Master of Science HES-SO in Engineering
Av. de Provence 6
CH-1007 Lausanne

Master of Science HES-SO in Engineering

Orientation : Technologies industrielles (TIN)

Low-complexity scalable IoT framework

Fait par

Lucas Bonvin

Sous la direction de
Prof. Dominique Gabioud
Dans l'institut de recherche énergie et environnement de la HES-SO Valais-Wallis

[Expert externe Alain Woeffray, Co-fondateur Des Choses Pareilles]

Sion, HES-SO//Master, 2020

Accepté par la HES-SO//Master (Suisse, Lausanne) sur proposition de

Prof. Dominique Gabioud, conseiller de travail de Master
[Alain Woeffray, Expert principal]

Lausanne, le 03 février 2020

Prof. Dominique Gabioud	Prof. Philippe Passeraub
Conseiller	Responsable de la filière MSE

Contents

List of Figures

Acknowledgment

At first, I want to thanks Dominique Gabioud, advisor of this master thesis, for giving me the opportunity to do my PA and my thesis with cloud.iO. Then a big thanks to Michael Clausen, the man behind cloud.iO v0.1 who always had time and answers to any of my problems.

Thank you to Alain Woeffray, expert of this master thesis, who did show real interest in the cloud.iO project.

I want to thanks the member of the W3C Web Of Thing Working Group starting with Jeff Jaffe, who introduced me to members of W3C dedicated to WoT. Then I want to thanks Dave Raggett and especially Kazuyuki Ashimura, who did invite me to two weekly calls with the W3C Working Group. During those calls, they kindly let me present cloud.iO and have open discussions over my implementation of their Candidate Recommendation. Finally, I want to thanks again Kazuyuki Ashimura, who held WoT main call and Sebastian Kaebisch, who held the WoT Thing Description call I had the chance to be part of.

Finally, I want to thanks my loved ones who were present the whole time of my thesis, and some of them will recognize themselves in this thanks during the many re-reading of this report.

Abstract

The Internet of Things is a constantly expanding world. The amount of connected devices keeps growing, inducing an augmentation of data transfer. This growth conduct to the increasing use of cloud-based solutions in the IoT. The Institute of Systems Engineering at HES-SO Valais Wallis developed its own cloud base solution called cloud.iO. This framework evolved through research projects to a mature version of the so-called v0.1. The cloud.iO framework needs to evolve according to the IoT landscape. This thesis describes how cloud.iO can be positioned in this landscape and what are the possible enhancement to create a new version of cloud.iO called v0.2. The main enhancements for cloud.iO covered and implemented by this thesis are User Access Control, Remote Job Execution, Logging, Transaction, RESTful API, and enhancement of interoperability by following Web of Thing candidate recommendation by W3C.

Key Words: cloud.iO, cloud, IoT, W3C, Web of Things, RESTful API, Database

1 Introduction

1.1 Overview

The Internet of Things (IoT) gains in popularity every year. The number of IoT connected devices continues to grow and could reach 41.6 billion by 2025 [1]. All those devices generate an enormous amount of data; Cisco predicts it could reach 20.6 Zettabytes a year (billion of Terabytes) by 2021[2]. But where are those data going? In an IoT environment, the leading answer is to an IoT cloud-based solution or simply IoT cloud.

The main goal of an IoT cloud is to connect different Things of an IoT environment to the web. All the data generated by the Things are sent to the cloud; they can then be stored, analyzed, or retrieved. Those IoT clouds also give plenty of tools to manage their connected Things.

One of these IoT solutions is called cloud.iO[1], an IoT cloud created by the Institute of Systems Engineering, HES-SO Valais-Wallis (Hevs). The cloud.iO project started in 2017 as an answer to projects' needs at the research institutes of Hevs. It has already proven its efficiency during the European FP7 SEMIAH and the Horizon 2020 GoFlex projects [3].

The main component of cloud.iO is the Endpoint; it can be considered as an IoT gateway for multiple Things or a Thing supporting cloud.iO. This Endpoint sends its state to the cloud who can be accessed by Users. This architecture creates a simple IoT environment as seen in Figure 1

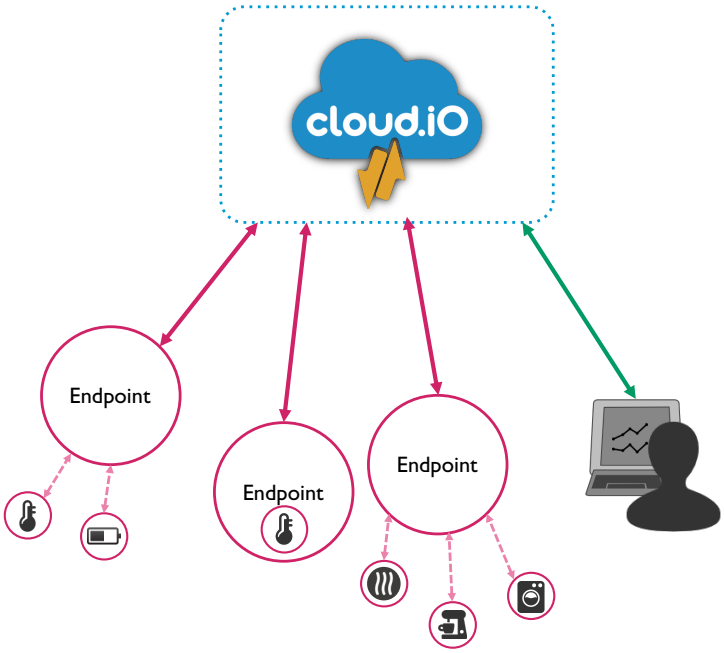


Figure 1: cloud.iO environment composed of different Endpoints and an User

1.2 Goal of the project

The initial version of cloud.iO (called v0.1 in this document) was evolving through the research projects adding features to their needs. This way of working was only letting cloud.iO grow in a specific direction, and so its evolution was restricted. To remedy this problem, a study of the IoT landscape is conducted in this document to see where are the weakness of cloud.iO v0.1 comparing to the other IoT clouds.

An in-depth study of cloud.iO v0.1 is also needed to understand well its architecture and how it can be enhanced.

With that in mind, a new specification for cloud.iO can be made and lead to the implementation of an evolved version (called v0.2 in this document).

[1] https://github.com/cloudio-project

2 Analysis

2.1 Specifications

The following specifications have been established with Dominique Gabioud, advisor of this master thesis.

2.1.1 Objectives

- Analysis of the IoT landscape and positioning of the new cloud.iO v0.2 in this landscape.
- Collection of requirements for cloud.iO v0.2 (from potential users) and elaboration of a corresponding specification.
- Development, documentation, test, and validation of cloud.iO v0.2.

The proposed project aims to develop a new version "v0.2" for cloud.iO. While keeping the strength of the first version, the new design should simplify the commissioning of new Things, ease the development of Things firmware, and facilitate the integration into legacy IoT environment.

2.1.2 Establishment of tasks

First, a State of the Art of the IoT landscape is needed; from this state of the art, we can retrieve the critical points for a more in-depth analysis.

Secondly, the first version of cloud.iO will be analyzed. This analysis will be based on the feedback of cloud.iO users and an analysis of other IoT cloud-based platforms. The outcome of state of the art and the study of the first version of cloud.iO will show which are the cloud.iO weak spots and what needs enhancement. A synthesis will conclude the analysis, and a list of tasks toward cloud.iO enhancement will be presented.

The conception will follow the path established in the analysis, and the most significant enhancement will be discussed and implemented. This conception will lead to a so-called *v0.2* of cloud.iO, which will be finally tested and evaluated.

2.2 State of the Art

The cloud.iO framework is a cloud-based solution for the IoT. The primary purpose of a cloud-based framework is to save data generated by a Thing to the cloud. In general, the interface between Things and clouds is made with Machine to Machine (M2M) protocol, for example, AMQP or MQTT. cloud.iO coexists around many other IoT cloud-based solutions, at least 152 cloud platform solutions, according to Postcapes[2] and we can expect this number to rise over the years.

A study called A survey of IoT cloud platforms [4] defines where IoT clouds are positioned in the IoT landscape. Figure 2 represent the different domain of IoT cloud platforms. Every cloud solution will be different, and not all of them will touch all those domains.

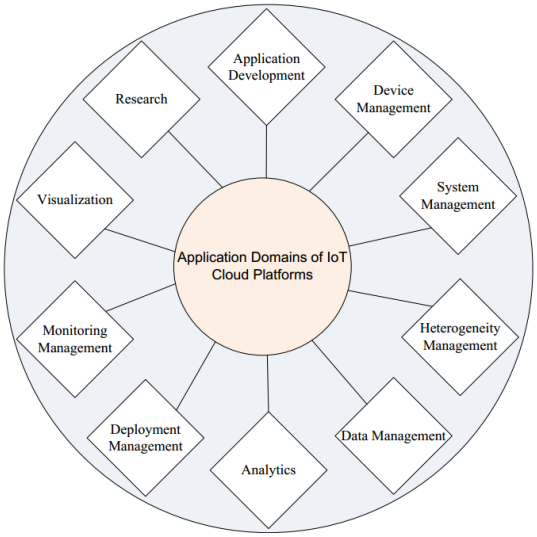


Figure 2: Application domains of IoT cloud platforms[4].

The previous figure gives a lot of domain about IoT cloud platform, for now, we will focus on three main keys that represent well IoT platform and are always found in those frameworks:

- **Thing Monitoring**: access to the Digital Twin stored on the cloud of the Thing and possibility to configure it
- **Thing History**: storing the evolution of the Thing state as time-series in the cloud
- **History Visualisation**: providing tool for the visualization of the evolution of the Thing state as time series

To complete the list of cloud features we can look to the leaders of IoT cloud: Azure IoT[3], Aws IoT[4], Google Cloud IoT Core[5] and Alibaba cloud[6]. Those four clouds provide really complete and mature solutions and provide, of course, the 3 main features described above.

We can retrieve new interesting features provided by those four clouds:

- **Remote Job Execution**: Executing a task/script/job remotely on the IoT gateway connected to the cloud
- **Rules Engine**: Implementation of simple rules that will guide the lifecycle of the Things and the IoT gateway
- **Cloud Computing**: Possibility of cloud computing on the retrieved data, e.g. data analysis, fault detection, etc

[2] https://www.postscapes.com/internet-of-things-platforms/
[3] https://docs.microsoft.com/en-us/azure/iot-hub/about-iot-hub
[4] https://docs.aws.amazon.com/iot/latest/developerguide/what-is-aws-iot.html
[5] https://cloud.google.com/iot/docs/
[6] https://www.alibabacloud.com/product/iot

The storage for those clouds is included in their solution and is on their own server. It is not possible to have an on-premise deployment as opposed to an open-source solution. We can take as example Eclipse Kapua[7], Thingsboard[8] or Mainflux labs[9] all three are open-source solution with the possibility to have an on-premise deployment of their cloud.

Another interesting feature that comes with the Mainflux labs solution is the fact that they also provide some hardware for their cloud. The specific hardware is not mandatory but is recommended for this solution. We can also cite Particle Cloud[10] and B+B SmartWorx[11] two other solutions with specific hardware for their IoT cloud.

To send data to the cloud, we already cited that communication is made using M2M protocols. Every cloud has its own message structure and way of interacting. It would be time-consuming for a user to create its own implementation of those messages with the correct M2M protocol, that is the reason why most of the IoT solutions provide SDK to add an abstraction for the user. If we take Azure IoT as an example, it supports AMQP, AMQP over WebSockets, MQTT and MQTT over WebSockets as M2M protocol, but the user does not need to use them directly since it has access to SDKs in C, C#, Java, Python or Node.js.

After looking at existing IoT cloud solution, let's address the research trends in this field. The architectures of the cloud were generally focused on cloud computing. We would send all the data to the cloud and then analyze this data with the computing power of the cloud. With the increasing amount of data and also the increasing compute power of embedded systems, this tendency is on the way of changing. Studies called "Deep Learning: Edge-Cloud Data Analytics for IoT"[5] and "Performance evaluation of full-cloud and edge-cloud architectures [...] based on deep learning"[6] show that Edge computing mixed with deep learning will bring plenty of advantages in IoT cloud. Edge computing has as main goals to do analytics on the data to do fault detection and to reduce the dimension of the data, consequently reducing its volume to be sent to the cloud. We now have enough computing power in our embedded system to run neural network dedicated to those tasks; it is a significant advantage to analyze the data closer to their origin.

As a final note of this state of the art, we already cited that cloud.iO surrounded by many other IoT cloud-based solutions. The market is competitive, and each solution is unique. This fact put forward a problem in the IoT ecosystem: The lack of interoperability. Every IoT solution comes with its own syntax and/or semantic, which creates a heterogeneous IoT environment. This problem will be detailed in Section 2.6

This analysis brings us the last interesting key features IoT cloud solution:

- **On-premise Deployement**: Can the cloud be deployed on-premise
- **Specific Hardware**: Does the IoT cloud work only on specific hardware
- **SDK**: Does the IoT cloud give SDK to access the cloud, and if yes in which language
- **Edge Computing**: Does the IoT cloud provide a tool for edge computing, the possibility of deep learning
- **Interoperability**: Does the IoT platform offer interoperability or follow a standardization

A few critical features of IoT cloud were exposed in this State of the Art, with examples among the existing IoT cloud solution. A more in-depth analysis of those different clouds is made in Section 2.5.2, focusing on the already cited features and comparing them with cloud.iO.

[7] https://www.eclipse.org/kapua
[8] https://thingsboard.io
[9] https://www.mainflux.com
[10] https://www.particle.io/
[11] http://advantech-bb.com/product-technology/iot-and-network-edge-platforms/smartworx-hub/

2.3 cloud.iO v0.1

2.3.1 Concept

cloud.iO is a scalable open-source IoT cloud-based solution. Its development started in 2017 at HES-SO Valais Wallis in the Institute of Systems Engineering. cloud.iO offers the possibility to monitor and control I/O devices or Things in an IoT ecosystem.

All the interactions to cloud.iO v0.1 are made by using M2M messaging. Two protocols are used: AMQP and MQTT. Every connection to the cloud with those messaging protocols are using state of the art encryption and certificate-based authentication.

cloud.iO v0.1 provides real-time monitoring of the Things and access to their historical data. In cloud.iO, an IoT gateway or a Things is called an Endpoint; it is one of the primary roles of cloud.iO described in the following subsection. The Endpoint will send data to the cloud with an M2M messaging protocol to a message Broker, RabbitMQ[12] in our case, and the cloud.iO core will then process those messages.

There are two main branches of development for cloud.iO: The cloud.iO core and the Endpoint libraries (or SDK) in Java and Python. The Python library is still in early development and following the model of the Java library. For enhancing cloud.iO, both core and libraries need to be updated. For this thesis, only the Java library will be updated and will benefit from the cloud.iO v0.2 changes.

All the development of cloud.iO, and so this thesis, is available on GitHub. Table 1 lists the main repositories for the cloud.iO project. A complete list of all cloud.iO repository is available in Appendix A1.0.

Description	URL
Deprecated version of cloud.iO v0.1 core and Java library	`https://github.com/cm0x4D/cloudio`
cloud.iO Core, development and release version	`https://github.com/cloudio-project/cloudio-services`
cloud.iO Java Endpoint library, development and release version	`https://github.com/cloudio-project/cloudio-endpoint-java`
cloud.iO Python Endpoint library, development and release version (not covered by this thesis)	`https://github.com/cloudio-project/cloudio-endpoint-python`

Table 1: Repositories of cloud.iO

[12] `https://www.rabbitmq.com`

2.3.2 Roles

The first version of cloud.iO was defining three different roles: the Endpoint, the Application, and the User. The Endpoint is the key element of cloud.iO in an IoT ecosystem. It represents an IoT gateway connected to one or multiple Things or a Thing capable of communication with cloud.iO. The Endpoint will send its actual state to the cloud with MQTT publishing following specific message formats and can receive commands from the cloud also with MQTT subscribing. Every update of the Endpoint states is saved in a time series database. Each Endpoint has its own data model; it is published to the cloud as well as every modification of it. All those MQTT transactions are secured with TLS, an SSL certificate is needed, and the Endpoint authentication is done with an x509 certificate.

The second role defined in cloud.iO is the Application. It can be described as software interacting with Endpoint. An application can subscribe to message sent by Endpoints, set value of an Endpoint set point, and access to the history of Endpoints state. The Application uses AMQP protocol with TLS/SSL, and an SSL certificate to communicate with cloud.iO is needed.

The last role is the User. The User is the owner of Endpoints. It can give access to its Endpoints to other Users or Applications. It is also to the User to create its own Application. The authentication of the User is made with a login and password. User permission toward Endpoints is not limited to "Own". There is in total 6 different permission levels that can be applied:

- Deny
- Read
- Write
- Configure
- Grant
- Own

The User also has authorities that can be seen as user-roles. In cloud.iO v0.1, the authorities are only used for authentication to the message broker management tools [13]. The available authorities are the following:

- administrator
- monitoring
- policymaker
- management

[13] `https://www.rabbitmq.com/management.html`

2.4 Global Architecture

The architecture of the cloud.iO's core is message-driven (using a message broker) and based on micro-service. This architecture gives three advantages:

- **Low-complexity**: The whole cloud.iO core can seem complex, but is composed of multiple simple services.
- **Scalability**: cloud.iO services can be distributed on multiple computing Nodes and deployed dynamically depending on the workload.
- **Extensibility**: cloud.iO can be easily expanded by adding new services, and existing services can be modified or updated individually.

The micro-service implementation is based on the Spring framework[14]. Spring is an open-source framework to define a Java or Kotlin application. In the case of cloud.iO core, its implementation is in Kotlin. The message broker of cloud.iO is RabbitMQ, an open-source solution based on AMQP. RabbitMQ offers the possibility to install plugins, and this is how we can use MQTT in it.

Figure 3 represent an instance of cloud.iO with all its actors. On the right, we have Endpoints sending and receiving data from cloud.iO using MQTT. In the middle down, we can see monitoring of the Endpoint, which can be achieved by MQTT subscribe or by using Grafana[15] , an open-source analytics and monitoring solution for databases. On the lower right, we can see the User interacting with an Application. Those Applications interact with cloud.iO either with AMQP and MQTT. On the cloud part, we can see the message broker handling all the messages. Then we have the core of cloud.iO with the micro-services in spring, which is doing the bridge between the backends and the message broker.

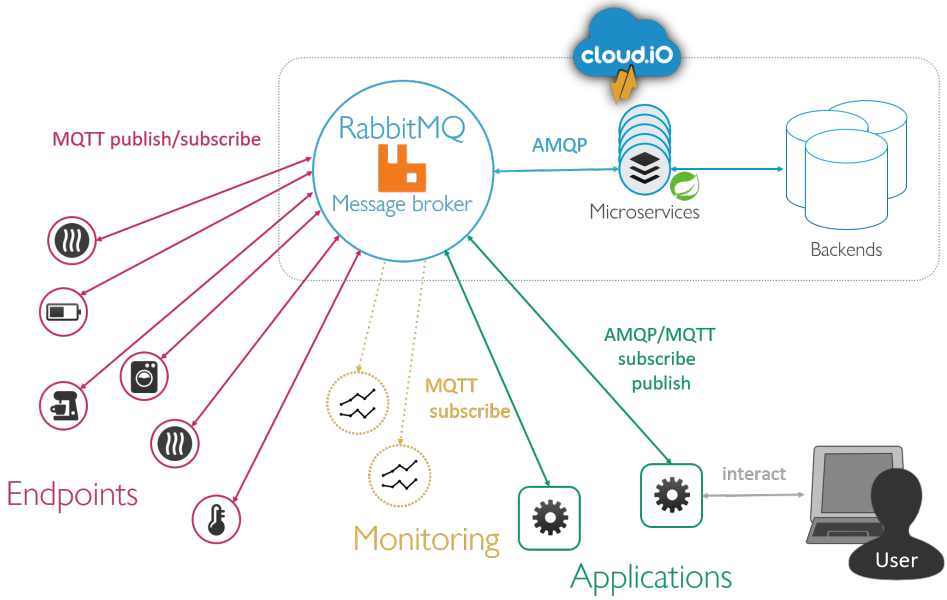


Figure 3: Global architecture of cloud.iO v0.1

A few points of the Figure 3 will be detailed in the following Subsection. Subsection 2.4.1 develops the two messaging protocols used in cloud.iO AMQP and MQTT; Subsection 2.4.2 details the Endpoint object and data model and Subsection 2.4.3 describes the backends of cloud.iO. Subsection 2.4.4 describes the cloud.iO core architecture. Finally Subsection 2.4.5 gives directions for the deployment of the core of cloud.iO.

[14] https://spring.io
[15] https://grafana.com

2.4.1 MQTT and AMQP

In cloud.iO, both AMQP and MQTT are used. Communications going from and to an Endpoint are made with MQTT. The Application can use both MQTT and AMQP, and finally, cloud.iO micro-services uses AMQP to exchange messages. Both protocols are based on the publish/subscribe pattern; they also both support the concept of topics. A topic will act as a filter for the message broker. We can publish or subscribe to a specific topic, and it is possible to add wildcards to it. Table 2 give examples of wildcard applied to AMQP and MQTT.

	MQTT	**AMQP**
Simple Topic	example/toto/titi	example.toto.titi
Single Level Wildcard	example/+/titi	example.*.titi
Multi-Level Wildcard	example/#	example.#

Table 2: Topic and wildcard example in MQTT and AMQP

Every MQTT and AMQP messages in cloud.iO arrive in RabbitMQ, the message broker. The routing of a message in RabbitMQ is represented in Figure 4. We first have the producer who will publish a message to an exchange of the message broker. It is now in charge of the exchange to route the message to the correct queue. Then the consumer will consume the message from the queue he did subscribe to.

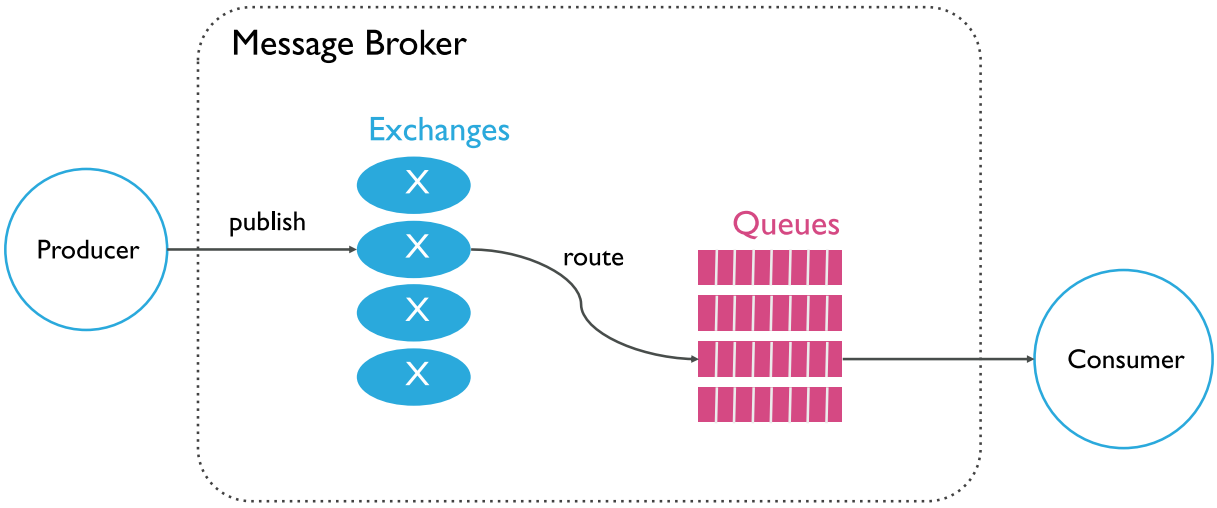


Figure 4: Routing of a AMQP message by a Message Broker

In RabbitMQ, MQTT is implemented as a plugin, and every message will then be routed, as explained in Figure 4. Every message will arrive in the *amq.topic* exchange. Then if we want to subscribe to a specific topic, RabbitMQ will create a queue and redirect the concerned message from the *amq.topic* exchange to the created queue.

cloud.iO also uses two features of MQTT, the Last Will and Testament (LWT), and the retained message. The LWT is a message that will be sent to the message broker before the client is disconnected; it is used in cloud.iO for the *@offline* message. The second feature is the retained message. It acts as a flag on a message and indicates that the message will be retained in the broker until a new message with the same topic arrives. If we subscribe to a specific MQTT topic, we will directly receive the last retained message at the subscription. This feature is used only in cloud.iO v0.2.

2.4.2 Endpoint Object and Data Model

As said earlier, Endpoints can be either an IoT gateway for multiple Things or a Thing supporting cloud.iO. The Endpoint needs to follow a specific data model. This model represented in UML can be seen on Figure 5. The structure starts with the Endpoint, which can contain one or multiple nodes. These nodes can be compared to a Thing or functionality of a Thing in the IoT world. Such a node contains objects whose can have either other objects or attributes as its children.

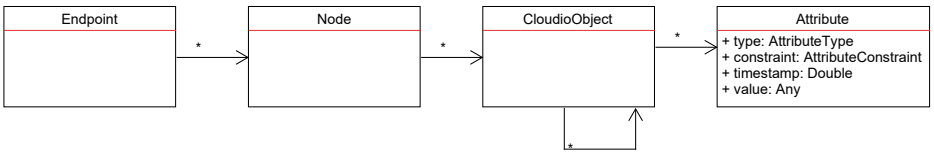


Figure 5: cloud.iO Endpoint object model

The very atomic element of this data model is the Attribute; we can see in Figure 5 that it contains four fields: type, constrain, timestamp, and value. We will review all those attributes; the first one, the type, can be one of the following:

- Bool
- Integer
- Number
- String
- Four above types as an array

The second field of the Attribute is the constraint field. It defines the kind of Attribute we are working with. The five different constraints are the following:

- **Static**: Read-only, Static value
- **Status**: Read-only, computed value
- **Measure**: Read-only, Measure representing a physical process
- **Parameter**: Read/Write, Parameter changing the Endpoint's behavior. Persists after reboots.
- **SetPoint**: Read/Write, Control value, non-persistent

Finally, we have the Attribute value and timestamp which goes in pair. The value must follow the Attribute type and is timestamped using epoch time.

In cloud.iO, it is possible to add some semantic to the data model with two concepts: The Node interfaces and Object classes. A Node can implement one or many Node Interfaces. They can be compared to Java interface with the difference that a Node Interfaces only define attributes, here Object classes, and not methods. A Node implements a Node interface if it contains, along with other objects, the structure of the interface. The second concept, the Object class, is used to enable an Object definition to be re-used. An Object is called conforms to an Object class if it contains precisely the Objects and Attributes declared in the Object class. An Object can only be conform to one Object class. One of the ideas behind those two concepts was to create a library of Node Interface and Object class that a Developer could use with the cloud.iO libraries. This library of Node interfaces has never been created. In the Java library, the concept of multiple Node interface cannot be done. Indeed, Java only support single inheritance and the attribute of a Java interface are final and static. It is up to the developer to put the same Objects in a Node implementing multiple interface and to be consistent with its cloud.iO architecture.

This data model is often serialized in JSON when using cloud.iO, for example, when we want to access the digital twin or when we want to update an Attribute. To illustrate this JSON serialization, we will take an example of a smart heating system. This system measures the ambient temperature and stores it in a Measure Attribute named `temperature` It will try to regulate this temperature according to a SetPoint Attribute named `setPointTemperature`. Those two Attributes are in an object called `temperatures` in a Node named `myHeater`. All this is in an Endpoint named by a unique UUID. This heater is represented in Listing 1.

```
{
  "id": "bc0f1bf8-bdae-11e9-9cb5-2a2ae2dbcce4"
  "online": false
  "blocked": false,
  "endpoint": {
    "nodes": {
      "myHeater": {
        "implements": [],
        "objects": {
          "temperatures": {
            "attributes": {
              "setPointTemperature": {
                "constraint": "SetPoint",
                "timestamp": -1.0,
                "type": "Number",
                "value": null
              },
              "temperature": {
                "constraint": "Measure",
                "timestamp": 1575547591858.0,
                "type": "Number",
                "value": 25.000000762939507
              }
            },
            "conforms": null,
            "objects": {}
          }
        }
      }
    }
  }
}
```

Listing 1: JSON serialization of Smart heating system

To interact with our Endpoint in cloud.iO v0.1 specific message following the detailled object model need to be send to specific topic. The structure of a topic is the following:

- @*prefix*/*EndpointUUID*/node/*nodeName*/object/*objectName*/.../attribute/*attributeName*

The presented structure does not need to finish with the attribute. For example action that act on the whole Endpoint will finish at the *EndpointUUID*. Table 3 show the differents topics used in cloud.iO v0.1 and their corresponding data. We have the @*update* and @*set* to update the attribute of an Endpoint and the @*online*, @*nodeAdded*, @*nodeRemoved* and @*offline* that represent the lifecycle of an Endpoint.

Topic	**Data**
@update/EndpointUUID/node/nodeName/object/objectName/.../AttributeName	The status or measure Attribute formatted in JSON
@set/EndpointUUID/node/nodeName/object/objectName/.../AttributeName	The setpoint or parameter Attribute formatted in JSON
@online/EndpointUUID	The Endpoint Data Model formatted in JSON
@nodeAdded/ EndpointUUID/node/ NodeNameToAdd	The Node to add formatted in JSON
@nodeRemoved/EndpointUUID/node/NodeNameToRemove	no Data
@offline/EndpointUUID	no Data

Table 3: Topic and the corresponding data to send to cloud.iO v0.1

As per usual we take back the heating system example. To update and set its Attribute, we can send message to the following topics:

- @update/bc0f1bf8-bdae-11e9-9cb5-2a2ae2dbcce4/nodes/myHeater/objects/temperatures/↪attributes/temperature
    - the Endpoint update the value of the temperature to the cloud
- @set/bc0f1bf8-bdae-11e9-9cb5-2a2ae2dbcce4/nodes/myHeater/objects/temperatures/↪attributes/setPointTemperature
    - an User or Application set the set point temperature to the Endpoint

Listing 2 is an example data of the *@update* message corresponding to the `temperature` Attribute.

```
{
  "constraint": "Measure",
  "timestamp": 1575547591858.0,
  "type": "Number",
  "value": 25.000000762939507
}
```

Listing 2: JSON serialization of temperature Attribute for an @update message

2.4.3 Backend

There are three types of data stored by cloud.iO v0.1: the authentication data for the users, the Endpoint with its device twin, and the history of the Endpoint. To save this data, two different types of database are used; a Document oriented database MongoDB[16] and a time-series database InfluxDB[17].

Figure 6 represents how the data are divided into the backend. The structure of the entities saved in the Authentication and Endpoint Repository is also shown in UML. In the History Repository, the value of an Attribute updated with *@update* is saved. All the messages with *@online*, *@offline*, *@nodeAdded*, and *@nodeRemove* are also saved as a time series.

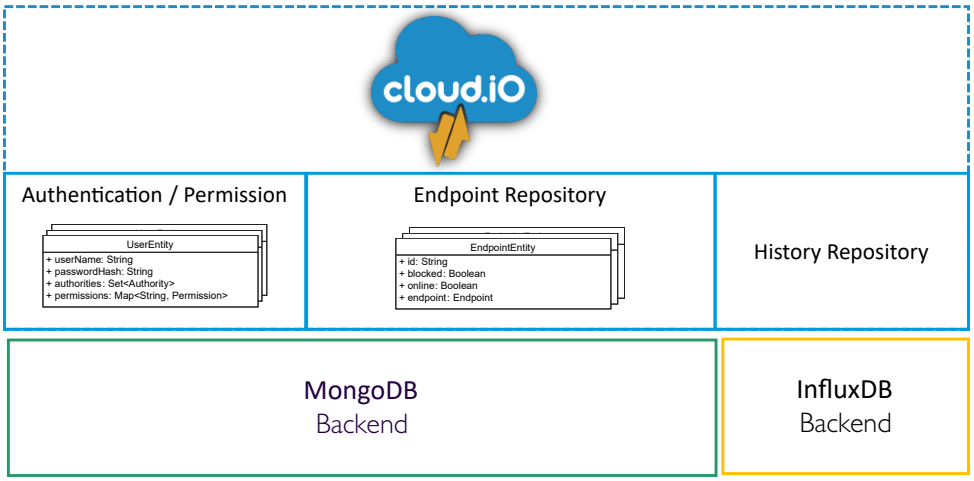


Figure 6: Backend Architecture of cloud.iO v0.1

In the case of the monitoring of data, it has already been noted that Grafana is used. With cloud.iO v0.1, Grafana is directly connected to InfluxDB to retrieve the time series value of the Attributes. In cloud.iO v0.1, Grafana is not aware of the permissions of a User, it is needed to create a new Grafana user and add specific access right to the InfluxDB database to access time series data only concerning the User's Endpoint.

[16] https://www.mongodb.com
[17] https://www.influxdata.com

2.4.4 Core Architecture

The core of cloud.iO is composed of three parts: RabbitMQ, the message broker, the micro-services using the Spring frameworks, and the backends with MongoDB and InfluxDB. Figure 7 represents those three elements and their interaction with input and Endpoint. To create a Service in Spring, we can simply use the `@Service` decorator on a class.

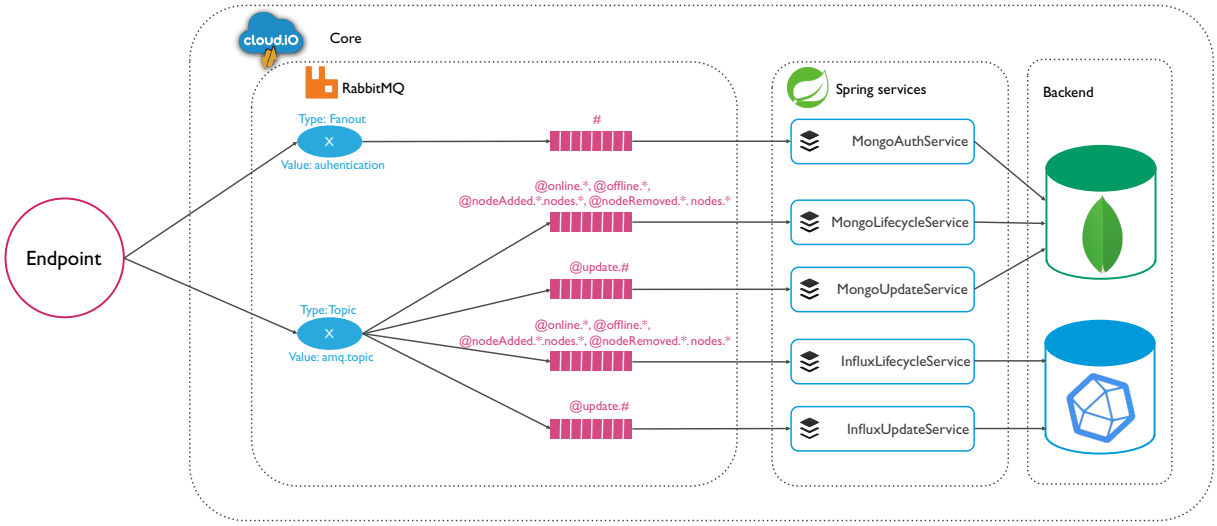


Figure 7: Architecture of the Core of cloud.iO v0.1

The Endpoint will send data in MQTT; this data will arrive in RabbitMQ exchange where specifics queue for specifics topic are instantiated. There is two exchanges used: *authentication* and *amq.topic*. The first one is used during the authentication of the MQTT publisher; here, the Endpoint. Every message of this exchange will be sent to a queue, and the *MongoAuthService* will check the Endpoint or User permissions in MongoDB. In any case of error, for example, wrong login, level of permission to low, etc., the service will deny the MQTT connection.

Then we have the *amq.topic* where every MQTT message with a topic is sent. Those messages will be redirected to four different queues according to their topic. Finally we have four services handling those messages:

- **MongoLifecycleService**: Update the Endpoint model according to *@online*, *@offline*, *@nodeAdded* and *@nodeRemoved* messages
- **MongoUpdateService**: Update the Endpoint model according to *@update* messages
- **InfluxLifecycleService**: Save the lifecycle of the Endpoint as time-series
- **InfluxUpdateService**: Save the update of the Endpoint as time-series

This concept is done in Spring by using the `@RabbitListener` function decorator. Listing 3 show the use of this decorator. In this example the function handleUpdateMessage will be linked to a queue receiving message filtered with the *@update.#* topic from the amq.topic exchange.

```
@RabbitListener(
bindings = [QueueBinding(value = Queue(),
exchange = Exchange(value = "amq.topic", type = ExchangeTypes.TOPIC,
    ↪ ignoreDeclarationExceptions = "true"),
key = ["@update.#"])])
fun handleUpdateMessage(message: Message) {
//handle message ...
}
```

Listing 3: Kotlin example of RabbitListener handeling @update.# messages

2.4.5 Deployment

The deployment of cloud.iO v0.1 is mainly done with docker[18] container. InlfuxDB, MongoDB, and RabbitMQ are deployed with this solution. A docker compose file is available to do the deployment[19]. For the RabbitMQ docker, a particular version of it has been developed including a patch of the 3.6.2 version of RabbitMQ. The patch was allowing to check the topic accessed during the authentication phase to RabbitMQ. All the dockers of cloud.iO v0.1 are now in deprecated version and need to be updated for cloud.iO v0.2.

For the micro-service, it is possible to launch them all on the same machine or separately on virtual machines, for example. For the services linked to an MQTT queue, and so an MQTT topic, RabbitMQ and Spring let the user instantiate multiple times a service according to the load of messages on this topic. For example, in cloud.iO v0.1, the MongoUpdateService has trouble to empty a queue with a high rate of message. If we instantiate multiple time this service, RabbitMQ will create multiple queues and route the message to each queue one by one with round robbin, this concept is illustrated in Figure 8.

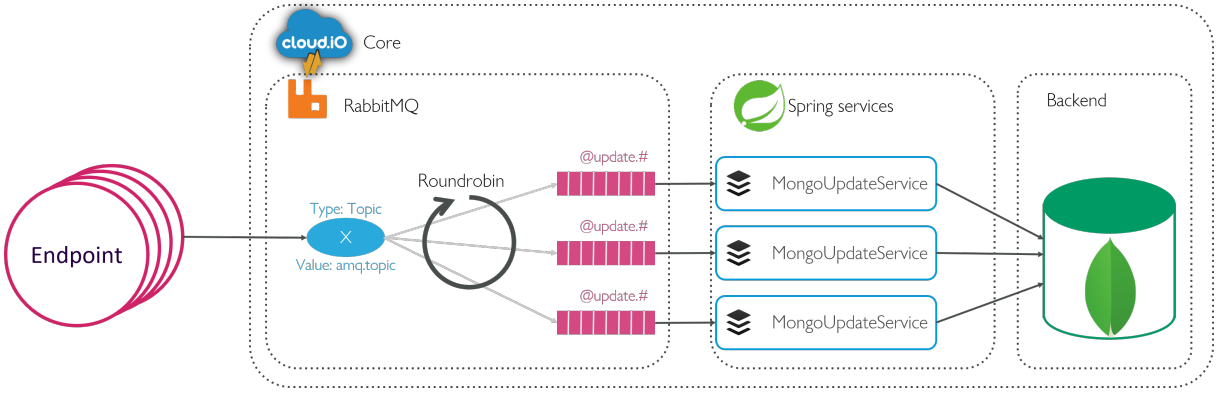


Figure 8: Multiple Instance of the MongoUpdateService using RabbitMQ roundrobin

A last note for the deployment is the management of the certificate. In cloud.iO v0.1, a makefile was created using openssl[20], an open-source cryptographic library. This makefile was used by cloud.iO administrators to create each Endpoint certificate manually.

[18] https://www.docker.com
[19] https://github.com/cm0x4D/cloudio/blob/master/cloud/cloudio-cloud-run/docker-compose.yml
[20] https://www.openssl.org

2.5 Enhance cloud.iO

The work of enhancing cloud.iO comes first with a feedback of users of cloud.iO. Then to complete the list of features for cloud.iO v0.2, a more in-depth analysis of the IoT landscape will be exposed. Different clouds will be evaluated against the features described from the state of the art.

2.5.1 Feedback from users

To enhance cloud.iO v0.1, feedbacks from the differents user of cloud.iO were taken. This panel of users goes from developers of cloud.iO, developers of Application using cloud.iO, to simple users.

Redefine roles:	The Application and User's roles are too similar. In cloud.iO v0.1 only the Users are technically implemented
Saving @set history:	Only the *@update* is saved in the History database when a setpoint or parameter is change, it is not logged
@update retained:	The *@update* give us the last state of the Thing, but when we subscribe to it in cloud.iO v0.1, we need to wait until the next *@update* to know the Thing state. A simple way to notify the actual state of the Thing at the subscription is to have the *@update* MQTT message retained.
Redifine topic & access control:	The topic structure can be simplified by passing from *endpointUUID*/nodes/*nodeName*/objects/*objectName*/.../attributes/*attributeName* to *endpointUUID*/*nodeName*/*objectName*/.../*attributeName*. Also the access control is only made on Endpoint, it would be more convenient to have the access control going down to the Attribute
Logging:	Debugging a cloud.iO Endpoint is not an easy task and often need external tools as ssh or a VPN. Having the log of the Endpoint sent on the cloud would be an excellent addition.
Update of Endpoint/Node/Object:	For an embedded system with low bandwidth, it would be useful to publish only one message updating the full Endpoints or multiple Attributes
Certificate creation:	The certificate creation is not optimized, it is needed to pass by an administrator and certificate creation is done one by one

2.5.2 IoT Cloud based framework analysis

In the state of the art, different functionalities about IoT cloud-based framework has been discussed; they will be evaluated in this subsection. Every cloud introduced by the state of the art are analyzed in this chapter. To this list of cloud, we add the kaa platform[21], who was the leader in open source IoT solution, is now selling a proprietary solution, and Siot[22], a project developed at the University of Applied Science in Bern with similar goals as cloud.iO.

The Table 4 is a resume of this analysis. The complete Table is available in Appendix A2.0.

IoT Platform	Device Monitoring	Device History	History Visualization	Rules Engine	Remote Robs Execution	Cloud Computing / Data Analysis	Own Cloud Hosting	On-premises Deployments	Open-Source	RESTful API	Interoperability
Azure IOT	x	x	x	x		x	x			x	
Aws IOT	x	x	x	x	x	x	x			x	
Google Cloud IOT Core	x	x	x		x	x	x			x	
Alibaba Cloud	x	x	x	x		x	x			x	
Eclipse Kapua	x	x	x					x	x	x	
thingsboard	x	x	x	x	x		x	x	x	x	
kaa platfrom	x	x	x		x		x	x		x	
mainflux labs	x	x	x	x				x	x	x	
Particle Cloud	x	x	x	x	x					x	
B+B SmartWorx	x	x	x								
Siot	x	x	x				x	x		x	
cloud.iO v0.1	x	x	x					x	x		

Table 4: IoT cloud features comparison

With Table 4, we can see that all the chosen framework have the first three critical features for an IoT cloud platform: Device Monitoring, Device History, and History visualization. The second feature is the Rules Engine. Since cloud.iO gives its own Endpoint Library, having a rules engine will add redundancy and is not an essential feature for it. The next feature is Remote Job Execution; it is not a pretty standard feature but can be an excellent addition for cloud.iO. Remote Job Execution would add the possibility to execute remote scripts or commands without the need for a direct connection to the endpoint. The current projects at the HEVs using cloud.iO need a VPN and ssh access for every action on the Endpoint. Remote Job execution can be a good tool to execute simple tasks on the Endpoint without the need for specific network infrastructure.

The next feature is Cloud Computing. We can see that this feature is only implemented by companies that also provide cloud computing as a standalone service. With the possibility of on-premise deployments, cloud computing comes as a difficult task without existing infrastructure and so cloud.iO is not in the capacity to offer cloud computing.

The two next features are Own Cloud Storage and On-premise deployments. We can correlate the possibility of on-premise deployment with the open source category. cloud.iO defines itself as an Internet of Things solution and not a cloud hosting solution. This is the reason we only provide on-premises deployment.

[21] https://www.kaaproject.org/overview
[22] https://siot.net

The next feature is the most implemented one: the RESTful API. This feature is a significant lack of cloud.iO v0.1. A RESTful API is a key feature for cloud.iO enhancement. It can add a lot of management tools, could open Endpoint monitoring for web development, and automatize different processes of cloud.iO, for example, certificate creation.

The last feature is interoperability, we can see that no framework proposes a solution for it. The lack of interoperability is a problem in the IoT environment, and cloud.iO v0.2 will give a solution to enhance its own interoperability. This solutions is detailed in subsection 2.6.

The technologies used by the selection of 12 IoT clouds has also been analyzed. Table 5 show the different SDK language proposed by the clouds Table 6 the API and messaging technologies used.

Those two tables confirm the choice of language for the cloud.iO v0.1 library: Java and Python. A good addition would be a C or C++ library; this would let cloud.iO be deployed on a micro-controller without OS. Then we can see that the messaging protocol the most used is MQTT, which is the choice for the communication of cloud.iO' Endpoint. Then come the RESTful technologies, which have already been identified as a crucial missing feature of cloud.iO v0.1.

SDK	N°of Cloud
Java	5
Python	4
C	4
Node.js	3
C++	2
Android	1
IOS	1
C#	1

Table 5: SDK programming language provided by IoT clouds

API/Messaging	N°of Cloud
MQTT	11
REST	10
MQTT over WebSockets	4
AMQP	2
AMQP over WebSockets	1
CoAp	4

Table 6: API and Messaging Protocol provided by IoT clouds

Linked to the messaging protocols, we have the message broker. Most of the 12 clouds analyzed have their own broker, for example, IoT hub for Azure IOT or Aws Message Broker for Aws IOT. Some cloud use already existing message broker, we can cite VerneMQ[23] used by Mainflux labs and Siot, Nats[24] used by Kaa platform and finally RabbitMQ used by cloud.iO.

[23] https://vernemq.com
[24] https://nats.io

Table 7 show different message broker and the number of company reportedly using it according to stackshare[25]. We can see that RabbitMQ is the most used message broker. RabbitMQ also provides the possibility to be used in clusters. With that ability, Google[7] has demonstrated it can handle over 1 million msg/sec.

All this information confirms that RabbitMQ is the right choice as a message broker; we cannot affirm it is the best choice since a comparison of each message broker would be needed, and this is not the goal of this work.

Message Broker	**N° Company Using ***
RabbitMQ	1232
CloudAMQP	18
Kafka	691
VerneMQ	6
Nats	33
ActiveMQ	52

Table 7: Message Broker and number of company using it accoarding to stackshare.io

[25] `https://stackshare.io` accessed the 16.10.2019

2.6 Interoperability

Interoperability can be defined as the ability of a system or a device to exchange and consume data from another system or device. The interoperability is important in an IoT environment and can touch plenty of different fields. For example, interoperability is needed between devices, from a device to a platform, or between applications.

We can differ from two different kinds of interoperability: semantic and syntactic.

- **Syntactic Interoperability**: Where two or more systems are able to exchange data successfully through compatible formats and protocols. Syntactic interoperability can be done with the help of formatting standards such as XML, JSON, SQL, etc.
- **Semantic Interoperability**: Where two or more systems are able to exchange data with unambiguous, shared meaning. The exchanged data can be interpreted automatically. Syntactic interoperability is necessary to achieve Semantic Interoperability.

In this document, we will focus on the interoperability between applications and platforms, and how an application using the cloud.iO solution can benefit from it. This allows an application to exchange data to any IoT devices connected through different IoT platforms without dedicated adapters. The interoperability we are trying to reach is described by Figure 9. In this example, two applications interacting with Things through their digital twins provided by three different IoT platforms. If those twins have a normalized syntax, the applications understanding it are interoperable with any IoT platform providing those normalized Digital Twin. On the other hand, the two applications can interact together to share access to their Things through their Digital Twins. These interactions are possible since they both understand the normalized syntax of the Digital Twins.

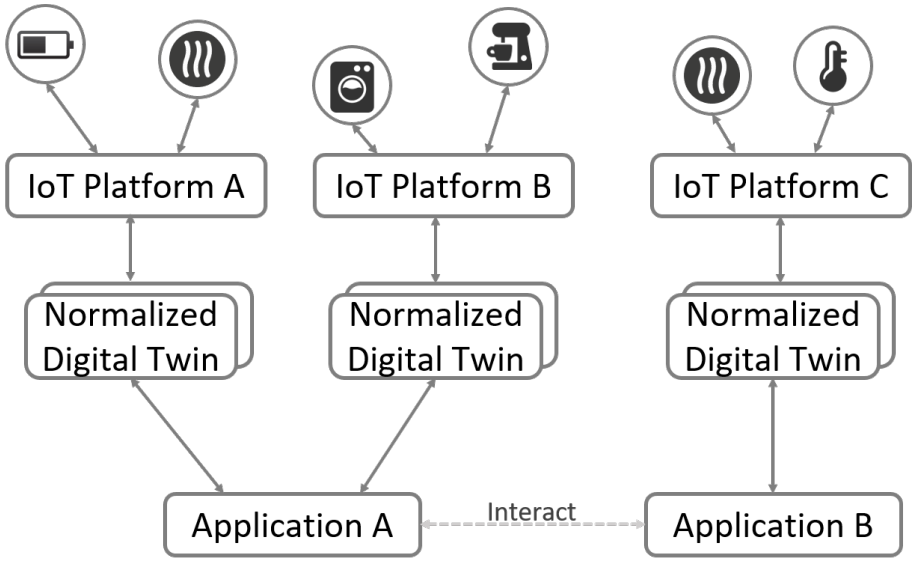


Figure 9: Interoperability between applications and Iot platforms

In the Internet of Things, lack of interoperability is a well-known problem. The growing number of connected devices and IoT platforms with non-standardized description amplify this problem. To this day, developing an IoT application, including an IoT cloud-based solution without interoperability, will lead to an insular solution. This solution will be strongly dependent on the chosen IoT platforms and their syntaxes and/or semantics.

To answer to the interoperability problem, different projects have been created. We can first cite the symbIoTe [8] solution, whose goal is to bring interoperability for cloud-based IoT platforms. SymbIoTe provides an abstraction layer to unified the different IoT platforms. Those platforms can then register their resources to symbIoTe. Resources can then be accessed by an application using the symbIoTe semantic search engine. We can note that symbIoTe was already interested in standardization since the symbIoTe Core Information Model is based on Semantic Sensor Network Ontology, a W3C recommendation, and OGC SensorThings API.

Another project proposing interoperability for the IoT is BIG IoT [9]. BIG IoT's goal was to build an IoT Ecosystem with multiple IoT platforms, services, and applications. For example, a platform in the will to bring interoperability

to its system could implement the BIG IoT API. By doing so, the platform services and applications would be accessible through the BIG IoT marketplace. The BIG IoT API and its information models have been built in collaboration with the Web of Things Interest Group at the W3C.

Both projects bring a solution for interoperability in an IoT environment. For symbIoTe, platforms need to implement the symbIoTe Interworking Interface, and for BIG IoT, platforms need to implement the BIG IoT API. It is up to an IoT platform to implement those solutions, and since they are non-standardized, these implementations are a risk for an IoT platform. This is why following the recommendation of a standards organization like W3C has been the chosen solution for cloud.iO.

2.6.1 Web Of Things

The World Wide Web Consortium (W3C) is a non-lucrative standardization organism created in 1994. W3C is working mostly for the compatibility of the World Wide Web. We can, for example, cite HTML, RDF, CSS, PNG, or SVG which are part of their work.

W3C is actively working on the standardization of the IoT called Web of Things (WoT) to enable interoperability across IoT Platforms and application domains. The work on WoT is held by Web of Things Interest Group (WoT IG, stakeholders interested in the Web of Things) and the Web of Things Working Group (WoT WG, W3C member in charge of WoT standardization). The latter has released two candidate recommendations: the Web of Things Architecture[10] and the Web of Things Descriptions[11].

WoT is composed of four building blocks: WoT Thing Description, WoT Binding Templates, WoT Scripting API, and WoT Security and Privacy Guidelines. The Wot Thing Description or TD describes a Thing using JSON-LD formatting. JSON-LD (JavaScript Object Notation for Linked Data) is an implementation of RDF (Resource Description Framework), a specification from the semantic web standard by W3C intended to formally describe Web resources and their metadata to allow automatic processing of such description. JSON-LD is designed under the concept of context introduced by the `@context` JSON field. The `@context` can be composed of a map of terms to IRIs composing the JSON-LD resource vocabulary or a simple IRI that will be dereferenced to obtain a full JSON-LD context file and so the full IRI-based Vocabulary Terms. The processing of those IRIs is part of the process of transforming JSON-based TD documents to RDF.

The JSON-LD TD contains metadata. An essential part of those metadata and Interaction Affordances describing how an application can interact with Things. The Interaction Affordances are linked to network locators (e.g. URLs) (communication protocols) through Binding Templates. The entity that interacts with a TD is called a Servient. A Thing can either be exposed or consumed by a Servient using a TD. A Servient consuming a Thing is called a Consumer. Figure 10 shows a simple architecture involving Servient and Thing Description.

Figure 10: Servient, Consumer, Thing and TD architecture from Web of Things Architecture[10]

To create an IoT application, W3C introduces the WoT Scripting API, a non-normative specification. It describes how to work with a Thing by using its TD. The WoT Scripting API describes how to discover, operate, and expose a Thing. We already introduce three Building blocks of WoT: WoT Thing Description, WoT Binding Templates, and WoT Scripting API. The last building block is the WoT Security and Privacy Guidelines. Every part of the system design is concerned with security, and W3C describes non-normative guidelines for those security aspects.

2.6.2 Thing Description

The Thing Description is the key building block of W3C WoT. In a TD is we can find those different components: textual metadata to help describe the Thing, Interaction Affordances, and data schemas that represent the data exchanged with the Thing and finally Weblinks.

As mentioned above, Interaction Affordances are an important part of a TD: They describe possible interactions with a given Thing. Interaction Affordances are divided into three categories: Property, Action, and Event. A Property describes a state of the Thing; it must be readable and may be writable by an application. An Action invokes a function on the Thing. Performing an Action may change the state of the Thing. The last Interaction Affordance category is Event, which describes asynchronous data exchange between a Thing and a TD Consumer. Interaction Affordances and Protocol Bindings form a self-describing API for the Thing.

We can go back to the smart heating system example for an applied case of TD. Both the real and target temperatures are Properties of the TD: they can be accessed by a hypothetical HTTP API and are both of integer type. On our system, we can advise the heating system to make a regulation to follow the set point or not. Toggling the regulation on the heating system can be described as an Action in the TD. Finally, our heating system will detect every modification of the ambient temperature. This information can be transposed as an Event in the TD. Listing 4 represents a basic implementation of this smart heating system TD.

```
{
  "@context": "https://www.w3.org/2019/wot/td/v1",
  "id": "urn:heater",
  "title": "myHeater",
  "securityDefinitions": {
    "basic_sc": {
      "scheme": "basic",
      "in": "header"
    }
  },
  "security": ["basic_sc"],
  "properties": {
    "temperature": {
      "type": "integer",
      "forms": [{
        "href": "https://myHeater.com/get"
      }]
    },
    "setPointTemperature": {
      "type": "integer",
      "forms": [{
        "href": "https://myHeater.com/set"
      }]
    }
  },
  "actions": {
    "toggleRegulation": {
      "forms": [{
        "href": "https://myHeater.com/toggle"
      }]
    }
  },
  "events": {
    "temperatureChange": {
      "data": {
        "type": "integer"
      },
      "forms": [{
        "href": "https://myHeater.com/change",
        "subprotocol": "longpoll"
      }]
    }
  }
}
```

Listing 4: TD example of a heater system

The Table 8 explains the differents fields of Listing 4 described by the Thing Description Candidate Recommendation. To follow the WoT TD Candidate Recommendation, all the mandatory fields are presents; `@context`, `title`, `securityDefinitions`, and `security`. Only the optional fields used for the smart heating system example are described in Listing 4. The exhaustive list of Vocabulary terms can be found online[26].

Vocabulary term	Description	Assignment	Type
@context	JSON-LD keyword to define short-hand names called terms that are used throughout a TD document.	mandatory	anyURI or Array
id	Identifier of the Thing in form of a URI [RFC3986] (e.g., stable URI, temporary and mutable URI, URI with local IP address, URN, etc.).	optional	anyURI
title	Provides a human-readable title (e.g., display a text for UI representation) based on a default language.	mandatory	string
properties	All Property-based Interaction Affordances of the Thing.	optional	Map of PropertyAffordance
actions	All Action-based Interaction Affordances of the Thing.	optional	Map of ActionAffordance
events	All Event-based Interaction Affordances of the Thing.	optional	Map of EventAffordance
securityDefinitions	Set of named security configurations (definitions only). Not actually applied unless names are used in a security name-value pair.	mandatory	Map of SecurityScheme
security	Set of security definition names, chosen from those defined in securityDefinitions. These must all be satisfied for access to resources.	mandatory	string or Array of string

Table 8: Fields of a Thing Description, taken from Thing Description Candidate Recommendation[11]

[26] https://www.w3.org/TR/wot-thing-description/#thing

2.7 Synthesis of Analysis

In the state of the art, we defined the critical feature for an IoT cloud platform. Those features helped us to understand more the IoT cloud landscape and where it is going. Then with the analysis of cloud.iO, we did define a roadmap for its enhancements. From the feedback of users, we retrieved the following enhancement: Redefine roles, saving *@set* history, *@update* retained, topic & redefine access control, logging, update of Endpoint/Node/Object, and certificate creation. All those features will be added to cloud.iO v0.2.

By comparing cloud.iO to other clouds, the main missing features are the Remote Job Execution, RESTful API, standardization (following WoT), and an SDK in other programming languages (C or C++ has been cited). Those features are important for cloud.iO; therefore, creating a complete cloud.iO library in another language is a full-blown task. This will not be part of this thesis. The two other features, Remote Job Execution, and RESTful API will be described in this document.

The most important part of this enhancement is the RESTful API. It will give tools for every functionality of cloud.iO. The RESTful API can help with User Management, Endpoint Management, Certificate Creation, History database access, Logs access, and Remote Job execution.

3 Conception

3.1 Preliminary tasks

3.1.1 Update of cloud.iO components

The first step of before implementation of new features to cloud.iO v0.2 is to re-create a cloud.iO core with all the components updated. Table 9 show the different components of cloud.iO and their versions. The first four components are from the cloud.iO core, and the last two are Java libraries used in the Endpoint SDK. The update of RabbitMQ to its last version allowed us to get rid of the patch needed in the previous version. Indeed RabbitMQ provides now natively the possibility to check the topic accessed during the authentication phase, allowing us to get rid off the patch developed for cloud.iO v0.1. A new version of the RabbitMQ docker image has been configured and is available on GitHub[27] and dockerhub[28]. The update of Spring Framework and Jackson core (JSON library for Java) induces no change to cloud.iO. In addition, the first version of cloud.iO was already using the latest release of InfluxDB and MongoDB docker. The updated code has been tested with the latest version of their docker images; no updates were needed in the cloud.iO core implementation. Finally, the Paho version used in the cloud.iO Java Endpoint library has been updated to version 1.2.2. This Paho update adds HTTPS hostname verification. It is important to take this feature into account when creating a server certificate for the message broker. In any case, the possibility to disable the HTTPS hostname verification has been added to the cloud.iO Java Endpoint library; these features should only be used during development phase.

Componants	Componant version for cloud.iO v0.1	Componant version cloud.iO v0.2
RabbitMQ docker	3.6.2	3.7.23
Spring Framework	1.5.8.RELEASE	2.1.5.RELEASE
InfluxDB	latest	latest
MongoDB	latest	latest
Jackson core	2.6.4	2.10
Paho	1.2.0	1.2.2

Table 9: Version of components used in cloud.iO for the v0.1 and v0.2 versions

The previous version of the cloud.iO core was separated into two Kotlin project: cloudio-cloud-backend-mongo-influx and cloudio-cloud-core. In the new implementation, only one project has been created, re-grouping the backend and the core in the same place. All the configuration of the core is done through the application.properties files. We can define there, for example, the RabbitMQ configuration (host, port, username, and password), the MongoDB and InfluxDB configuration (host, port), and add cloud.iO specific configuration. An example of application.properties file is shown in Appendix A3.0.

3.1.2 cloud.iO v0.2 Roles

There are 3 roles in cloud.iO v0.1: the Endpoint, the User, and the Application. Therefore the implementation of User was minimalistic (only access control), and the implementation of Application was not existing.

In cloud.iO v0.2, an Application can be assimilated as a User. Their purpose is the same: accessing and or modifying Endpoints. cloud.iO v0.2 will have new tools only accessible for Users, all coming with the RESTful API. The access to the RESTful API can be restricted to some Users. This restriction is done with the authorities assigned to Users; two new authorities are defined to this end:

- **http_access**: the User can access the regular RESTful API
- **http_admin**: the User can access additional admin tools from the RESTful API

[27] https://github.com/cloudio-project/cloudio-rabbitmq-docker
[28] `https://hub.docker.com/r/cloudio/cloudio-rabbitmq`

3.1.3 @set and @update

In cloud.iO v0.1, only *@update* messages were saved in InfluxDB and updated in MongoDB. Only the status and measure Attributes had history leaving the setpoint and parameter linked to the *@set* without history. To solve this problem, the core services responsible for updating the Attribute has been modified to react also to the *@set* messages. This is done by adding the *@set* key in the RabbitListener `key = ["@update.#", "@set.#"]`. With this new RabbitListener, we need to verify that the message we are receiving from *@update* is either a Measure or a Status, and from *@set* is either a Parameter or a SetPoint. The Listing 5 show a RabbitListener reacting to *@set* and *@update* message with the verification of the Attribute constraint as seen on line 17.

```
1  @RabbitListener(
2  bindings = [QueueBinding(value = Queue(),
3  exchange = Exchange(value = "amq.topic", type = ExchangeTypes.TOPIC,
       ↪ ignoreDeclarationExceptions = "true"),
4  key = ["@update.#", "@set.#"])])
5  fun handleUpdateMessage(message: Message) {
6    val prefix = message.messageProperties.receivedRoutingKey.split(".")[0]
7
8    val attributeId = message.messageProperties.receivedRoutingKey.removePrefix("$prefix.")
9
10   val data = message.body
11
12   val attribute = Attribute()
13   JsonSerializationFormat.deserializeAttribute(attribute, data)
14
15   ...
16
17   if (prefix.equals("@update") && (attribute.constraint == AttributeConstraint.Measure ||
       ↪ attribute.constraint == AttributeConstraint.Status) ||
18   (prefix.equals("@set") && (attribute.constraint == AttributeConstraint.Parameter ||
       ↪ attribute.constraint == AttributeConstraint.SetPoint))) {
19     attributeUpdatedSet(attributeId, attribute, prefix)
20   } else {
21     log.error("The Attribute $attributeId with the constraint ${attribute.constraint} can't be
       ↪  changed with the prefix $prefix")
22   }
23 }
```

Listing 5: Kotlin example of RabbitListener handeling *@update.#* and *@set.#* messages

3.1.4 Topic

In cloud.iO v0.1 the structure of topics was following this model:

- @*prefix*/*EndpointUUID*/nodes/*nodeName*/objects/*objectName*/.../attributes/*attributeName*

This structure contains redundancy in its model. Indeed the model of the Endpoint is fix and so in a topic structure we will always go from then Endpoint UUID, to the Node name, to one or more Object names and finishing with the Attribute name. With that in mind we can define the new topic model like so:

- @*prefix*/*EndpointUUID*/*nodeName*/*objectName*/.../*attributeName*

We can now redefine the cloud.iO v0.2 topics as seen in Table 10

Topic	Data
@update/*EndpointUUID*/.../AttributeName	The status or measure Attribute formatted in JSON
@set/*EndpointUUID*/.../AttributeName	The setpoint or parameter Attribute formatted in JSON
@online/*EndpointUUID*	The Endpoint Data Model formatted in JSON
@nodeAdded/*EndpointUUID*/*NodeNameToAdd*	The Node to add formatted in JSON
@nodeRemoved/*EndpointUUID*/*NodeNameToRemove*	no Data
@offline/*EndpointUUID*	no Data

Table 10: Topic and the corresponding data to send to cloud.iO v0.2

3.1.5 @update retained

The main goal of using the retained feature of MQTT with the @*update* messages is to let the user know what is the last state of the Attributes when subscribing to their topic. Without retained messages, we would need to wait until the next @*update* message to know the Attribute state, which is inconvenient in the case of low-frequency update from the Endpoint.

This modification touches the Endpoint libraries on cloud.iO and specifically how we send MQTT messages with them. The Java library use Eclipse Paho[29] open-source client implementations of MQTT. Listing 6 shows how the Java library publishes an @*update* message. It is using the publish method of the MqttAsyncClient object, which takes 4 arguments: The topic, the data to be sent, the QoS, and the retain flag. By setting this retain flag to true, every @*update* message will be retained.

```
1 MqttAsyncClient mqtt;
2 mqtt = new MqttAsyncClient(...); //initialisation of the MQTT client
3
4 mqtt.publish("@update/" + attribute.getUuid().toString(), data, 1, true); //publish of @update
    ↪ message
```

Listing 6: Java Example of MQTT publish with retained flag set to true

[29] https://www.eclipse.org/paho/

3.2 Access Control

In cloud.iO v0.1, the access control of Users was based on permission on Endpoints. This access control was limited but straightforward. An access control going from the Endpoint to the Attribute would let more flexibility about the User Permission. In cloud.iO, access to resources like attribute is based on the topic. This is why having an access control following the topic model with the use of topic wildcard character in topics can be convenient; it would permit to choose the permission from the Endpoint to the Attribute. For example, we could allow access on a whole Endpoint by giving Write permission to a User to the *EndpointUUID/#* topic and then restrain only Read Permission to a specific Attribute by giving its full topic, e.g. *EndpointUUID/NodeName/ObjectName/AttributeName*.

This design offers a lot of flexibility to have different access right inside the same Endpoint. It still has a problem: Two topics can point to the same object in the Endpoint Structure and have different rights, for example, if we want to access the following Attribute: *EndpointUUID/NodeName/ObjectName/AttributeName* and the User has Deny Permission on *EndpointUUID/NodeName/*/AttributeName* but Write Permission on *EndpointUUID/NodeName/#*. To overcome this problem, a priority has been added to the permission. In case of conflict, the cloud.iO core will take the permission with the highest priority. Four levels of priority for permission are introduced:

- Low
- Medium
- High
- Highest

The User permission now consists of a map of prioritized permission: a priority and a permission level. The keys of this map are MQTT topics whose can contain wildcard character. The topic wildcard can be either following MQTT or AMQP format; * or + for single-level wildcard and # for multilevel wildcard. The topic can only be a "/" as in MQTT, owing to the serialization of the User in MongoDB. Indeed, MongoDB does not recommend using dots in field names[30] due to support of some special character in the query language. Those permissions will be checked every time the user tries to access a resource via MQTT, AMQP, or the RESTful API. The permission levels are the same as in cloud.iO v0.1:

- Deny
- Read
- Write
- Configure
- Grant
- Own

[30] `https://docs.mongodb.com/manual/reference/limits/#Restrictions-on-Field-Names` accessed the 29/12/2019

Listing 7 show an example of permissions for a User serialized in JSON, taking the smart heating system, in example. In this case, a second heating system called mySecondHeater is added as a Node. In this example, we have 3 permissions coexisting. The lowest level of permission on line 3 gives access to the whole Endpoint with Reading right. The second-lowest permission on line 7 precise that the permission of mySecondHeater Node is DENY. Finally, the highest level of permission on line 11 changes the level of permission of a specific Attribute to READ.

```
1  {
2    "permissions": {
3      "endpointUUID/#": {
4        "permission": "WRITE",
5        "priority": "LOW"
6      },
7      "endpointUUID/mySecondHeater/#": {
8        "permission": "DENY",
9        "priority": "HIGH"
10     },
11     "endpointUUID/myHeater/temperatures/setPointTemperature": {
12       "permission": "READ",
13       "priority": "HIGHEST"
14     }
15   }
16 }
```

Listing 7: Example of prioritized permission based on topic

The class diagram in Figure 11 shows a diagram class of our heating system with two Nodes. We can see the effect of the permission from Listing 7 on it.

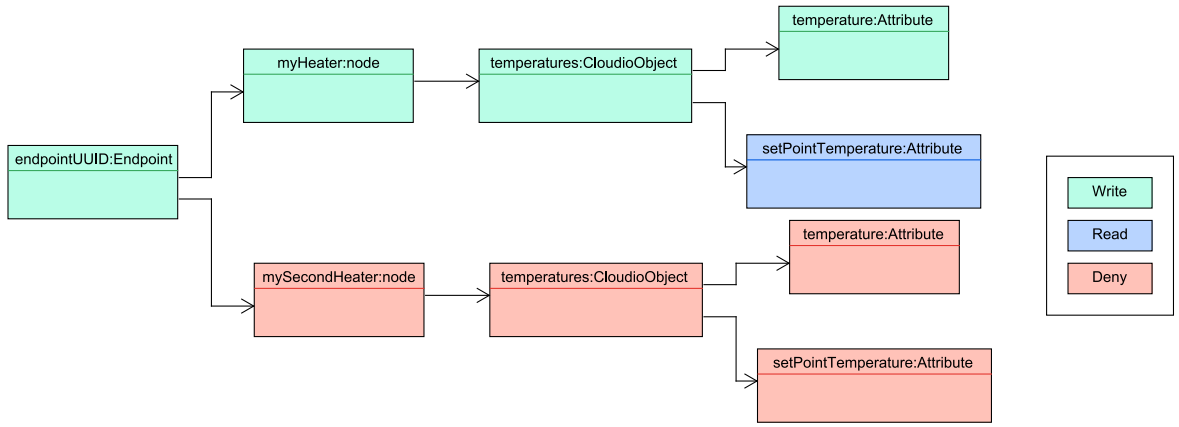


Figure 11: Class Diagram representation of Permission induced by Listing 7

3.2.1 User Group

To facilitate the User Management and User access control, we introduce the concept of User Group in cloud.iO v0.2. A User Group can be assigned to any User and has a map of Prioritized Permission like the User. The User Group is so composed of three attributes: the User Group name, the User list, and the map of permissions.

To facilitate the retrievement of all the permissions linked to a User, the User contains a list of all the User Group of which it is a part.

Figure 12 show a class diagram with all the class linked to the user access control. We have the UserRepository and UserGroup with all their attributes. All the instances of those classes are saved in MongoDB and are used for the cloud.iO access control.

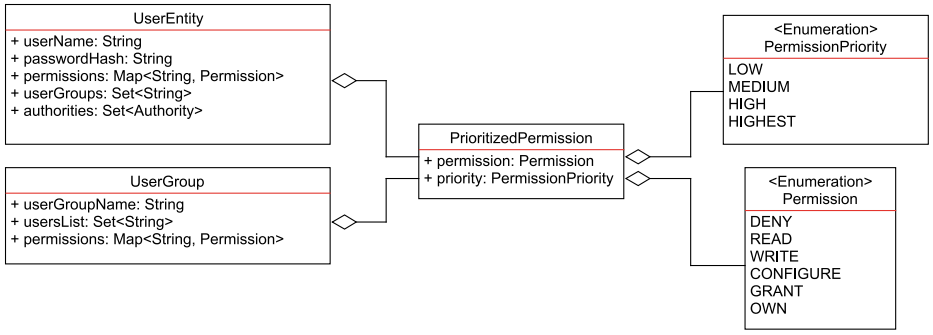


Figure 12: Class diagram of UserEntity and User Group for User Access Control

3.3 Remote Job Execution

The remote access of an Endpoint can be cumbersome once it is deployed. For example, we would need a remote shell using SSH or executing some remote procedure call (RPC) via Ethernet. Depending on where the Endpoint is in the Network, the use of both remote access technique may need a VPN.

Though if the Endpoint is connected to cloud.iO, we already have a communication channel open that could support some kind of remote access: MQTT. By publishing and subscribing to specific message from the User or the Endpoint, we can create remote access to the Endpoint without the need for any particular network infrastructure.

By using MQTT, we can send commands from a User to an Endpoint. With our solution, we can send specific commands and also start the execution of a shell script on the Endpoint. Those commands or execution of scripts are called Jobs in cloud.iO, and so cloud.iO provide Remote Job Execution via MQTT. The Remote Jos Execution is also possible via the RESTful API; this implementation is covered in Subsection 3.7. Only the Users with OWN permission on the desired Endpoint can execute a Remote Job on it. The topics used for the MQTT implementation are shown in Table 11.

Topic	Data
@exec/*EndpointUUID*	Job Parameter formatted in JSON as seen in Listing 8. User publish on this topic, Endpoint subscribe.
@execOutput/*EndpointUUID*	Job output line formatted in JSON by line as seen in Listing 9. Endpoint publish on this topic, User subscribe.

Table 11: Topic and the corresponding data to send to cloud.iO v0.2 for the Remote Job Executions

```
{
  "jobURI": "cmd://listJobs",
  "correlationID": "1a45a6ca",
  "data": "",
  "sendOutput": true
}
```

Listing 8: JSON serialization of JobParameter to send on @exec topic

```
{
  "data": "This is an output of Remote Job Execution",
  "correlationID": "1a45a6ca"
}
```

Listing 9: JSON serialization of JobsLineOutput to send on @execOutput topic

Listing 8 shows the message sent from the User to the Endpoint. The first field is the `jobURI`, a string in URI format. The second field is the `correlationID`, who is also in the Jobs Output. This ID can be used to retrieve the correct output message corresponding to a Job execution. The third field is `data`; it can be used if a Job needs data as input. The last field is `sendOutput`. This boolean field indicates to the Endpoint if it needs to stream the output of the Job in @execOutput MQTT messages. For the jobURI of the Jobs parameters, cloud.iO only support two scheme: *file* and *cmd*.

With *file*, we can execute a file (shell script). This file need to be in the folder specified in the *.properties* of the Java project; `ch.hevs.cloudio.endpoint.jobs.folder = /PATH/TO/JOBSFOLDER`. If the User does not define a path for the jobs folder, the Java library will assume that the available files for jobs are in /etc/cloud.io/. The output of the commands are sent in and *@execOutput* message line by line. It has been chosen to send the output line by line. By doing so, the User does not have to wait for the execution of the script to have outputs of the launched Job.

With *cmd*, we can execute commands implemented in the cloud.iO library. Two commands are already implemented: cmd://listJobs and cmd://updateJobs. *lisJobs* show the available jobs in the jobs folder and also list the available command cited previously. *updateJobs* let the user update its list of script files by downloading a zip file on the Web and unzip it in the jobs folder. This command needs the zip file URL as data. The *cmd* will send the output of the command *@execOutput* message line by line as for the *file* scheme.

Listing 9 show the output message of a job. The Endpoint will send those messages line by line to the *@execOutput* message. Those output messages have two fields, the `data`, and the `correlationID`. `data` consists of a simple string corresponding to one line of the output, and the `correlationID` is the same ID as in the Job Parameter request.

A full sequence of a Remote Job execution is shown in Figure 13. The User will launch the execution of the cmd://listJobs job with an *@exec* message and the Endpoint will send the output of this job in multiple *@execOutput* messages.

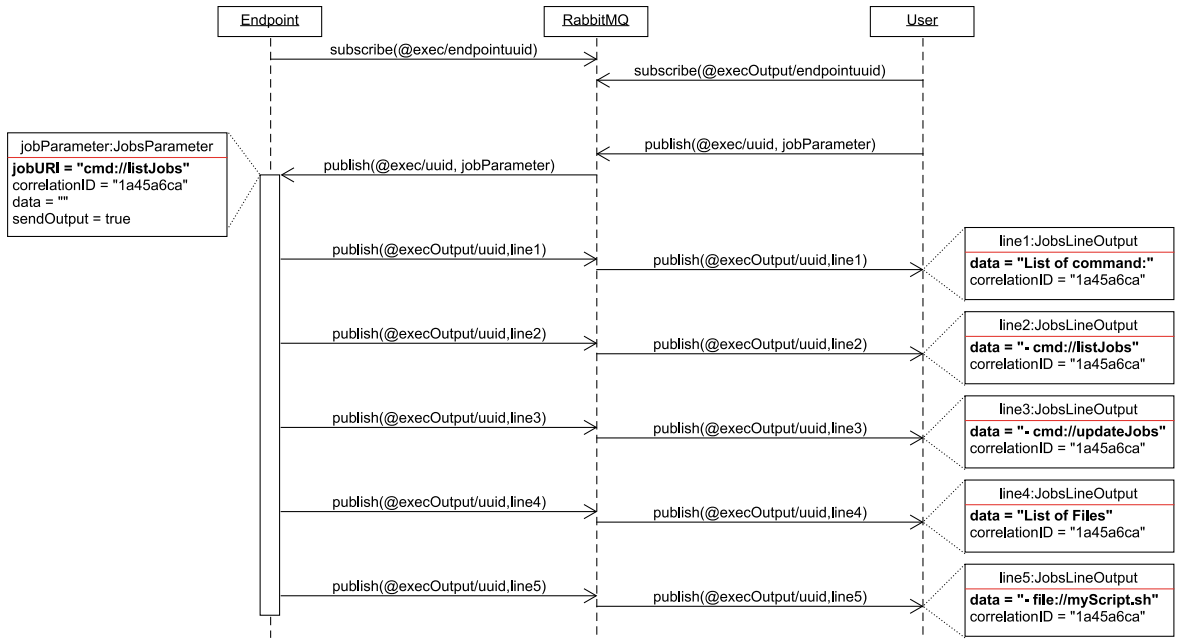


Figure 13: Sequence Diagram of an Remote Job execution

The implementation of the Remote Job Execution does not need work on the cloud.iO core side, only on the Java library. Indeed, on the core side, the access control already verifies the User permission on the Endpoint, and it is up to a User to do its own implementation of to execute Remote Job as its will. For the Java library, the first step is to subscribe to the *@exec* topics with Paho `mqtt.subscribe("@exec/" + internal.uuid , 1);`. The Endpoint will then receive all the messages it is subscribed to in the overridden `void messageArrived(String top\ ↪ right ic, MqttMessage message)` method that can be obtained by implementing the Paho `MqttCallback` class. In this class, we can filter the *@exec* message and execute the desired Jobs launched by a User.

3.4 Logging

When using the Endpoint library to create the logic of the Endpoint, a developer will surely create logs to know the lifecycle of its application, to help in any case of error, or in general log all the event linked to the Endpoint.

In the previous version of cloud.iO, it was up to the developer to use a logging system and to save those logs. Many logging libraries exist with different techniques to save the logs, like in a file, in a database, etc. In cloud.iO v0.2, we introduce the possibility of sending logs directly to the cloud with MQTT. The logs are saved in InfluxDB and so can be retrieved at any time. cloud.iO logging system is based on Apache Log4j 2[31], logging utility under the free Apache License 2.0. Like in any logging system, Log4j and so cloud.iO works with logs level. Log4j defines the following ones:

- OFF
- FATAL
- ERROR
- WARN
- INFO
- DEBUG
- TRACE
- ALL

Those levels define the severity of a log message. They are ordered in decreasing order of severity. We can set the severity level of a system, and so an Endpoint, to the desired log level. By changing the log level, we set the lowest severity level log message that will be logged and so, sent to the cloud via MQTT. The set of the log level can also be done with MQTT. The two topics dedicated to the logging are shown in Table 12. The log level is added to the Endpoint Model and so saved in MongoDB at every change.

Topic	Data
@logs/*EndpointUUID*	cloud.iO log message formatted in JSON as seen in Listing 10. User publish on this topic.
@logsLevel/*EndpointUUID*	Log level formatted in JSON as seen in Listing 11. User publish on this topic, Endpoint subscribe.

Table 12: Topic and the corresponding data to send to cloud.iO v0.2 for the logging

```
{
  "level": "ERROR",
  "timestamp": 1577802897012,
  "message": "Error occured while parsing data",
  "loggerName": "HeaterLogger",
  "logSource": "Heater/main, line 27"
}
```

Listing 10: JSON serialization of CloudioLogMessage to send on @logs topic

```
{
  "level": "TRACE"
}
```

Listing 11: JSON serialization of LogParameter to send on @logsLevel topic

Listing 10 show a log message sent from the Endpoint to cloud.iO. Its first field is the log `level` following Log4j levels. The second field is an epoch `timestamp`. The next field is the log `message`. Then we have the `loggerName` who will take the Log4j logger name and finally the logSource showing where the log appeared. In the Java library implementation, the logSource is structured like so: *ClassName/method, line lineNumber*. Not every logging system can retrieve that information (other libraries, other programming languages) and so this structure is not fix.

Listing 11 shows the message to send to change the log level of an Endpoint; it only contains the field `level`.

[31] https://logging.apache.org/ /2.x/

The implementation of the logging system in the Java library is made by creating a custom Log4j Appender. An appender is in charge of taking all the log created and append them in its specified support. Log4j provides, for example, console, file, HTTP, or NoSQL appender. Log4j let us create our own appender; we can create them by extending the `AbstractAppender`. This is the chosen solution for cloud.iO: a custom appender taking the logs created with Log4j and sending them via *@logs* message through MQTT. A simplified version of the custom cloud.iO appender is shown in Listing 12. The log appending, and so log sending via MQTT to cloud.iO, can be seen on line 14.

```
1  @Plugin(name = "CloudioLogAppender",
2          category = Core.CATEGORY_NAME,
3          elementType = Appender.ELEMENT_TYPE)
4  public class CloudioLogAppender extends AbstractAppender {
5
6    @PluginFactory
7    public static CloudioLogAppender createAppender(
8            @PluginAttribute("name") String name,
9            @PluginElement("Filter") Filter filter) {
10       return new CloudioLogAppender(name, filter);
11   }
12
13   @Override
14   public void append(LogEvent event) {
15
16     //build the log message
17     CloudioLogMessage cloudioLogMessage = new CloudioLogMessage(
18             event.getLevel().toString(),
19             (double)event.getTimeMillis(),
20             event.getMessage().getFormattedMessage(),
21             loggerName,
22             event.getSource().getClassName()+"/"+
23                     event.getSource().getMethodName()+
24                     ", line:"+event.getSource().getLineNumber()
25     );
26
27     //send the mqtt log message
28     mqtt.publish("@logs/" + uuid,
29             messageFormat.serializeCloudioLog(cloudioLogMessage), 1, false);
30   }
31 }
```

Listing 12: Simplified version of CloudioLogAppender

In Log4j, we can specify appenders to logger dynamically to any class, however by default it is made hierarchically. Logger can have child logger, in example, an object logger will have a child the object attribute logger; this means if we set an appender for the object logger, the child will be affected the same. This architecture is useful when we introduce the concept of RootLogger, a logger parent of every other instance of logger. This means we can set a default appender to all the loggers just by assigning it to the RootLogger. With Log4j, we can specify the appender configuration either with a configuration file either with code. In our case, this is made by code as seen in Listing 13. This configuration of appenders is made in the Endpoint library of cloud.iO and so is static when using the library. A developer can then simply use Log4j logger and the generated logs will be sent to cloud.iO according to the log level. By doing so, we let developers using their Log4j configuration with a file.

```
1  org.apache.logging.log4j.core.Logger coreLogger =
2      (org.apache.logging.log4j.core.Logger) LogManager.getRootLogger();
3
4  CloudioLogAppender cloudioLogAppender = new CloudioLogAppender("CloudioLogAppender", null);
5
6  coreLogger.addAppender(cloudioLogAppender);
```

Listing 13: Adding cloud.iO custom appender to Root Log4j logger

With the custom appender, by default all Log4j logs are send to the cloud. Listing 14 shows how a developer can generate some log with Log4j using the Java Endpoint library. User can retrieve logger of a specific class with getLogger and even retrieve the Root Logger with getRootLogger as seen in the example. Then the common Log4j api can be used: logger.fatal, logger.error, logger.debug, etc.

```
import ...
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;

public class Application {

  static CloudioEndpoint myEndpoint;

  static Logger logger = LogManager.getLogger(Application.class);
  static Logger rootLogger = LogManager.getRootLogger();

  public static void main(String[] args) {


    myEndpoint = new CloudioEndpoint("bc0f1bf8-bdae-11e9-9cb5-2a2ae2dbcce4");

    // create Endpoint structure
    ...

    //all the following logs will be pushed to the cloud according to Endpoint log level
    logger.trace("This is a trace from Application logger");
    rootLogger.trace("This is a trace from Root logger");

    logger.error("Error log");
    logger.debug("Debug log");
  }
}
```

Listing 14: Example of log in with the cloud.iO Java library and Log4j

In the case of log level modification by the User, the Endpoint is subscribed to the *@logsLevel* message in the same way as seen in Subsection 3.3 for the *@exec* message. The main difference in the treatment of the message is that the retrieved log level of the Endpoint needs to be saved on the Endpoint in the case of restart of the Endpoint code. To this end, we are saving the log level in Endpoint non-volatile memory using an embedded database with the MapDB[32] library under the free Apache License 2.0. MapDB provides a simple solution to save Maps, Sets, Lists, Queues, and Bitmaps in an embedded database. By using this library, we can save an retrieve the Endpoint Log Level in any case. An example of saving and retrieving this log level is shown in Listing 15

```
//save the log level
DB dbPersistenceData = DBMaker.fileDB(PERSISTENCE_FILE).make();
ConcurrentMap map = dbPersistenceData.hashMap(PERSISTENCE_MAP_NAME).createOrOpen();
map.put(PERSISTENCE_LOG_LEVEL, logParameter.getLevel());
dbPersistenceData.close();

//retreive the log level
DB dbPersistenceData = DBMaker.fileDB(PERSISTENCE_FILE).make();
ConcurrentMap map = dbPersistenceData.hashMap(PERSISTENCE_MAP_NAME).createOrOpen();
dbPersistenceData.hashMap(PERSISTENCE_MAP_MQTT_MESSAGES).createOrOpen();
String logLevel = (String)map.getOrDefault(PERSISTENCE_LOG_LEVEL,"");
// process the logLevel
dbPersistenceData.close();
```

Listing 15: MapDB use for saving and retreiving the log level of the Endpoint

[32] http://www.mapdb.org

3.5 Transaction

The original vision of the *@update* messages of cloud.iO was to update not only the Attribute but any part of the cloud.iO object model according to the topic. For example we could update a whole cloud.iO Object by sending its JSON serialization to the *@update/EndpointUUID*/nodes/*nodeName*/objects/*objectName*. The origin of this idea was to reduce the number of messages to send for an embedded system with low network bandwidth. This feature was never implemented; only the Attribute can be updated with *@update* message.

The idea of a special message containing more than just an Attribute to update is taken back and redesigned in the concept of Transaction. Transaction differs from the *@update* model from the idea that a transaction is a Map of Attributes to be updated with their topics as keys. This gives us flexibility in updating only the Attribute needed and not updating a whole Object or full Node just to update some Attribute as the *@update* first vision. The Transaction introduces a new topic, as shown in Table 13. The *@transaction* is the last keyword added to cloud.iO. Appendix A4.0 take back all those topic and their definition. All the topics described through this document for cloud.iO v0.2 and their corresponding Data following the heater system example are available in Appendix A5.0.

Topic	Data
@transaction/*EndpointUUID*	cloud.iO Transaction message formatted in JSON as seen in Listing 16.

Table 13: Topic and the corresponding data to send to cloud.iO v0.2 for the logging

```
{
  "attributes": {
    "endpointUUID/myHeater/temperatures/temperature": {
      "constraint": "Measure",
      "timestamp": 1577802897520,
      "type": "Number",
      "value": 25.1
    },
    "endpointUUID/mySecondHeater/temperatures/temperature": {
      "constraint": "Measure",
      "timestamp": 1577802897750,
      "type": "Number",
      "value": 23.1
    },
    "endpointUUID/exampleNode/exampleObject/exampleAttribute": {
      "constraint": "Status",
      "timestamp": 1577802898290,
      "type": "String",
      "value": "Example"
    }
  }
}
```

Listing 16: JSON serialization of Transaction to send on @transaction topic

Listing 16 show a Transaction with three Attributes to update. They are all serialized in the Attributes field map and stored under their topics. In the Java library, three methods are added to the Endpoint class to use Transaction:

- **beginTransaction**(): Start the transaction, every update of Attribute after the use of this method will be saved in the Transaction map
- **commitTransaction**(): Stop the transaction, send all the updated Attribute from the Transaction map in a *@transaction* message
- **rollbackTransaction**(): Clear the Transaction map during a Transaction

It is important to note that an Attribute will appear only once in a Transaction, only its last updated value will be sent. Listing 17 shows an example of a Transaction with the Java library. We can see that the Attributes, during the Transaction, are using the usual setValue method used to send *@update* message. In the case of an ongoing Transaction, no MQTT messages are sent until the end of it.

```java
public class Application {

  static CloudioEndpoint myEndpoint;

  public static void main(String[] args) {
    try {
      myEndpoint = new CloudioEndpoint("bc0f1bf8-bdae-11e9-9cb5-2a2ae2dbcce4");

      //demoNode contains demoObject with demoStatus and demoMeasure
      myEndpoint.addNode("demoNode", DemoNode.class);

      DemoNode demoNode = myEndpoint.getNode("demoNode");

      myEndpoint.beginTransaction();

      //this setValue is forgotten by the transaction because of the rollback
      demoNode.demoObject.demoStatus.setValue(20);
      myEndpoint.rollbackTransaction();

      demoNode.demoObject.demoStatus.setValue(20.0);

      demoNode.demoObject.demoMeasure.setValue(20.0);
      demoNode.demoObject.demoMeasure.setValue(21.0);
      //only the second demoMeasure.setValue will be in the transaction

      //demoStatus = 20.0 demMeasure = 21.0
      myEndpoint.commitTransaction();
    }
    catch (Exception e){
      e.printStackTrace();
    }
  }
}
```

Listing 17: Example of use of Transaction with the Java Endpoint Library

On the cloud.iO core side, when a transaction message is received, it is processed, and every Attribute is re-sent by the cloud.iO core under *@update* message. As usual, a specific micro-service dedicated to Transaction is created, and it contains a RabbitListener. A simplified version of this RabbitListener is show in Listing 18. We can see that the listener does subscribe to *@transaction.** messages and will process the transaction message at Line 14 and send every Attribute in *@update* messages.

```kotlin
@RabbitListener(bindings = [QueueBinding(value = Queue(),
        exchange = Exchange(value = "amq.topic", type = ExchangeTypes.TOPIC,
    ↪ ignoreDeclarationExceptions = "true"),
        key = ["@transaction.*"])])
fun handleTransactionMessage(message: Message) {
  //retreive EndpointId and data from mqtt message
  val endpointId = message.messageProperties.receivedRoutingKey.split(".")[1]
  val data = message.body

  //deserialize the data to Transaction
  val transaction = Transaction()
  JsonSerializationFormat.deserializeTransaction(transaction, data)

  //process every Attribute from the transaction
  transaction.attributes.forEach { (topic, attribute) ->
      rabbitTemplate.convertAndSend("amq.topic",
              "@update." + topic.replace("/", "."), JsonSerializationFormat.serializeAttribute
    ↪ (attribute))
  }
}
```

Listing 18: Simplified version of Kotlin RabbitLister of the EndpointTransactionService

3.6 Certificate Management

In cloud.iO, Endpoint communication is using MQTT with TLS/SSL and authentication using an x509 certificate. For a valid connection, three components are needed: The certification authority (ca) certificate, the client (here the Endpoint) certificate, and the private key. As mentioned earlier, the management of certificates was done manually in cloud.iO v0.1. Indeed, a makefile using OpenSSL was used to generate certificates (containing the public key), and private key for the Endpoints using the ca certificate and ca private key. Every certificate and keys are firstly generated using Privacy-Enhanced Mail (PEM), a Base64 encoded file. For an MQTTS connection PEM file can be used, although, in the Java Endpoint library, Keystore is used. A Keystore can be defined as an encrypted archive file containing cryptographic objects such as certificate or privates keys, in general, passwords protect Keystores. For the Java Endpoint library, the ca certificate is stored in Java in a Keystore (.jks file), and the client certificate and key are inside PKCS12 Keystore (.p12 file). The path of those files and their passwords are stored in the endpointUUID.properties of the Java application using the cloud.iO library. An example of .properties files is shown in Appendix A6.0

For cloud.iO v0.2, we wanted to generate all those certificates and keys automatically. It is possible to now create a certificate for existing Endpoint by using the RESTful API. The RESTful API concepts are described in Subsection 3.7, only details about the generation of certificate and the type of files used is detailed. Three different solutions to create certificates are provided:

- Generating certificate and private keys for an Endpoint in PEM format, letting the Endpoint user generating Keystore if needed. The API returns a JSON file with the certificate and key in PEM format.
- Generating certificate in PEM from a given public key in the case of a user not trusting cloud.iO handling his private key. The API returns a JSON file with the certificate in PEM format.
- Generating a Zip file containing the ca certificate in JKS file, a P12 file containing the Endpoint certificate and private key, and the endpointUUID.properties files filled with all the parameters to start a Java project with the cloud.iO libraries and containing the JKS password and the randomly generated P12 password. The API returns the Zip file, detailed in subsection 3.7.2 about RESTful API and the Certificate controller.

Those three ways to generate certificates are managed by a Spring Service using the cryptography library BouncyCastle[33]. The RESTful API will access this Service to create certificates. cloud.iO also provides the possibility to retrieve the ca certificate serialized in JSON following the PEM format using the RESTful API.

When deploying cloud.iO, you will need a valid ca certificate and ca private key for cloud.iO core. They will be used for the generation of Endpoint certificates. A broker server certificate and server private key with their own ca certificate (can be the ca certificate already cited if the server certificate and key are signed by this ca) is also needed for RabbitMQ. For the cloud.iO core, the certificate and key need to be added in PEM format to the application.properties files, an example is shown in Appendix A3.0. For RabbitMQ, as said earlier, a custom docker image has been created, and the certificates and keys can be passed as parameters during the launch of the docker. Description of the docker images with examples of certificates and key as parameters is shown in Appendix A7.0.

3.6.1 Security Recommendation

The TLS protocol includes the following cryptography operation: asymmetrical cryptography with RSA after authentication, creation of shared secret between server and client with symmetrical cryptography using AES, and hashing function are used for integrity control and authentication of data. All those operations can have different configurations (number of bit for key, different hash function). For cloud.iO, the TLS configuration is done in RabbitMQ; the configuration is the following: TLS 1.2 RSA (key of 2048 bits) with AES 256 and either SHA256 or GCM SHA384 hash. The version 1.3 of TLS has been released since august 2018 but is still not supported for RabbitMQ; this is the reason why cloud.iO is still using TLS 1.2.

The actual security configuration of cloud.iO follows state of the art in term of cryptography. But it is important to note that security in informatics is evolving. It is imperative to follow the latest recommendation in terms of security. For example, cloud.iO is using RSA keys of 2048 bits. If we follow the recommendation of the American National Institute of Standards and Technology (NIST) key of 2048 bits can be used until 2030, for then passing to keys of 3072 bits. The French Agence nationale de la sécurité des systèmes d'information (ANSSI) has the same recommendation for key length. More strict recommendation exists, we can cite the European Network of

[33] https://www.bouncycastle.org

Excellence in Cryptology (ECRYPT) already recommending 3072 bits keys since 2018 or the American National Security Agency recommending, in general, to use 3072 bits key. All those recommendations are made available by BlueKrypt[34].

One of the critical points of using public key infrastructure is the storage of the private key. In cloud.iO, we are already using a Keystore following the PKCS12 standard. But this is only reporting the storage problem to the Keystore password. We can distinguish between two use cases of key storage: the storage of the key on the Endpoint itself and the storage of multiple Endpoint keys by a User developer or owner of Endpoint. For the first case, we consider the certificate and key of an Endpoint compromise as soon as someone has physical access on it (via SSH, for example). In that case, it is possible to block the Endpoint with cloud.iO API. To enhance the security of the key storage, in the case of physical access on an Endpoint, we recommend the use of Hardware Security Module to store the private key (we can cite, for example, component made by Infineon Technologies) or the use of encrypted file system like LUKS.

For the second use case presented before where someone wants to store multiple Endpoint keys, the same recommendation of Hardware Security Module or encrypted file system is applied. Another solution could be the use of dedicated software like Vault[35] whose can be used to manage secrets, like private keys, and protect sensitive data. The deployment of the private keys on the Endpoint could even be done using the Vault HTTP API, giving secure remote access to the stored secrets.

[34] https://www.keylength.com
[35] https://www.vaultproject.io

3.7 RESTful API

The RESTful API was the biggest missing part of cloud.iO v0.1. Adding RESTful API to cloud.iO can add plenty of management tools for the Users and also for a system administrator. Before starting the implementation, an analysis of the needs in cloud.iO is really important. To this end, a simple mockup of a webpage to monitor the whole cloud.iO system has been designed. Figure 14 show those mockups. We can see two different user interfaces: on the right an interface for a simple User and on the left for a cloud.iO administrator (Two user roles defined in cloud.iO v0.2 Role subsection. 3.1.2)).

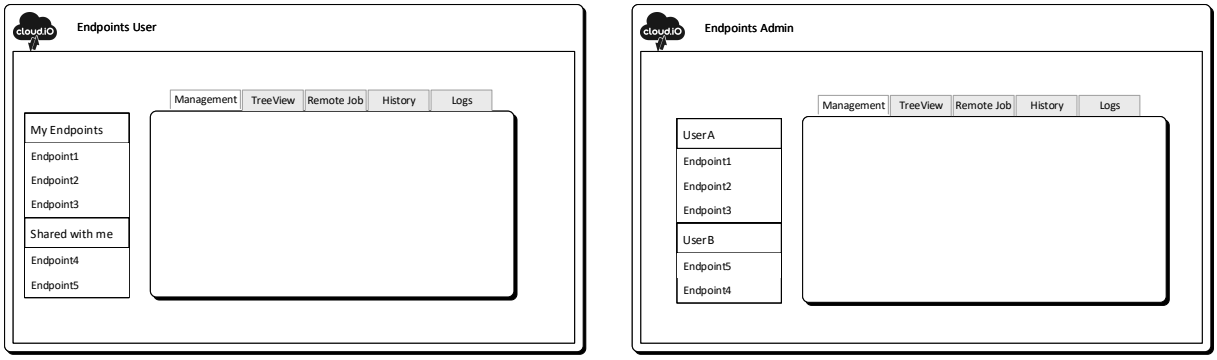


Figure 14: Simple Mockup of WebPage User Interface for cloud.iO Management

These mockups show different points:

- The normal User manages its own Endpoint an Endpoint who are shared with him
- The Administrator can manage any Endpoint and also Users (and so User Group)
- The User and Administrator can execute the same action on Endpoint (tabs)
- Important function show in the tab are Management (Endpoint or User), TreeView (Endpoint view as a tree, similar to digital twin), Remote Job, History and Logs

Concerning the Endpoint Management, we can add that it also induces the support of certificate creation. Another note is that the User and User Group Management brings up the need for Access Control Management.

With all those observations, we can define the different category for the RESTful API:

- User Management
- User Access Control
- User Group
- User Group Access Control
- Endpoint Management
- Certificate
- History
- Logs
- Remote Job Execution

With all those categories, we can define the desired route of the API before implementing them. An API proposal has been made and can be found in Appendix A8.0. The final and implemented version of the API can be found in Appendix A9.0. The whole RESTful API has been documented with Swagger[36], an open-source software framework used to design, build, document, and consume RESTful web services. In our case, it has only been used for the documentation but will be used to generate API SDK and online documentation. The work in progress swagger project is open and can be accessed online[37].

The implementation of the API is made with Spring Web MVC, a web framework included in the Spring Framework. We can easily create a RESTful API with it by creating RestController by using the `@RestController` decorator. A RestController acts as a Controller of the MVC pattern but is specific to Rest Request. In the cloud.iO core, an abstraction layer has been added for the API. Indeed, the logic of the API is used in the RESTful API, but we could want to re-use it with another technology. This abstraction layer is located in the apiutils package of cloud.iO core, and their implementation for the RESTful API is in the restapi/controllers package.

Listing 19 show an example of the RestController dedicated to the UserManagement and the implementation of the /getUser route. In a RestController, we can create methods that represent routes of a RestAPI by using the `@RequestMapping` decorator to define the route whose the method should react to as see in line 5. The `@RequestMapping` can also be used in the class to define a parent route, as seen on line 2. On line 6, we can see the class we are expecting a body with the `@RequestBody` decorator and the return type of the Rest Route, here User. The Request body comes in JSON and will be deserialized into the userRequest value, and the returned value from the getUser Method will be serialized to the Response Body. Both classes are serialized and deserialized automatically by Spring MVC. On line 14, we can see the call to the abstraction layer where the logic of the API is made. Before this API utils call, the method contains logic specific on the RESTful API, here we test if the User is making the request has HTTP admin authority.

```
1  @RestController
2  @RequestMapping("/api/v1")
3  class UserManagementController(var userRepository: UserRepository) {
4
5    @RequestMapping("/getUser", method = [RequestMethod.GET])
6    fun getUser(@RequestBody userRequest: UserRequest): User {
7      //get User Name (from the request)
8      val userName = SecurityContextHolder.getContext().authentication.name
9      //test if user has http admin authority
10     if (!userRepository.findById(userName).get().authorities.contains(Authority.HTTP_ADMIN))
11       throw CloudioHttpExceptions.ForbiddenException(CLOUDIO_AMIN_RIGHT_ERROR_MESSAGE)
12     else {
13       //execute request in api util
14       val user = UserManagementUtil.getUser(userRepository, userRequest)
15       if (user == null)
16         throw CloudioHttpExceptions.BadRequestException("User doesn't exist")
17       else
18         return user
19     }
20   }
21
22   //Other routes concerning the UserManagemenent
23   @RequestMapping()
24   ...
25 }
```

Listing 19: Kotlin Example of RestController with the /api/v1/getUser route in UserManagementController

The whole RESTful API is built on the same model as Listing 19. Although there are some API routes with specific behavior that are described in the continuation of this subsection: the use of path variable, sending a file with the API, implementation of longpoll, and server-sent event answer. Finally, in this subsection, we will discuss how the RESTful API will enhance the data visualization and how the authentication using MongoDB User data and so the user access controls implemented on the RESTful API with a final note about HTTPS.

[36] https://swagger.io
[37] https://app.swaggerhub.com/apis/luc_blender/cloud.iO_2.0/1.0.0

3.7.1 Path Variable; Endpoint Management Controller

Most of the API have parameters in the Request Body. Although, when it comes to retrievement of Endpoints, Nodes, Objects, Attributes, and their parameters, we need to define unique API routes (URI) for every resource according to the Web Of Thing Candidate Recommendation (see Subsection 3.8). We achieve this by using the `@PathVariable` decorator in a RequestMapping method. Listing 20 show the two different ways to get parameter from a request: Using `@PathVariable` Line 9 or by using `@RequestBody` Line 2. The Path Variable is retrieved from the URL by Spring MVC and can be used as any function parameter. This concept has always been applied to retrivement of User, User Group, User Access Right and User Group Access Right.

To get an Attribute we can either send a Post Request to `https://cloud.io:8081/api/v1/getAttribute` a Request Body following the EndpointRequest Model containing the Endpoint UUID or send a Get Request to `https://cloud.io:8081/api/v1/getAttribute/endpointUuid.nodeName.objectName.attributeName` without any Request Body.

```
1  @RequestMapping("/getAttribute", method = [RequestMethod.POST])
2  fun getAttribute(@RequestBody attributeRequest: AttributeRequest): Attribute {
3    val userName = SecurityContextHolder.getContext().authentication.name
4
5    return getAttribute(userName, attributeRequest)
6  }
7
8  @RequestMapping("/getAttribute/{attributeTopic}", method = [RequestMethod.GET])
9  fun getAttribute(@PathVariable attributeTopic: String): Attribute {
10   val userName = SecurityContextHolder.getContext().authentication.name
11
12   return getAttribute(userName, AttributeRequest(attributeTopic.replace(".", "/")))
13 }
14
15 fun getAttribute(userName: String, attributeRequest: AttributeRequest): Attribute {
16   ...
17 }
```

Listing 20: Example of two differents way of handeling parameters for /api/v1/getAttribute route in EndpointManagementController

This concept is applied on the following routes: getEndpoint, getNode, getWotNode, getObject, getAttribute, setAttribute, getEndpointFriendlyName, notifyAttributeChange, getLogsLevel, getUser, getUserGroup, getUserAccessRight, getUserGroupAccessRight.

Both getAttribute methods from Listing 20 will give us the same output, but we can see that one is a POST request, and the other one is a GET. When using HTTP, it is not common to use a Request body in GET Request based on the HTTP RFC specification. If we look at the RFC2626 specification of HTTP/1.1, here is what we can find in the Message Body Section 4.3: "*A message-body MUST NOT be included in a request if the specification of the request method (section 5.1.1) does not allow sending an entity-body in requests. [...] then the message-body SHOULD be ignored when handling the request.*"[12] And in GET Section 9.3: "*The GET method means retrieve whatever information (in the form of an entity) is identified by the Request-URI.*"[12]

The request body is not part of the identification of the resource in a GET request. This means a request body can exist but should not influence the response of a GET request. However, RFC2616 is now obsolete and replaced by RFC 7230 since 2014. In RFC 7230, both previously cited part of RFC 2616 has been deleted, removing specifications about request body on GET requests. (Message Body Section 3.3[13])

Even if there is no more specification about the request body in a GET method, it is still common usage to not use it and to employ a POST method instead.

3.7.2 Send a file; Certificate Controller

As seen earlier, cloud.iO can generate the certificate and the configuration files needed for the Java Endpoint library: A PKCS12 Keystore for the Endpoint certificate and private key, the broker certificate in JKS format land the .properties files. With the RESTful API, we can retrieve the Endpoint key and certificate in text form as a String in PEM format JSON field. Serialization in JSON can not be used for sending a file.

To return a file, we can use the ResponseEntity class whose is a whole HTTP response class in SpringMVC. This class has a generic type; it is used to specify any type as a response body. To send a file, we will specify the ResponseEntity type as UrlResource to send a resource from a URL; here a file. Listing 21 shows the use of this ResponseEntity class to send the Zip file containing the certificates and configuration. On line 7, we can see the call of the CertificateUtil that will create the Zip file and return the absolute path of this file. This path is then formated and URL in the UrlRessource object. Finally the ResponseEntity is created on line 11. We can see that the answer will contain a Zip file by the application/zip content type and that the UrlResource is added as the request body.

```
1  @RequestMapping("/createCertificateAndKeyZip", method = [RequestMethod.GET])
2  fun createCertificateAndKeyZip(@RequestBody certificateAndKeyZipRequest:
       ↪ CertificateAndKeyZipRequest): ResponseEntity<UrlResource> {
3
4    //check the user access right here
5    ...
6
7    val pathToReturn = CertificateUtil.createCertificateAndKeyZip(rabbitTemplate,
       ↪ certificateAndKeyZipRequest)!!.replace("\"", "")
8    val resource = UrlResource("file:$pathToReturn")
9    val contentType = "application/zip"
10
11   return ResponseEntity.ok()
12       .contentType(MediaType.parseMediaType(contentType))
13       .header(HttpHeaders.CONTENT_DISPOSITION, "attachment; filename=\"" + resource.filename +
       ↪  "\"")
14       .header("EndpointUuid", certificateAndKeyZipRequest.endpointUuid)
15       .body(resource)
16 }
```

Listing 21: Example of file returning for /api/v1/createCertificateAndKeyZip route in CertificateController

3.7.3 Longpoll; Endpoint Management Controller

With the RESTful API, we can easily access and modify cloud.iO resources. The most important resource is the Attributes. Without the RESTful API, we can access the attribute with MQTT by publishing or subscribing to it. The publishing is easily done with a RESTful API with a POST on the setAttribute route. However, subscribing is a bit more complicated. We can first retrieve the last attribute value with the getAttribute route, but this route does not notify the update of the Attribute like an MQTT subscribe will do.

To overcome this problem, another route name notifyAttributeChange is implemented using the concept of HTTP longpoll. An HTTP longpoll is a server request whose is held open by the server until the information is ready. The client opens a longpoll request, processes the message when it arrives, and re-open a new longpoll request as soon as possible not to miss any information.

In the case of the notifyAttriuteChange, the User will open the longpoll request; the server will then create a queue in RabbitMQ with the Attribute topic and wait until an MQTT message come to transmit it to the User with the HTTP Response and finally closing the request and deleting the MQTT queue.

The implementation of the longpoll with Spring MVC is done by returning and DeferredResult. The DeferredResult let perform asynchronous request processing in a separate thread resulting in a non-blocking RequestMapping method. DeferredResult also has a generic type used to specify any type as a response body. Listing 22 show how this DeferredResult is implemented for the notifyAttributeChange route.

```
@RequestMapping("/notifyAttributeChange", method = [RequestMethod.GET])
fun notifyAttributeChange(@RequestBody attributeRequestLongpoll: AttributeRequestLongpoll):
    ↪ DeferredResult<Attribute> {

  //check the user access right here
  ...
  val attribute = EndpointManagementUtil.getAttribute(endpointEntityRepository,
    ↪ AttributeRequest(attributeRequestLongpoll.attributeTopic))

  val result = DeferredResult<Attribute>(attributeRequestLongpoll.timeout,
      CloudioHttpExceptions.TimeoutException("No attribute change after " +
    ↪ attributeRequestLongpoll.timeout + "ms on topic: " + attributeRequestLongpoll.
    ↪ attributeTopic))

  if (attribute.constraint == AttributeConstraint.Measure || attribute.constraint ==
    ↪ AttributeConstraint.Status) {
    CompletableFuture.runAsync {
      object :
      AttributeChangeNotifier(connectionFactory, "@update." + attributeRequestLongpoll.
    ↪ attributeTopic.replace("/", ".")) {
        override fun notifyAttributeChange(attribute: Attribute) {
          result.setResult(attribute)
        }
      }
    }
  } else if (attribute.constraint == AttributeConstraint.Parameter || attribute.constraint ==
    ↪ AttributeConstraint.SetPoint) {
    CompletableFuture.runAsync {
      object :
      AttributeChangeNotifier(connectionFactory, "@set." + attributeRequestLongpoll.attributeTopic
    ↪ .replace("/", ".")) {
        override fun notifyAttributeChange(attribute: Attribute) {
          result.setResult(attribute)
        }
      }
    }
  } else
    throw CloudioHttpExceptions.BadRequestException("Attribute with constraint ${attribute.
    ↪ constraint} can't send notification")

  return result
}
```

Listing 22: Example of longpoll for /api/v1/notifyAttributeChange route in EndpointManagementController

On line 8, we can see the instantiation of the DeferredResult; it takes as parameter the timeout after where it forces to close the connection and the value to return in the case of timeout. For the notifyAttribute change route, we need to differentiate between the topic reacting to *@set* or *@update* topic, we can see the start of handling of *@update* related attribute on line 11 and the start of handling of *@set* related attribute on line 20. After the tests on the attribute constraint, a new Thread is launched with a CompletableFuture.runAsync. This thread will continue even after the execution of the notifyAttributeChange method. Then inside this thread a AttributeChange notifier is instantiate on line 14. This class is responsible for the subscription on the MQTT topic, and to return the first received MQTT message to be set as a result of the DeferredResult object. The implementation of this class is shown in Listing 23.

```
1  abstract class AttributeChangeNotifier(connectionFactory: ConnectionFactory, topic: String) {
2
3    companion object {
4      private val log = LogFactory.getLog(AttributeChangeNotifier::class.java)
5    }
6
7    init {
8      //create a new queue with parameter topic and bind it to default amq.topic exchange
9      val connection = connectionFactory.newConnection()
10     val channel = connection.createChannel()
11     val queueName = channel.queueDeclare().getQueue()
12     channel.queueBind(queueName, "amq.topic", topic)
13
14     //create a callback or the queue
15     val deliverCallback = DeliverCallback { _, delivery ->
16       val message = String(delivery.body, Charset.defaultCharset())
17       val messageFormat = JsonSerializationFormat.detect(message.toByteArray())
18       if (messageFormat) {
19         val attribute = Attribute()
20         JsonSerializationFormat.deserializeAttribute(attribute, message.toByteArray())
21         if (attribute.timestamp != -1.0 && attribute.value != null) {
22           notifyAttributeChange(attribute)
23           channel.queueDelete(queueName)
24         }
25       } else {
26         log.error("Unrecognized message format in $topic message")
27       }
28     }
29     channel.basicConsume(queueName, true, deliverCallback, CancelCallback {})
30   }
31
32   open fun notifyAttributeChange(attribute: Attribute) {
33     log.error("function not overridden")
34   }
35 }
```

Listing 23: AttributeChangeNotifier class used to create dynamically queue on RabbitMQ on specific topic

The presented MQTT listener is dedicated to cloud.iO Attribute. It can be instantiated dynamically with a custom topic, as seen earlier with the notifyAttributeChange route. Therefore we cannot use the RabbitListener decorator. In this case, the native RabbitMQ API is created to create dynamically a queue with the dedicated topic, as seen on line 12. We then create a DeliverCallback in charge of handling the MQTT message arriving in our newly created queue. We then process the message, deserialize it in an Attribute object to finally send it to the notifyAttributeChange method. This method is open and so it can be overridden as seen on line 15 of Listing 22.

3.7.4 Server-sent events; Jobs Controller

The Remote Job Execution in MQTT works by publishing the Job request on a topic and subscribing to a specific topic to get all the outputs of Jobs from an Endpoint. In this case, the Longpoll seen previously cannot be applied since we might want to be aware of more than one data received on the *@execOutput* topic. With an HTTP connection, we can reproduce similar behavior of this topic subscription with server-sent events[38], a standard part of HTML5 by the W3C. Server-sent events consist of an HTTP request held open by the server, which is going to stream data to the client. The format of the event consists of keyword fields followed by ":" and the field content; two events are separated by two carriage-return. The events fields are `data`, `id`, `event`, and `retry`; none of those fields are mandatory for an event. In our case, we only use the datal field to send the Jobs Output Line data and the idl field to send the correlation ID. With the concept of Server-sent events, we can reproduce the behaviour of the MQTT Remote Job execution; the executeJob API route let the user send a Job Request (the Job request will be transferred to the Endpoint via MQTT) and receive the output of the Jobs from the *@execOutput* MQTT as a stream. An example of received stream can be seen in Listing 24.

```
data:List of command
id:1a45a6ca

data:-cmd://listJobs
id:1a45a6ca

data:-cmd://updateJobs
id:1a45a6ca
```

Listing 24: Example of Data receive through the stream of a Server-sent events for the executeJob route

The Server-sent events mechanism is available in Spring MVC with the SseEmitter class, and its implementation for the executeJob route is shown in Listing 25. We start by creating a SseEmitter on line 6, the event will be sent on the open stream through this object. On line 11, we can see the instantiation of an ExecOutputNotifier, this class creates an MQTT listener for @execOutput message, its concept is similar to the AttributeChangeNotifier seen earlier with an open method acting as listener for the MQTT message. Then, every time a message arrives on the @execOutput topic, it is sent to through the SseEmitter as seen on line 15

```
@RequestMapping("/executeJob", method = [RequestMethod.POST])
fun executeJob(@RequestBody jobExecuteRequest: JobExecuteRequest): SseEmitter {

  //check the user access right here
  ...
  val emitter = SseEmitter()

  executor.execute {
    try {
      //create a listener for the correct execOutput topic
      val execOutputNotifier = object : ExecOutputNotifier(connectionFactory, "@execOutput." +
    ↪  jobExecuteRequest.endpointUuid) {
        override fun notifyExecOutput(jobsLineOutput: JobsLineOutput) {
          if (jobsLineOutput.correlationID == jobExecuteRequest.correlationID)
          //send the output as a Sse event
            emitter.send(SseEmitter.event().id(jobsLineOutput.correlationID).data(
    ↪ jobsLineOutput.data))
        }
      }
      JobsUtil.executeJob(rabbitTemplate, jobExecuteRequest)
      Thread.sleep(jobExecuteRequest.timeout)
      emitter.complete()
      execOutputNotifier.deleteQueue()

    } catch (e: Exception) {
      emitter.completeWithError(e)
    }
  }
  return emitter
}
```

Listing 25: Example of Sse for /api/v1/executeJob route in JobsController

[38] https://www.w3.org/TR/eventsource/

3.7.5 Data Visualisation; History Controller

In cloud.iO v0.1, data visualization was done through the InfluxDB plugin for Grafana. The Grafana server needed to be part of the cloud.iO core infrastructure to have direct access to InfluxDB and be configured by a cloud.iO administrator. Every cloud.iO User that wanted to visualize their data needed to have also a Grafana login with similar permissions to their cloud.iO access right.

With cloud.iO v0.2, we want to get rid of this redundancy of account for the data visualization by developing a cloud.iO plugin for Grafana using the RESTful API, and so the cloud.iO User access control. Indeed Grafana provides the possibility to create plugins in JavaScript to add custom datasource[39].

The requirements of this route were: providing data from cloud.iO to Grafana following the InfluxDB plugin model, and being easy to integrate into a custom Grafana plugin. The actual cloud.iO plugin is in early development at the Institute of Systems Engineering at the HEVs. Its development will not be covered in this document, only the REST API route created for its History access: getAttributeHistoryExpert.

Custom data source Grafana plugins are designed around the query model shown on Listing 26. This query object is passed to datasource.query function in charge of retrieving the data. In our case, this function will call the cloud.iO API according to the query object. The first field of the object is the `range` of time from where the data need to be retrieved, then we have the `interval` of time between two data. The next field is `targets`; it defines the resource to retrieve. In our case, the `targets` fild will contain the attribute topic. The next field is the `format`, defining the return format for the time series, and finally, we have the `maxDataPoints` limiting the amount of point to return. The three essential fields to have in our Rest Request are the time range, the interval, and maxDataPoints.

```
{
  "range": { "from": "2020-01-26T02:34:12.145Z", "to": "2020-01-26T02:37:12.145Z" },
  "interval": "5s",
  "targets": [
    { "refId": "B", "target": "endpointUuid.nodeName.objectName.attributeNameB" },
    { "refId": "A", "target": "endpointUuid.nodeName.objectName.attributeNameA" }
  ],
  "format": "json",
  "maxDataPoints": 4096
}
```

Listing 26: Grafana query Example for custom Data Source Plugin

Figure 15 show the InfluxDB plugin user interface used with cloud.iO v0.1. We can see how we build a custom InfluxQL (SQL like query for InfluxDB) with it. The most important part is the data we want to retrieve following the FROM keyword in green. The InfluxDB plugin directly lists all available field values that we can retrieve; this behavior can be achieved for the cloud.iO plugin by using the getAccessibleAttributes route from the REST API listing all the Attributes topics accessible by a user as seen on Listing 27. Two other features taken back from the InfluxDB plugin are: aggregation and fill.

When we group data by time, there is a different way to aggregate it. On Figure 15 the mean aggregation (in blue) is used, but it can be one of those value:

- **COUNT**: return the number of retrieved field values
- **DISTINCT**: return the number of unique retrieved field values
- **INTEGRAL**: return the integral of retrieved field values
- **MEAN**: return the arithmetic mean of retrieved field values
- **MEDIAN**: return the median of retrieved field values
- **MODE**: return the most frequent value of retrieved field values
- **SUM**: return the sum of retrieved field values

[39] https://grafana.com/docs/grafana/latest/plugins/developing/datasources/

Also, when grouping data by time, some interval of time might have empty data. InfluxDB let us fill those empty data as seen in Figure 15 where None is used (in orange):

- **NULL**: fill with null the interval
- **NONE**: do not fill the interval
- **ZERO**: fill with '0'
- **PREVIOUS**: fill gaps with previous data
- **LINEAR**: fill with linear interpolation

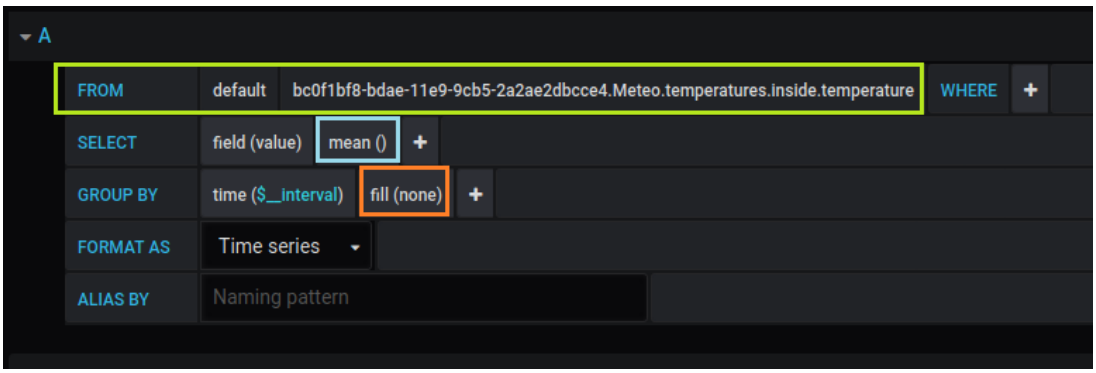


Figure 15: User interface of InfluxDB plugin for Grafana

```
{
  "accessibleAttributes": {
    "bc0f1bf8-bdae-11e9-9cb5-2a2ae2dbcce4/demoNode/demoObject/demoStatic": "READ",
    "bc0f1bf8-bdae-11e9-9cb5-2a2ae2dbcce4/demoNodeTwo/demoObject/demoMeasure": "GRANT"
  }
}
```

Listing 27: JSON answer body for the route of the /api/v1/getAccessibleAttributes API

With all this information, we can build the getAttributeHistoryExpert request body as shown in Listing 28. The first field is the `attributeTopic`; it is the targets we want to retrieve, then we have the aggregation| and `fill` the field from InfluxQL. Then we have the two dates,`dateStart` and `dateStop`, representing the time range. Finally, we have the `interval` and `maxDataPoints` fields from the Grafana query.

```
{
  "attributeTopic": "110e8400-e29b-11d4-a716-446655440000/nodeName/objectName/attiruteName",
  "aggregation": "COUNT",
  "fill": "NULL",
  "dateStart": "2019-03-10 00:00:00",
  "dateStop": "2019-03-10 01:17:00",
  "interval": "10m",
  "maxDataPoints": 2048
}
```

Listing 28: JSON request body for the /api/v1/getAttributeHistoryExpert route of the API

3.7.6 Authentication and Https

The RESTful API is a tool for cloud.iO Users. We need to authenticate them based on the User database in MongoDB. In Spring MVC, we can configure the security of our API separately of its implementation. This means modifications of the security and authentication does not affect the API.

First of all, we need to configure Spring MVC to retrieve the Users from MongoDB; this is done by creating a class implementing the UserDetailsService interface. UserDetailsService is used to retrieve user data; it has a method named loadUserByUsername. This method needs to be overridden to adapt the way we retrieve User information. Listing 29 show how the UserDetailsService in implemented in cloud.iO to retrieve MongoDB User.

```
1  @Component
2  class MongoCustomUserDetailsService(var userRepository: UserRepository) : UserDetailsService {
3
4    companion object {
5      private val log = LogFactory.getLog(MongoCustomUserDetailsService::class.java)
6    }
7
8    override fun loadUserByUsername(username: String?): UserDetails {
9
10     val user = userRepository.findByIdOrNull(username)
11     if (user == null) {
12       log.error("User not found")
13       throw UsernameNotFoundException("User not found")
14     } else {
15       val authorities = listOf(SimpleGrantedAuthority("user"))
16
17       if (!user.authorities.contains(Authority.HTTP_ACCESS)) {
18         log.error("$username tried to log in without HTTP_ACCES authority")
19         throw CloudioAuthorityException("User don't have http access as authority")
20       }
21
22       return org.springframework.security.core.userdetails.User(
23           user.userName, user.passwordHash, authorities
24       )
25     }
26   }
27 }
```

Listing 29: User authentication class for the RESTful API with MongoDB

On line 8 we can see the overridden loadUserByUsername function. It searches in the userRepository object, linked to MongoDB User repository, the User data. Line 17 shows that the UserDetailsService already test if the cloud.iO User has the https_access authority to access the RESTful API, if this authority is not present and error is thrown, and the User will receive an HTTP 401 Unauthorized exception. The User data can now be retrieved from MongoDB; it is needed to do the complete Spring MVC security configuration and specify that the User authentication is done with the MongoCustomUserDetailsService class. This configuration is done by implementing the WebSecurityConfigurerAdapter interface as shown in Listing 30 with the SecurityConfiguration class.

```
1  @Configuration
2  @EnableConfigurationProperties
3  class SecurityConfiguration(var customUserDetailsService: MongoCustomUserDetailsService) :
       ↪ WebSecurityConfigurerAdapter() {
4
5    @Throws(Exception::class)
6    override fun configure(http: HttpSecurity) {
7      http.csrf().disable()
8          .authorizeRequests().anyRequest().permitAll()
9          .and().httpBasic()
10         .and().sessionManagement().disable()
11   }
12
13   @Throws(Exception::class)
14   public override fun configure(builder: AuthenticationManagerBuilder?) {
15     builder!!.userDetailsService(customUserDetailsService)
16   }
17
18   @Bean
19   fun passwordEncoder(): PasswordEncoder {
20     return BCryptPasswordEncoder()
21   }
22
23 }
```

Listing 30: Security Configuration class for RESTful API

Two configurations are done with this class through configure methods. The first one on line 6 describe the HTTP security, we can see that we are using basic authentication. The second configure method on line 14 shows the authentication configuration to retrieve User detail through the customUserDetailsService object, an instance of the MongoCustomUserDetailsService seen above. Finally, the SecurityConfiguration instantiates a PasswordEncoder as a Spring Bean on line 19. This Bean can be used by the Security of Spring MVC when a PasswordEncoder is needed; in our case, we are using a BCrypt password encoder.

The actual security configuration of the RESTful API of cloud.iO is basic authentication. It consists of sending the user and password to the server coded in base64. Basic authentication itself is a bad solution for security; the user and password can be easily retrieved. This is why it is MANDATORY to add HTTPS to the RESTful API. The best solution is by adding an HTTPS proxy in front of the RESTful API. But it is also possible to pass the RESTful API directly in https with Spring MVC. This can be done by modifying the application.properties file and adding the line shown in Listing 31. The first parameter is the type of certificate used for the SSL configuration, here PKCS12. The second parameter it the Path to the Keystore containing the SSL certificate with the type specified before. The two last parameters are the alias and the password of the Keystore. We need to create a signed certificate in the Keystore to enable SSL on the Spring MVC server. A quick solution is to generate a self-signed certificate with tools like OpenSSL.

```
1 server.ssl.key-store-type=PKCS12
2 server.ssl.key-store=PathToKeyStoreForHttps
3 server.ssl.key-alias=KeyStoreAlias
4 server.ssl.key-store-password=KeyStorePassword
```

Listing 31: Example of SSL configuration for RESTful API in application.properties file

We already saw that during the authentication to the RESTful API, the User username and password are tested, but also the http_access authority is verified (all in the MongoCustomUserDetailsService show in Listing 29). However, each route of the API can have its specificity in terms of access control. Two examples are shown in Listing 32. First we have the createUser route where we test if the User also has the http_admin authority on line 4. The second example is the executeJob route where we test if the User owns the Endpoint it is trying to access for a remote Job execution on line 18.

```
1  @RequestMapping("/createUser", method = [RequestMethod.POST])
2  fun createUser(@RequestBody user: User) {
3    val userName = SecurityContextHolder.getContext().authentication.name
4    if (!userRepository.findById(userName).get().authorities.contains(Authority.HTTP_ADMIN))
5      throw CloudioHttpExceptions.ForbiddenException(CLOUDIO_AMIN_RIGHT_ERROR_MESSAGE)
6    ...
7  }
8
9  @RequestMapping("/executeJob", method = [RequestMethod.POST])
10 fun executeJob(@RequestBody jobExecuteRequest: JobExecuteRequest): SseEmitter {
11   val userName = SecurityContextHolder.getContext().authentication.name
12
13   val permissionMap = PermissionUtils
14           .permissionFromUserAndGroup(userName, userRepository, userGroupRepository)
15   val genericTopic = jobExecuteRequest.endpointUuid + "/#"
16
17   val endpointGeneralPermission = permissionMap.get(genericTopic)
18   if (endpointGeneralPermission?.permission == Permission.OWN) {
19     ...
20   }
21 }
```

Listing 32: Example of special Authority and Access control verification for the createUser and executeJob api route

The last particularity about the authentication and User access control using the RESTful API that needs to be covered is the access of the Endpoint Repository. In the previous version of cloud.iO access rights to an Endpoint were concerning the whole Endpoint where now we can decide what part of Endpoint has what type of access right. When we want to retrieve a full Endpoint from the Endpoint Repository through the RESTful API, the whole permission of the User enters into account, and the Endpoint cannot always be returned as-is.

It has been decided that the structure of the Endpoint can be known by anybody having access right linked to and Endpoint but that the Attribute field (constraint, type, timestamp, and value) have to be "censored" if the User does not have the adequate permission. Listing 33 show and Endpoint retrieved with the getEndpoint route following the heater system example. For this example, we assume that the User has a deny permission on the temperature attribute. We can see the result with a "censorship" of the fields of the mentioned Attribute with a human-readable message on the value field.

```
{
  "endpointUuid": "bc0f1bf8-bdae-11e9-9cb5-2a2ae2dbcce4",
  "friendlyName": "test",
  "blocked": false,
  "online": false,
  "logLevel": "ERROR",
  "endpoint": {
    "nodes": {
      "myHeater": {
        "implements": [],
        "objects": {
          "temperatures": {
            "conforms": null,
            "objects": {},
            "attributes": {
               "temperature": {
                "constraint": "Invalid",
                "type": "Invalid",
                "timestamp": -1,
                "value": "You don't have the right to see this attribute value"
              },
              "setPointTemperature": {
                "constraint": "SetPoint",
                "type": "Number",
                "timestamp": 1.576827781E12,
                "value": 22.5
              }
            }
          }
        }
      }
    }
  }
}
```

Listing 33: JSON serialized version of heating system with denied permission for the temperature attribute

3.8 Web Of Things Compliance

cloud.iO is strongly linked to its data model. It gives excellent flexibility to represent any IoT system in the cloud. But this data model has been created for cloud.iO and would not be compatible as is by any other IoT platform, which induces inadequate interoperability. To remedy this problem, we introduced the WoT Thing Description in cloud.iO.

W3C describes in the WoT Architecture how to implement a WoT Servient. Since cloud.iO has not been build in the beginning with the WoT compatibility in mind, converting it to a Servient by implementing all the WoT building blocks would not be the best choice. A more straightforward solution ensuring the compatibility with WoT is to provide WoT TD of the existing Thing of the cloud.iO solution, the Node. W3C describe this solution in their candidate recommendation as Alternative Servient and WoT Implementations[40]. This implementation applied to cloud.iO is shown in Figure 16. With this solution, we are bringing interoperability between cloud.iO and any TD Consumer. To this end, a syntactic translation of a cloud.iO Node to a Thing Description has been implemented.

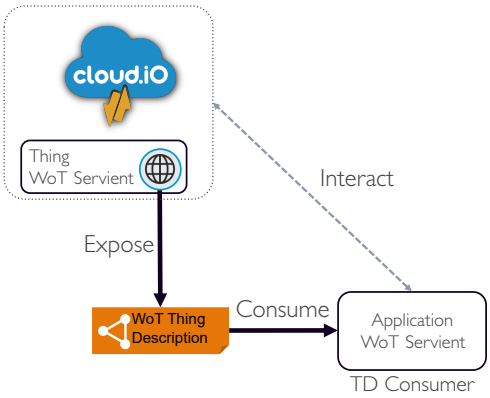


Figure 16: Interaction between cloud.iO and a TD consumer

If we go back to the heating system example, we can study how the data model of a heater Node will be translated as a TD. Listing 34 show the isolated myHeater Node from the smart heating system Endpoint example. As seen earlier, it has two Attributes, the ambient temperature as a Measure and a set point temperature as a Setpoint.

```
{
  "myHeater": {
    "implements": [],
    "objects": {
      "temperatures": {
        "conforms": null,
        "objects": {
        },
        "attributes": {
          "temperature": {
            "constraint": "Measure",
            "type": "Integer",
            "timestamp": 1.57476099056E9,
            "value": 25
          },
          "setPointTemperature": {
            "constraint": "SetPoint",
            "type": "Integer",
            "timestamp": 1.57476098001E9,
            "value": 24
          }
        }
      }
    }
  }
}
```

Listing 34: Digital Twin of cloud.iO Node

[40] https://www.w3.org/TR/wot-architecture/#existing-impl accessed the 06.01.2020

As seen earlier, the TD is composed of metadata and Interaction Affordances. We will first look at the TD generated by cloud.iO without the Interaction Affordance, as shown in Listing 35. As usual, we find the `@context` field specific to JSON-LD with a URI containing the TD Information Model provided by W3C; with cloud.iO, we are not using specific ontology so we can just keep this unique URI for the `@context` field. However, if formal descriptions using Nodes Interfaces and Object class were added to cloud.iO, they could be added through this field of the TD. The second field is the `id`; it describes the Node with and URN composed of the Endpoint UUID and the Node name. Then we have three mandatory fields, the `title`, here the Node name, followed by the `securityDefinitions` and `security`. We can see the description of two SecurityScheme, one for HTTPS and one for MQTTS.

```
{
  "@context": "https://www.w3.org/2019/wot/td/v1",
  "id": "urn:bc0f1bf8-bdae-11e9-9cb5-2a2ae2dbcce4:myHeater",
  "title": "myHeater",
  "securityDefinitions": {
    "https_sc": {
      "scheme": "basic",
      "in": "query"
    },
    "mqtts_sc": {
      "scheme": "cert"
    }
  },
  "security": ["https_sc",
  "mqtts_sc"],
...
}
```

Listing 35: Thing Description of Heater System without Interaction Affordance

For a more in-depth study of the Interaction Affordances, we will only take the *setPointTemperature* Attribute as an example. The TD generated by cloud.iO only contains Property Affordances and Event Affordance. In cloud.iO, we do not have the concept of Action on an Endpoint. The logic of *toggleRegulation* action, seen in TD from Listing 4, can be recreated on the Endpoint using cloud.iO library or on an application using cloud.iO API, but this action is up to the cloud.iO user and so cannot be described pragmatically in the TD.

The *setPointTemperature* is described in cloud.iO as a SetPoint, so a read and write parameter. We have three possible ways to interact with the *setPointTemperature* whose are described in the forms of its Property Affordance: by using the https REST API to read the property by using MQTTS to publish the property to a topic and by using the HTTPS REST API to write the property. This Property Affordance is represented in Listing 36. In WoT, the Property Affordance is a subclass of DataSchema whose metadata describes the data format used. In our case, the Property Affordance is as for type object, and so we can use the ObjectSchema subclass of DataSchema to describe our Property Affordance. This is where the *properties* field comes from. This field describes the cloud.iO Attribute data model. It specifies the data schema expected when updating or retrieving an Attribute in cloud.iO. The class diagram showing relation through the different Interaction Affordances is shown in Figure 17.

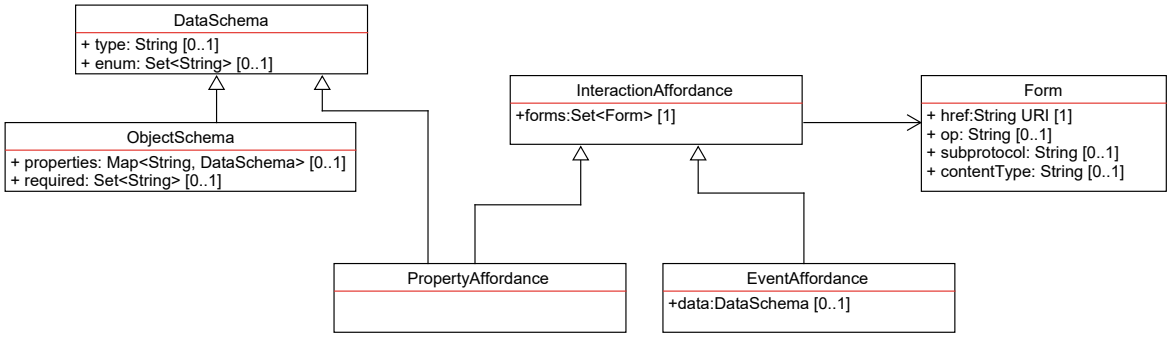


Figure 17: Class diagram of the Interaction Affordance used in cloud.iO following W3C Candidate Recommendation

```
{
...
  "properties": {
    ...
    "endpointUUID.myHeater.temperatures.setPointTemperature": {
        "type": "object",
        "properties": {
          "constraint": {
              "type": "string",
              "enum": ["SetPoint"]
          },
          "type": {
              "type": "string",
              "enum": ["Integer"]
          },
          "timestamp": {
              "type": "number"
          },
          "value": {
              "type": "number"
          }
        },
        "required": ["constraint", "type", "timestamp", "value"],
       "forms": [{
          "href": "https://cloud.iO:8081/api/v1/getAttribute/endpointUUID.myHeater.
    ↪ temperatures.setPointTemperature",
          "op": "readproperty",
          "contentType": "application/json"
        },
        {
          "href": "mqtts://cloud.iO:8883/@set/endpointUUID/myHeater/temperatures/
    ↪ setPointTemperature",
          "op": "writeproperty",
          "contentType": "application/json"
        },
        {
          "href": "https://cloud.iO:8081/api/v1/setAttribute/endpointUUID.myHeater.
    ↪ temperatures.setPointTemperature",
          "op": "writeproperty",
          "contentType": "application/json"
        }]
      }
    }
  },
  ...
}
```

Listing 36: Property affordance of Heater System setpoint

Listing 37 show the Event Affordance linked to *setPointTemperature* located under its topic. Its forms describe how a TD consumer can be notified of any change to the parameters. We can see the two different possibilities: by using the HTTPS REST API and using a longpoll or by using MQTTS and subscribing to a topic. The Event Affordance has a field named Data. This field is also a DataSchema, and it defines the event instance messages pushed by the Thing. In our case, this message is also a cloud.iO Attribute, and so the data field will describe its data model. The full TD of the smart heating system can be found in appendix A10.0.

```
{
...
  "events": {
    ...
    "endpointUUID.myHeater.temperatures.setPointTemperature": {
      "data": {
        "type": "object",
        "properties": {
          "constraint": {
              "type": "string",
              "enum": ["SetPoint"]
          },
          "type": {
              "type": "string",
              "enum": ["Integer"]
          },
          "timestamp": {
              "type": "number"
          },
          "value": {
              "type": "number"
          }
        },
        "required": ["constraint", "type", "timestamp", "value"]
      },
      "forms": [{
        "href": "https://cloud.io:8081/api/v1/notifyAttributeChange/endpointUUID.myHeater.
    ↪ temperatures.setPointTemperature",
        "op": "subscribeevent",
        "subprotocol": "longpoll",
        "contentType": "application/json"
      },
      {
        "href": "mqtts://cloud.io:8883/@set/endpointUUID/myHeater/temperatures/
    ↪ setPointTemperature",
        "op": "subscribeevent",
        "contentType": "application/json"
      }]
    }
  }
}
```

Listing 37: Event affordance of Heater System setpoint

3.9 cloud.iO v0.2 core architecture

All the enhancement presented in this paper added plenty of modifications to the cloud.iO core architecture. Listing 18 show the architecture of cloud.iO core updated for the version 0.2. The first services are roughly the same, and we can see the services linked to new features: MongoLogsService, InfluxLogsService, ExecJobsService, and Endpoint-TransactioneService. We can also see the update of the InfluxUpdateService now called InfluxUpdateSetService handling *@update* and *@set* messages. The last observation that can be done is the absence of service linking *@update* messages and MongoDB, previously called MongoUpdateService. It has been chosen to remove this service to enhance cloud.iO performance. Indeed, it was taking to many resources to retrieve data from MongoDB, modifying the JSON model and saving it again at each *@update* messages. When we are retrieving a resource model (Endpoint, Node, Object, or Attribute) with the API, the main model is taken from MongoDB, and the value and timestamps of the Attribute are populated with the last retrieved value from InfluxDB. This workaround is detailed in the Test And Result under the Subsection 4.2 dedicated to performance evaluations.

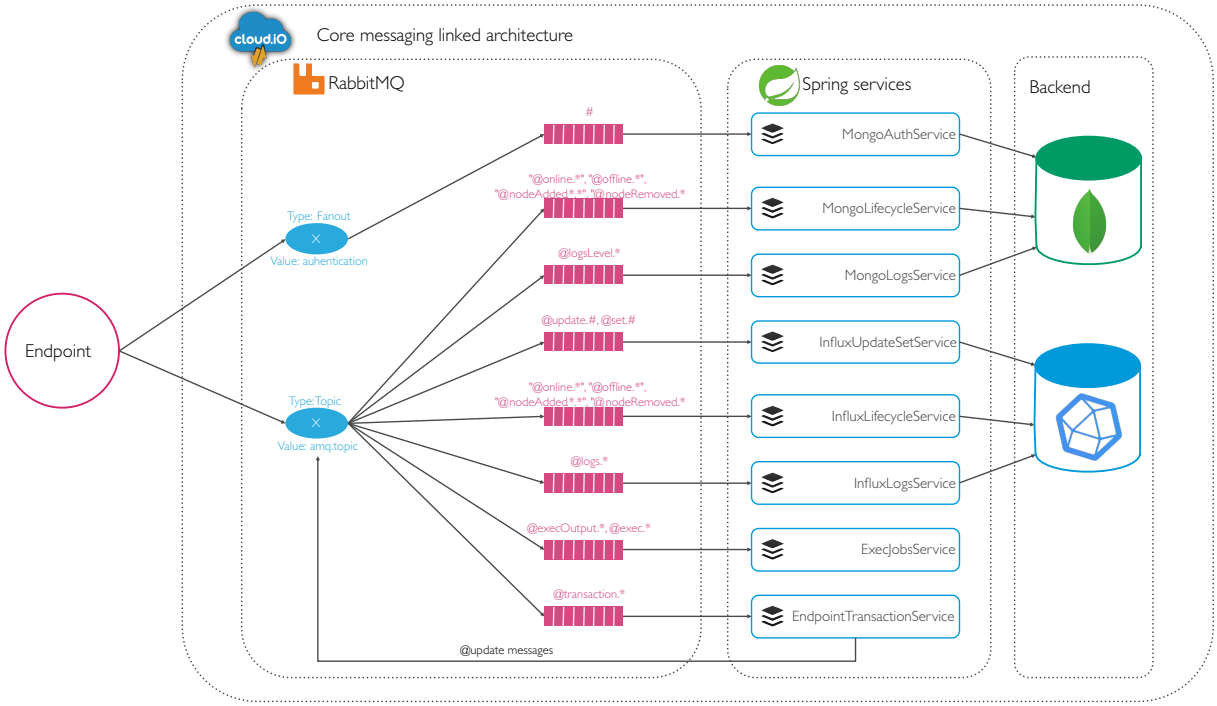


Figure 18: Final Architecture of the Core of cloud.iO v0.2

We reviewed the Messaging linked part of the core of cloud.iO, but a new big part of the core has been added with version 0.2: the RESTful API. Figure 19 show the different components linked to this API. We can see that a User will make every API request on a specific port of cloud.iO. The User is then authenticated by using SpringMVC SecurityConfiguration and a custom UserDetailsService retrieving User data in MongoDB. The HTTP request of the User is handled by the correct Rest Controller whose will interact with the backend or with RabbitMQ.

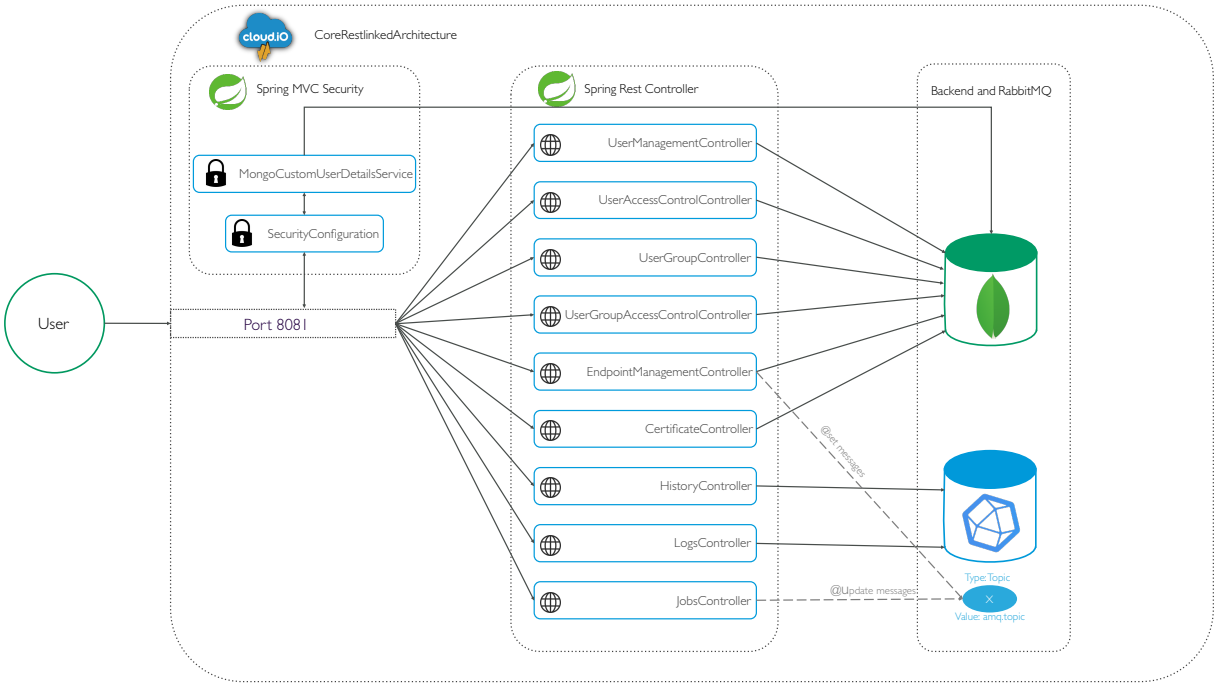


Figure 19: Final Architecture of the Core of cloud.iO v0.2

Finally, we will take a look at the evolution of the backends. In Figure 20, we can see the new organization of the cloud.iO v0.2 backend. For MongoDB, besides the minor changes on the UserEntity and EndpointEntity model, we can see the new UserGroup model linked to cloud.iO access control. For InfluxDB, we still have the History Repository saving the lifecycle of the Endpoint and Attribute modification with *@update* and now *@set* messages. We also now have the Logs Repository in InfluxDB, allowing Endpoint logging directly to the cloud.

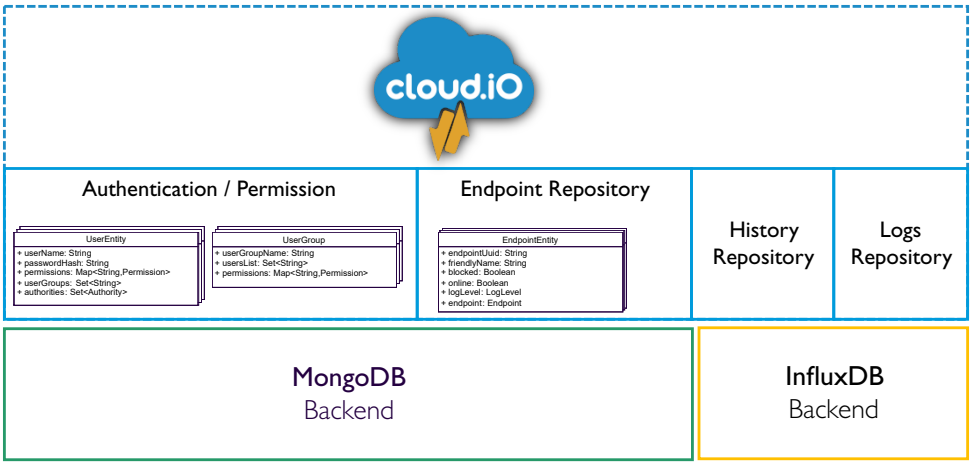


Figure 20: Final Backend Architecture of cloud.iO v0.2

4 Tests and Results

4.1 Web of Thing TD Evaluation

To ensure compatibility with the Web of Thing candidate recommendation, we have to ensure that the TD implementation of cloud.iO is correct. Firstly W3C provide a JSON Schema document that can be used to validate the syntax of the TD serialized in JSON[41]. A JSON Schema describes the structure of data for its validation. Plenty of tools are available to do the validation. In our case, jsonschemalint[42] has been used, and with the JSON Schema proposed by WoT, we had confirmed the syntactic validity of our implementation.

After validation of the JSON serialization, we can ensure that the JSON-LD implementation is correct. There are also tools available online. Json-ld.org[43] was used and did validate that our TD was following the JSON-LD standard.

Finally, the W3C Working Group did develop a Thing Description playground[44] providing Browser-based Thing Description Validation, Script based Thing Description Validation, Script based Assertion Tester and Batch Testing. The Browser-based Thing Description[45] has been used because of its simplicity. It is doing the JSON validation, JSON Schema validation, JSON-LD validation, and additional checks. With this tool, we fully validated the implementation of the TD.

4.2 Performance Evaluations

4.2.1 MQTT Messages Services

To evaluate the performance of the cloud.io core, we focused on the micro-services reacting to MQTT messages. The messages creating the biggest load on for the cloud.iO core are the *@update* messages. In cloud.iO v0.1, two services were reacting to those messages, one serializing the Attributes to MongoDB and one storing the Attributes as time-series in InfluxDB. As explain in the cloud.iO v0.2 core architecture subsection, we get rid of the service serializing the Attribute to MongoDB since it was taking a lot of resources, and the data was already present in InfluxDB. Only one service is now concerned by the *@update* message, and it takes many resources since its implementation take points and inserts them one by one in InfluxDB. It is possible to use round-robin to remove the load on the service, but saving data point by point would always result in bad performance. To enhance the performance of this service, we are using the Batch concept of InfluxDB. The InfluxDB library does accumulate all the points we want to send in a batch and send them all at once. We can configure the number of points in a Batch and the flush duration where the batch is sent, even if it is not full. Listing 38 shows how the Batch is enabled in InfluxUpdateSetService. We let the possibility of the cloud.iO administrator to configure the batch size and the flush duration in the application.properties file.

```
@PostConstruct
fun initialize() {
// Create database if needed
if (influx.query(Query("SHOW DATABASES", "")).toString().indexOf(database) == -1)
influx.query(Query("CREATE DATABASE $database", ""))

influx.enableBatch(BatchOptions.DEFAULTS.actions(influxBatchSize).flushDuration(
    ↪ influxBatchTimeMs))
}
```

Listing 38: Batch initialisation of the InfluxUpdateSetService

With this setup, we can set how the cloud.iO core does react to stress tests with or without batches. Thousand @update messages per second were sent to a cloud.iO test environment on a ubuntu virtual machine, and its performance was evaluated. To send those messages, MQTT.fx[46] an MQTT client under the free Apache License 2.0 has been used. MQTT.fx provides the possibility to write simple scripts to send messages, and this has been

[41] https://www.w3.org/TR/wot-thing-description/#json-schema-for-validation
[42] https://jsonschemalint.com
[43] https://json-ld.org/playground/
[44] https://github.com/thingweb/thingweb-playground
[45] http://plugfest.thingweb.io/playground/
[46] https://mqttfx.jensd.de

done to send a burst of @*update* message to cloud.iO. The script is available on Appendix A11.0. Table 14 show the evaluation of the InfluxUpdateSetService service in charge of processing the @*update* message.

	Batch Disabled	**Batch Enabled (flush duration = 1s, batch size = 3000)**
1 message/1batch	7ms	50ms
20'000 messages (~2000 messages/s)	2min52 (~111ms messages/s)	10s (~2000 messages/s)

Table 14: Evaluation of the InfluxUpdateSetService to process @update messages

We can see that the time needed to send a batch is higher than the time to send only one data point. But since the batch is composed of 3000 points in our example, the gain in time is significant. In the example, we also did send 2000 messages/s. Without batch, cloud.iO could only process and send 111ms messages/s to InfluxDB, but with batch enabled, it was following the sending rate. Those measures of time are only valid for the test implementation: a cloud.iO installation on a virtual machine. They are presented to show the significant improvement using the batches. We can see that with one service, we can easily handle a significant highest number of messages, and the need for round-robin implementation is not needed anymore.

4.2.2 Reactivity

In an IoT installation, it is important to evaluate the reactivity of the sensors toward the gateway or cloud. In cloud.iO, this reactivity can be seen as the time needed to be notified of a change in an Endpoint triggered by @*update* messages. Two protocols can be used in cloud.iO v0.2 to be notified of those changes:

- **MQTT**: by subscribing on @*update* topic
- **RESTful API**: by using the notifyAttributeChange route and doing a longpoll

Figure 21 illustrate the sequence needed to evaluate this reactivity. We have the Endpoint, in charge of sending the @*update* message on the Attribute topic but also re-receiving it, and the User using the notifyAttributeChange route to be notified of changes on the same topic. The delay between the sending of the @*update* message and the delivery of it by MQTT and HTTP is measured.

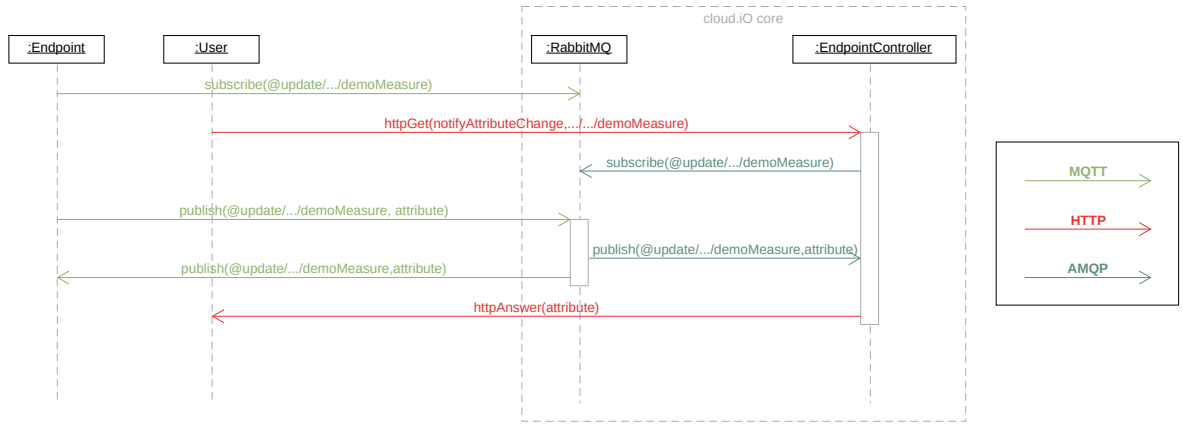


Figure 21: Sequence diagram for reactivity evaluation of cloud.iO messages

This sequence has been tested on 3 different network architectures: In the development environment locally on a virtual machine, with a cloud.iO dedicated server and an Endpoint on the same local network, and with a cloud.iO dedicated server and an Endpoint outside of this network. Table 15 show the results of this time analysis on 500 requests send from an Endpoint to cloud.iO. Appendix A15.0 show the full histograms regrouping the time measure of each request.

Delta time between	Environment	min	max	mean	SD
MQTT publish and MQTT subscribe	Virtual Machine	0.139ms	9.182ms	0.428ms	0.614ms
	Local network	0.165ms	10.596ms	0.513ms	0.512ms
	Non local network	0.145ms	10.688ms	0.513ms	0.488ms
MQTT publish and HTTP longpoll	Virtual Machine	2.336ms	142.098ms	5.884ms	9.833ms
	Local network	4.695ms	98.572ms	4.789ms	8.975ms
	Non local network	10.147ms	273.748ms	16.543ms	12.506ms

Table 15: Reactivity time analysis on 500 requests

As expected, MQTT is more reactive. Indeed, the messages are directly handled by the message broker as opposed to HTTP, where a dedicated service creates a queue dynamically to do the bridge between the MQTT message and the HTTP request.

From the measure in Table 15, we can conclude that the MQTT connection is stable compared to HTTP ones. If we look at the extrema of times (min, max) and the standard deviation (SD), which act as a dispersion indicator, we can observe that the MQTT connection stays stable and the connection environment between the Endpoint and cloud.iO does not affect it.

MQTT is more efficient for high-frequency notification, no data is missing, and the reactivity is good. With the HTTP longpoll, it is possible to miss data since it is needed to restart a request as soon as the previous one ends. It can still be used for Attributes with low-frequency of updates, but notification by MQTT is recommended.

4.3 Evaluations of the API

The RESTful is the major update for cloud.iO v0.2. It is essential to test its functionality; to do so, unit tests were implemented using JUnit. The RESTful API was not directly put to the test through HTTP requests, it is the abstraction layer for the API, the ApiUtils, that were tested. Every section of the API was tested separately, and a Test Report composed of 55 unit tests identifying 140 different Use-Cases is available on Appendix A12.0.

The API has also been tested with a Rest client: Insomnia[47]. All the API routes were described in Insomnia and tested with it. It is possible to import the Insomnia configuration for cloud.iO available on GitHub[48].

[47] https://insomnia.rest
[48] https://github.com/lucblender/cloudio-api-tools/blob/master/Insomnia-cloudiO-Api.json

4.4 Demonstrator

In this whole document, the smart heating system was taken as example. As a demonstration, the heating system example has been implemented with the Java library. In addition, a GUI has been developed in Python using the User RESTful API to:

- retrieve the ambient temperature
- retrieve and set the setpoint temperature
- retrieve and show the Endpoint model in JSON
- retrieve and show a graph of the ambient temperature

The cloud.iO data model implementation is shown in Listing 39 for the Node and the Object. We can see the DemoHeater inheriting CloudioNode, having only one DemoTemperature Object. This DemoTemperatures inheriting CloudioObject has the two Attributes temperature and setPointTemperature.

```
1  public class DemoHeater extends CloudioNode {
2
3    public DemoTemperatures temperatures;
4  }
5
6  public class DemoTemperatures extends CloudioObject {
7
8    @Measure
9    public CloudioAttribute<Double> temperature;
10
11   @SetPoint
12   public CloudioAttribute<Double> setPointTemperature;
13
14 }
```

Listing 39: Java code to represent the smart heater system, example composed of a CloudioNode and a CloudioObject

Listing 40 shows how the cloud.iO Endpoint is used in a Runnable class that has a Run method allowing it to be launched in a separate thread. On line 10 we can see the creation of the Endpoint with an UUID. Then on line 16, we have the assignation of the DemoHeater Node. On line 23 is the implementation of the Attribute listener to retrieve the set point of temperature. Finally, on line 35 we can find the run function in charge of simulating the heater regulation. After the regulation (eluded in the example to keep simplicity), the ambient temperature is measured and send again to cloud.iO with the setValue method.

```java
class Heater implements Runnable {
  CloudioEndpoint myEndpoint;
  DemoHeater demoHeater;

  private double ambientTemperature;
  private double setPointTemperature;

  Heater(double ambientTemperature, double setPointTemperature){
    try{
      myEndpoint = new CloudioEndpoint("bc0f1bf8-bdae-11e9-9cb5-2a2ae2dbcce4");
    }catch (Exception e){
      e.printStackTrace();
    }

    try {
      myEndpoint.addNode("myHeater", DemoHeater.class);
    } catch (Exception e1) {
      e1.printStackTrace();
    }

    demoHeater = myEndpoint.getNode("myHeater");

    demoHeater.temperatures.setPointTemperature.addListener(new CloudioAttributeListener() {
      @Override
      public void attributeHasChanged(CloudioAttribute attribute) {
        setSetPointTemperature((double)attribute.getValue());
      }
    });

    this.ambientTemperature = ambientTemperature;
    this.setPointTemperature = setPointTemperature;
  }

  @Override
  public void run() {
    while(true){
      try {
        // do heater regulation
        ...
        demoHeater.temperatures.temperature.setValue(ambientTemperature);
      }
      catch (Exception e){
      }
    }
  }
}
```

Listing 40: Java implemenation of the Heating System Logic

Finally, we can instantiate our heating system and launch the regulation thread, as shown on Listing 41. A more in-depth description of this software is available on Appendix A13.0 and the full code of this example is available on GitHub[49].

[49] https://github.com/lucblender/cloudio-endpointHeater-demo

```java
public class DemoHeaterApplication {

  static CloudioEndpoint myEndpoint;

  public static void main(String[] args) {
    Heater heater = new Heater(23.2, 25.1);
    heater.run();
  }
}
```

Listing 41: Java example of instantiation of heater system

As said earlier, a Python application with a GUI has been developed to interact with the simulated smart heater. This application is exclusively using the User RESTful API. HTTP requests can easily be made with the *requests* library; the response body is serialized in a dictionary. Listing 42 shows how to execute a request to the cloud.iO API. The used route is *getEndpoint* to retrieve and Endpoint Entity model and finally print the friendlyName and logLevel of the Endpoint. The User-Interface of the Software is shown in Figure 22. A more in-depth description of this software is available on Appendix A14.0 and the full code is available on GitHub[50].

```python
import requests

endpointUUID = "bc0f1bf8-bdae-11e9-9cb5-2a2ae2dbcce4"
baseURL = "https://cloud.iO:8081/api/v1/"
credential = ("userName", "password")

response = requests.request(method='post',url=baseURL+'getEndpoint', auth=credential, json={"
    ↪ endpointUuid": endpointUUID})
print(response["friendlyName"])
print(response["logLevel"])
```

Listing 42: Python example of an HTTP request on the cloud.iO RESTful API

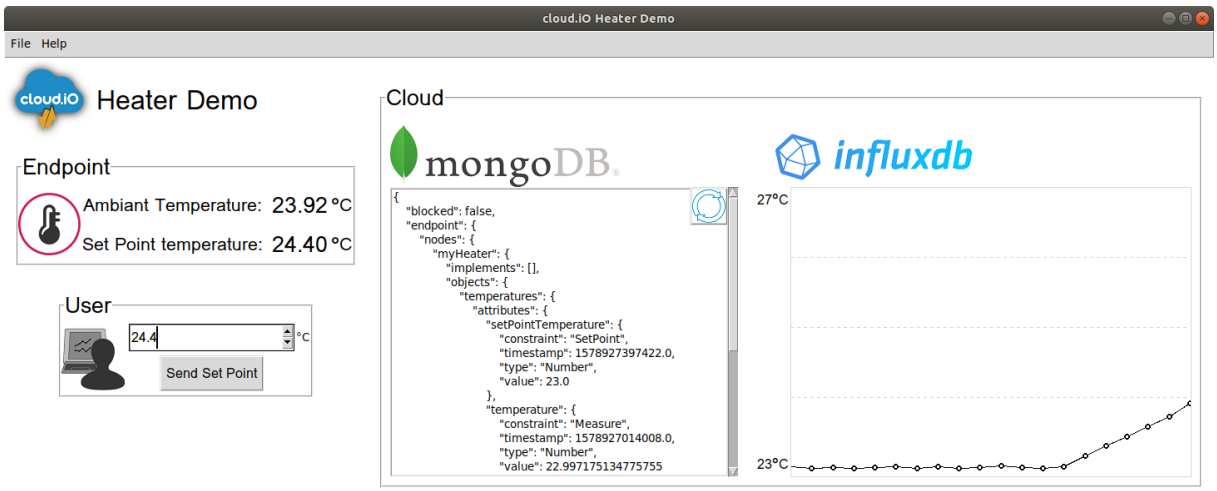


Figure 22: User Interface of monitoring software for the smart heater demo

[50] https://github.com/lucblender/cloudio-heater-demo

4.5 Deployment

All the new tools helping for the deployment of cloud.iO, described further in this section, are available on GitHub in the cloud.iO core repository cloudio-services under the cloudio-dev-environment package[51].

As for cloud.iO v0.1, this new version works with docker containers. A docker compose file has been created to launch the docker containers of InfluxDB, MongoDB, and the custom version of RabbitMQ for cloud.iO.

For the first deployment of an instance of cloud.iO v0.2, the creation of the certificate for the MQTT broker can be automatized with a MakeFile. In contrast with the MakeFile of cloud.iO v0.1, it is only in charge of creating the ca certificate and server certificates. All the management of certificates is then be done from those created certificates by the cloud.iO core and the RESTFul API.

Finally, a build.gradle file has been created to simplify the whole deployment. This gradle file uses the docker compose, and the MakeFile previously cited to ease the deployment of cloud.iO core.

The deployment of the micro-services can be simply done by deploying the whole gradle project cloudio-services. A docker container containing the cloud.iO core is also available, it is created automatically with GitHub and the Travis, a continuous integration tool. The docker container is available on dockerhub[52].

[51] https://github.com/cloudio-project/cloudio-services/tree/develop/cloudio-dev-environment
[52] https://hub.docker.com/r/cloudio/cloudio-services

5 Conclusion

5.1 Project Overview

The first version of cloud.iO was evolving through the research project of the HEVs. For cloud.iO v0.2, it has been decided to evolve based on the feedback of cloud.iO v0.1 users and according to the IoT landscape. After an in-depth analysis of the first version of cloud.iO and the IoT landscape to retrieve the recurrent features of IoT cloud solutions, the specifications for a v0.2 of cloud.iO have been established. Those specifications regroup the following features: redesign of access control, Remote Job Execution, logging, transactions, certificate management, RESTful API, and Web of Things compatibility.

The features composing the specifications of cloud.iO v0.2 has been successfully implemented. The access control has been enhanced with permission based on topics and the introduction of User Groups. Features like Remote Job Execution and Logging has been implemented in the cloud.iO core and Java library. The certificate management has been implemented directly in the cloud.iO core, removing the need for manual certificate management, as seen in cloud.iO v0.1. The major part of the cloud.iO enhancement part was the RESTful API. It has been implemented successfully, adding tools for users, User Group, access control and Endpoint management, certificate creation, history, and log access, and Remote Job Execution. Finally, cloud.iO has been made compliant to the W3C candidate recommendation aiming to the standardization of the IoT called Web of Things.

The many new features of cloud.iO v0.2 have been tested. Firstly, the Web of Thing implementation has been tested and validated with tools provided by W3C. Particular attention has been made to the RESTful API whose has been deeply tested. The performance of this new version of cloud.iO has also been evaluated, and a test deployment has been made. Finally, a little demonstrator has been implemented, taking back the smart heating system.

All those enhancements give a new version of cloud.iO, who kept the strength of cloud.iO v0.1 and did evolve according to the IoT landscape.

5.2 Future Improvement

cloud.iO v0.2 comes with many new features. Those features are meant to evolve with the cloud.iO project and are open to improvements.

Firstly we can talk about cloud.iO access control. The actual state of the access control is authentication with certificates for Endpoints, and user name and password for Users. An interesting alternative to this authentication is the use of JSON Web Token (JWT). JWT is defined by the RFC 7519[14] standard. It consists of information contained in a JSON object that will be securely transmitted. A JWT can be encrypted and be signed digitally. By its characteristics, JWT can be used for authentication, authorization, secure data exchange, and so a good alternative to certificates. In the case of cloud.iO, it could be used for User and Endpoint authentication to handle resource authorization and even to pass data, for example, different Endpoint parameters, from the Endpoint to the cloud.

cloud.iO v0.2 did bring compliance with W3C Web Of Things. As said earlier, WoT is still a candidate recommendation and is in constant evolution, thanks to the WoT Working Group. cloud.iO needs to follow the WoT evolution actively. We can take as an example of the Protocol Binding Templates that are continually evolving. For example, cloud.iO would have an interest in implementing the MQTT Protocol Binding Template. This Protocol Binding Template is described in their Working Group Note [53] whose last version to this day is dated to March 2018 and in their Editor's Draft [54] dated to the 22 November 2019. In those two versions, the MQTT Protocole Binding Template already evolved, and its implementation would be interesting when The Protocole Binding Template is in a stable form.

To connect to cloud.iO, a User has two possibilities, the use of a messaging protocol and the RESTful API provided by cloud.iO v0.2. For web development, only the RESTful API can be used, it is a reliable but old technology. It would be interesting to adapt the cloud.iO API to a more recent web technology like WebSocket, for example. Websocket also comes as a good solution since it is possible to do MQTT over WebSocket, making any cloud.iO transaction possible with this technology. Since the API has a decoupling layer, adding more web technology to interface it can be an easy task. Another interesting web technology that could be applied to the cloud.iO API is socket.io[55] that provides realtime analytics. socket.io uses technologies like WebSocket or traditional HTTP 1.1 mechanism to ensure the most reliable realtime communication.

We did an overview of the future improvement that are all linked to cloud.iO core. We will now focus on the cloud.iO library. In the first place, it is essential to maintain the existing library. The Java library is already compatible with cloud.iO v0.2 since it was part of this thesis. This open access of the Endpoint library to Kotlin User, although we could add a Kotlin layer to add language-specific features. In the case of the Python library, as said earlier, it is still in early development. The first step will be to make the existing feature compatible with cloud.iO v0.2. To do so, the Python library needs to works with reduced topics and support retained messages for *@update* notifications. Then new features as logging, transaction, and Remote Job Execution can be added. The final evolution of the cloud.iO Endpoint library is the creation of a Native C++ library whose all other libraries will be derivated. This means the main cloud.iO library will be made in C++, making possibility for micro-controller to use cloud.iO and to build language wrapper around it. With this idea in mind, we can understand that major updates of the library will only be needed in the C++ library.

Lucas Bonvin | Date

[53] https://www.w3.org/TR/2018/NOTE-wot-binding-templates-20180405/
[54] https://w3c.github.io/wot-binding-templates/
[55] https://socket.io/

6 References

[1] IDC, "*The Growth in Connected IoT Devices Is Expected to Generate 79.4ZB of Data in 2025, According to a New IDC Forecast*," 2019, https://www.idc.com/getdoc.jsp?containerId=prUS45213219.

[2] Cisco and/or its affiliates, "Cisco Global Cloud Index: Forecast and Methodology, 2016–2021," 2018.

[3] P. Roduit, D. Gabioud, G. Basso, G. Maitre, and P. Ferrez, "cloud.io: A decentralised iot architecture to control electrical appliances in households," in *Smart Cities, Green Technologies and Intelligent Transport Systems*, B. Donnellan, C. Klein, M. Helfert, and O. Gusikhin, Eds. Cham: Springer International Publishing, 2019, pp. 67–89.

[4] P. P. Ray, "A survey of iot cloud platforms," *Future Computing and Informatics Journal*, vol. 1, no. 1, pp. 35 – 46, 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2314728816300149

[5] A. M. Ghosh and K. Grolinger, "Deep learning: Edge-cloud data analytics for iot," in *2019 IEEE Canadian Conference of Electrical and Computer Engineering (CCECE)*, May 2019, pp. 1–7.

[6] P. Ferrari, S. Rinaldi, E. Sisinni, F. Colombo, F. Ghelfi, D. Maffei, and M. Malara, "Performance evaluation of full-cloud and edge-cloud architectures for industrial iot anomaly detection based on deep learning," in *2019 II Workshop on Metrology for Industry 4.0 and IoT (MetroInd4.0 IoT)*, June 2019, pp. 420–425.

[7] Google Cloud, "*One click to deploy a RabbitMQ cluster handling over 1 million msg/sec*," 2014, https://cloudplatform.googleblog.com/2014/06/rabbitmq-on-google-compute-engine.html accessed the 19.12.2019.

[8] I. P. Žarko, S. Mueller, M. Płociennik, T. Rajtar, M. Jacoby, M. Pardi, G. Insolvibile, V. Glykantzis, A. Antonić, M. Kušek, and S. Soursos, "The symbiote solution for semantic and syntactic interoperability of cloud-based iot platforms," in *2019 Global IoT Summit (GIoTS)*, June 2019, pp. 1–6.

[9] T. Jell, A. Bröring, and J. Mitic, "Big iot – interconnecting iot platforms from different domains," in *2017 International Conference on Engineering, Technology and Innovation (ICE/ITMC)*, June 2017, pp. 86–88.

[10] M. Kovatsch, R. Matsukura, M. Lagally, T. Kawaguchi, K. Toumura, and K. Kajimoto, "*Web of Things (WoT) Architecture*," 2019, https://www.w3.org/TR/wot-architecture/.

[11] S. Kaebisch, T. Kamiya, M. McCool, V. Charpenay, and M. Kovatsch, "*Web of Things (WoT) Thing Description*," 2019, https://www.w3.org/TR/wot-thing-description/.

[12] *Hypertext Transfer Protocol (HTTP/1.1) (DEPRECATED)*, Internet Engineering Task Force, 06 1999. [Online]. Available: https://tools.ietf.org/html/rfc2616

[13] *Hypertext Transfer Protocol (HTTP/1.1)*, Internet Engineering Task Force, 06 2014. [Online]. Available: https://tools.ietf.org/html/rfc7230

[14] *JSON Web Token (JWT)*, Internet Engineering Task Force, 05 2015. [Online]. Available: https://tools.ietf.org/html/rfc7519

Appendices

A1.0

All available cloud.iO repositories and docker containers location

Description	URL
Main GitHub page for the cloud.iO project	https://github.com/cloudio-project
Deprecated version of cloud.iO v0.1 core and Java library	https://github.com/cm0x4D/cloudio
cloud.iO Core, development and release version	https://github.com/cloudio-project/cloudio-services
cloud.iO Java Endpoint library, development and release version	https://github.com/cloudio-project/cloudio-endpoint-java
cloud.iO Python Endpoint library, development and release version	https://github.com/cloudio-project/cloudio-endpoint-python
Any types of documents about cloud.iO	https://github.com/cloudio-project/cloudio-documents
Grafana data source plugin, still in early development	https://github.com/cloudio-project/cloudio-grafana-plugin
cloud.iO documentation (Deprecated, will be updated)	https://github.com/cloudio-project/cloudio-documentation
Docker image of RabbitMQ for cloud.iO	https://github.com/cloudio-project/cloudio-rabbitmq-docker
cloud.iO website, available online at http://cloudio.hevs.ch/	https://github.com/cloudio-project/cloudio-project.github.io
Demo of a smart heating system using Java cloud.iO Endpoint library	https://github.com/lucblender/cloudio-endpointHeater-demo
Demo of a GUI for the smart heating system using the RESTful API of cloud.iO	https://github.com/lucblender/cloudio-heater-demo
Tools to be used for the RESTful API: Swagger definition and Insomnia implementation	https://github.com/lucblender/cloudio-api-tools
Docker container for cloud.iO core: cloudio/cloudio-services	https://hub.docker.com/r/cloudio/cloudio-services
Docker conainter for custom RabbitMQ for cloud.iO: cloudio/cloudio-rabbitmq	https://hub.docker.com/r/cloudio/cloudio-rabbitmq

IoT Cloud based framework analysis

N°	IoT Platform	Device Monitoring	Device History	History visualization	Rules Engine	Remote Jobs Execution	Cloud Computing / Data Analysis	Cloud Storage	On-premises deployments	OpenSource	Rest API	Normalization	Security / Authentication	Specific Hardware	Prix	SDK	IoT Protocol	URL	Device Twin URL	HTTP/REST Api URL	Access Control	Message Broker	Note
1	Azure IOT	x	x	x	x		x	x			x		token certificate	No		C C# Java Python Node.js	REST AMQP AMQP over WebSockets MQTT MQTT over WebSockets	https://docs.microsoft.com/en-us/azure/iot-hub/about-iot-hub	https://docs.microsoft.com/en-us/azure/iot-hub/iot-hub-devguide-device-twins	https://docs.microsoft.com/en-us/rest/api/iothub/device	Role-based access control on virtual machin and database	Iot hub	Provide machine-learning on cloud
2	Aws IOT	x	x	x	x	x	x	x			x		token certificate	No		Android IOS C C++ Java JavaScript Python	MQTT MQTT over WebSockets REST	https://docs.aws.amazon.com/iot/latest/developerguide/what-is-aws-iot.html	https://docs.aws.amazon.com/iot/latest/developerguide/device-shadow-document-syntax.html	https://docs.aws.amazon.com/en_pv/iot/latest/apireference/API_Operations_AWS_IoT.html	Role-based access control on Amazon Service	Aws Message Broker	
3	Google Cloud IOT Core	x	x	x		x	x	x			x		token certificate	No		Google Cloud SDK --> Set of cmd Google provide example	REST MQTT	https://cloud.google.com/iot/docs/		https://cloud.google.com/iot/docs/reference/rest/	Role-based access control on Google IoT Service Access either on project or registry level		Provide machine-learning on gateways and edge device
4	Alibaba Cloud	x	x	x	x		x	x			x		token certificate	No		C Node.js Java	REST MQTT CoAP	https://www.alibabacloud.com/product/iot	https://www.alibabacloud.com/help/doc-detail/53932.html?spm=a2c5t.10695662.1996646101.searchclickresult.23f34366H9zdVe	https://www.alibabacloud.com/help/doc-detail/30560.htm?spm=a2c63.p38356.879954.6.5cd46a8aFzynfC#reference-vy5-swc-xdb	User access Control on actions		2G 3G 4G NB-IoT Lora
5	Eclipse Kapua	x	x	x					x	x	x		login-password token certificate	No		-	REST MQTT MQTT over WebSockets	https://www.eclipse.org/kapua		https://www.eclipse.org/kapua/docs/1.1.0/user-manual/en/rest.html	Permissions		
6	thingsboard	x	x	x	x	x		x	x	x	x		token certificate	No		-	REST MQTT CoAP	https://thingsboard.io		https://thingsboard.io/docs/user-guide/telemetry/	Role Base Access Control on Device Difference between Free and Paid version 3 roles on free, customable on paid	Own broker using Apache Netty	Support also non Lan LoRaWAN SigFox NB IoT or SMS
7	kaa platfrom	x	x	x		x		x	x		x		login-password token certificate	No		-	REST MQTT CoAP	https://www.kaaproject.org/overview		https://docs.kaaiot.io/CM/docs/current/REST-API/	Global user access Control	Nats	
8	mainflux labs	x	x	x	x				x	x	x		token certificate	Yes, not mandatory		C C++ Javascript Python	HTTP MQTT MQTT over WebSockets CoAP REST	https://www.mainflux.com		https://mainflux.readthedocs.io/en/latest/messaging/#http	Acces control based on Channel (topic where things can communicate)	Aedes Verne	LoRa
9	Particle Cloud	x	x	x	x	x					x		token	Yes		Node.js	REST	https://www.particle.io/		https://docs.particle.io/reference/device-cloud/api/	Global Role-based access control		Small board with either Wifi & Cellular
10	B+B SmartWorx	x	x	x										Yes		MQTT	MQTT	http://advantech-bb.com/product-technology/iot-and-network-edge-platforms/smartworx-hub/			Not documented		Big gateway with wifi, Modbus, cellular and some sensor
11	Siot	x	x	x				x	x		x		UID based			-	MQTT REST	https://siot.net		https://siot.net/download	None	Verne	
12	Cloud.iO	x	x	x					x	x			login-password certificate	No		Java Python	MQTT AMQP				Role Base Access Control on Ressource (Endpoint)	RabbitMQ	
13	Cloud.iO 2.0	x	x	x		x			x	x	x	W3C	login-password certificate	No		Java Python C	MQTT MQTT over WebSockets AMQP REST				Role Base Access Control on Ressource (Topic -> Endpoint to Attribute)	RabbitMQ	

Low-complexity scalable IoT framework
Lucas Bonvin

A3.0

File application.properties example for cloud.iO core

```
spring.rabbitmq.host=rabbitHostURL
spring.rabbitmq.port=rabbitPort
spring.rabbitmq.username=rabbitUser
spring.rabbitmq.password=rabbitPassword

spring.influx.url=http://influxHostURL:influxPort


cloudio.caPrivateKey= -----BEGIN PRIVATE KEY-----\r\n\
...\r\n\
-----END PRIVATE KEY-----


cloudio.caCertificate=-----BEGIN CERTIFICATE-----\r\n\
...\r\n\
-----END CERTIFICATE-----

server.port=restApiPort

cloudio.caCertificateJksPath=Path/ca-cert.jks
cloudio.caCertificateJksPassword=caCertificatePassword

cloudio.influxBatchIntervallMs=10000
cloudio.influxBatchSize=2000

server.ssl.key-store-type=PKCS12
server.ssl.key-store=PathToKeyStoreForHttps
server.ssl.key-alias=KeyStoreAlias
server.ssl.key-store-password=KeyStorePassword
```

A4.0

Complete Topic list and the corresponding data to send to cloud.iO v0.2

Topic	Data
@update/*EndpointUUID*/.../AttributeName	The status or measure Attribute formatted in JSON
@set/*EndpointUUID*/.../AttributeName	The setpoint or parameter Attribute formatted in JSON
@online/*EndpointUUID*	The Endpoint Data Model formatted in JSON
@nodeAdded/*EndpointUUID*/*NodeNameToAdd*	The Node to add formatted in JSON
@nodeRemoved/*EndpointUUID*/*NodeNameToRemove*	no Data
@offline/*EndpointUUID*	no Data
@exec/*EndpointUUID*	Job Parameter formatted in JSON. User publish on this topic, Endpoint subscribe.
@execOutput/*EndpointUUID*	Job output line formatted in JSON by line. Endpoint publish on this topic, User subscribe.
@logs/*EndpointUUID*	cloud.iO log message formatted in JSON. User publish on this topic.
@logsLevel/*EndpointUUID*	Log level formatted in JSON . User publish on this topic, Endpoint subscribe.
@transaction/*EndpointUUID*	cloud.iO Transaction message formatted in JSON.

A5.0

@update/bc0f1bf8-bdae-11e9-9cb5-2a2ae2dbcce4/Meteo/temperatures/inside/temperature

```
{
  "type": "Number",
  "constraint": "Measure",
  "timestamp": 1560772747,
  "value": 19.2
}
```

@set/bc0f1bf8-bdae-11e9-9cb5-2a2ae2dbcce4/Meteo/temperatures/inside/temperature

```
{
  "type": "Number",
  "constraint": "SetPoint",
  "timestamp": 1560772747,
  "value": 19.2
}
```

@online/bc0f1bf8-bdae-11e9-9cb5-2a2ae2dbcce4

```
{
  "nodes": {
    "Meteo": {
      "implements": ["MeteoStation"],
      "objects": {
        "temperatures": {
          "conforms": null,
          "objects": {
            "inside": {
              "conforms": "TemperatureMeasure",
              "attributes": {
                "unit": {
                  "constraint": "Static",
                  "timestamp": null,
                  "value": "K"
                },
                "temperature": {
                  "type": "Number",
                  "constraint": "Measure",
                  "timestamp": 1560772747,
                  "value": 19.2
                }
              }
            }
          }
        }
      }
    }
  }
}
```

A5.1

@nodeAdded/bc0f1bf8-bdae-11e9-9cb5-2a2ae2dbcce4/newNode


```
{
  "implements": ["MeteoStation"],
  "objects": {
    "temperatures": {
      "conforms": null,
      "objects": {
        "inside": {
          "conforms": "TemperatureMeasure",
          "attributes": {
            "unit": {
              "constraint": "Static",
              "timestamp": null,
              "value": "K"
            },
            "temperature": {
              "type": "Number",
              "constraint": "Measure",
              "timestamp": 1560772747,
              "value": 19.2
            }
          }
        }
      }
    }
  }
}
```


@nodeRemoved/bc0f1bf8-bdae-11e9-9cb5-2a2ae2dbcce4/newNode

No data for this topic

@offline/bc0f1bf8-bdae-11e9-9cb5-2a2ae2dbcce4

No data for this topic

@exec/bc0f1bf8-bdae-11e9-9cb5-2a2ae2dbcce4e


```
{
  "jobURI":"cmd://listJobs",
  "sendOutput":true,
  "correlationID":"1fa65431"
}
```


execOutput/bc0f1bf8-bdae-11e9-9cb5-2a2ae2dbcce4


```
{
    "correlationID":"1fa65431",
    "data":"\tlistJobs"
}
```


@logs/bc0f1bf8-bdae-11e9-9cb5-2a2ae2dbcce4


```
{
  "level":"ERROR",
  "timestamp":1.578583913934E9,
  "message":"error message",
  "loggerName":"DemoHeaterApplication",
  "logSource":"DemoHeaterApplication/main, line:19"
}
```


@logsLevel/bc0f1bf8-bdae-11e9-9cb5-2a2ae2dbcce4


```
{
  "level":"TRACE"
}
```

A5.2

@transaction/bc0f1bf8-bdae-11e9-9cb5-2a2ae2dbcce4


```
{
  "attributes": {
    "bc0f1bf8-bdae-11e9-9cb5-2a2ae2dbcce4/demoNode/demoObject/demoMeasure": {
      "constraint": "Measure",
      "type": "Number",
      "timestamp": 1574330321,
      "value": 99.9
    },
    "bc0f1bf8-bdae-11e9-9cb5-2a2ae2dbcce4/demoNode/demoObject/demoStatus": {
      "constraint": "Status",
      "type": "Integer",
      "timestamp": 1574330321,
      "value": 11
    }
  }
}
```

A6.0

File *endpointUUID*.properties example for use with cloud.iO Endpoint Library in Java

```
ch.hevs.cloudio.endpoint.hostUri=ssl://localhost:8883
ch.hevs.cloudio.endpoint.ssl.authorityCert=file:PATH/ca-cert.jks
ch.hevs.cloudio.endpoint.ssl.clientCert=file:PATH/ENDPOINTUUID.p12
ch.hevs.cloudio.endpoint.ssl.clientPassword=password
ch.hevs.cloudio.endpoint.ssl.authorityPassword=password
ch.hevs.cloudio.endpoint.persistence=false
ch.hevs.cloudio.endpoint.ssl.verifyHostname=false

ch.hevs.cloudio.endpoint.jobs.folder=PATH
```

Docker image of RabbitMQ for cloud.iO

Description

Docker image of **RabbitMQ** adapted to the needs of the *cloud.iO* IoT project.

The image contains an installation of the **RabbitMQ** message broker along with the **auth_backend_amqp** plugin and is already configured to be used in the context of a *cloud.iO* installation.

RabbitMQ is configured to listen to port **5672** for incomming **AMQP** (TCP) connections and to port **1883** for **MQTT** connections. We discourage to expose them as data is exchanged in plain text. The broker listens to port **5671** for **AMQPS** connections and on port **8883** for **MQTTS** connections. Both support **username/password** and **x509 certiciate** (common name is used as username) authentication.

Authentication and authorization is - apart some system accounts - handled by an external service that will connect to the message broker via AMQPS and handle all authentication and authorizatoin requests via **AMQP RPC**.

The following plugins are activated:

- **rabbitmq_auth_backend_amqp**:
  This plugin will forward the authentication and authorization requests to the *cloud.iO* security microservice, which is connected via AMQPS to the broker.
- **rabbitmq_auth_backend_cache**:
  Caches the decisions from the *cloud.iO* security microservice in order to limit the number of messages to send and receive.
- **rabbitmq_auth_mechanism_ssl**:
  Enables *cloud.io* endpoints and microservices to connect to the message broker via MQTTS or AMQPS using an x509 certificate.
- **rabbitmq_mqtt**:
  Enables MQTT and MQTTS compatibility layer.
- **rabbitmq_management**:
  RabbitMQ management interface.

Starting a container

The broker needs a valid SSL configuration in order to start. The certificate of the **Cretificate Authority** (CA) certificate, the **server certificate** and the **server private key** are all passed via environment variables (required):

- **CA_CERT**: The Certificate Authority's certificate in PEM format.
- **SERVER_CERT**: The server's certificate in PEM format.
- **SERVER_KEY**: The server's private key in PEM format.

If a local (RabbitMQ) admin user is required, you can set the environment variable **ADMIN_PASSWORD** to a secret password and the container will add a user called **admin** with that password to the RabbitMQ's local user base.

A typical example:

```
docker run -d -p 8883:8883 -p 5671:5671 -e CA_CERT='-----BEGIN CERTIFICATE-----
...
-----END CERTIFICATE-----' -e SERVER_CERT='-----BEGIN CERTIFICATE-----
...
-----END CERTIFICATE-----' -e SERVER_KEY='-----BEGIN PRIVATE KEY-----
...
-----END PRIVATE KEY-----' cloudio/cloudio-rabbitmq:latest
```

If you like to add an admin user, specify the admin user's password:

```
docker run -d -p 8883:8883 -p 5671:5671 -e CA_CERT='-----BEGIN CERTIFICATE-----
...
-----END CERTIFICATE-----' -e SERVER_CERT='-----BEGIN CERTIFICATE-----
...
-----END CERTIFICATE-----' -e SERVER_KEY='-----BEGIN PRIVATE KEY-----
...
-----END PRIVATE KEY-----' -e ADMIN_PASSWORD='MySecretPassword' cloudio/cloudio-rabbitmq:latest
```

General API list

- UserManagement
  - createUser
  - getUser
  - deleteUser
- UserAccessControl
  - getUserAccessRight
  - addUserAccessRight
  - modifyUserAccessRight
  - removeUserAccessRight
  - giveUserAccessRight
- UserGroup
  - createUserGroup
  - getUserGroup
  - getUserGroupList
  - addUserToGroup
  - deleteUserToGroup
  - deleteUserGroup
- UserGroupAccessControl
  - getUserGroupAccessRight
  - addUserGroupAccessRight
  - modifyUserGroupAccessRight
  - removeUserGroupAccessRight
  - giveUserGroupAccessRight
- EndpointManagement
  - createEndpoint
  - getEndpoint
  - getNode
  - getWOTNode
  - getObject
  - getAttribute
  - setAttribute
  - notifyAttributeChange
- Certificate
  - TODO
- History
  - getAttributeHistoryRequest
  - getAttributeHistoryByDateRequest
  - getAttributeHistoryWhere
  - getSetpointeHistoryRequest
  - getSetpointHistoryByDateRequest
  - getSetpointHistoryWhere
- Logs
  - getEndpointLogs
  - getLogsLevel
  - setLogsLevel
- Jobs/ScriptService
  - executeScript
  - updateScriptRepository

A8.0

API Access

API access control is based on user access control, in addition, the authority of the user is used. The user need to have **http_access** to access the rest api and need to **administrator** to access Admin tool.

User

User Management

createUser

Admin tool

Request Body:

```
{
    "userName": "username",
    "userRight": [
        {
            "topic": "toto/#",
            "permission": "OWNER"
        },
        {
            "topic": "toto/titi/*",
            "permission": "DENY"
        }
    ]
}
```


Lucas Bonvin
Low-complexity scalable IoT framework

Anwser: Json result of query

Answer Origin: MongoDB

getUser

Admin tool

Request Body:

```
{
    "userName": "username"
}
```

Anwser: Json of user

Answer Origin: MongoDB

deleteUser

Admin tool

Request Body:

```
{
    "userName": "username"
}
```

Anwser: Json result of query

Answer Origin: MongoDB

User Access Control

getUserAccessRight

Admin tool

Request Body:

```
{
    "userName": "username"
}
```

Anwser: Json containing user access right

Answer Origin: MongoDB

addUserAccessRight

Admin tool

Request Body:

```
{
    "userName": "username",
    "userRight": [
        {
            "topic": "toto/#",
            "permission": "OWNER"
        },
        {
            "topic": "toto/titi/*",
            "permission": "DENY"
        }
    ]
}
```

Anwser: Json result of query

Answer Origin: MongoDB

modifyUserAccessRight

Admin tool

Request Body:

```
{
    "userName": "username",
    "userRight": [
        {
            "topic": "toto/#",
            "permission": "DENY"
        }
    ]
}
```

Anwser: Json result of query Answer Origin: MongoDB

removeUserAccessRight

Admin tool

Request Body:

```
{
    "userName": "username",
    "userRightTopic": [ "toto/#", "toto/titi/*"]
}
```

Anwser: Json result of query

Answer Origin: MongoDB

giveUserAccessRight

To give right to an user on Owned Endpoint

Request Body:

```
{
    "userName": "username",
    "userRight": [
        {
            "topic": "toto/#",
            "permission": "OWNER"
        },
        {
            "topic": "toto/titi/*",
            "permission": "DENY"
        }
    ]
}
```

Anwser: Json result of query

Answer Origin: MongoDB

User Group

createUserGroup

Admin tool

Request Body:

```
{
    "userGroup": "userGroup",
    "users": ["toto", "titi"],
    "userGroupRight": [
        {
            "topic": "toto/#",
            "permission": "OWNER"
        },
```



```
        {
            "topic": "toto/titi/*",
            "permission": "DENY"
        }
    ]
}
```

Anwser: Json result of query

Answer Origin: MongoDB

getUserGroup

Admin tool

Request Body:

```
{
    "userGroup": "userGroup",
}
```

Anwser: Json of userGroup

Answer Origin: MongoDB

getUserGroupList

Admin tool

Request Body: empty

Anwser: Json list of all user group

Answer Origin: MongoDB

addUserToGroup

Admin tool

Request Body:

```
{
    "userGroup": "userGroup",
    "users": ["toto", "titi"]
}
```

Anwser: Json result of query

Answer Origin: MongoDB

deleteUserToGroup

Admin tool

Request Body:

```
{
    "userGroup": "userGroup",
    "users": ["toto", "titi"],
}
```

Anwser: Json result of query

Answer Origin: MongoDB

deleteUserGroup

Admin tool

Request Body:

```
{
    "userGroup": "userGroup"
}
```

Anwser: Json result of query

Answer Origin: MongoDB

UserGroup Access Control

getUserGroupAccessRight

Admin tool

Request Body:

```
{
    "userGroup": "userGroup"
}
```

Anwser: Json containing userGroup access right

Answer Origin: MongoDB

addUserGroupAccessRight

Admin tool

Request Body:

```
{
    "userGroup": "userGroup",
    "userGroupRight": [
        {
            "topic": "toto/#",
            "permission": "OWNER"
        },
        {
            "topic": "toto/titi/*",
            "permission": "DENY"
        }
    ]
}
```

Anwser: Json result of query

Answer Origin: MongoDB

modifyUserGroupAccessRight

Admin tool

Request Body:

```
{
    "userGroup": "userGroup",
    "userGroupRight": [
        {
            "topic": "toto/#",
            "permission": "DENY"
        }
    ]
}
```

Anwser: Json result of query Answer Origin: MongoDB

removeUserGroupAccessRight

Admin tool

Request Body:

```
{
    "userGroup": "userGroup",
    "userGroupRightTopic": [ "toto/#", "toto/titi/*"]
}
```

Anwser: Json result of query

Answer Origin: MongoDB

giveUserGroupAccessRight

To give right to an user group on Owned Endpoint

Request Body:

```
{
    "userGroup": "userGroup",
    "userGroupRight": [
        {
            "topic": "toto/#",
            "permission": "OWNER"
        },
        {
            "topic": "toto/titi/*",
            "permission": "DENY"
        }
    ]
}
```

Anwser: Json result of query

Answer Origin: MongoDB

Endpoint

Endpoint Management

createEndpoint

Request Body:

```
{
    "endpointFriendlyName":"toto",
    "returnCertificate": true
}
```


Anwser: Json with endpoint UUID

Answer Origin: MongoDB

getEndpoint

Request Body:

```
{
    "endpointUUID":"110e8400-e29b-11d4-a716-446655440000"
}
```

Anwser: Json of endpoint

Answer Origin: MongoDB

getNode

Request Body:

```
{
    "nodeTopic":"110e8400-e29b-11d4-a716-446655440000/nodeName"
}
```

Anwser: Json of node

Answer Origin: MongoDB

getWOTNode

Request Body:

```
{
    "nodeTopic":"110e8400-e29b-11d4-a716-446655440000/nodeName"
}
```

Anwser: JSON-LD of node

Answer Origin: MongoDB

getObject

Request Body:

```
{
    "objectTopic":"/endpointname/nodename/objectname"
}
```

Anwser: Json of Object

Answer Origin: MongoDB

getAttribute

Request Body:

```
{
    "attributeTopic":"/endpointname/nodename/objectname/attributename"
}
```

Anwser: Json of Attribute

Answer Origin: MongoDB

setAttribute

Request Body:

```
{
    "attributeTopic":"/endpointname/nodename/objectname/attributename"
    "value":"1010"
}
```

Anwser: Json result of query

Answer Origin: Query

notifyAttributeChange

longpoll Request Body:

```
{
    "attributeTopic":"/endpointname/nodename/objectname/attributename"
    "value":"1010"
}
```

Anwser: Json of Attribute when it is updated

Certificate

generateEndpointKeyAndCertificatePair

Request Body:


```
{
    "endpointUUID":"110e8400-e29b-11d4-a716-446655440000"
}
```

Anwser: Json with endpoint private key and certificate

Answer Origin: certificate service

generateEndpointCertificateFromPublicKey

Request Body:

```
{
    "endpointUUID":"110e8400-e29b-11d4-a716-446655440000"
    "publicKey":"rsaKey"
}
```

Anwser: Json with endpoint certificate

Answer Origin: certificate service

getCaCertificate

NoRequestBody

Anwser: Json with ca certificate

Answer Origin: certificate service

History

getAttributeHistoryRequest

Body:

```
{
    "attributeTopic":"/endpointname/nodename/objectname/attributename",
    "number":10
}
```

Anwser: Json version of sql answer

Answer Origin: InfluxDB

getAttributeHistoryByDateRequest

Body:

```
{
    "attributeTopic":"/endpointname/nodename/objectname/attributename",
    "dateStart" : "2019-03-10 00:00:00"
    "dateStop" : "2019-03-10 01:17:00"
}
```

Anwser: Json version of sql answer

Answer Origin: InfluxDB

getAttributeHistoryWhere

Request Body:

```
{
    "attributeTopic":"/endpointname/nodename/objectname/attributename",
    "where" : "time> now() - 2400h and time<now() - 1200h"
}
```

Anwser: Json version of sql answer

Answer Origin: InfluxDB

getSetpointeHistoryRequest

Body:

```
{
    "setpointTopic":"/endpointname/nodename/objectname/attributename"
}
```

Anwser: Json version of sql answer

Answer Origin: InfluxDB

getSetpointHistoryByDateRequest

Body:

```
{
    "setpointTopic":"/endpointname/nodename/objectname/attributename",
    "dateStart" : "2019-03-10 00:00:00"
    "dateStop" : "2019-03-10 01:17:00"
}
```

Anwser: Json version of sql answer

Answer Origin: InfluxDB

getSetpointHistoryWhere

Request Body:

```
{
    "setpointTopic":"/endpointname/nodename/objectname/attributename",
    "where" : "time> now() - 2400h and time<now() - 1200h"
}
```

Anwser: Json version of sql answer

Answer Origin: InfluxDB

Logs

getEndpointLogs

Request Body:

```
{
    "endpointUUID":"110e8400-e29b-11d4-a716-446655440000"
}
```

Anwser: Logs formated in JSON

Answer Origin: InfluxDB

getLogsLevel

Request Body:

```
{
    "endpointUUID":"110e8400-e29b-11d4-a716-446655440000"
}
```

Anwser: Logs level formated in JSON

Answer Origin: InfluxDB

setLogsLevel

Request Body:

```
{
    "endpointUUID":"110e8400-e29b-11d4-a716-446655440000"
}
```

Anwser: Json of query

Answer Origin: InfluxDB

Jobs/Script Service

executeScript

Request Body:

```
{
    "jobURI":"file://example.sh",   //cmd and file supported
    "getOuput": true
}
```

Anwser: Json of query, maybe output of script

Answer Origin: script

updateScriptRepository

Request Body:

```
{
    "scriptURL":"www.example.com"
}
```

Anwser: Json, result of update

Answer Origin: script pull

Lucas Bonvin

cloud.iO

1.0.0

[Base URL: virtserver.swaggerhub.com/luc_blender/cloud.iO-2.0/1.0.0]

User RESTful API to use cloud.iO

Schemes
HTTPS

Authorize

UserManagement

- POST /api/v1/createUser
- POST /api/v1/getUser
- GET /api/v1/getUser/{userName}
- DELETE /api/v1/deleteUser
- POST /api/v1/modifyUserPassword
- POST /api/v1/addUserAuthority
- DELETE /api/v1/removeUserAuthority
- POST /api/v1/getUserList

UserAccessControl

- POST /api/v1/getUserAccessRight
- GET /api/v1/getUserAccessRight/{userName}
- POST /api/v1/addUserAccessRight
- POST /api/v1/modifyUserAccessRight
- DELETE /api/v1/removeUserAccessRight
- POST /api/v1/giveUserAccessRight

UserGroup

- POST /api/v1/createUserGroup
- POST /api/v1/getUserGroup
- GET /api/v1/getUserGroup/{userGroupName}
- POST /api/v1/getUserGroupList
- POST /api/v1/addUserToGroup
- DELETE /api/v1/deleteUserToGroup
- DELETE /api/v1/deleteUserGroup

UserGroupAccessControl

- POST /api/v1/getUserGroupAccessRight
- GET /api/v1/getUserGroupAccessRight/{userGroupName}
- POST /api/v1/addUserGroupAccessRight
- POST /api/v1/modifyUserGroupAccessRight
- DELETE /api/v1/removeUserGroupAccessRight
- POST /api/v1/giveUserGroupAccessRight

EndpointManagement

POST	/api/v1/createEndpoint
POST	/api/v1/getEndpoint
GET	/api/v1/getEndpoint/{endpointUUID}
POST	/api/v1/getEndpointFriendlyName
GET	/api/v1/getEndpointFriendlyName/{endpointUUID}
POST	/api/v1/getNode
GET	/api/v1/getNode/{nodeTopic}
POST	/api/v1/getWOTNode
GET	/api/v1/getWOTNode/{nodeTopic}
POST	/api/v1/getObject
GET	/api/v1/getObject/{objectTopic}
POST	/api/v1/getAttribute
GET	/api/v1/getAttribute/{attributeTopic}
POST	/api/v1/setAttribute
POST	/api/v1/setAttribute/{attributeTopic}
POST	/api/v1/notifyAttributeChange
POST	/api/v1/notifyAttributeChange/{attributeTopic}
POST	/api/v1/blockEndpoint
POST	/api/v1/unblockEndpoint
GET	/api/v1/getOwnedEndpoints
GET	/api/v1/getAccessibleAttributes

Certificate

POST	/api/v1/createCertificateAndKey
POST	/api/v1/createCertificateAndKeyZip
POST	/api/v1/createCertificateFromKey
POST	/api/v1/getCaCertificate

History

POST	/api/v1/getAttributeHistoryRequest
POST	/api/v1/getAttributeHistoryByDateRequest
POST	/api/v1/getAttributeHistoryWhere
POST	/api/v1/getAttributeHistoryExpert

Logs

POST	/api/v1/getEndpointLogsRequest
POST	/api/v1/getEndpointLogsByDateRequest
POST	/api/v1/getEndpointLogsWhereRequest
POST	/api/v1/setLogsLevel
POST	/api/v1/getLogsLevel
GET	/api/v1/getLogsLevel/{endpointUUID}

Remote Jobs Execution

POST	/api/v1/executeScript

A10.0


```
{
  "@context": "https://www.w3.org/2019/wot/td/v1",
  "id": "urn:bc0f1bf8-bdae-11e9-9cb5-2a2ae2dbcce4:myHeater",
  "title": "myHeater",
  "securityDefinitions": {
    "https_sc": {
      "scheme": "basic",
      "in": "query"
    },
    "mqtts_sc": {
      "scheme": "cert"
    }
  },
  "security": [
    "https_sc",
    "mqtts_sc"
  ],
  "properties": {
    "bc0f1bf8-bdae-11e9-9cb5-2a2ae2dbcce4.myHeater.temperatures.temperature": {
      "type": "object",
      "properties": {
        "constraint": {
          "type": "string",
          "enum": [
            "Measure"
          ]
        },
        "type": {
          "type": "string",
          "enum": [
            "Number"
          ]
        },
        "timestamp": {
          "type": "number"
        },
        "value": {
          "type": "number"
        }
      },
      "required": [
        "constraint",
        "type",
        "timestamp",
        "value"
      ],
      "forms": [
        {
          "href":
          "https://localhost:8081/api/v1/getAttribute/bc0f1bf8-bdae-11e9-9cb5-2a2ae2dbcce4.myHeater.temperatures.temperature",
          "op": "readproperty",
          "contentType": "application/json"
        }
      ]
    },
    "bc0f1bf8-bdae-11e9-9cb5-2a2ae2dbcce4.myHeater.temperatures.setPointTemperature":
    {
      "type": "object",
      "properties": {
      "constraint": {
        "type": "string",
        "enum": [
          "SetPoint"
        ]
      },
      "type": {
        "type": "string",
        "enum": [
          "Number"
        ]
      },
      "timestamp": {
        "type": "number"
      },
      "value": {
        "type": "number"
      }
    },
    "required": [
      "constraint",
      "type",
      "timestamp",
      "value"
    ],
    "forms": [
      {
        "href":
        "https://localhost:8883/@set/bc0f1bf8-bdae-11e9-9cb5-2a2ae2dbcce4/myHeater/temperatures/setPointTemperature",
        "op": "writeproperty",
        "contentType": "application/json"
      },
      {
        "href":
        "https://localhost:8081/api/v1/setAttribute/bc0f1bf8-bdae-11e9-9cb5-2a2ae2dbcce4.myHeater.temperatures.setPointTemperature",
        "op": "writeproperty",
        "contentType": "application/json"
      }
    ]
  }
},
"events": {
  "bc0f1bf8-bdae-11e9-9cb5-2a2ae2dbcce4.myHeater.temperatures.temperature": {
    "data": {
      "properties": {
        "constraint": {
          "type": "string",
          "enum": [
            "Measure"
          ]
        },
        "type": {
          "type": "string",
          "enum": [
            "Number"
          ]
        },
        "timestamp": {
          "type": "number"
        },
        "value": {
          "type": "number"
        }
      },
      "constraint",
      "type",
      "timestamp",
      "value"
    ],
    "type": "object"
  },
"forms": [
```

A10.1


```
    {
      "href":
      "https://localhost:8081/api/v1/notifyAttributeChange/bc0f1bf8-bdae-11e9-9cb5
      -2a2ae2dbcce4.myHeater.temperatures.temperature/15000",
      "op": "subscribeevent",
      "subprotocol": "longpoll",
      "contentType": "application/json"
    },
    {
      "href": "
      mqtts://localhost:8883/@update/bc0f1bf8-bdae-11e9-9cb5-2a2ae2dbcce4/myHeater
      /temperatures/temperature",
      "op": "subscribeevent",
      "contentType": "application/json"
    }
  ]
},
"bc0f1bf8-bdae-11e9-9cb5-2a2ae2dbcce4.myHeater.temperatures.setPointTemperature":
{
  "data": {
    "properties": {
      "constraint": {
        "type": "string",
        "enum": [
          "SetPoint"
        ]
      },
      "type": {
        "type": "string",
        "enum": [
          "Number"
        ]
      },
      "timestamp": {
        "type": "number"
      },
      "value": {
        "type": "number"
      }
    },
    "required": [
      "constraint",
      "type",
      "timestamp",
      "value"
    ],
    "type": "object"
  },
  "forms": [
    {
      "href": "
      https://localhost:8081/api/v1/notifyAttributeChange/bc0f1bf8-bdae-11e9-9cb5-
      2a2ae2dbcce4.myHeater.temperatures.setPointTemperature/15000",
      "op": "subscribeevent",
      "subprotocol": "longpoll",
      "contentType": "application/json"
    },
    {
      "href": "
      mqtts://localhost:8883/@set/bc0f1bf8-bdae-11e9-9cb5-2a2ae2dbcce4/myHeater/te
      mperatures/setPointTemperature",
      "op": "subscribeevent",
      "contentType": "application/json"
```

A11.0

JavaScript code used to send burst of @update message with MQTT.fx

```
var Thread = Java.type("java.lang.Thread");

function execute(action) {
    out("Test Script: " + action.getName());
    for (var i = 0; i < 20000; i++) {
        sendData(i);
    }
    out("Test Script: Done");
    return action;
}

function sendData(index) {
    var milliseconds = (new Date).getTime()/1000;
    var text = "{\n\"type\": \"Number\",\n\"constraint\": \"Measure\",\n\"timestamp\": "+milliseconds+",\n
    ↪ \"value\": "+index+"\n}"

    try{
      mqttManager.publish("@update/bc0f1bf8-bdae-11e9-9cb5-2a2ae2dbcce4/demoNode/demoObject/demoMeasure",
    ↪ text);
  }
  catch(error){
    out("error while publish")
  }
}

function out(message){
     output.print(message);
}
```

A12.0

Class	Junit Test method	Description	Status
UserManagementUtilTest.kt	createDeleteUser	Create a User.	Passed
	getUserList	Users must be in User list.	Passed
	getUser	Users can be retrieved.	Passed
	modifyUserPassword	Password can be modified.	Passed
	addUserAuthority	Authority can be added.	Passed
	removeUserAuthority	Authority can be removed.	Passed
	createDeleteUser	Users can be deleted.	Passed
	randomCharacterUserTest	Non-existing Users can't be retrieved.	Passed
	randomCharacterUserTest	Can't modify password of non-existing Users.	Passed
	randomCharacterUserTest	Can't add authority do non-existing Users.	Passed
	randomCharacterUserTest	Can't remove authority do non-existing Users.	Passed
	randomCharacterUserTest	Can't delete non-existing Users.	Passed
UserAccessControlUtilTest.kt	addUserAccessRight	New permission can be added to a User.	Passed
	getUserAccessRight	Access control from an existing User can be retrieved.	Passed
	getUserAccessRight	Permission added when retrieved should be the same.	Passed
	getUserAccessRight	When permissions retrieved, non-existing permission shouldn't be retrieved.	Passed
	modifyUserAccessRight	Permission can be modified.	Passed
	modifyUserAccessRight	Modified permission should be retrievable.	Passed
	removeUserAccessRight	Permission can be removed.	Passed
	removeUserAccessRight	Removed permission cannot be retrieved.	Passed
	giveUserAccessRightOwn	User A can give Endpoint permission to User B if A own Endpoint.	Passed
	giveUserAccessRightNotOwn	User A cannot give Endpoint permission to User B if A doesn't own Endpoint.	Passed
	randomCharacterUserAccessTest	Cannot get permission from non-existing Users.	Passed
	randomCharacterUserAccessTest	Cannot add permission to non-existing Users.	Passed
	randomCharacterUserAccessTest	Cannot modify permission to non-existing Users.	Passed
	randomCharacterUserAccessTest	Cannot remove permission to non-existing Users.	Passed
	randomCharacterUserAccessTest	Cannot remove non-existing permission.	Passed
	randomCharacterUserAccessTest	Cannot give permission to non-existing Users.	Passed
UserGroupUtilTest.kt	createUserGroup	Create a UserGroup.	Passed
	getUserGroup	UserGroup can be retrieved.	Passed
	getUserGroupList	UserGroup must be in UserGroup list.	Passed
	addUserToGroup	User can be added to UserGroup.	Passed
	addUserToGroup	User must be added to UserGroup's User list.	Passed
	addUserToGroup	User must contain the added UserGroup in its UserGroup list.	Passed
	deleteUserToGroup	User can be deleted from UserGroup.	Passed
	deleteUserToGroup	User must be removed to UserGroup's User list.	Passed
	deleteUserToGroup	User must not contain the UserGroup in its UserGroup list.	Passed
	deleteUserGroup	UserGroup can be deleted.	Passed
	deleteUserGroup	Deleted UserGroup cannot be retrieved.	Passed
	deleteUserGroup	User must not contain the deleted UserGroup in its UserGroup list.	Passed
	randomCharacterUserGroupTest	Non-existing UserGroup can't be retrieved.	Passed
	randomCharacterUserGroupTest	Cannot add User to non-existing group.	Passed
	randomCharacterUserGroupTest	Cannot delete non-existing group.	Passed
	randomCharacterUserGroupTest	Cannot delete User from non-existing group.	Passed
UserGroupAccessControlUtilTest.kt	addUserGroupAccessRight	New permission can be added to a UserGroup.	Passed
	getUserGroupAccessRight	Access control from existing UserGroup can be retrieved.	Passed
	getUserGroupAccessRight	Permission added when retrieved should be the same.	Passed
	getUserGroupAccessRight	When permissions retrieved, non-existing permission shouldn't be retrieved.	Passed
	modifyUserGroupAccessRight	Permission can be modified.	Passed
	modifyUserGroupAccessRight	Modified permission should be retrievable.	Passed
	removeUserGroupAccessRight	Permission can be removed.	Passed
	removeUserGroupAccessRight	Removed permission cannot be retrieved.	Passed
	giveUserGroupAccessRightOwn	User can give Endpoint permission to UserGroup if User owns Endpoint.	Passed
	giveUserGroupAccessRightNotOwn	User cannot give Endpoint permission to UserGroup if User doesn't own Endpoint.	Passed
	randomCharacterUserGroupAccessTest	Cannot get permission from non-existing UserGroup.	Passed
	randomCharacterUserGroupAccessTest	Cannot add permission to non-existing UserGroup.	Passed
	randomCharacterUserGroupAccessTest	Cannot modify permission to non-existing UserGroup.	Passed
	randomCharacterUserGroupAccessTest	Cannot remove permission to non-existing UserGroup.	Passed
	randomCharacterUserGroupAccessTest	Cannot remove non-existing permission.	Passed
	randomCharacterUserGroupAccessTest	Cannot give permission to non-existing UserGroup.	Passed
EndpointManagementUtilTest.kt	createEndpoint	Create an Endpoint wit friendlyName.	Passed
	createEndpoint	Retrieve friendlyName is the same as input.	Passed
	createEndpoint	Retrieve UUID is a valid UUID.	Passed
	getEndpoint	Retrieve Endpoint model is the same as the one saved.	Passed
	getNode	Retrieve Node model is the same as the one saved.	Passed
	getObject	Retrieve Object model is the same as the one saved.	Passed
	getAttribute	Retrieve Attribute model is the same as the one saved.	Passed
	setAttributeSetPoint	Attribute can be set.	Passed
	setAttributeSetPoint	Endpoint model filled with influxDB value gives correct Attribute value.	Passed
	setAttributeSetPoint	Node model filled with influxDB value gives correct Attribute value.	Passed
	setAttributeSetPoint	Object model filled with influxDB value gives correct Attribute value.	Passed
	setAttributeSetPoint	Attribute model filled with influxDB value gives correct Attribute value.	Passed
	setAttributeNotSetPoint	Attribute who is not SetPoint or parameter cannot be set.	Passed
	blockEndpoint	Endpoint can be blocked.	Passed
	unblockEndpoint	Endpoint can be unblocked.	Passed
	getAccessibleAttributes	Accessible Attributes from User can be retrieved.	Passed
	getOwnedEndpoints	Owned Endpoint of User can be retrieved.	Passed
	randomCharacterEndpointTest	Non existing enpoint can't be retrieved.	Passed
	randomCharacterEndpointTest	Non-existing Nodes can't be retrieved.	Passed
	randomCharacterEndpointTest	Retrieving Node must be with a valid MQTT topic.	Passed
	randomCharacterEndpointTest	Non-existing Objects can't be retrieved.	Passed
	randomCharacterEndpointTest	Retrieving Objects must be with a valid MQTT topic.	Passed
	randomCharacterEndpointTest	Non-existing Attributes can't be retrieved.	Passed
	randomCharacterEndpointTest	Retrieving Attributes must be with a valid MQTT topic.	Passed
	randomCharacterEndpointTest	Non-existing Attributes can't be set.	Passed
	randomCharacterEndpointTest	Setting Attributes must be with a valid MQTT topic.	Passed
	randomCharacterEndpointTest	Non existing enpoint can't be blocked.	Passed
	randomCharacterEndpointTest	Non existing enpoint can't be unblocked.	Passed
	randomCharacterEndpointTest	Cannot retrieve accessible Attributes from non-existing Users.	Passed
	randomCharacterEndpointTest	Cannot retrieve owned Endpoints from non-existing Users.	Passed
CertificateUtilTest.kt	getCaCertificate	Get Ca certificate.	Passed
	createCertificateAndKey	Create pem certificate and pem key.	Passed
	createCertificateAndKey	Certificate is valid.	Passed
	createCertificateAndKey	Serial from certificate is UUID of Endpoint.	Passed
	createCertificateAndKey	Begining validity date is "now".	Passed
	createCertificateAndKey	Ending validity date is in 100 years.	Passed

	createCertificateAndKey	Certificate issuer is ca certificate subject.	Passed
	createCertificateFromKey	Create pem certificate from key.	Passed
	createCertificateFromKey	Certificate is valid.	Passed
	createCertificateFromKey	Serial from certificate is UUID of Endpoint.	Passed
	createCertificateFromKey	Begining validity date is "now".	Passed
	createCertificateFromKey	Ending validity date is in 100 years.	Passed
	createCertificateFromKey	Certificate issuer is ca certificate subject.	Passed
	createCertificateAndKeyZip	Create zip file containing certificate in p12, ca in jks, and .properties.	Passed
	createCertificateAndKeyZip	Return valid output path of zip.	Passed
	createCertificateAndKeyZip	Zip contains jks.	Passed
	createCertificateAndKeyZip	Zip contains p12.	Passed
	createCertificateAndKeyZip	Zip contain .properties.	Passed
HistoryUtilTest.kt	getAttributeHistoryRequest	User can get simple history.	Passed
	getAttributeHistoryRequest	Retrieved data has the correct name.	Passed
	getAttributeHistoryRequest	Retrieved the correct amount of data from history.	Passed
	getAttributeHistoryByDateRequest	User can get history between date.	Passed
	getAttributeHistoryByDateRequest	Retrieved data has the correct name.	Passed
	getAttributeHistoryByDateRequest	Retreived the correct amount of data from history.	Passed
	getAttributeHistoryWhereRequest	User can get history with custom "where".	Passed
	getAttributeHistoryWhereRequest	Retrieved data has the correct name.	Passed
	getAttributeHistoryWhereRequest	Retrieved the correct amount of data from history.	Passed
	getAttributeHistoryExpert	User can get history with "expert" setting.	Passed
	getAttributeHistoryExpert	Retrieved data has the correct name.	Passed
	getAttributeHistoryExpert	Retreived the correct amount of data from history.	Passed
	basicSqlInjection	getAttributeHistoryByDateRequest is protected again sql injection.	Passed
	basicSqlInjection	getAttributeHistoryWhereRequest is protected again sql injection.	Passed
	basicSqlInjection	getAttributeHistoryExpert is protected again sql injection.	Passed
LogsUtilTest.kt	getEndpointLogsRequest	User can get simple log history.	Passed
	getEndpointLogsRequest	Retrieved data has the correct name.	Passed
	getEndpointLogsRequest	Retrieved the correct amount of data from history.	Passed
	getEndpointLogsByDateRequest	User can get log history between date.	Passed
	getEndpointLogsByDateRequest	Retrieved data has the correct name.	Passed
	getEndpointLogsByDateRequest	Retrieved the correct amount of data from history.	Passed
	getEndpointLogsWhereRequest	User can get log history with custom "where".	Passed
	getEndpointLogsWhereRequest	Retrieved data has the correct name.	Passed
	getEndpointLogsWhereRequest	Retreived the correct amount of data from history.	Passed
	basicSqlInjection	getEndpointLogsByDateRequest is protected again sql injection.	Passed
	basicSqlInjection	getEndpointLogsWhereRequestt is protected again sql injection.	Passed
	getLogsLevel	Logs level of an Endpoint can be retrieved.	Passed
	setLogsLevel	Logs level of an Endpoint can be changed.	Passed
	setLogsLevel	Set log level is correct.	Passed
JobsUtilTest.kt	executeJob	Remote execute job can be executed.	Passed
	executeJob	Remote execute job is sent by MQTT.	Passed
	executeJob	Received JobParameter message parameters are equal to sent message.	Passed

Number of unit test:	55
Number of use case in unit test:	140

cloudio-endpointHeater-demo

Description

This is an implementation of a smart heating system with the cloud.iO java librarie (using 2.0 snapshot). A naive implementation of a heater trying to regulate the ambient temperature following a set point.

This software is the cloud.iO MQTT Endpoint interface. It is needed to have Endpoint certificates to use this software and to have an instance of the cloud.iO core running. This software has for goal to be used with the cloudio-heater-demo GUI software, but it is not mandatory.

Requirement

As explained earlier, you'll need cloud.iO Endpoint certificates, those are available through cloud.iO RESTful API, contact your cloud.iO administrator if you don't know how to retrieve them.

You'll have to have the following files in the resources/cloud.io directory accoarding to your own "EndpointUUID":

- "EndpointUUID".properties --> can follow the given example
- "EndpointUUID".p12
- ca-cert.jks

You'll also have to modify the Heater.java at line 14 and insert your endpointUUID.

cloud.iO Endpoint structure

The model (following cloud.iO Endpoint->Node->Object->Attribute template) of the heater is the following:

- "enpointUUID".myHeater.temperatures.temperature
  - Represent the ambiant temperature, will try to follow the set point
- "enpointUUID".myHeater.temperatures.setPointTemperature
  - Represent the set point temperature

Those two Attributes can be access via grafana, the RESTFul API or with mqtt subscribes:

- @update/"enpointUUID"/myHeater/temperatures/temperature
- @set/"enpointUUID"/myHeater/temperatures/setPointTemperature

cloudio-heater-demo

Description

This software is a demonstration of a smart heating system using cloud.iO. This software represents the two actors of cloud.iO, the Endpoint and the user, and the cloud part with the two databases of cloud.iO: MongoDB and influxDB.

This software is using the cloud.iO RESTful API. It is needed to have a cloud.iO username and password to use this software and to have an instance of the cloud.iO core running. You may also need the demo java implementation fo the heater available in this cloudio-EndpointHeater-demo repository.

Requirement

This software run with python 3, tested for 3.6 and above. The following packages are needed:

- requests
- tkinter
- pygubu

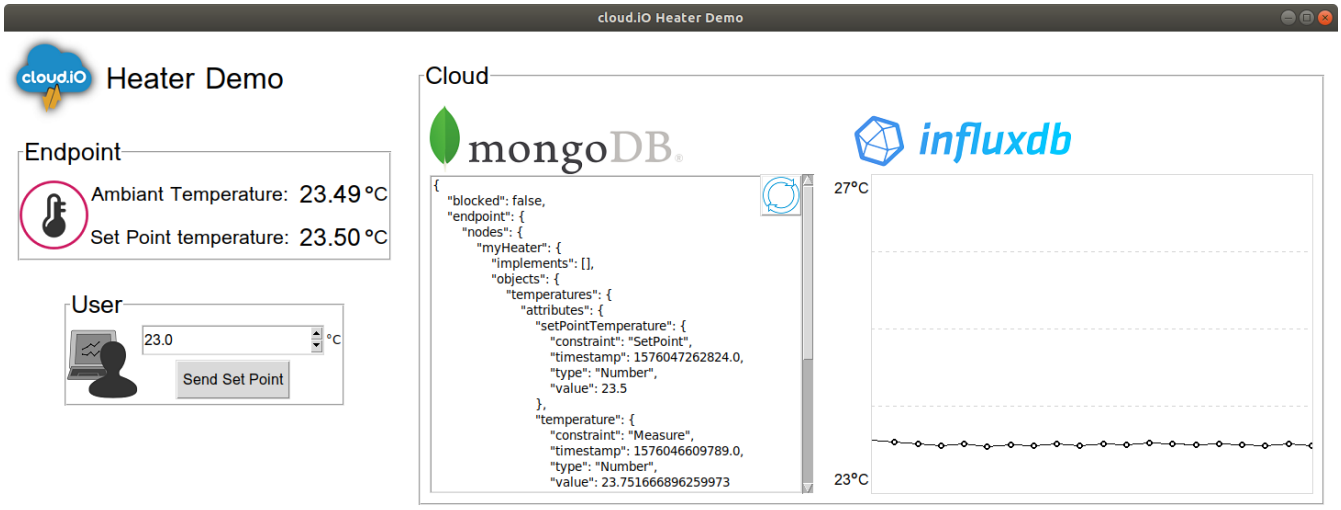
Launch

```
python heaterDemo.py
```

GUI description

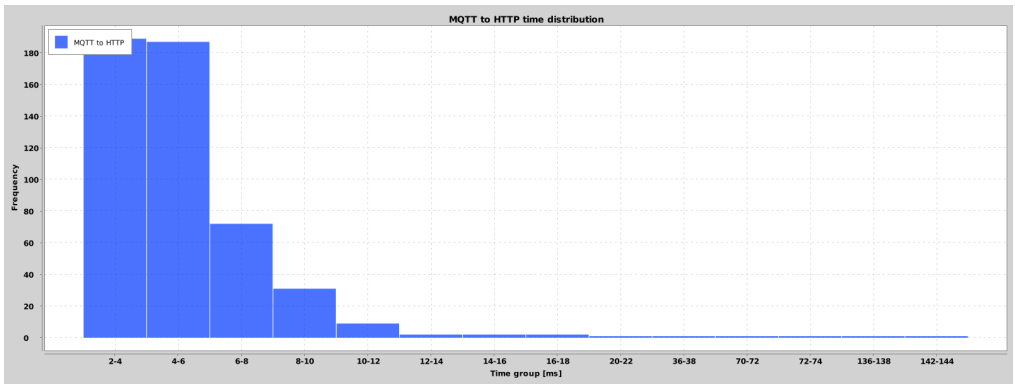
The upper left part represents the actual heater (Endpoint part). It shows the ambient temperature, and the set point temperature, both refreshed every second. The lower left part represents an interaction from the user to the Endpoint: setting the set point temperature. With the spinner, it is possible to change the temperature and with the button under it, to send the temperature to the Endpoint for it to regulate.

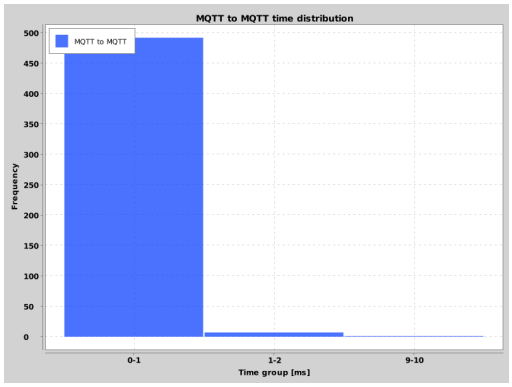
The right part represents the data stored in the cloud. On the MongoDB part, we can see the retrieved digital twin on the heater following the cloud.iO model; this model can be updated with the upper right button. On the influxDB part, we can see the temperature evolving through time.



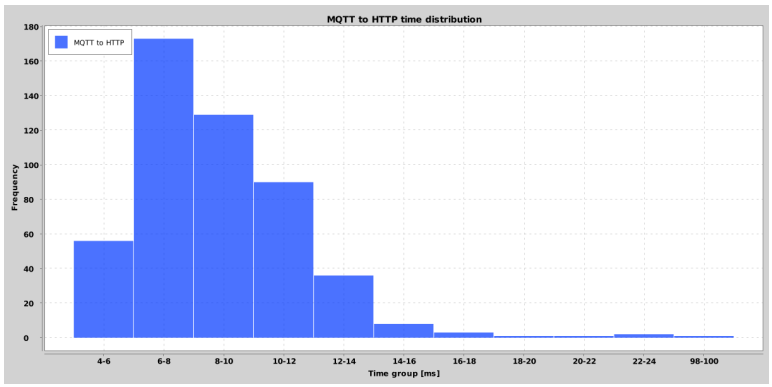
A15.0

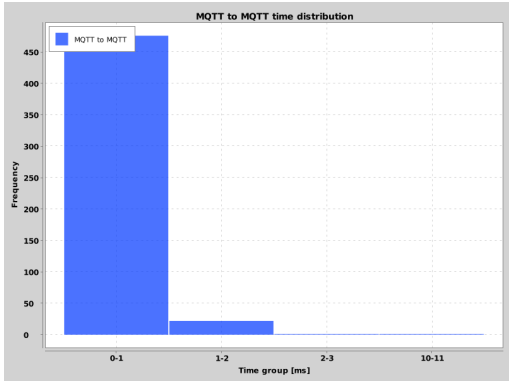
Performance analysis, Virtual Machine[1]





Performance analysis, Local Network





[1]The interval between two bars is not constant for high value of time to reduce the size of the graph

A15.1

Performance analysis, Non Local Network