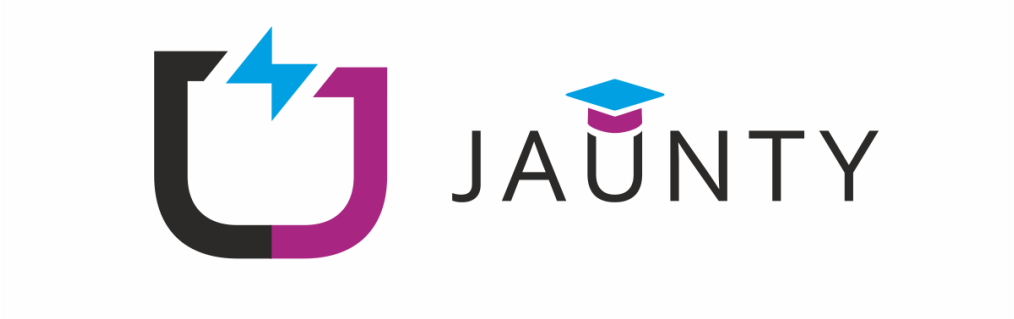
Real-time Monitoring with ThingsBoard and CoAP



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

CoAP	Constrained Application Protocol
UPS	Uninterruptible Power Supply
MQTT	MQ Telemetry Transfer
HTTP	HyperText Transport Protocol
RFC	Request for Comments
URI	Unique Resource Identifier
UDP	User Datagram Protocol
RPI4B	Raspberry Pi 4B
VM	Virtual Machine
DC	Direct Current
I/O	Input Output
ASCII	American Standard Code for Information Interchange
JSON	JavaScript Object Notation

2. Scope

The scope of this laboratory exercise is to provide a comprehensive introduction to the ThingsBoard IoT platform and its most important features as well as to the CoAP protocol, that is used for lightweight communication between IoT devices. In particular, the students will implement two applications that will retrieve measurements from an Uninterruptible Power Supply (UPS) and an SHT40 sensor and post those measurements to ThingsBoard via CoAP. Finally, the students will leverage the ThingsBoard Rule Engine to transform the incoming data streams and will configure a custom dashboard to visualise the telemetry in real-time.

2.1 Specific outcomes

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

- Install/deploy, operate, and maintain the ThingsBoard IoT platform.
- Configure ThingsBoard as tenant administrator and provision devices, apply data transformations, raise alarms, create dashboards and widgets.
- Develop asynchronous code in Python.
- Posting data with the Constrained Application Protocol (CoAP).

2.2 General description

Purpose of this laboratory is to utilise ThingsBoard and CoAP to monitor the power supply and the ambient temperature/humidity of a computer room. Monitoring includes not only real-time notifications and real-time visualisation of data, but also historic search and visualisation of data, potentially useful for diagnosing power issues [1].

Interaction will be established with two IoT devices, namely a) an SHT40 sensor, and b) an APC Smart-UPS SRT 6KVA.

The SHT40 sensor is depicted in Figure 1. SHT40 can measure both the relative humidity[1] and the temperature of the environment, with ± 1.8% typical relative humidity accuracy from 25 to 75% and ± 0.2°C from 0 to 75°C. It is imperative to constantly monitor the ambient temperature and humidity of the computer room since the servers and sensitive electronic devices need low temperatures to operative normally. Measurements can be retrieved by connecting the sensor to an I2C bus [2].

The APC Smart-UPS SRT 6KVA is depicted in Figure 2. Smart-UPS is a product series from APC Schneider Electric, which provides network power protection for servers, routers, switches, and other mission critical devices. The UPS not only provides emergency power to equipment when the main power source fails, but also acts as power meter/analyser, since it can measure various attributes of the

[1] Relative humidity is the ratio of how much water vapour is in the air and how much water vapour the air could potentially contain at a given temperature (e.g., colder air can hold less vapour). Therefore, depending on the air temperature, the relative humidity can change, even when the absolute humidity remains constant.

electric system, including the output power, the input voltage, the system frequency etc. Measurements can be retrieved by the UPS via the Modbus TCP protocol. Table 1 provides the Modbus register mapping of the UPS, summarising the Modbus registers that are of interest in the context of this laboratory exercise [3].

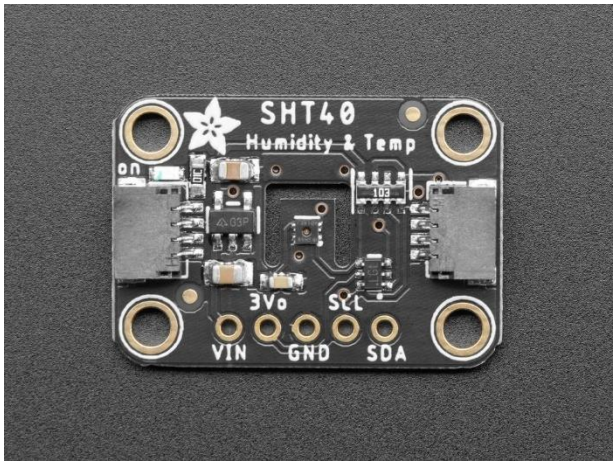


Figure 1: The SHT40 sensor

Figure 2: The APC Smart-UPS SRT 6KVA

The measurements from both devices will be collected and illustrated to ThingsBoard. This is an open source IoT platform for the collection, processing, and visualisation of IoT data as well as device management. IoT devices can be integrated with the IoT platform via industry standard protocols, including MQTT, HTTP, and CoAP.

In this lab, the integration will be done with the Constrained Application Protocol (CoAP). CoAP is an Internet application protocol defined in RFC 7252. It is a dedicated protocol for use with constrained nodes in low-power operation. CoAP operates in a server-client manner, supports discovery of services and includes known Web concepts, like Unique Resource Identifiers (URIs). It operates over UDP and is characterised for its low overhead and simplicity [1].

Table 1: Modbus register mapping of the APC UPS

Modbus address	Bit	Field name	Description	Scale (divide by)
130		StateOfCharge_Pct	The percent state of charge in the battery.	512
135		Battery.Temperature	Battery temperature in Degrees C.	128
140		Output[0].CurrentAC	Phase 1 - Measured AC RMS Current.	32
142		Output[0].VoltageAC	Phase 1 - Measured Output Voltage.	64
144		Output.Frequency	Measured frequency on the output.	128
147	0	Bypass.InputStatus_BF. Acceptable	Input (both voltage and frequency) is acceptable and all other system constraints are met so that the UPS can power the output with this input source.	
	1	Bypass.InputStatus_BF. PendingAcceptable	Input (both voltage and frequency) is acceptable but at least one other system constraint is not met preventing the line from being declared acceptable (e.g. line is not stable for a long enough time).	
	2	Bypass.InputStatus_BF. VoltageTooLow	The input voltage is too low to be acceptable.	
	3	Bypass.InputStatus_BF. VoltageTooHigh	The input voltage is too high to be acceptable.	
	4	Bypass.InputStatus_BF.Distorted	Indicates a distorted input waveform. The input voltage is too different from reference waveform, the frequency is moving too fast to track, or the frequency is out of measurable range.	
	5	Bypass.InputStatus_BF. BoostVoltage	The UPS is attempting to amplify the input voltage. Not applicable for bypass input.	
	6	Bypass.InputStatus_BF. TrimVoltage	The the UPS is attempting to attenuate the input voltage. Not applicable for bypass input.	
	7	Bypass.InputStatus_BF. FrequencyTooLow	Indicates frequency is measurably too low.	
	8	Bypass.InputStatus_BF. FrequencyTooHigh	Indicates frequency is measurably too high.	
	9	Bypass.InputStatus_BF. FreqAndPhaseNotLocked	The system is not frequency and phase locked to the input frequency and phase.	
	10	Bypass.InputStatus_BF. PhaseDeltaOutOfRange	The difference in phase angle between phases is out of range.	
	11	Bypass.InputStatus_BF. NeutralNotConnected	The Neutral connection is missing.	
151		Input[0].VoltageAC- Phase 1	Phase 1 - Measured Input Voltage.	64
564		Serial Number	-	
596		Name_STR	The name assigned to the UPS.	

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	
RPI4B IP address	
RPI4B username	
RPI4B password	

3. Exercise 1: Deploy the IoT Environment

The first exercise has to do with deploying all the necessary IoT components for this lab as well as to establish communication with the IoT devices. In particular, the following need to be deployed:

- A Raspberry Pi 4B (RPI4B) that interfaces with an SHT40 sensor via an I2C bus.
- An APC UPS that supports and monitors the power supply.
- The Raspberry Pi 4B can communicate with the APC UPS via Modbus TCP.
- ThingsBoard, which can easily be deployed via Docker on a Linux VM.

Figure 3 depicts an overview of the deployment that will be followed in this lab.

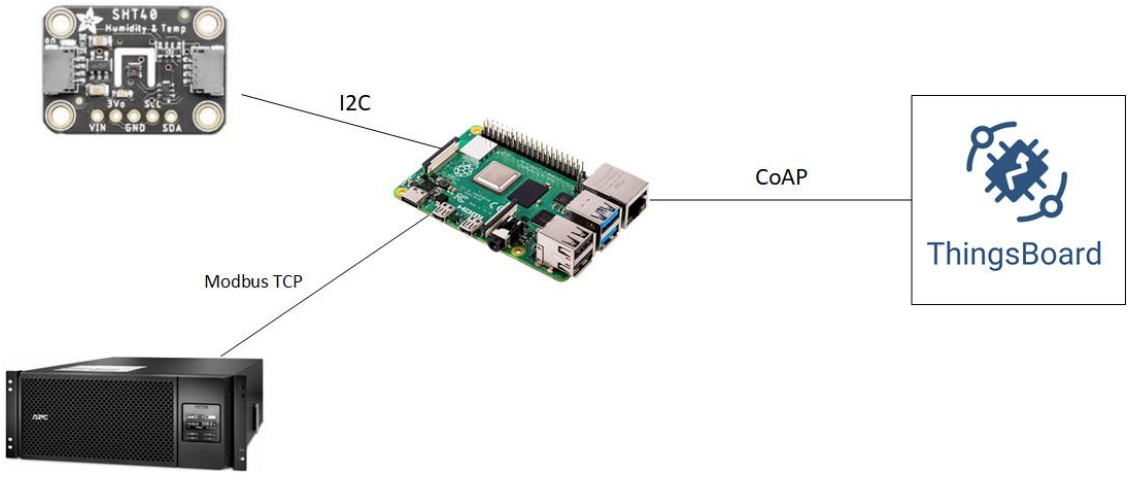


Figure 3: The deployment scheme for this lab

Step 1: Implement the breadboard diagram

This prototype topology has already been implemented and is remotely available in the physical laboratory premises. It is recommended to follow the steps of this exercise to replicate the prototype with your own equipment. In case that this is not possible, then you could proceed to step 2 by accessing the RPI4B platform already provided. Access details are provided in section 2.3.

Before implementing any change to the topology, you must first deactivate any DC power supply. Provide power supply to RPI4B and activate the bus supply, ONLY when the implementation has been completed.

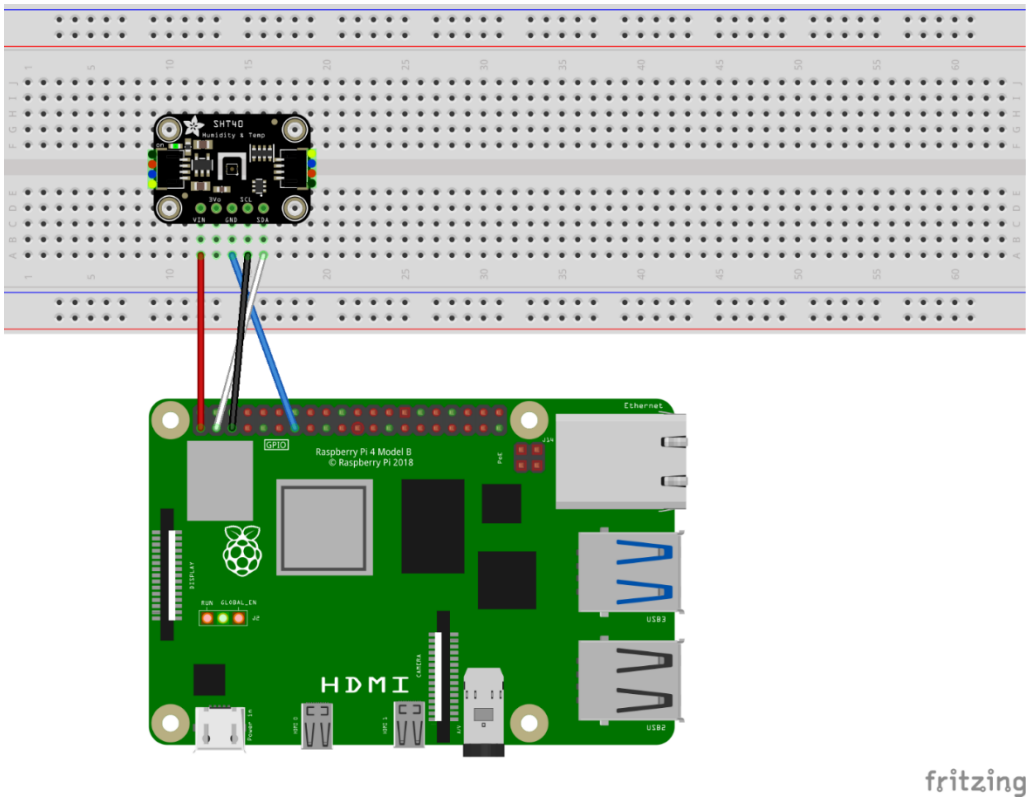


Figure 4: Breadboard diagram for the SHT40 sensor

Based on the exercise of the lab "I2C communication with Raspberry Pi 4B", try to implement the topology illustrated in Figure 4.

Step 2: Asynchronous Connectivity with the SHT40 sensor via I2C

Due to package dependencies at later steps of this lab, the asynchronous feature of Python has to be applied on the Python code. Asynchronous is a feature of modern programming languages (e.g., JavaScript) that permits running multiple operations without waiting time. This is a preferred way to handle multiple network tasks or I/O tasks, where the main program waits for the other tasks to finish.

The following Python source code can be used for retrieving the humidity and temperature measurements from the SHT40 sensor.

Based on the guidelines included in the lab "I2C communication with Raspberry Pi 4B", prepare the Python virtual environment with all the necessary packages to run the code bellow. Make sure to install the following packages: `board`, `adafruit_sht4x`, `asyncio`. Deploy the Python script on the RPI4B platform and ensure its working.

```
1  import board
2  import adafruit_sht4x
```

```
3  import asyncio
4
5  async def getMeasurements():
6      sht = adafruit_sht4x.SHT4x(board.I2C())
7      measurements = sht.measurements
8      print(measurements)
9      return measurements
10
11 async def main():
12     while True:
13         measurements = await getMeasurements()
14         await asyncio.sleep(5)
15
16 if __name__ == "__main__":
17     asyncio.run(main())
18
```

In lines 5-9, the asynchronous function (or coroutine) is defined, which undertakes to retrieve, print, and return the humidity and temperature measurements. The async keyword is required to specify a coroutine instead of a synchronous function.

The main coroutine is defined in lines 11-14. The main coroutine runs an infinite loop, which calls the getMeasurements() coroutine to return the current measurements each 5 seconds. Please note that coroutines (functions defined with async) are awaitable objects, thus, they can be awaited by other coroutines. The await getMeasurements() in line 13 means that the execution of the main() coroutine is interrupted, waiting for the getMeasurements() function to be completed. Upon getMeasurements() is completed, the main() coroutine resumes.

Finally, line 17 utilises the asyncio.run() function to run the top-level entry point main() coroutine.

Step 3: Asynchronous Connectivity with the APC UPS via Modbus TCP

The Python code provided bellow undertakes to retrieve electricity measurements from the APC UPS.

Based on the guidelines included in the lab “Introduction to Modbus TCP”, prepare a new Python virtual environment with all the necessary packages to run the code bellow. Make sure to install the following packages: `pymodbus`, `asyncio`. Deploy the Python script on the RPI4B platform and ensure its working.

```
1  from pymodbus.client.sync import ModbusTcpClient
2  import json
3  import asyncio
4
5  ANALOG_DATA_ADDRESSES = (130, 135, 140, 142, 144, 147, 151)
6  DELAY = 1
7
8  def holding_register_to_string(hr_response):
9      final_string = ""
```

```
    for item in hr_response.registers:
        binary_string = bin(item).replace('0b',"").zfill(16)
        first_letter = int(binary_string[:8], 2)
        second_letter = int(binary_string[8:], 2)
        final_string = final_string + chr(first_letter) + chr(second_letter)
    return final_string.strip()

async def main():
    UNIT_ID = 1
    slave = ModbusTcpClient('192.168.86.225', port=502)

    if not slave.is_socket_open():
        slave.connect()
    device_SN = slave.read_holding_registers(564, count=8, unit=UNIT_ID)
    slave.close()
    device_SN = holding_register_to_string(device_SN)

    if not slave.is_socket_open():
        slave.connect()
    device_name = slave.read_holding_registers(596, count=8, unit=UNIT_ID)
    slave.close()
    device_name = holding_register_to_string(device_name)

    while True:
        response_holding_registers = [0]*len(ANALOG_DATA_ADDRESSES)

        for i in range(len(ANALOG_DATA_ADDRESSES)):
            if not slave.is_socket_open():
                slave.connect()

            response_holding_registers[i] =
slave.read_holding_registers(ANALOG_DATA_ADDRESSES[i], unit=UNIT_ID)
            slave.close()

            if response_holding_registers[i].isError():
                print('A Modbus exception occurred. Terminating..')
                exit(-1)

        InputStatus_BF_Decimal = response_holding_registers[5].registers[0]

        record_json = {
            "device": {
                "name": device_name,
                "sn": device_SN
            },
            "StateOfChargePct": response_holding_registers[0].registers[0],
            "BatteryTemperature": response_holding_registers[1].registers[0],
            "OutputCurrentAC": response_holding_registers[2].registers[0],
            "OutputVoltageAC": response_holding_registers[3].registers[0],
            "OutputFrequency": response_holding_registers[4].registers[0],
            "BypassInputStatus_BF": {
                "InputAcceptable": InputStatus_BF_Decimal & 1 != 0,
                "PendingAcceptable": InputStatus_BF_Decimal & 2 != 0,
                "VoltageTooLow": InputStatus_BF_Decimal & 4 != 0,
                "VoltageTooHigh": InputStatus_BF_Decimal & 8 != 0,
                "Distorted": InputStatus_BF_Decimal & 16 != 0,
                "BoostVoltage": InputStatus_BF_Decimal & 32 != 0,
                "TrimVoltage": InputStatus_BF_Decimal & 64 != 0,
                "FrequencyTooLow": InputStatus_BF_Decimal & 128 != 0,
                "FrequencyTooHigh": InputStatus_BF_Decimal & 256 != 0,
                "FreqAndPhaseNotLocked": InputStatus_BF_Decimal & 512 != 0,
                "PhaseDeltaOutOfRange": InputStatus_BF_Decimal & 1024 != 0,
                "NeutralNotConnected": InputStatus_BF_Decimal & 2048 != 0,
```

```
74                 },
75                 "InputVoltage": response_holding_registers[6].registers[0],
76             }
77             record_json = json.dumps(record_json)
78
79             print(record_json)
80
81             await asyncio.sleep(DELAY)
82
83             continue
84
85 if __name__ == "__main__":
86     asyncio.run(main())
87
```

The main coroutine starts in line 19 by initiating connection with the Modbus server of the APC UPS. Subsequently, the register address 564 is queried, which holds a string representing the device serial number. According to the official documentation of Schneider Electric, the length of any ASCII field is 8 bytes, so the count=8 input parameter is also specified in lines 23 and 31 respectively.

It should be noted that the pymodbus library returns the contents of each register in decimal form. Therefore, it is necessary to convert the decimal values to ASCII characters, to retrieve the human-readable string. For this reason, the holding_register_to_string() has been defined in lines 8-15. This function receives the Modbus response in the form of an array with decimal numbers, each number corresponds to the contents of a register address in decimal form (keep in mind that a string occupies 8 register addresses according to the official documentation of the UPS). An example of such array/response is [16720, 17184, 21840, 21280, 8224, 8224, 8224, 8224] which corresponds to “APC UPS”. The conversion is implemented as follows: Since each register address occupies 16 bits, two letters are stored in each register address (each letter sizes 8 bits according to ASCII). To retrieve the letters, each number of the array is converted to binary (line 11), and then is split, the first half corresponds to the binary representation of the first letter (line 12) and the second half on the binary representation of the second letter (line 13). Each binary representation is converted to decimal (lines 13-14) and finally the letters are converted from decimal to characters according to the ASCII mapping and are concatenated (line 14). For example, 16720 corresponds to 0100000101010000 (binary signed 2's complement) which splits to 01000001 and 01010000. 01000001 decimal value is 65, which corresponds to “A” in ASCII. 01010000 decimal value is 80, which corresponds to letter “P”. So, 16720 corresponds to “AP”.

After retrieving the device serial number (lines 21-25) and the device name (lines 29-33), an infinite loop starts in line 35 which frequently retrieves the UPS measurements. For each holding register address, the read_holding_registers() function is called (line 42) to retrieve the corresponding contents.

Whereas the contents of almost all register correspond to a continuous decimal or float number (e.g., frequency, percentage, voltage, etc), the register 147 contains a binary value, where each bit has a different meaning. So, we need to convert the contents of that register to binary and match each bit to a separate measurement. This process is illustrated in lines 61-73. To quickly ascertain whether each bit is 1 or 0, an AND operation is applied between the decimal value of the register and the decimal

value of the position we are checking. For example, the operation InputStatus_BF_Decimal & 4 is true if the third bit is 1 (2^2=4), and InputStatus_BF_Decimal & 256 is true if the ninth bit is 1 (2^8=256), etc.

Finally, the dictionary of all current measurements Is converted to JSON format, is printed on screen and the loop runs again after sleeping for some seconds.

Step 4: Deploy ThingsBoard Using Docker

Create a new folder on your working directory, named “thingsboard”. In this folder, create a file named “docker-compose.yml” and add the following content:

```
1  version: '2.2'
2  services:
3    zookeeper:
4      restart: always
5      image: "zookeeper:3.5"
6      ports:
7        - "2181:2181"
8      environment:
9        ZOO_MY_ID: 1
10       ZOO_SERVERS: server.1=zookeeper:2888:3888;zookeeper:2181
11   kafka:
12     restart: always
13     image: wurstmeister/kafka
14     depends_on:
15       - zookeeper
16     ports:
17       - "9092:9092"
18     environment:
19       KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
20       KAFKA_LISTENERS: INSIDE://:9093,OUTSIDE://:9092
21       KAFKA_ADVERTISED_LISTENERS:
INSIDE://:9093,OUTSIDE://kafka:9092
22       KAFKA_LISTENER_SECURITY_PROTOCOL_MAP:
INSIDE:PLAINTEXT,OUTSIDE:PLAINTEXT
23       KAFKA_INTER_BROKER_LISTENER_NAME: INSIDE
24     volumes:
25       - /var/run/docker.sock:/var/run/docker.sock
26   mytb:
27     restart: always
28     image: "thingsboard/tb-postgres"
29     depends_on:
30       - kafka
31     ports:
32       - "8080:9090"
33       - "1883:1883"
34       - "7070:7070"
35       - "5683-5688:5683-5688/udp"
```

```
36      environment:
37        TB_QUEUE_TYPE: kafka
38        TB_KAFKA_SERVERS: kafka:9092
39      volumes:
40        - ~/.mytb-data:/data
41        - ~/.mytb-logs:/var/log/thingsboard
```

The docker-compose.yml describes the deployment of three services, namely a) Apache Zookeeper, b) Apache Kafka, c) ThingsBoard with PostgreSQL. In more detail, Apache Kafka is the recommended option for production deployments and acts as queue for asynchronous communication between ThingsBoard components. In other words, Apache Kafka works in a similar way to MQTT, allowing software components (called consumers by Kafka) to subscribe to specific topics, and other components (called producers by Kafka) to publish contents to those topics.

To deploy the ThingsBoard services, issue the following command while being in the directory where docker-compose.yml is located:

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

4. Exercise 2: Initialise ThingsBoard

In this exercise you are going to intialise the ThingsBoard installation as a system and tenant administrator, by creating all the necessary ThingsBoard entities and users.

Step 1: Create a tenant

Tenant is a business entity (e.g., company or organisation) that owns IoT devices. In this lab, we are going to create a tenant that corresponds to JAUNTY. Under this tenant, we are going to add the IoT devices that generate the monitoring data.

Tenants can be created by system administrators, therefore, use a web browser to navigate at **http://***WORKSTATION_IP***:8080** and login using the following credentials:

Username: **sysadmin@thingsboard.org**

Password: **sysadmin**

Figure 5 depicts the homepage of the system administrator. Through this page, the administrator can manage the existing tenants as well as to access various system settings (e.g., mail server, security settings, authentication methods, etc).

While being on the system administrator homepage, click on “Tenants” to access the menu that allows to manage the system’s tenants. Figure 6 depicts the Tenants menu.

To create a new tenant, click on the + button (annotated in Figure 6). This action will open a form to create a new tenant, as illustrated in Figure 7. Fill the form appropriately and click “Add”. The new tenant is depicted in Figure 8. The name of the new tenant is “JAUNTY Project”.

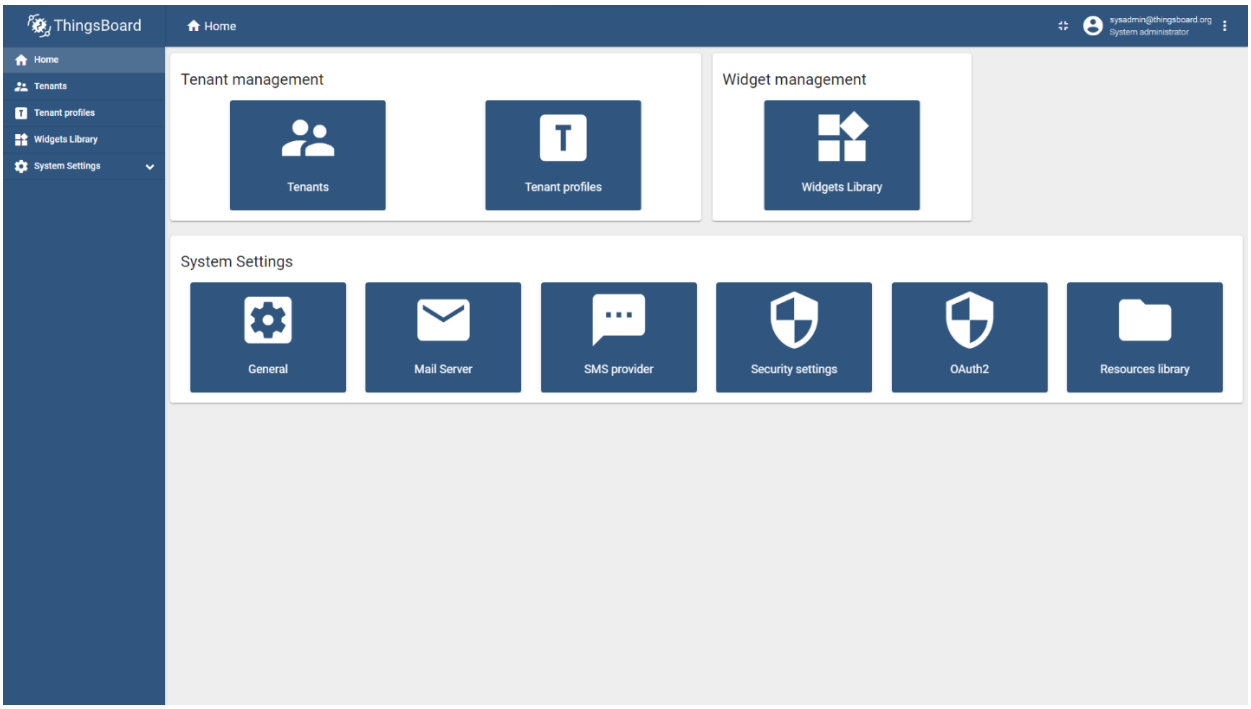


Figure 5: Homepage of the System Administrator

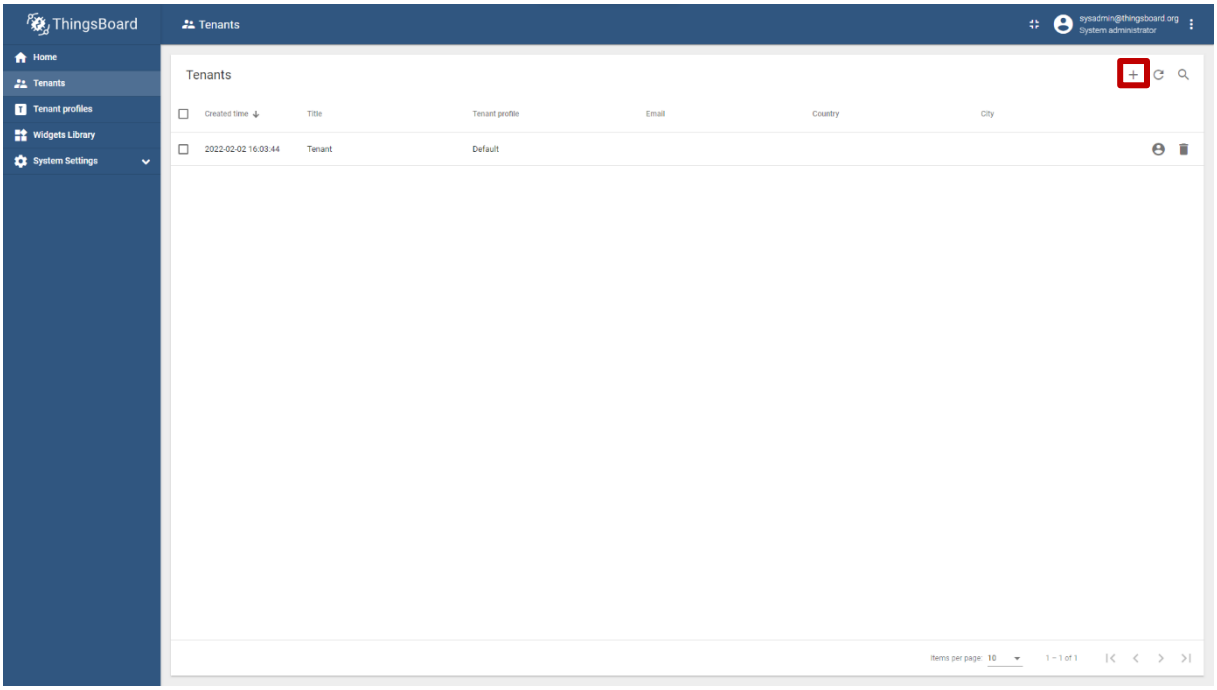


Figure 6: The tenants menu

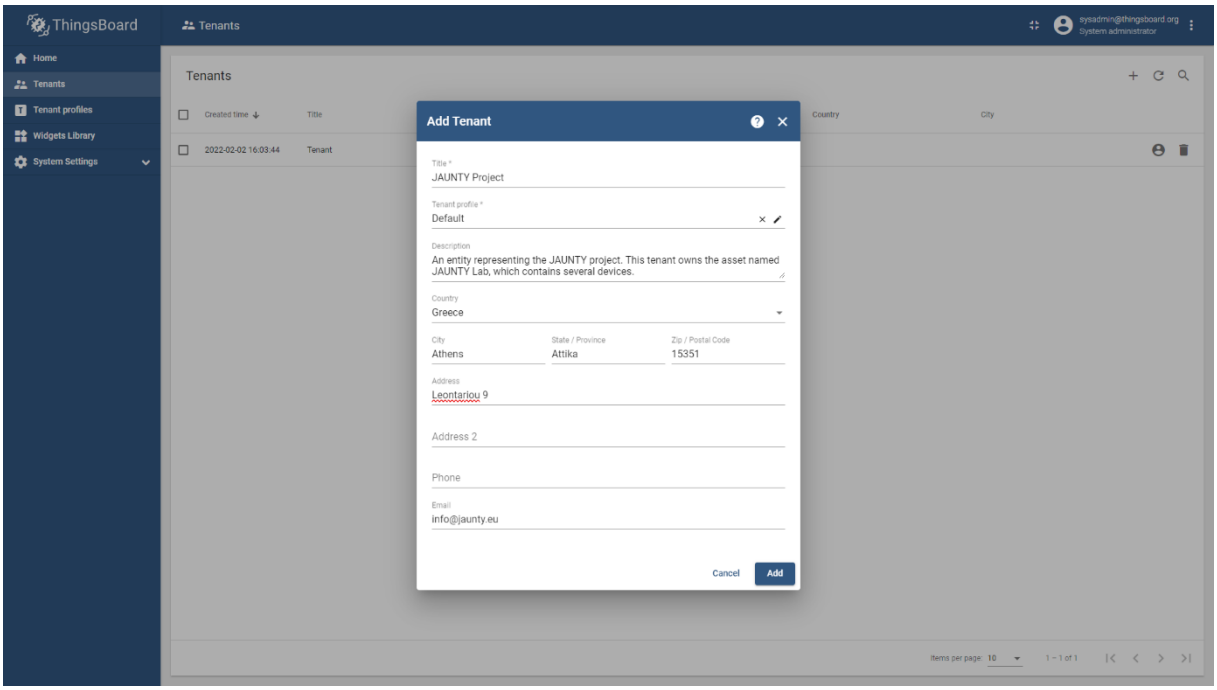


Figure 7: Add Tenant form

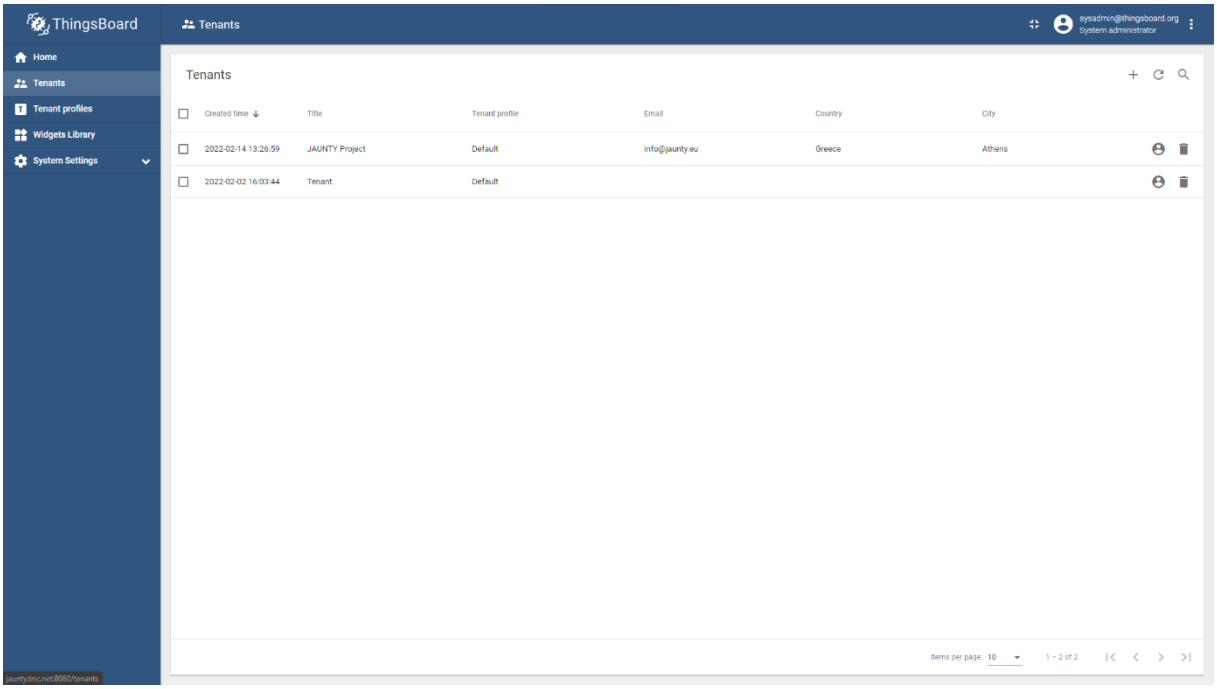


Figure 8: The new tenant

Step 2: Add user to tenant

As a business entity, a tenant needs one or multiple users that can login to ThingsBoard and administer the assets/devices owned by the tenant. For this purpose, in this step, you are going to add a new user which will be mapped to the tenant created for JAUNTY.

To manage tenant's admins, click on the user icon of the JAUNTY tenant, as annotated in Figure 8. The Tenants admin menu is depicted in Figure 9. To create a new tenant admin, click on the + icon (annotated in Figure 9).

On the new tenant menu, complete all the fields appropriately, as depicted in Figure 10, and make sure you select the "Display activation link" option on the Activation method. This method is required to initialise the password of your account, however, it is not possible to receive the activation link via email, since the mail service has not been configured.

After clicking "Add", the activation link will pop up, as depicted in Figure 11.

Open the activation link of Figure 11 on a new browser window or tab. This will open the form to initialise a password for your account, as illustrated in Figure 12. After defining the new password, you will be able to login using the email specified in Figure 10.

The homepage of the tenant administrator is illustrated in Figure 13.

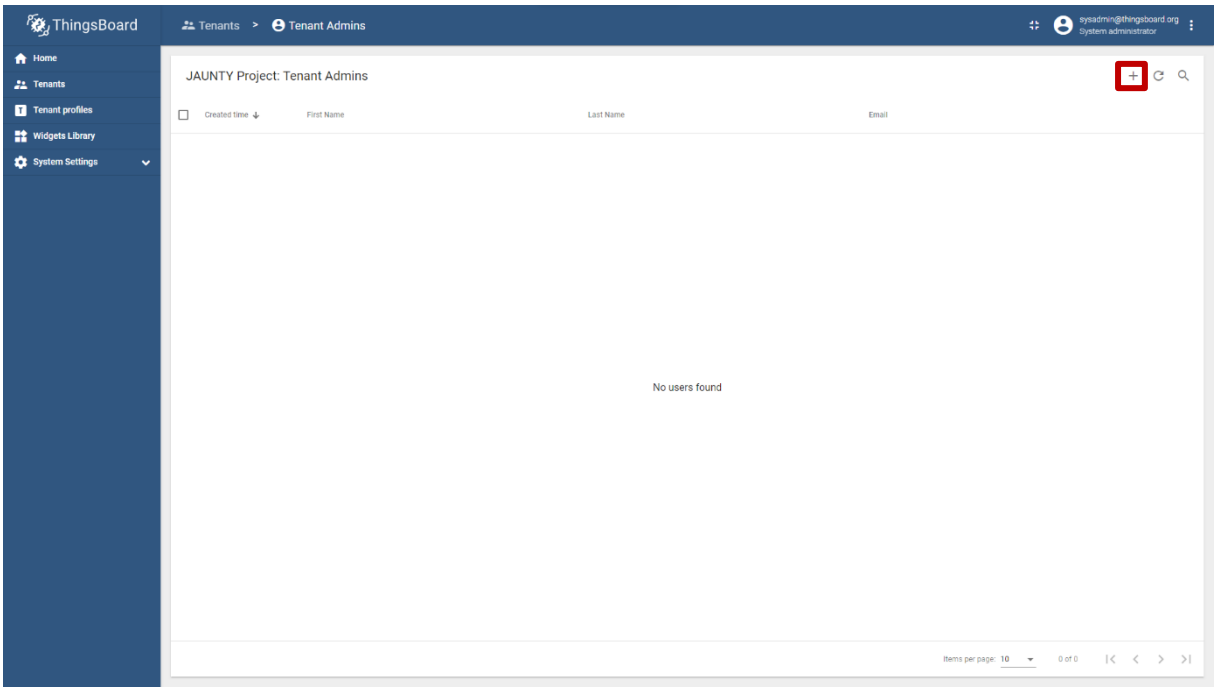


Figure 9: Tenants admin

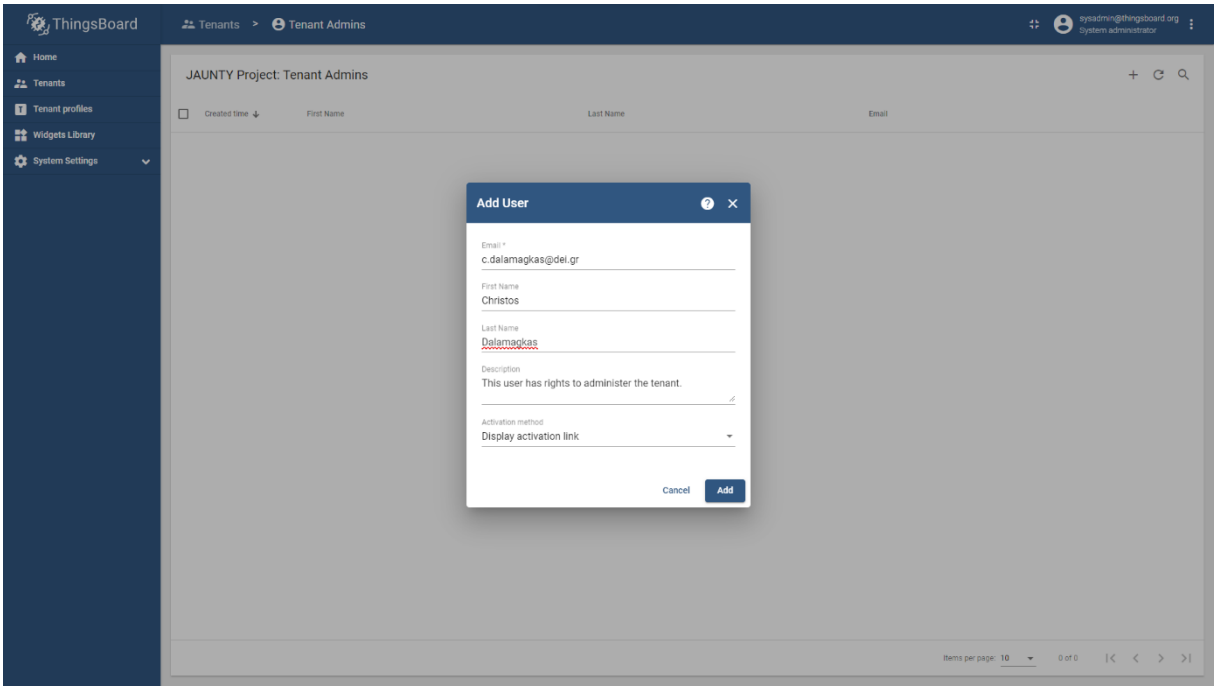


Figure 10: Add user to tenant

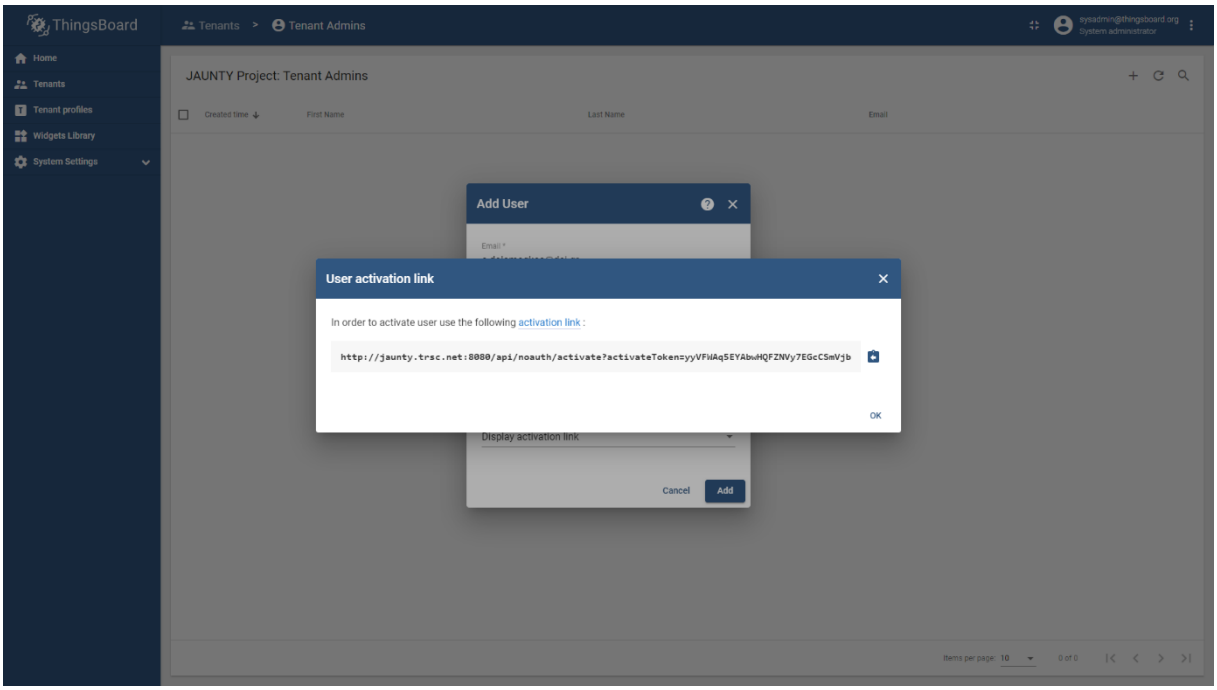


Figure 11: User activation link

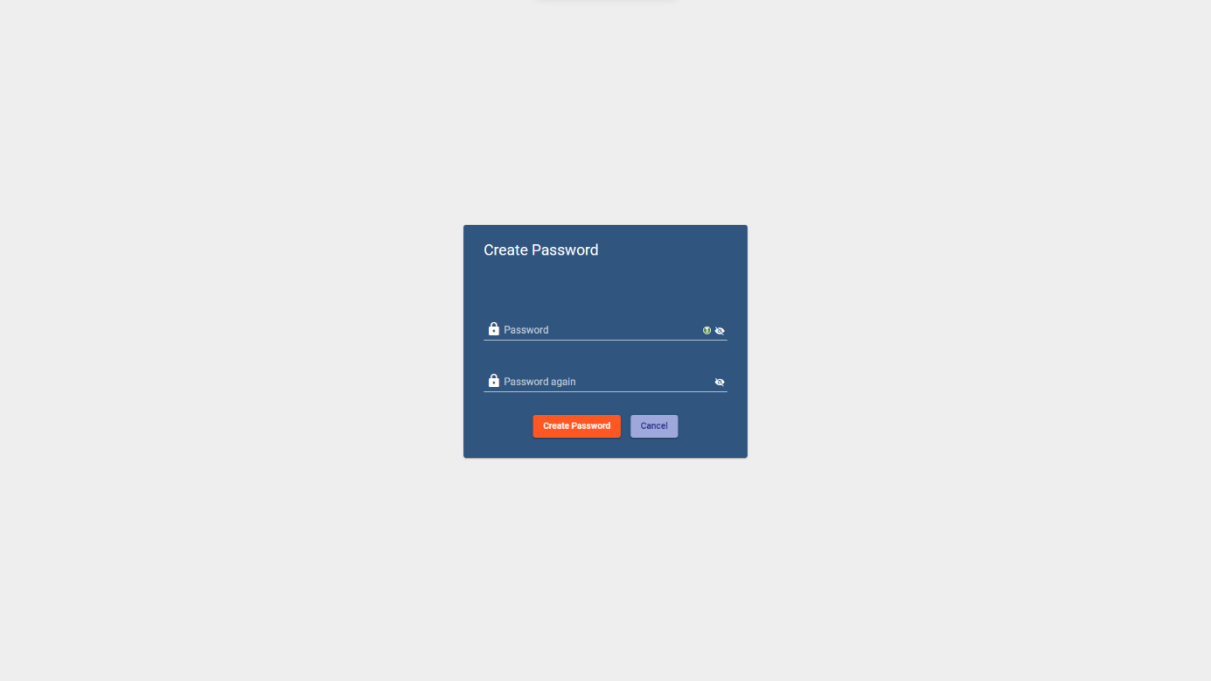


Figure 12: Password Initialisation

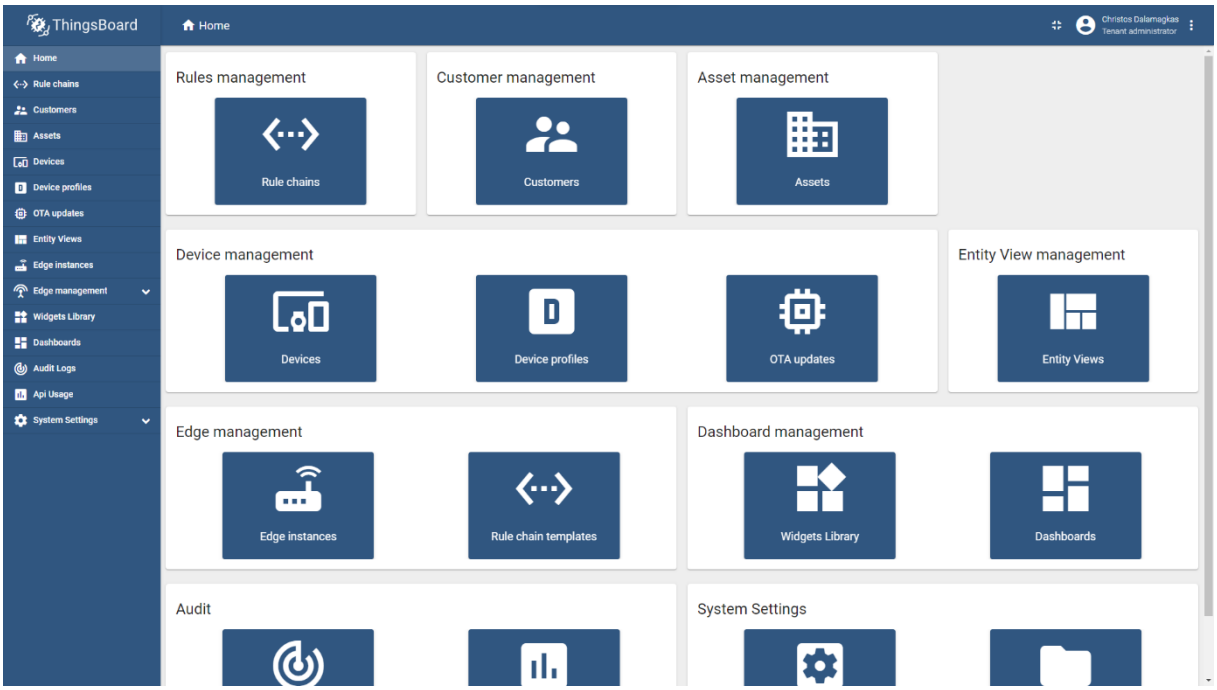


Figure 13: Homepage of the tenant administrator

Step 3: Add customer

Customer is another business entity (e.g., organisation or company), which is provisioned by ThingsBoard. A customer may purchase IoT devices and assets that are owned by one or multiple tenants. Therefore, it can be said that the tenant is the provider of metering services and devices, whereas the customer is the end user of those services. So, in our case, the customer will be the entity that will access the ThingsBoard measurements and dashboards, which are to be provided by the tenant previously created.

To access the Customers menu, click on the "Customers" option, located on the left main sidebar, as depicted in Figure 14. To create a customer entity, click on the + icon (annotated in Figure 14).

Complete the fields for the new customer appropriately as depicted in Figure 15. Then, click "Add" to create the new customer. The customer menu will appear as illustrated in Figure 16.

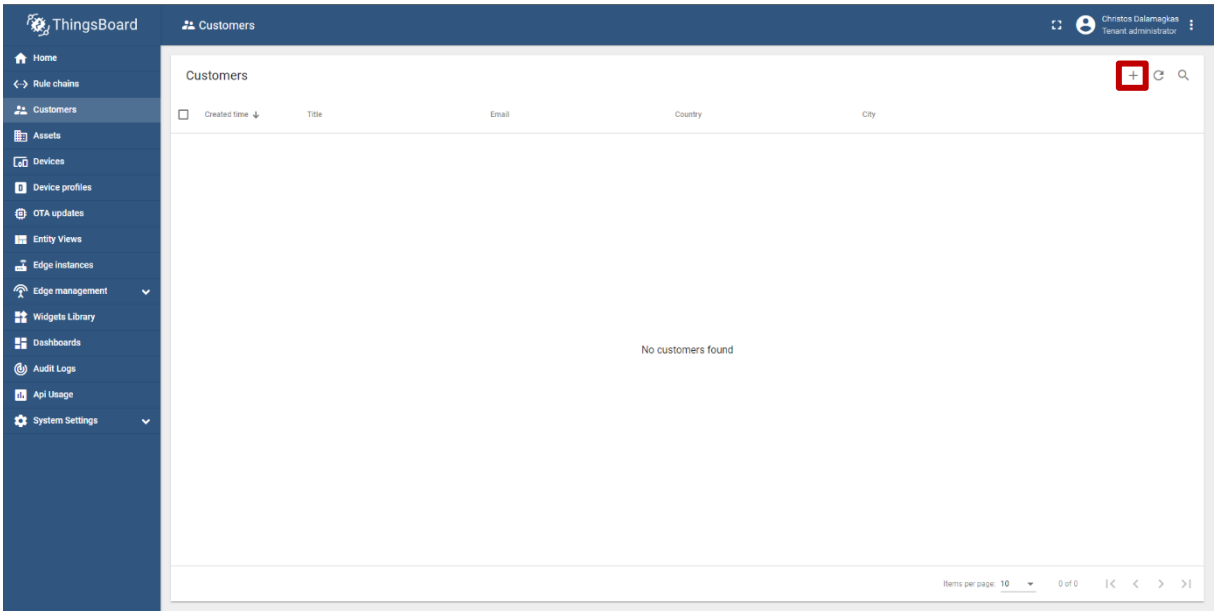


Figure 14: ThingsBoard - The Customers menu

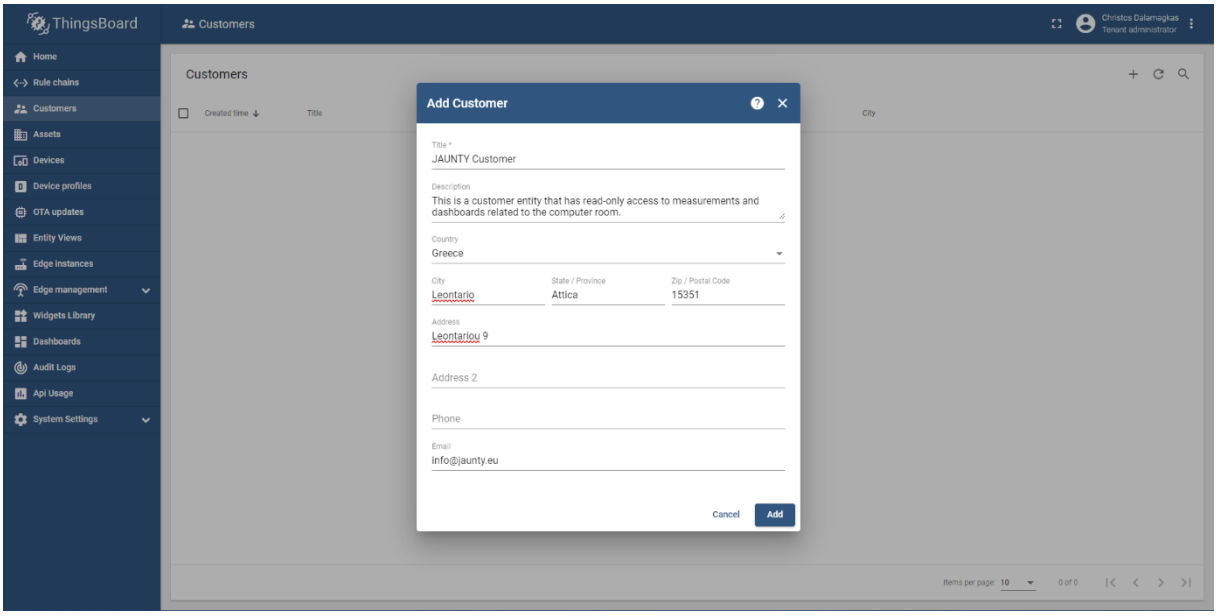


Figure 15: New Customer menu

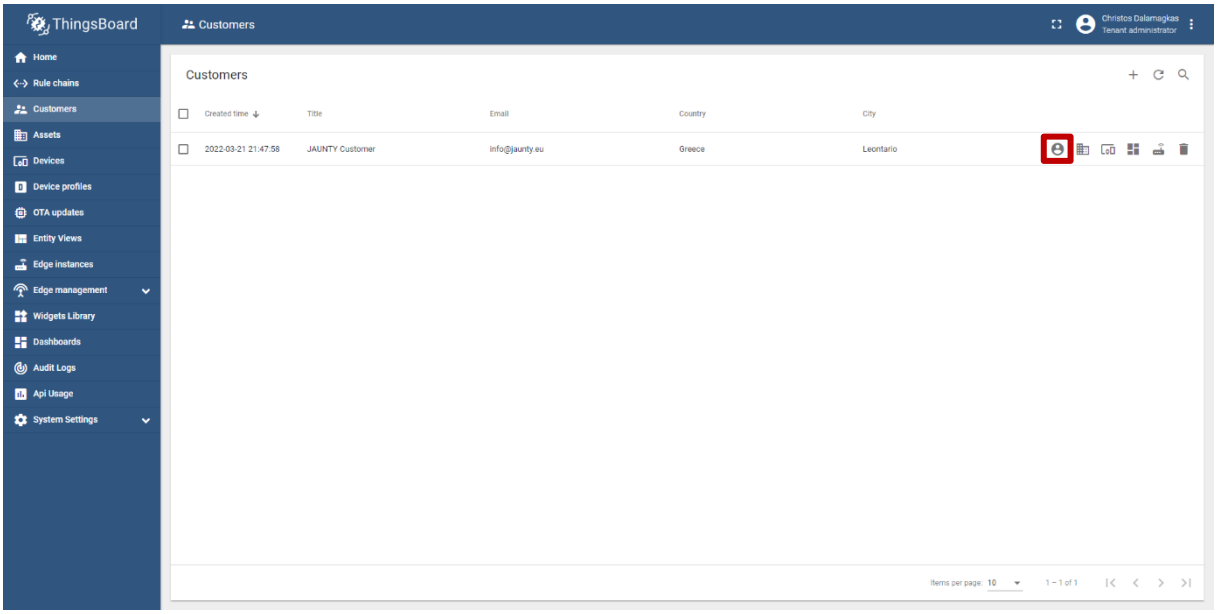


Figure 16: The Customers menu after the addition

Step 4: Add user to customer

In a similar manner with the tenant, a user must be added to the customer entity. This user will be able to login to ThingsBoard and have the rights that are assigned to the customer entity.

To add a user, click on the user icon as annotated in Figure 16. The customer user's menu is depicted in Figure 17. Click on the + icon to add the new user.

Fill the new user form as depicted in Figure 18. As with the tenant user, make sure you select the "Display activation link" option on the Activation method.

After clicking on "Add", open the activation link in a new browser tab or window, and initialise the password of the new user.

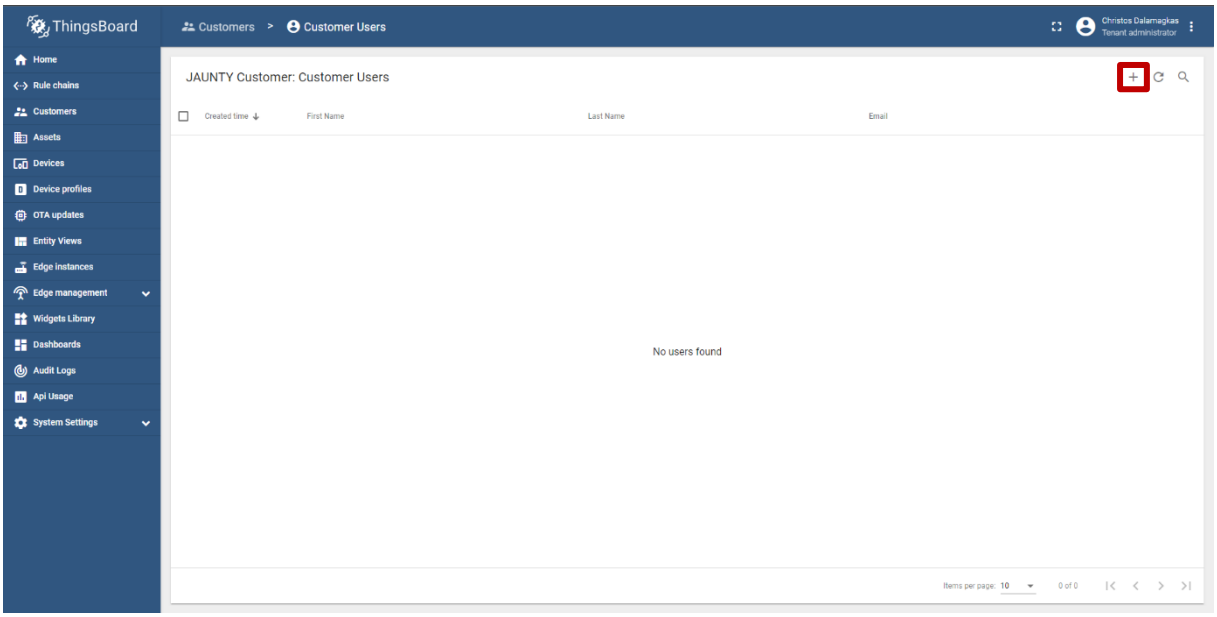


Figure 17: Customer users

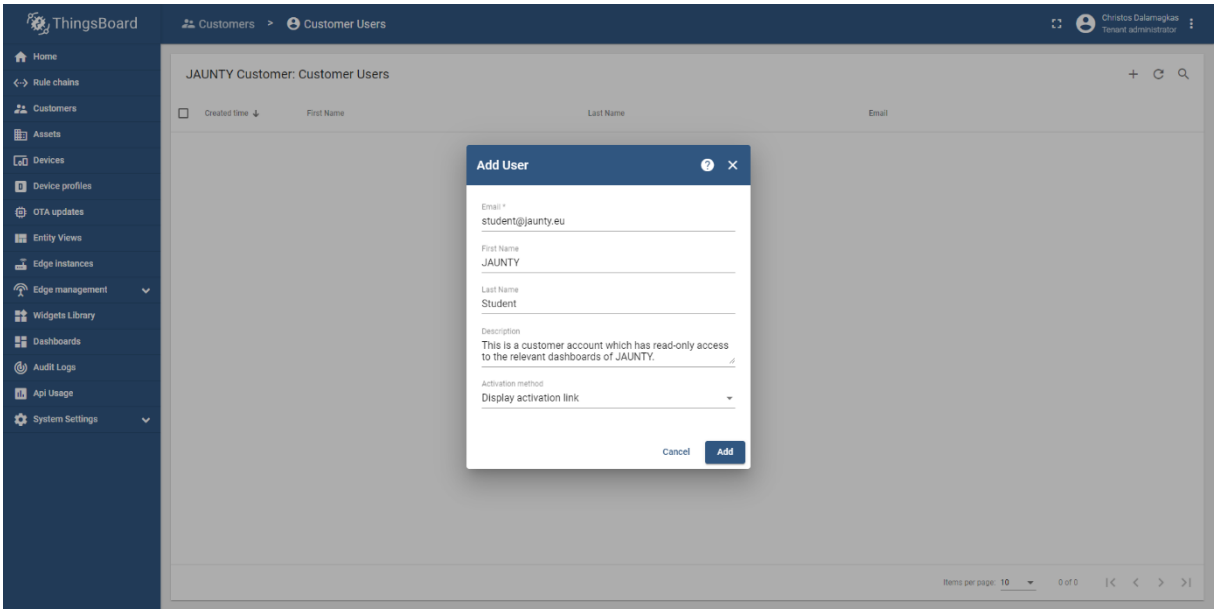


Figure 18: New Customer user menu

Step 5: Add asset

In the context of ThingsBoard, asset is an abstract IoT entity that encompass one or multiple IoT devices. For example, the asset could be a factory, a vehicle, or a laboratory, amongst other. For this lab, we are going to create one asset that represents the computer room that we are monitoring.

To create a new asset, navigate to the “Assets” menu by clicking on the “Assets” option on the left sidebar menu. The Assets menu is depicted in Figure 19. Click on the + icon of that menu to create the new asset. Fill the form as illustrated in Figure 20 and click “Add”. All assets are depicted in Figure 21.

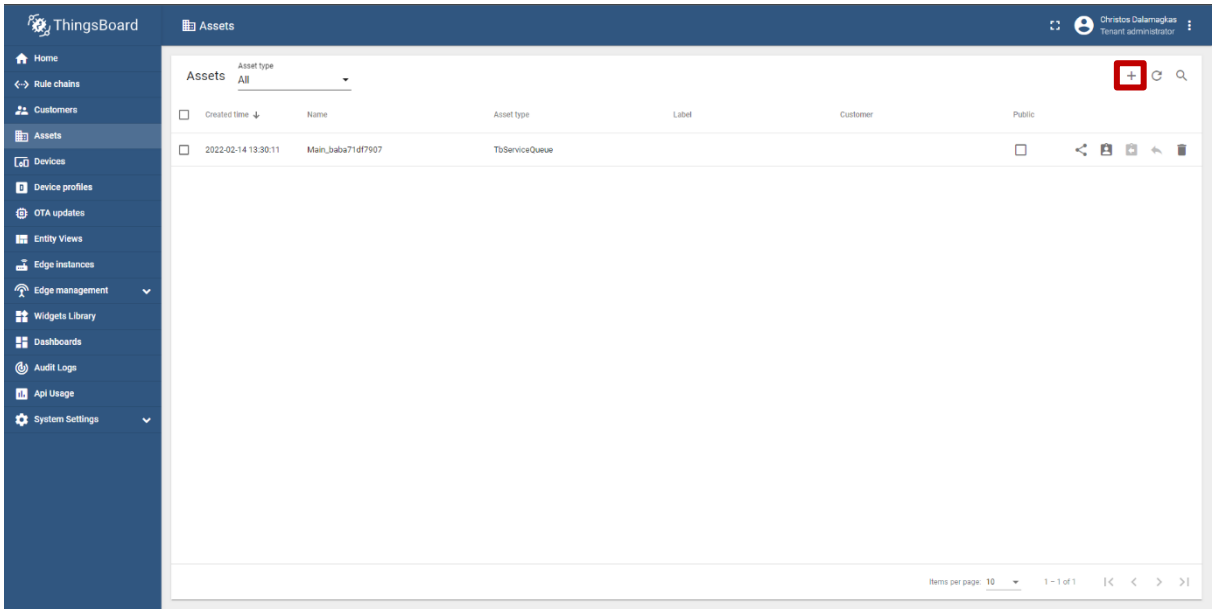


Figure 19: The Assets menu

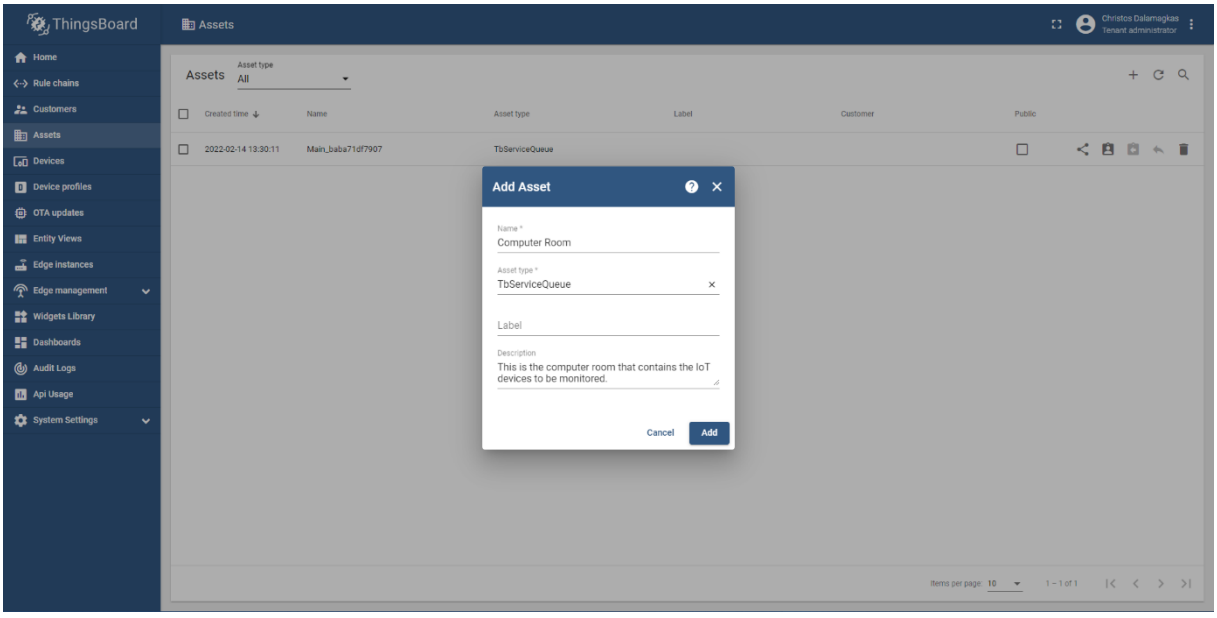


Figure 20: New asset menu

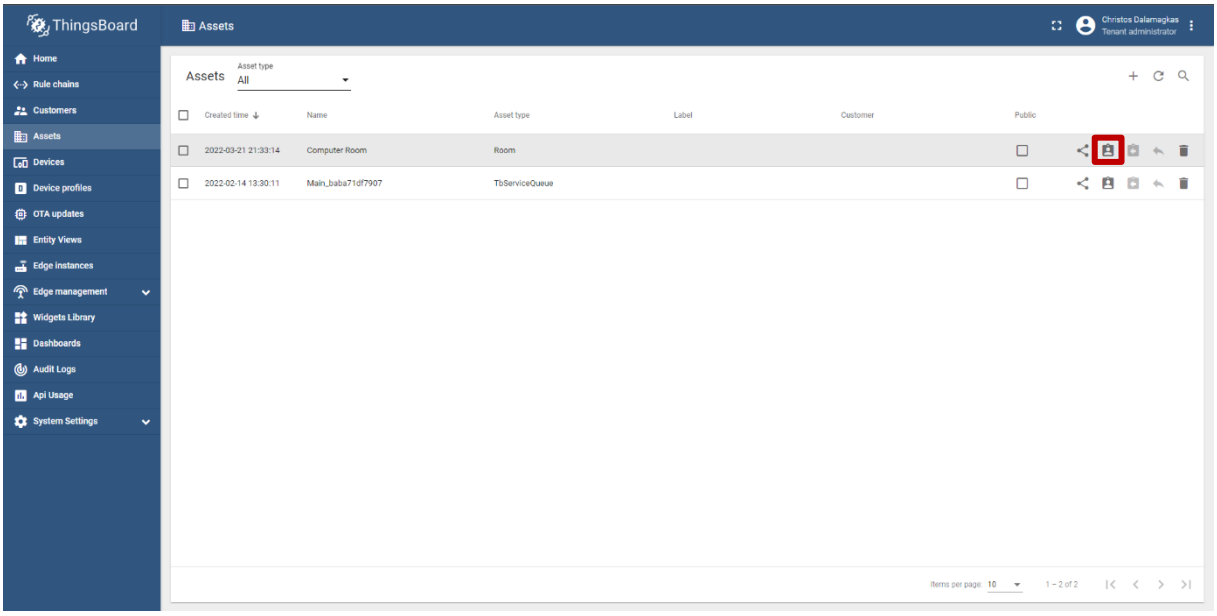


Figure 21: All assets including the new one

Step 6: Assign asset to customer

To assign the asset to the customer, click on the client icon of the asset, annotated in Figure 21. Then, click on the empty Customer field and choose the “JAUNTY Customer” option, as depicted in Figure 22, and click “Assign”. The assignment is depicted in Figure 23.

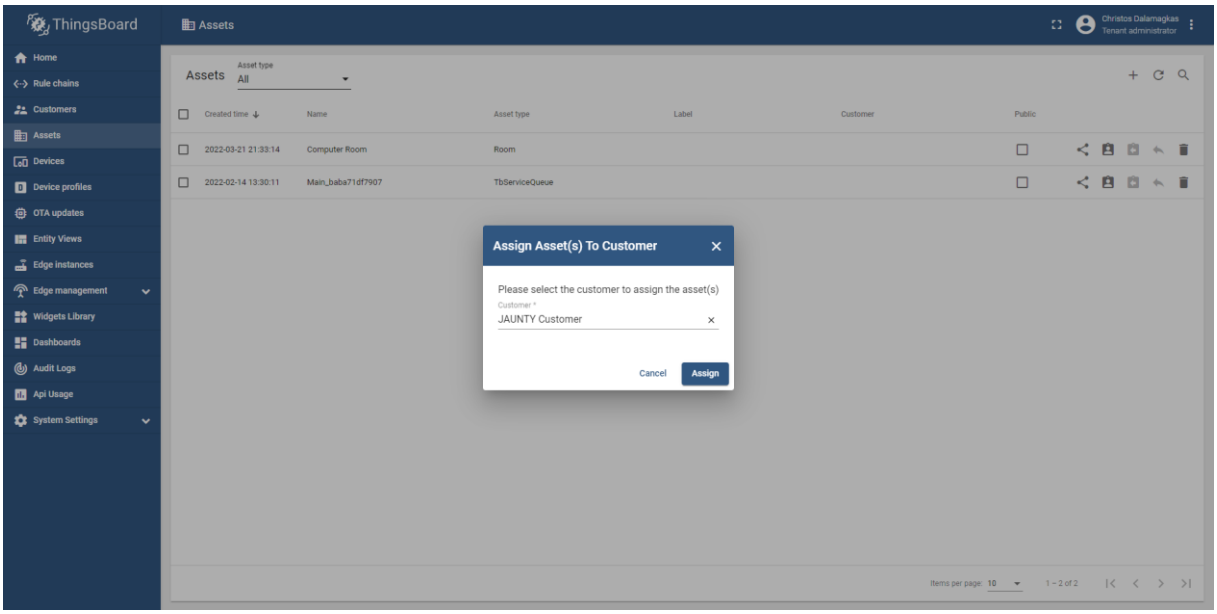


Figure 22: Assign asset to customer

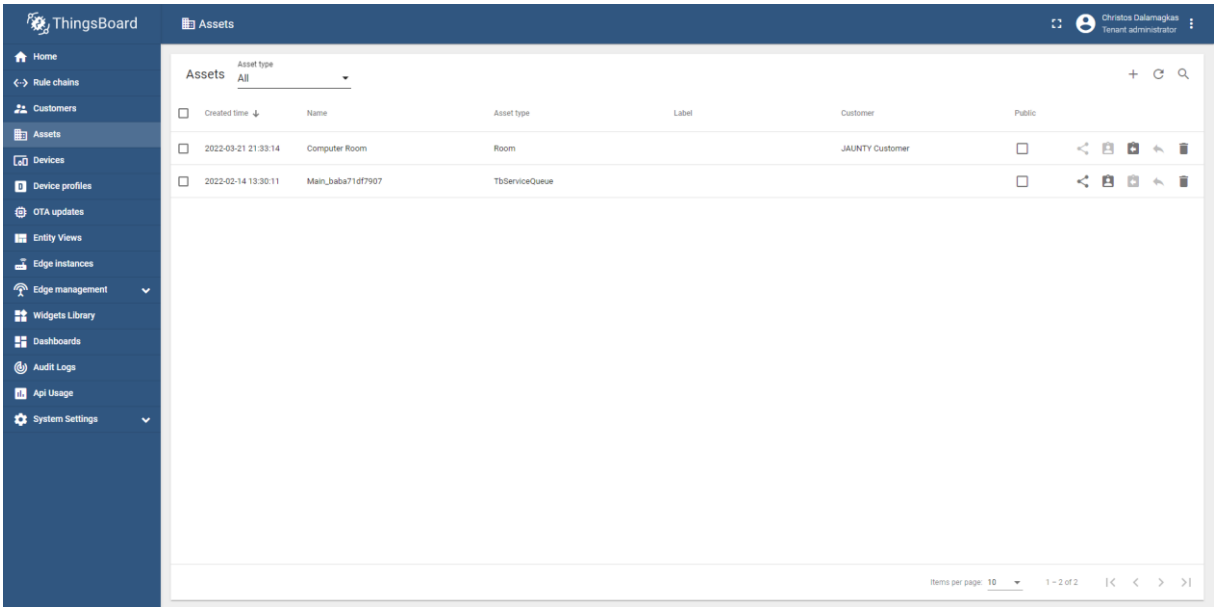


Figure 23: Customer/Asset assignment completed

5. Exercise 2: Add devices to ThingsBoard and collect telemetry via CoAP

As a tenant administrator, in this exercise, two device objects will be provisioned via the ThingsBoard Graphical User Interface (GUI), corresponding to the SHT40 and the APC UPS. The Device is the most basic IoT entity of ThingsBoard, under which telemetry data is matched. Moreover, every device bears unique credentials, which could be Access Token, X.905 certificates, or MQTT credentials. In this exercise, each device will have its unique access token, which will be used as unique identifier when posting telemetry data. Finally, the Python scripts will be expanded accordingly to post telemetry to ThingsBoard via CoAP.

Step 1: Set a new device for the APC UPS

To create a new device entity, open the Devices menu by clicking on the “Devices” option of the left main sidebar menu. The Devices menu is illustrated in Figure 24. To add a device, click on the “+” button, annotated in Figure 24. This will pop-up the “Add new device” dialogue.

In the “Add new device” dialogue depicted in Figure 25, provide the name of the new device (e.g., apcups) and click on the “Next: Credentials” button annotated in the same figure.

On the credentials page, also depicted in Figure 26, enable the “Add credentials” option and choose the “Access token” credential type from the dropdown menu. Provide a random string composed of small/caps letters and numbers as the access token and click on the “Next: Customer” button annotated in the same figure. In this view, assign the JAUNTY Customer to the device, as depicted in Figure 27, and click “Add”. The new device is depicted in Figure 28.

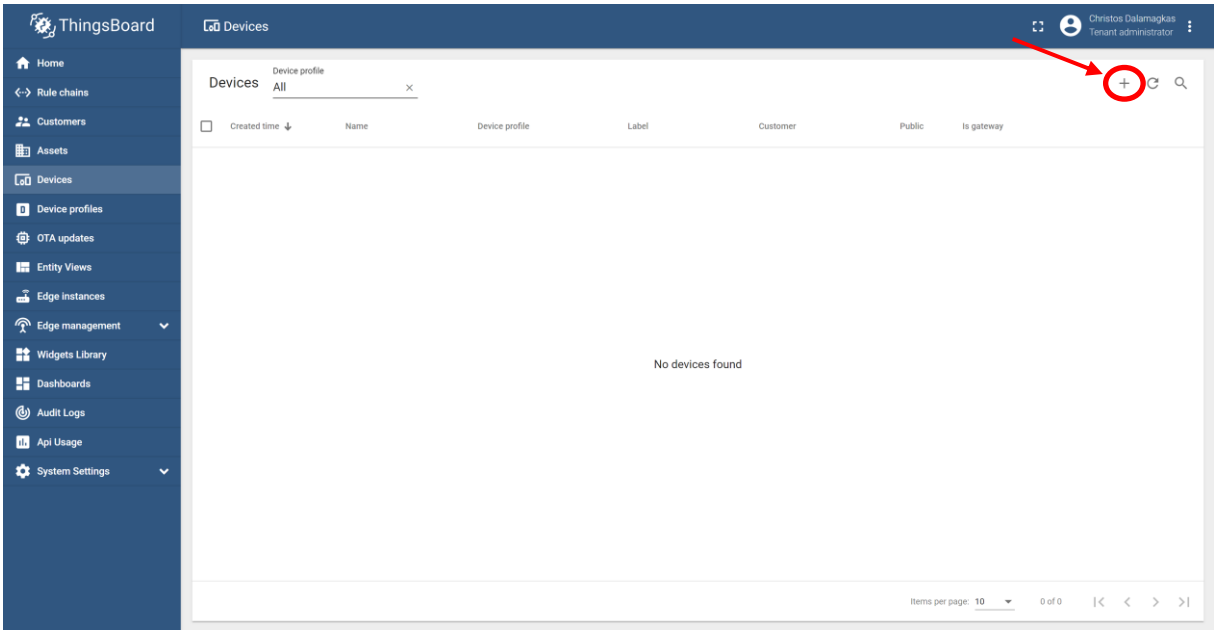


Figure 24: The Devices menu

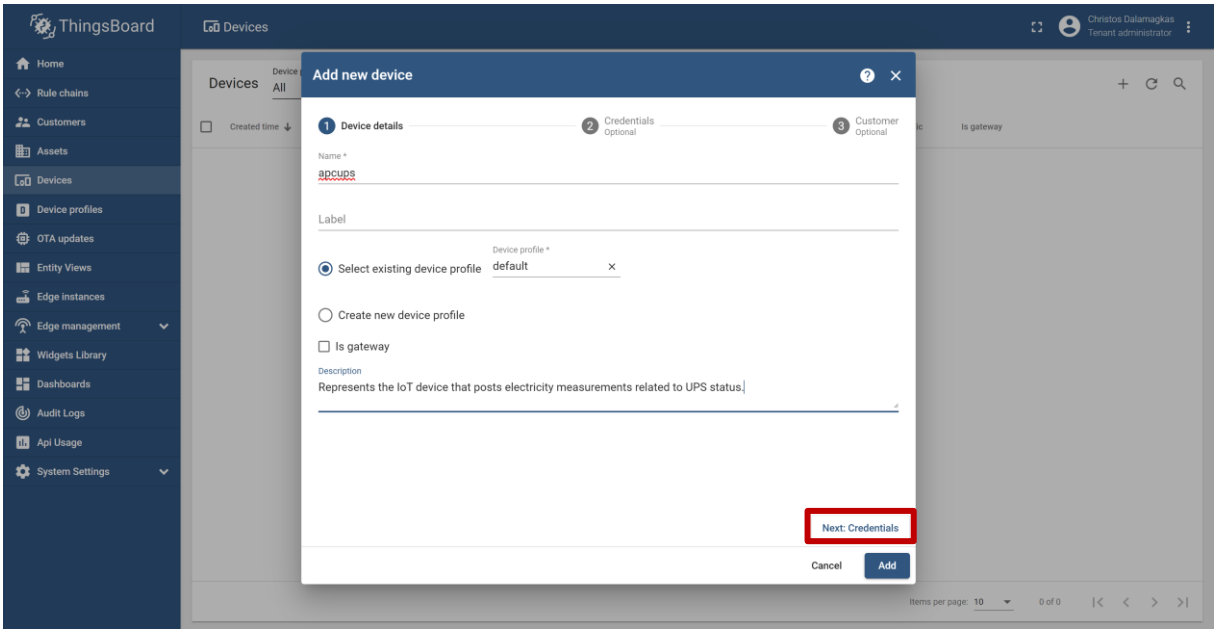


Figure 25: New device for the APC UPS

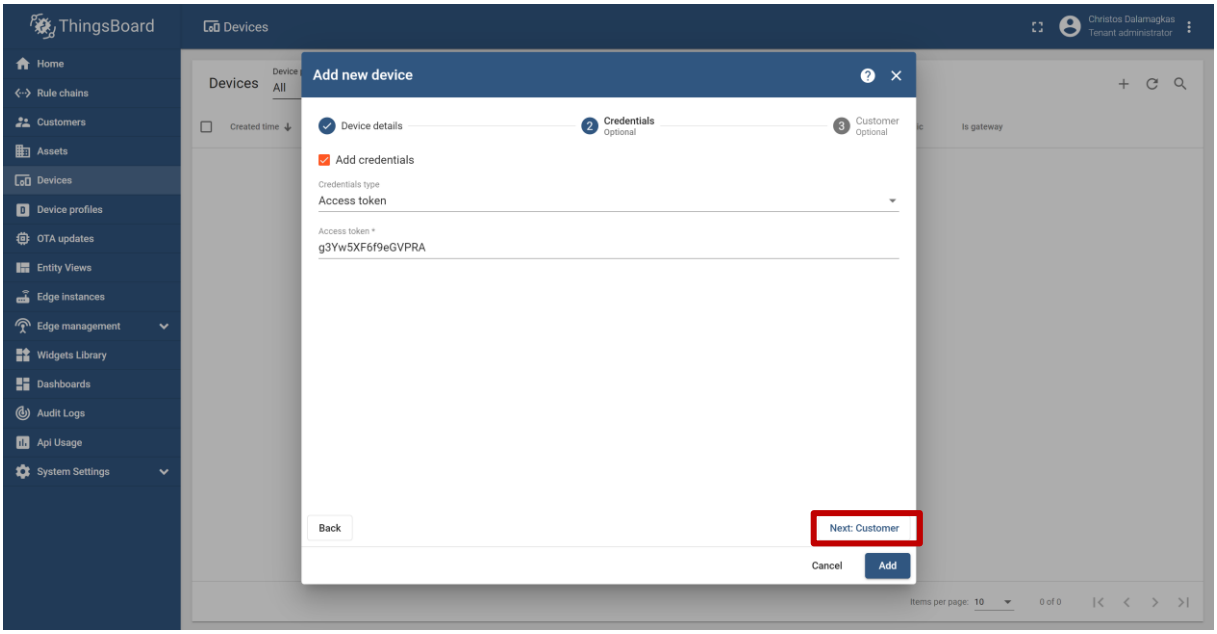


Figure 26: Credentials for the new device

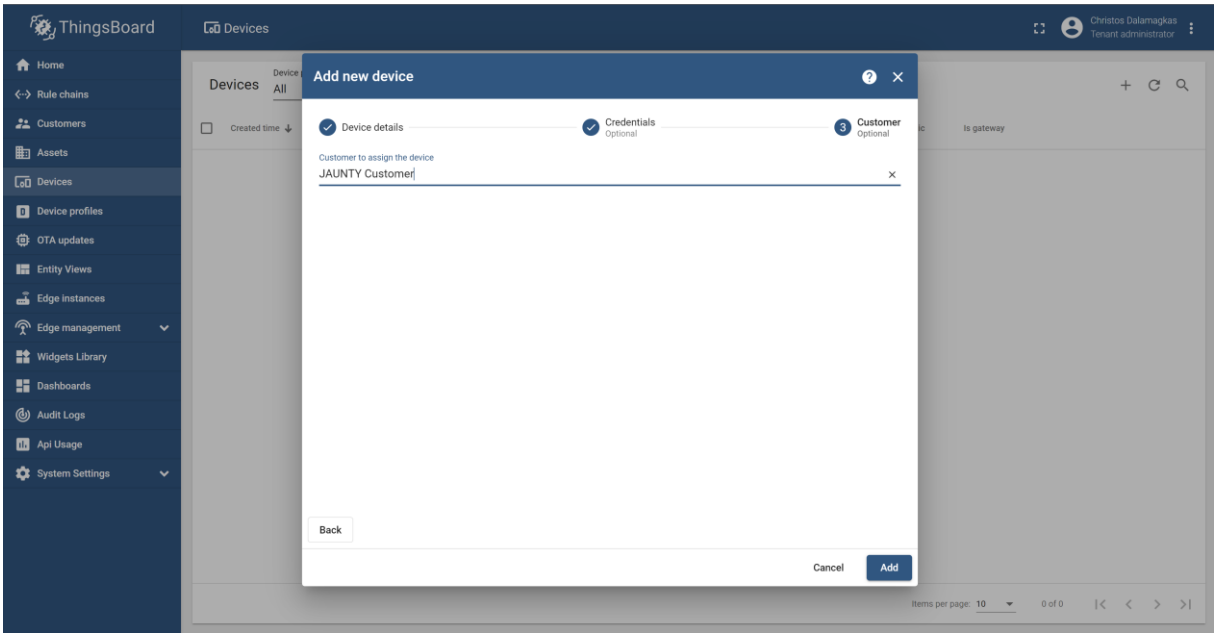


Figure 27: Assign customer to device

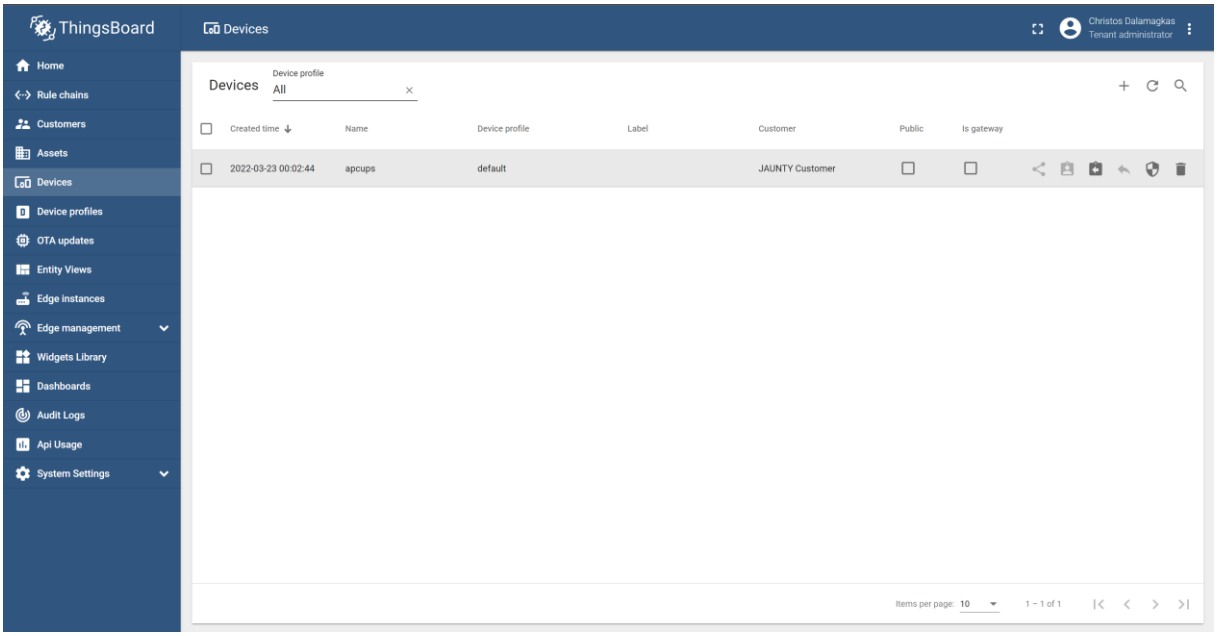


Figure 28: The new entry in Devices menu

Step 2: Post UPS measurements to ThingsBoard via CoAP

The source code of Exercise 1 is expanded appropriately in this step, to post the retrieved telemetry to ThingsBoard. The new source code is provided bellow. The new parts of the code are in red colour.

```
1  from pymodbus.client.sync import ModbusTcpClient
2  import json
3  import asyncio
4  from aiocoap import *
5
6  ANALOG_DATA_ADDRESSES = (130, 135, 140, 142, 144, 147, 151)
7  DELAY = 1
8  THINGSBOARD_ACCESS_TOKEN = 'g3Yw5XF6f9eGVPRA'
9
10 def holding_register_to_string(hr_response):
11     final_string = ""
12     for item in hr_response.registers:
13         binary_string = bin(item).replace('0b',"").zfill(16)
14         first_letter = int(binary_string[:8], 2)
15         second_letter = int(binary_string[8:], 2)
16         final_string = final_string + chr(first_letter) + chr(second_letter)
17     return final_string.strip()
18
19 async def postCOAP(payload):
20     context = await Context.create_client_context()
21     request = Message(code=Code.POST, payload=payload.encode(),
   uri="coap://THINGSBOARD_IP/api/v1/" + THINGSBOARD_ACCESS_TOKEN +"/telemetry")
22     response = await context.request(request).response
23     print('Result: %s\n%r'%(response.code, response.payload))
24
25 async def main():
27     UNIT_ID = 1
29     slave = ModbusTcpClient('192.168.86.225', port=502)
30
31     if not slave.is_socket_open():
```

```
32          slave.connect()
33      device_SN = slave.read_holding_registers(564, count=8, unit=UNIT_ID)
34      slave.close()
35      device_SN = holding_register_to_string(device_SN)
36
37      if not slave.is_socket_open():
38          slave.connect()
39      device_name = slave.read_holding_registers(596, count=8, unit=UNIT_ID)
40      slave.close()
41      device_name = holding_register_to_string(device_name)
42
43      while True:
44          response_holding_registers = [0]*len(ANALOG_DATA_ADDRESSES)
45
46          for i in range(len(ANALOG_DATA_ADDRESSES)):
47              if not slave.is_socket_open():
48                  slave.connect()
49
50              response_holding_registers[i] =
    slave.read_holding_registers(ANALOG_DATA_ADDRESSES[i], unit=UNIT_ID)
51              slave.close()
52
53              if response_holding_registers[i].isError():
54                  print('A Modbus exception occurred. Terminating..')
55                  exit(-1)
56
57          InputStatus_BF_Decimal = response_holding_registers[5].registers[0]
58
59          record_json = {
60              "device": {
61                  "name": device_name,
62                  "sn": device_SN
63              },
64              "StateOfChargePct": response_holding_registers[0].registers[0],
65              "BatteryTemperature": response_holding_registers[1].registers[0],
66              "OutputCurrentAC": response_holding_registers[2].registers[0],
67              "OutputVoltageAC": response_holding_registers[3].registers[0],
68              "OutputFrequency": response_holding_registers[4].registers[0],
69              "BypassInputStatus_BF": {
70                  "InputAcceptable": InputStatus_BF_Decimal & 1 != 0,
71                  "PendingAcceptable": InputStatus_BF_Decimal & 2 != 0,
72                  "VoltageTooLow": InputStatus_BF_Decimal & 4 != 0,
73                  "VoltageTooHigh": InputStatus_BF_Decimal & 8 != 0,
74                  "Distorted": InputStatus_BF_Decimal & 16 != 0,
75                  "BoostVoltage": InputStatus_BF_Decimal & 32 != 0,
76                  "TrimVoltage": InputStatus_BF_Decimal & 64 != 0,
77                  "FrequencyTooLow": InputStatus_BF_Decimal & 128 != 0,
78                  "FrequencyTooHigh": InputStatus_BF_Decimal & 256 != 0,
79                  "FreqAndPhaseNotLocked": InputStatus_BF_Decimal & 512 != 0,
80                  "PhaseDeltaOutOfRange": InputStatus_BF_Decimal & 1024 != 0,
81                  "NeutralNotConnected": InputStatus_BF_Decimal & 2048 != 0,
82              },
83              "InputVoltage": response_holding_registers[6].registers[0],
84          }
85          record_json = json.dumps(record_json)
86
87          await postCOAP(record_json)
88
89          await asyncio.sleep(DELAY)
90
91          continue
92
93  if __name__ == "__main__":
```

```
94      asyncio.run(main())
95
```

An additional function has been defined in lines 18-22, which undertakes to post the JSON string of measurements via the CoAP protocol. In particular, the aiocoap library is utilised, which implements the CoAP protocol and uses the native asyncio methods of Python 3 to facilitate concurrent operations. Since CoAP follows the server/client model, a request is prepared via the Message() function of aiocoap in line 21. The various messages exchanged by CoAP are distinguished using message codes; the GET code of a CoAP request is sent to retrieve a remote web resource, while POST code is used to send payload information to the CoAP server. In our case, the POST code is used. Moreover, the payload is provided, however, is encoded (converted to bytes string) since this is a prerequisite of the function. Finally, the URI is specified according to ThingsBoard. Please note that only the device access token is specified in the URI (THINGSBOARD_ACCESS_TOKEN corresponds to a constant defined earlier in the script), this is enough to identify the device itself and post the telemetry on that device. Finally, the request is sent in line 22.

Step 3: Create a new device for the SHT40 sensor

Based on step 1 of this exercise, try to create a new device on ThingsBoard that represents the SHT40 sensor.

Step 4: Post temperature and humidity measurements to ThingsBoard via CoAP

Based on step 2 of this exercise, modify the Python source code of Exercise 1 step 2, so that the humidity and temperature measurements of the SHT40 sensor are posted to ThingsBoard.

6. Exercise 3: Data pre-processing with the ThingsBoard Rule Engine and Generating Alarms

An essential part of ThingsBoard is the Rule Engine, an event-based system that allows the definition of custom actions and processing of incoming messages. In particular, the tenant administrator can define how ThingsBoard should process, filter, and transform incoming messages (e.g., telemetry) originated by IoT devices. Moreover, customised notifications and alarms can be raised by the Rule Engine, based on the values of the telemetry. The following terminology applies in the context of the Rule Engine:

- Rule Node: This is the most basic component of the Rule Engine, which processes incoming messages and generates some output. The rule node can filter, enrich, transform incoming messages, perform action, or communicate with external systems.
- Rule Node Relation: This is the type of association between two nodes. For example, node 1 provides output to node 2 if an operation is successful or if the incoming message is telemetry. The condition on which node 1 provides output to node 2 is called “tag”.
- Rule Chain: A logical group of rule nodes along with their relations. The Root Rule Chain is pre-installed on ThingsBoard and configured to handle all incoming messages.

In this exercise, you are going to modify the Root Rule Chain to process and transform the incoming telemetry as well as to generate alarms based on the values received.

Step 1: View the current telemetry (before changing the rules)

Navigate to the telemetry of the UPS by opening the “Devices” menu and clicking on the UPS device. After that, click on the “Latest telemetry” tab. This tab is depicted in Figure 29 (Please note that some minor differences on the depicted keys are expected). According to the UPS Modbus map, some values need to be divided by a specific scale value, e.g., the StateOfChargePct needs to be divided by 512 to retrieve the human-readable value (100). This information is provided in Table 1. Moreover, the BypassInputStatus_BF object needs to be parsed and separate the nested keys. This is because other components of ThingsBoard (e.g., widgets) does not process JSON objects.

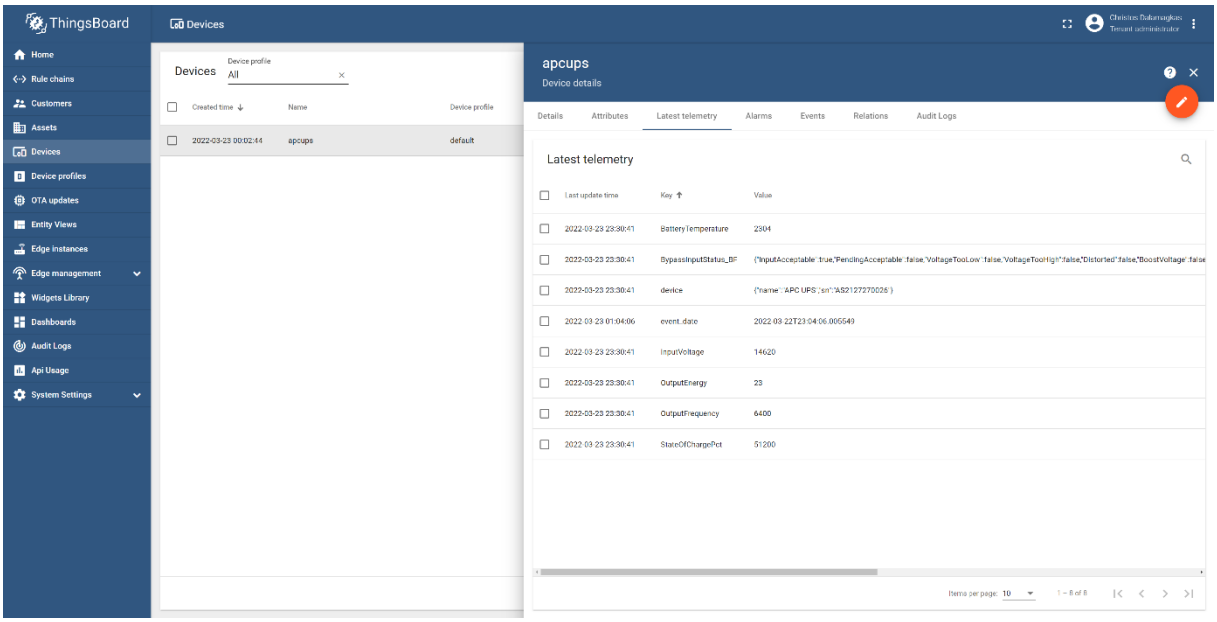


Figure 29: The latest telemetry of the APC UPS

Step 2: Examine the Root Rule Chain

To access the Rule Engine, click on the “Rule chains” option of the main left sidebar menu. The rule engine is depicted in Figure 30. All rule chains are presented in this menu, along with the root rule chain, which is the default and pre-configured with any ThingsBoard installation.

To open the root rule chain, click on the chain of Figure 30 and subsequently on the “Open rule chain” option highlighted in Figure 31.

The contents of the root rule chain are illustrated in Figure 32. Starting point of the chain is the input node which outputs all incoming messages. The first node processes alarm rules that are attached to the device profile (not subject of this lab). Since we do not modify the first node, its input remains intact and is forwarded to a message type switch. Depending on the message type, the message is forwarded to the appropriate node for specialised processing. In our case, we care about the “Post telemetry” option. As depicted, all messages that include telemetry are directly saved to the database. However, this is not desirable in our case since the measurements have the wrong format. We need to add an intermediary node which transforms the telemetry before saving it to the database.

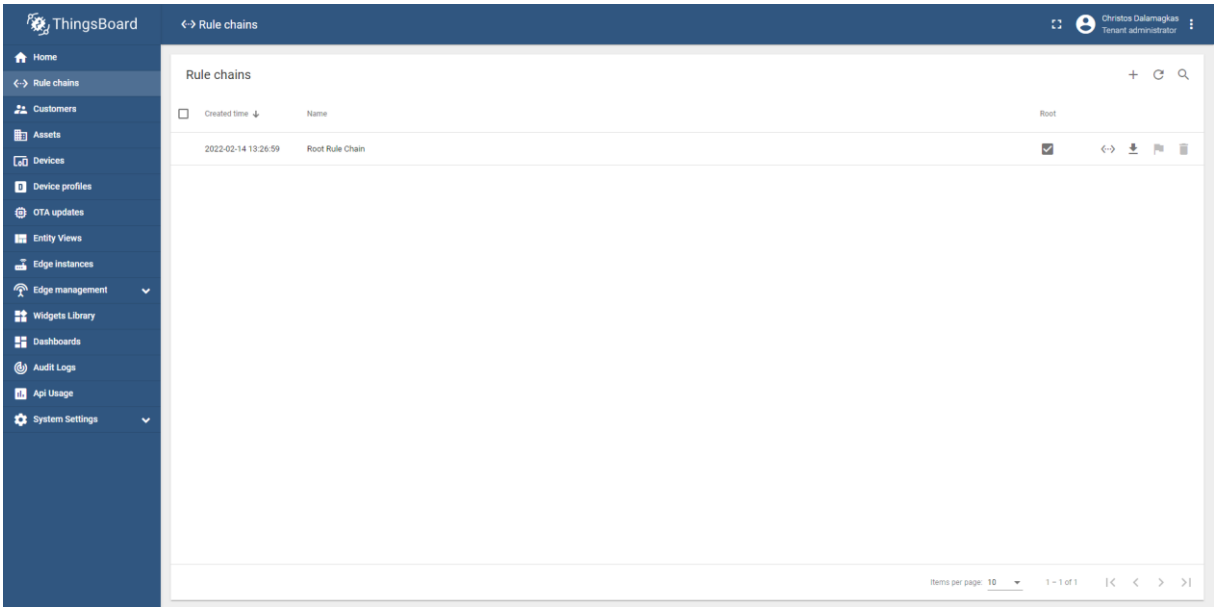


Figure 30: The ThingsBoard Rule Engine

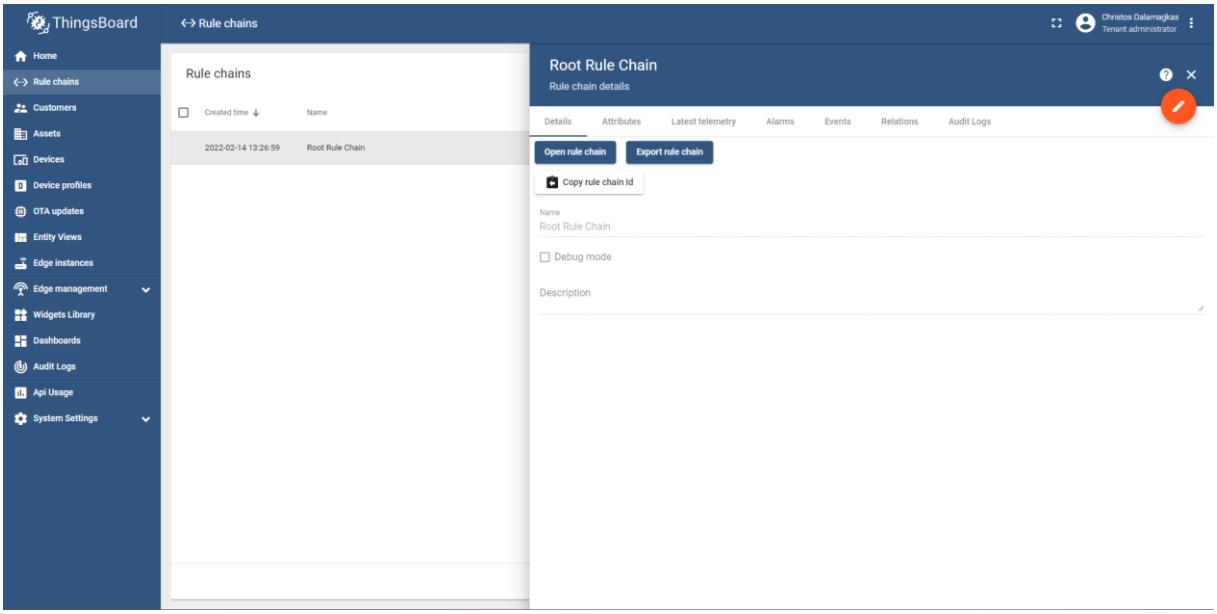


Figure 31: Opening the Root Rule Chain

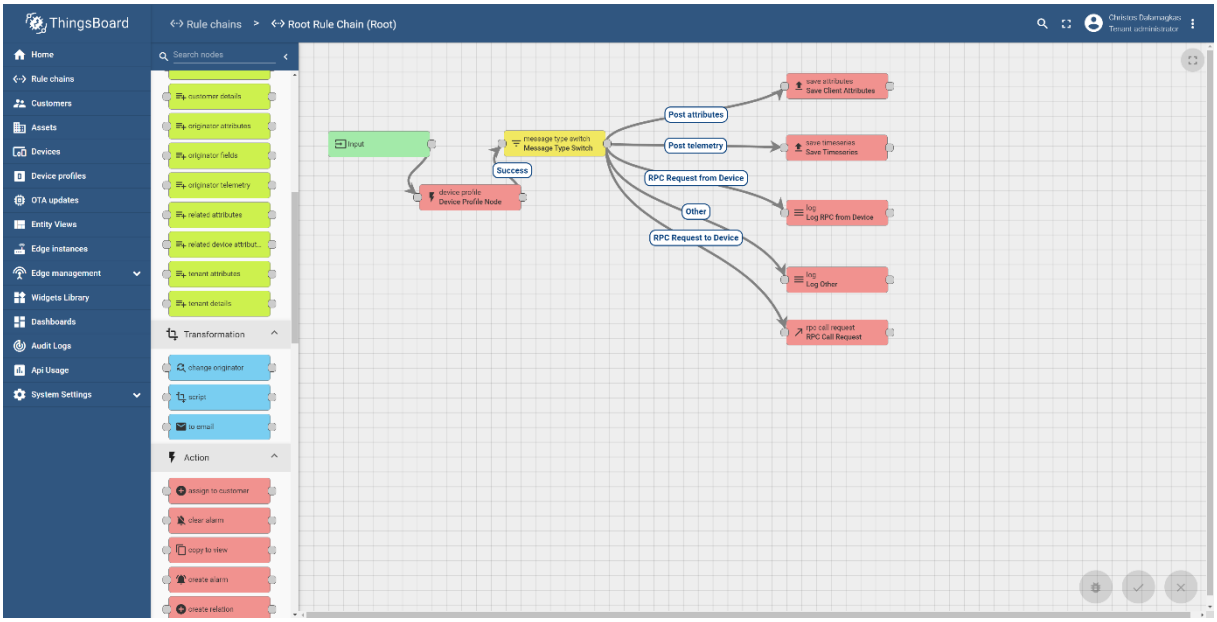


Figure 32: The Root Rule Chain

Step 3: Add and Connect a Transformation Node

The Rule Engine provisions various type of nodes identified by their colour. Filter nodes route messages based on conditions, enrichment nodes add extra information on transient messages, transformation nodes change the payload of the message, and action nodes apply specific actions using the incoming messages. Finally, the external nodes undertake the communication with external systems. Since we need to apply formulas and correct the incoming messages, we are going to create a transform node.

To this aim, drag a “script” transformation node from the left sidebar and drop it freely on the dashboard. This will open the menu illustrated in Figure 33.

Specify a name for this node (e.g., TransformUPSMeasurements) and provide the following source code in the transform function (written in JavaScript):

```
1  if (typeof msg.StateOfChargePct !== 'undefined') {
2      msg.StateOfChargePct = msg.StateOfChargePct / 512;
3  }
4
5  if (typeof msg.BatteryTemperature !== 'undefined') {
6      msg.BatteryTemperature = msg.BatteryTemperature / 128;
7  }
8
9  if (typeof msg.OutputCurrentAC !== 'undefined' && typeof msg.OutputVoltageAC
   !== 'undefined') {
10     msg.OutputCurrentAC = msg.OutputCurrentAC / 32;
11     msg.OutputVoltageAC = msg.OutputVoltageAC / 64;
12     msg.PowerConsumption = msg.OutputCurrentAC * msg.OutputVoltageAC;
13 }
14
15 if (typeof msg.OutputFrequency !== 'undefined') {
16     msg.OutputFrequency = msg.OutputFrequency / 128;
17 }
18
19 if (typeof msg.InputVoltage !== 'undefined') {
20     msg.InputVoltage = msg.InputVoltage / 64;
21 }
22
23 if (typeof msg.BypassInputStatus_BF !== 'undefined') {
24     msg.InputAcceptable = msg.BypassInputStatus_BF.InputAcceptable;
25     msg.PendingAcceptable= msg.BypassInputStatus_BF.PendingAcceptable;
26     msg.VoltageTooLow = msg.BypassInputStatus_BF.VoltageTooLow;
27     msg.VoltageTooHigh = msg.BypassInputStatus_BF.VoltageTooHigh;
28     msg.Distorted = msg.BypassInputStatus_BF.Distorted;
29     msg.BoostVoltage = msg.BypassInputStatus_BF.BoostVoltage;
30     msg.TrimVoltage = msg.BypassInputStatus_BF.TrimVoltage;
31     msg.FrequencyTooLow = msg.BypassInputStatus_BF.FrequencyTooLow;
32     msg.FrequencyTooHigh = msg.BypassInputStatus_BF.FrequencyTooHigh;
33     msg.FreqAndPhaseNotLocked =
   msg.BypassInputStatus_BF.FreqAndPhaseNotLocked;
34     msg.PhaseDeltaOutOfRange = msg.BypassInputStatus_BF.PhaseDeltaOutOfRange;
35     msg.NeutralNotConnected = msg.BypassInputStatus_BF.NeutralNotConnected;
36     msg.PoweringLoad = msg.BypassInputStatus_BF.PoweringLoad;
37     delete msg.BypassInputStatus_BF;
38 }
39
40 return {
41     msg: msg,
42     metadata: metadata,
43     msgType: msgType
44 };
```

The following transformation are taking place:

- The StateOfChargePct, BatteryTemperature, OutputFrequency, and InputVoltage are divided by the correct scale value, according to Table 1.

- A new measurement is calculated (PowerConsumption) by utilising existing measurements of voltage and current, which are first corrected too.
- The nested keys of BypassInputStatus_BF are defined as independent keys to be processed by other components of ThingsBoard. Subsequenty, the BypassInputStatus_BF is deleted since it is no longer needed.

Click “Add” to create the node. The result should be like the one depicted in Figure 34.

To appropriately connect the new code, drag the arrow pointing to save timeseries node, and drop it on the input of the new node. The result of this action is illustrated in Figure 35.

Finally, drag the output of the new node and drop on the input of the save timeseries node. This will suddenly open the menu to add tags on the new relationship, illustrated in Figure 36. Select “Success” and click “Add”. The final outcome of this step is depicted in Figure 37.

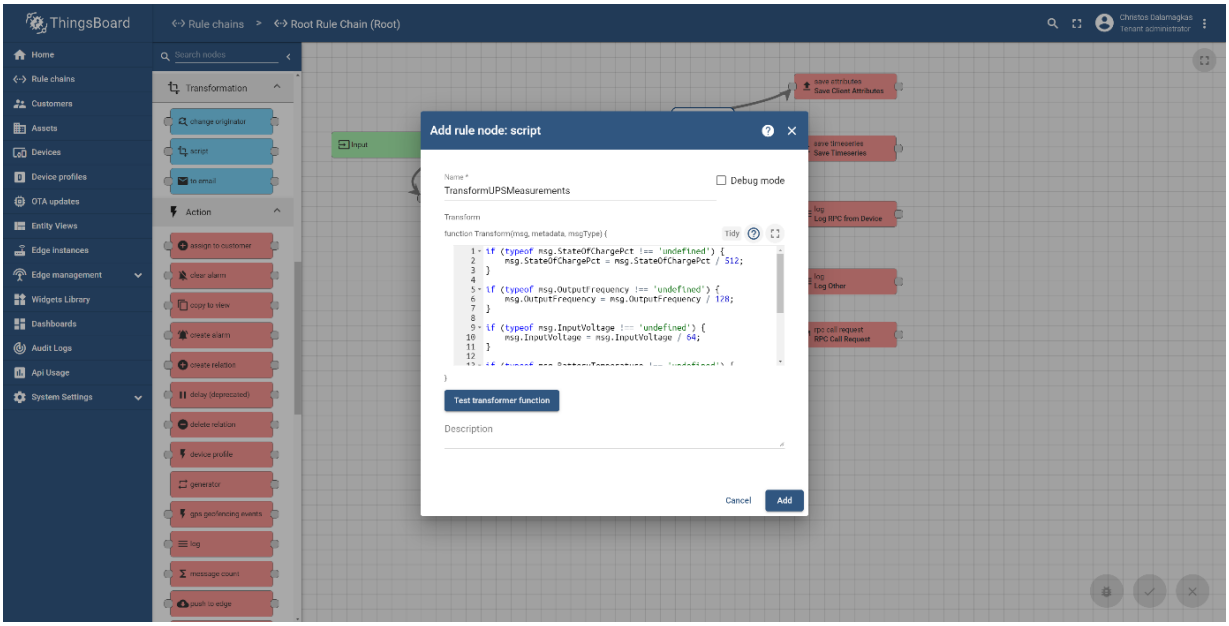


Figure 33: Adding a new script node

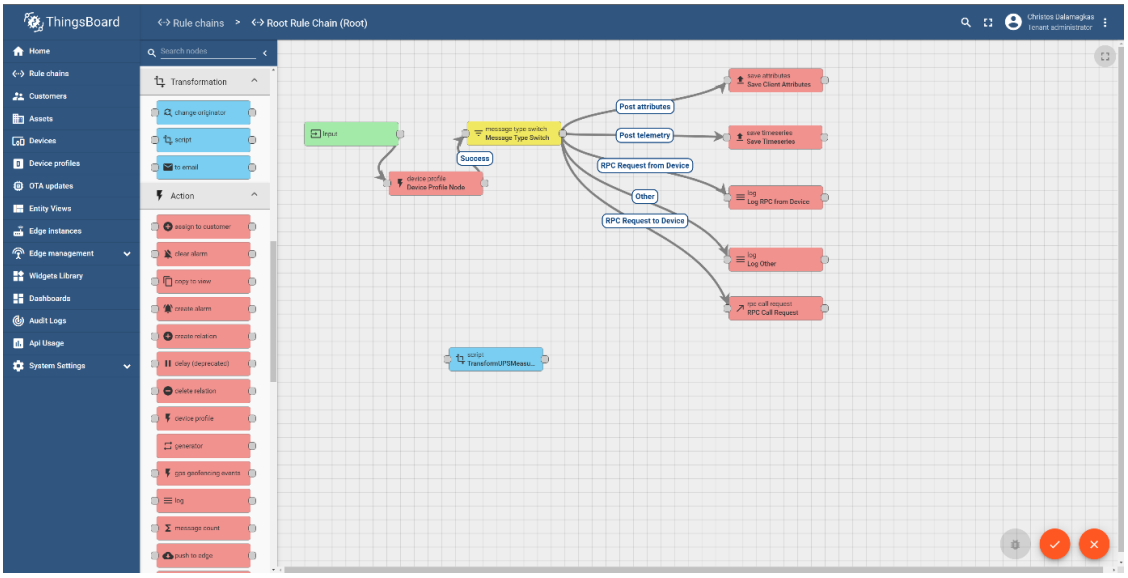


Figure 34: The new script node

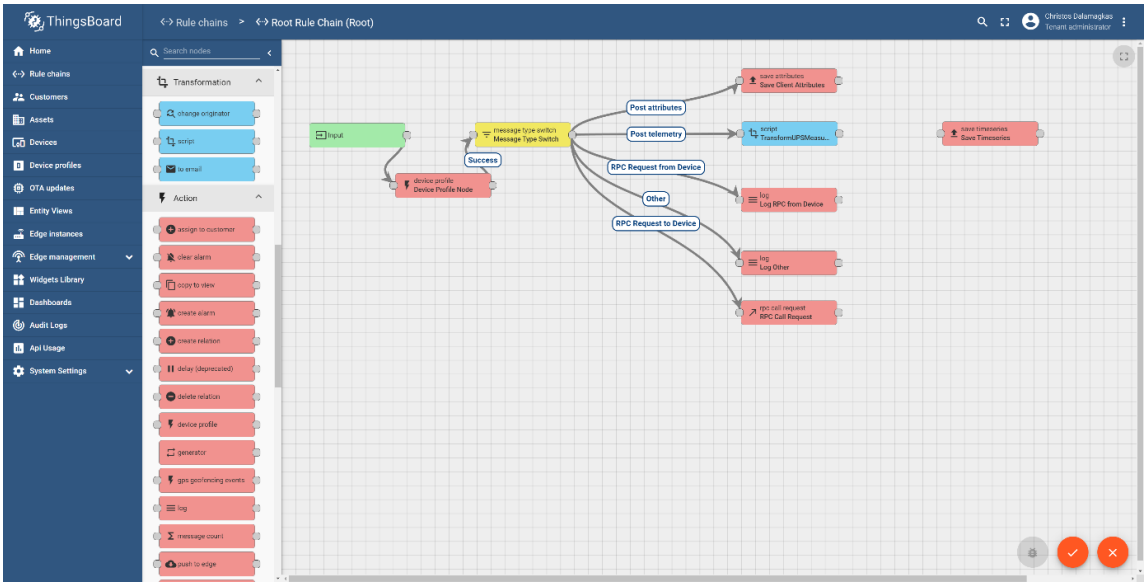


Figure 35: Feeding the new transformation node with telemetry

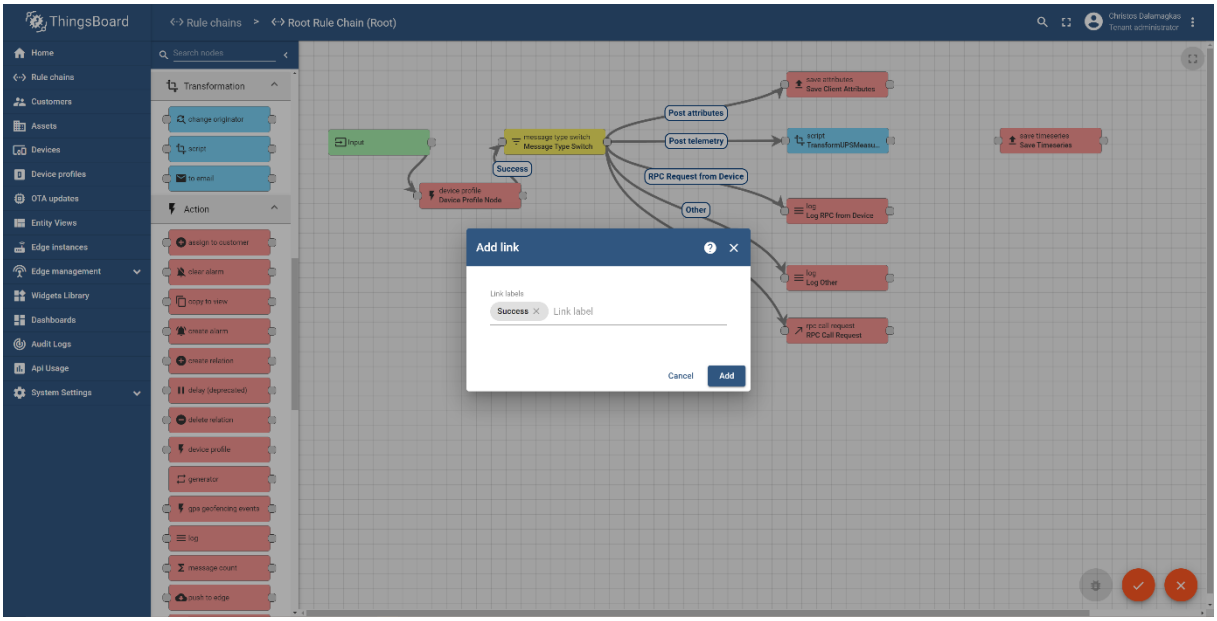


Figure 36: Add new link

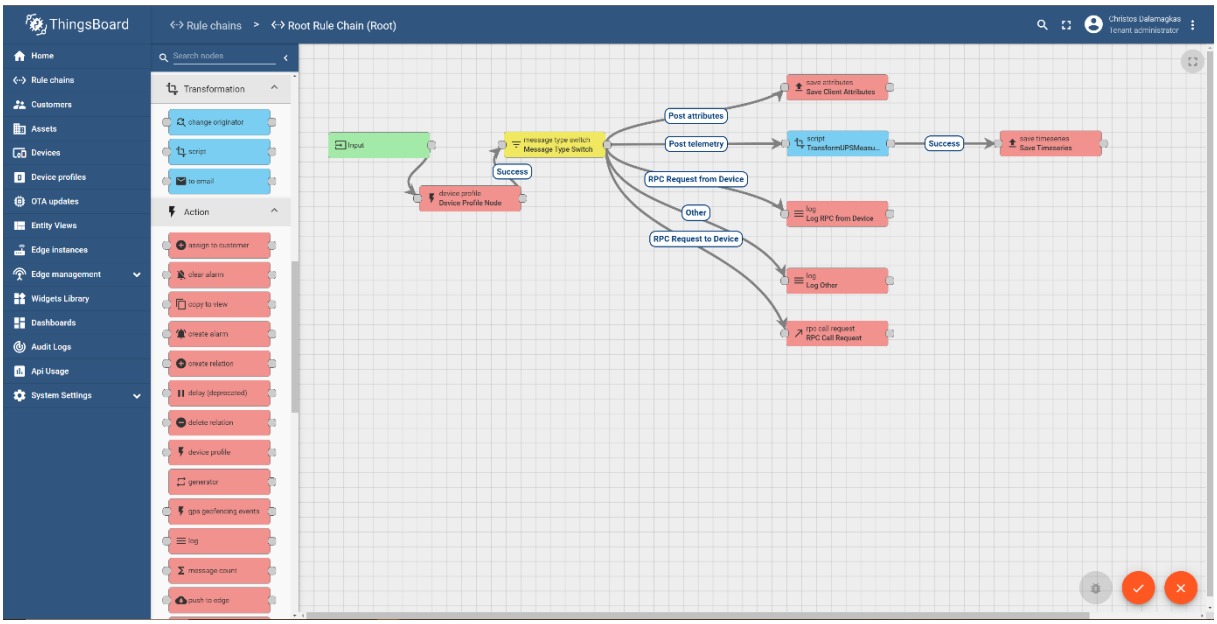


Figure 37: The new link between the transformation node and the save timeseries action

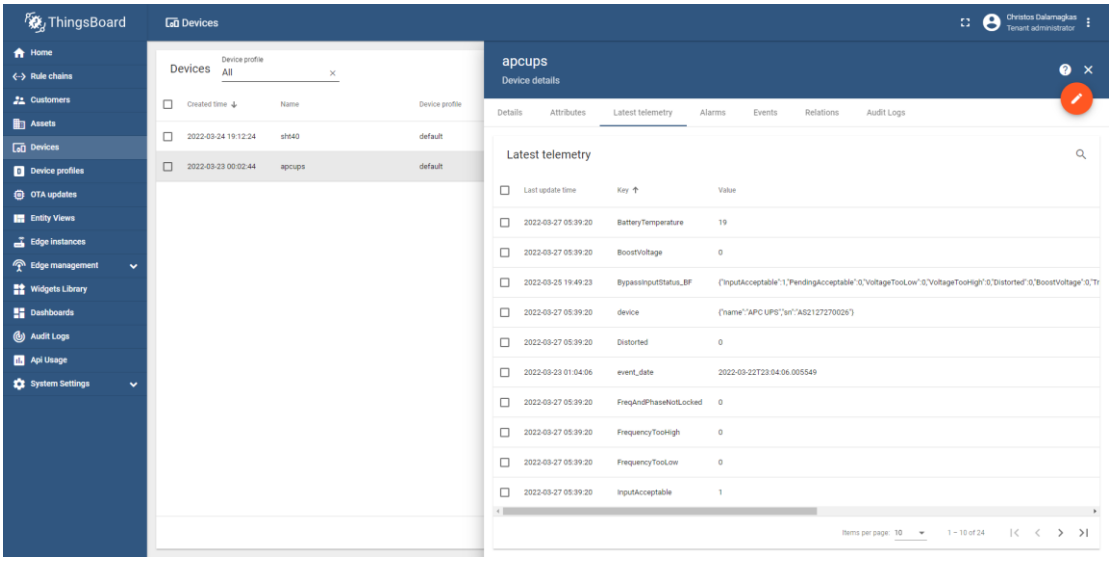


Figure 38: The UPS telemetry after transformation

Step 4: Add Rules for Alarms

As a final step, the telemetry can be filtered to generate alerts and warnings, if necessary. For example, we would like to be alerted if the input signal is distorted, or if the frequency or the voltage deviates significantly. Since the UPS already uses Boolean variables to indicate such issues (the keys included in BypassInputStatus_BF), the rule engine needs only to evaluate those variables and generate the corresponding alerts.

For each alert, three nodes are necessary, namely a) a script filter node, b) a create alert node, and c) a clear alert node. First, the script filter node applies a logical operation on the telemetry contents to determine whether the key corresponding to the alert is true or not. If true, the corresponding alert node is activated, and the alert is created. The alert remains active until is cleared. If the same filter evaluates to false at some other point, then the clear alert node is activated, and the alert is cleared.

To create a filter node for the VoltageTooLow, drag the “script” filter node and drop it freely on the dashboard. This will suddenly open the Add rule node dialog. As node name type “VoltageTooLow” and provide the following source code for the function:

```
if (msg.VoltageTooLow === 1)
    return true;
else
    return false;
```

Click the “Add” button on the rule node script dialogue to create the new filter node.

To create an alert node, drag and drop a “create alarm” action node from the left sidebar menu. The alarm node dialogue will open, in which at least the alarm name and the criticality should be specified as annotated in Figure 41. Click “Add” to create the node.

To create a clear alert node, drag and drop a “clear alarm” action node from the left sidebar menu. The clear alarm node dialogue will open, in which the alarm name should be specified. Please note that the same name specified in the “create alarm” node must be used here. The dialogue is illustrated in Figure 42.

Finally, the three nodes should be connected as illustrated in Figure 39. The input of the script filter node must be connected with the output of the save timeseries node, which outputs the telemetry that has been saved.

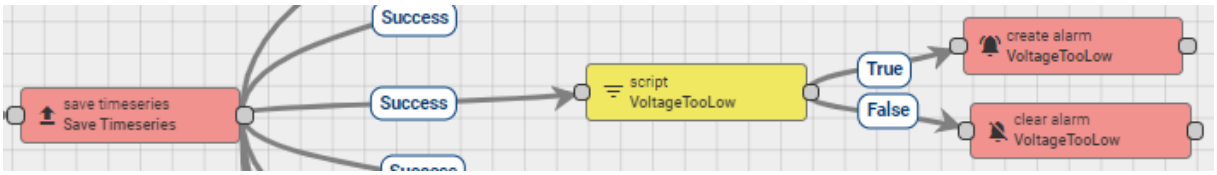


Figure 39: Interconnecting the alert and filter nodes

Try to create additional script and alarms nodes that are activated if the Distorted, VoltageTooHigh, FrequencyTooLow, FrequencyTooHigh, FreqAndPhaseNotLocked, PhaseDeltaOutOfRange, and NeutralNotConnected are 1. The final rule chain should look like the one illustrated in Figure 43.

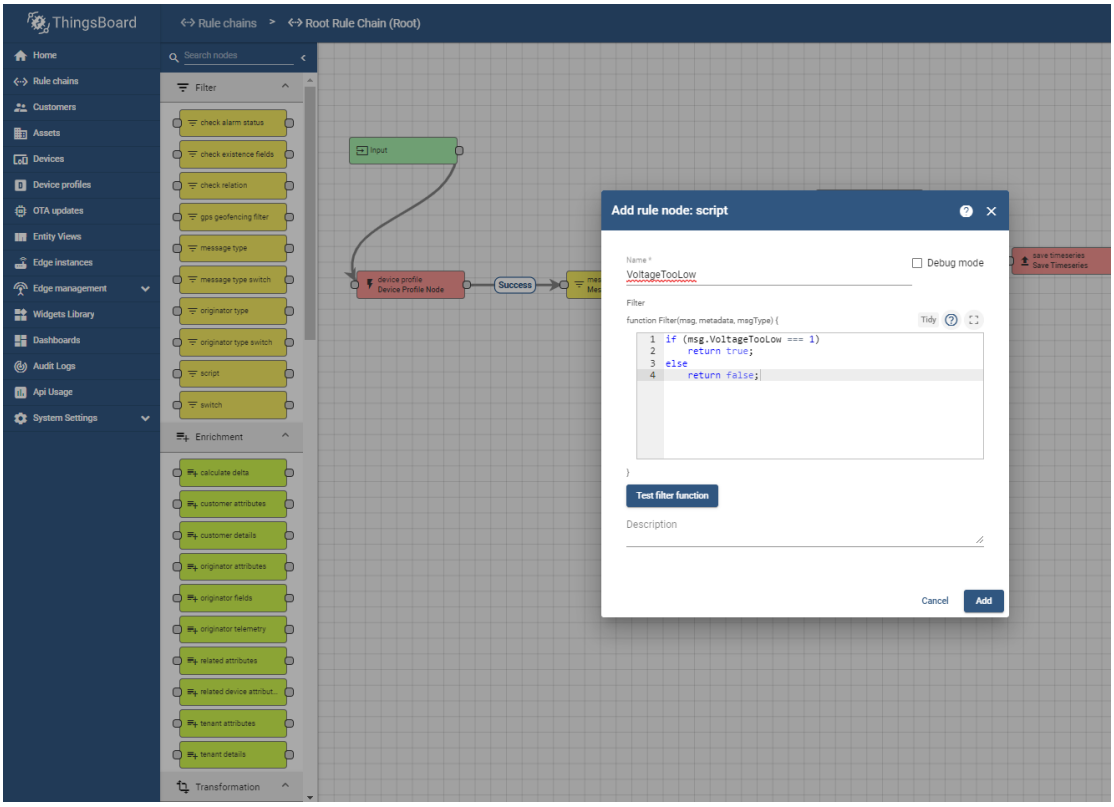


Figure 40: Creating a new script filter node

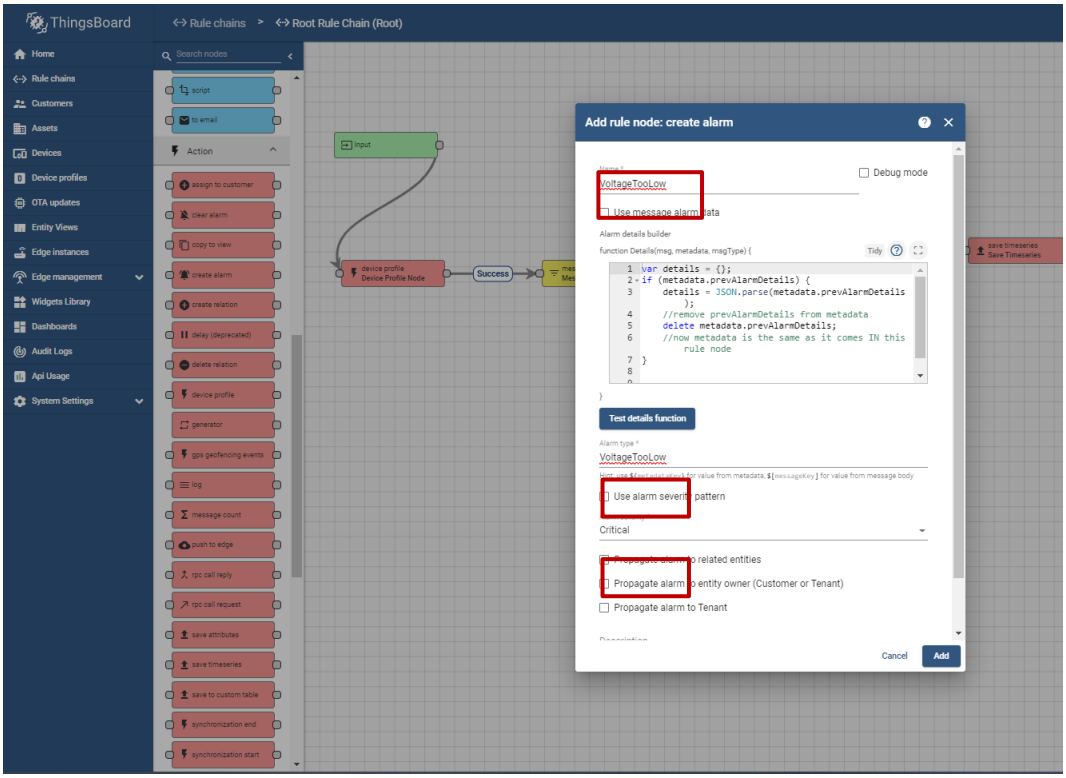


Figure 41: Create a new alarm node

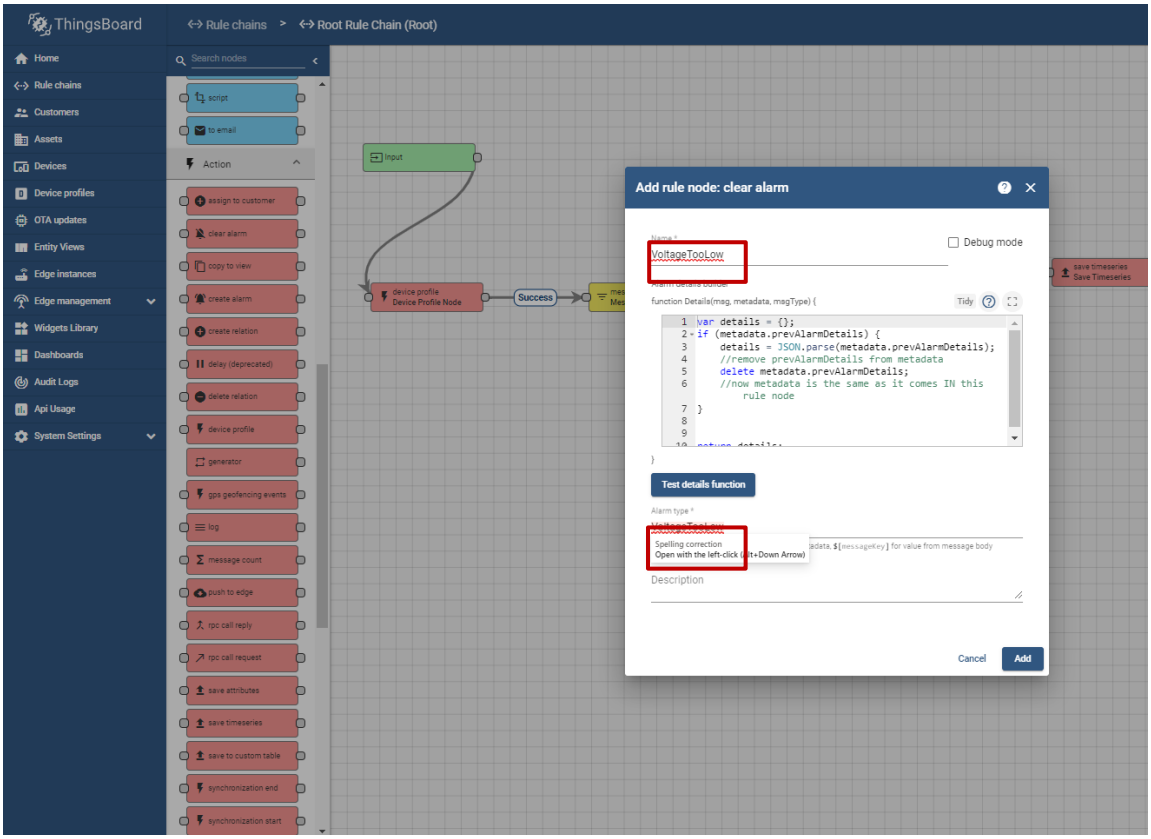


Figure 42: Creating a clear alarm node

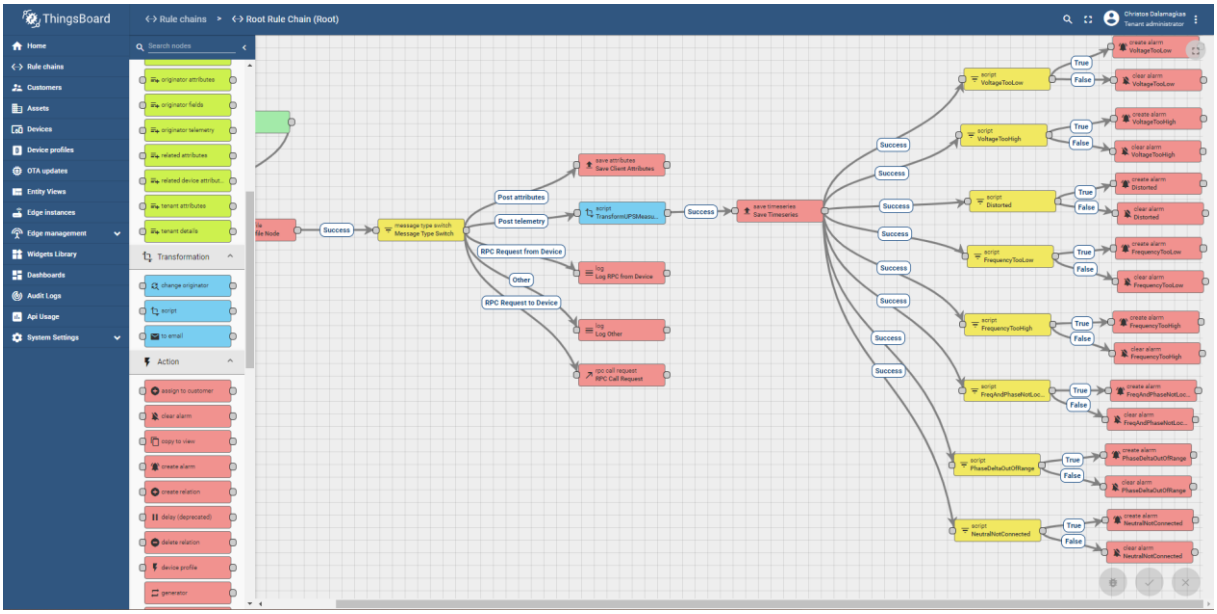


Figure 43: The final root rule chain

7. Exercise 4: Creating a ThingsBoard Dashboard

ThingsBoard allows tenants to create multiple dashboards. Each dashboard could serve a specific purpose, for example to monitor the energy consumption of a smart home or office, for smart farming or water consumption analysis, amongst other. A dashboard can contain plenty of widget, which visualise data and alerts from many IoT devices.

Purpose of this exercise is to build a dashboard that monitors the most important metrics of the computer room, including the power consumption, the battery temperature, and the humidity of the environment amongst other.

The dashboard of this exercise is illustrated in Figure 44.

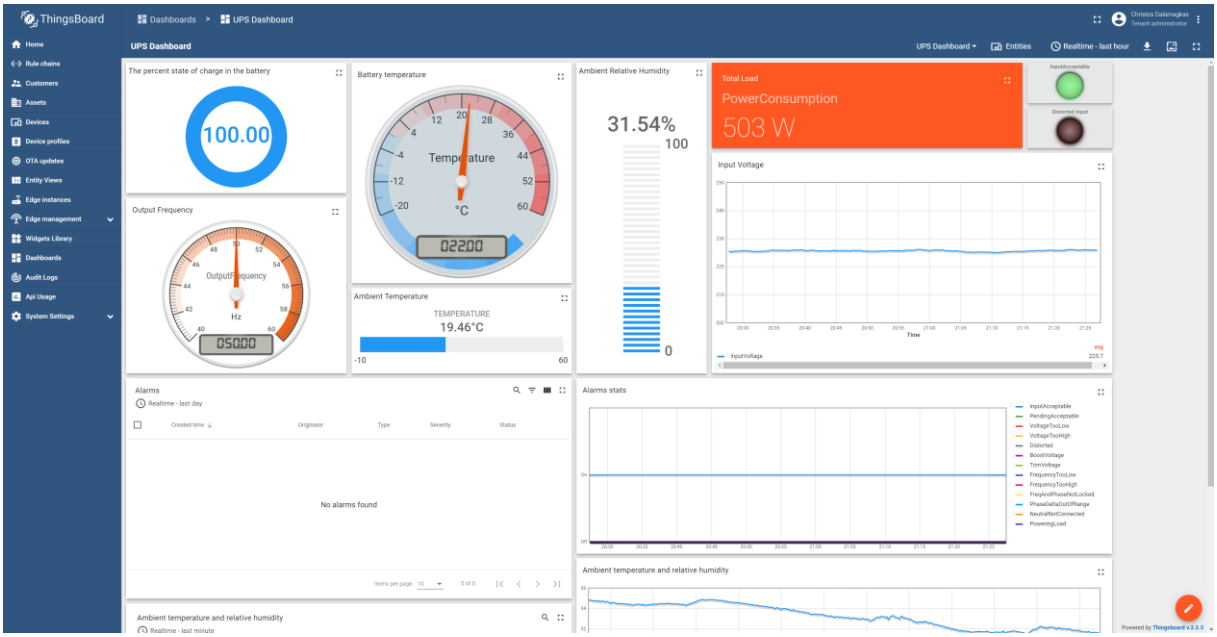


Figure 44: The UPS Dashboard

Step 1: Create Dashboard and assign Customer

To create a new dashboard, navigate to the Dashboards menu by clicking on the “Dashboards” option of the main left sidebar menu. The Dashboards menu is depicted in Figure 45. To create a new dashboard, click on the “+” icon annotated in this figure.

On the popup dialogue, depicted in Figure 46, provide at least a name of the new dashboard (e.g., JAUNTY Dashboard) and click “Add”.

Now, the dashboard needs to be assigned to the customer entity, so that all users belonging to that customer entity can access the dashboard and the widgets added in it. To make this assignment, click on the customers icon of the dashboard, annotated in Figure 47. Choose the JAUNTY customer as illustrated in Figure 48 and click “Update”.

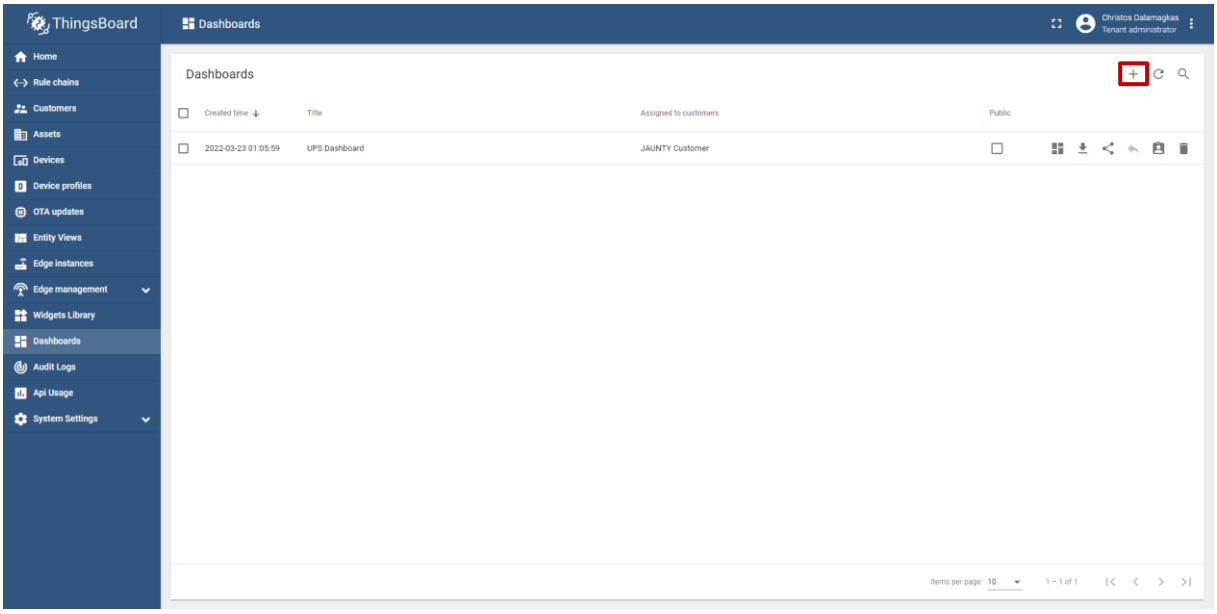


Figure 45: The Dashboards menu

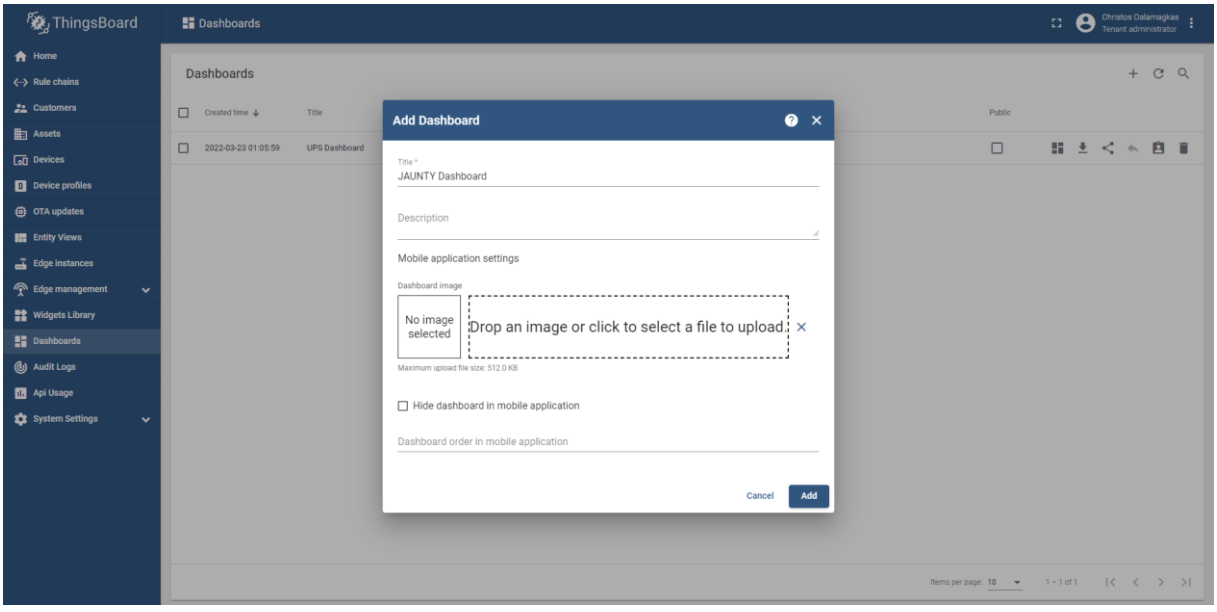


Figure 46: New Dashboard dialogue

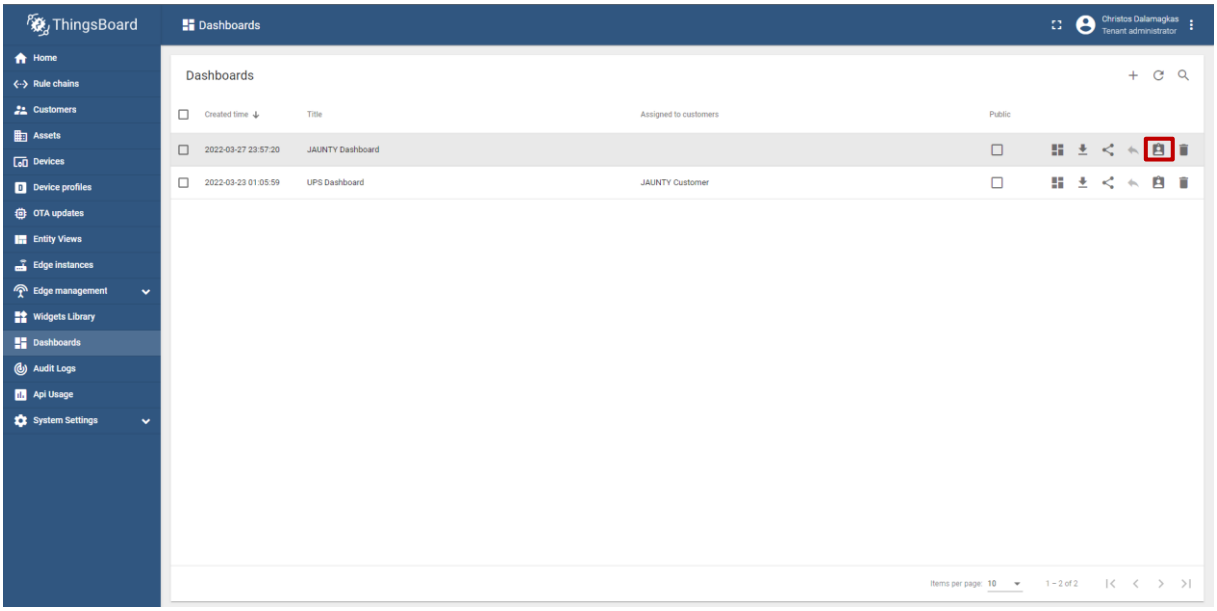


Figure 47: Open the Customer assignment of the new dashboard

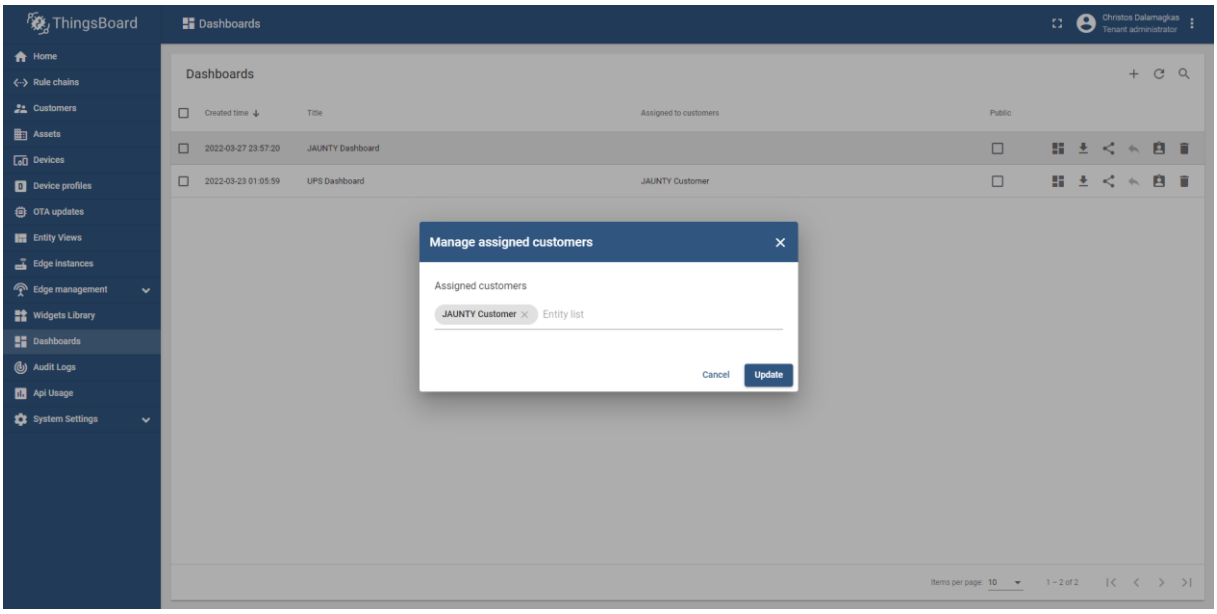


Figure 48: Assign Dashboard to Customer

Step 2: Add alias for the APC UPS device

Before adding widgets to the dashboard, we need first to create aliases corresponding to the IoT devices. An alias is a reference to a single entity or group of entities, which is used by the widgets as a search query to retrieve telemetry. In this step, an alias referring to the APC UPS will be created.

For this purpose, open the JAUNTY Dashboard and click on the “Edit” icon, annotated in Figure 49. Then, open the aliases dialogue by clicking on the Aliases icon, annotated in Figure 50. Click the “Add” button of the dialog to create the new alias.

After finishing with the alias, click on the “Apply Changes” icon, on the right bottom of the page, to save changes

Remember to frequently press the “Apply Changes” button located on the right bottom of the page and annotated bellow to ensure that your changes are always permanently saved.

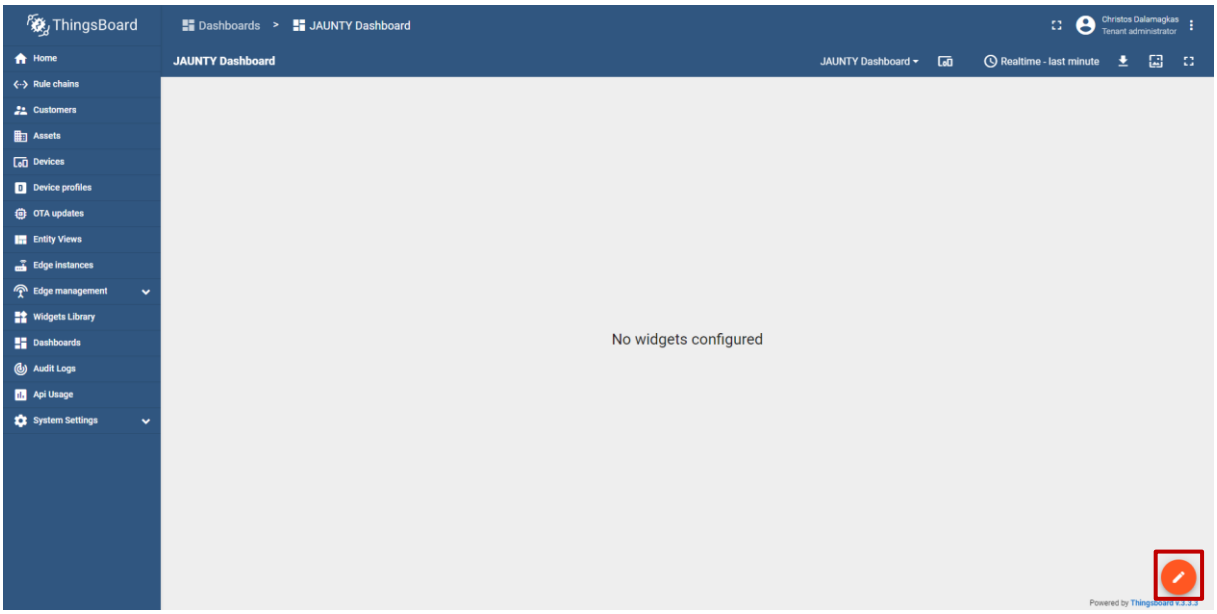


Figure 49: The empty JAUNTY Dashboard

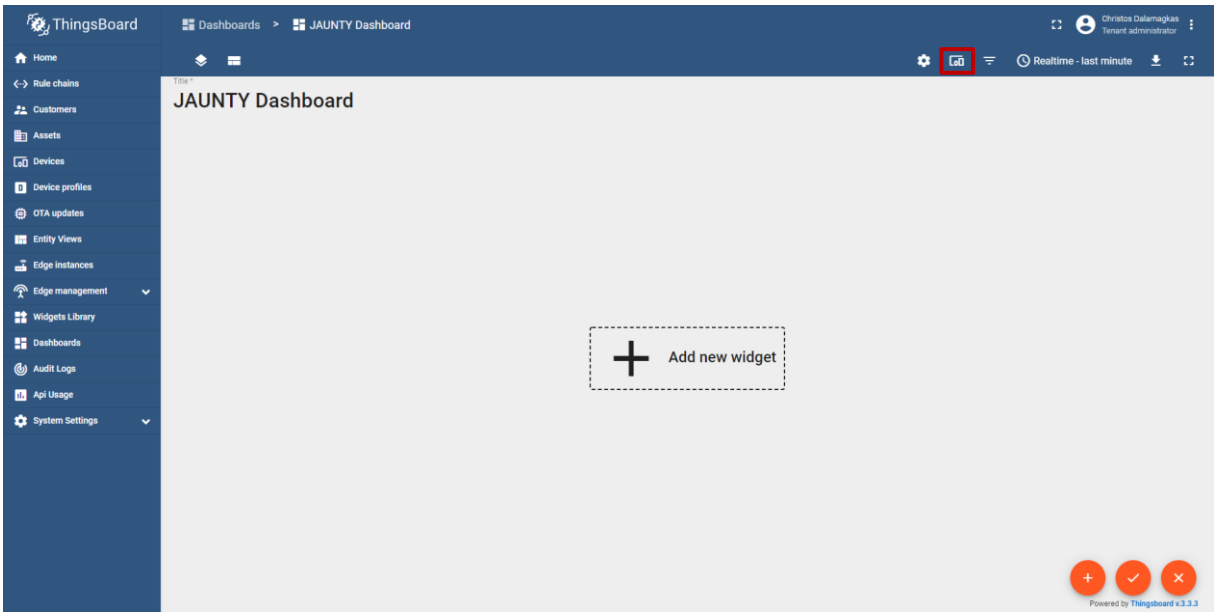


Figure 50: The Dashboard editing mode

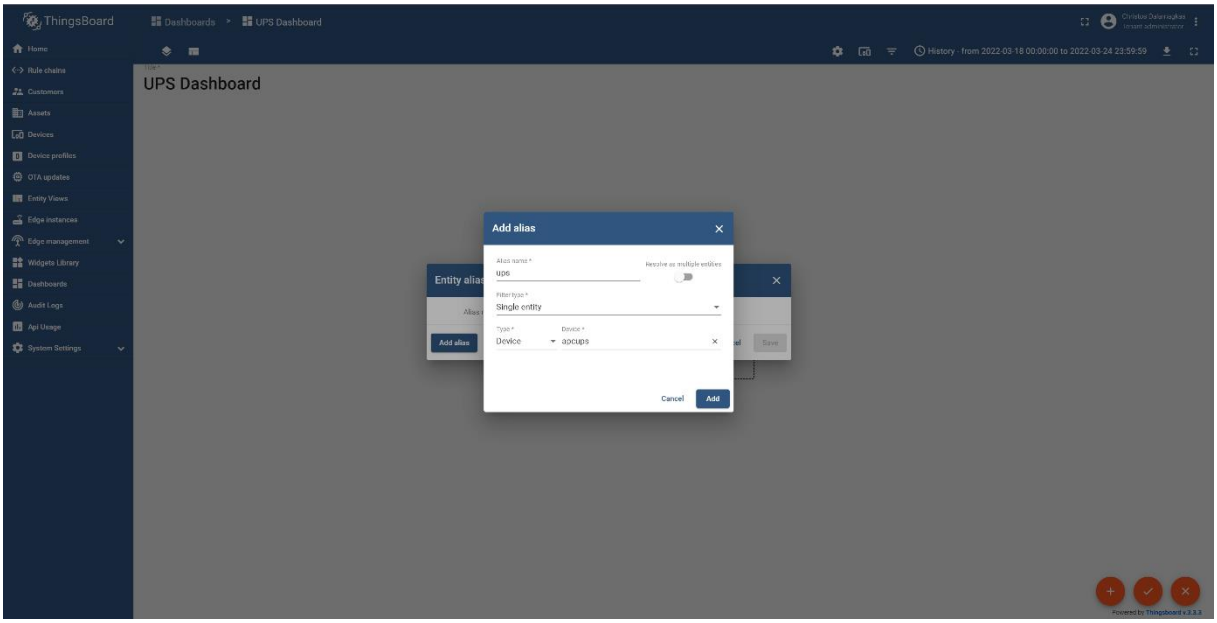


Figure 51: Adding the alias for the APC UPS

Step 3: Add alias for the SHT40 sensor

Follow the procedure of step 2 to add an extra alias that corresponds to the SHT40 sensor. All aliases are depicted in Figure 52.

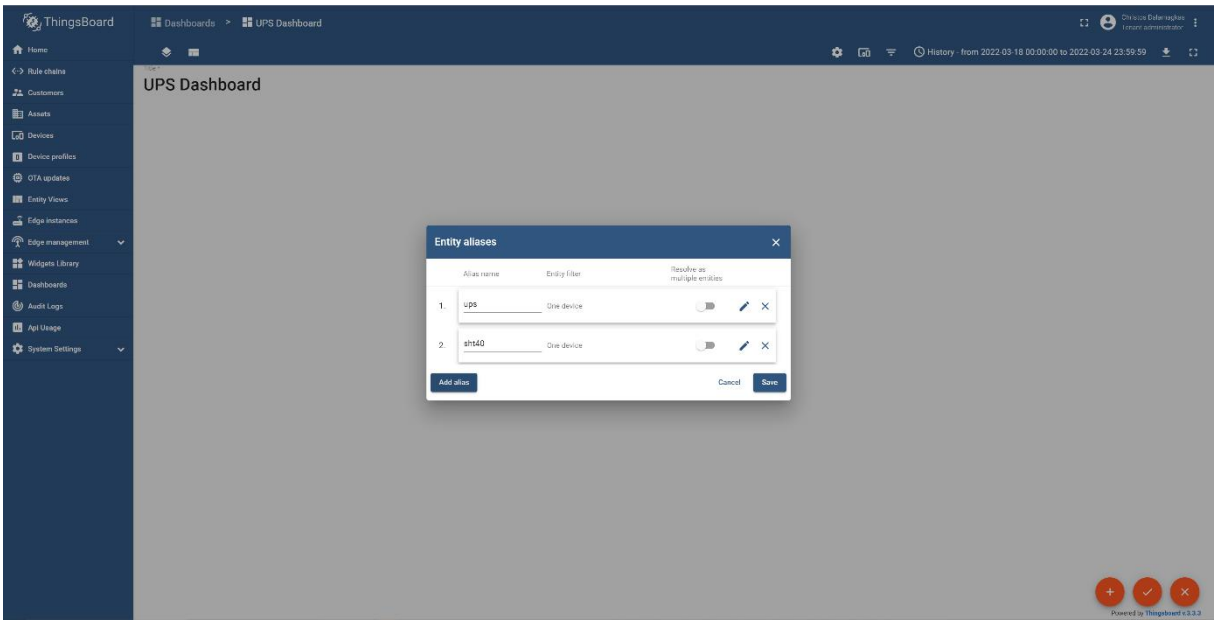


Figure 52: All entity aliases

Step 4: Create a gauge widget

In this step, you are going to add a new gauge widget that illustrates the charging state percentage of the UPS battery. To this aim, open the empty dashboard and click on the create new widget icon, annotated in Figure 53. This will open the widget bundle menu depicted in Figure 54.

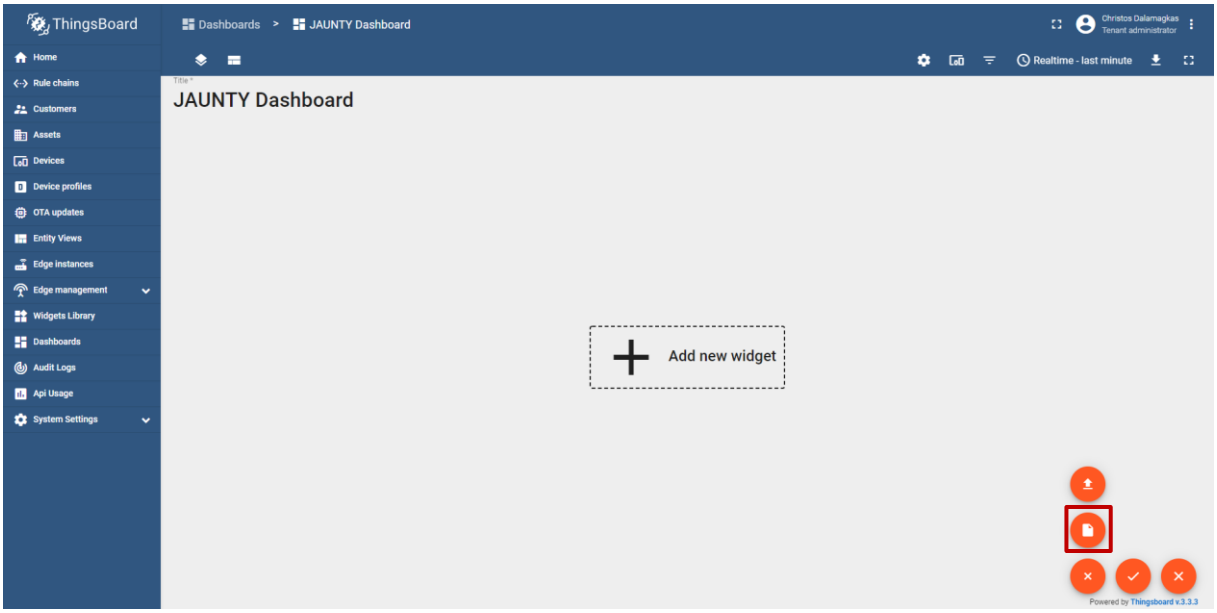


Figure 53: Choose the create widget option

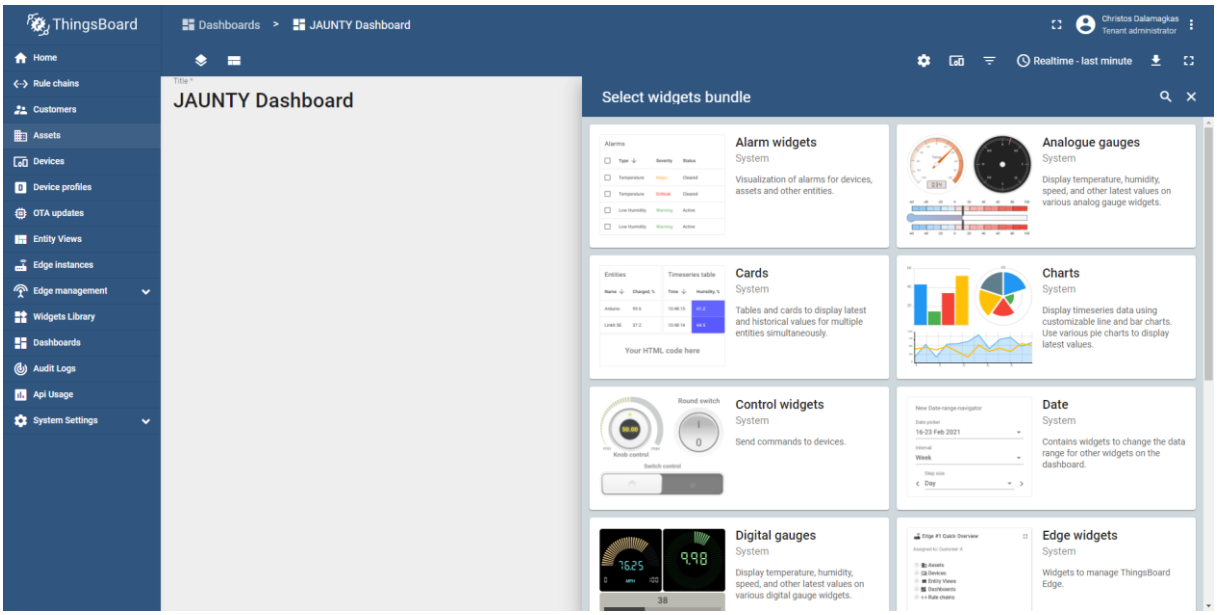


Figure 54: The widget bundle menu

On the widgets bundle menu, navigate to “Digital gauges”, and click on the “Mini gauge” widget, which will open the widget configuration dialogue. On the active dialogue, under the tab “Data” and the Datasources, click the “Add” button to specify the source of information for this widget. Then, specify the entity alias and the StateOfChargePct key, as annotated in Figure 55. Click “Add” to create the

widget. Click on the “Apply Changes” orange button to save the new widget. The mini gauge widget is depicted in Figure 55.

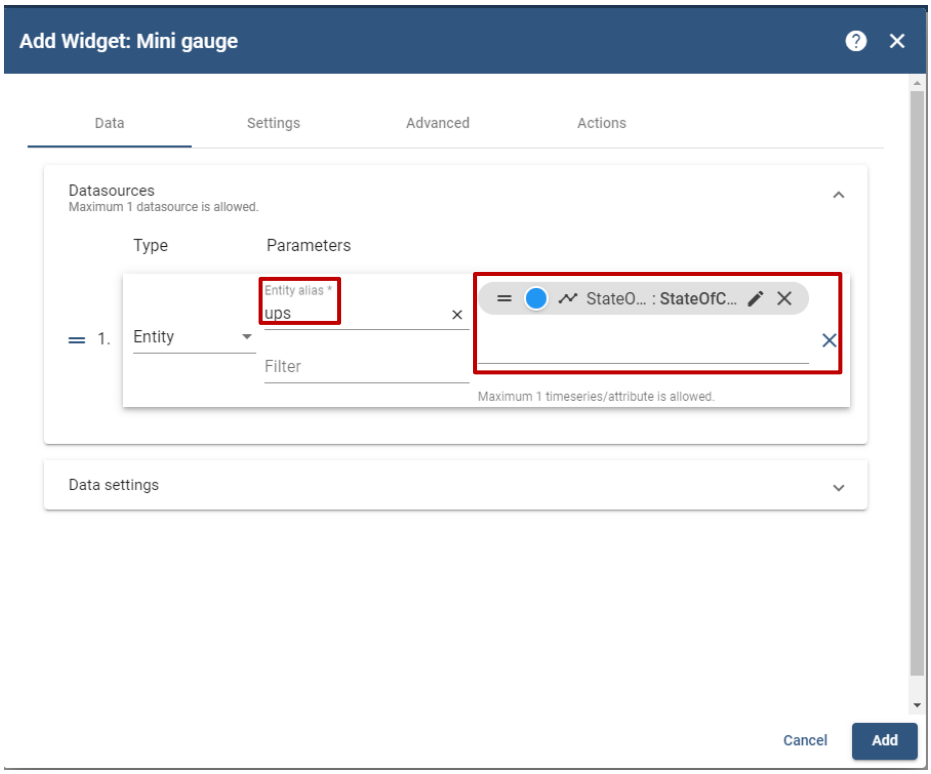


Figure 55: The data tab of the mini gauge configuration dialogue

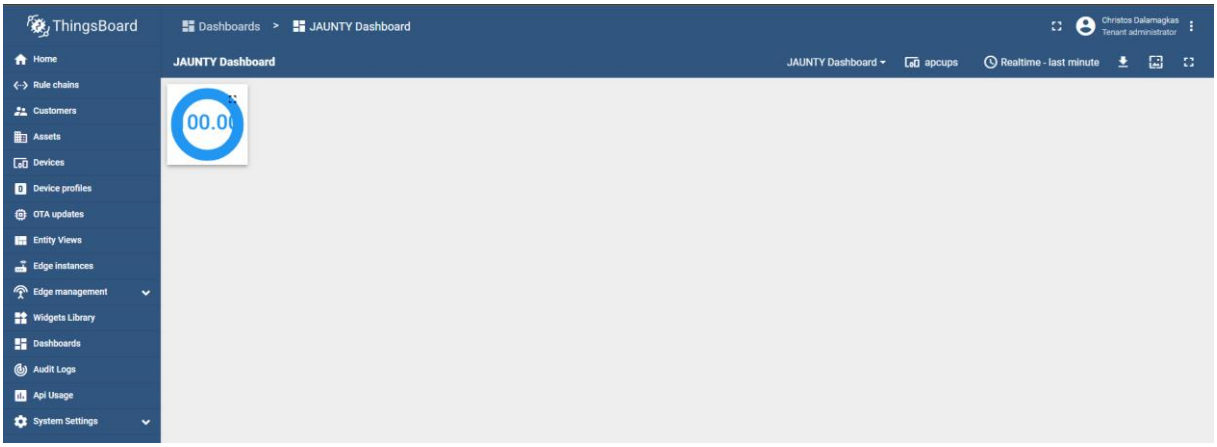


Figure 56: The new mini gauge widget

Unfortunately, the rendering of this widget is not appropriate, since the font is large enough so that the number is partially hidden by the ring. To correct this, open the editing mode of the dashboard, edit the widget properties, and open the “Advanced” tab, depicted in Figure 57. Then, scroll down to the font size and change it so that the number is properly displayed inside the donut. After changing the font, you can press the tick button, annotated in Figure 57, to instantly preview the changes.

Feel free to experiment and change the attributes of the widget, to customise it according to your preferences. For example, you could change the colours or the gauge type to arc or horizontal bar.

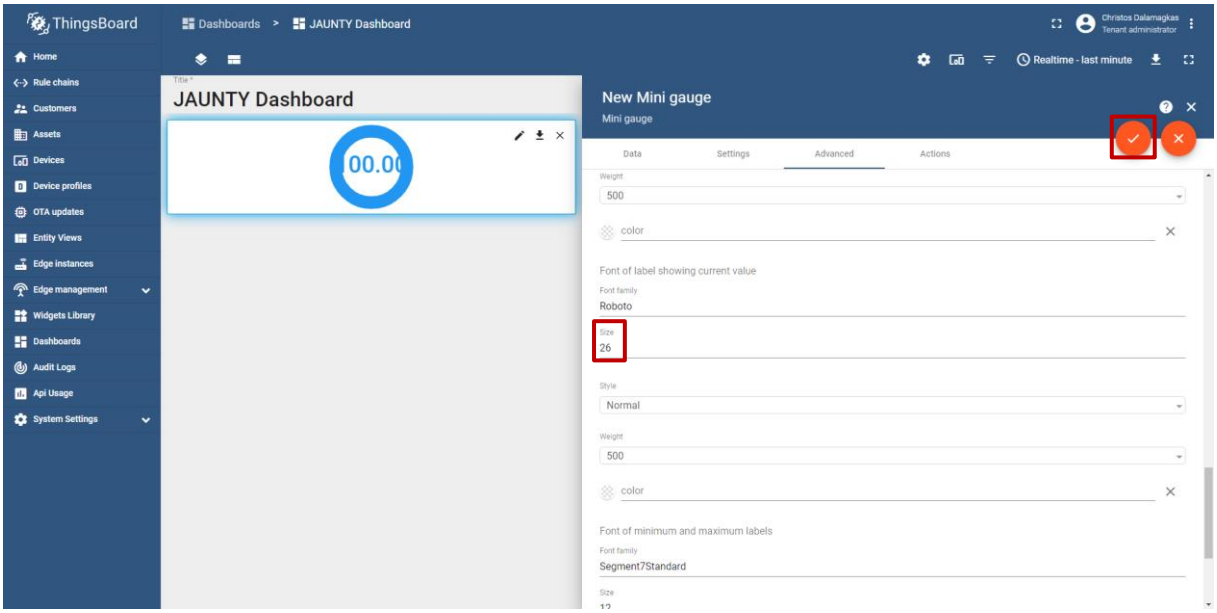


Figure 57: Advanced properties of the mini gauge widget

Step 5: Complete the Dashboard

Try to create more widgets to imitate the dashboard depicted in Figure 44. The following widgets will be needed:

- Analogue gauge for the battery temperature, located in “Analogue gauges” > “Temperature radial gauge”.
- Simple card for the power consumption, located in “Cards” > “Simple card”. On the configuration of this card, you need to expand the “Data settings” of the “Data” tab, and replace °C with W.
- Two LEDs for the Distorted and InputAcceptable keys, located in “Control widgets” > “Led indicator”. While configuring those widgets, you need to open the “Advanced” tab and apply the configurations according to Figure 58.
- A line chart for the input voltage, located in “Charts” > “Timeseries Line Chart”.
- An analogue gauge for the output frequency.
- The alarms table, located in “Alarm widgets”.
- A state chart for all Boolean keys, located in “Charts” > “State Chart”.

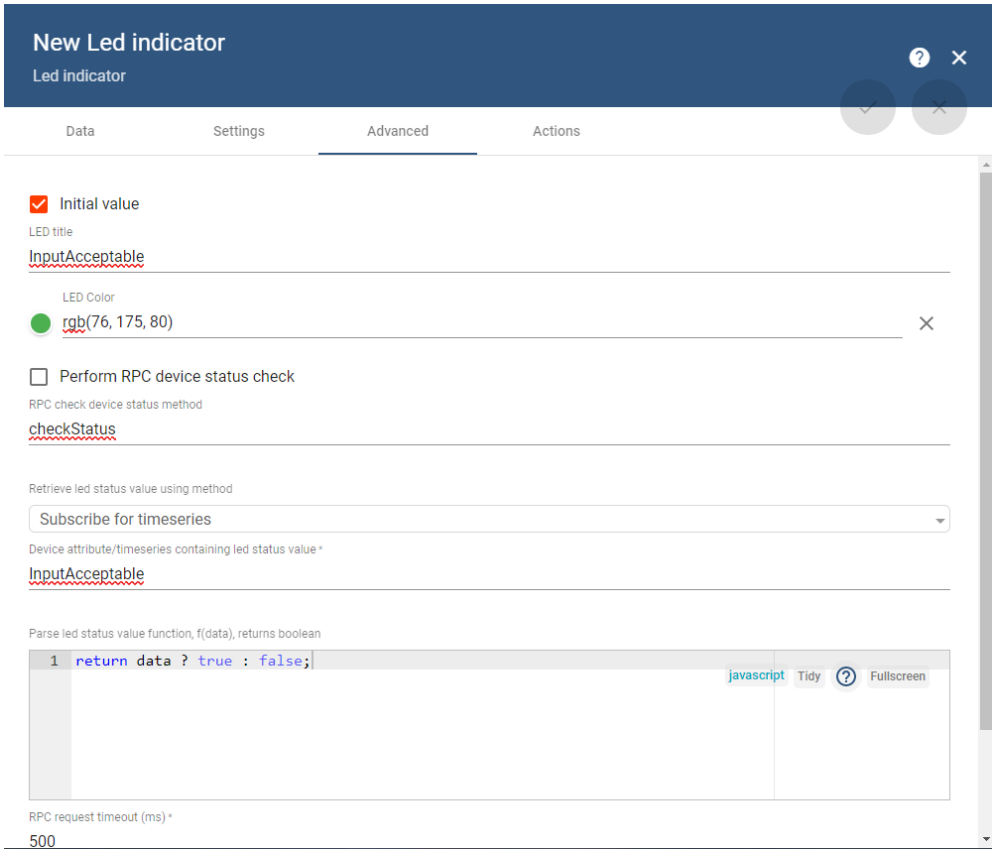


Figure 58: The configuration of the InputAcceptable Led indicator widget

8. Exam Questions

1. How a device can be connected with ThingsBoard?
2. What are the main benefits of CoAP?
3. What is the necessary information you need to post telemetry to ThingsBoard via CoAP?
4. What is the difference between MQTT and CoAP?

References

[1] T. L. Scott and A. Eleyan, “CoAP based IoT data transfer from a Raspberry Pi to Cloud,” in *International Symposium on Networks, Computers and Communications (ISNCC)*, Istanbul, Turkey, 2019.

[2] Sensirion AG, “±1.8% / max. ±3.5% Digital humidity and temperature sensor,” [Online]. Available: https://sensirion.com/products/catalog/SHT40/. [Accessed 28 April 2022].

[3] Schneider Electric, “https://www.se.com/us/en/download/document/SPD_LFLG-A32G3L_EN/,” [Online]. Available: https://www.se.com/us/en/download/document/SPD_LFLG-A32G3L_EN/ . [Accessed 28 April 2022].

6. 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