
dojot Documentation

Release 0.0.0

Matheus Magalhaes

Feb 21, 2018

Contents:

1	Architecture	3
1.1	Components	4
1.2	Infrastructure	8
1.3	Communications	8
1.4	Deployment strategies	8
1.5	Comparative analysis	8
2	Operations Guide	9
2.1	Deployment	9
2.2	Device Management	9
2.3	User Management	10
2.4	Resources Management	10
2.5	System Dashboard	10
3	User Guide	11
3.1	Who should read this	11
3.2	Getting Started	12
3.3	dojot basics	12
3.4	Integrating physical devices	18
3.5	Flow Management	18
4	Components and APIs	19
4.1	Components	19
4.2	Exposed APIs	20
5	Installation Guide - Docker compose	21
5.1	Dependencies	21
5.2	Installation	22
5.3	Configuration	23
5.4	Usage	23
6	Installation Guide - Google Cloud Platform	25
6.1	Creating a Project	25
6.2	Creating a Cluster	26
6.3	Getting the credentials	26
7	Installation Guide - Kubernetes	27

7.1	Dependencies	27
7.2	Deployment	28
8	Running dojoy on VirtualBox	31
9	Frequently Asked Questions	39
9.1	General	40
9.2	Usage	41
9.3	Devices	42
9.4	Data Flows	44
9.5	Applications	46
10	Mutual Authentication	47
10.1	Using Mutual Authentication	48
10.2	Application Registration	48
10.3	Authentication	48
10.4	Accessing <i>dojoy</i> APIs	50
11	Crypto Service	51
11.1	REST APIs	51
11.2	Usage Examples	52
12	MQTT-TLS Tutorial	55
12.1	tl;dr	56
12.2	Components	56
12.3	Mosquitto configuration files	58
12.4	Certificate retriever	58
12.5	Important Notes	59

This is the high-level documentation for dojot IoT platform developed by CPqD. This platform is largely based on [FIWARE](fiware.org), and aims to provide the application and device developers with a more concise and integrated interaction, while benefiting for the highly customizable and efficient infrastructure provided by FIWARE.

While based on FIWARE, this platform actually has a large set of components of its own, and interaction between components was modified to allow better packaging and performance for the solution as a whole.

While this does provide an overall glimpse of the platform, this documentation is not suited for middleware developers that might want to better understand the components that compose the solution themselves. For that, please check the component's own documentation repositories and ReadTheDocs pages.

This document describes the current architecture that guides the platform implementation, detailing the components that comprise the solution, as well as their functionalities and how each of them contribute to the platform as a whole.

While a brief explanation of each component is provided, this high level description does not explain (or aims to explain) the minutia of each component's implementation. For that, please refer to each component's own documentation.

Table of Contents

- *Components*
 - *Kafka + Subscription Manager + NGSI*
 - *Device Manager*
 - *Iot-agent*
 - *User Authorization Service*
 - *Service Orchestrator*
 - *Fiware Perseo*
 - *History (Logstash)*
 - *Logging and Auditing Service*
 - *Kong API Gateway*
 - *Management Application*
 - *Elastic Service Controller*
- *Infrastructure*
- *Communications*
- *Deployment strategies*

- *Comparative analysis*

1.1 Components

With the idea of utilizing open-source and consolidated components, as a starting point for the *dojot* IoT middleware we adopted the european project Fiware (FIWARE, 2016). The solutions developed from this framework followed a micro-services architecture, having as the main component a context broker that is responsible for redistributing events among services that are part of the middleware.

The first architecture proposal took into account a basic group of Fiware services together with new services, developed in the scope of this project, having as the main purpose, increasing the usability of the *dojot* platform. This initial architecture can be seen on Fig. 1.1.

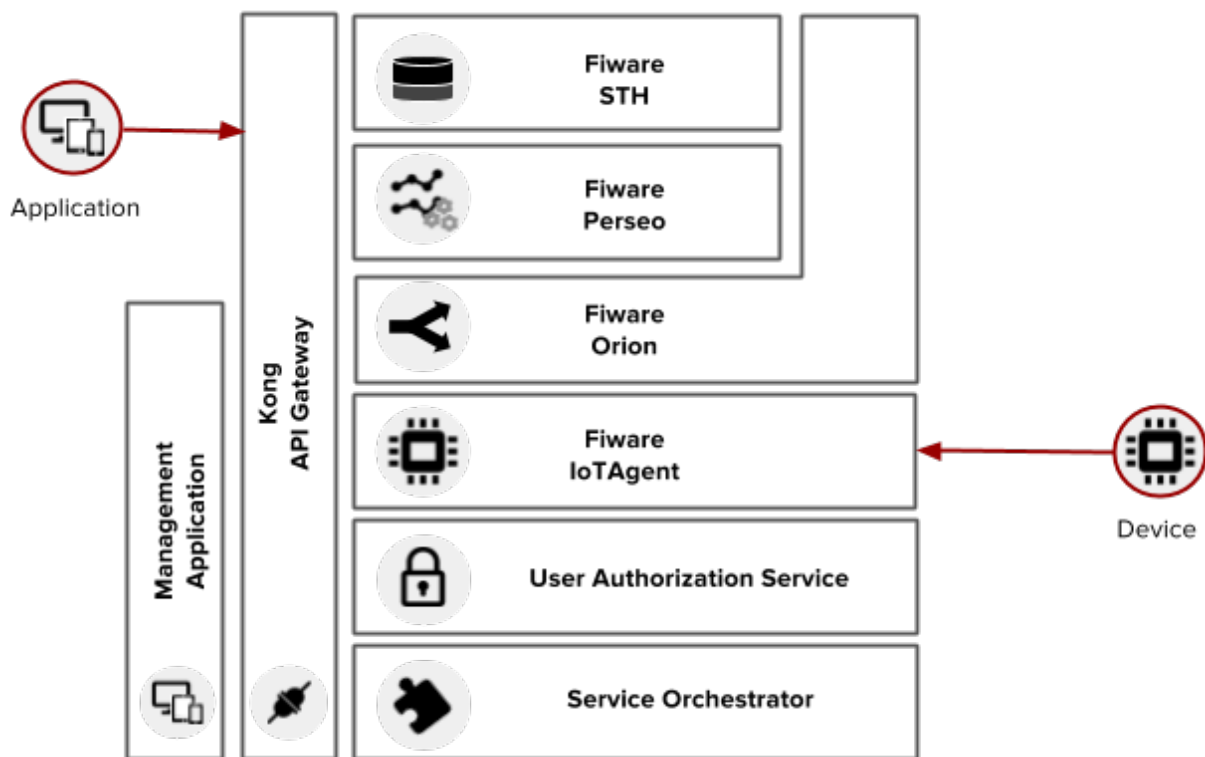


Fig. 1.1: Initial Architecture

In this proposal we utilize the following components from Fiware:

- **Orion**: a context broker used as the communication bus for all the internal services of the middleware
- **STH**: the history service used for storing IoT devices data in a MongoDB database
- **Perseo CEP**: the service that is responsible for treating events in real time
- **Iot-agent**: service used as an abstraction layer for integrating MQTT and HTTP devices

To this services we added the Kong API Gateway to act as a centralized point-of-access removing the need of direct communication with each one of the services, an orchestration service to abstract the middleware configuration, an authentication service to validate user access credentials and also an user application with graphical interface with the purpose of managing the middleware (users, devices and data flows management).

Considering this architecture the basic utilization flow is as follows: The user configures IoT devices through the GUI or directly using the REST APIs provided by the API Gateway, he also configures processing flows to deal with the data generated by the configured devices. As an example we can generate notifications when the data of a device has one of its values reaching a threshold or we can add an entry to a database when a device enters or leaves an specific geographic location. This user operations on the API result in configurations across the internal services of the middleware (Broker, CEP and iot-agent), being partially abstracted by the orchestration service.

The user contexts are isolated and there is no data sharing, the access credentials are validated by the authorization service for each and every operation (API Request). Once the devices and the flows are configured, the iot-agent is capable of mapping the data received from devices, encapsulated on MQTT for example, and send then to the context broker for internal distribution, reaching, for instance, the history service so it can persist the data on a database and the CEP for processing it based on rules. If certain conditions are matched when rules are being processed, a new event is generated and sent to the broker service to be redistributed to the interested services.

This architecture made possible the validation of ideas and limitations and possible improvements were identified, converging to a reviewed architectural proposition as described on [Fig. 1.2](#). This new proposal is under development and should become part of the solution.

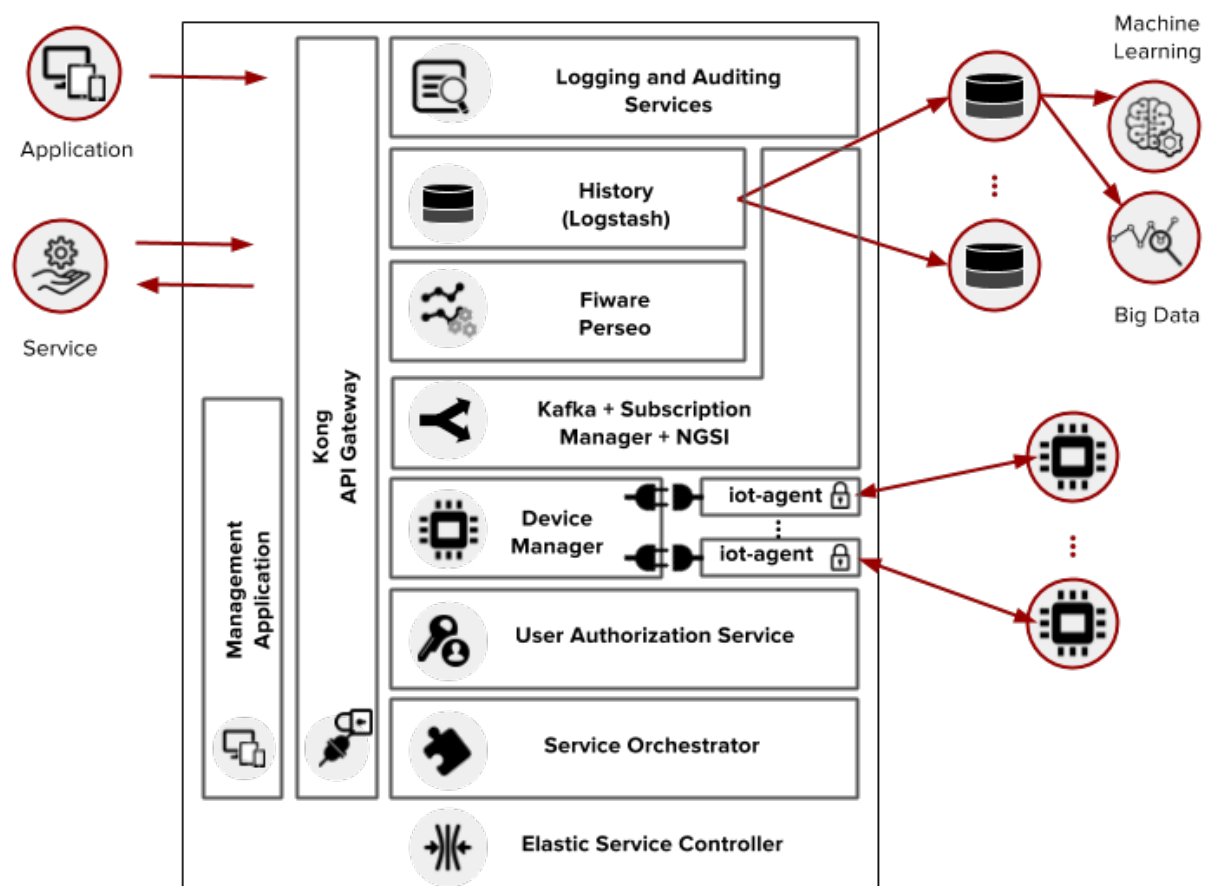


Fig. 1.2: Revised Architecture

More detailed and updated information can be found on the [dojot Github repository](#).

Each one of the components that are part of the architecture are briefly described on the sub-sections below.

1.1.1 Kafka + Subscription Manager + NGSI

Apache Kafka is a distributed messaging platform that can be used by applications which need to stream data or consume/produce data pipelines. In contrast to what Orion is, Kafka seems to be more appropriate to fulfil *dojot*'s architectural requirements (responsibility isolation, simplicity, and so on). And using it with a subscription manager and a NGSI interface translator, we can compose an entity which is very close to the features offered by Orion, in addition to improved speed and easier scalability.

In Kafka, a specialized topics structure is used to insure isolation between different users and applications data, enabling a multi-tenant infrastructure.

The subscription manager service makes use of an in-memory database for efficiency. It adds context to Apache Kafka, making it possible that internal or even external services are able to subscribe or query data based on context. The subscription manager is also a distributed service to avoid it being a single point of failure or even a bottleneck for the architecture.

To keep a certain level of compatibility with Fiware services, meaning, for using Fiware services and components in the *dojot* platform with the minimal amount of adaptations, we added a NGSI interface translation service.

1.1.2 Device Manager

The IoT Device manager is a core entity, responsible for maintaining the devices data models and its abstractions, it is also responsible for propagating this models to services that are interested in this kind of information, for example, the *iot-agent*.

This service is stateless, having its data persisted to a database, with data isolation for users and applications, making possible a multi-tenant architecture for the middleware.

1.1.3 *iot-agent*

The *iot-agent* is an adaptation service between the data model of the context broker and the devices data models. The *dojot* platform can have multiple *iot-agents*, each one of them being specialized in a specific protocol like, for instance, MQTT/JSON, CoAP/LWM2M and HTTP/JSON.

Security measures like the management of the secure channel used for the communication of the platform with the device is also treated by this service.

1.1.4 User Authorization Service

This service is responsible for managing user profiles and access control. Basically any API call that reaches the platform via the API Gateway is validated by this service.

To be able to deal with a high volume of authorization calls, it uses caching, it is stateless and it is scalable horizontally. Its data is stored on a database.

1.1.5 Service Orchestrator

This service provides a high level API for configuring the *dojot* with the objective of reducing the need of knowing how to handle each one of the services that are part of the platform. More specifically, it is responsible for modeling different services, exposing a simplified configuration interface and propagating this configuration to the various services when requested. It acts as a centralized configuration manager for multiple services.

1.1.6 Fiware Perseo

The CEP service is responsible for analysing in real time the data processing flows for selected events and triggering actions when specific conditions or thresholds are reached. This component is used for creating notification events from the pure data that is incoming from the IoT devices. It is integrated with the platform through the context broker and its configuration is abstracted by the service orchestrator.

1.1.7 History (Logstash)

The Logstash connects to the context broker and works as a pipeline for data and events that must be persisted on a database. The data is converted into an storage structure and is sent to the corresponding database.

For internal storage, the MongoDB non-relational database is being used, it allows a Sharded Cluster configuration that may be required according to the use case.

The data may also be directed to databases that are external do the *dojot* platform, requiring only a proper configuration of Logstash and the data model to be used.

1.1.8 Logging and Auditing Service

All the services that are part of the *dojot* platform generate usage metrics of its resources and make then available to the logging and auditing service, which process this registers and summarize then based on users and applications.

The consolidated data is presented back to the services, allowing then, for example, to expose this data to the user via a graphical interface, to limit the usage of the system based on resource consumption and quotas associated with users or even to be used by billing services to charge users for the utilization of the platform.

1.1.9 Kong API Gateway

The Kong API Gateways is used as the entry point for applications and external services to reach the services that are internal to the *dojot* platform, resulting in multiple advantages like, for instance, single access point and ease when applying rules over the API calls like traffic rate limitation and access control.

1.1.10 Management Application

Web Application responsible for providing responsive interfaces to manage the *dojot* platform, including functionalities like:

- **User Profile Management:** define profiles and the API permission associated to those profiles
- **User Management:** Creation, Visualization, Edition and Deletion Operations
- **Applications Management:** Creation, Visualization, Edition and Deletion Operations
- **Device Models Management:** Creation, Visualization, Edition and Deletion Operations
- **Devices Management:** Creation, Visualization (real time data), Edition and Deletion Operations
- **Processing Flows Management:** Creation, Visualization, Edition and Deletion Operations

1.1.11 Elastic Service Controller

This is a service specialized for cloud environments, that is capable of monitoring the utilization of the platform, being able to increase or decrease its storage and processing capacity in an dynamic and automatic fashion to adapt to the variability on the demand.

This controller depends that the dojot platform services are horizontally scalable, as well as the databases must be clusterizable, which match with the adopted architecture.

1.2 Infrastructure

TODO: This section should describe the components that are used as ready-made pieces of working software that compose the solution, but have no implementation specific to the project. Relevant topics that might be discussed here are:

- The API gateway
- Storage components (mongo, redis, HDFS, CEPH, etc.)
- Processing libraries and environments (Spark, Flink, Storm, kafka-streaming, map-reduce, etc.)
- Broker components (rabbitMQ, mosquitto, kafka, verneMQ, emqtt, etc.)

1.3 Communications

TODO: This section should provide the reader with the communication strategy used to bind together the components that comprise the solution, as well as the interfaces (protocols, serialization formats) available to the applications and devices developers.

1.4 Deployment strategies

TODO: This section should list the deployment requirements and implementation decisions made to satisfy those requirements. “Why orchestrator platform ‘x’?”, “How can this be deployed on commercial cloud environments?”, “How can this be deployed on stand-alone environments?” are all questions that should be answered here.

1.5 Comparative analysis

TODO: This section should detail the features that differentiate the platform from a “stock” deployment of fiware, as well as a feature summary comparing the proposed solution with a reduced set of third-party implementations of IoT platforms available.

This document provides information on how to properly deploy and manage an instance of dojot. For documentation regarding the usage of the platform from the perspective of either an application, or device developer please refer to the [user guide]().

Table of Contents

- *Deployment*
- *Device Management*
- *User Management*
- *Resources Management*
- *System Dashboard*

2.1 Deployment

TODO: This section should describe the steps required to deploy the solution on all “homologated” environments (e.g. standalone, aws, google cloud, bluemix, etc.). For each environment, there’ll also be a link pointing to the environment-specific section of the *Resources Management* sub-section that describes how cloud resources are managed (allocated, released and pertinent configuration).

2.2 Device Management

TODO: This section should describe the steps required to configure a new device on the platform. While this information will also be presented on the user guide, here the idea is to give more focus to the specific infrastructure that has to be managed in order to guarantee the device’s authenticity and communication.

2.3 User Management

TODO: This section should describe the steps required to configure user roles and role permissions for the platform itself, as well as handle application authentication features.

2.4 Resources Management

TODO: For each “homologated” deployment scenario, this section should describe how the deployment is done, as well as which parameters are available for each of them.

2.5 System Dashboard

TODO: This should be a brief description of the system dashboard that is made available for the system administrator to check the system’s overall status and alerts.

This document provides information on how to use doJot. On that regard, this should describe the steps required to install and operate the platform from a device developer or application developer point of view. For documentation regarding the operation of the platform itself, please refer to the *Operations Guide*.

Table of Contents

- *Who should read this*
- *Getting Started*
- *doJot basics*
 - *User authentication*
 - *Devices and templates*
 - *Flows*
 - *Step-by-step device management*
 - * *Getting access token*
 - * *Device creation*
 - * *Sending messages*
 - * *Checking historical data*
- *Integrating physical devices*
- *Flow Management*

3.1 Who should read this

- Users that want a deeper look at how doJot works;

- Application developers.

3.2 Getting Started

To start, please follow dojot's installation guide in [Installation Guide - Docker compose](#). There you should find how to properly download a working copy of the components, how to minimally configure them, how to start them up and how to check whether they are working.

3.3 dojot basics

Before using dojot, you should be familiar with some basic operations and concepts. They are very simple to understand and use, but without them, all operations might become obscure and senseless. It is advisable to checkout our [Architecture](#) to get acquainted with all internal components.

First of all, you should check out how to access dojot through its APIs, which is detailed in the next section. After that, there's an explanation of a few basic entities in dojot: devices, templates and flows (including a simple tutorial on how to create and use them).

All these instructions considers only API access. For a guided tour on how to use the web interface, check dojot's [YouTube channel](#).

3.3.1 User authentication

All HTTP requests supported by dojot are sent to the API gateway. In order to control which user should access which endpoints and resources, dojot makes uses of [JSON Web Token](#) (a useful tool is [jwt.io](#)) which encodes things like (not limited to these):

- User identity
- Validation data
- Timestamp

The component responsible for user authentication is [auth](#). You can find a tutorial of how to authenticate a user and how to get an access token in [auth documentation](#).

3.3.2 Devices and templates

In dojot, a device is a digital representation of an actual device or gateway with one or more sensors or of a virtual one with sensors/attributes inferred from other devices. Throughout the documentation, this kind of device will be called simply as 'device'. If the actual device must be referenced, we'll be calling it as 'physical device'.

Consider, for instance, a physical device with temperature and humidity sensors; it can be represented in dojot as a device with two attributes (one for each sensor). We call this kind of device as regular device or by its communication protocol, for instance, MQTT device or CoAP device.

We can also create devices which don't directly correspond to their physical counterparts, for instance, we can create one with higher level of information of temperature (is becoming hotter or is becoming colder) whose values are inferred from temperature sensors of other devices. This kind of device is called virtual device.

All devices are created based on a *template*, which can be thought as a model of a device. As "model" we could think of part numbers or product models - one *prototype* from which devices are created. Templates in dojot have one label (any alphanumeric sequence), a list of attributes which will hold all the device emitted information, and optionally a

few special attributes which will indicate how the device communicates, including transmission methods (protocol, ports, etc.) and message formats.

In fact, templates can represent not only “device models”, but it can also abstract a “class of devices”. For instance, we could have one template to represent all thermometers that will be used in dojot. This template would have only one attribute called, let’s say, “temperature”. While creating the device, the user would select its “physical template”, let’s say *TexasInstr882*, and the ‘thermometer’ template. The user would have also to add translation instructions in order to map the temperature reading that will be sent from the device to a “temperature” attribute.

In order to create a device, a user selects which templates are going to compose this new device. All their attributes are merged together and associated to it - they are tightly linked to the original template so that any template update will reflect all associated devices.

The component responsible for managing devices (both real and virtual) and templates is [DeviceManager](#). [DeviceManager documentation](#) explains in more depth all the available operations.

3.3.3 Flows

This section will explain what a flow is and how to use it. It will be filled as soon as [mashup](#) documentation is ready.

3.3.4 Step-by-step device management

This section provides a complete step-by-step tutorial of how to create, update, send messages to and check historical data of a device. We will create a simple device with only one attribute, send a few messages emulating the physical device and check the historical data for the only attribute this device has.

Also, this tutorial assumes that you are using [docker-compose](#), which has all the necessary components to properly run dojot (so all API requests will be sent to localhost:8000).

Getting access token

As said in [User authentication](#), all requests must contain a valid access token. You can generate a new token by sending the following request:

```
curl -X POST http://localhost:8000/auth \
  -H 'Content-Type:application/json' \
  -d '{"username": "admin", "passwd" : "admin"}'

{"jwt": "eyJ0eXAiOiJKV1QiL..."}
```

If you want to generate a token for other user, just change the username and password in the request payload. The token (“eyJ0eXAiOiJKV1QiL...””) should be used in every HTTP request sent to dojot in a special header. Such request would look like:

```
curl -X GET http://localhost:8000/device \
  -H "Authorization: Bearer eyJ0eXAiOiJKV1QiL..."
```

Remember that the token must be set in the request header as a whole, not parts of it. In the example only the first characters are shown for the sake of simplicity. All further requests will use a bash variable called `bash ${JWT}`, which contains the token got from auth component.

Device creation

In order to properly configure a physical device in dojot, you must first create its representation in the platform. The example presented here is just a small part of what is offered by DeviceManager. For more information, check the [DeviceManager how-to](#) for more detailed instructions.

First of all, let's create a template for the device - all devices are based off of a template, remember.

```
curl -X POST http://localhost:8000/template \  
-H "Authorization: Bearer ${JWT}" \  
-H 'Content-Type:application/json' \  
-d '{  
  "label": "Thermometer Template",  
  "attrs": [  
    {  
      "label": "temperature",  
      "type": "dynamic",  
      "value_type": "float"  
    }  
  ]  
'
```

This request should give back this message:

```
1 {  
2   "result": "ok",  
3   "template": {  
4     "created": "2018-01-25T12:30:42.164695+00:00",  
5     "data_attrs": [  
6       {  
7         "template_id": "1",  
8         "created": "2018-01-25T12:30:42.167126+00:00",  
9         "label": "temperature",  
10        "value_type": "float",  
11        "type": "dynamic",  
12        "id": 1  
13      }  
14    ],  
15    "label": "Thermometer Template",  
16    "config_attrs": [],  
17    "attrs": [  
18      {  
19        "template_id": "1",  
20        "created": "2018-01-25T12:30:42.167126+00:00",  
21        "label": "temperature",  
22        "value_type": "float",  
23        "type": "dynamic",  
24        "id": 1  
25      }  
26    ],  
27    "id": 1  
28  }  
29 }
```

Note that the template ID is 1 (line 27).

To create a template based on it, send the following request to dojot:

```

1 curl -X POST http://localhost:8000/device \
2 -H "Authorization: Bearer ${JWT}" \
3 -H 'Content-Type:application/json' \
4 -d ' {
5     "templates": [
6         "1"
7     ],
8     "label": "device"
9 } '

```

The template ID list on line 6 contains the only template ID configured so far. To check out the configured device, just send a GET request to /device:

```
curl -X GET http://localhost:8000/device -H "Authorization: Bearer ${JWT}"
```

Which should give back:

```

{
  "pagination": {
    "has_next": false,
    "next_page": null,
    "total": 1,
    "page": 1
  },
  "devices": [
    {
      "templates": [
        1
      ],
      "created": "2018-01-25T12:36:29.353958+00:00",
      "attrs": {
        "1": [
          {
            "template_id": "1",
            "created": "2018-01-25T12:30:42.167126+00:00",
            "label": "temperature",
            "value_type": "float",
            "type": "dynamic",
            "id": 1
          }
        ]
      },
      "id": "0998",
      "label": "device_0"
    }
  ]
}

```

Sending messages

So far we got an access token and created a template and a device based on it. In an actual deployment, the physical device would send messages to dojot with all its attributes and their current values. For this tutorial we will send MQTT messages by hand to the platform, emulating such physical device. For that, we will use `mosquitto_pub` from Mosquitto project.

Attention: Some Linux distributions, Ubuntu in particular, have two packages for [mosquitto](#) - one containing tools to access it (i.e. `mosquitto_pub` and `mosquitto_sub` for publishing messages and subscribing to topics) and another one containing the MQTT broker. In this tutorial, only the tools are going to be used. Please check if MQTT broker is not running before starting dojot (by running commands like `ps aux | grep mosquitto`).

The dojot compatible format for messages sent by devices is a simple key-value JSON, such as:

```
{
  "temperature" : 10.6
}
```

Let's send this message to dojot:

```
mosquitto_pub -t /admin/0998/attrs -m '{"temperature": 10.6}'
```

If there is no output, the message was sent to MQTT broker. The topic is build from the following information:

- admin: user tenant. This is retrieved from “service” attribute from user configuration.
- 0998: device ID. This is retrieved from the device itself. It is returned when the device is created or read from /device endpoint.

To check if it was correctly processed by dojot, send the following request:

```
curl -X POST http://localhost:8000/metric/v2/entities/0998 \
-H "Authorization: Bearer ${JWT}" \
-H "Fiware-Service: admin" \
-H "Fiware-ServicePath:/"
```

This would result in the following message:

```
{
  "id": "0998",
  "type": "template_1",
  "temperature": {
    "type": "Number",
    "value": 10.6,
    "metadata": {}
  }
}
```

Note: The device type is a string formed by “template_” concatenated with all template IDs that form it.

For more information on how dojot deals with data sent from devices, check the [Integrating physical devices](#) section.

Checking historical data

In order to check all values that were sent from a device for a particular attribute, you could use the [history APIs](#). Let's first send a few other values to dojot so we can get a few more interesting results:

```
mosquitto_pub -t /admin/0998/attrs -m '{"temperature": 10.6}'
mosquitto_pub -t /admin/0998/attrs -m '{"temperature": 15.6}'
mosquitto_pub -t /admin/0998/attrs -m '{"temperature": 36.5}'
```

To retrieve all values sent for temperature attribute of this device:

```
curl -X GET http://localhost:8000/history/STH/v1/contextEntities/type/template_1/id/
↪0998/attributes/temperature?lastN=3 \
-H "Authorization: Bearer ${JWT}" \
-H "Fiware-Service:admin" \
-H "Fiware-ServicePath:/"
```

The history endpoint is built from these values:

- .../type/template_1/id/0998/...: the device type is template_1 - this is retrieved from the type attribute from the device. Same for the ID (0998)
- .../attributes/temperature?lastN=3: the requested attribute is temperature and it should get the last 3 values. More operators are available in [STH data retrieval](#)

The request should result in the following message:

```
{
  "contextResponses": [
    {
      "contextElement": {
        "attributes": [
          {
            "name": "temperature",
            "values": [
              {
                "recvTime": "2018-01-25T14:57:21.027Z",
                "attrType": "Number",
                "attrValue": 10.6
              },
              {
                "recvTime": "2018-01-25T14:57:21.063Z",
                "attrType": "Number",
                "attrValue": 15.6
              },
              {
                "recvTime": "2018-01-25T14:57:21.701Z",
                "attrType": "Number",
                "attrValue": 36.5
              }
            ]
          }
        ]
      },
      "id": "0998",
      "isPattern": false,
      "type": "template_1"
    },
    "statusCode": {
      "code": "200",
      "reasonPhrase": "OK"
    }
  ]
}
```

This message contains all previously sent values. More information about what can be done with historical data can be found in [STH documentation](#).

3.4 Integrating physical devices

This section should detail how to integrate a new device with the system. That should encompass the both the communication requirements imposed on the device in order to allow its usage with the platform, as well as the steps (if any, depending on the protocol used) to configure this new device within the platform.

This could also explain (if indeed implemented) the device management functionalities made available by the platform to the device developer.

Regarding the requirements imposed on the devices, it is foreseen that, for each communication scheme (protocol/serialization format) officially supported by the platform, a step by step guide on how to “develop” a device is supplied. Such guide can, if applicable, make use of a platform-provided library or SDK.

3.5 Flow Management

Moving to the perspective of an application developer, this section should list and explain the usage of the information flow configuration process within the platform - how to use the provided gui, high level description of the APIs that can be used to configure such flows, available actions to be used when building the flows, so on and so forth.

Components and APIs

4.1 Components

Table 4.1: Components

Component	GitHub repository	Documentation
History (STH)	GitHub - STH	readthedocs - STH
mongodb		mongodb documentation
Mosquitto (MQTT broker)		Mosquitto documentation
GUI	GitHub - GUI	
iotagent-json	GitHub - iotagent-json	readthedocs - iotagent-json
Orion	GitHub - Orion	readthedocs - Orion
DeviceManager	GitHub - DeviceManager	readthedocs - DeviceManager
auth	GitHub - auth	
postgres		postgres documentation
Kong API gateway		Kong documentation
Perseo Core	GitHub - perseo-core	Docs - perseo-core
Perseo Front-End	GitHub - perseo-fe	Docs - perseo-fe
Orchestrator/Mashup	GitHub - mashup	
redis		Redis documentation
zookeeper		Zookeeper documentation
Kafka		Kafka documentation
EJBCA-REST	GitHub - EJBCA-REST	

4.2 Exposed APIs

Table 4.2: APIs

Endpoint	Purpose	Component API	Repository
/device	Device management	API - DeviceManager	GitHub - DeviceManager
/template	Template management	API - DeviceManager	GitHub - DeviceManager
/flows	Flow management		GitHub - mashup
/auth	User authentication	API - auth	GitHub - auth
/auth/revoke	User authentication	API - auth	GitHub - auth
/auth/user	User authentication	API - auth	GitHub - auth
/history	Device historical data	API - STH	GitHub - STH
/metric	Context broker	Orion v1, Orion v2	GitHub - Orion
/gui	Graphical User Interface		GitHub - GUI
/sign	Public key signing	API - EJBCA-REST	GitHub - EJBCA-REST
/ca	Certification-Auth. functions	API - EJBCA-REST	GitHub - EJBCA-REST

The API gateway used in dojot reroutes some of these endpoints so that they become uniform: all of them are accessible through the same port (default is TCP port 8000) and have the same naming scheme. Each component, though, might have something different in its configuration and API documentation. The following table shows which endpoint exposed by the API gateway is mapped to which component endpoint.

Table 4.3: Original endpoints

Service	Original endpoint	Endpoint
DeviceManager	host:5000/device	host:8000/device
DeviceManager	host:5000/template	host:8000/template
mashup	host:3000/	host:8000/flows
auth	host:5000/	host:8000/auth
auth	host:5000/auth/revoke	host:8000/auth/revoke
auth	host:5000/user	host:8000/auth/user
STH	host:8666/	host:8000/history
Orion v1 or Orion v2	host:1026/	host:8000/metric
GUI	host/	host:8000/gui
ejbca	host:5583/sign	host:8000/sign
ejbca	host:5583/ca	host:8000/ca

Installation Guide - Docker compose

This document provides instructions on how to create a trivial deployment environment on single host for dojot, using docker-compose as the processes orchestration platform.

While very simple, this deployment option is best suited to development and assessment of the platform and should not be used for production environments.

This guide has been checked on an Ubuntu 16.04 LTS environment.

Table of Contents

- *Dependencies*
 - *Docker engine*
 - *Docker Compose*
- *Installation*
- *Configuration*
 - *API gateway configurarion*
 - *User creation*
- *Usage*

5.1 Dependencies

This setup has two software requirements docker engine and docker-compose.

5.1.1 Docker engine

Up to date information and installation procedures for the docker engine can be found at the project's documentation:

<https://docs.docker.com/engine/installation/>

Note: An optional step on the installation and configuration process of docker on any given machine is the setting of who is eligible for creating/spawning docker instances.

Should the post-installation steps (more specifically the “Manage docker as non-root user”) have not been run, all docker and docker-compose commands should be run by the super user (root), or as sudo.

<https://docs.docker.com/engine/installation/linux/linux-postinstall/>

5.1.2 Docker Compose

Up to date information and installation procedures for the docker-compose can be found at the project’s documentation:

<https://docs.docker.com/compose/install/>

5.2 Installation

To setup the environment, merely clone the deployment repository and run the commands below.

The docker-compose enabled deployment scripts and configuration repository can be found at:

<https://github.com/dojot/docker-compose>

or as git clone command::

```
git clone git@github.com:dojot/docker-compose.git
```

Once the repository is properly cloned, select the version to be used by checking out the appropriate tag (do notice that the tagname has to be replaced):

```
# Must be run from within the deployment repo
git checkout [tag name]
```

That done, the environment can be brought up by:

```
# Must be run from the root of the deployment repo.
# May need sudo to work: sudo docker-compose up -d
docker-compose up -d
```

To check individual container status, docker’s commands may be used, for instance:

```
# Shows the list of currently running containers, along with individual info
docker ps

# Shows the list of all configured containers, along with individual info
docker ps -a
```

Note: All docker, docker-compose commands may need sudo to work.

To allow non-root users to manage docker, please check docker’s documentation:

<https://docs.docker.com/engine/installation/linux/linux-postinstall/>

5.3 Configuration

Once the environment is up, a few configuration steps are required to make it operational.

5.3.1 API gateway configurarion

In order to guarantee the proper mapping of API into processing services, the API gateway must be configured. To do so, please run `kong_config.sh`, present at the root of the repository.

```
./kong.config.sh
```

5.3.2 User creation

To be able to use the system's web front-end and make API calls, a user must be created. To create a first *admin* user, the following script can be run on the host machine of the platform (that is, the machine where docker-compose was run). The script is located at the root of the repository.

```
./create.user.sh
```

5.4 Usage

The web interface is available at `http://localhost:8000`. The user is `admin` and the password is `admin`. You also can interact with platform using the REST API.

Read the [User Guide](#) for more information about how to interact with the platform.

Installation Guide - Google Cloud Platform

This document provides instructions on how to prepare a Google Cloud environment for a dojot deployment using Kubernetes as the orchestrator.

This document will provide steps for creating a test and assessment environment for those who want to learn and experiment with the dojot platform but prefer to run it on a cloud environment.

The steps as presented here can be evolved to real world deployments with proper changes to fulfill your deployment use case

Table of Contents

- *Creating a Project*
- *Creating a Cluster*
- *Getting the credentials*

6.1 Creating a Project

To prepare an environment to deploy dojot on the Google Cloud Platform, the first thing that must be done is to create a project for the deployment.

To create a project, go to the page <https://console.cloud.google.com/projectcreate> and define the new project's name.

Wait until the project creation is complete. Then, go to the page <https://console.cloud.google.com/projectselector/home/dashboard>, click on the select button and choose the recently created project.

6.2 Creating a Cluster

Having the desired project selected, got to the kubernetes page of the GCP at the link <https://console.cloud.google.com/kubernetes/>.

Wait for the Kubernetes Engine to be ready and click on the create cluster button.

In the cluster creation page, define a name for your cluster and select an appropriate region for your deployment. To create an evaluation and testing environment for dojot a cluster with 3 machines with 1 vCPU is enough for the sake of experimenting with the kubernetes deployment. With the options properly set, click on the create button and wait for the cluster to be created.

6.3 Getting the credentials

With the kubernetes cluster created, the next step is obtaining the cluster access credentials so your machine is able to access the cluster and proceed with the deployment.

On the kubernetes page, on the list of created clusters, locate the cluster you just created, on the right side of it click on the “Connect” button. Copy the first command that is provided and run this on a terminal on your machine, this command will install the credentials. To run this command it is required that the gcloud client is installed and properly configured on your machine. To install this client follow the instructions provided by the google documentation at: <https://cloud.google.com/sdk/docs/quickstarts>

With the credential configured, proceed to the *Kubernetes Deployment Guide*

Installation Guide - Kubernetes

This document provides instructions on how to create a simple dojoy deployment environment on a multi-node environment, using kubernetes as the orchestration platform.

This deployment option as presented in this document is best suited to tests and assessment of the platform, but with the appropriate changes might be evolved for production environments.

This guide has been checked on a Kubernetes cluster with Ceph as the underlying storage infrastructure and it has also been tested on a Kubernetes cluster over the Google Cloud Platform

Table of Contents

- *Dependencies*
 - *Kubernetes Cluster*
 - *Persistent Storage*
 - *Kubernetes Client*
- *Deployment*
 - *Google Cloud Platform*
 - *Cluster with Ceph*

7.1 Dependencies

This setup has as the first requirement a Kubernetes cluster that is properly configured and running.

The second requirement is a Kubernetes client correctly installed on the machine that will start the deployment process

7.1.1 Kubernetes Cluster

For this guide it is advised that you already have a functioning cluster.

If you desire to prepare a Kubernetes cluster from scratch, up to date information and installation procedures can be found at the project's documentation:

<https://kubernetes.io/docs/setup/>

7.1.2 Persistent Storage

To make sure that all the data from the containers running databases is persisted when containers fail or are moved to different nodes of the Kubernetes environment it is necessary to attach persistent storage to the database pods.

Kubernetes requires that an infrastructure for persistent storage already exists on the cluster. As an example for how to configure your persistent storage we provide files for two different kind of deployments, the first is for a local deployment where a Ceph Cluster is used as storage backend, more information on Ceph may be found at: <http://ceph.com/>. The second example is based on a Google Cloud deployment and use the existing persistent storage services that are provided by Google Cloud. If you're deploying dojot using Kubernetes to a different cloud provider, some adjustments to fit the different deployments might be necessary.

Information about the currently supported persistent storage for Kubernetes can be found at: <https://kubernetes.io/docs/concepts/storage/persistent-volumes/#types-of-persistent-volumes>

7.1.3 Kubernetes Client

To install the Kubernetes client on your machine before proceeding with this guide, follow the proper instructions as presented on the Kubernetes documentation:

<https://kubernetes.io/docs/tasks/tools/install-kubectl/>

Also, verify that your client is capable of connecting to the cluster.

For providing access for a local cluster, follow the documentation below:

<https://kubernetes.io/docs/tasks/access-application-cluster/access-cluster/>

If the Kubernetes cluster is running on a specific cloud platform like Google Cloud, follow the steps as presented by your cloud provider.

7.2 Deployment

To deploy dojot to a Kubernetes environment, we provide sample scripts and templates for two kinds of clusters. The examples are for an environment comprised by Kubernetes with Ceph for storage, the second is a deployment to a Kubernetes environment running on Google Cloud Platform.

For both environment it is necessary to download the scripts and templates before performing the deployment.

To download the required files using git, run the following command:

```
git clone https://github.com/dojot/kubernetes.git
```

or, to download a compressed zip file containing the data, use the following link: <https://github.com/dojot/kubernetes/archive/master.zip>

Enter the downloaded folder and follow the instructions in the section that corresponds to your specific environment.

All the instructions provided in the following sections assume that the commands are being run on a linux terminal.

7.2.1 Google Cloud Platform

To deploy dojot to a Kubernetes cluster running over Google Cloud the only requirements are that you have your cluster configured on Google Cloud and your local Kubernetes client is properly configured to access that cloud.

To execute the script to deploy to Google Cloud just run the following command on the terminal:

```
./deploy.sh GCP LB
```

The selected parameters set the type of storage to be used as GCP persistent storage and also set the external access to use the load balancers as provided by the Google Cloud Platform.

Just wait until the script finishes running and then check for when all the pods have finished starting, to check if all the pods are running correctly, run the command below and verify that all pods have reached a “Running” state, this may take a while and retries for some pods.

```
kubect1 get pods -n dojot
```

After all the pods are running, run the following command in order to obtain the public ip address that is being used by the load balancer

```
kubect1 -n dojot get services external
```

The command will return the external ip used by the load balancer, with this IP you can access that ip using any browser at http://EXTERNAL_IP

The initial user and password are admin and admin.

7.2.2 Cluster with Ceph

To deploy dojot to a Kubernetes Cluster where you have as persistent storage infrastructure a Ceph Cluster you will need the configuration file for accessing Ceph.

Also you will need to set some information regarding your Ceph cluster on the manifest files.

Edit the file “manifests/STORAGE/CEPH/rbd-provisioner.yaml” and change the values of the pool and the userId to match those of your specific environment. Also it is necessary to get the key for the admin user and the client user. With this keys at hand, convert then to base 64, this may be done at your terminal running the command:

```
echo "KEY" | base64
```

The value that is returned must be added to the “manifests/STORAGE/CEPH/ceph-secret-admin.yaml” and “manifests/STORAGE/CEPH/ceph-secret-user.yaml” respectively at the field key.

Also you may choose to deploy with a load balancer if your infrastructure provide one, otherwise you may deploy selecting a public ip of one of the kubernetes cluster nodes as the point of access for the environment.

To execute the script and deploy with Ceph and a public ip just run the following command on the terminal:

```
./deploy.sh CEPH PUBLIC_IP
```

Wait while the script starts the deployment, you will be prompted for two parameters during the deployment, the path for the ceph configuration file and the desired public ip. Enter this parameters and type enter when prompted.

Just wait until the script finishes running and then check for when all the pods have finished starting, to check if all the pods are running correctly, run the command below and verify that all pods have reached a “Running” state, this may take a while and retries for some pods.

```
kubect1 get pods -n dojot
```

After all the pods are running, you can access your dojot deployment using the public ip that was defined http://PUBLIC_IP

The initial user and password are admin and admin.

Running dojot on VirtualBox

This guide provides instructions to run dojot platform on VirtualBox.

You should only run dojot this way if you don't have any familiarity with docker and just want to learn how to use dojot. We don't recommend it for development and much less for experimental or real deployments.

The steps described here were checked on Windows 7, but you shouldn't have problems to run them in different operational systems.

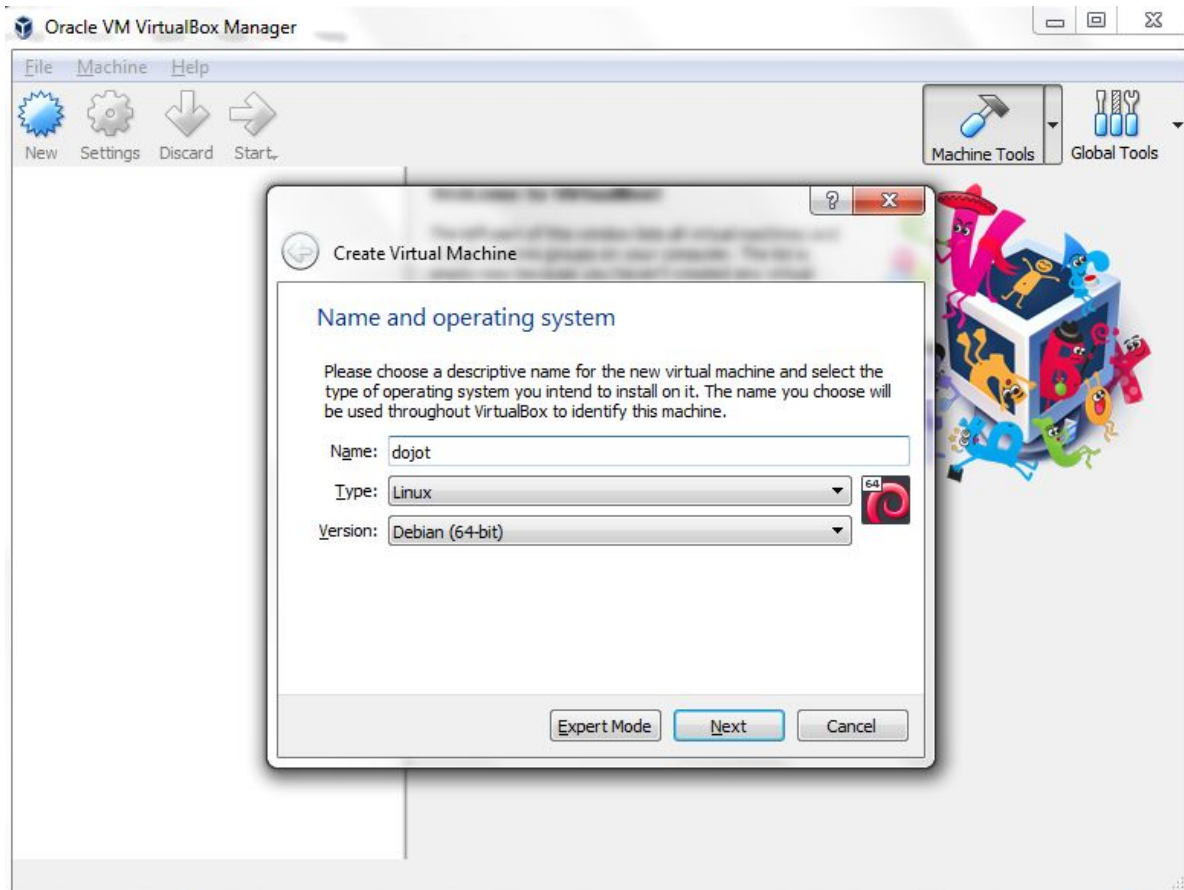
First of all, if you don't have VirtualBox you'll need to install it. Up to date information and installation procedures can be found at the project's documentation:

<https://www.virtualbox.org/>

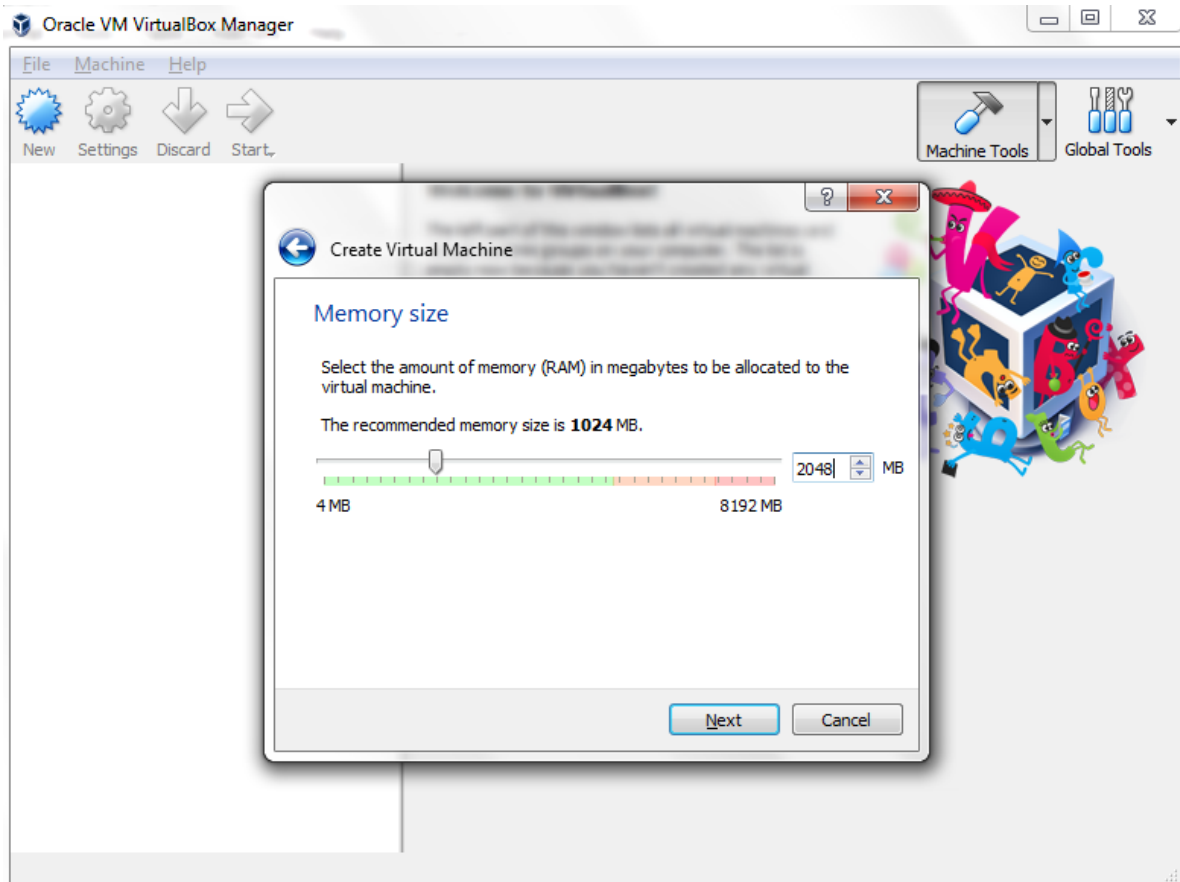
Then, you need to download a virtual machine image with dojot, which is available at:

<http://dojot-iso.s3.amazonaws.com/imagen/dojot.0.1.0-dojot.vdi>

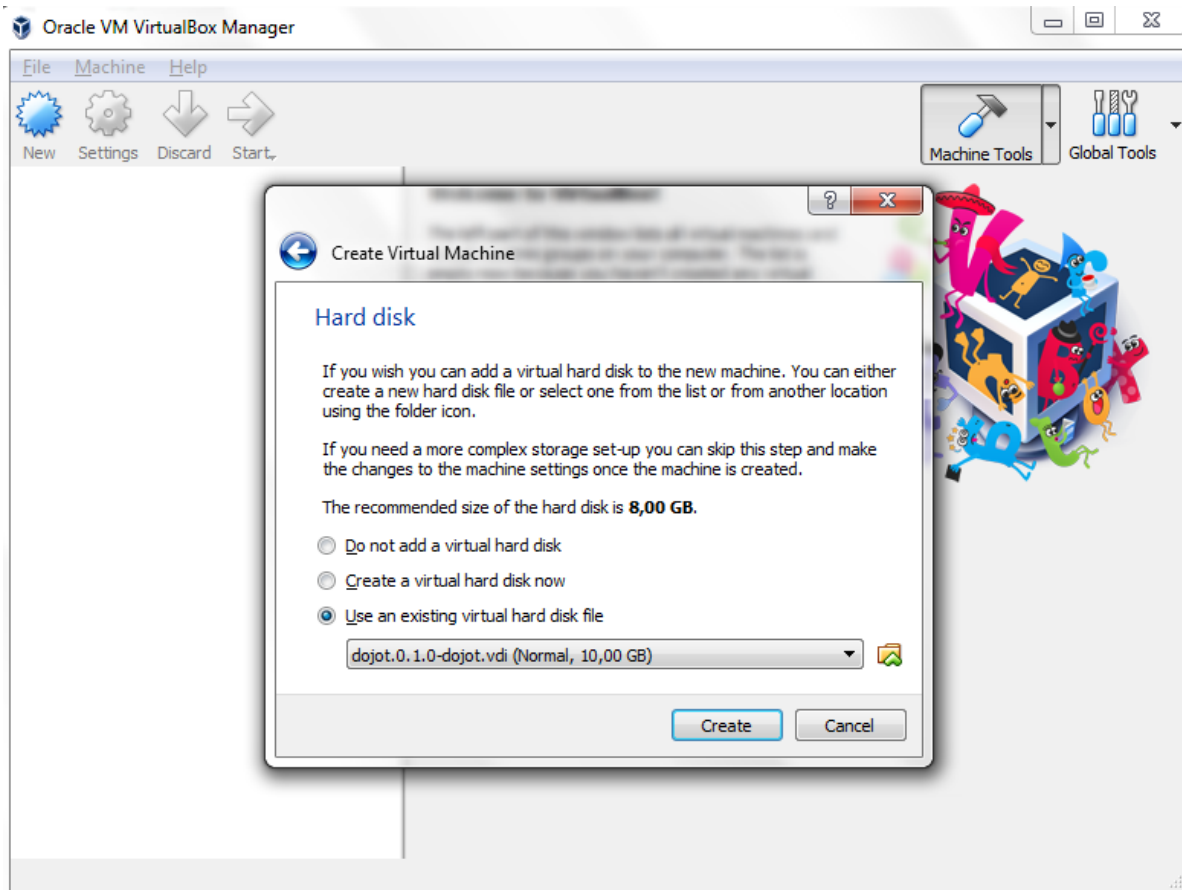
The next step is to create a virtual machine on VirtualBox. Click on the New button, then set the name as you wish, type to Linux and version to Debian (64-bit).



Click on Next, and set the memory size. We recommend at least 2048 MB.

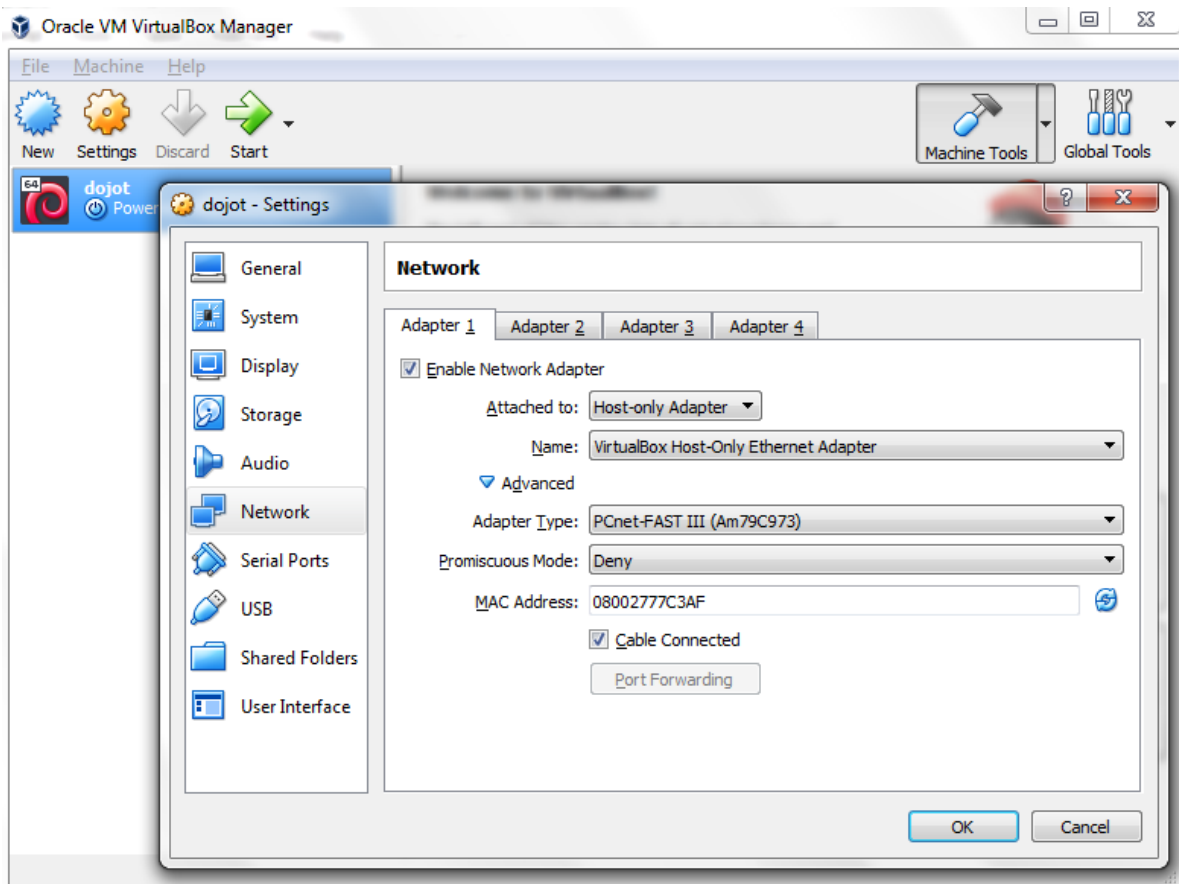


Click on Next, and set the hard disk to use an existing virtual hard disk file and choose the downloaded image.



Click on Create.

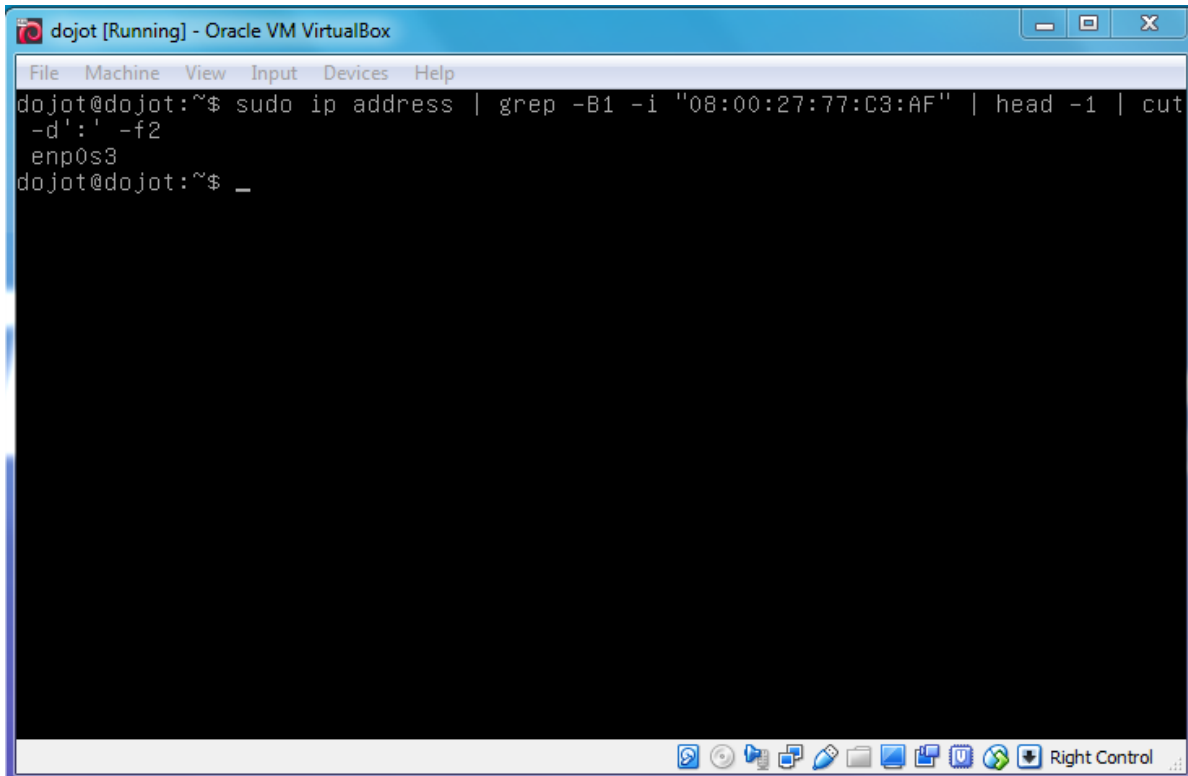
Next, click on Settings/Network and check whether the network adapter is enabled and set it to `Host-only`. This will allow host and guest to communicate to each other. Write down the MAC Address, you will need it later.



Click on OK and start the virtual machine.

Login in the virtual machine (login/password are dojot/dojot) to set the network interface. Firstly, get the interface name:

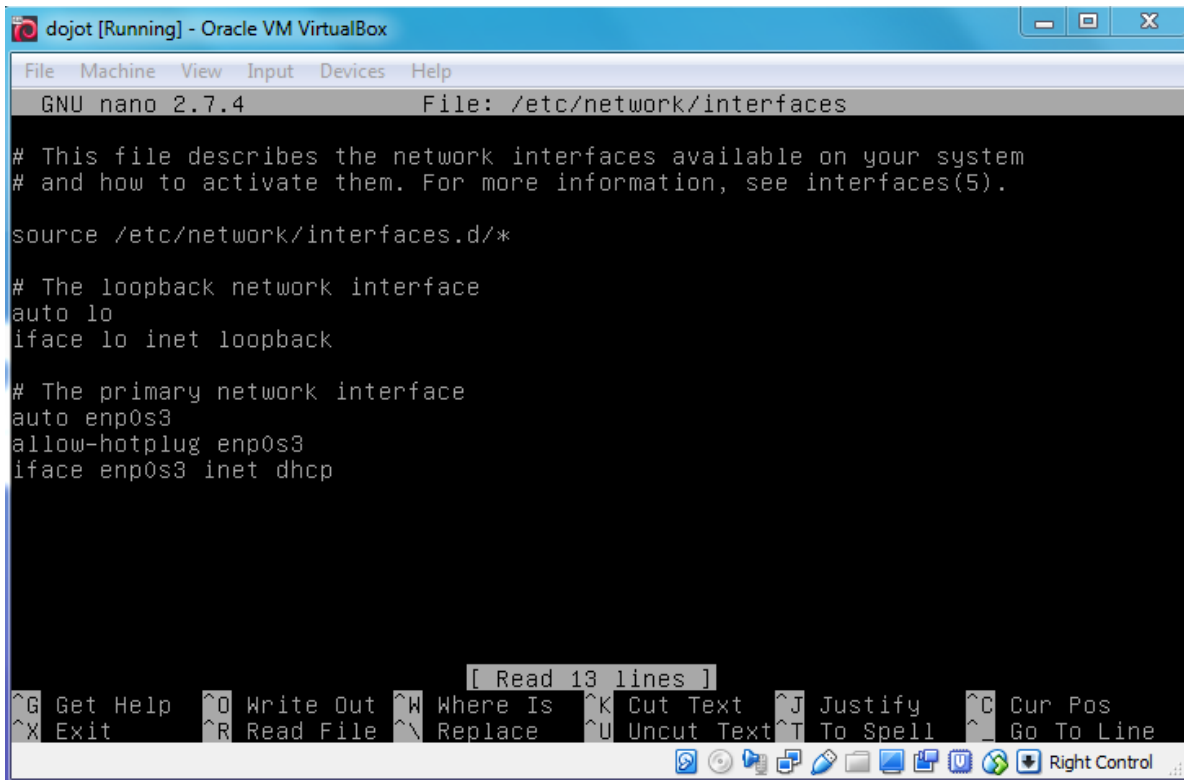
```
$ sudo ip address | grep -B1 -i "<YOUR MAC ADDRESS>" | head -1 | cut -d':' -f2
```



```
dojot [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
dojot@dojot:~$ sudo ip address | grep -B1 -i "08:00:27:77:C3:AF" | head -1 | cut
-d':' -f2
enp0s3
dojot@dojot:~$ _
```

Edit the file `/etc/network/interfaces`, adding

```
# The primary network interface
auto <YOUR INTERFACE NAME>
allow-hotplug <YOUR INTERFACE NAME>
iface <YOUR INTERFACE NAME> inet dhcp
```

```
dojot [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
GNU nano 2.7.4 File: /etc/network/interfaces

# This file describes the network interfaces available on your system
# and how to activate them. For more information, see interfaces(5).

source /etc/network/interfaces.d/*

# The loopback network interface
auto lo
iface lo inet loopback

# The primary network interface
auto enp0s3
allow-hotplug enp0s3
iface enp0s3 inet dhcp

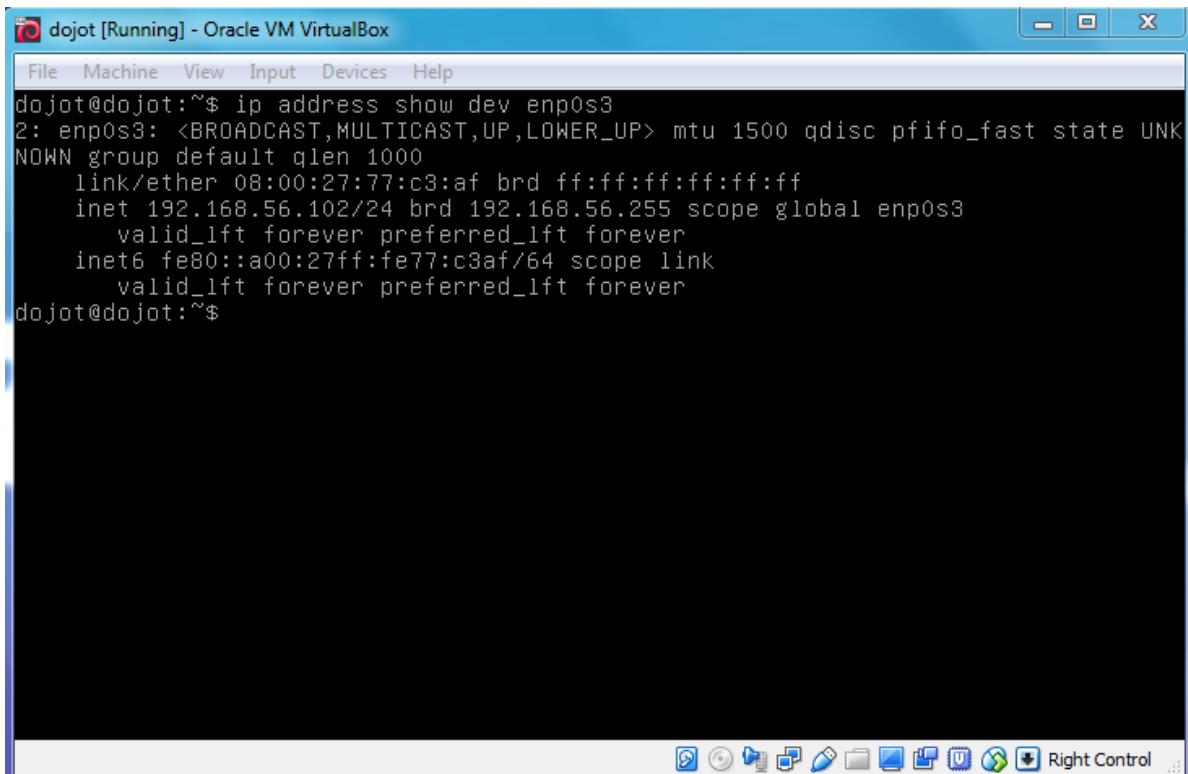
[ Read 13 lines ]
^G Get Help ^O Write Out ^W Where Is ^K Cut Text ^J Justify ^C Cur Pos
^X Exit ^R Read File ^_ Replace ^U Uncut Text ^T To Spell ^_ Go To Line
Right Control
```

Restart the networking service:

```
$ systemctl restart networking.service
```

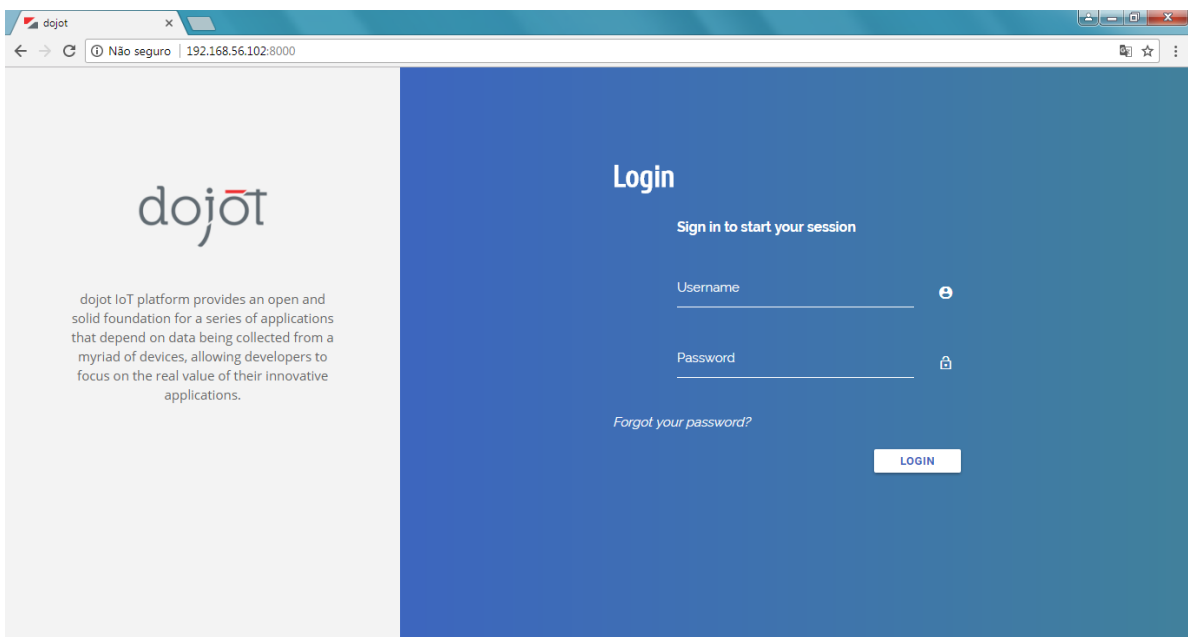
Get the ip address assigned to the interface:

```
ip address show dev <YOUR INTERFACE NAME>
```



```
dojot@dojot:~$ ip address show dev enp0s3
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UNK
NOWN group default qlen 1000
    link/ether 08:00:27:77:c3:af brd ff:ff:ff:ff:ff:ff
    inet 192.168.56.102/24 brd 192.168.56.255 scope global enp0s3
        valid_lft forever preferred_lft forever
    inet6 fe80::a00:27ff:fe77:c3af/64 scope link
        valid_lft forever preferred_lft forever
dojot@dojot:~$
```

In the host machine, open a browser and type <YOUR IP ADDRESS>:8000.



Frequently Asked Questions

Here are some answers to frequently-asked questions from dojot platform.

Got a question that isn't answered here? Please, open an issue on [dojot's Github repository](#).

Table of Contents

- *General*
 - *What is dojot? Why should I use it? Why open source it?*
 - *Where can I get it?*
 - *Which repository is the main one?*
 - *So, I found this pesky bug. How can I inform you about it?*
- *Usage*
 - *How do I start it? Is it CLI-based or it has a graphical user interface?*
 - *Ok, I started it and I logged in. Now what?*
 - *How can I update my deploy to dojot's latest version?*
- *Devices*
 - *What are devices for dojot?*
 - *What is the relationship between this device and my actual device?*
 - *What are virtual devices? How are they different from the other one?*
 - *How can I send MQTT data to dojot so that it appears on the dashboard?*
 - *On the dashboard some attributes are shown as tables and others as charts. How are they chosen/set?*
 - *I'm interested in integrating my super cool device with dojot. How can I do it?*
 - *Is there any restrictions about the message my device will send to dojot? Format, size, frequency?*

- *How can I send some commands to my device through dojot?*
 - *I didn't find the protocol supported by my device in the type list, is there anything I can do?*
 - *I saved an attribute, but it disappeared from the device. Is it a bug?*
 - *How can I retrieve historical data for a particular device?*
- *Data Flows*
 - *What is data flow?*
 - *The data flow UI... really looks like node-RED. Are they related in some way?*
 - *Why should I use it?*
 - *What can it do, exactly?*
 - *So, how can I use it?*
 - *Can I apply the same flow to multiple devices?*
 - *Can I correlate data from different devices in the same flow?*
 - *I want to send an email, what should I do?*
 - *What about a HTTP POST request, how can I send it?*
 - *I want to rename the attributes of a device, what should I do?*
 - *I want to aggregate the attributes of multiple devices, what should I do?*
 - *It would be cool a WhatsApp node, is it in roadmap?*
- *Applications*
 - *What APIs are available for applications?*
 - *How can I use them?*
 - *I'm interested in integrate my application with dojot. How can I do it?*

9.1 General

9.1.1 What is dojot? Why should I use it? Why open source it?

It's a brazilian IoT platform launched as open source software with aims to ease the development of solutions and the IoT ecosystem with local resources geared towards brazilians needs. It takes a role as an enabler platform with:

- Open APIs which makes the access to the platform resources easy.
- Capacity to store large volumes of data in different formats.
- Connectors to different types of devices.
- Graphical user interface with flow builder to prototype IoT solutions very quickly.
- Real time event processing with customizable rules.

dojot is based on Fiware, also an open source project, compromised to build an open and sustainable ecosystem grounded on open standards with the aim of easing the application development in different segments.

9.1.2 Where can I get it?

All components are available in dojot's GitHub repositories: <https://github.com/dojot>.

9.1.3 Which repository is the main one?

There are two main ones:

- <https://github.com/dojot/dojot>: this is where we keep track of all the things related to this project as a whole, such as architectural enhancements.
- <https://github.com/dojot/docker-compose>: repository for Docker compose files and configurations. This is what we would recommend to use to start with.

9.1.4 So, I found this pesky bug. How can I inform you about it?

We ask you to open an issue in [dojot's Github repository](#). If you know exactly which component is failing, you could open the issue in its repository (it will work the same way).

If you are able to analyze and fix this bug, please do so. Create a pull-request with a quick description of what you've done.

9.2 Usage

9.2.1 How do I start it? Is it CLI-based or it has a graphical user interface?

dojot can be accessed by a nice web-based interface and by REST APIs. Considering that you installed `docker` and `docker-compose` and cloned the `docker-compose` repository, there are a few steps to start it up:

```
$ docker-compose up -d
$ ./kong.config.sh
$ ./create-user.sh
```

And that's it.

The web interface is available at `http://localhost:8000`. The user is `admin`, password `admin`.

REST APIs are explained in the *Applications* section.

9.2.2 Ok, I started it and I logged in. Now what?

Nice! Now you can add your first devices, described in *Devices*, build some flows and subscribing to device events, both described in *Data Flows*.

9.2.3 How can I update my deploy to dojot's latest version?

You need to follow some steps:

1. Update the docker-compose repository to the latest version.

```
$ cd <path-to-your-clone-of-docker-compose>
$ git checkout master && git pull
```

2. Deploy the latest docker images.

```
$ docker-compose pull && docker-compose up -d --build
```

This procedure also applies to the available virtual machines once they do use docker-compose.

9.3 Devices

9.3.1 What are *devices* for dojot?

In dojot, a device is a digital representation of an actual device or gateway with one or more sensors or of a virtual one with sensors/attributes inferred from other devices.

Consider, for instance, an actual device with temperature and humidity sensors; it can be represented into dojot as a device with two attributes (one for each sensor). We call this kind of device as *regular device* or by its communication protocol, for instance, *MQTT device* or *CoAP device*.

We can also create devices which don't directly correspond to their actual ones, for instance, we can create one with higher level of information of temperature (*is becoming hotter* or *is becoming colder*) whose values are inferred from temperature sensors of other devices. This kind of device is called *virtual device*.

9.3.2 What is the relationship between this *device* and my actual device?

It is simple as it seems: the *regular device* for dojot is a mirror (digital twin) of your actual device. You can choose which attributes are available for applications and other components by adding each one of them at the device creation interface. If you don't want some attributes to be available to applications or other elements, just don't add them in dojot.

9.3.3 What are *virtual devices*? How are they different from the other one?

Regular devices are created to serve as a mirror (digital twin) for the actual devices and sensors. A *virtual device* is an abstraction that models things that are not feasible in the real world. For instance, let's say that a user has few smoke detectors in a laboratory, each one with different attributes. Wouldn't it be nice if we had one device called *Laboratory* that has one attribute *isOnFire*? So, the applications could rely only on this attribute to take an action.

Another difference is how virtual devices are populated. Regular ones will be filled with information sent by devices or gateways to the platform and virtual ones will be filled by flows or by applications (they won't accept messages addressed to them via MQTT, for example).

9.3.4 How can I send MQTT data to dojot so that it appears on the dashboard?

First of all, you create a digital representation for your actual device. Then, you configure it to send data to dojot so that it matches its digital representation.

Let's take as example a weather station which measures temperature and humidity, and publishes them periodically through MQTT. First, you create a device of type MQTT with two attributes (temperature and humidity). Then you set your actual device to push the data to dojot. Here, you need to follow some rules:

- MQTT topic must follow the pattern `/<service-id>/<device-id>/attrs`, where `<service-id>` is an identifier associated with the user account and the `<device-id>` is a unique identifier assigned by dojot. For example, topic `/admin/882d/attrs` must be used for user `admin` and device ID `882d`.
- MQTT payload must be a JSON with each key being an attribute of the dojot device, such as:

```
{ "temperature" : 10.5, "pressure" : 770 }
```

It's worth to point out that we are relaxing these rules so that you'll have more flexibility to configure both topic and payload. This feature will be available in the next official release.

9.3.5 On the dashboard some attributes are shown as tables and others as charts. How are they chosen/set?

The type of an attribute determines how the data is shown on the dashboard as follows:

- Geo: geo map.
- Boolean and Text: table.
- Integer and Float: line chart.

9.3.6 I'm interested in integrating my super cool device with dojot. How can I do it?

If your device is able to send messages using MQTT (with JSON payload), CoAP or HTTP, there is a good chance that your device can be integrated with minor or no modifications whatsoever. The requirements for such integration is described in the question [How can I send MQTT data to dojot so that it appears on the dashboard?](#).

9.3.7 Is there any restrictions about the message my device will send to dojot? Format, size, frequency?

None but format, which is described in the question [How can I send MQTT data to dojot so that it appears on the dashboard?](#).

9.3.8 How can I send some commands to my device through dojot?

This feature is not supported right now, but it is in roadmap and will be available in the next official release. If you are craving for this feature, please help us to develop it.

9.3.9 I didn't find the protocol supported by my device in the type list, is there anything I can do?

There are some possibilities. The first one is to develop a proxy to translate your protocol to one supported by dojot. The second one is to develop a connector similar to the existing ones for MQTT, CoAP and HTTP.

9.3.10 I saved an attribute, but it disappeared from the device. Is it a bug?

You might have saved the attribute, but not the device. If you don't click on the save button for the device, the added attributes will be discarded. We're improving the system messages to caveat the users and remember them to save their configurations.

9.3.11 How can I retrieve historical data for a particular device?

You can do this by sending a request to `/history` endpoint, such as:

```
curl -X GET \
  -H 'Authorization: Bearer eyJhbGciOiJIUzI1NiIsIn... ' \
  -H 'Fiware-Service:admin' \
  -H 'Fiware-ServicePath:/' \
  http://localhost:8000/history/STH/v1/contextEntities/type/device/id/3ba9/
↪attributes/temperature?lastN=10
```

which will retrieve the last 10 entries of *temperature* attribute from the device *3ba9*. There are more operators that could be used to filter entries. Check [STH](#) documentation to check out all possible operators.

9.4 Data Flows

9.4.1 What is data flow?

It's a processing flow for incoming messages/data of a device. With a flow you can dynamically analyse each new message in order to apply validations, infer information, and trigger actions or notifications.

9.4.2 The data flow UI... really looks like node-RED. Are they related in some way?

It's based on the Node-RED frontend, but uses its own engine to process the messages. If you're familiar with Node-Red, you won't have any difficulty to use it.

9.4.3 Why should I use it?

It allows one of the coolest things of IoT in an easy and intuitive way, which is to analyse data for extracting information, then take actions.

9.4.4 What can it do, exactly?

You can do things such as:

- Create virtual viewers of a device (rename attributes, aggregate attributes, change values, etc).
- Infer information based on switch rules.
- Infer information based on edge-detection rules.
- Infer information based on geo-fence rules.
- Notify through email.
- Notify through HTTP.

The data flows component is in constant development with new features being added every new release.

9.4.5 So, how can I use it?

It follows the basic usage flow as node-RED. You can check its [documentation](#) for more details about this.

9.4.6 Can I apply the same flow to multiple devices?

Multiple devices can be used both as input and output of data flows. It's worth to point out that the flow is processed individually for each new input message, i.e. for each input device.

9.4.7 Can I correlate data from different devices in the same flow?

As the data flow is processed individually for each message, you need to create a virtual device to aggregate all attributes, then use this virtual device as the input of the flow.

9.4.8 I want to send an email, what should I do?

Basically, you need to add an email node and configure it. This node is pre-configured to use the Gmail server `gmail-smtp-in.l.google.com`, but you're free to choose your own. For writing an email body, you can use a template before the email.

It is important to point out that dojot contains no e-mail server. It will generate SMTP commands and send them to the specified e-mail server.

9.4.9 What about a HTTP POST request, how can I send it?

It is almost the same process as sending an e-mail.

One important note: make sure that dojot can access your server.

9.4.10 I want to rename the attributes of a device, what should I do?

First of all, you need to create a virtual device with the new attributes, then you build a data flow to rename them. This can be done connecting a 'change' node after the input device to map the input attributes to the corresponding ones into an output, and finally connecting the 'change' to the virtual device and assigning to it the output.

9.4.11 I want to aggregate the attributes of multiple devices, what should I do?

First of all, you need to create a virtual device to aggregate all attributes, then you build a data flow to map the attributes of each device to the virtual one. This can be done connecting a 'change' node after each input device to put the input values into an output, and finally connecting all changes to the virtual device and assigning to it the output.

9.4.12 It would be cool a WhatsApp node, is it in roadmap?

It's under analysis. We intend to support other notifications systems besides email, including WhatsApp, Twitter and Telegram. If you also have interest, please help us to develop them.

9.5 Applications

9.5.1 What APIs are available for applications?

You can check all available APIs in the [API Listing](#) page

9.5.2 How can I use them?

First, you will need an access token, which can be retrieved sending a HTTP POST request to `/auth` endpoint with the following JSON content:

```
{  "username" : <>,  "passwd" : <> }
```

Obviously the values of each attribute should be correctly filled in. An example of such request using `curl` would be:

```
$ curl -X POST http://localhost:8000/auth -H 'Content-Type:application/json' \
$ -d '{"username" : "admin", "passwd" : "admin"}'
```

which gives us back:

```
{"jwt": "eyJhbGciOiJIUzI1..."}
```

This token (which is a lengthy alpha-numeric string) should be used in every request that is sent to dojot (excluding, of course this request). Each call for this API will generate a different token.

This token should be placed in a `Authorization` HTTP header, such as:

```
$ curl -X GET http://localhost:8000/device -H 'Authorization: Bearer eyJhbGciOiJIUzI1...
↪..''
```

A few endpoints requires two more headers, the `Fiware-Service` and `Fiware-ServicePath`. They are: `/metrics/`, `/iot/` and `/history/`

`Fiware-Service` header should contain the service name associated to the user. In general, it should be the username. `Fiware-ServicePath` is always a forward slash (`/`). An example:

```
curl -X GET http://localhost:8000/metric/v2/entities -H 'Authorization: Bearer_
↪eyJhbGciOiJIUzI1...' \
-H 'Fiware-Service:admin' -H 'Fiware-ServicePath:/'
```

9.5.3 I'm interested in integrate my application with dojot. How can I do it?

This should be pretty straightforward. There are two ways that your application could be integrated with dojot:

- **Retrieving historical data:** you might want to periodically read all historical data related to a device. This can be done by using this API (one side-note: all endpoints described in this apiary should be preceded by `/history/`).
- **Subscribing to events related to devices:** if your application is able to listen to events, you might rather use subscriptions, which can be created using this API (also, all endpoints should be preceded by `/metrics/`).
- **Using mashup to pre-process data:** if you want to do something more, you could use flows. They can help process and transform data so that they can be properly sent to your application via HTTP request, by e-mail or stored in a virtual device (which can be used to generate notifications as previously described).

All these endpoints should bear an access token, which is retrieved as described in the question *[How can I use them?](#)*.

Mutual Authentication

A security role is to ensure that only legitimate users have access to the resources and information they need to perform their duties. Authentication is part of this access control, when validating entities identity. At the same time, another security role is to ensure that an entity accesses legitimate resources and information, thereby avoiding situations such as sending information to fraudulent servers, for example.

Mutual authentication is the process in which two entities authenticate each other. In a client-server communication, the client must prove its identity to the server and the server must prove its identity to the client. Thus, each entity can ensure that they are communicating with a legitimate interlocutor.

Mutual authentication protects access to data the application accesses from *dojot* and therefore protects access to data of that application's user. It is done by ensuring that only registered applications can access platform data and functionality. In addition, it ensures that the platform the application is accessing is legitimate, meaning that no attacker can pass themselves by the platform and get user or application data.

Dojot offers a mutual authentication service through a Docker image. This service runs inside the platform and can be accessed using its interfaces.

Table of Contents

- *Using Mutual Authentication*
- *Application Registration*
- *Authentication*
 - *Library Initialization*
 - *Callback Registration*
 - *Call mutual authentication function*
- *Accessing dojot APIs*

10.1 Using Mutual Authentication

Applications can access *dojot* functionality to interact with its components and connected devices. For an application to ensure that it is communicating with a legitimate platform (and vice versa), it must make use of the mutual authentication functionality *dojot* provides. This is a simple process and its use requires only three steps to follow:

- **Application Registration.** When an application is registered in *dojot*, it receives an identifier and a key that must be kept secret. This key is used to authenticate the application on the platform.
- **Authentication.** At the beginning of the communication between application and *dojot*, the application initiates a handshake in which the two entities will exchange information to ensure they are legitimate.
- **Using the platform.** When accessing *dojot* interfaces, the platform informs a session identifier that is obtained at the time of authentication. Thus, the platform can verify that the mutual authentication process was performed by the application.

10.2 Application Registration

An application that is registered with *dojot* will receive an identifier and a key that must be kept secret. The registration indicates that an application will communicate and use platform features.

Currently, the method used to register an application is the use of a REST interface. After making the request for the registration, the application will receive a unique identifier and a key. The API is described below

REGISTER COMPONENT - Register new application

```
POST /kerberos/registerComponent

Response  200

Headers
Content-Type: application/json

Body
{
  "AppId": "0001020304050607",
  "AppKey": "000102030405060708090a0b0c0d0e0f000102030405060708090a0b0c0d0e0f"
}
```

Received identifier and key will be used at the moment the application authenticates with *dojot*. In order to do this, a client library is provided to perform the authentication process (available in github.com/dojot/ma-client-lib) and therefore, the library should have knowledge about the values of the identifier and the key. The file <https://github.com/dojot/ma-client-lib/kerberos/src/protocol/unique.h> is used to store these values and will be used by the library at the moment of authentication.

10.3 Authentication

When communicating with *dojot*, the application must perform mutual authentication. This process is done through the library provided in github.com/dojot/ma-client-lib. By using the library, three steps should be followed:

1. Initialize the library with server addresses
2. Register the callback function
3. Call mutual authentication function

10.3.1 Library Initialization

Initialization tells the library which URLs will be used to perform mutual authentication. The function to be used is described below:

Initialize Kerberos

```
errno_t initializeKerberos(uint8_t* host, uint8_t hostLength, uint8_t* uriRequestAS,
↳uint8_t requestASLength, uint8_t* uriRequestAP, uint8_t requestAPLength)
```

The arguments used in the function are described below.

- host - Platform main URL
- hostLength - Host string size
- uriRequestAS - requestAS endpoint
- requestASLength - requestAS string size
- uriRequestAP - requestAP endpoint
- requestAPLength - requestAP string Size

The following code snippet shows an example of how the function can be used.

```
char* host = "http://localhost:8000/"; // dojot URL
char* reqAS = "kerberos/requestAS";
char* reqAP = "kerberos/requestAP";

errno_t ret = initializeKerberos(host, strlen(host), reqAS, strlen(reqAS), reqAP,
↳strlen(reqAP));
```

10.3.2 Callback Registration

On the mutual authentication process, the library communicates with the server and checks received data. If an error occurs during this process, the library will call a callback function.

This callback function is implemented by the library user and must be registered before the authentication process. The callback function can include code for error handling and logging, for example.

Set Callback

```
errno_t setCallback(void (*callback)(int))
```

The following code shows an example of how the callback function can be created and registered.

```
void errorCallback(int err){
    // Error handling and logging code
}

errno_t ret = setCallback(&errorCallback);
```

10.3.3 Call mutual authentication function

After initializing the library with platform URL and registering the callback function, the library is ready to perform the mutual authentication process. The function that is used to perform the process is shown below.

```
errno_t executeKerberosHandshake()
```

The code below shows an example of how the function may be used.

```
errno_t ret = executeKerberosHandshake();
```

10.4 Accessing *dojot* APIs

After the mutual authentication process completes, the application may send additional data in the calls to the platform interfaces. This data is the mutual authentication session identifier and is sent through an HTTP header.

The following is an example of a call to a *dojot* API where mutual authentication session identifier is also sent.

```
GET /device HTTP/1.1
Host: localhost:8000
ma-session-id: a4cdad05441940c5c07ee9f55b8fafbdc0eba14afce449c9c9ec052bb20f50f4
```

CHAPTER 11

Crypto Service

Crypto Service provides data encryption and decryption functions to other *dojot* components. It is used only by internal services so they can protect data communication (both internally and externally) and data storage.

Available as a Docker image, Crypto Service can be instantiated easily and integrated in a short time. Encrypt and decrypt data functionalities are accessed through REST APIs.

Table of Contents

- *REST APIs*
- *Usage Examples*

11.1 REST APIs

Encrypt and decrypt data APIs are described below.

Decrypt

POST /crypto/decrypt

Request

Headers

```
Content-Type: application/json
```

Body

```
{
  "data": "Clear or cipher data",
  "tagSize": 16,
  "key": "2034F6E32958647FDFF75D265B455EBF40C80E6D597092B3A802B3E5863F878C",
}
```

(continues on next page)

(continued from previous page)

```
{  
  "iv": "AD0ACC568C88C116D57B273D98FB92C0"  
}
```

Response 200

Headers

```
Content-Type: application/json
```

Body

```
{  
  "data": "Cipher or clear data",  
  "result": "SUCCESS"  
}
```

Encrypt

POST /crypto/encrypt

Request

Headers

```
Content-Type: application/json
```

Body

```
{  
  "data": "Clear or cipher data",  
  "tagSize": 16,  
  "key": "2034F6E32958647FDDFF75D265B455EBF40C80E6D597092B3A802B3E5863F878C",  
  "iv": "AD0ACC568C88C116D57B273D98FB92C0"  
}
```

Response 200

Headers

```
Content-Type: application/json
```

Body

```
{  
  "data": "Cipher or clear data",  
  "result": "SUCCESS"  
}
```

11.2 Usage Examples

In order to use cryptographic functions provided by Crypto Service, one must access the available REST APIs through a HTTP request.

Examples of how those requests can be made are showed bellow using the command line tool curl.

Encrypt


```
curl -X POST \
  http://localhost:8080/cryptointegration/rest/crypto/encrypt \
  -H 'content-type: application/json' \
  -d '{
    "data": "000102030405060708090A0B0C0D0F",
    "tagSize": 16,
    "key": "2034F6E32958647FDFF75D265B455EBF40C80E6D597092B3A802B3E5863F878C",
    "iv": "AD0ACC568C88C116D57B273D98FB92C0"
  }'
```

Decrypt

```
curl -X POST \
  http://localhost:8080/cryptointegration/rest/crypto/decrypt \
  -H 'content-type: application/json' \
  -d '{
    "data": "C0FBC8DB5F72AD8DC04ECA2E32DA793F86D59D6",
    "tagSize": 16,
    "key": "2034F6E32958647FDFF75D265B455EBF40C80E6D597092B3A802B3E5863F878C",
    "iv": "AD0ACC568C88C116D57B273D98FB92C0"
  }'
```


This document describes how to configure dojot to use MQTT over TLS.

Table of Contents

- *tl;dr*
- *Components*
 - *EJBCA-REST*
 - * *What is a certificate?*
 - * *I have a CSR. How can I ask EJBCA to sign it for me?*
 - * *So, how does EJBCA work in dojot?*
 - *MQTT Manager*
- *Mosquitto configuration files*
- *Certificate retriever*
- *Important Notes*
 - *CRL (Certification Revocation List)*
 - *Debugging*
 - * *How to read a certificate*
 - * *Errors in secure connection handshake between device and Mosquitto*
 - * *Handshake is OK, but no published data reaches iotagent*

12.1 tl;dr

For a device to connect using TLS with Mosquitto, it must possess:

- A key pair (.key file);
- A certificate signed by a Certificate Authority (CA) trusted by Mosquitto (.crt file);
- The certificate of this CA (.crt file);
- An entry on Mosquitto Access Control List (ACL), allowing the device to publish on a specific topic;
- (optional) A Certificate Revocation List (CRL).

When a device is created, DeviceManager will automatically notify the following components:

- IoTAgent: will register the new device on its internal cache.
- MQTT-Manager: will create an entry on the ACL, allowing the device to publish on a specific topic.
- EJBCA: will create an end entity so a certificate can be created on the future.

By default, dojot uses clear MQTT. To activate TLS, docker-compose.yml must be changed:

- The image for service 'mqtt' must be changed from 'ansi/mosquitto' to 'dojot/mqtt-manager';
- The public port for 'mqtt' service must be changed from '1883:1883' to '8883:8883';
- The MQTT_TLS variable of 'iotagent' service must be set to true (lowercase).

On the configuration file 'iotagent/config.json':

- The flag 'secure' should be changed to true

12.2 Components

12.2.1 EJBCA-REST

EJBCA is a complete Private Key Infrastructure (PKI) capable to manage CAs, cryptography keys and certificates. EJBCA provides a SOAP, web and a command line interface. EJBCA-REST is an wrapper on top of EJBCA that provides modern interfaces, like REST and Kafka.

EJBCA provides SOAP, web and command line interfaces. EJBCA-REST is a wrapper on top of EJBCA that complements those, allowing the CA to be configured using REST. When used within dojot, it also listens to Kafka events, allowing its automatic configuration.

What is a certificate?

A certificate contains the public key for an entity (a user, device, website), along with information about this entity, about the CA which signs the certificate, the allowed certificate usage and a checksum. When a entity wants a certificate to be signed, the entity should create a CSR file and send it to the desired CA. The CSR file is an 'intention of certification'. The file contains the information required from the entity and some information about the certificate use, hostnames and IPs where the certificate will reside, alternative names for the entity, etc. EJBCA can decide, using its configured policies, what information to keep, to discard and to overwrite of the received CSR. EJBCA can refuse to sign a CSR if it concludes that it is not safe enough according to its policies.

These configurable policies are called 'Certificate Profiles'. One Certificate profile named CFREE, specialized for MQTT TLS, is provided out of the box.

In short, CFREE have the following configurations (and many more):

- Cryptography keys must have between 2048 and 8192 bits;
- Certificate expires in 730 days;
- Entities can define hostnames and IPs;
- Key usage is marked as not critical (for now);
- The hash algorithm is SHA256. The sign algorithm is RSA.

I have a CSR. How can I ask EJBCA to sign it for me?

Calm down! EJBCA will not allow strangers to ask for certification. You need to authenticate yourself. EJBCA use a username+password authentication system. The term ‘end entity’ will be used to refer to EJBCA users to follow the terms on EJBCA documentation and to avoid ambiguities between EJBCA users and dojot users. An administrator should create the end entity. An entity that was just created has the state ‘New’ and can generate a certificate. After signing a certificate for an entity, the end entity’s state changes to ‘Generated’ and will no longer accept this username and password. EJBCA ‘End entities’ can create only one certificate.

So, how does EJBCA work in dojot?

When creating a new device, an associated end entity is created in EJBCA. Its name will be the device’s ID (like ‘8fa3’) and its password will be always ‘dojot’.

A certificate can be signed by sending a HTTP POST request to host:1234/sign/<cname>/pkcs10. CName is the end entity’s name (or device). The payload sent with this request should be a JSON containing the end entity password and a CSR file (certificate intention) in base64 format.

Note that the URL is ‘routed’ by the API gateway. As in other APIs in dojot, a JWT is needed. You can find how to generate and how to use such token in [User Guide](#).

In order to create the CSR file and ask for a certificate signature, a user can use a helper script called ‘Certificate Retriever’, which is detailed in [Certificate retriever](#) section.

12.2.2 MQTT Manager

MQTT-Manager is a helper service used to configure Mosquitto MQTT broker in a simple and ‘on-the-fly’ way. It can be configured using REST interfaces and Kafka. Thus, HTTP requests or Kafka messages can be used to create and remove devices, as well as update CRL file (certification revocation list). This service is distributed as a docker container for easy deploy and its source code repository can be accessed in [MQTT Manager repository](#).

Mosquitto by itself doesn’t generate nor revoke certificates, it only relies on a CA and implements TLS protocol. The ‘creation’ of a particular device consists only in adding a new rule to ACL file in Mosquitto. Such file looks like:

```
user iotagent
topic read /#
user 24f6
topic write /admin/24f6/attrs
```

Each rule is composed by two lines: the first one specifies the user (device) and the second one defines which action (write or read) is allowed to which topic. In the example above, the user iotagent can read all topics (# is a wildcard). Also, the device with ID 24f6 can write to topic /admin/24f6/attrs. The device ID is retrieved in ‘Common name’ certificate field.

If a device sends data to a topic which it has no write permissions, then all data is discarded. Mosquitto won’t log any errors related to this.

When the ACL is changes, Mosquitto must be restarted (or a SIGDUP signal can be sent to its process). MQTT-Manager does this automatically when creating or removing devices.

A script is executed when firing the container up. This script will generate a pair of keys to Mosquitto, retrieves the certificate and CRL from a CA and asks it to sign its public key. All generated files are placed in `/usr/local/src/mosquitto-1.4.13/certs` (inside the container).

Mosquitto will only accept device connections that have certificate signed by its trusty CA.

Also note that MQTT-Manager is used only in case when a TLS-enabled broker is needed. If this is not the case, then the vanilla [Mosquitto docker image](#) can be used.

12.3 Mosquitto configuration files

Checkout this commented Mosquitto configuration file:

```
# network port on which Mosquitto will accept new connections
port 8883

# Trusted CA certificate
cafile /usr/local/src/mosquitto-1.4.13/certs/ca.crt

# Mosquitto certificate
certfile /usr/local/src/mosquitto-1.4.13/certs/mosquitto.crt

# Mosquitto key par
keyfile /usr/local/src/mosquitto-1.4.13/certs/mosquitto.key

tls_version tlsv1.2

# If false, a device will check Mosquitto certificate, but Mosquitto won't check
# the device counterparts.
# If true, both checks are performed (2-way TLS)
require_certificate true

# Certificate Common Name field will be used as username.
# Thus, a device with 'CN=abc1' will have a 'user abc1' entry in Mosquitto's ACL
use_identity_as_username true

# Permission list file
acl_file /usr/local/src/mosquitto-1.4.13/certs/access.acl

# CA CRL.
crlfile /usr/local/src/mosquitto-1.4.13/certs/ca.crl
```

Note that for all configuration updates, it is mandatory to restart Mosquitto or to send a SIGDUP signal to its process.

12.4 Certificate retriever

This component is a helper script for device certificates creation. It is available at [Certificate Retriever GitHub repository](#) and it coded using Python 3.

A user can use it by executing:

```
./certificate-retriever.py HOST DEVICE-NAME CA [OPTIONS]
```

The mandatory parameters are:

- HOST: where dojot is. Example: <http://localhost:8000>
- DEVICE-NAME: device name that will get a new certificate. Example: ac32
- CA: CA which will sign the certificate. Example: IOTmidCA (this is the CA name used in dojot)

Other options are:

- -u or --username USERNAME: dojot's username. If this parameter is not specified here, it will be asked iteratively.
- -w or --overwrite: overwrites any certificate files or cryptographic keys if already existent.
- -k or --key KEYLENGTH: size of the cryptographic key being generated (in bits).
- -d or --dns: Hostname where the certificate owner can be reached out. Note that this has no relation with DNS (Domain Name System) servers - this name was kept because x509 certificates have an attribute that is called DNS.
- -i or --ip: same as -d, but to specify IP address.
- --skip-https-check: if dojot accepts HTTPS connections but it has no valid certificate, then this option will allow the connection to be made.

Note that authentication is performed in dojot. The script will ask for user credentials and will invoke user authentication automatically. The user needs permission for certificate signing to be able to use this script.

An end entity must exist in EJBCA in 'New' state before asking for a new certificate signature. When a new device is created, an end entity is automatically created in EJBCA by DeviceManager. This new end entity's name is the device ID itself. Its password is 'dojot'.

The script authenticates users with given username and password, retrieves CA certificate, generates a key pair as well as a CSR file and asks for certificate signature, in this order. Any error in any step will halt its execution.

After successfully executed, all certificates can be found in './certs' folder.

12.5 Important Notes

These are a few but important notes related to device security and associated subjects.

12.5.1 CRL (Certification Revocation List)

A CRL is a list which contains all revoked certificates. It is used to indicate which certificates are no longer valid (administratively set to invalid) as a normal certificate can be used for 1 to 5 years. This list is signed by CA and also has an expiration date - 1 day by default. In TLS protocol, if CRL is expired then the recommended action to be taken is to refuse all incoming connections, as there is no way to check if the certificates used in those connections are invalid or not. This procedure is implemented in Mosquitto.

Therefore, CA must generate a new list periodically. All components that use it must be updated.

12.5.2 Debugging

TLS errors might be not so verbose as other problems. If an error occurs, the user might not know what went wrong because no component indicates any problem. In this section there are some tips, frequent problems and debugging tools to find out what's happening.

How to read a certificate

A certificate file can be in two formats: PEM (base64 text) or DER (binary). OpenSSL offers tools to read such formats:

```
openssl x509 -noout -text -in certFile.crt
```

To read a CRL:

```
openssl crl -inform PEM -text -noout -in crlFile.crl
```

Errors in secure connection handshake between device and Mosquitto

If any errors occur during connection handshake, something like the following error might appear in Mosquitto's logs:

```
1514550332: New connection from 172.20.0.1 on port 8883.  
1514550332: OpenSSL Error: error:140940E5:SSL routines:ssl3_read_bytes:ssl handshake_  
↪ failure
```

If this happens, try to establish connection using 'openssl client', as it is more verbose in error description.

```
openssl s_client -connect localhost:8883 -CAfile ca.crt -cert device.crt -key device.  
↪ key
```

Common errors are shown by openssl_client (and _server as well):

- SSL alert number 45: this error indicates that a certificate expired. Keep in mind that CRL also expires.
- SSL alert number 48: received a valid certificate chain or partial chain, but the certificate was not accepted because the CA certificate could not be located or could not be matched with a known, trusted CA. This message is always fatal.
- Alert unknown CA: check whether sent CA certificate is correct. If it is a sub-CA, check if all of its certificate chain was sent. This error also occurs if the CA certificate data (specially common name attribute) is the same as those from client certificate.

Handshake is OK, but no published data reaches iotagent

You can check whether the device could connect to MQTT broker by checking Mosquitto's log:

```
1514482004: New client connected from 172.20.0.10 as mqttjs_c011c22d (cl, k10, u  
↪ 'deviceName')
```

If that line shows up, it means that the TLS handshake worked and the device successfully connected to Mosquitto. Check if the device has an ACL entry in Mosquitto to allow it to publish data in the specified topic. Keep in mind that if a device publishes something in another topic (which it has no permission to publish) all data is discarded by Mosquitto with no warnings.