Fraunhofer
IIS

FRAUNHOFER INSTITUTE FOR INTEGRATED CIRCUITS IIS

GETTING STARTED WITH MIOTY^{TM}

Tutorial for setting up and operating a MIOTY^{TM} Network

Internal

GETTING STARTED WITH MIOTY^{TM}

Tutorial for setting up and operating a MIOTY^{TM} Network

Fraunhofer Institute for Integrated Circuits IIS, Erlangen

Michael Rüger

Contents

1 Data transfer and data visualization

When working with MIOTY^{TM} the first step is to deploy the sensors which transmit their data over MIOTY^{TM}. The base station can now receive and decode the transmitted signals as well as store the sensor data for a limited amount of time. The data must go through several steps before it can be displayed using an IoT platform. The complete processing chain is shown in Figure 1. To access the data from the base station, the base station runs a MQTT broker, which publishes the data to subscribed devices. To simplify the data extraction and conversion to the right format, a tool called NodeRed is used. It is well suited for data flow modification and offers several interfaces to retrieve or store data. Once the sensor data has been converted into the needed representation, it must be assigned to the correct sensor node and subsequently stored inside a database. Thingsboard and Grafana are two great and powerful tools for data visualization and offer great ways of working with IoT data. Thingsboard uses a built-in database to store its data. Data can again be published using the MQTT protocol. On the other hand, Grafana is only responsible for the visualization of the data and therefore requires an external database to store the sensor data. For that the time-series database InfluxDB is used because it offers great advantages in speed as well as an easy to operate web-interface.

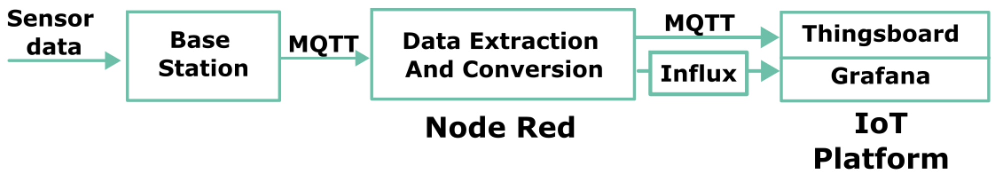


Figure 1 MIOTY Processing Chain

To deploy these multiple different applications and have them work with each other is not a simple task. The use of Docker is a great way to simplify this process and offer good support for different types of hardware.

In this tutorial it is shown how to setup an IoT platform for storing and displaying sensor data with the help of Docker. Multiple approaches with either Thingsboard or Grafana offer great alternatives for different use cases. At the end new sensor data is automatically retrieved from the base station, stored inside a database and can be visualized with the different platforms.

1.1 Introduction to Docker

Docker can be a very powerful tool, when working with different applications on various types of hardware. The platform enables the user to quickly and easily deploy programs. The programs are encapsulated in so called images. Once an image gets deployed, it is called a container and is run in an isolated environment. Multiple of those containers can be deployed on one host system. Since a container contains all necessary files for execution it does not rely on dependencies on the host system. This means that Docker containers can be run on any operating system or hardware configuration, which makes it very versatile and consistent in deployment. Communication between containers over shared virtual networks is still possible and enables the integration of multiple programs. Ready to deploy images can be downloaded on a central database, called DockerHub.

This makes Docker very well suited for self-hosting an IoT-Platform like Thingsboard or Grafana in combination with a time-series database InfluxDB. To achieve the correct data representation, the data from the base station has to be converted into the right data format. One possibility to achieve this step is with the help of Node-Red.

1.1.1
Docker Compose

If an application requires multiple containers to function it can be difficult to keep an overview of all the different ports, networks and dependencies. The tool Docker Compose simplifies this process by combining everything into one file called *docker-compose.yml*. An exemplary file is given below. It starts a service called *mytb* with the Thingsboard image from the Docker Hub. Important for the correct functionality is the port mapping. The container requires specific ports to be mapped for communication. For instance, the port 1883 is internally used for the MQTT protocol. MQTT messages sent to port 1884 of the host system, then get forwarded to port 1883 inside the container. Environment variables can be used to set specific functions inside the container. To preserve data generated by the container it can be useful to create volumes and map them to the file system of the container. When the container gets deleted or recreated the data is still protected.

```yaml
version: '3.0'  # Specifies the Docker Compose version
services:       # All containers are defined under this section
  mytb:         # Name of the first service
    restart: always # Restart policy
    image: "thingsboard/tb-postgres"  # Image of the container (local or from the Docker Hub)
    container_name: Thingsboard       # Name of the container
    ports:
      - "8080:9090"         # Port mappings  xxxx:yyyy ->
      - "1884:1883"         # Port xxxx on Host System gets mapped to Port yyyy of the container
      - "7070:7070"
      - "5683-5688:5683-5688/udp"
    environment:
      TB_QUEUE_TYPE: in-memory  # Setting environment variables of the container
    volumes:
      - ~/.mytb-data:/data                   # Maps Volumes to the container ->
      - ~/.mytb-logs:/var/log/thingsboard     # Source_on_Host_System:Destination_in_container
```

This collection of services is often referred to as a stack and can be started with the command inside the directory of the docker-compose.yml file:

```
docker compose up –d
```

Required images are automatically downloaded and all containers are started. To stop a running stack simply run the command

```
docker compose stop
```

Further information and documentation regarding Docker can be found on the Docker website.

1.2
Introduction to Node-Red, InfluxDB, Grafana and Thingsboard

To display and work with the data sent by the MIOTY endpoints several deployment options are conceivable. The data is firstly received by the base station and subsequently made available by a MQTT broker running on the base station. This data has to be formatted in a specific way and then forwarded to the IoT platform. This task can be done relatively simple with the help of Node-Red.

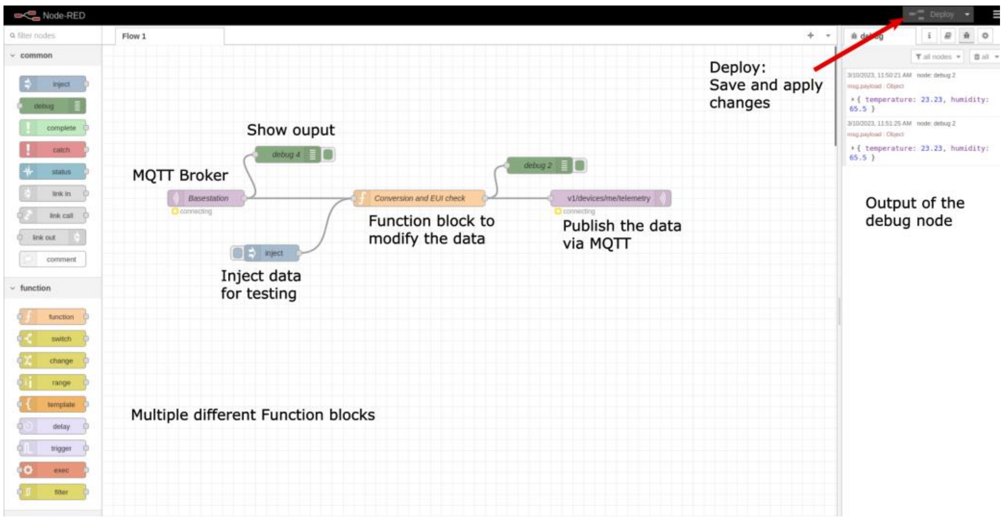
Multiple deployment options with different advantages and disadvantages are considered in this guide.

1) <u>Thingsboard as a cloud service</u>
   - Easy and fast to deploy, constant availability
   - Subscription fee
2) <u>Thingsboard as a local service in Docker</u>
   - Free, same functionality as cloud service
   - More complicated to setup, additional steps for availability outside the local network
3) <u>Grafana as a local service in Docker</u>
   - Free, very good data visualization options, fast
   - Requires additional database container, additional steps for availability outside the local network

Dependent on which option is suited the best, there are different containers required. The following section gives a short overview about the different programs and their functionality. Implementation details regarding the different options can be found in chapter 2

1.2.1
Node-Red

Node-Red is a graphical development tool that allows the user to manage data flow. It is built as a modular system with many different function nodes. These nodes can be connected with each other to exchange messages and modify the data. There are also many different input and output nodes for exchanging data with many different interfaces. For instance setting up a MQTT broker is as simple as adding a node and specifying the address and the subscribed topic. In addition to the basic functions it is also possible to install several additional nodes with extended functionalities.



```
3/10/2023, 11:50:21 AM  node: debug 2
msg.payload : Object
▸{ temperature: 23.23, humidity:
65.5 }
3/10/2023, 11:51:25 AM  node: debug 2
msg.payload : Object
▸{ temperature: 23.23, humidity:
65.5 }
```

Figure 2 Node-Red example flow

In Figure 2 a minimalistic flow shows how the processing of data can be done. The data either comes in form the base station broker that is subscribed to relevant topics or from an inject node, that can simulate the data arriving for testing purposes. The function node contains JavaScript code that extracts the relevant information from the incoming message and returns the data in the required format. The output of the node can be seen on the right side. The egress node publishes the data using the MQTT protocol to the Thingsboard server. It is important to deploy any changes to the running version via the button in the top right corner. Further information about Node-Red can be found on the homepage.

1.2.2 InfluxDB

Data storage is an important aspect for countless applications. For structured data, as it is mostly the case for IoT applications, databases are well suited option of storing large amounts of data. InfluxDB is a database that is especially suited for time series data (data that is associated with a timestamp, like a sensor reading). It allows fast read and write times for queries regarding a time interval. This can be useful during the data visualization. In most cases IoT devices collect sensor readings in regular intervals, which very quickly leads to a large amount of data for each sensor. For many applications the data is only needed with that high precision for a limited time. InfluxDB allows automatic data combining strategies to reduce the data size. Accessing the database is made easy by using API tokens.

1.2.2.1 Data organization

InfluxDB uses a hierarchical data storage model. On the top level the data is organized in buckets. One bucket can have multiple measurements and these measurements can again have multiple tags and fields.

- **Bucket**: Named location where time series data is stored. A bucket can contain multiple measurements.
    - **Measurement**: Logical grouping for time series data. All points in a given measurement should have the same tags. A measurement contains multiple tags and fields.
        - **Tags**: Key-value pairs with values that differ, but do not change often. Tags are meant for storing metadata for each point – for example, something to identify the source of the data like host, location, station, etc.
        - **Fields**: Key-value pairs with values that change over time – for example: temperature, pressure, stock price, etc.
        - **Timestamp**: Timestamp associated with the data. When stored on disk and queried, all data is ordered by time[1].

The web-interface of InfluxDB offers a great visualization of this hierarchy. Figure 3 InfluxDB shows a configuration of storing the temperature values from a sensor. From the bucket *mioty* there is one measurement available called *temperature_sensor*. The temperature values are stored in the *temperature* field. In the case that this sensor would also provide humidity readings, these values are shown as an additional entry in the field section. The collected data can also directly be viewed in the web-interface as a graphical chart as well as only the raw data.

[1] Quelle: https://docs.influxdata.com/influxdb/v2.6/get-started/

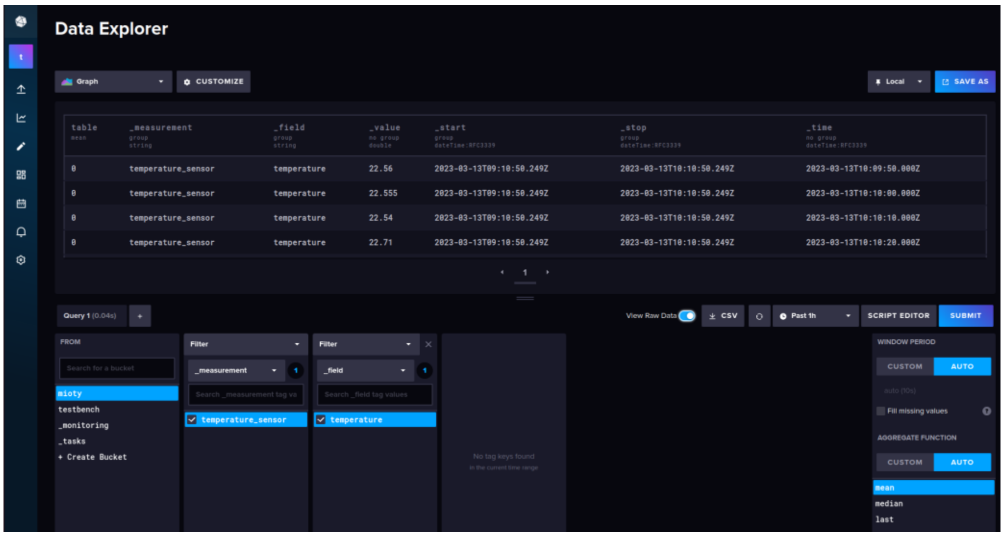


table	_measurement	_field	_value	_start	_stop	_time
mean	group string	group string	no group double	group dateTime:RFC3339	group dateTime:RFC3339	no group dateTime:RFC3339
0	temperature_sensor	temperature	22.56	2023-03-13T09:10:50.249Z	2023-03-13T10:10:50.249Z	2023-03-13T10:09:50.000Z
0	temperature_sensor	temperature	22.555	2023-03-13T09:10:50.249Z	2023-03-13T10:10:50.249Z	2023-03-13T10:10:00.000Z
0	temperature_sensor	temperature	22.54	2023-03-13T09:10:50.249Z	2023-03-13T10:10:50.249Z	2023-03-13T10:10:10.000Z
0	temperature_sensor	temperature	22.71	2023-03-13T09:10:50.249Z	2023-03-13T10:10:50.249Z	2023-03-13T10:10:20.000Z

Figure 3 InfluxDB Web interface

InfluxDB is a great solution to store data collected from IoT devices. Using the database is made easy through integration in Node-Red and Grafana. It can be easily deployed via Docker and offers a beginner friendly, graphical web-interface for interacting with the database. In combination with Grafana it is a very powerful tool in visualizing and working with IoT data.

1.2.3
Grafana

The first version of Grafana was released in 2014 and has now become a well-established software for the visualization of data. It allows easy integration of a variety of different data sources as well as a huge possibility of customizing the visualization. To get familiar with the software Grafana offers an online version with many different example dashboards and data sources to play around with. The website can be found here. Figure 4 is taken from this website and shows an exemplary dashboard for the visualization of the data.

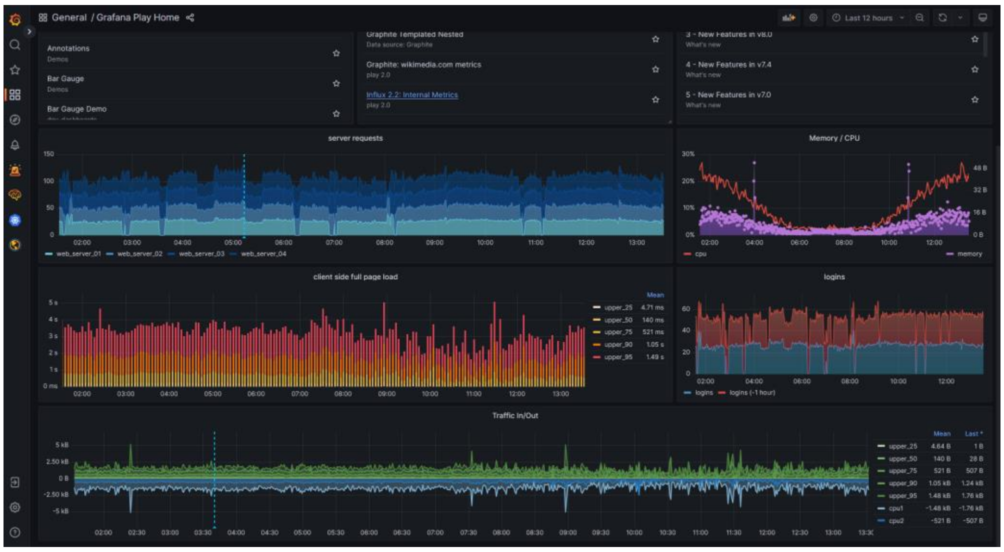


Figure 4 Grafana Example Dashboard (https://play.grafana.org)

What makes Grafana and InfluxDB interact well together is the time aspect of the data. The desired timespan can be selected in the top right corner and InfluxDB can make a fast query to its database.

2 Installation and Setup of the containers

A very easy solution to deploy and manage an IoT platform is via Docker. Docker simplifies many steps, is well suited for different hardware and is very easy and fast to deploy. To further simplify the container management, it is recommended to work with Portainer. Furthermore, it has to be decided which deployment option referred to in chapter 1.2 is suited the best for the application.

2.1 Docker Installation

A positive aspect is that Docker runs on a variety of different hardware and operating systems. Docker provides a desktop version for all major operating systems (Windows, macOS and Linux), simply called Docker Desktop. It comes with all the necessary tools needed to build and run containers and stacks. To run a container a virtual environment is created where it is deployed. The installation steps and requirements for Windows are found here. The installation guides and files for other operating systems can be selected on the left. It is only recommended to install Docker Desktop on Windows and macOS based systems, since Docker Desktop creates a virtual machine to run the Docker daemon which can run natively on Linux.

On Linux based systems there exists another possibility to install Docker without using the Docker Desktop app. The Docker Engine is available for many different distributions of Linux and offers the same functionality without the graphical interface and without the need of a virtual machine in the background to run the Docker daemon. If you are planning to run Docker on Linux this is the preferred installation method. Detailed steps for the installation for Ubuntu bases systems can be found here. To be able to run the Docker-command as a non-root user, follow the steps given here.

Once the installation is successfully completed open a new terminal and verify it by running:

```
docker run hello-world
```

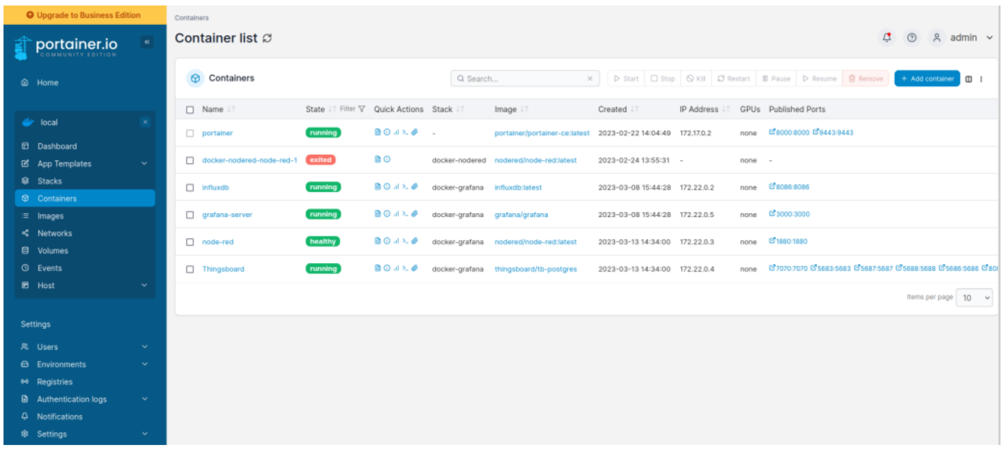
If everything is working as intended this will print out a message confirming that the installation was successful.

2.2 Portainer - managing containers

Docker is a program that is mainly used from the command line. Managing containers without a graphical interface can be quite challenging especially for beginners. A great tool for working with containers using an easy to use and intuitive interface is Portainer. It offers a web-interface where new containers can be created, deployed or stopped. It is also possible to check log files or open a command prompt inside a running container. It is not a necessity for the further steps in this tutorial, but is still recommended for easier usability.

The easiest way of running Portainer is inside a Docker container. The community edition is free to use and easy to setup. Simply follow the steps for your operating system given on the Portainer website. After a short startup time of the container the web-interface of Portainer should be accessible in the browser.

When logged in, chose the local environment. A navigation bar on the left with all the relevant topics for managing containers, like networks or volumes will appear. Under the container section you should see at least see one running container – Portainer. Created containers can be easily started, stopped or deleted. Under the "Quick Actions" section log files can be inspected or command prompt inside the container can be opened. Figure 5 shows the Portainer web-interface under the container section.



Name	State	Quick Actions	Stack	Image	Created	IP Address	GPUs	Published Ports
portainer	running		-	portainer/portainer-ce:latest	2023-02-22 14:04:49	172.17.0.2	none	8000:8000 9443:9443
docker-nodered-node-red-1	exited		docker-nodered	nodered/node-red:latest	2023-02-24 13:55:31	-	none	-
influxdb	running		docker-grafana	influxdb:latest	2023-03-08 15:44:28	172.22.0.2	none	8086:8086
grafana-server	running		docker-grafana	grafana/grafana	2023-03-08 15:44:28	172.22.0.5	none	3000:3000
node-red	healthy		docker-grafana	nodered/node-red:latest	2023-03-13 14:34:00	172.22.0.3	none	1880:1880
Thingsboard	running		docker-grafana	thingsboard/tb-postgres	2023-03-13 14:34:00	172.22.0.4	none	7070:7070 5683:5683 5687:5687 5688:5688 5686:5686 80

Figure 5 Portainer Dashboard

All currently downloaded images can be inspected under the *Images* section as well as their size. To enable communication between containers, networks must be defined. Volumes allows the storage of data independent of the container. This is useful to persist data beyond the life of a single container.

2.3 Starting the containers stacks

To process and visualize the data from the IoT sensors, this tutorial describes two different ways to achieve this. The visualization is either done with Thingsboard or Grafana. In both cases, additional software is required besides the actual program. The tool Docker Compose allows several containers to be combined into a single stack and thus significantly simplifies the deployment. The configuration files contain everything that is needed to start and enable the communication between the different containers. With that the service can be online within a few minutes and only minor steps inside the programs have to be made to visualize the first IoT data.

There exist different options to deploy a stack, either with the use of the command line or directly in Portainer under the *stacks* section. In general these two ways do not differ much but if the Portainer option is chosen, the stack can once again be stopped or deleted in the web interface of Portainer. Once deployed all the containers of the stack can be monitored via Portainer and usually don't require any further setup steps.

2.3.1 Docker-Compose in the terminal

First create a new folder in your preferred directory and subsequently create a file called *docker-compose.yml*. This file defines all the containers, called services, which have to be started as well as volumes to persist the data of the container beyond its life. Depending on your preferred option of IoT platform, choose the correct configuration and paste them into the *docker-compose.yml* file. The different file contents can be found under section 5.1.1 – 5.1.3.

To start the stack only a single command is needed:

```
docker-compose up -d
```

Firstly, the correctness of the configuration file is verified and then all the required images are automatically downloaded and the containers are started. The flag *–d* specifies, that the stack should be executed in the background (detached mode).

To display all created and running containers use:

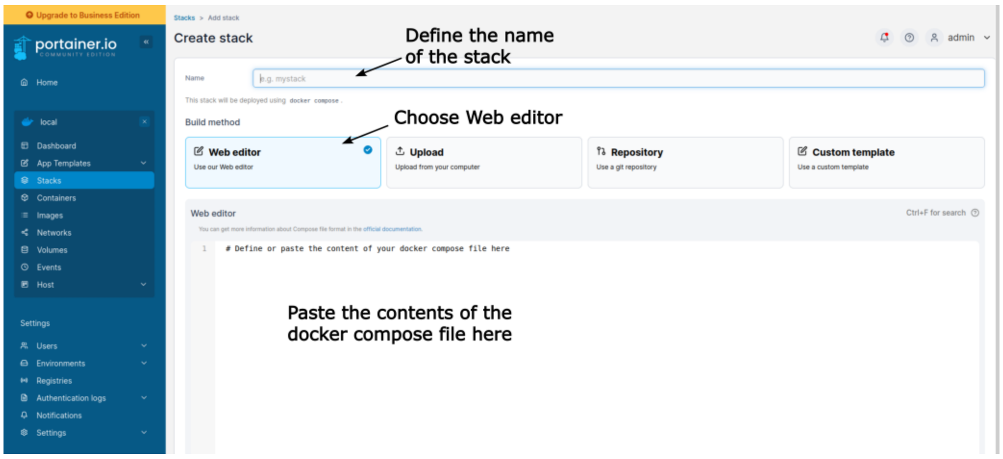
```
docker ps
```

To stop the stack simply use the command

```
docker-compose stop
```

2.3.2 Docker-Compose in Portainer

Creating stacks in Portainer is very easy and fast through the use of the graphical interface. Under the *stack* section click on *add stack* in the top right to add a new stack. Now the stack can be given a name and the configuration can be added. Depending on your preferred option of IoT platform (Grafana or Thingsboard), choose the correct configuration and paste them into the field. The different file contents can be found under section 5.1.1 – 5.1.3. Figure 6 shows the procedure of adding a stack in Portainer.



This stack will be deployed using docker compose.

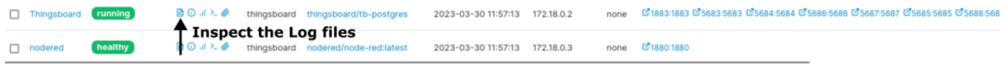
You can get more information about Compose file format in the official documentation.

```
# Define or paste the content of your docker compose file here
```

Figure 6 Adding a stack in Portainer

To start the stack, scroll to the bottom of the page and click deploy. After that all missing images are automatically downloaded and the containers are started.

Under the *Stacks* section should now be one stack listed. To verify that all containers are running correctly, they can be inspected in the *Container* section. After all containers are fully started, they should either have the status active or healthy. If any errors occur, verify the contents of the Docker-compose file. Also, the containers' log-files can give further details on the problem. To inspect the contents, simply click on the corresponding icon in the list of containers.



☐	Thingsboard	running	thingsboard	thingsboard/tb-postgres	2023-03-30 11:57:13	172.18.0.2	none	1883:1883 5683:5683 5684:5684 5686:5686 5687:5687 5685:5685 5688:568
☐	nodered	healthy	thingsboard	nodered/node-red:latest	2023-03-30 11:57:13	172.18.0.3	none	1880:1880

Figure 7 Portainer logfile inspection

3 Visualization via Thingsboard

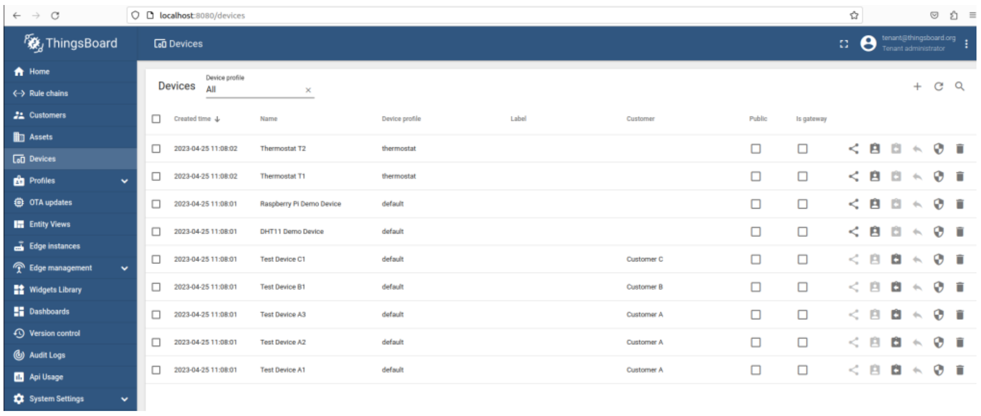
Thingsboard is an open-source IoT Platform, that is designed for data visualization via the use of widgets. To store the recorded data, it has to be sent via the MQTT protocol to a broker running on the Thingsboard instance. The rest, like data storage and visualization, is done by Thingsboard. Depending on the requirements a deployment option as a local installation of Thingsboard or a cloud solution is feasible. The cloud solution offers easier remote access but is also tied to a subscription fee. Details about the different cloud options and pricing can be found here.

With Thingsboard being deployed in the cloud, only a local instance of NodeRed is required to receive the data from the base station, convert it into the right data format and forward it via MQTT to Thingsboard. For a local installation additionally the Thingsboard container has to be deployed**. Refer to chapter 5.1.1 and 5.1.2 for the required Docker-compose files and deploy the stack as shown in chapter 2.3.1 or 2.3.2.**

3.1 Thingsboard Web Interface

After a short startup time the container's web interface can be accessed from a browser on port 8080 on the host machine (URL: *http://localhost:8080*). After in using the credentials ***tenant@thingsboard.org*** as the username and ***tenant*** as a password, all features can be accessed.

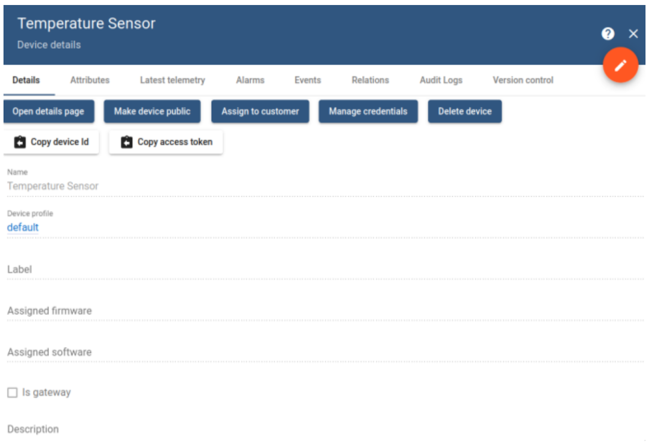
For a first and simple setup only the *Devices* and *Dashboard* section is needed, to first add the device and then displaying its data using for instance a time series graph. In Figure 8 the overlay of Thingsboard is shown.



Created time ↓	Name	Device profile	Label	Customer	Public	Is gateway
2023-04-25 11:08:02	Thermostat T2	thermostat			☐	☐
2023-04-25 11:08:02	Thermostat T1	thermostat			☐	☐
2023-04-25 11:08:01	Raspberry Pi Demo Device	default			☐	☐
2023-04-25 11:08:01	DHT11 Demo Device	default			☐	☐
2023-04-25 11:08:01	Test Device C1	default		Customer C	☐	☐
2023-04-25 11:08:01	Test Device B1	default		Customer B	☐	☐
2023-04-25 11:08:01	Test Device A3	default		Customer A	☐	☐
2023-04-25 11:08:01	Test Device A2	default		Customer A	☐	☐
2023-04-25 11:08:01	Test Device A1	default		Customer A	☐	☐

Figure 8 Thingsboard Web Interface

To add a new device, click the plus sign in the top right corner under the *Devices* tab and provide a name and an optional description. Once added an access token is automatically generated. It is used to authenticate the user and to update the database with new sensor readings. It is required in section 3.2 when setting up the data conversion and publishing of the data.



Name
Temperature Sensor

Device profile
default

Label

Assigned firmware

Assigned software

☐ Is gateway

Description

Figure 9 Thingsbaord Device Configuration

Figure 9 shows the configuration settings of a device. Once sensor data is published, the newest values can be inspected in the *Lastest telemetry* tab. The access token can also be found here.

Once all desired devices are added, Thingsboard is now setup to receive sensor data via the MQTT interface. These messages can be easily sent using Node-Red.

3.2 Managing Data Flow with Node-Red for Thingsboard

Since Thingsboard only accepts a specific data format to store the data, conversion is needed. The base station provides a JSON string containing the data of all sensors as well as addition information about the base station. Each sensor node is identified by a unique identifier called EUI. The goal is to find the correct data path for a received EUI from the base station and extract the relevant data from the string. Thingsboard requires the data to be transmitted as a JSON string, containing at least one key and value pair. An example of a possible message to the Thingsboard server is:

```
{
   "tempeature":22.43,
   "humidity":52,
   "pressure":1025
}
```

One pair consists of a key ("temperature") and a corresponding value (22.43). The unit of the measurement is not transmitted and must be specified later during the visualization process. To be able to publish this data, an access token must be provided as authorization during this procedure. This token is generated once per IoT device in the Thingsboard overlay and connects the EUI with the registered device in Thingsboard. With that each device can be uniquely identified.

Node-Red is running inside a container and can be accessed on port 1880 (URL: http://localhost:1880). With the import feature it is easy to deploy a preconfigured flow, where all the important functions are already defined. The configuration is provided in a JSON string (see section 5.1.4) and can be easily **imported using the feature found inside the menu in the top right**.

The flow consist of three different tasks (Figure 10) :

1. The reception of data from the base station
2. The data conversion and validation of the correct EUI
3. The publishing to a Thingsboard device using an access token

Different nodes each provide different functionalities:

• Purple Node (MQTT in/out):	MQTT Broker or Client
• Blue Node (Inject node):	Used to manually inject data into the flow
• Green Node (Debug node):	Used to display messages of the flow
• Ocher Node (Function node):	Used to implement own functions in JavaScript

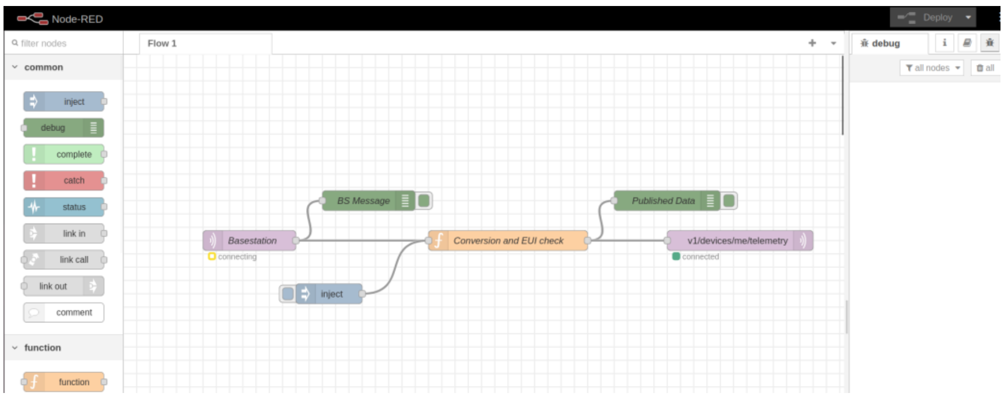


Figure 10 Node-Red Processing Flow

The Base station node represents a MQTT broker, which is subscribed to the topic *mioty/+/+/uplink*. This includes the newest uplink messages from each sensor node. This means that for each individual sensor node with a unique EUI a message is processed by the flow. Once new data is available from any sensor it is automatically received and processed by the flow. To connect to the base station, its **IP address must be entered** in the configuration. To open the configuration **double click the node** and **click on the pencil** next to the server to change the setup of the broker. Enter the correct IP address in the server field. It is the same IP address that is needed to access the web interface of the base station. The port is likely not changed and should stay 1883. To check if the connection can be established, click deploy in the top right corner and wait for a green box next to the node. Otherwise check the IP address again and make sure the base station and the PC are in the same network. Don't forget to deploy each change.

The next step is to extract the desired sensor data from the message coming from the base station and returning a JSON string in the format described above. Additionally the EUI of the message has to match the one specified by the function. This assures, that each message gets forwarded and stored only once and gets connected with the correct device in Thingsboard. The flow is preconfigured for a single device. If more devices are added, simply copy and paste (*Ctrl+C* and *Ctrl+V*) the conversion and publishing node. Refer to Figure 11 for the layout.

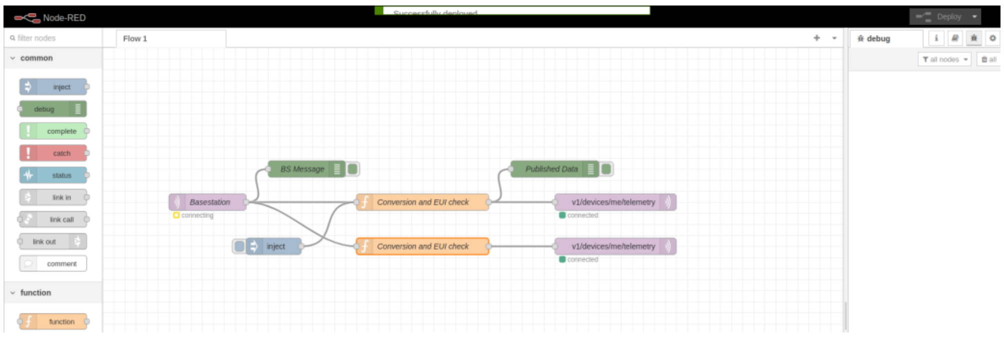
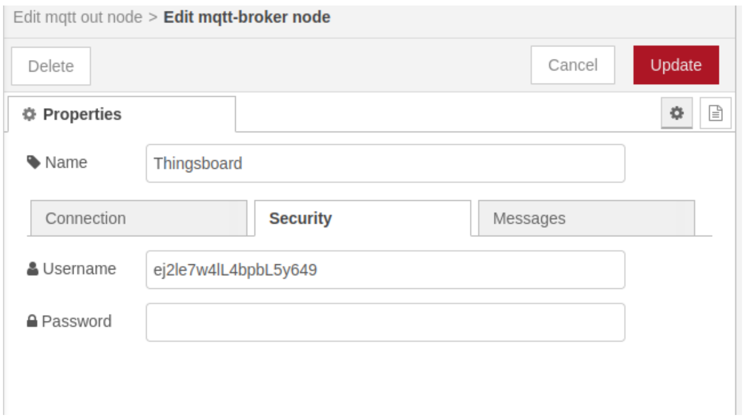


Figure 11 Node-Red multiple Sensor Nodes

For each sensor node one function block should exist and each node should be configured to the EUI of each sensor. **Double click on the node** to open its implementation and **enter the correct EUI** as a **decimal number** in the variable. If the EUI of the message complies with the specified value, this function returns a JSON string containing all the endpoint data to be published to Thingsboard. If no conversion block exists for a given EUI the message is ignored.

As the last step the publishing node has to be configured to store the data for the correct device in Thingsboard. For each EUI in a function block **determine to what device in Thingsboard the data should be sent and copy its access token** from Thingsboard. This token is used to uniquely identify the device and authenticate the access to the data. Again **double click on the outgoing node** and **press the pencil** next to the server field to edit the configuration. Docker brings the advantage that the IP address of the Thingsboard instance does not need to be specifically entered. It can just be referenced by the name and resolved in the background. In the **security tab paste the access token from the Thingsboard device into the username field** (Figure 12).


Edit mqtt out node > **Edit mqtt-broker node**

Delete | Cancel | Update

Properties

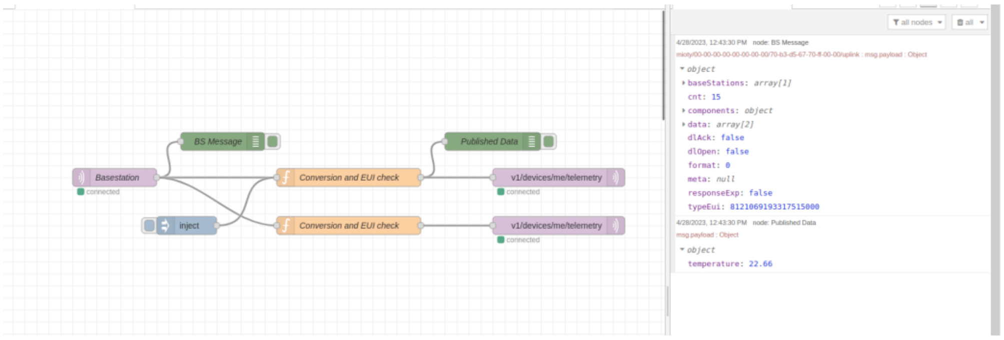
Name	Thingsboard

Connection | **Security** | Messages

Username	ej2le7w4lL4bpbL5y649
Password	

Figure 12 Node-Red Thingsboard Authentication

This links the EUI of the sensor to a device in Thingsboard. Once new sensor data arrives, it gets automatically forwarded and stored in Thingsboard. In Figure 13 shows the data arriving from the base station and the processed JSON string that gets forwarded to Thingsboard inside the debug window.



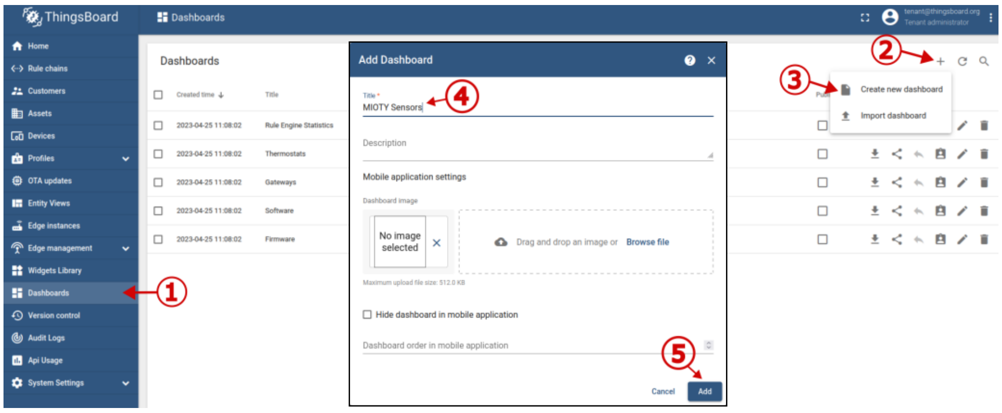
```
4/28/2023, 12:43:30 PM   node: BS Message
mioty/00-00-00-00-00-00-00-00/70-b3-d5-67-70-ff-00-00/uplink : msg.payload : Object
▾ object
  ▸ baseStations: array[1]
    cnt: 15
  ▸ components: object
  ▸ data: array[2]
    dlAck: false
    dlOpen: false
    format: 0
    meta: null
    responseExp: false
    typeEui: 8121069193317515000
4/28/2023, 12:43:30 PM   node: Published Data
msg.payload : Object
▾ object
    temperature: 22.66
```

Figure 13 Node-Red Data Processing

Inside the Latest Telemetry section in Thingsboard the newest data should be displayed.

3.3 Building Dashboards in Thingsboard

If the data flow is setup correctly and the sensor is sending data in regular intervals. Its data can be displayed in Thingsboard with the help of dashboards. Dashboards offer a wide variety of different data representation possibilities to present the data in an appealing way.
To create a new dashboard follow the steps shown in Figure 14. In the Dashboard section select the plus icon, create a new dashboard and give it a suiting name.



Created time	Title
2023-04-25 11:08:02	Rule Engine Statistics
2023-04-25 11:08:02	Thermostats
2023-04-25 11:08:02	Gateways
2023-04-25 11:08:02	Software
2023-04-25 11:08:02	Firmware

Add Dashboard
Title *: MIOTY Sensors
Description
Mobile application settings
Dashboard image: No image selected — Drag and drop an image or Browse file
Maximum upload file size: 512.0 KB
☐ Hide dashboard in mobile application
Dashboard order in mobile application

Figure 14 Thingsboard Dashboard Creation

To add widgets to the newly created dashboard, enter the edit mode by clicking on the *pencil* in the bottom right corner and then *add widget*. There can be found a big collection of different types of widgets that can be added. As a starting point a time series graph is a good possibility of representing temperature data. It can be found under the chart section.
To specify what data should be displayed a new data source must be added first. Select *Entity* as the type and then click in the *Entry alias* * field and create a new alias for your desired device. For instance your device could be similar to the configuration shown below. Make sure to select *Device* as a type. These steps are also shown in Figure 15.

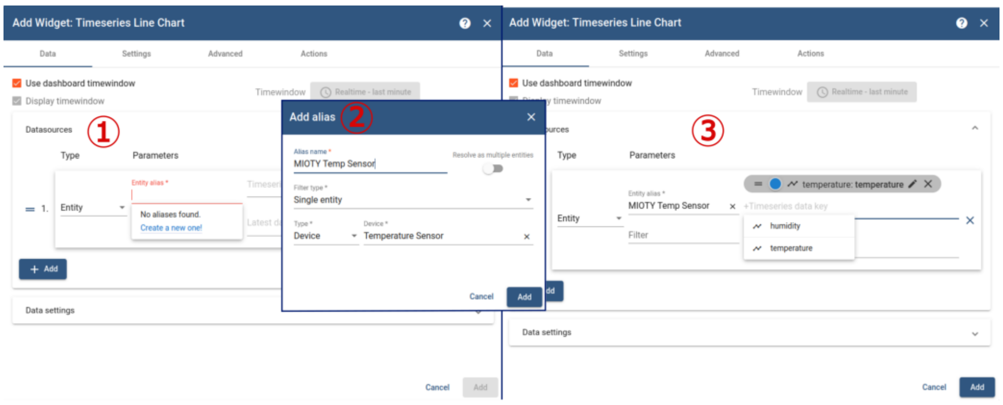


Figure 15 Thingsboard Adding A Datasource

If the data source is configured correctly and there are already enough measurement points, there should be a graph visible. The time interval can also be adjusted in the top right corner (Figure 16).

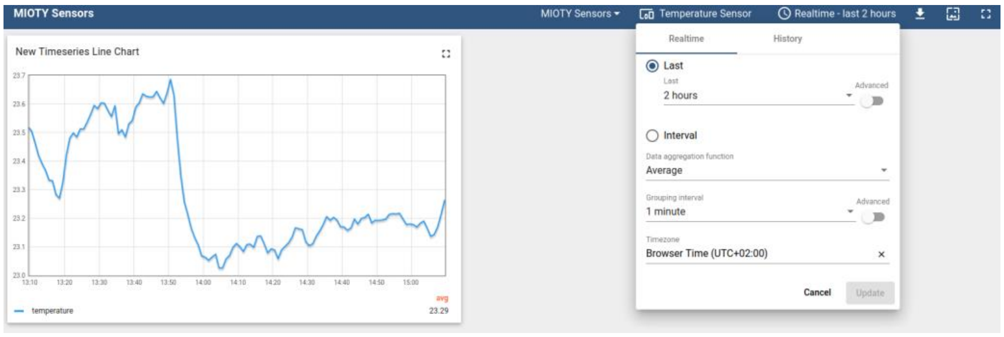


Figure 16 Thingsboard Dashboard

New widgets can now easily be added as well as new data sources for more sensor data. Feel free to try out different things to find a suiting representation.

4
Visualization via Grafana

Another data visualization software that is commonly used for IoT applications is Grafana. It offers great customizability and easy integration of data sources. It is used in combination with Node-Red to process and forward the sensor data provided by the base station and InfluxDB to store the key value pairs.

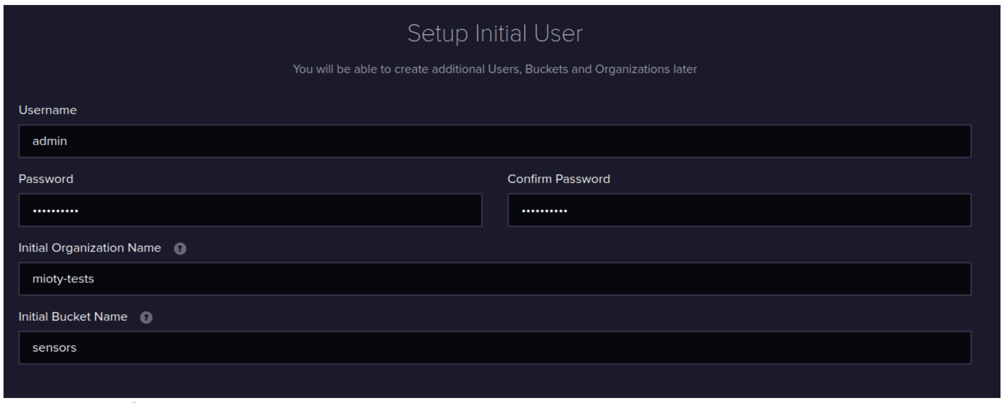
To start all the required containers refer to chapter 2.3 on the options on how to start a stack. The required configuration, which contains all the different containers and port mappings is given in section 5.1.3. Once successfully started, three containers should be running: Node-Red, InfluxDB and Grafana. These containers also offer web interfaces for configuration and controlling of the data flow and can be directly accessed inside the browser under the following links:

- Node-Red: http://localhost:1880
- InfluxDB: http://localhost:8086
- Grafana: http://localhost:3000

4.1
Storing data with InfluxDB

First of it is necessary to setup the data storage with the help of InfluxDB. The database is specifically optimized for storing and accessing data associated with a timestamp. A short overview about how the data storage is organized can be found in section 1.2.2.

First of open the web interface on port 8086 and create a new user as well as an organization. Since most of the time only one person works with this data, it is not relevant for later. Also a bucket has to be created to later store the sensor values. See Figure 17 for reference.


Setup Initial User

You will be able to create additional Users, Buckets and Organizations later

Username: admin

Password: ••••••••••

Confirm Password: ••••••••••

Initial Organization Name: mioty-tests

Initial Bucket Name: sensors

Figure 17 InfluxDB User Creation

To access the database an API token is used. It is generated once and is used to authenticate the user for publishing new data to the database as well as making a query to retrieve data. The token is only displayed once during creation and must therefore be saved for later usage.
To create a token, navigate to the *API Tokens* section under the third category on the left side. Select **All Access Token** to allow access to all data and buckets. A custom

token with limits to specific buckets is also possible. Enter a **description** and then **click safe**. **Store this token** in a safe place for later usage.

To store the data a bucket has to be setup. A bucket can contain multiple measurements, which refer to a logical grouping of time series data, for example one specific sensor. So one bucket can store the data of multiple sensors. Verify that you have created a bucket where the data should be stored.

Now everything is setup to be able to store and retrieve data from InfluxDB.

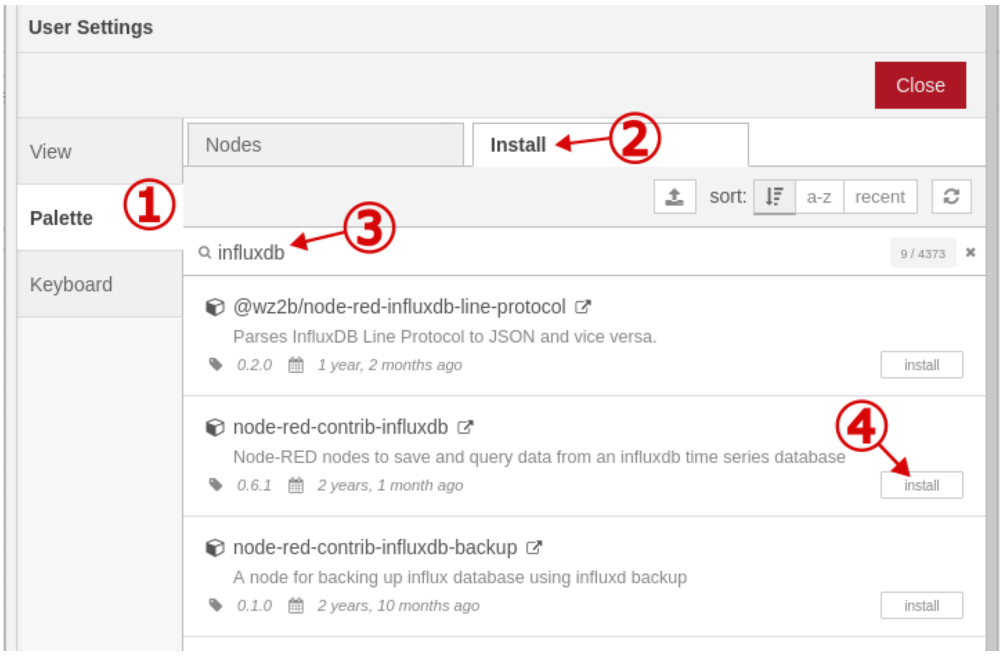
4.2
Managing Data Flow with Node-Red for Grafana

The aim of Node-Red is to extract and convert the data coming from the base station to match the required data format of InfluxDB. The base station provides a JSON string containing the data of all sensors as well as addition information about the base station. Each sensor node is identified by a unique identifier called EUI. The goal is to find the correct data path for a received EUI from the base station and extract the relevant data from the string. InfluxDB requires the data to be transmitted as a JSON string, containing at least one key and value pair. An example of a possible message to InfluxDB is:

```
{
   "tempeature":22.43,
   "humidity":52,
   "pressure":1025
}
```

One pair consists of a key ("temperature") and a corresponding value (22.43). The unit of the measurement is not transmitted and must be specified later during the visualization process. To be able to publish this data, the API token of the last chapter must be provided.

Node-Red is running inside a container and can be accessed on port 1880 (URL: http://localhost:1880). Node-Red does not offer direct support of interacting with InfluxDB but this functionality can be imported. To **extend the palette** of functions either press ***Alt+Shift+P* or select *Manage Palette*** in the menu in the top right corner. Subsequently switch to the ***Install* section** and **search for *influxdb*** and **install the package *node-red-contrib-influxdb***. These steps are also shown in Figure 18.



User Settings

Close

View | Palette | Keyboard

Nodes | Install

sort: a-z recent

influxdb 9 / 4373

@wz2b/node-red-influxdb-line-protocol
Parses InfluxDB Line Protocol to JSON and vice versa.
0.2.0 1 year, 2 months ago install

node-red-contrib-influxdb
Node-RED nodes to save and query data from an influxdb time series database
0.6.1 2 years, 1 month ago install

node-red-contrib-influxdb-backup
A node for backing up influx database using influxd backup
0.1.0 2 years, 10 months ago install

Figure 18 Node-Red Installing InfluxDB

With the newly installed nodes further steps can be initiated. With the import feature it is easy to deploy a preconfigured flow, where all the important functions are already defined. The configuration is provided in a JSON string (see section 5.1.4) and can be easily **imported using the feature found inside the menu in the top right**. Copy the configuration in the empty field and **click on import**.

The flow consist of three different tasks (Figure 19) :

1. The reception of data from the base station
2. The data conversion and validation of the correct EUI
3. The storage of data to InfluxDB

Different nodes each provide different functionalities:

• Purple Node (MQTT in/out):	MQTT Broker or Client
• Blue Node (Inject node):	Used to manually inject data into the flow
• Green Node (Debug node):	Used to display messages of the flow
• Ocher Node (Function node):	Used to implement own functions in JavaScript
• Brown Node (InfluxDB out):	Used to store data in Influx database

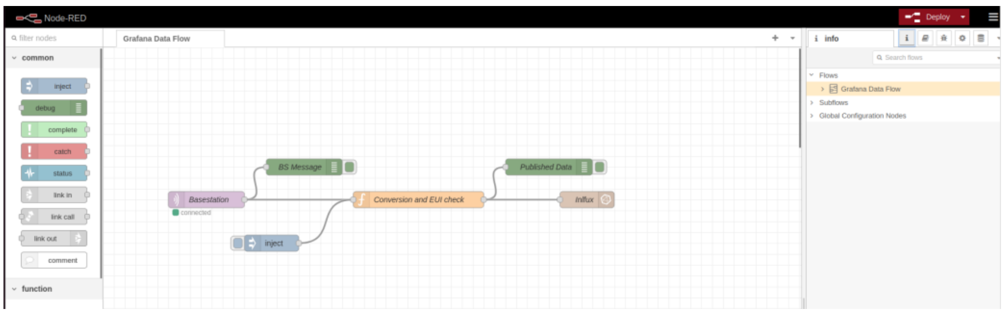


Figure 19 Node-Red Processing Flow

The Base station node represents a MQTT broker, which is subscribed to the topic *mioty/+/+/uplink*. This includes the newest uplink messages from each sensor node. This means that for each individual sensor node with a unique EUI a message is processed by the flow. Once new data is available from any sensor it is automatically received and processed by the flow. To connect to the base station, its **IP address must be entered** in the configuration. To open the configuration **double click the node** and **click on the pencil** next to the server to change the setup of the broker. Enter the correct IP address in the server field. It is the same IP address that is needed to access the web interface of the base station. The port is likely not changed and should stay 1883. To check if the connection can be established, click deploy in the top right corner and wait for a green box next to the node. Otherwise check the IP address again and make sure the base station and the PC are in the same network. Don't forget to deploy each change.

The next step is to extract the desired sensor data from the message coming from the base station and returning a JSON string in the format described above. Additionally the EUI of the message has to match the one specified by the function. This assures, that each message gets forwarded and stored only once and gets connected with the correct device in Thingsboard. The flow is preconfigured for a single device. If more devices are added, simply copy and paste (*Ctrl+C* and *Ctrl+V*) the conversion and publishing node. Refer to Figure 11 for the layout.

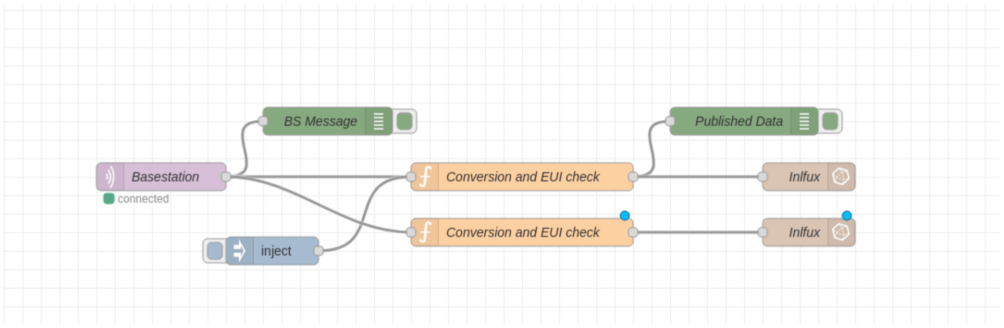


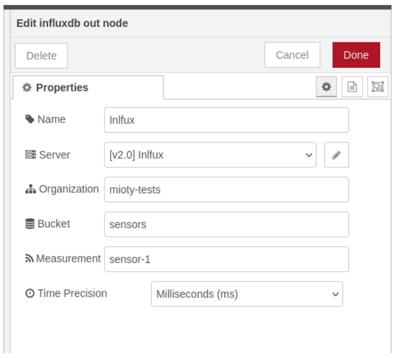
Figure 20 Node-Red multiple Sensor Nodes

For each sensor node one function block should exist and each node should be configured to the EUI of each sensor. **Double click on the node** to open its implementation and **enter the correct EUI** as a **decimal number** in the variable (**Hint**: many conversion tools can't handle large numbers, to convert the hex value to decimal inside Node-Red, click on the EUI value inside the debug window for an arriving message from the base station). If the EUI of the message complies with the specified value, this function returns a JSON string containing all the endpoint data to be stored in InfluxDB. If no conversion block exists for a given EUI the message is ignored.

In the last step the data gets sent to the database via the *InfluxDB out* node, which has to be configured first. This can be done by double clicking the node. First the database server has to be setup. You can edit the preconfigured settings by clicking the pencil on the right side. Make sure to select 2.0 as the version, because this allows the authentication via an API token. The URL of the server does in most cases not have to be changed, since all containers should be running in one environment. In the last step copy your generated API token from section 4.1. After the token is entered, update the configuration to save everything.

In the next step enter your chosen organization name from the last chapter as well as the bucket name, which should store the data. One bucket can be responsible for multiple sensor nodes. To distinguish the sensors form another, each one is stored in a new measurement with a unique name. So if multiple sensors should be added, only the

Measurement name in the *InfluxDB out* node has to be changed for each. As an example you can refer to Figure 21 for a setup.



Name: Inlfux
Server: [v2.0] Inlfux
Organization: mioty-tests
Bucket: sensors
Measurement: sensor-1
Time Precision: Milliseconds (ms)

Figure 21 Node-Red Influx Configuration

After everything is configured, make sure to deploy the changes by clicking the button in the top right corner. If a green square appears next to the base station, it could connect successfully. Otherwise make sure to verify the IP address of the base station. If new data is arriving from the sensor and its EUI is specified in the conversion function, the data should be displayed by the *Published Data* node. Inside the debug window you can also specify what messages from which node should be displayed to make it easier to follow. If no errors appear, the data should be successfully stored inside InfluxDB. Otherwise make sure the API token is setup correctly.

4.3
Data visualization in InfluxDB

Not only Grafana allows the displaying of data, but there is also a quick and easy way to inspect data and build dashboards in InfluxDB. To just inspect data head to the *Data Explorer*. Otherwise if you want to create a dashboard navigate to the *Dashboard* section by clicking the icon in the web interface of InfluxDB, create a new dashboard and add a cell to it.

The data explorer will open where you can choose the desired method of representing the data in the top left corner as well as specify what data should be displayed. For that select a bucket at the top level and subsequently specify which measurement and field you want to display. In most cases a measurement represents a specific sensor which again can have multiple sensor values like temperature and humidity. Figure 22 shows a reference on how this can be setup. To update the graph click the SUBMIT button.

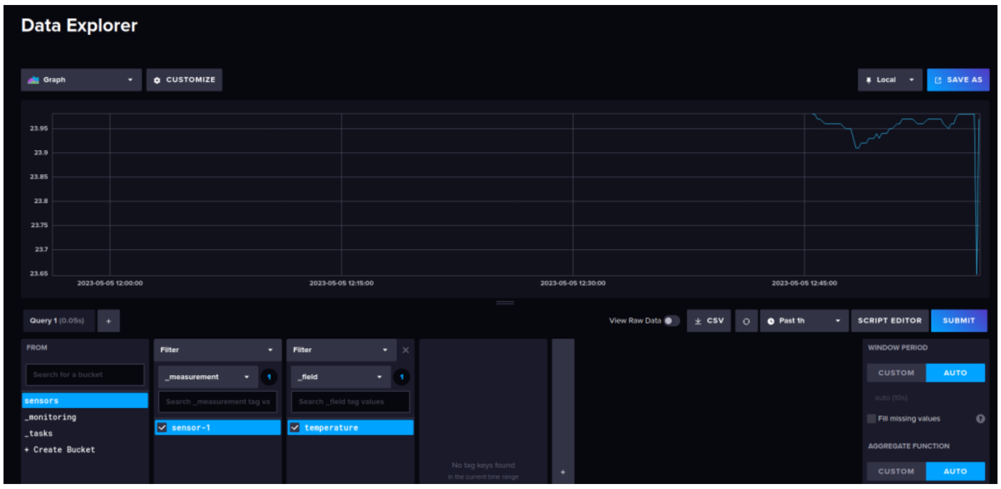


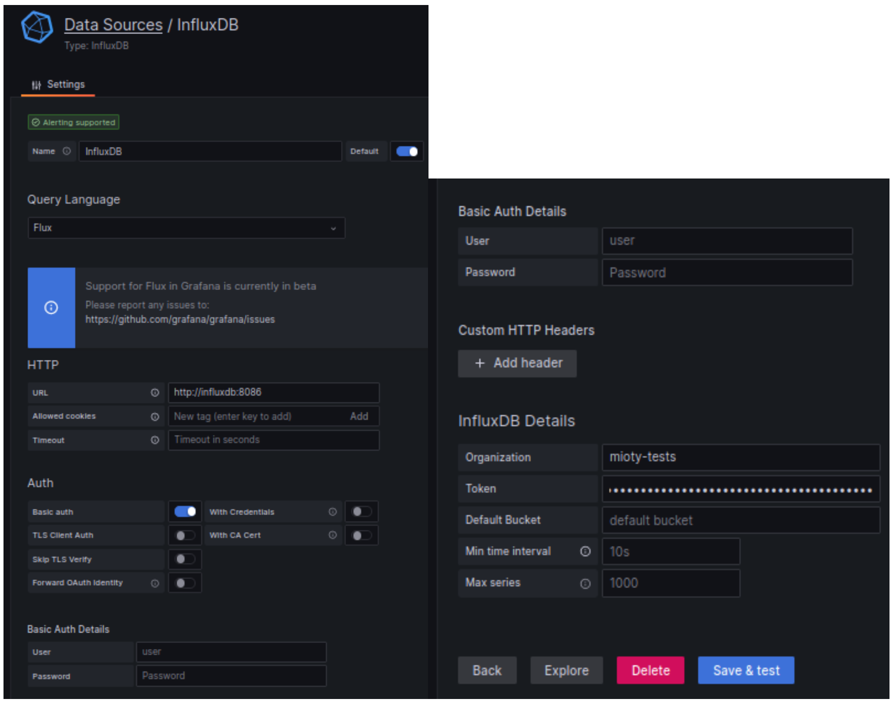
Figure 22 Displaying data in InfluxDB

For simple applications this representation might be sufficient, but for more advanced scenarios Grafana is the preferred option.

4.4 Data visualization with Grafana

Grafana is a software specifically designed for displaying data and is therefore a solid option to display the collected data from InfluxDB. The web interface is available via http://localhost:3000 on the host machine where the Docker containers are running. When trying to access Grafana for the first time, the default login information are *admin* for both the username and the password. You can configure a more secure password after the first login.

Before the data can be displayed, InfluxDB must first be added as a source to Grafana. On the Home tab you can find a direct link for that or navigate to *configuration* and *Data sources* in the bottom left corner. Now select *InfluxDB* as your desired data source and after that choose *Flux* as the *query language*. The only values that have to be filled out are under *InfluxDB Details* (Organization and (API) Token) and the URL of the Influx server, which is http://influxdb:8086. See Figure 23 for reference. By clicking *Save & Test*, Grafana automatically tests the data source and displays how many buckets are found in this server. If this was successful the first dashboard can be created.



Data Sources / InfluxDB
Type: InfluxDB

Settings

Alerting supported

Name	InfluxDB	Default	●

Query Language: Flux

Support for Flux in Grafana is currently in beta
Please report any issues to:
https://github.com/grafana/grafana/issues

HTTP

URL	http://influxdb:8086
Allowed cookies	New tag (enter key to add) Add
Timeout	Timeout in seconds

Auth

Basic auth	●	With Credentials	○
TLS Client Auth	○	With CA Cert	○
Skip TLS Verify	○		
Forward OAuth Identity	○		

Basic Auth Details

User	user
Password	Password

Custom HTTP Headers

InfluxDB Details

Organization	mioty-tests
Token	••••••••••••••••••••••••••••••••••••••
Default Bucket	default bucket
Min time interval	10s
Max series	1000

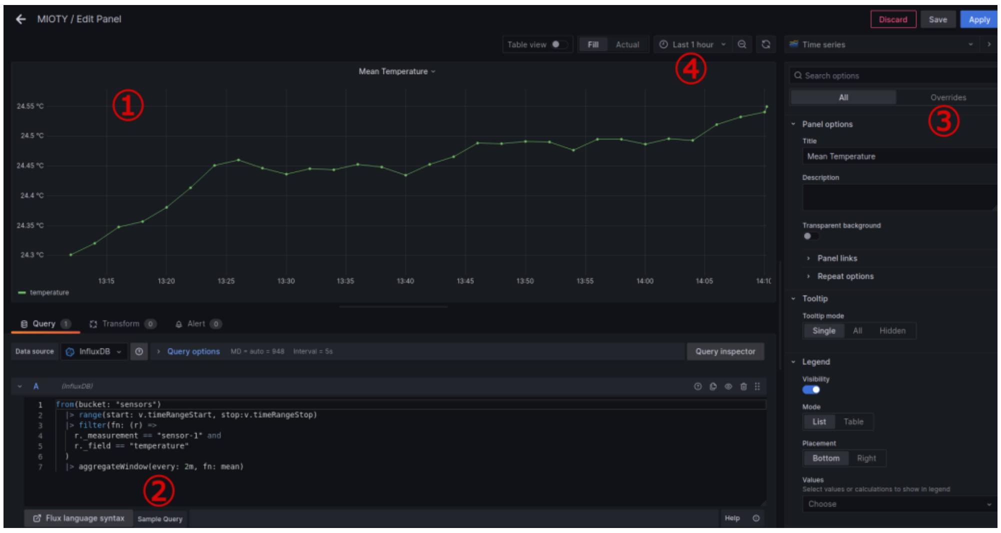
Figure 23 Grafana Data Source Configuration

Dashboards are used to display data in various formats and styles. A dashboard usually consists of multiple panels, which are used to display one dataset. For instance some types are time series graph, gauge, bar chart, heat map and histogram. The dashboards offer a high degree of customizability and can be build according to the needs of the application.

First head back to the home tab where you can find a panel to create your first dashboard and add the first panel to it. At first this may seem a bit complicated so let's take it step by step. Figure 24 shows an overview of the panel with the separate sections. The main field (number 1) contains the data representation like a graph or a gauge. Switching between the different possibilities can be done in the top right corner. Another important section (number 2) is the specification of which data should be represented. Firstly make sure, that the InfluxDB data source is selected. For accessing data a special programming language, called Flux, is used. Documentation can be found here: Documentation Flux. To make the process easier, select *Sample Query* -> *Simple Query* to get the skeleton of a request to the database. Now the bucket, measurement and field of the data needs to be specified. After that click in the graph window to update everything. If data is available for the specified time period, a graph should be visible. The period can be adjusted in the top right of the graph window (number 4). Additionally functions and filters can be applied on the data for instance smoothing the graph. This can be handled by a specific function *aggregateWindow()*. See the documentation for more details. As an example the call below applies a mean filter with the duration of 2 minutes on the data:

```
|> aggregateWindow(every: 2m, fn: mean)
```

In the right column title, measurement unit, scaling etc. can be configured. This section is pretty self-explanatory.



```
from(bucket: "sensors")
  |> range(start: v.timeRangeStart, stop:v.timeRangeStop)
  |> filter(fn: (r) =>
    r._measurement == "sensor-1" and
    r._field == "temperature"
  )
  |> aggregateWindow(every: 2m, fn: mean)
```

Figure 24 Grafana Panel Building

To finish the configuration, click on apply in the top right corner. Also make sure to save the dashboard, before closing it. Grafana offers many possibilities of data visualization. Figure 25 shows an example dashboard on how temperature data can be visualized.

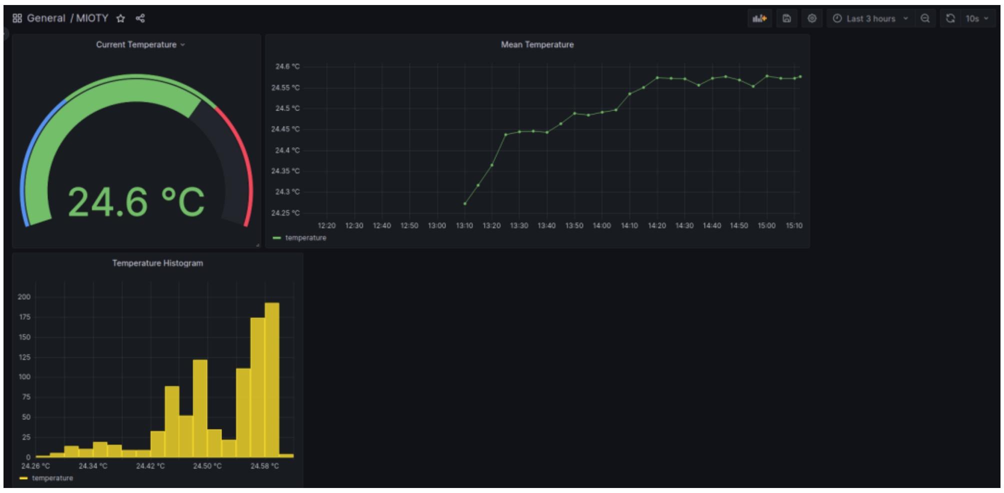


Figure 25 Grafana Example Dashboard

5 Appendix

5.1.1 Docker-Compose file Thingsboard Cloud Installation

```
version: '3.7'
services:
  node-red:
      image: nodered/node-red:latest
      container_name: nodered
      restart: always
      environment:
        - TZ=Europe/Amsterdam
      ports:
        - "1880:1880"
      volumes:
        - node_red_data:/data

volumes:
  node_red_data:
```

5.1.2 Docker-Compose file Thingsboard Local Installation

```
version: '3.7'
services:
  mytb:
    restart: always
    image: "thingsboard/tb-postgres"
    container_name: Thingsboard
    ports:
      - "8080:9090"
      - "1883:1883"
      - "7070:7070"
      - "5683-5688:5683-5688/udp"
    environment:
      TB_QUEUE_TYPE: in-memory
    volumes:
      - thingsboard_data:/data
      - thingsboard_log:/var/log/thingsboard

  node-red:
      image: nodered/node-red:latest
      container_name: nodered
      restart: always
      environment:
        - TZ=Europe/Amsterdam
      ports:
        - "1880:1880"
      volumes:
        - node_red_data:/data

volumes:
  node_red_data:
  thingsboard_data:
  thingsboard_log:
```

5.1.3 Docker-Compose file Grafana Installation

```
version: '3.7'
services:

  node-red:
      image: nodered/node-red:latest
      container_name: nodered
      restart: always
      environment:
        - TZ=Europe/Amsterdam
      ports:
        - "1880:1880"
      volumes:
        - node_red_data:/data

  influxdb:
      image: influxdb:latest
      container_name: influxdb
      restart: always
      environment:
        - INFLUXDB_DB=influx
        - INFLUXDB_ADMIN_USER=admin
        - INFLUXDB_ADMIN_PASSWORD=admin
      ports:
        - '8086:8086'
      volumes:
        - influxdb_data:/var/lib/influxdb

  grafana:
   image: grafana/grafana
   container_name: grafana-server
   restart: always
   depends_on:
     - influxdb
   environment:
     - GF_SECURITY_ADMIN_USER=admin
     - GF_SECURITY_ADMIN_PASSWORD=admin
     - GF_INSTALL_PLUGINS=
   links:
     - influxdb
   ports:
     - '3000:3000'
   volumes:
     - grafana_data:/var/lib/grafana

volumes:
  node-red-data: {}
  grafana_data: {}
  influxdb_data: {}
  node_red_data:
```

5.1.4
Node-Red Configuration Thingsboard

```
[
    {
        "id": "84d6f5ec8bdf3a8d",
        "type": "tab",
        "label": "Thingsboard Data Flow",
        "disabled": false,
        "info": ""
    },
    {
        "id": "3e38263fb3a29716",
        "type": "mqtt in",
        "z": "84d6f5ec8bdf3a8d",
        "name": "Basestation",
        "topic": "mioty/+/+/uplink",
        "qos": "2",
        "datatype": "json",
        "broker": "07827416c33c117e",
        "nl": false,
        "rap": true,
        "rh": 0,
        "inputs": 0,
        "x": 190,
        "y": 280,
        "wires": [
            [
                "f55004ed3cd17e0e",
                "6018209582f4d5d9"
            ]
        ]
    },
    {
        "id": "6018209582f4d5d9",
        "type": "function",
        "z": "84d6f5ec8bdf3a8d",
        "name": "Conversion and EUI check",
        "func": "\nvar eui = \"8121069193317515000\"; // In Decimal\n\nif
(String(msg.payload.typeEui) != eui) {\n    return null;\n}\n\nvar values = {};\nfor (var name in
msg.payload.components) {\n    var j = msg.payload.components[name];\n    values[name] =
j.value;\n}\n\nreturn { payload: values };\n",
        "outputs": 1,
        "noerr": 0,
        "initialize": "",
        "finalize": "",
        "libs": [],
        "x": 580,
        "y": 280,
        "wires": [
            [
                "feb91c841d654f61",
                "513cf183ab494b0d"
            ]
        ]
    },
    {
        "id": "feb91c841d654f61",
        "type": "debug",
        "z": "84d6f5ec8bdf3a8d",
        "name": "Published Data",
        "active": true,
        "tosidebar": true,
        "console": false,
        "tostatus": false,
        "complete": "payload",
        "targetType": "msg",
        "statusVal": "",
        "statusType": "auto",
        "x": 820,
        "y": 220,
        "wires": []
    },
    {
        "id": "a9ae1895bbd7447f",
        "type": "inject",
        "z": "84d6f5ec8bdf3a8d",
        "name": "",
        "props": [
            {
                "p": "payload"
            }
        ],
        "repeat": "",
        "crontab": "",
        "once": false,
        "onceDelay": 0.1,
        "topic": "",
        "payload":
"{\"baseStations\":[{\"bsEui\":812106946544412000,\"eqSnr\":18.389999389648438,\"mode\":\"ulp\",\
"profile\":\"eu1\",\"rssi\":-
59.70573806762695,\"rxTime\":1677148533954006000}],\"cnt\":533,\"components\":{\"temperature\":{\
"label\":\"\",\"unit\":\"°C\",\"value\":23.23},\"humidity\":{\"label\":\"\",\"unit\":\"%\",\"valu
e\":65.5}},\"data\":[9,19],\"dlAck\":false,\"dlOpen\":false,\"format\":0,\"meta\":null,\"response
Exp\":false,\"typeEui\":8121069193317515000}",
        "payloadType": "json",
        "x": 310,
        "y": 360,
        "wires": [
            [
                "6018209582f4d5d9"
            ]
        ]
    },
    {
        "id": "513cf183ab494b0d",
        "type": "mqtt out",
        "z": "84d6f5ec8bdf3a8d",
        "name": "",
        "topic": "v1/devices/me/telemetry",
        "qos": "",
        "retain": "",
        "respTopic": "",
        "contentType": "",
        "userProps": "",
        "correl": "",
        "expiry": "",
        "broker": "6b5d5d9152c7526e",
        "x": 930,
        "y": 280,
        "wires": []
    },
    {
        "id": "f55004ed3cd17e0e",
        "type": "debug",
        "z": "84d6f5ec8bdf3a8d",
        "name": "BS Message",
        "active": true,
        "tosidebar": true,
        "console": false,
        "tostatus": false,
        "complete": "payload",
        "targetType": "msg",
        "statusVal": "",
        "statusType": "auto",
        "x": 370,
        "y": 220,
        "wires": []
    },
    {
        "id": "07827416c33c117e",
        "type": "mqtt-broker",
        "name": "Basestation",
        "broker": "ava01b5.guests.iis.fhg.de",
        "port": "1883",
        "clientid": "",
        "autoConnect": true,
        "usetls": false,
        "protocolVersion": "4",
        "keepalive": "60",
        "cleansession": true,
        "birthTopic": "",
        "birthQos": "0",
        "birthPayload": "",
        "birthMsg": {},
        "closeTopic": "",
        "closeQos": "0",
        "closePayload": "",
        "closeMsg": {},
        "willTopic": "",
        "willQos": "0",
        "willPayload": "",
        "willMsg": {},
        "userProps": "",
        "sessionExpiry": ""
    },
    {
        "id": "6b5d5d9152c7526e",
        "type": "mqtt-broker",
        "name": "Thingsboard",
        "broker": "Thingsboard",
        "port": "1883",
        "clientid": "",
        "autoConnect": true,
        "usetls": false,
        "protocolVersion": "4",
        "keepalive": "60",
        "cleansession": true,
        "birthTopic": "",
        "birthQos": "0",
        "birthPayload": "",
        "birthMsg": {},
        "closeTopic": "",
        "closeQos": "0",
        "closePayload": "",
        "closeMsg": {},
        "willTopic": "",
        "willQos": "0",
        "willPayload": "",
        "willMsg": {},
        "userProps": "",
        "sessionExpiry": ""
    }
]
```

5.1.5 Node-Red Configuration Grafana

```
[
    {
        "id": "cda02be842129a63",
        "type": "tab",
        "label": "Grafana Data Flow",
        "disabled": false,
        "info": ""
    },
    {
        "id": "eb40d63af64b7495",
        "type": "mqtt in",
        "z": "cda02be842129a63",
        "name": "Basestation",
        "topic": "mioty/+/+/uplink",
        "qos": "2",
        "datatype": "json",
        "broker": "25fa4437ed512bbe",
        "nl": false,
        "rap": true,
        "rh": 0,
        "inputs": 0,
        "x": 190,
        "y": 280,
        "wires": [
            [
                "c4c277f9f2494228",
                "4d16f63252ed20c9"
            ]
        ]
    },
    {
        "id": "4d16f63252ed20c9",
        "type": "function",
        "z": "cda02be842129a63",
        "name": "Conversion and EUI check",
        "func": "\nvar eui = \"8121069193317515000\"; // In Decimal\n\nif
(String(msg.payload.typeEui) != eui) {\n    return null;\n}\n\nvar values = {};\nfor (var name in
msg.payload.components) {\n    var j = msg.payload.components[name];\n    values[name] =
j.value;\n}\n\nreturn { payload: values };\n",
        "outputs": 1,
        "noerr": 0,
        "initialize": "",
        "finalize": "",
        "libs": [],
        "x": 580,
        "y": 280,
        "wires": [
            [
                "5a821bbf4f4083c0",
                "95eafe0459769668"
            ]
        ]
    },
    {
        "id": "5a821bbf4f4083c0",
        "type": "debug",
        "z": "cda02be842129a63",
        "name": "Published Data",
        "active": true,
        "tosidebar": true,
        "console": false,
        "tostatus": false,
        "complete": "payload",
        "targetType": "msg",
        "statusVal": "",
        "statusType": "auto",
        "x": 820,
        "y": 220,
        "wires": []
    },
    {
        "id": "98892ade85cc9a4d",
        "type": "inject",
        "z": "cda02be842129a63",
        "name": "",
        "props": [
            {
                "p": "payload"
            }
        ],
        "repeat": "",
        "crontab": "",
        "once": false,
        "onceDelay": 0.1,
        "topic": "",
        "payload":
"{\"baseStations\":[{\"bsEui\":812106946544412000,\"eqSnr\":18.389999389648438,\"mode\":\"ulp\",\
"profile\":\"eu1\",\"rssi\":-
59.70573806762695,\"rxTime\":1677148533954006000}],\"cnt\":533,\"components\":{\"temperature\":{\
"label\":\"\",\"unit\":\"°C\",\"value\":23.23},\"humidity\":{\"label\":\"\",\"unit\":\"%\",\"valu
e\":65.5}},\"data\":[9,19],\"dlAck\":false,\"dlOpen\":false,\"format\":0,\"meta\":null,\"response
Exp\":false,\"typeEui\":8121069193317515000}",
        "payloadType": "json",
        "x": 310,
        "y": 360,
        "wires": [
            [
                "4d16f63252ed20c9"
            ]
        ]
    },
    {
        "id": "c4c277f9f2494228",
        "type": "debug",
        "z": "cda02be842129a63",
        "name": "BS Message",
        "active": true,
        "tosidebar": true,
        "console": false,
        "tostatus": false,
        "complete": "payload",
        "targetType": "msg",
        "statusVal": "",
        "statusType": "auto",
        "x": 370,
        "y": 220,
        "wires": []
    },
    {
        "id": "95eafe0459769668",
        "type": "influxdb out",
        "z": "cda02be842129a63",
        "influxdb": "b7d663d984df12d2",
        "name": "Inlfux",
        "measurement": "sensor-1",
        "precision": "",
        "retentionPolicy": "",
        "database": "database",
        "precisionV18FluxV20": "ms",
        "retentionPolicyV18Flux": "",
        "org": "mioty-tests",
        "bucket": "sensors",
        "x": 890,
        "y": 280,
        "wires": []
    },
    {
        "id": "25fa4437ed512bbe",
        "type": "mqtt-broker",
        "name": "Base Station",
        "broker": "ava01b5.guests.iis.fhg.de",
        "port": "1883",
        "clientid": "",
        "autoConnect": true,
        "usetls": false,
        "protocolVersion": "4",
        "keepalive": "60",
        "cleansession": true,
        "birthTopic": "",
        "birthQos": "0",
        "birthPayload": "",
        "birthMsg": {},
        "closeTopic": "",
        "closeQos": "0",
        "closePayload": "",
        "closeMsg": {},
        "willTopic": "",
        "willQos": "0",
        "willPayload": "",
        "willMsg": {},
        "userProps": "",
        "sessionExpiry": ""
    },
    {
        "id": "b7d663d984df12d2",
        "type": "influxdb",
        "hostname": "127.0.0.1",
        "port": "8086",
        "protocol": "http",
        "database": "database",
        "name": "Inlfux",
        "usetls": false,
        "tls": "",
        "influxdbVersion": "2.0",
        "url": "http://influxdb:8086",
        "rejectUnauthorized": true
    }
]
```