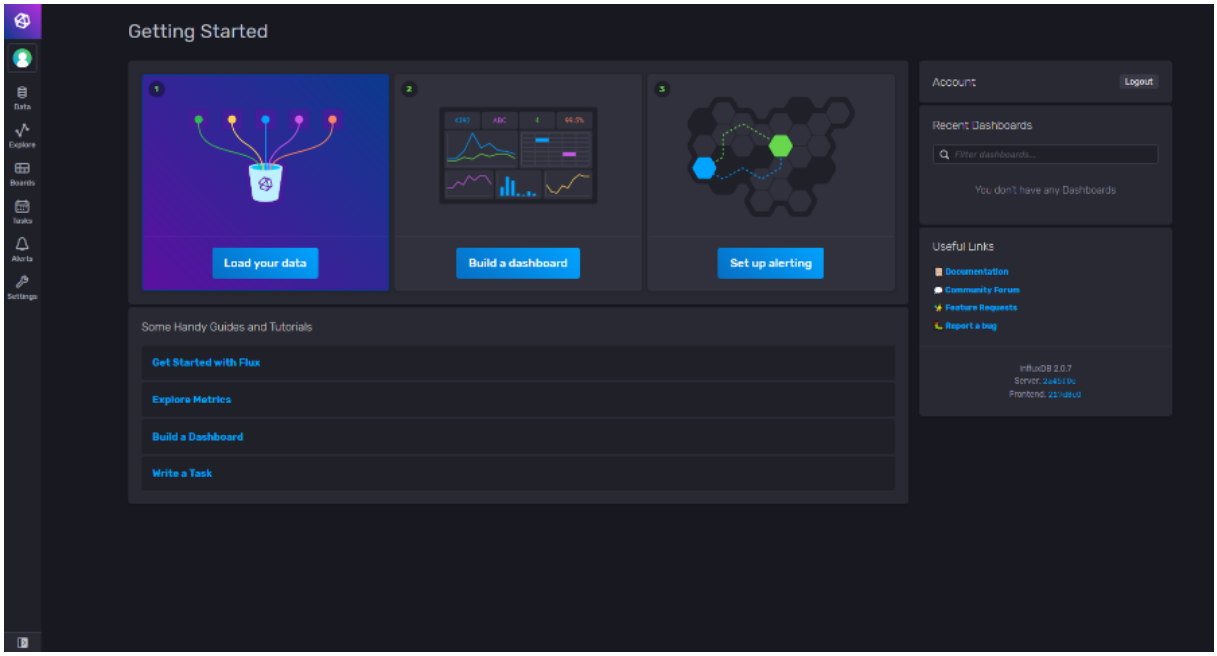


Figure 1: Homepage of InfluxDB

To access the Web Interface, visit the web interface of InfluxDB (the link is provided in section 2.3). After you enter the username and password (also provided in section 2.3), you should see the screen depicted in Figure 1.

The left-side sidebar serves as the menu. There by selecting **Data** and then Buckets, you can see the available buckets, like with `influx bucket list`. The respective page is depicted in Figure 2.

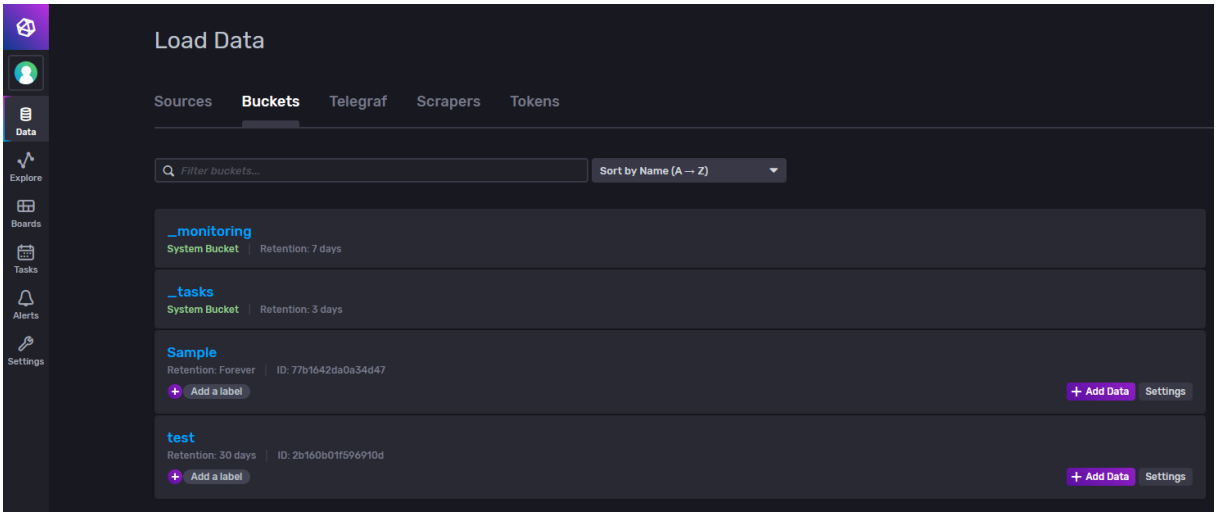


Figure 2: The available buckets in InfluxDB

By clicking the **Add Data** button in a bucket you can add data by uploading a file, just like we did from the CLI. The Add data dialogue is depicted in Figure 3.

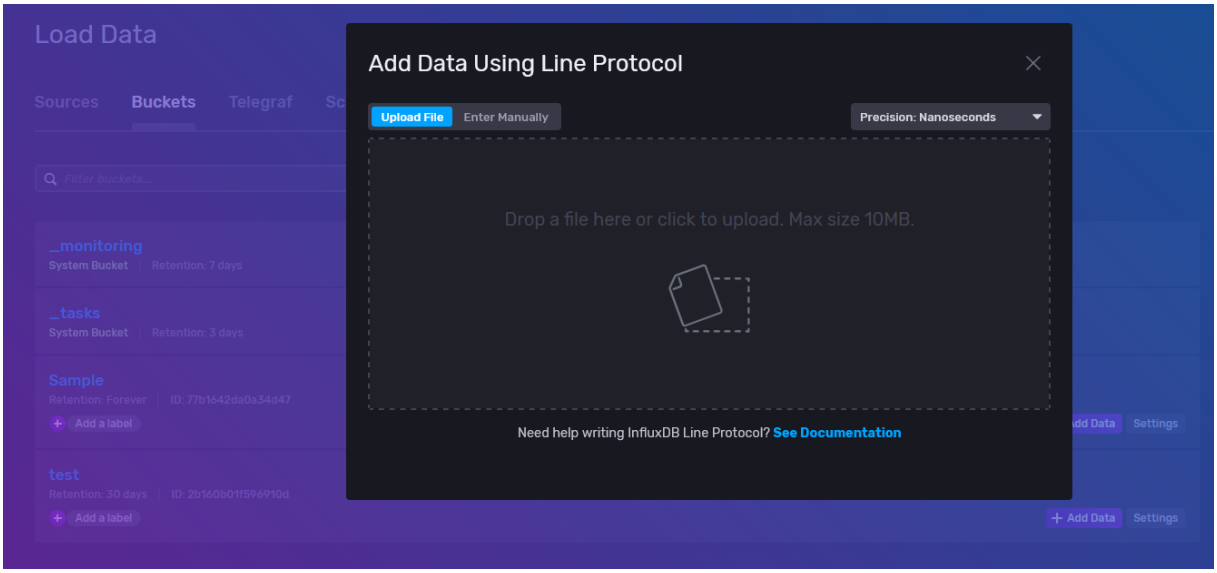


Figure 3: The add data dialogue

In the **Explore** tab, you can find the Query builder, which will allow you to graphically create a query by selecting the appropriate filters. The Explore tab is depicted in Figure 4.

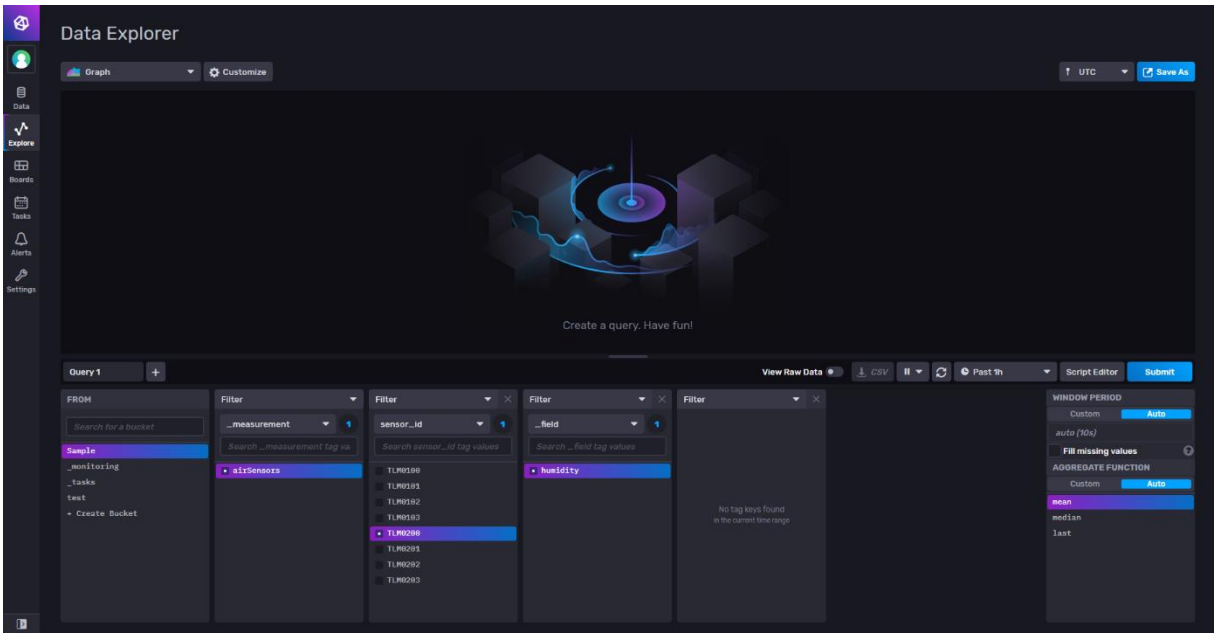


Figure 4: The Explore tab of InfluxDB

Then, after you have submitted the query, you can select the way the data will be presented on the top part of the screen like in Figure 5.

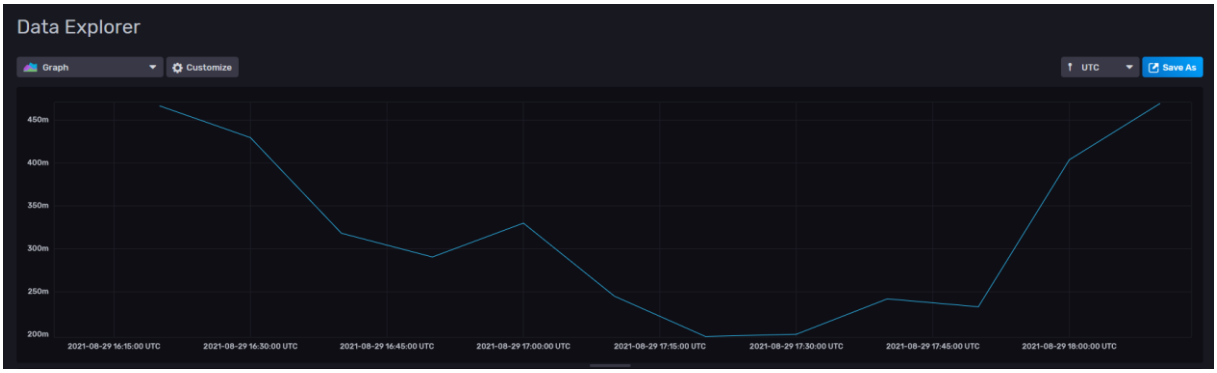


Figure 5: Query result in Data Explorer

Before moving to the next exercise, you are encouraged to experiment with the Web Interface of the InfluxDB. Try to redo the various queries and manipulations we did through the CLI, but on the Web Interface this time. Explore the various presentation options.

4. Exercise 2: Collecting measurements

Now that we have seen how data is stored in InfluxDB along with how you could query this data to retrieve what you want, it is time to explore how you can automate the data insertion. For this we will use the MQTT installation of the previous lab. More specifically we will build two data listeners, based on different tools, who will subscribe to the appropriate MQTT topics and send the data to InfluxDB.

Step 1: Using Telegraf

For the first listener we will use Telegraf. Telegraf is a tool that belongs to the wider influx family of tools. It is a server agent that can read data from various sources (databases, systems, IoT sensors, etc.) and write them in InfluxDB. It is built around different plugins that integrate with existing platforms. Central to Telegraf's operation is its configuration file that define the plugins to be used along with their configuration.

Before creating a configuration file, first create a new Bucket for the data. The easiest way to create a new bucket is through the Web UI, in the **Data** tab and then in **Buckets** select **Create Bucket**. Set the name to "**SensorData**", or something similar and the retention period to at least **1d**.

Now to create the Telegraf configuration file, go to the **Data** tab and then select **Telegraf**. There you will see a **Create Configuration** button. The Load Data menu is depicted in Figure 6.

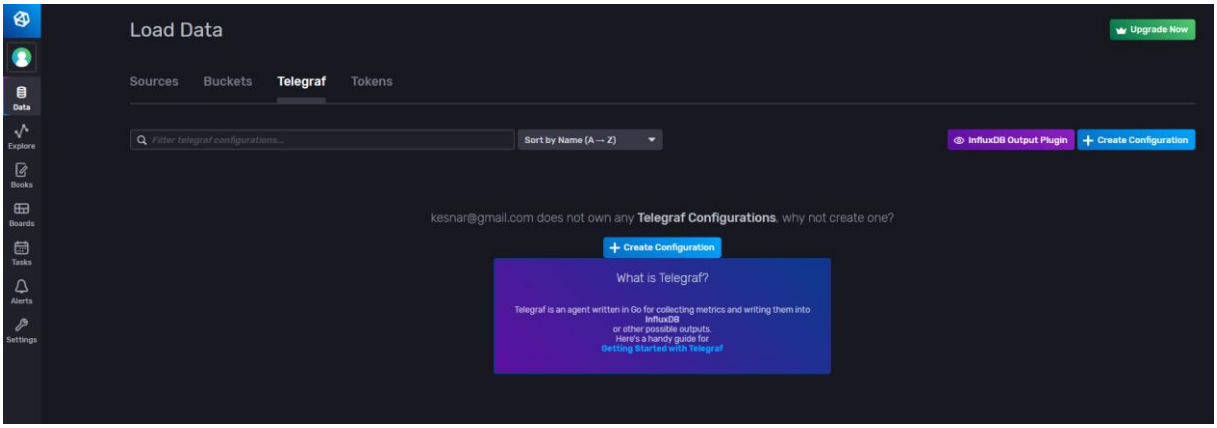


Figure 6: The Load Data menu for Telegraf

By pressing the "Create Configuration" button you will be presented with a selection of plugins to choose from (Figure 7). Unfortunately, the MQTT plugin is not default. So just select one of the existing and we will change it later. Also, select the bucket that we have created.

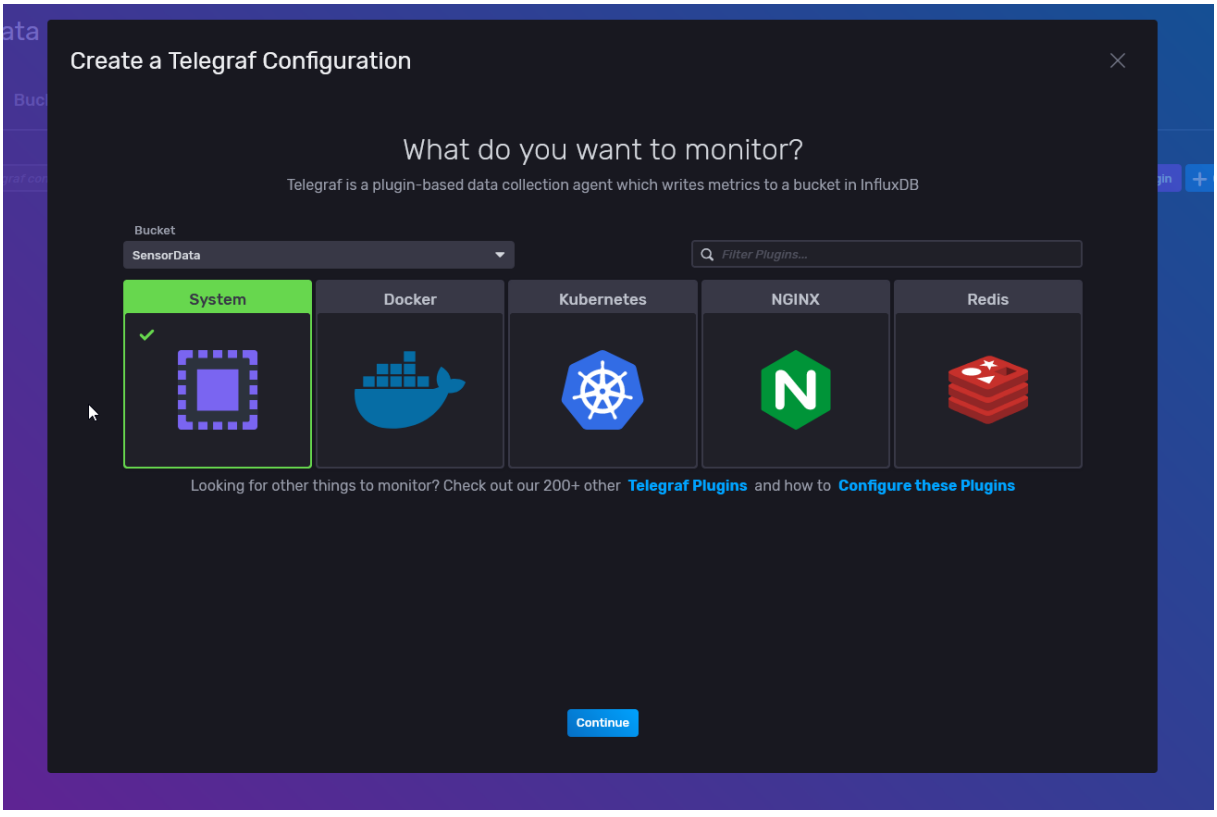


Figure 7: Create a new Telegraf Configuration

Follow the steps by giving a name to the configuration we will create, for example "Sensors". Before the last step make sure to copy the token provided like in the following figure. Here the token is:

```
Eg9gcP57D0in[...]Z7IhNvJVfdMcRUNhc9gnavXRn7ODSe33zw7moaUAxKetKqfnT44w==
```

Like the token you have been provided with, this will be used so that Telegraf will have access to the InfluxDB.

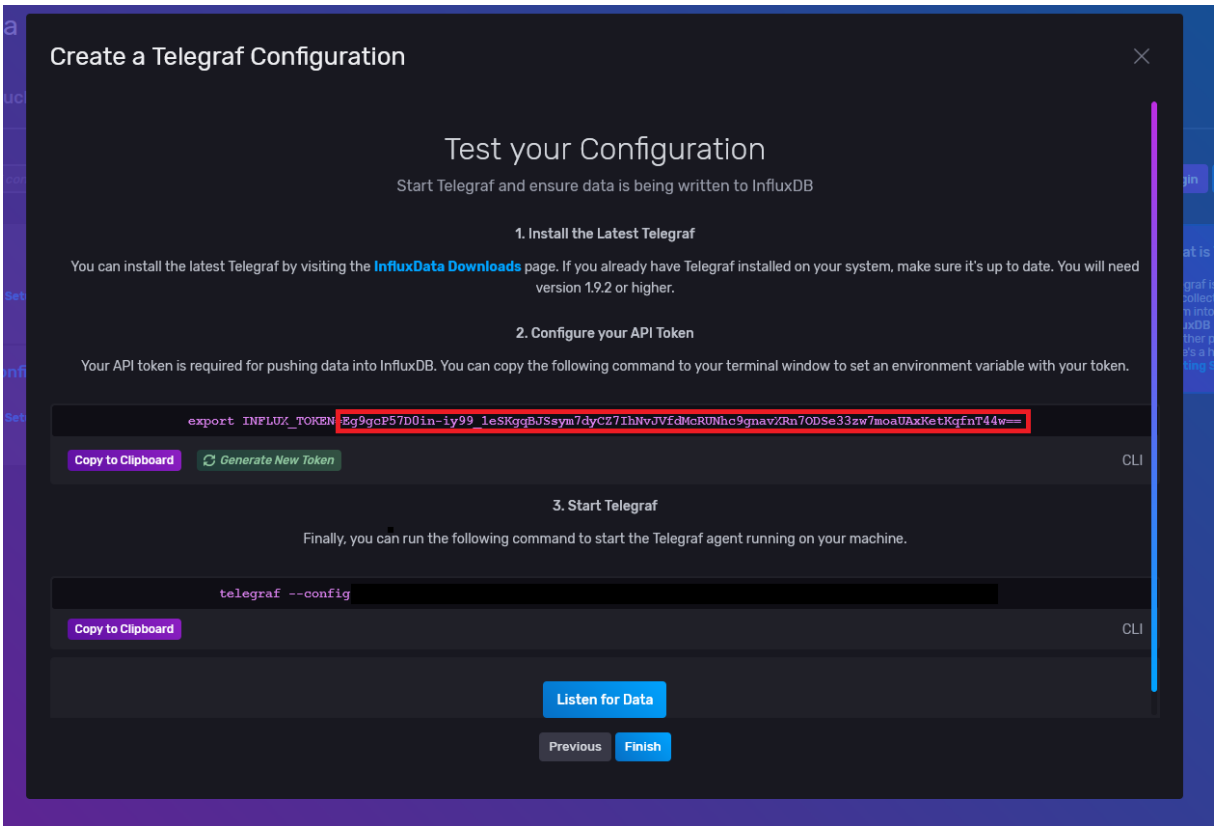


Figure 8: Last screen of the Creating Configuration procedure

The Token, along with any other tokens can also be accessed in the **Data**->**Tokens** section of the Web UI.

Now after pressing Finish, we have created a sample configuration for Telegraf. However, we have not deployed it yet. Remember also that this configuration is not complete, as we have not configured MQTT yet.

To access the MQTT plugin go to **Data**->**Sources** section and search for MQTT. You should see the screen depicted in Figure 9.

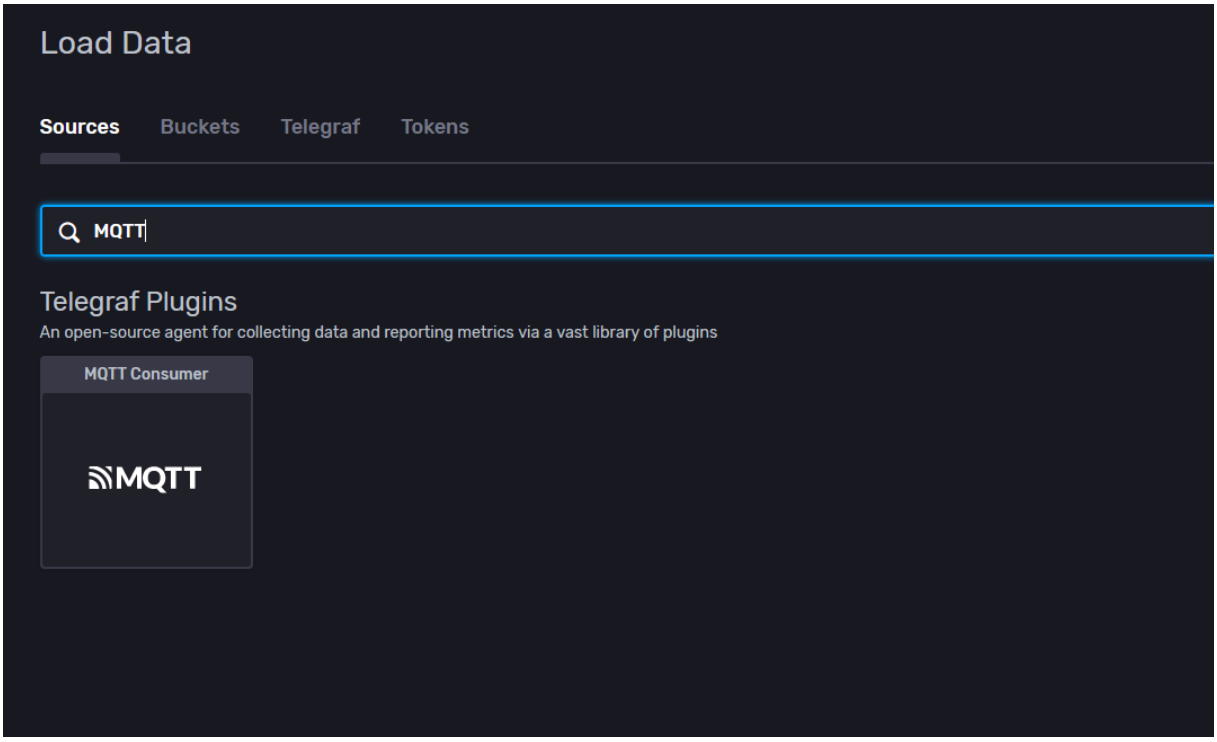


Figure 9: Searching for the MQTT plugin

Pressing the MQTT Consumer will bring you to another screen that will have a sample configuration like below.

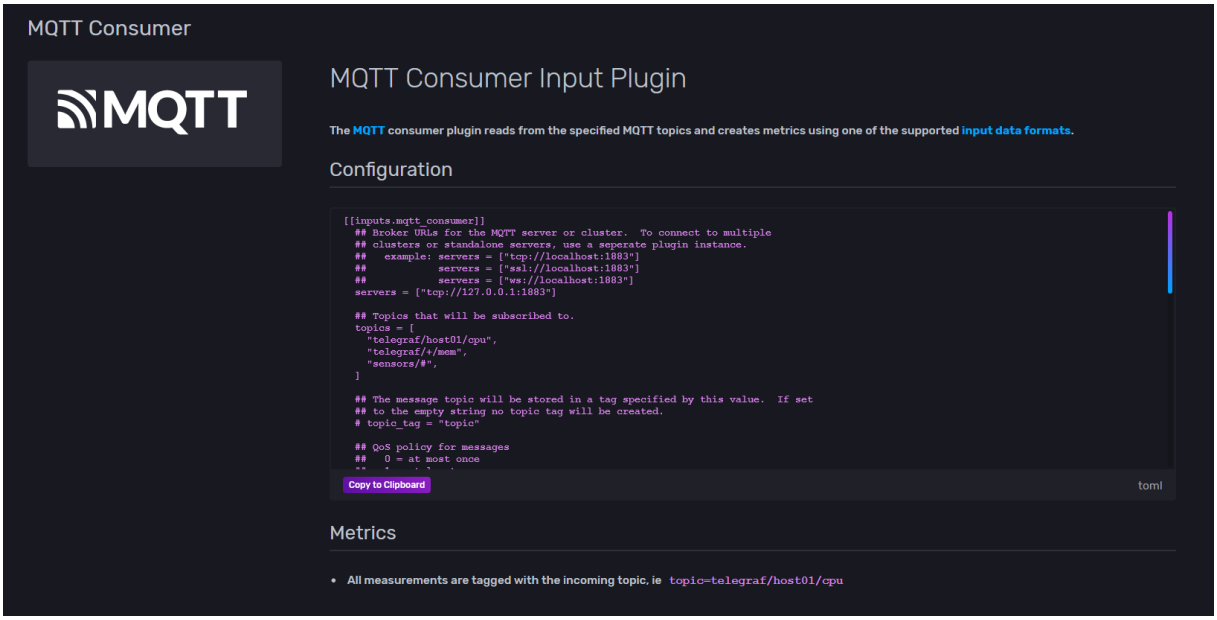
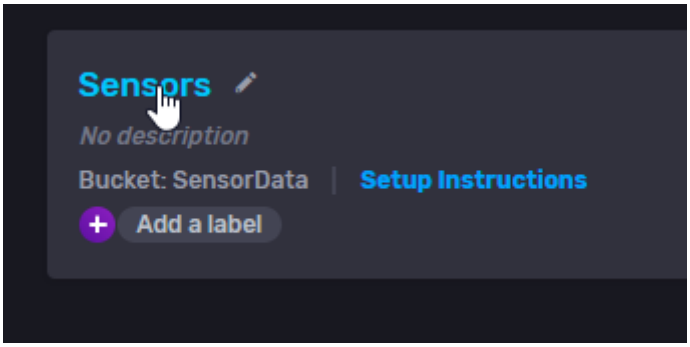


Figure 10: The MQTT Consumer Input plugin

Copy the configuration and return to **Data->Telegraf**. There by pressing the name of the configuration you have created you are able to edit the configuration in an editor.



The configuration file is split into parts denoted by square brackets. The first part is called `[agent]` and has information related to how Telegraf itself will work. When it shall send data, it buffer size, etc. We will leave this part as is.

Next is what is called **Output Plugins**, in our configuration, there should be an `[[outputs.influxdb_v2]]` part. Here we want to change the token line. By default it would be:

```
token = “$INFLUX_TOKEN”
```

Set it to the token you have copied above. For example:

```
token = Eg9gcP57D0i[...]hc9gnavXRn7ODSe33zw7moaUAxKetKqfnT44w==
```

Next you will want to replace all `[[inputs.*]]` parts with the configuration you have copied from the MQTT Consumer plugin. If you have selected System in the original screen, these should be `[[inputs.cpu]]`, `[[inputs.disk]]`, `[[inputs.diskio]]`, `[[inputs.mem]]`, `[[inputs.net]]`, `[[inputs.processes]]`, `[[inputs.swap]]` and `[[inputs.system]]`. Delete these parts.

The MQTT Consumer plugin configuration has many different options to be set. However, most of them are commented out, let’s look at the ones that are active. By default, they should be:

```
1  [[inputs.mqtt_consumer]]
2  [...]
3          servers = ["tcp://127.0.0.1:1883"]
4  [...]
5          topics = [
6                  "telegraf/host01/cpu",
7                  "telegraf/+/mem",
8                  "sensors/#",
9          ]
10 [...]
11         data_format = "influx"
```

The `servers` variable defines the MQTT broker. You should replace this with the MQTT broker socket. This information should be retrieved from section 2.3.

In the `topics` variable you define the topics that telegraf will subscribe and listen to. You should replace this with the MQTT topic provided in section 2.3.

Finally, the `data_format` variable defines the format in which the data will be. In the previous lab we used the JSON format. Specifically, we used messages like the one bellow:

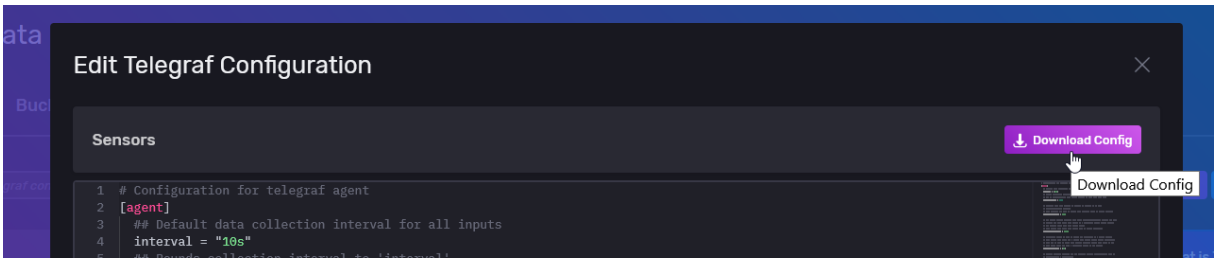
```
{
        "timestamp": 2020-06-30T13:00:43.53849,
        "ina260": {
                "voltage": 3.5,
                "current": 0.004
        },
        "sht40": {
                "humidity": 32,
                "temperature" 40
        }
}
```

Therefore, instead of "influx" this should be set to "json"[1].

The final MQTT plugin configuration should be:

```
[[inputs.mqtt_consumer]]
[...]
        servers = [" ***MQTT_BROKER_LOCATION***"]
[...]
        topics = [
                ***MQTT_TOPICS***,
        ]
[...]
        data_format = "json"
```

Now press the **Download Config** button to download the configuration file.



Then run the Telegraf with the appropriate configuration file:

```
telegraf --config /Path/To/File/sensors.conf
```

[1] For a full list of the various format telegraf can parse refer to **https://github.com/influxdata/telegraf/blob/master/docs/DATA_FORMATS_INPUT.md**

Run a Query on the SensorData Bucket and check the data that Telegraf has written. You may have to wait a little depending on the frequency of the MQTT messages.

Now check the points written and notice that their format is strange. The tagset is **host** and **topic** and the fields are `sht40_humidity, sht40_temperature, ina260_voltage, ina260_current`. This is because we have not set the way Telegraf should parse the message.

Unfortunately, the way we have structured our message in the previous lab does not allow Telegraf to parse it correctly, as it does not have the functionality to recognise json key names as tag keys. To solve this we would have to restructure the messages our sensors send like below:

```
1  {
2      "timestamp": 2020-06-30T13:00:43.53849,
3      "measurements" : [
4          {"sensor": "ina260", "voltage": 3.5,  "current": 0.004}
5          {"sensor": "sht40",  "humidity": 32, "temperature": 40}
6      ]
7  }
```

Additionally, we would have to change the configuration of the MQTT plugin like so:

```
1  data_format = "json"
2  json_time_key = "timestamp"
3  json_time_format = "2020-01-01T00:00:00.00000"
4  tag_keys = ["sensor"]
5  json_query = "obj.measurements"
```

In line 1, we describe the general format as we did before. Then we describe the specific format of our messages. In line 2 we define the json key which holds the time and in line 3, the format in which it is. In line 4, we define which fields will be used as tags. Specifically, the tag set will have “sensor” as key and the respective values. Finally, in line 5 we describe that telegraf must read the array of the measurements field to get the appropriate data.

Of course, changing the messages, a sensor sends, is not always possible. So we need another more flexible way to parse the messages.

Step 2: Using Python

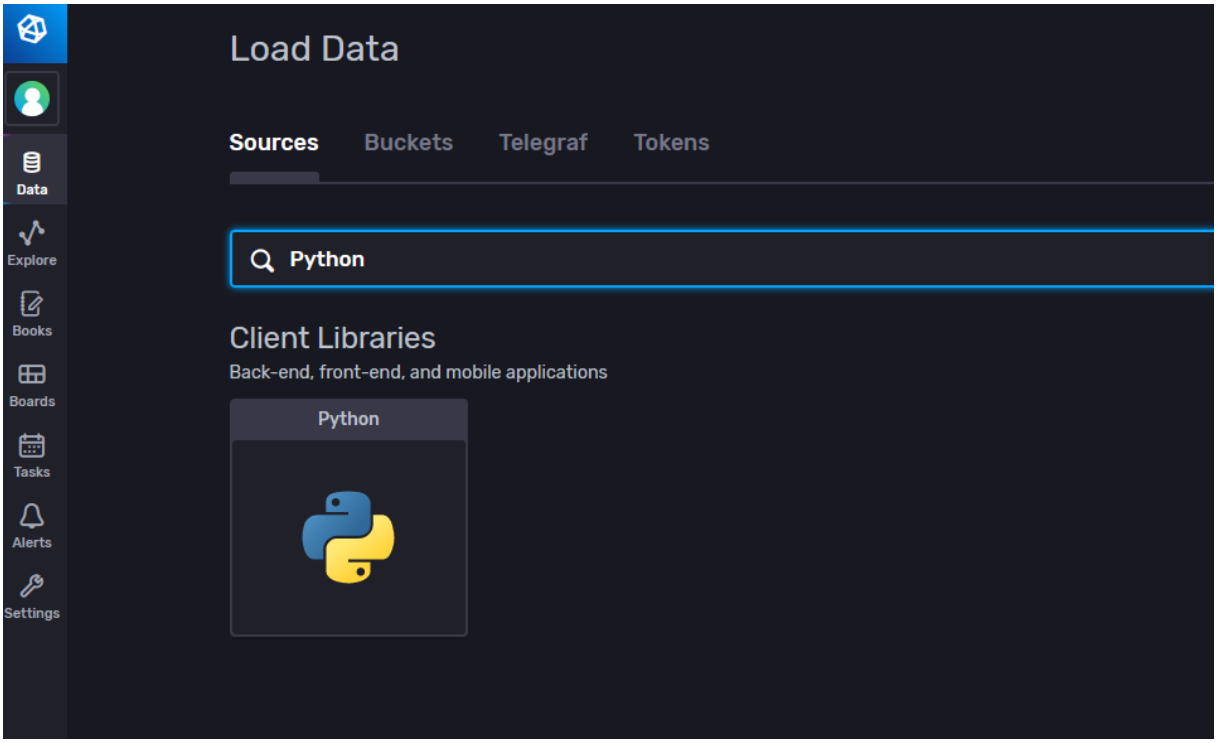
A more flexible way to subscribe to an MQTT broker, parse messages and write them as data points in InfluxDB is by using a Programming Language like Python. This will be the implementation of the second listener.

Before continuing stop Telegraf, then delete the SensorData Bucket and create it again. Now we have an empty bucket. Let’s see how we can use Python to populate it with data from MQTT messages.

We will not cover how to write a Python script to act as an MQTT subscriber in this lab. You should be able to write one yourself, based on the previous labs. Try to write a script that listens to the measurements and prints them in the stdin.

If you have any problems regarding this, study again the previous labs.

By accessing **Data**->**Sources** and searching for Python, InfluxDB will provide you with some snippets of code that can be used to insert points to a database.



We will use the following:

```
1  from influxdb_client import InfluxDBClient, Point, WritePrecision
2  from influxdb_client.client.write_api import SYNCHRONOUS
3
4  #Connection to InfluxDB
5  token = "***TOKEN***"
6  org = "***ORG***"
7  bucket = "SensorData"
8  client = InfluxDBClient(url="***INFLUXDB_IP", token=token)
9  write_api = client.write_api(write_options=SYNCHRONOUS)
10 [...]
11 #Writing of Points
12 points = ["***Point1***",
13             "***Point2***"]
14 write_api.write(bucket, org, points)
```

In lines 5-8 we set the parameters to connect to the InfluxDB. These should be the same as the ones we used earlier for Telegraf. In lines 11-14 we create an object that can write a list of points to the bucket and then write them.

Our only problem now is how we will populate the points list. For that we should again study the format of the sensor messages we receive. There we can see that for each MQTT message that we receive, we have to construct two points (one for each sensor) and each point should have 1 tag (the sensor tag) and 2 fields that correspond to the measurements.

For this we will use the "`json.loads()`" of the Python standard library that converts a json string as a python dictionary.

Before moving forward try to write your own solution to the problem. Begin with describing the steps we should take, convert this description to an algorithm written in pseudo-code and then try to implement it in Python

The code that parses the json message, creates the points and populates the points list is below:

```
1  import json
2  from datetime import datetime
3  [...]
4  def populatePoints():
5      points =[]
6      #Parse the JSON message as a dictionary
7      message = json.loads(msg)
8
9      #Remove timestamp from the dictionary and convert it to epoch format
10     timestamp = message.pop("timestamp")
11     timestamp = datetime.strptime(timestamp, "%Y-%m-%dT%H:%M:%S.%f").timestamp()
12
13     #Iterate over the dictionary and create the point
14     for sensor in message.keys():
15         sensorMeasurements = message[sensor]
16         #Add measurements and tags
17         tmpPoint = "measurements, sensor={tag} ".format(tag=sensor)
18         for measurement in sensorMeasurements.keys():
19             #Add fields
20             tmpPoint += "{field}={value},".format(field=measurement,
   value=sensorMeasurements[measurement])
21         #Add timestamp
22         tmpPoint = tmpPoint[:-1]
23         tmpPoint += " " + str(timestamp)
24
25         #Append it to the points list
26         points.append(tmpPoint)
27
28     return points
```

The most interesting parts of the code above are:

- In line 10: We `pop()` the "`timestamp`" field from the dictionary. In that way we are left with only tags and fields in our dictionary.
- In line 11: We convert the sent time to epoch format, which is the one InfluxDB recognises.

- In line 14: We iterate over the dictionary and we know, from the JSON schema, that this iterator will correspond to sensors, so we create the appropriate part of the point, which is the tag.
- In line 18: We iterate over the nested dictionary. In the same sense as before, we know that this corresponds to the fields.
- In line 22-23: Finally we add the timestamp we have calculated before.

By combining the snippets above with the MQTT subscriber you should be able to write the measurements in InfuxDB in the exact format we want them. In essence, you just have to add the appropriate imports. Then add the configurations. Finally, you have to run `populatePoints()` every time a new message arrives and then send the points list as shown.

We have seen that by using the Python InfluxDB client we can be more flexible regarding the format of the message we receive as we create the point ourselves.

5. Exercise 3: Visualization

As we have said before, the most important part of a database is not just that it stores data, but that we can access this data in a structured, efficient way. In the case of time-series database there is another layer to that. The nature of time-series is such that the best way to present data is through visualization. That is why when we use a time-series database, we usually couple it with a dashboard, where the user can monitor the measurements.

Step 1: InfluxDB Dashboards

InfluxDB has a built-in tool with which you can create dashboards. This used to be a separate tool in the Influx family, called Chronograf. However, from InfluxDB v2.0 onwards, it has been incorporated in InfluxDB itself and is now called simply InfluxDB dashboards.

We have seen a part of this tool earlier when we used the Data Explorer of InfluxDB. There we saw that we can present the data from a query in various ways, as graphs, as gauges, as tables etc. The idea behind InfluxDB dashboards is to use these various visualizations with specific queries to create dashboards.

You can access this tool from the left-hand menu, under **Boards**. There you will see all, if any, dashboards you have. This menu is depicted in Figure 11. Click on the “Create Dashboard” button to create a new dashboard. This action will open the menu depicted in Figure 12.

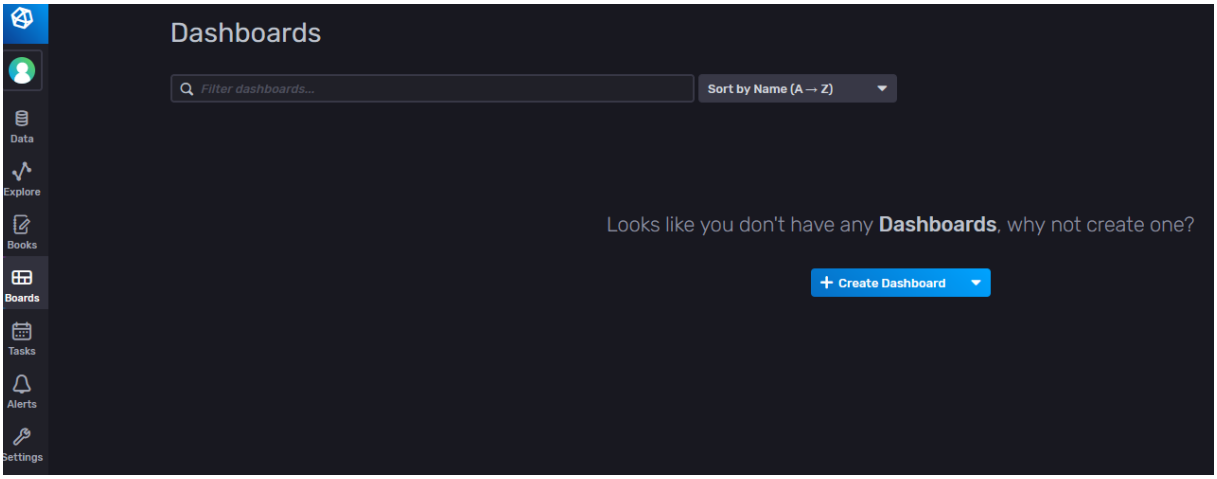


Figure 11: The Dashboards menu

A Dashboard contains Cells, which you can create by pressing the **Add Cell** button. This will open the dialogue depicted in Figure 13.

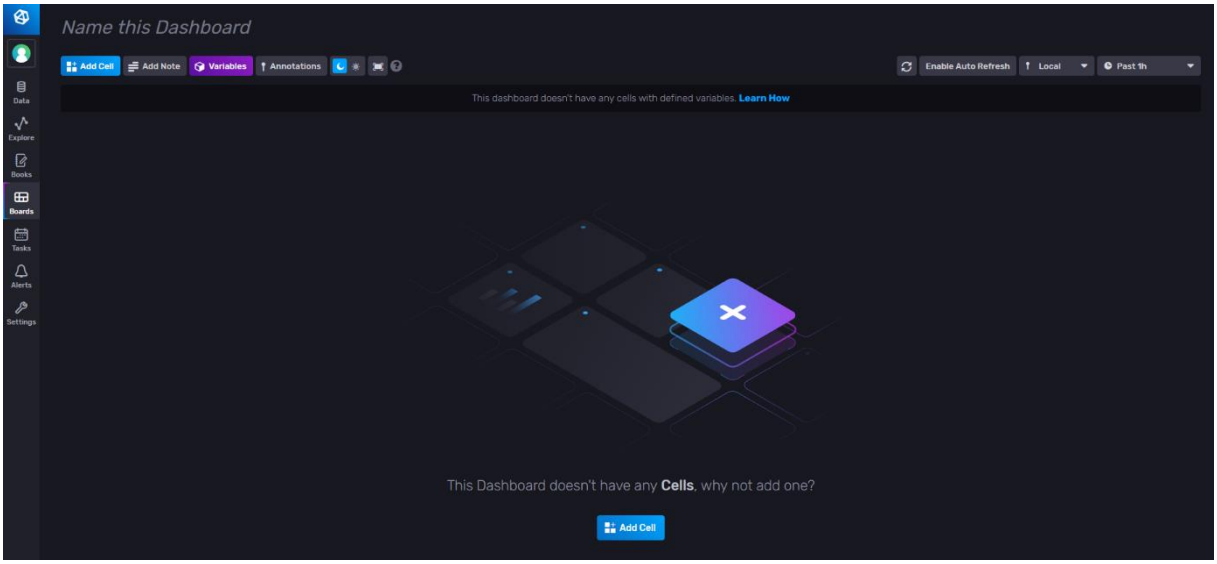


Figure 12: The new dashboard menu

Each cell is nothing more but a query on the data as we have seen before. However, creating specific, structured queries and presenting them all together can create a very specific image on the system we are monitoring.

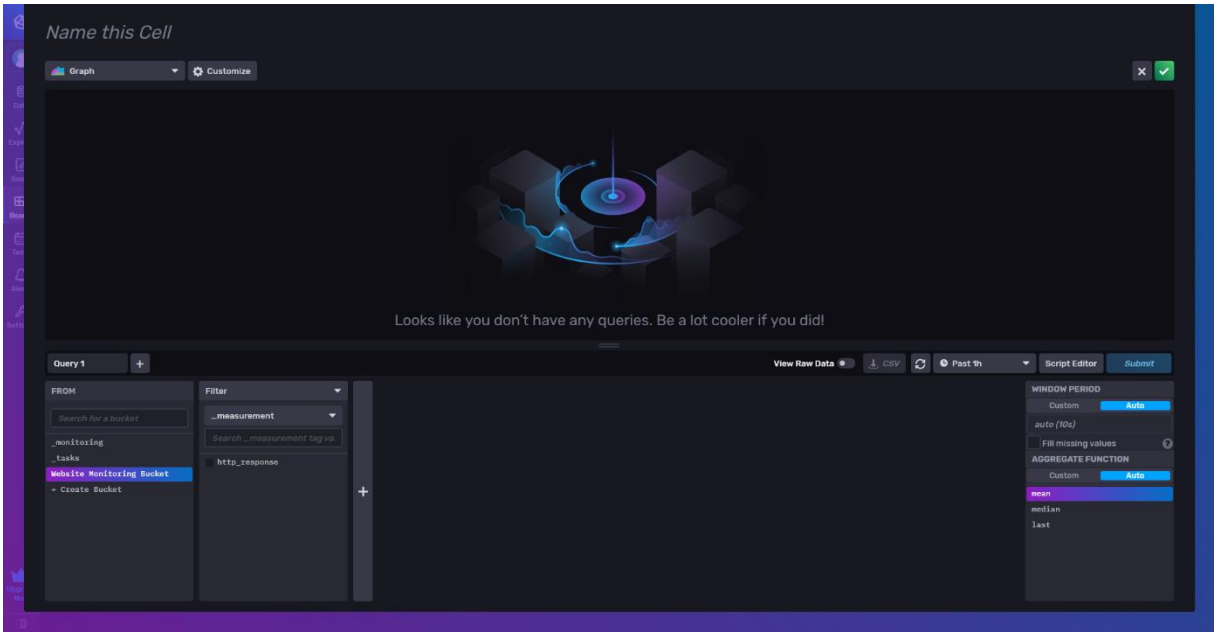


Figure 13: The dialogue for creating a new cell

A complementary tool to the dashboards of InfluxDB is the Alerts. These enable you to create certain queries that check for a specific event to happen (threshold exceeded, service stopped, etc.). When that happens, you can create preconfigured notifications to be send to some service, like an e-mail.

To access these alerts, press the **Alerts** button on the left-side menu. The Alerts menu is depicted in Figure 14. There you can set the queries that will check for events and then set the endpoints, which will receive a notification if the alert is triggered.

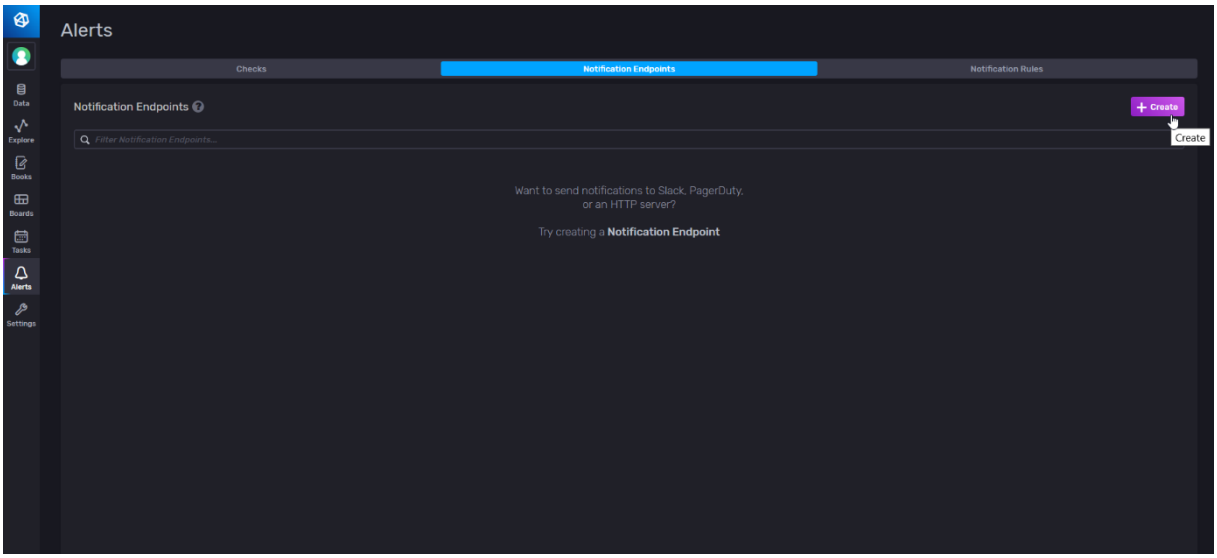


Figure 14: The Alerts menu

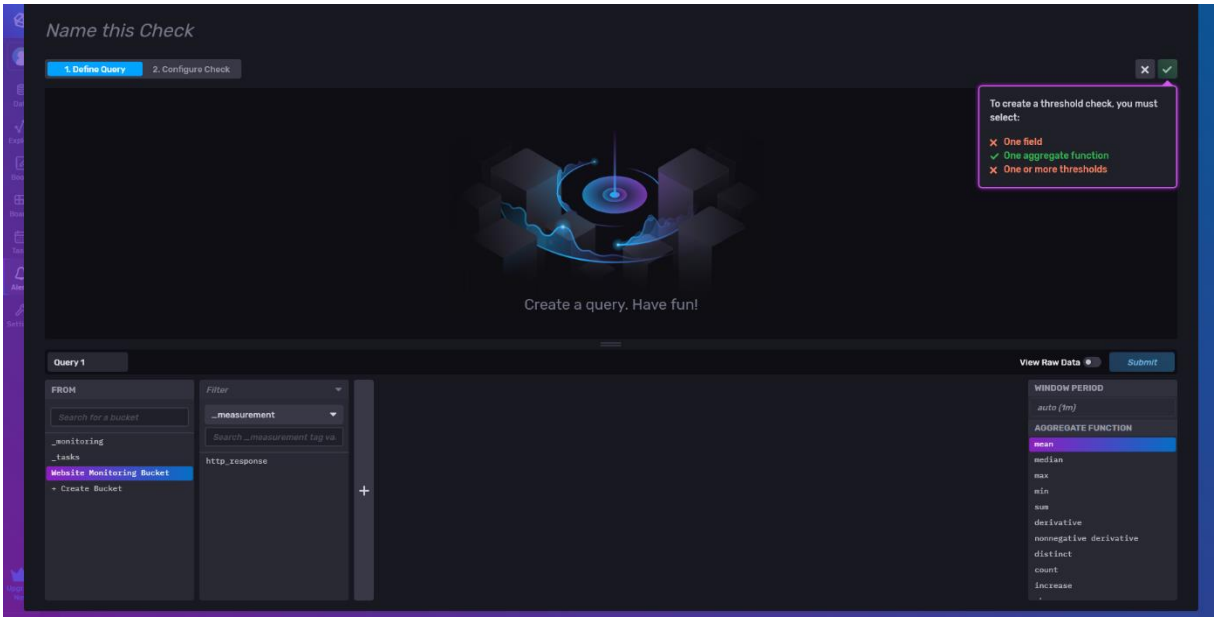


Figure 15: Creating a new alert

You can send notifications to various endpoints. By default, InfluxDB supports Slack, PagerDuty and custom HTTP APIs.

As an exercise, create a dashboard for the data you have inserted to the SensorData Bucket. You are free to decide what information is important to present along with the appropriate graphs that best present the information.

A Dashboard is just a collection of queries. So, if you have any questions regarding how to build a query, refer to Exercise 1, along with the InfluxDB documentation. Remember that you can also use the visual editor of InfluxDB to create queries.

Step 2: Grafana

While you can build simple dashboards with InfluxDB, there are also other tools that are dedicated to Dashboard building. These tools provide more flexible and full ways to present your data, but more importantly, they provide ways of integrating data from various sources. For example, you can use data stored in various InfluxDB databases, or even different databases altogether.

One of the most famous and useful such tools is called Grafana and is already installed in your workstations. You can access the Grafana Web GUI by visiting the link provided in section 2.3 and log in with the credentials you have been given in section 2.3.

Before configuring Grafana, we should create an InfluxDB Token, with which we will connect Grafana and InfluxDB.

In InfluxDB, access **Data**->**Tokens** and select **Generate Token**->**Read/Write Token**. You will be asked to name the token and set the buckets you can read and write with this token. In our case, we just want to read from the **SensorData** bucket and not write to anything. So, make the appropriate configurations. The new token dialogue is depicted in Figure 16.

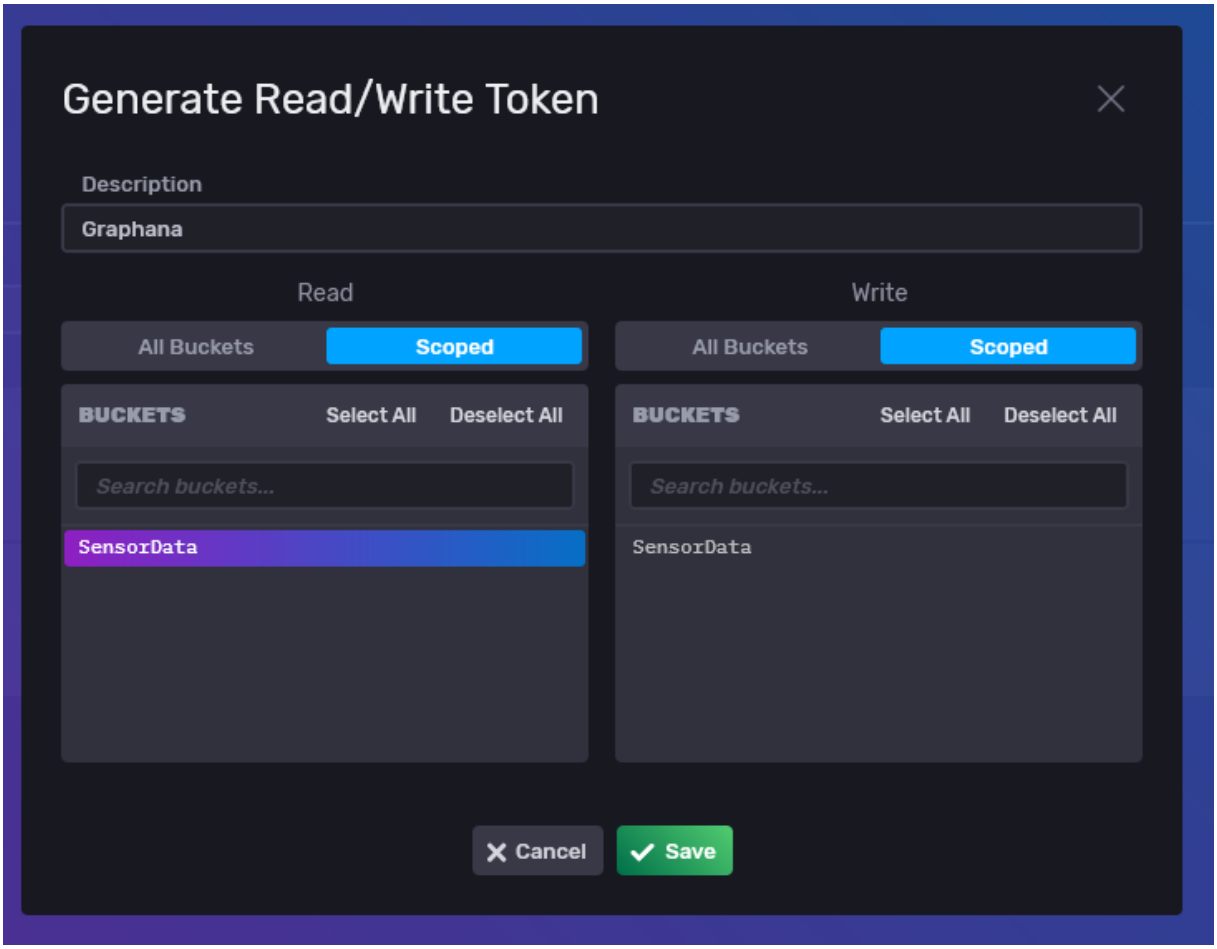


Figure 16: Creating a token on InfluxDB for Grafana

After the token has been created, you can access it by pressing its name, as depicted in Figure 17.

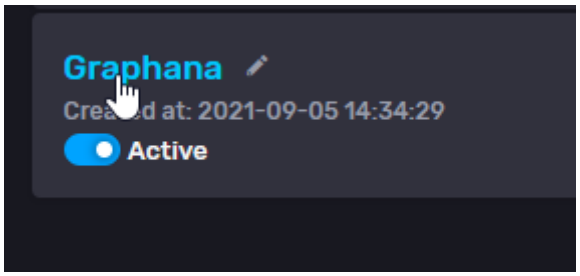


Figure 17: Opening the Grafana token by pressing its name

A dialog will open that will show you the token, which you can copy, along with a summary of its permissions.

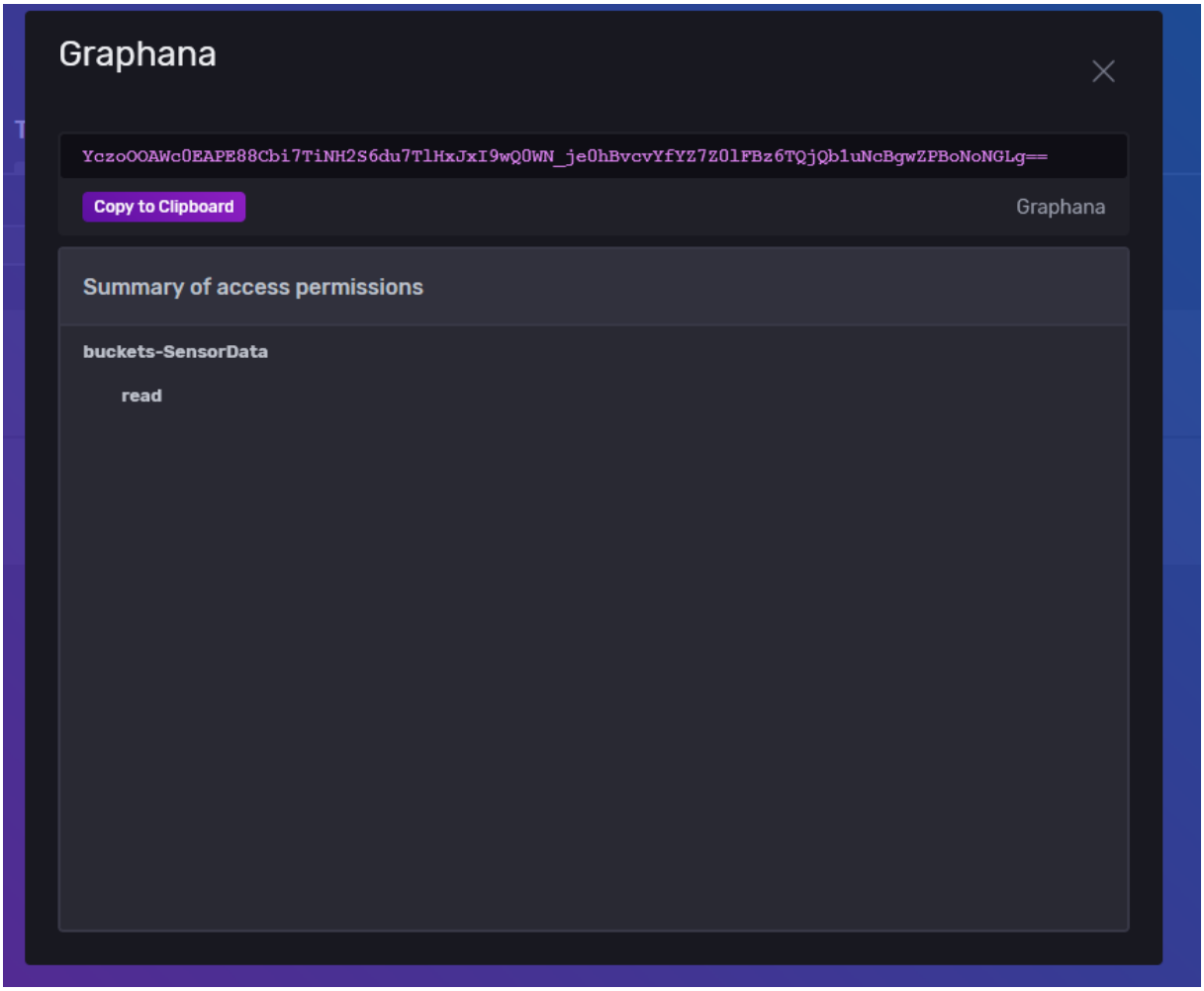


Figure 18: Viewing the token for Grafana

Copy the Token and then access the Grafana Web GUI to connect Grafana with InfluxDB. To do this, access the **Configuration**->**Data Sources** menu on the left side of the screen, as shown in Figure 19.

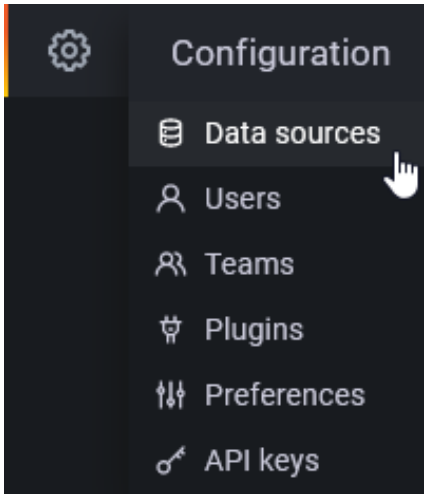


Figure 19: Accessing the Data sources menu on Grafana

Next, select the **Add data source** button. There you can select the kind of data source you want to connect to. As you can see, Grafana can connect to many different databases. Select InfluxDB. You will be presented with many options to configure the connection. The menu for configuring the new data source is depicted in Figure 20.

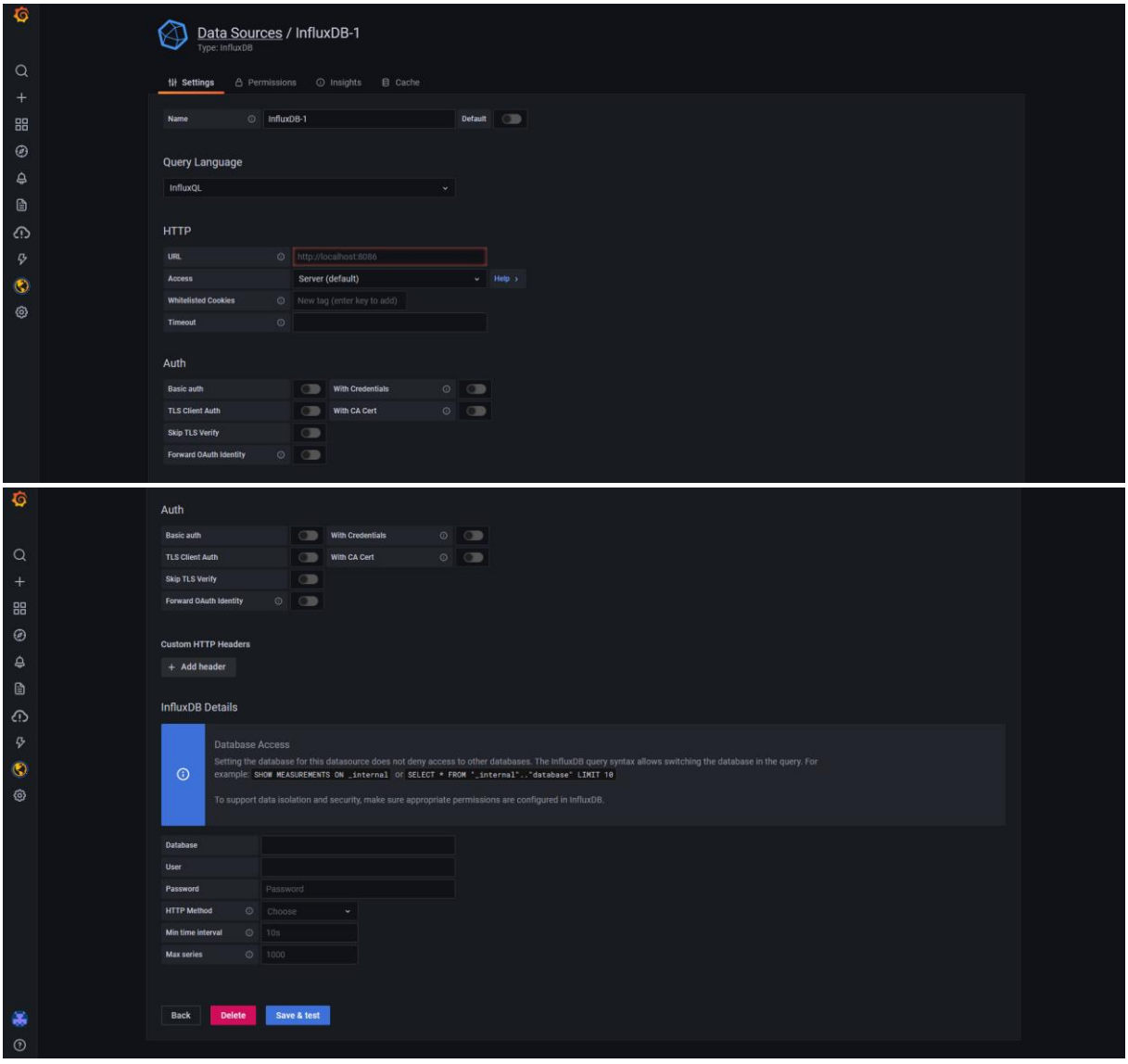


Figure 20: Configuration of the new InfluxDB data source

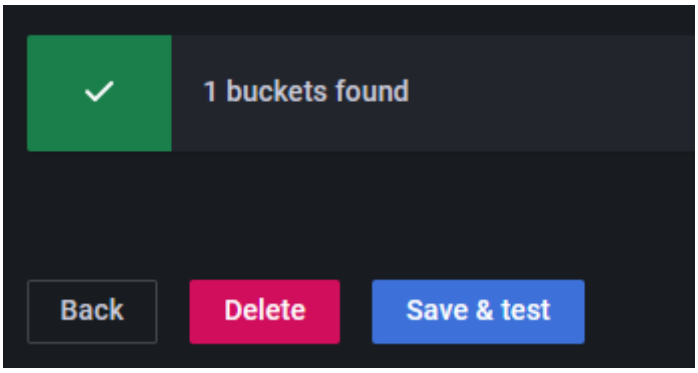
First of all, you should name the connection. Then select the Query Language you want to use. Currently Grafana supports two different languages, InfluxQL and Flux. InfluxQL was used in older versions of InfluxDB and has been replaced by a much more powerful query language Flux. We saw the use of Flux in Exercise 1 and we will use it here too. Select Flux as the Query Language.

Next, we want to set the HTTP connection configurations. Enter the InfluxDB URL we used in previous exercises. Leave the rest of the options as they are.

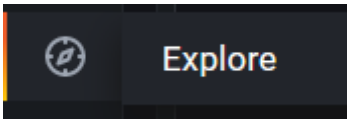
For Authentication, we will use “Basic Auth” and no username or password is needed.

Next, you should enter the InfluxDB Details. Enter the Organization you have been given and the Bucket from which we want to query data. Finally, you have to enter the token we have created.

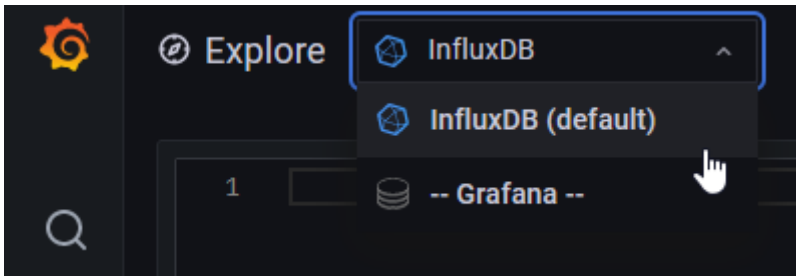
Make sure that you have entered all the necessary information, as described and press **Save & Test**. You should see the following message that confirms all is set right.



Now that we have configured the connection, we can query data stored in InfluxDB from within Graphana. Press the **Explore** menu button.

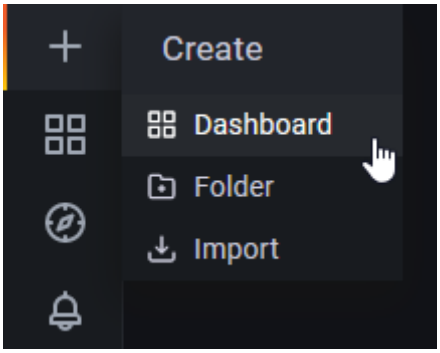


And select InfluxDB as the data source



In the field below, you can now write Flux queries as we have written before. Try it now.

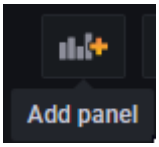
To create an actual dashboard, you should access the **Create**->**Dashboard** menu



You will be presented with a dashboard with a panel, which has three options:

1. Add an empty panel, which visualizes a query
2. Add a new row, which is used for organizing the various panels.
3. Add a panel from the panel library, which is used to import an already saved panel.

By pressing the **Add panel** button, you can add more panels



Add some panels and set some of them to be new rows. Now move the panels around to familiarize yourself with the possible structures you can create.

Next, select an empty panel. You will be presented with a screen like the one in Figure 21, where you can build your query.

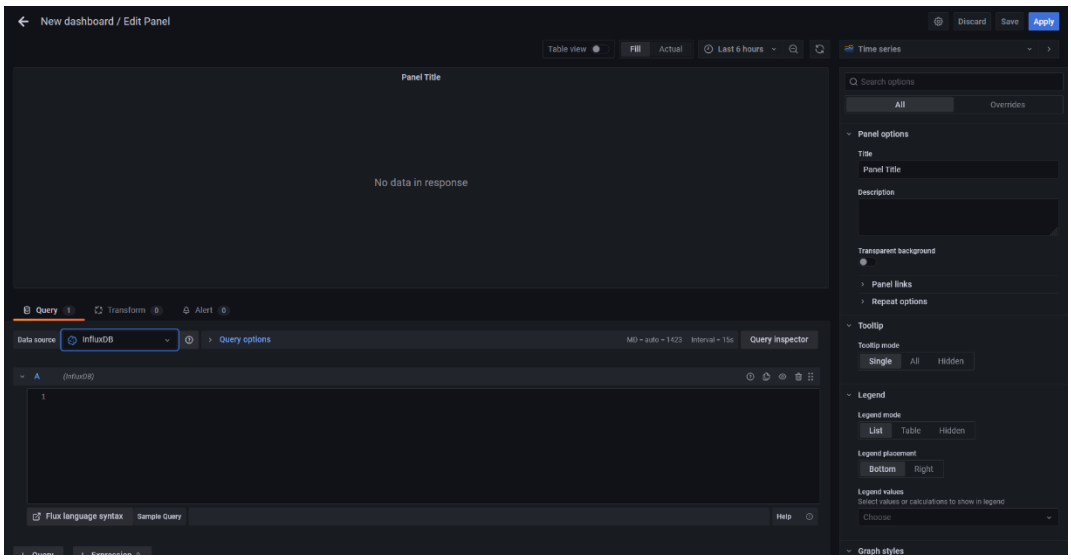


Figure 21: The new panel menu of Grafana

Again, you should select the appropriate Data source and write the query.

On the right side, you can select the visualization options to be used. As you can see, visualization in Grafana is much more flexible than the one in InfluxDB. Both in terms of options and customization.

Along the query, there are two other possible tabs. **Transform** and **Alert**.

In **Transform** (Figure 22) you can select various filters to apply to your data before visualizing. These are like the *Transform Functions* of Flux, however there are additional options that can be utilised.

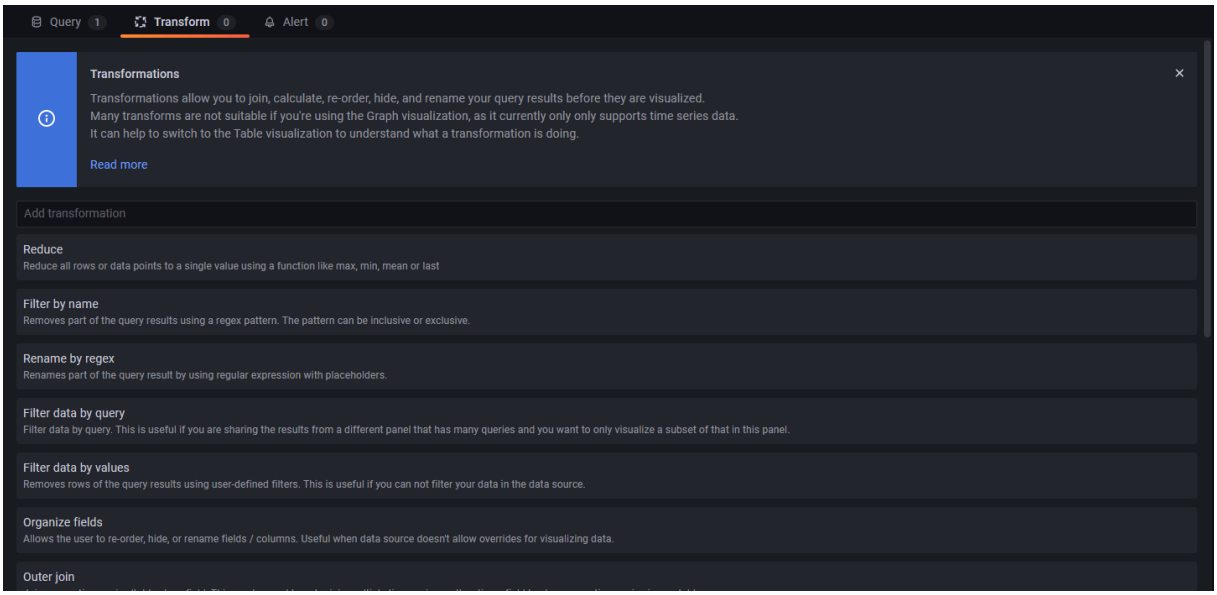


Figure 22: The transform menu

Finally, in the **Alert** tab, you can set alerts to be triggered in case of some event, like the alerts in InfluxDB. However, again the breadth of options is much greater.

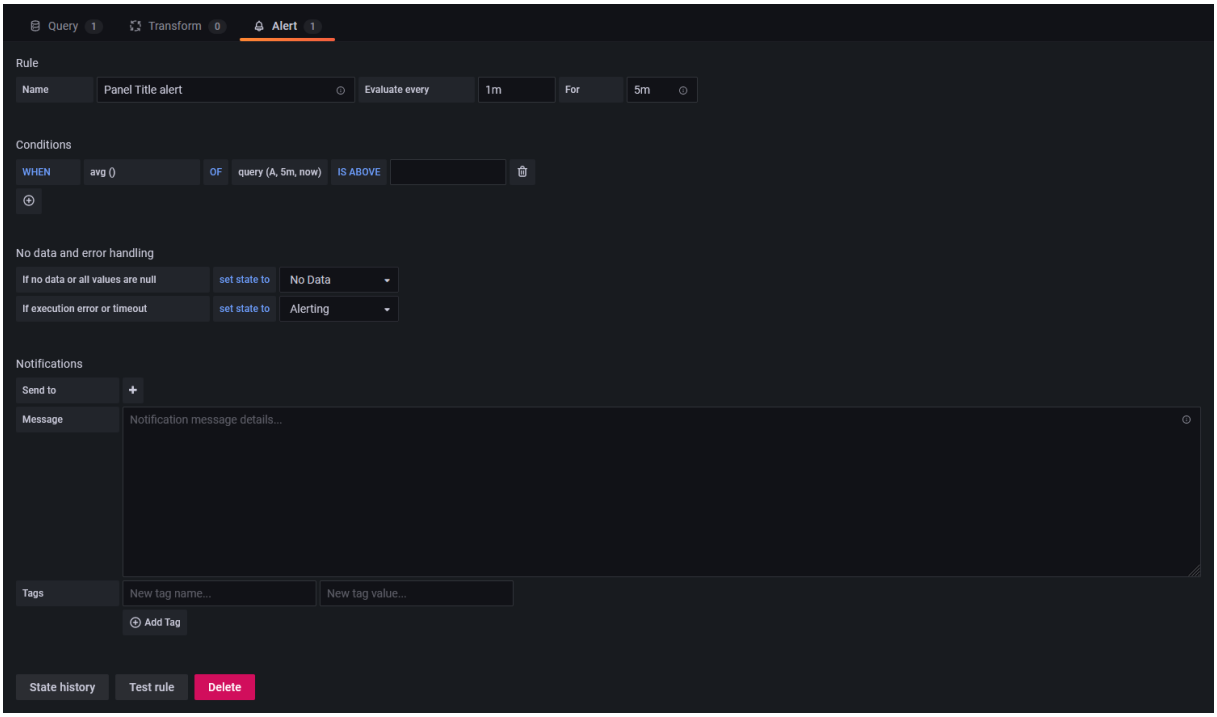


Figure 23: The Alerts menu

Before setting an alert, you should set the notification channels. To do this click the **Alerts**->**Notification Channels** menu.

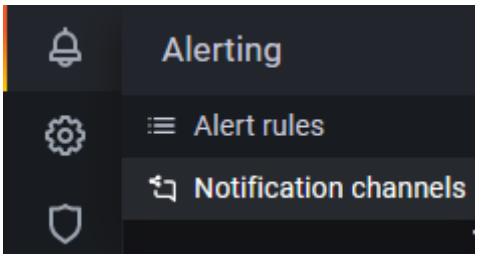


Figure 24: Reaching the Notification channels menu

There you can add different channels. Like Slack, Discord, E-mail, Microsoft Teams, Telegraf, etc. Again, the breadth of options is much greater than in InfluxDB.

One of the notification channels options is Telegraf, a tool we have seen and used before that can write in InfluxDB. You can have Grafana monitor an InfluxDB bucket and write any event that triggers an alarm to another InfluxDB bucket for example.

With the above information, you should be able to build a sample Dashboard to monitor the SensorData Bucket. Do that now as an exercise and configure an alert that will send a notification to your e-mail in case a threshold is exceeded.

6. Exam Questions

1. In which category does the InfluxDB database belong to?
2. Please explain shortly the benefits of NoSQL databases.
3. What information is needed to retrieve data from an InfluxDB database?
4. What information is necessary in order to post and save data to an InfluxDB database?
5. What is a retention policy and how would you configure one in InfluxDB?

References

[1] M. Nasar and M. Abu Kausar, “Suitability Of Influxdb Database For Iot Applications,” *International Journal of Innovative Technology and Exploring Engineering (IJITEE),* vol. 8, no. 10, pp. 1850-1857, 2019.

[2] G. Klyne and C. Newman, “Request for Comments (RFC) 3339: Date and Time on the Internet: Timestamps,” Internet Engineering Task Force (IETF), 2002.

[3] InfluxData Inc., “InfluxDB OSS 2.0 Documentation,” [Online]. Available: https://docs.influxdata.com/influxdb/v2.0/. [Accessed 28 April 2022].

[4] InfluxData Inc., “Sample air sensor data,” [Online]. Available: https://github.com/influxdata/influxdb2-sample-data/tree/master/air-sensor-data. [Accessed 28 April 2022].

7. Contacts

Project Coordinator:

- Name: Technical University of Sofia
- Address:
    - Technical University of Sofia,
      Kliment Ohridsky Bd 8
      1000, Sofia, Bulgaria
- Phone: +3592623073

Output 2 Leader:

- Name: FOSS Research Centre for Sustainable Energy, University of Cyprus
- Address:
    - University of Cyprus,
      Panepistimiou 1 Avenue
      P.O. Box 20537
      1678, Nicosia, Cyprus
- Email: foss@ucy.ac.cy
- Phone: +357 22 894288