dojot Documentation

Release v0.8.0

Dojot Team

Jul 07, 2021

CONTENTS:

This is the high-level documentation for dojot IoT platform developed by CPqD. This platform aims to provide the application and device developers with a more concise and integrated interaction, while benefiting for a highly customizable and efficient infrastructure.

ARCHITECTURE

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 minutiae of each component's implementation. For that, please refer to each component's own documentation.
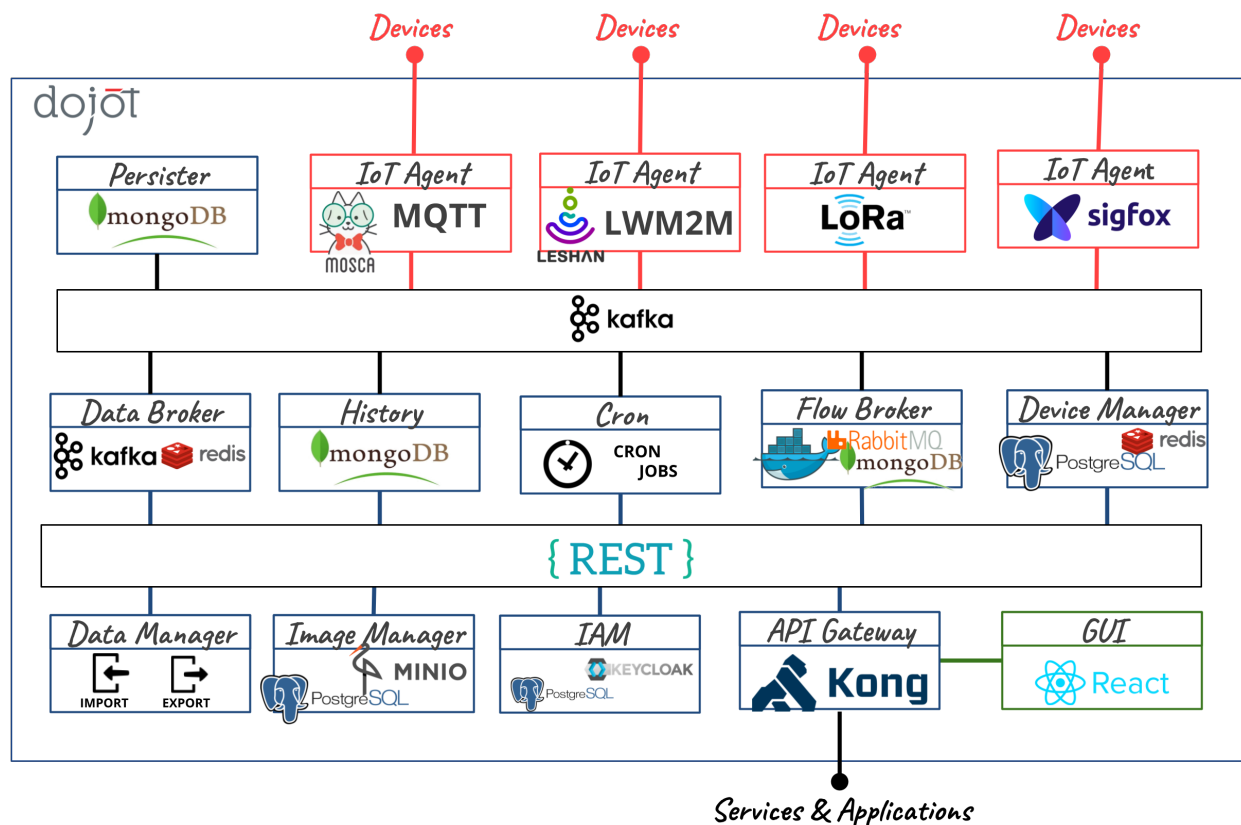
Fig. 1.1: : The microservice architecture of dojot platform.

A big picture of the whole architecture is shown in the figure above and in the following sections more details are given about each component.

1.1 Components

Dojot was designed to make fast solution prototyping possible, providing a platform that's easy to use, scalable and robust. Its internal architecture makes use of many well-known open-source components with others designed and implemented by dojot team.

Using dojot is as follows: a user configures IoT devices through the GUI or directly using the REST APIs provided by the API Gateway. Data processing flows might be also configured - these entities can perform a variety of actions, such as generate notifications when a particular device attribute reaches a certain threshold or send data generated by a device to an external service. As devices start sending their readings to dojot, a user can:

- receive these readings in real time by socket.io or websocket channels;
- consolidate all data into virtual devices;
- gather all data from historical database, and so on.

These features can be used through REST APIs - these are the basic building blocks that any application based on dojot should use. dojot GUI provides an easy way to perform management operations for all entities related to the platform (users, devices, templates and flows) and can also be used to check if everything is working fine.

The tenant contexts are isolated and there are no data sharing, the access credentials are validated by the authorization service (keycloak) for each and every operation (API Request). Therefore, a user belonging to a particular context (tenant) cannot reach any data (including devices, templates, flows or any other data related to these resources) from other ones.

Once devices are configured, the IoT Agent is capable of mapping the data received from devices, encapsulated on MQTT for example, and send them to the message broker for internal distribution. This way, the data reaches the persistence service, for instance, so it can persist the data on a database.

For more information about what's going on with dojot, you should take a look at dojot GitHub repository. There you'll find all components used in dojot.

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

1.1.1 Kafka + DataBroker

Apache Kafka is a distributed messaging platform that can be used by applications which need to stream data or consume/produce data pipelines. In comparison with other open-source messaging solutions, Kafka seems to be more appropriate to fulfil *dojot*'s architectural requirements (responsibility isolation, simplicity, and so on).

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

The DataBroker 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 consume real-time data based on context. DataBroker can also be a distributed service to avoid it being a single point of failure or even a bottleneck for the architecture.

1.1.2 DeviceManager

DeviceManager is a core entity which is responsible for keeping devices and templates data models. It is also responsible for publishing any updates to all interested components through Kafka.

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

1.1.3 IoT Agent

An IoT agent is an adaptation service between physical devices and *dojot's* core components. It could be understood as a *device driver* for a set of devices. 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, Lora/ATC, Sigfox/WDN and HTTP/JSON.

Communication via secure channels with devices is also the responsibility of IoT agents.

1.1.4 Flowbroker (Flow builder)

This service provides mechanisms to build data processing flows to perform a set of actions. These flows can be extended using external processing blocks (which can be added using REST APIs).

1.1.5 Data Manager

This service manages the dojot's data **configuration**, making possible to import and export configuration like templates, devices and flows.

1.1.6 Cron

Cron is a dojot's microservice that allows you to schedule events to be emitted - or requests to be sent - to other microservices inside dojot platform.

1.1.7 Kafka2Ftp

The kafka2ftp service allows forwarding messages from Apache Kafka to FTP servers. It subscribes to Kafka's topic `tenant.dojot.ftp`, where the messages must follow a specific schema. Messages can be redirected to these topics using a specific node in the flowbroker.

1.1.8 Persister/History

The Persister component works as a pipeline for data and events that must be persisted on a database. The data is converted into a 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 persisted data can be queried through a Rest API provided by the History microservice.

1.1.9 InfluxDB Storer and Retriever

The services InfluxDB Storer and InfluxDB Retriever work together, the InfluxDB Storer is responsible for consuming Kafka data from dojot devices and writing it to InfluxDB, while the InfluxDB Retriever has the role of obtaining the data that were written by InfluxDB Storer in InfluxDB via API REST.

1.1.10 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.

1.1.11 GUI

The Graphical User Interface in *dojot* is responsible for providing responsive interfaces to manage the 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
- **Templates 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
- **Notifications**: View system notifications (unified real time and history)

1.1.12 Image manager

This component is responsible for device (firmware) image storage and retrieval. It is used by the firmware update mechanism.

1.1.13 X.509 Identity Management

This component is responsible for assigning identities to devices, such identities are represented in the form of x.509 certificates. It behaves similarly to a *Certificate Authority* (CA), where it is possible to submit a CSR and receive a certificate back. Once the certificate has been installed on the device, it is possible to communicate securely with the dojot platform, as the data collected by the device is transmitted over a secure (encrypted) channel and it is also possible to guarantee the integrity of the data.

1.1.14 Kafka WS

This component is responsible for retrieving data from Apache Kafka through pure WebSocket connections. It was designed to allow dojot users to retrieve realtime raw and/or processed data from dojot devices.

1.2 Infrastructure

A few extra components are used in dojot,they are:

- keycloak: an open source software product to allow single sign-on with Identity and Access Management aimed at modern applications and services.
- postgres: this database is used to persist data from many components, such as Device Manager.
- redis: in-memory database used as cache in many components, such as service orchestrator, subscription manager, IoT agents, and so on. It is very light and easy to use.
- rabbitMQ: message broker used in service orchestrator in order to implement action flows related that should be applied to messages received from components.
- mongo database: widely used database solution that is easy to use and doesn't add a considerable access overhead (where it was employed in dojot).
- zookeeper: keeps replicated services within a cluster under control.

1.3 Communications

All components communicate with each other in two ways:

- Using HTTP requests: if one component needs to retrieve data from other one, say an IoT agent needs the list of currently configured devices from Device Manager, it can send a HTTP request to the appropriate component.
- Using Kafka messages: if one component needs to send new information about a resource controlled by it (such as new devices created in Device Manager), the component may publish this data through Kafka. Using this mechanism, any other component that is interested in such information needs only to listen to a particular topic to receive it. Note that this mechanism doesn't make any hard associations between components. For instance, Device Manager doesn't know which components need its information, and an IoT agent doesn't need to know which component is sending data through a particular topic.

IOT AGENT ARCHITECTURE

This document describes the IoT agent architecture used by dojot. It defines a set of basic features and choices that must be followed in order to be aligned with dojot architecture.

2.1 Who should read this?

Developers that want to create new IoT agents to be used with dojot.

2.2 Introduction

Using dojot involves dealing with the following entities:

- **physical devices**: devices that send messages to IoT agents. They might have sensors and might be configurable, but this is not mandatory. Also, they must have some kind of connectivity to other services so that they can send their readings to these services.

- **users**: whoever sends requests to dojot in order to manage resources, retrieve historical device data, create subscriptions, manage flows, and so on.

- **tenants**: logical separation between resources that might be associated with multiple users.

- **resources**: elements that are associated to a particular entity. They are:

  - *devices*: representation of a element which has attributes. This element can be a physical device or a virtual one - one that doesn't receive attribute updates directly by a device.

  - *templates*: device blueprints that contain a list of attributes associated to that class of devices. All devices are created based on a template, from which it will inherit attributes.

  - *topics*: Kafka communication channels that are used to send and receive messages between dojot services.

  - *flows*: Sequence of processing blocks that are created by a user or an application and are used to analyze and pre process data.

- **subjects**: group of topics that share a common message flow. For instance, there might be many topics that are used to transmit device data. All of them belong to the same subject *device-data*.

When a new IoT agent is created, all these entities must be taken into account in a coordinated way. This document lists all basic requirements for a new IoT agent and they are categorized in the following groups:

1. **Device security**: IoT agents must be able to check whether a device connection is valid or not. A valid device connection is defined as one originated by a trusted physical device (or any representative element, such as gateways) which is allowed to connect to the IoT agent. A device is deemed as trusted by: (1) creating a device associated with it (which may include security information such as cryptographic keys) or (2) indicating directly

to IoT agent that a device or a representative element is allowed to connect to it (so that elements that serves as relay connections can be properly and securely used).

2. **Information context separation**: each resource (device, templates, topics and flows) is associated to a particular tenant and entities that don't belong to that tenant must not be allowed to access its resources. This is valid throughout dojot and it is no exception for IoT agents. Therefore, an IoT agent must treat separately all devices that belong to different tenants - including the fact that no one from one tenant should be able to know of the existence of other tenants. For instance, a MQTT IoT agent should not allow messages sent to its broker from devices associated to tenant A to be published to devices subscribed to the same topic belonging to tenant B.

3. **IoT agent information and management**: any IoT agent should publish its capabilities and information models. For instance, it should let other services know about what is the device template which it accepts in order to properly receive and send messages to a particular physical device. It should also offer a management interface so that a user can change and retrieve its behavior, such as logging options, statistics, quotas and so on.

4. **IoT agent operation**: IoT agents must be able to receive and send messages (if allowed by the protocol) to devices and, therefore, send updates to other dojot services based on received device messages. All messages received from a particular device and sent to other dojot services must be sent in the same order as it was received. IoT agents should also be able to enable or disable message processing from a particular device and detect device liveness.

   An extra feature that an IoT agent might implement is firmware updates. Depending on its underlying protocol, it might be possible to do such thing in an easy, secure and reliable way.

Each one of these groups is going to be detailed in the following sections.

2.3 Device security

An IoT manager should take into account the following aspects of device communication:

1. Device identity: it should only accept connections from authorized physical devices. The verification of whether a new connection was originated by an authorized device (which includes verifying whether a particular device is authorized or not) should rely on public keys and/or signed certificates.

2. Communication channel security: all messages exchanged with a physical device should be encrypted using well-known cryptographic standards, such as TLS. Any in-house security protocols should be avoided.

3. Certificate revocation: the IoT agent should be able to discard any messages from previously authorized device if its security data has been somehow compromised. For instance, if the private key associated to a particular device is leaked, then all its messages should be ignored as there is no guarantee that they came from that device.

Each of these aspects will be detailed in the following sections.

2.3.1 Device identity

The device identity verification is the starting point when dealing with communication security. This validation will indicate to the IoT agent if the device that opened the connection is whoever it says it is. Furthermore, the IoT agent must, once this validation succeeds, check whether this device can connect to it by checking its ID. This section will show how to do that.

For connection-oriented protocols, the IoT agent should only accept connections for devices that have a certificate that was signed by an authority that is trusted by dojot. Once this certificate is valid, device identity can be checked in two forms:

- Device ID encoded in certificate: although this is a less-reliable mechanism, it allows greater flexibility using many devices in a controlled deployment. This is based on setting the common name (CN certificate field) as dojot device ID. Therefore, IoT agent should check whether this device exists or not and allow or deny the

connection right away depending on this verification. The weak points of this mechanisms is that the device certificate must be signed by dojot's internal CA (once there is a procedure to sign only one certificate per device) and, if this certificate is valid, then its ID must also be valid. If any other CA is used, then this mechanism has no valid use.

- IoT agent has all valid certificates: if an administrator wants to use an external CA to sign all device certificates, then there is no actual control of which device ID was used to generate a particular certificate. Therefore, IoT agent must have all valid certificates properly mapped onto a device list - this will guarantee that only one certificate is allowed to a particular device and vice-versa.

Using the first mechanism, the device (or an operator configuring a device for the first time) must call dojot CA to generate a signed certificate for itself. There is no further action for IoT agent to take as long as dojot CA is used.

The second mechanism, however, requires that an IoT agent offer methods to manage certificates. The developer must take into account also that this IoT agent must be able to scale - these certificates must be accessible to all IoT agent instances, if allowed by deployment.

2.3.2 Communication security

With a valid certificate, a device can create a communication channel with dojot. For connection-oriented channels, this certificate should be used alongside cryptographic keys in order to provide an encrypted channel. For other channel types (such as channels for exchanging messages through a gateway, such as LoRa or sigfox), it suffice to be sure that the connection between dojot and the backend server is secure. The backend identity should be asserted beforehand. Once it is known to be trusted, all its messages can be processed with no major concern.

2.3.3 Certificate revocation

An IoT agent should be able to be informed about revoked certificates. It should expose an API or configuration messages to allow such thing. It should not allow any communication with a particular device that uses a revoked certificate.

2.4 Information context separation

A tenant could be thought simply as a group of users that share some resources. But its meaning might go beyond that - it might implies that these resources would not share any common infrastructure (considering anything that transmits, processes or stores data) with resources belonging to other tenants. One might want to have separate software instances to process data from different tenants so that processing data from one tenant will not affect processing data from the other, achieving a higher level of context separation.

Although this is desirable, some deployment scenarios might force using some of the same infrastructure for different tenants (for instance, when the deployment has as reduced numbers of processing units or network connections). So, in order to have a minimum context separation among tenants, an IoT agent should use everything it can to separate them, such as using different threads, queues, sockets, etc., and should not rely solely in deployment scenarios features (such as different IoT agents for different tenants). For instance, for topic based protocols, such as MQTT, one might want to force different topics for different tenants. Should a device publish data to a particular topic that is owned by other tenant, this message is ignored or blocked (sending an error back to the device might be an optional behavior). Therefore no device from one tenant can send messages to any device from other tenant.

The mechanism through which context separation is implemented highly depends on which protocol is used. A thorough analysis should be performed to properly implement this feature.

2.5 IoT agent information and management

An IoT agent should expose all the necessary information to use it properly. It should expose:

- **Device template**: an IoT agent should publish which is the data model it accepts for a valid device. This should be done by publishing a new device template to other dojot services. There should be a mechanism so that different instances of the same IoT agent publishes the same device template (including any template IDs). If the device template is updated in a newer version of an IoT agent, the device template ID should change.

- **Management APIs**: an IoT agent should be manageable and should expose its APIs to do that. The minimum set of management APIs that an IoT agent should offer are:

  - *Logging*: there should be a way to change the log level of an IoT agent;

  - *Statistics*: an IoT agent may expose an API to let a user or application retrieve statistical information about its execution. An administrator might want to switch on or off the generation of a particular statistical variable, such as processing time.

An IoT agent should also be able to gather statistics information related to its execution. Furthermore, it should let an administrator set quotas on those measured quantities. These quantities might include, but are not limited to:

- transmission statistics

  - number of received device messages from device (total, per device, per tenant)

  - number of published device messages to dojot (total, per device, per tenant)

  - number of messages sent to devices (total, per device, per tenant)

  - [optional] time taken between receiving a message from a physical device and publishing it (total - mean, per device - mean, per tenant - mean)

- IoT agent service health check - system statistics (memory, disk, etc.) used by the service

Many other values might be gathered. The list above is the minimum list that an IoT agent is expected to expose to other services. Particularly for health check, there is a document detailing how to expose it.

2.6 IoT agent operation

The main purpose of an IoT agent is to publish data from a particular device to other dojot services. Its operation is two fold: receive and process messages related to device management from other services as well as receive messages from the devices themselves (or their representative elements) and publish these data to other services.

The following sections describe how an IoT agent can send and receive messages to/from other dojot services and what are the considerations it must take into account when receiving messages from physical devices.

2.6.1 Messages

Tenants

It's necessary to subscribe to receive tenant events using the Kafka topic `dojot-management.dojot.tenancy`.

The Kafka topic `dojot-management.dojot.tenancy` will be used to receive tenant lifecycle events. Whenever a new tenant is created or deleted, the following message will be published:

Topic: *dojot-management.dojot.tenancy*	
Body format (JSON)	`type="CREATE"/"DELETE"` `tenant=>string`

A sample message received by this topic is:

```
{
  "type": "CREATE",
  "tenant": "new_tenant"
}
```

See more about *Bootstrapping tenants* in internal communication.

Subjects

The following subjects should be used by IoT agents:

- *dojot.device-manager.device*
- *device-data*

With the list of tenants, the IoT agent can request topics for receiving device lifecycle events and for publishing new device attribute data. This is done by sending the subjects for the following request to DataBroker:

Host: DataBroker	
Endpoint: /topic/{subject}	Method: GET
Request	
Headers	Authorization: Bearer ${JWT}
Response	
Headers	Content-Type: application/json
Body format	`topic => string`

A sample response for this request is:

```
{
  "topic": "admin.device-data"
}
```

Each one will be detailed in the following sections

dojot.device-manager.device

The topic related to this subject will be used to receive device lifecycle events for a particular tenant. Its format is:

Subject: *dojot.device-manager.device*	
| Body format (JSON) | ```
event => "create" / "update"
meta => service
  service => string
data =>
  id => string
  label => string
  templates => *number
  attrs => [*template_attrs]
  created => iso_date
``` |
| Body format (JSON) | ```
event => "remove"
meta =>
  service => string
data =>
  id => string
``` |
| Body format (JSON) | ```
event => "configure"
meta =>
  service => string
  timestamp => int (Unix Timestamp - ms)
data =>
  id => string
  attrs => *device_attrs
``` |

The *device_attrs* attribute is a even simpler key/value JSON, such as:

```json
{
  "temperature" : 10,
  "height" : 280
}
```

A sample message received by this topic is:

```json
{
  "event": "create",
  "meta": {
    "service": "admin"
  },
  "data": {
    "id": "efac",
    "label": "Device 1",
    "templates": [1, 2, 3],
    "attrs": {
      "1": [
        {
          "template_id": "1",
          "created": "2018-01-05T15:41:54.840116+00:00",
```

```
            "label": "this-is-a-sample-attribute",
            "value_type": "float",
            "type": "dynamic",
            "id": 1
        }
    ]
  },
  "created": "2018-02-06T10:43:40.890330+00:00"
 }
}
```

device-data

The topic related to this subject will be used to publish data retrieved from a physical device to other dojot services. Its format is:

Subject: *device-data*	

Body format (JSON)	`metadata => deviceid tenant timestamp`
	`  deviceid => string`
	`  tenant => string`
	`  timestamp => int (Unix Timestamp - ms`
	`→or s)`
	`attrs => *device_attrs`

The timestamp is associated to when the attribute values were gathered by the device (this could be done by the device itself - by directly sending the *timestamp* attribute in the message - or by the IoT agent, if no timestamp was defined by the device). The timestamp should be in UNIX or ISO formats, in milliseconds or seconds.

A sample message received by this topic is:

```
{
  "metadata": {
    "deviceid": "c6ea4b",
    "tenant": "admin",
    "timestamp": 1528226137452,
  },
  "attrs": {
    "humidity": 60
  }
}
```

See more about *Sending Kafka messages* in internal communication.

2.6.2 Firmware update

An IoT agent might implement mechanisms in order to update firmware in devices.

2.7 Behavior

The order in which a physical device sends its attributes must not be changed when IoT agent publishes these data to other dojot services.

If the protocol imposes any unique ID to each device, the IoT agent must build a correlation table to properly translate this unique ID into dojot device ID and vice-versa.

2.8 Libraries to assist the development of new IotAgents

We have libraries that abstract some points described in previous topics to facilitate the development of an IotAgent.

There are two libraries:

- node.js **recommended** (https://www.npmjs.com/package/@dojot/iotagent-nodejs)

- java (https://jitpack.io/#dojot/iotagent-java)

CONCEPTS

This document provides information about dojot's concepts and abstractions.

Note:

- **Audience**

    - Users that want to take a look at how dojot works;

    - Application developers.

- Level: basic

3.1 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. We will focus on explaining what devices, models and flows are in dojot.

If you want more information on how dojot works internally, you should checkout the *Architecture* to get acquainted with all internal components.

3.1.1 User authentication

The dojot platform is integrated with keycloak. Each keycloak realm is equivalent to one dojot tenant. See more in *Using Keycloak in dojot*.

3.1.2 Device authentication

Device authentication is based on the use of asymmetric cryptographic keys and is done through x.509 certificates managed by dojot. For that, it is necessary to install such a certificate on the device. Dojot has a certificate issuing service where it is possible to obtain a certificate to be installed on the device.

In addition to the certificate and asymmetric keys, the device must *trust* the dojot *Certificate Authority*, that is, it is also necessary to install the root certificate of the dojot platform.

The certificate is requested by to which the devices are registered. Once the administrator follows the necessary steps to obtain the certificate, (s)he will have to install the issued certificate and the root certificate of the dojot on the device. It is important to emphasize that the certificate has an asymmetric cryptographic key in its composition, this key is called a public key (which anyone has access to), while the private key must be installed on the device (a key that only the device has access to).

Once the device contains the private key, the certificate (containing the public key) and also the root certificate of the dojot, it is possible to establish a secure communication channel with the dojot platform, in which the device is identified through its certificate.

After the secure channel is established, the device is able to publish data and also receive data as long as it is authorized to do so.

Check *Using MQTT with security (TLS)* for more information on its usage.

3.1.3 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, CoAP device, etc.

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 one or more *templates*, 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.

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 (implemented in terms of data flows, build in flowbuilder) 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 .

The *DeviceManager* documentation on GitHub ReadMe explains in more depth all the available operations. You can find the link in *Components and APIs*.

3.1.4 Flows

A flow is a sequence of blocks that process a particular event or device message. It contains:

- entry point: a block representing what is the trigger to start a particular flow;

- processing blocks: a set of blocks that perform operations using the event. These blocks may or may not use the contents of such event to further process it. The operations might be: testing content for particular values or ranges, geo-positioning analysis, changing message attributes, perform operations on external elements, and so on.

- exit point: a block representing where the resulting data should be forwarded to. This block might be a database, a virtual device, an external element, and so on.

The component responsible for dealing with such flows is flowbroker.

Check *Using flow builder* for more information on its usage.

COMPONENTS AND APIS

4.1 Components

Table 4.1: Components

Component	Repository / Main site	Documentation for Component	Component API Documentation
Keycloak	Keycloak site	Keycloak doc.	
MongoDB	MongoDB site	MongoDB doc.	
Postgres	PostgreSQL site	PostgreSQL doc.	
Kong API gateway (Community Edition)	Kong site	Kong doc.	
Redis	Redis site	Redis doc.	
Zookeeper	Zookeeper site	Zookeeper doc.	
Kafka	Kafka site	Kafka doc.	
VerneMQ	VerneMQ site	VerneMQ doc.	
Leshan	Leshan site	Leshan doc.	
InfluxDB	InfluxDB site	InfluxDB doc.	
Dojot Kong	GitHub - Dojot Kong		
History	GitHub - history		API - history
Device Manager	GitHub - DeviceManager		API - DeviceManager
Image Manager	GitHub - image-manager		API - image-manager
GUI	GitHub - GUI		
GUI - V2	GitHub - V2		
Flowbroker	GitHub - flowbroker		API - flowbroker
Databroker	GitHub - data-broker		API - data-broker
IotAgent VerneMQ (MQTT) - **Default**	GitHub - iotagent-vernemq		
IotAgent Mosca (MQTT) - **Legacy**	GitHub - iotagent-mosca		
IotAgent Leshan (LWM2M)	GitHub - iotagent-leshan		
Data Manager	GitHub - Data Manager		API - Data Manager

4.2 Exposed APIs (API Gateway)

The API gateway used in dojot reroutes some of endpoints from component. The following table shows which **Exposed endpoint by the API gateway** is mapped to which **component endpoint**, its **component endpoint Documentation** and whether the endpoint **needs authentication** when used via API Gateway. See more about how using APIs in *Using API interface*.

Table 4.2: Exposed endpoints

Component	Exposed endpoint by the API gateway	Component Endpoint	Component Endpoint Documentation	Needs Authentication
GUI	/	/		No
Dashboard	/v2	/		No
Device Manager	/device	/device	API - DeviceManager	Yes
Device Manager	/template	/template	API - DeviceManager	Yes
Flowbroker	**/flows**	/	API - flowbroker	Yes
History	**/history**	/	API - history	Yes
Data Manager	/import	/import	API - Data Manager	Yes
Data Manager	/export	/export	API - Data Manager	Yes
Cron	/cron	/cron	API - Cron	Yes
Image Manager	**/fw-image**	/	API - image-manager	Yes
Data Broker	/device/ {deviceID} /latest	/device/ {deviceID} /latest	API - data-broker	Yes
Data Broker	/subscription	/subscription	API - data-broker	Yes
Data Broker	/stream	/stream	API - data-broker	Yes
Data Broker	/socket.io	/socket.io	API - data-broker	No
X.509 Identity Management	**/x509/v1**	**/api/v1**	API - x509-identity-mgmt	Yes
Kafka WS	**/kafka-ws/v1/ticket**	/v1/ticket	API - kafka-ws	Yes
Kafka WS	**/kafka-ws/v1**	/v1	API - kafka-ws	No
InfluxDB Retriever	/tss/v1/	/tss/v1/	API - InfluxDB-Retriever	Yes
InfluxDB Retriever - Documentation	/tss/v1/api-docs	/tss/v1/api-docs	API - InfluxDB-Retriever	No
Backstage	/backstage/v1	/backstage/v1	API - Backstage	No
Keycloak	/auth	/auth	Keycloak doc.	No

NOTE: Some of the components' endpoints aren't exposed, but are used internally.

In addition, the API gateway reroutes the endpoints with their ports from component, so that they become uniform: all of them are accessible through the same port (default is TCP port 8000), see the following table.

Table 4.3: Original endpoints to The API gateway

Component	Original endpoint	Gateway Endpoint
GUI	host:80/	host:8000/
Dashboard	host:80/	host:8000/v2
Device Manager	host:5000/device	host:8000/device
Device Manager	host:5000/template	host:8000/template
Flowbroker	host:80/	host:8000/flows
History	host:8000/	host:8000/history
Data Manager	host:3000/import	host:8000/import
Data Manager	host:3000/export	host:8000/export
Cron	host:5000/cron	host:8000/cron
Image Manager	host:5000/	host:8000/fw-image
Data Broker	host:80/device/{{deviceID}}/latest	host:8000/device/{deviceID}/latest
Data Broker	host:80/subscription	host:8000/subscription
Data Broker	host:80/stream	host:8000/stream
Data Broker	host:80/socket.io	host:8000/socket.io
X.509 Identity Management	host:3000/api/v1	host:8000/x509/v1
Kafka WS	host:8080/v1/ticket	host:8000/kafka-ws/v1/ticket
Kafka WS	host:8080/v1/topics	host:8000/kafka-ws/v1/topics
InfluxDB Retriever	host:3000/tss/v1/	host:8000/tss/v1/
InfluxDB Retriever - Documentation	host:3000/tss/v1/api-docs	host:8000/tss/v1/api-docs
Backstage	host:3000/backstage/v1	host:8000/backstage/v1
Keycloak	host:8080/auth	host:8000/auth

4.3 Libraries

Dojot also has several libraries used in their own components. These libraries are listed below:

Table 4.4: Component Libraries by Language

Components	Language	Libraries
Module	Python	https://github.com/dojot/dojot-module-python https://pypi.org/project/dojot.module/
Module	Java	https://github.com/dojot/dojot-module-java https://jitpack.io/#dojot/dojot-module-java
Module	Node JS	https://github.com/dojot/dojot-module-nodejs https://www.npmjs.com/package/@dojot/dojot-module
IoT Agent	Java	https://github.com/dojot/iotagent-java https://jitpack.io/#dojot/iotagent-java
IoT Agent	Node JS	https://github.com/dojot/iotagent-nodejs https://www.npmjs.com/package/@dojot/iotagent-nodejs
Module Logger	Node JS	https://github.com/dojot/dojot-module-logger-nodejs https://www.npmjs.com/package/@dojot/dojot-module-logger
Helthcheck	Node JS	https://github.com/dojot/healthcheck-nodejs https://www.npmjs.com/package/@dojot/healthcheck
Microservice SDK	Node JS	https://github.com/dojot/dojot-microservice-sdk-js https://www.npmjs.com/package/@dojot/microservice-sdk

4.4 Kafka messages

These are the messages sent by components and their subjects. If you are developing a new internal component (such as a new IoT agent), see API - data-broker to check how to receive messages sent by other components in dojot.

Table 4.5: Original endpoints

Component	Message	Subject
DeviceMan-ager	Device CRUD (Messages - DeviceManager)	`dojot.device-manager.device`
device data	Device data (*IoT agent*)	`device-data`
Keycloak	Tenants creation/removal (*Emitted messages from custom keycloak provider*)	`dojot.tenancy`

INTERNAL COMMUNICATION

This page describes how each service in dojot communicate with each other.

5.1 Components

The main components that are currently in dojot platform are Fig. 5.1.
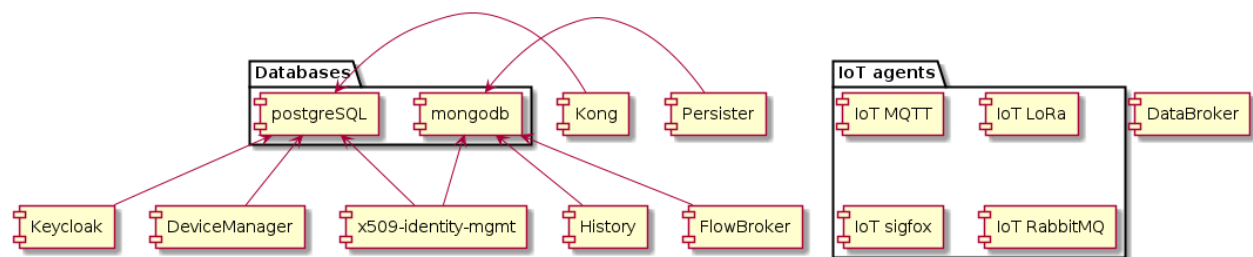
Fig. 5.1: dojot components

They are:

- Keycloak: an open source identity and access management solution.

- DeviceManager: device and template storage.

- Persister: component that stores all device-generated data.

- History: component that exposes all device-generated data.

- DataBroker: deals with subjects and Kafka topics, as well as socket.io connections.

- Flowbroker: handles flows (both CRUD and flow execution)

- IoT agents: agents for different protocols.

Some services will be briefly described in this page. More information can be found in each component documentation.

5.2 Messaging and authentication

There are two methods through which dojot components can talk to each other: via HTTP REST requests and via Kafka. They are intended for different purposes, though.

HTTP requests can be sent at boot time when a component want, for instance, information about particular resources, such as list of devices or tenants. For that, they must know which component has which resource in order to retrieve them correctly. This means - and this is a very important thing that drives architectural choices in dojot - that only a single service is responsible for retrieving data models for a particular resource (note that a service might have multiple instances, though). For example, DeviceManager is responsible for storing and retrieving information model for devices and templates, FlowBroker for flow descriptions, History for historical data, and so on.

Kafka, in the other hand, allows loosely coupled communication between instances of services. This means that a producer (whoever sends a message) does not know which components will receive its message. Furthermore, any consumer doesn't know who generated the message that it being ingested. This allows data to be transmitted based on "interests": a consumer is interested in ingesting messages with a particular *subject* (more on that later) and producers will send messages to all components that are interested in it. Note that this mechanism allows multiple services to emit messages with the same "subject", as well as multiple services ingesting messages with the same "subject" with no tricky workarounds whatsoever.

5.2.1 Sending HTTP requests

In order to send requests via HTTP, a service must *Getting Access Token (Access Token JWT)*. What is shown Fig. 5.2 is after obtaining authorization. Note: the actual authorization mechanism is detailed in *Keycloak + API gateway (Kong)*.
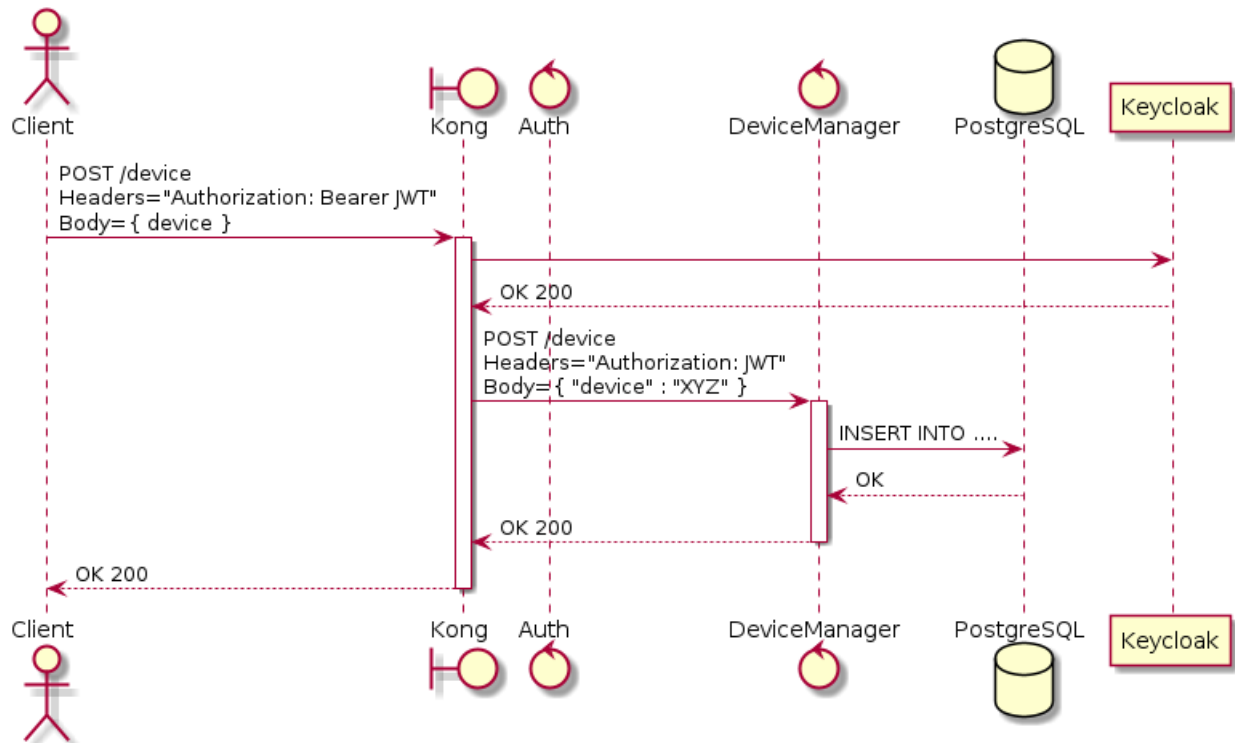
Fig. 5.2: Sending messages to HTTP API

In this figure, a client creates a new device using the token retrieved. This request is analyzed by Kong, which will

invoke Keycloak to check whether the user set in the token is allowed to `POST` to `/device` endpoint. Only after the approval of such request, Kong will forward it to DeviceManager.

5.2.2 Sending Kafka messages

Kafka uses a quite different approach. Each message should be associated to a subject and a tenant. This is show in Fig. 5.3;
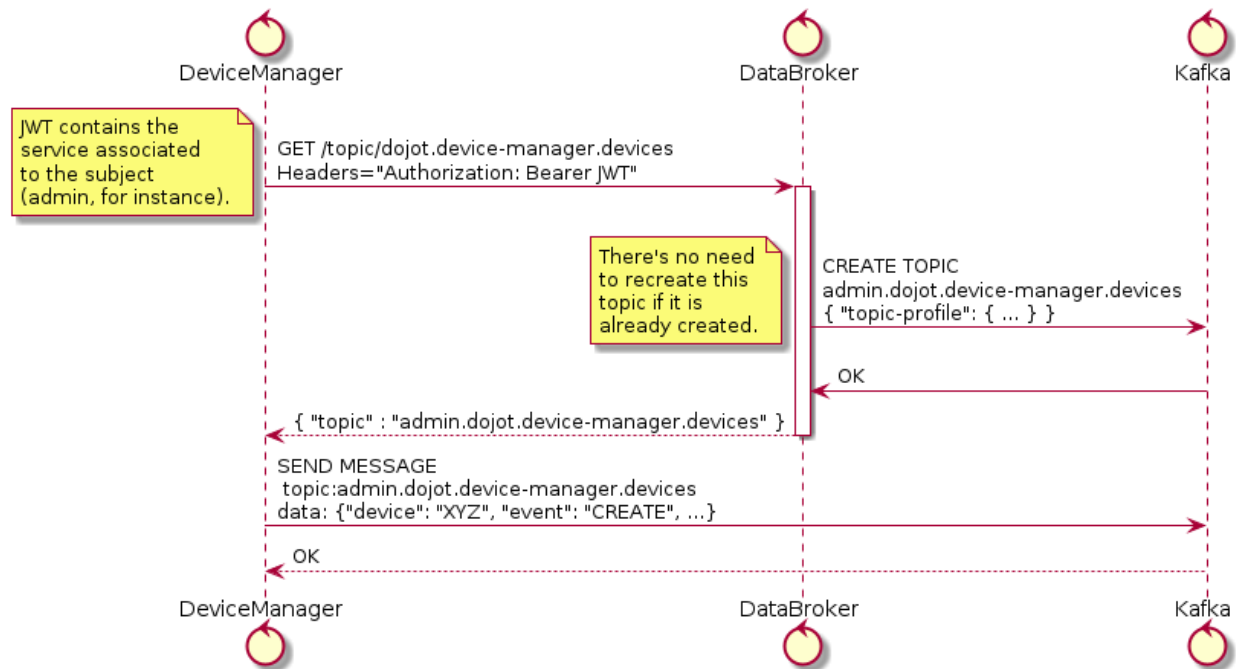
Fig. 5.3: Retrieving Kafka topics

In this example, DeviceManager needs to publish a message about a new device. In order to do so, it sends a request to DataBroker, indicating which tenant (within JWT token) and which subject (`dojot.device-manager.devices`) it wants to use to send the message.

To better understand how it all works, you can check the *Data Broker* documentation for the component and API, the links are in *Components and APIs*.

5.2.3 Bootstrapping tenants

All components are interested in a set of subjects, which will be used to either send messages or receive messages from Kafka. As dojot groups Kafka topics and tenants into subjects (a subject will be composed by one or more Kafka topics, each one transmitting messages for a particular tenant), the component must bootstrap each tenant before sending or receiving messages.

Its purpose is to process all messages received through Kafka subscribing in `dojot-management.dojot.tenancy`. Our custom keycloak provider that publishes tenant creation and deletion messages to kafka. This will include any tenant that is created after all services are up and running. Fig. 5.4 shows how to deal with these messages.
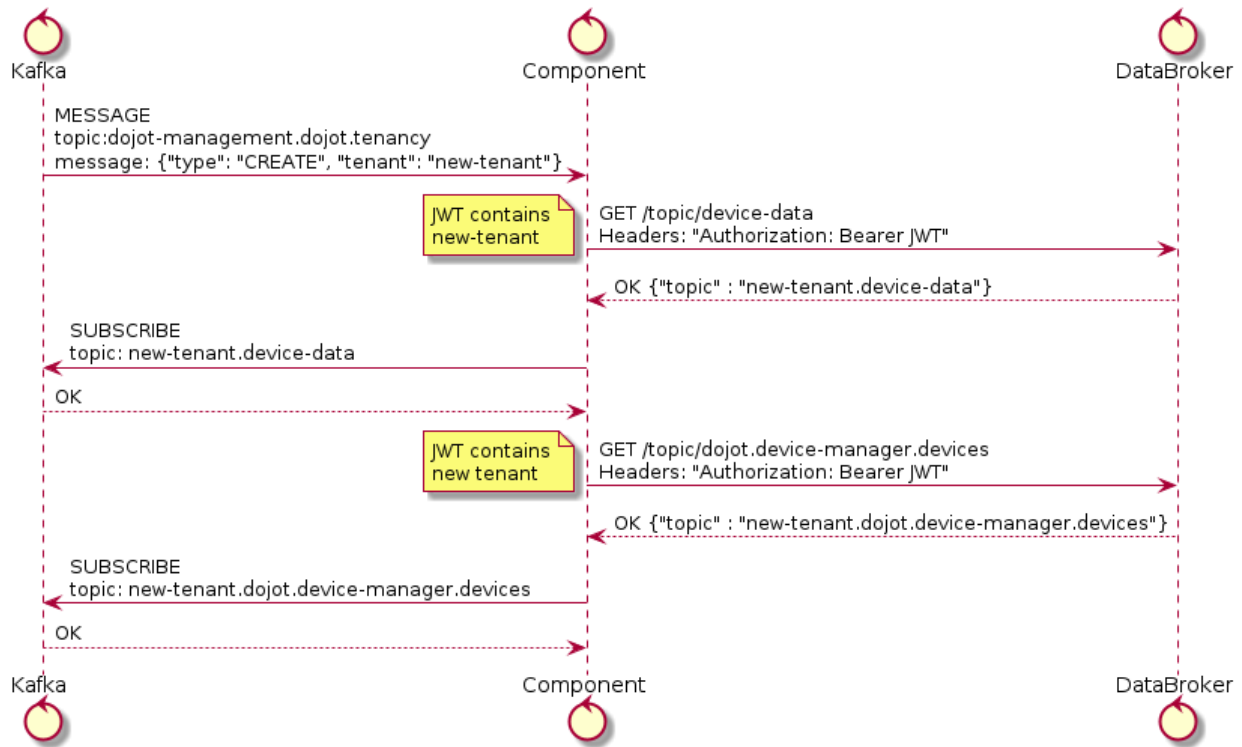
Fig. 5.4: Tenant bootstrapping

All services that are somehow interested in using subjects should execute this procedure in order to correctly receive all messages.

5.3 Keycloak + API gateway (Kong)

Keycloak is a service deeply connected to Kong. It is responsible for user management, authentication and authorization. As such, it is invoked by Kong whenever a request is received by one of its registered endpoints. This section will detail how this is performed and how they work together. See more at Kong + Keycloak and Keycloak Dojot Provider.

5.3.1 Emitted messages from custom keycloak provider

Keycloak Dojot Provider will emit just one message via Kafka for tenant creation:

```
{
  "type" : "CREATE",
  "tenant" : "XYZ"
}
```

And one for tenant deletion:

```
{
  "type" : "DELETE",
  "tenant" : "XYZ"
}
```

By default these messages are created in kafka topic `dojot-management.dojot.tenancy`.

5.4 Device Manager

DeviceManager stores and retrieves information models for devices and templates and a few static information about them as well. Whenever a device is created, removed or just edited, it will publish a message through Kafka. It depends only on DataBroker and Kafka for reasons already explained in this document.

The *DeviceManager* documentation on GitHub ReadMe explains in more depth all messages published. You can find the link in *Components and APIs*.

5.5 IoT agent

IoT agents receive messages from devices and translate them into a default message to be published to other components. In order to do that, they might want to know which devices are created in order to properly filter messages which are not allowed into dojot (using, for instance, security information to block messages from unauthorized devices). It will use the `device-data` subject and bootstrap tenants as described in *Bootstrapping tenants*.

After requesting the topics for all tenants within *device-data* subject, IoT agent will start receiving data from devices. As there are a plethora of ways by which devices can do that, this step won't be detailed in this section (this is highly dependent on how each IoT agent works). It must, though, send a message to Kafka to inform other components of all new data that the device just sent. This is shown in Fig. 5.5, in this case we are using the tenant *admin*.
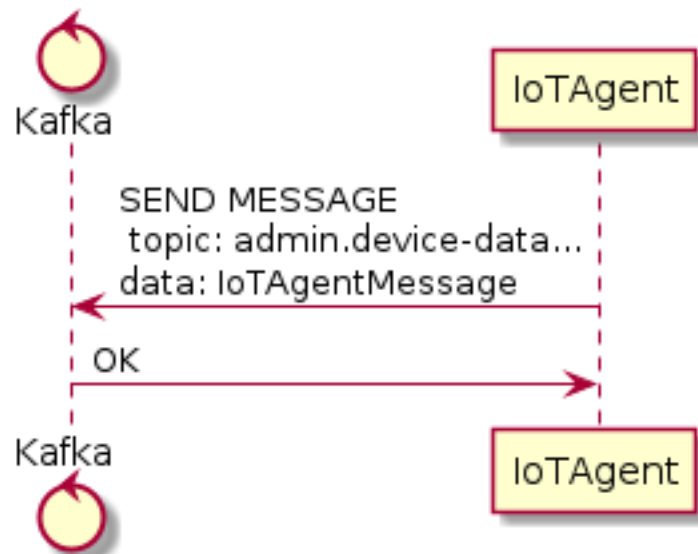
Fig. 5.5: IoT agent message to Kafka

The data sent by IoT agent has the structure shown in Fig. 5.6.

Such message would be:

```
{
    "metadata": {
        "deviceid": "c6ea4b",
        "tenant": "admin",
        "timestamp": 1528226137452
    },
    "attrs": {
```
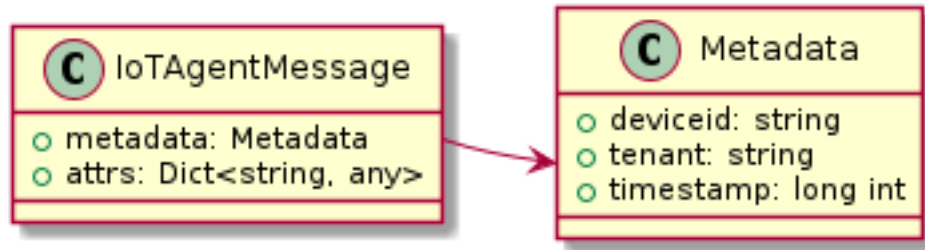
Fig. 5.6: IoT agent message structure

```
        "humidity": 60,
        "temperature" : 23
    }
}
```

5.6 Data Broker

DataBroker has a few more functionalities than only generating topics for {tenant, subject} pairs. It will also serve socket.io connections to emit messages in real time. In order to do so, it retrieves all topics for *device-data* subject, just as in any other component interested in data received from devices. As soon as it receives a message, it will then forward it to a 'room' (using socket.io vocabulary) associated to the device and to the associated tenant. Thus, all client connected to it (such as graphical user interfaces) will receive a new message containing all the received data. For more information about how to open a socket.io connection with DataBroker, check DataBroker documentation in *Components and APIs*.

Note: The real time socket.io connections via Data Broker will be discontinued in future releases. Use *Kafka WS* instead.

5.7 Certificate authority

The dojot has an internal *Certificate Authority* (CA) capable of issuing x.509 certificates so that devices can communicate with the platform through a secure channel (using the TLS protocol). When requesting a certificate for the platform, it is necessary to inform a CSR, which will go through a series of validations until arriving at the internal Certificate Authority, which, in turn, if all checks pass successfully, will sign a certificate and link this certificate to the device registration. The *x509-identity-mgmt* component is responsible for providing certificate-related services for devices.

5.8 Kafka WS

The *Kafka WS* service allows users to retrieve conditional and/or partial real time data from a given dojot topic in a Kafka cluster. It works with pure websocket connections, so you can create websocket clients in any language you want as long as they support RFC 6455.

5.8.1 Connecting to the service

The connection is done in two steps: you must first obtain a single-use ticket through a HTTP request, then connect to the service via websocket passing it as a parameter.

First step: Get the single-use ticket

A ticket allows the user to subscribe to a dojot topic. To obtain it is necessary to have a JWT access token that is issued by the platform's Authentication/Authorization service. The ticket must be retrieved via a HTTP request using the GET verb to the *<base-url>/kafka-ws/v1/ticket* endpoint. The request must contain the *Authorization* header with the previously retrieved JWT token as a value. Example:

GET <base-url>/kafka-ws/v1/ticket
Authorization: Bearer [Encoded JWT]

The component responds with the following syntax:

HTTP/1.1 200 OK
Content-type: application/json

```
{
  "ticket": "[an opaque ticket of 64 hexadecimal characters]"
}
```

Note: In the context of a dojot deployment the JWT Token is provided by the Auth service, and is validated by the API Gateway before redirecting the connection to the *Kafka WS*. So, no validations are done by the Kafka WS.

Second step: Establish a websocket connection

The connection is done via pure websockets using the URI *<base-url>/kafka-ws/v1/topics/:topic*. You must pass the previously generated ticket as a parameter of this URI. It is also possible to pass conditional and filter options as parameters of the URI.

5.8.2 Behavior when requesting a ticket and a websocket connection

Below we can understand the behavior of the Kafka WS service when a user (through an user agent) requests a ticket in order to establish a communication via websocket with Kafka WS.

Note that when the user requests a new ticket, Kafka WS extracts some information from the *user's access token* (JWT) and generates a *signed payload*, to be used later in the decision to authorize (or not) the websocket connection. From the payload a *ticket* is generated and both are stored in Redis, where the ticket is the key to obtain the payload. A TTL is defined by Kafka WS, so the user has to use the ticket within the established time, otherwise, Redis automatically deletes the ticket and payload.

After obtaining the ticket, the user makes an HTTP request to Kafka WS requesting an upgrade to communicate via *websocket*. As the specification of this HTTP request limits the use of additional headers, it is necessary to send the ticket through the URL, so that it can be validated by Kafka WS before authorizing the upgrade.

Since the ticket is valid, that is, it corresponds to an entry on Redis, Kafka WS retrieves the payload related to the ticket, verifies the integrity of the payload and deletes that entry on Redis so that the ticket cannot be used again.

With the payload it is possible to make the decision to authorize the upgrade to websocket or not. If authorization is granted, Kafka WS opens a subscription channel based on a specific topic in Kafka. From there, the upgrade to websocket is established and the user starts to receive data as they are being published in Kafka.
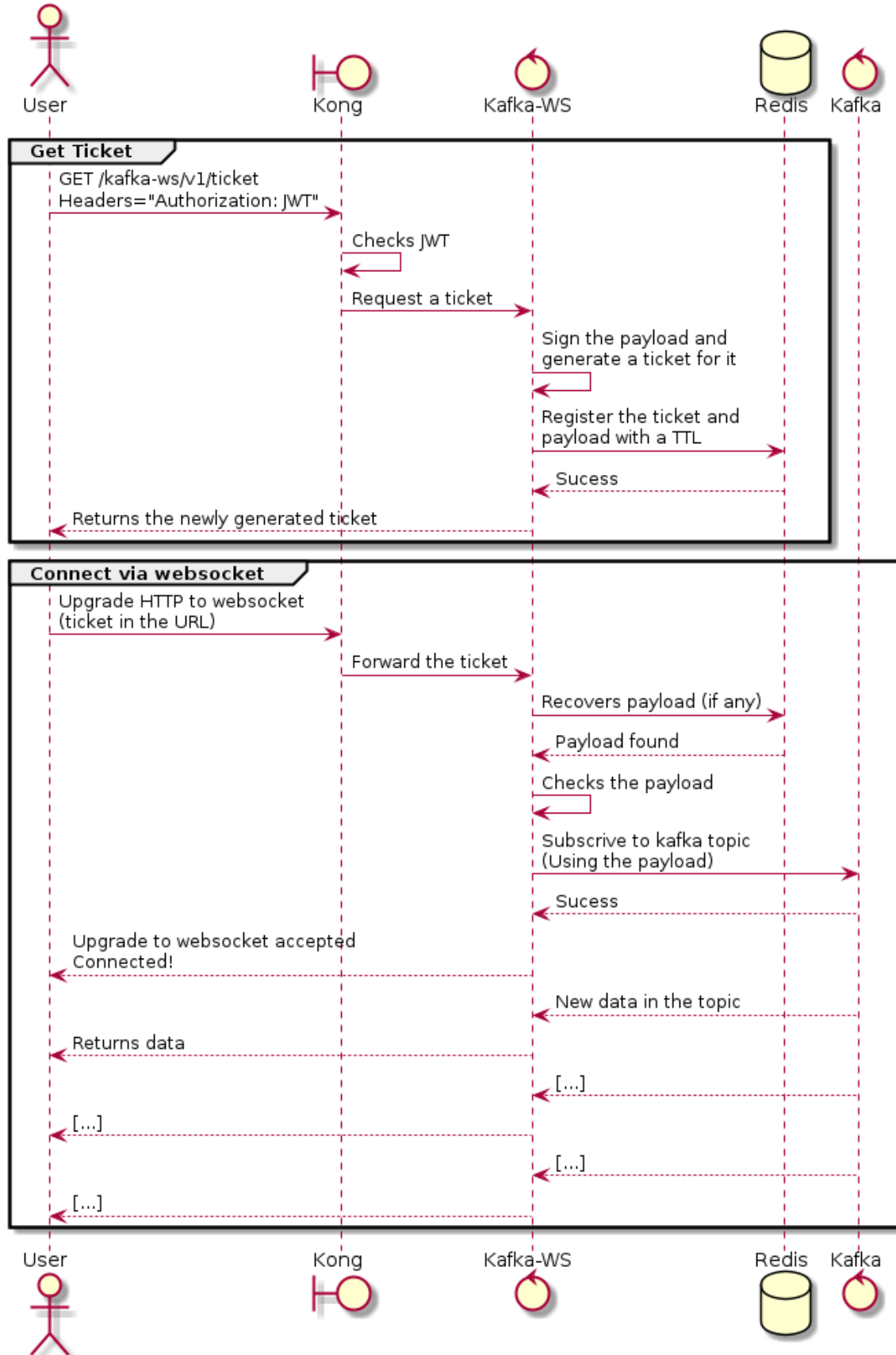
Fig. 5.7: Obtaining a ticket and connecting via websocket

INSTALLATION GUIDE

This page contains information about how to deploy dojot using Docker Compose and Kubernetes.

6.1 Hardware requirements

When choosing hardware configurations for Dojot *deployment*, the number of devices and the message interval they will be configured with should be considered. As an example, the estimated hardware requirements for 500 devices with a message interval every 15s are:

Table 6.1: Hardware requirements for 500 devices

Deployment		CPU	RAM	Free disk space
Docker Compose		4 Cores	6GB	20GB
Kubernetes	Master	2 Cores	2GB	30GB
Kubernetes	Worker	4 Cores	6GB	40GB
Kubernetes	Balancer	1 Core	1GB	10GB

In addition, you need:

- **Network access**

- **The following ports should be opened for Docker Compose deployment:**

    - **TCP**: 80 *(web interface access)*; 1883 *(MQTT, If you are going to use MQTT)*; 5896 *(LWM2M, If you are going to use LWM2M file server via HTTP instead of coap, UDP)*.

- **TLS**: 443 *(web interface access)*; 8883 *(MQTTS, If you are going to use MQTT with TLS, in secure mode.)*.

- **UDP**: 5683 and 5693 *(LWM2M, If you are going to use LWM2M)*; 5684 and 5694 *(LWM2M, If you are going to use LWM2M with DTLS)*.

- **The following ports should be opened on load balancer for Kubernetes deployment:**

  - **TCP**: 80 *(web interface access)*; 1883 *(MQTT, If you are going to use MQTT)*; 5896 *(LWM2M, If you are going to use LWM2M file server via HTTP instead of coap, UDP)*.

  - **TLS**: 443 *(web interface access)*; 8883 *(MQTTS, If you are going to use MQTT with TLS, in secure mode.)*.

  - **UDP**: 5683 and 5693 *(LWM2M, If you are going to use LWM2M)*; 5684 and 5694 *(LWM2M, If you are going to use LWM2M with DTLS)*.

Note: The above cores are approximately 3.5 GHz (x86-64)

6.2 Docker Compose

6.2.1 Instructions

This document provides instructions on how to create a trivial deployment environment on single host for *dojot*, using 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 18.04 and 20.04 LTS environment.

The following sections describe all Docker Compose dependencies.

6.2.2 Docker Engine

This guide has been checked using Docker Engine 19.03

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

https://docs.docker.com/engine/install/ubuntu/

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 Compose commands should be run by the super user (root), or as sudo.

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

6.2.3 Docker Compose

This guide has been checked using Compose 1.27

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

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

6.2.4 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 https://github.com/dojot/docker-compose.git
# Let's move into the repo - all commands in this page should be executed
# inside it.
cd docker-compose
```

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

```
# Must be run from within the deployment repo

git checkout tag_name -b branch_name
```

For instance:

```
git checkout v0.8.0 -b v0.8.0
```

That done, the environment can be brought up by following this documentation docker-compose dojot.

Stop environment without cleaning volumes:

```
docker-compose down
```

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/

6.2.5 Volumes

When we deploy dojot with the command 'docker-compose up -d' the volumes are enabled and created by default.

The volumes of microservices that Dojot uses can be incompatible between dojot versions. This means that you are unable to use dojot v0.4.x volumes in dojot v0.5.x or above and vice versa.

To use different versions of dojot in the same environment, you must first drop the volumes of the other version.

Note: If you drop the dojot volumes you will also lose all data that you have collected on the platform so far.

To drop the volumes just pass the '-v' parameter in the 'docker-compose down' command as displayed below:

```
docker-compose down -v
```

That way volumes and dojot will be dropped and you will be able to deploy a different dojot version.

6.2.6 Usage

Read the *Using API interface*, *Using Keycloak in dojot* and *Using web interface* for more information about how to interact with the platform.

6.3 Kubernetes

For simple installation with kubernetes please check the pdf below.

```
click here to access the dojot installation guide with kubernetes
```

If you want to install a more robust Dojot that supports up to 100k devices, check the pdf below.

Note: In the 100k environment, dojot does not process or store messages sent by devices. This environment will only work for load tests and only a few dojot components will be available.

```
click here to access the dojot 100k installation guide with kubernetes
```

Note: Unfortunately in this tutorial we do not have support for the English language yet.

FREQUENTLY ASKED QUESTIONS

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

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

Table of Contents

7.1 General

7.1.1 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.

The dojot acts as a facilitating 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 quickly prototype IoT solutions.

• Real time event processing with customizable rules.

7.1.2 Where can I get it?

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

7.1.3 Which repository is the main one?

There are 3 main ones:

- https://github.com/dojot/dojot: this is where we keep track of all the things related to this project as issues and there are also some component codes here.

- https://github.com/dojot/docker-compose: repository with the files and settings to perform dojot in a Docker environment using the *docker-compose* tool. This is the repository we recommend to start with dojot.

- https://github.com/dojot/ansible-dojot: repository with files and settings to perform dojot in *Kubernetes* environment.

See how to use *docker-compose* and *ansible-dojot* repositories in *Installation Guide*.

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

7.2 Usage

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

Dojot can be accessed by a web-based interface and by REST APIs. Considering that you installed `docker` and `docker-compose` and cloned the `docker-compose` repository, starting it up is done by just one command:

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

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.

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

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

7.2.3 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
```

If you need another version, you could checkout a tag instead:

```
$ git tag
  0.1.0-dojot
  0.1.0-dojot-RC1
  0.1.0-dojot-RC2
  0.2.0
  v0.3.0-beta.1
  v0.3.1
  v0.4.0
  v0.4.1
  v0.4.1_rc2
  v0.4.2
  v0.4.2-rc.1
  v0.4.3
  v0.4.3-rc.1
  v0.4.3-rc.2
  v0.5.0-alpha.1
  v0.5.0-alpha.2
  v0.5.0-alpha.3
  v0.5.0-alpha.4
  v0.5.0-rc.1
  v0.5.0
  v0.5.1
  v0.5.2
  v0.6.0
  v0.7.0
  v0.8.0


$ git checkout v0.8.0
```

7.3 Devices

7.3.1 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 thermal and humidity sensors; it can be represented inside 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 a higher level of temperature information (*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*.

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

It is as 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.

7.3.3 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*? Therefore, 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.

7.3.4 And what are *templates*?

Templates, simply put, are "blueprints for devices" which serve as basis to create a new device. A single device is built using a set of templates - its attributes will be inherited from each template (their names must not be exactly the same, though). If one template is changed, then all associated devices will also be changed.

7.3.5 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.

> **Attention:** Since version **v0.5.2**, you can choose the between two MQTT brokers: Mosca or VerneMQ. By default, VerneMQ is used, but you can use Mosca too. Check the *Installation Guide* for more information.

In order to send data to dojot via MQTT (using Mosca or VerneMQ), there are some things to keep in mind:

- When using Mosca, the topic should look like `/<tenant>/<device-id>/attrs` (e.g.: `/admin/efac/attrs`). Depending on how IoT agent MQTT was started (more strict), the client ID must also be set to `<tenant>:<device-id>`, such as `admin:efac`.
- When using VerneMQ, the topic should look like `<tenant>:<device-id>/attrs` (e.g.: `admin:efac/attrs`). You must also set the username for the client as `<tenant>:<device-id>`, such as `admin:efac`, and it should match the same part in the topic. You can also set the client ID too (not required).
- MQTT payload must be a JSON with each key being an attribute of the dojot device, such as:

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

You can use certificates with MQTT, check *Using MQTT with security (TLS)* for more information.

7.3.6 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`, `Text` and `JSON`: table.

- `Integer` and `Float`: line chart.

7.3.7 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.

7.3.8 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?*.

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

For now, you can send HTTP requests to dojot containing a few instructions about which device should be configured and the actuation payload itself. More details on that can be found *Sending messages*.

7.3.10 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 IotAgent, a connector, similar to the existing ones for MQTT, CoAP and HTTP. Take a look at https://github.com/dojot/iotagent-nodejs.

7.3.11 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...' \
  "http://localhost:8000/history/device/3bb9/history?lastN=3&attr=temperature"
```

which will retrieve the last 3 entries of *temperature* attribute from the device *3bb9*:

```
[
  {
    "device_id": "3bb9",
    "ts": "2018-03-22T13:47:07.050000Z",
    "value": 29.76,
    "attr": "temperature"
  },
```

```
{
    "device_id": "3bb9",
    "ts": "2018-03-22T13:46:42.455000Z",
    "value": 23.76,
    "attr": "temperature"
},
{
    "device_id": "3bb9",
    "ts": "2018-03-22T13:46:21.535000Z",
    "value": 25.76,
    "attr": "temperature"
}
]
```

For more details on data retrieval from the history, check the tutorial in *Checking historical data*.

In addition, there are more operators that could be used to filter entries. Look for the History API in *Components and APIs* documentation to check out all possible operators and other filters.

7.4 Data Flows

7.4.1 What is data flow?

It's a sequence of functional blocks to process incoming device messages. With a flow you can dynamically analyze each new message in order to apply validations, infer information and trigger actions or notifications.

7.4.2 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, it won't be difficult to use it.

7.4.3 Why should I use it?

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

7.4.4 What can it do, exactly?

You can do things such as:

- Create views from a particular device, by renaming, aggregating and changing values, etc).
- Infer information based on switch, edge-detection and geo-fence rules.
- Notify through many ways, like HTTP.

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

There are mechanisms to add new processing blocks to new flows. Check the *How can I add a new node type to its menu?* question for more information on that.

7.4.5 So, how can I use it?

For more details on how to use flows, check the tutorial in *Using flow builder*.

7.4.6 Can I apply the same flow to multiple devices?

You can use a template as input to indicate that the flow should be applied to all devices associated to that template. It's worth to point out that the flow is processed individually for each new input message, i.e. for each input device.

7.4.7 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.

You can also create a node (or use an already existing one) that deals with contexts.

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

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

7.4.9 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.

7.4.10 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.

7.4.11 How can I add a new node type to its menu?

There is a tutorial on how to add new nodes and two examples of node too, check the flowbroker library for more details.

7.5 Applications

7.5.1 What APIs are available for applications?

You can check all available APIs in the *Components and APIs*.

7.5.2 How can I use them?

There is a very quick and useful tutorial in the *Using API interface*.

7.5.3 I'm interested in integrating my application with dojot. How can I do it?

This should be pretty straightforward. There are three 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 the *history* API

- **Retrieving real time data**: you might want to read device-related real time data. This can be done by using *Kafka WS*, a *websocket* based implementation. To better understand how to use *Kafka WS* check *Kafka WS*.

- **Using flowbroker to pre-process data**: for more flexible ways of data manipulation, you can use flows. They can process/transform data so they can be properly sent to your application via HTTP request, or stored in a virtual device (which can be used to generate notifications as previously described).

All these endpoints should bear an access token, see more *Getting access token*.

Check the documentation API for *History* and *Kafka WS* in *Components and APIs*. And to a tutorial on how to use the flow, check *Using flow builder*.

COPYRIGHT AND LICENSE

The dojot IoT Platform is based on well-known open source software such as Apache Kafka, RabbitMQ, PostgreSQL, MongoDB, Redis, Kong Gateway, Keycloak, and VerneMQ, which have their own licenses.

The services developed by the dojot team to integrate these open-source software components and implement the business logics are copyrighted by CPQD and till release v0.5.1 were licensed under the GPLv3. From release v0.5.2 onwards, these services are being licensed under the Apache License, Version 2.0.

You can use and/or modify the software for free (no license costs apply).

RELEASE HISTORY

9.1 Gatka - 2021.08

9.1.1 Services

New Services

Keycloak

- The old service *auth* has been removed and we now use Keycloak.

Improvements and fixes

- Changes were made to several services to make them work with keycloak.

Others

- Minor bug fixes

9.1.2 Deployments

Docker-compose

- Documentation improvements
- Minor bug fixes

Ansible-dojot

- Documentation improvements
- Volumes feature
- Labels feature
- Docker log files rotation
- Update Kubernetes version to 1.19.8
- Minor bug fixes

9.1.3 Libraries

- Changes were made to dojo-module-nodejs, iot agent-nodejs, dojot-module-python, dojot-module-java, iotagent-jav libs to work with keycloak.

USING WEB INTERFACE

This tutorial will show how to do basic operations in dojot, such as creating devices, checking its attributes and creating flows, import/export, firmware update, generating certificates and device history report.

Note:

- Who is this for: entry-level users

- Level: basic

- Reading time: 20m

10.1 Device management

This section will show how to manage device. For this tutorial we will show how to add two thermometers and a virtual device that will represent an alarm system that will monitor both sensors.

As described in *Concepts*, all devices are based on a template. To create one, you should access the template tab at the left and then create one new template, as shown below.

To create new devices, you should go back to the devices tab and create a new device, selecting the templates it will be based on, as shown below.

Note that, when you select the template in the right panel at device creation screen, all attributes are inherited from that device. You could add more templates as needed, keeping in mind that templates used to compose a device must not share an attribute with the same name.

Attention: As devices are tightly associated to templates, if you want to remove a template, you should remove all its associated devices first. If such thing happens the following error message will appear:
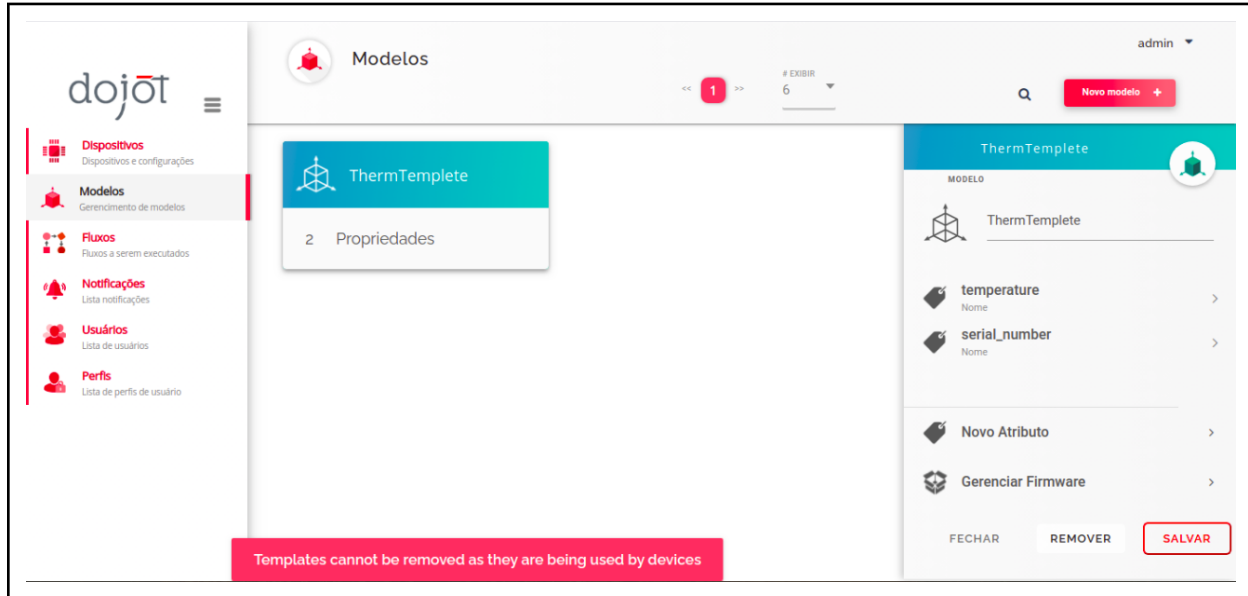
Attention:

> You can add and remove attributes from templates and they will be immediately available to devices. In case of new attributes being added, though, you should keep in mind that there must not be any device with templates which have attributes with same name. If such thing happens, the following message will appear:
>
>
>
> This snapshot was generated by creating a new template (`ExtraTemplate`) with one attribute, called `level`. Then a new device based on both templates was created and, afterwards a new attribute also called `level` was added to `ThermTemplate`.
>
> When this happens, no modification is applied to the template (no attribute named "level" related to the "ThermTemplate" is created). However, it remains in the template configuration so the user can figure out what is happening. If the user refreshes the page, it will be reverted to what it was before the modification.
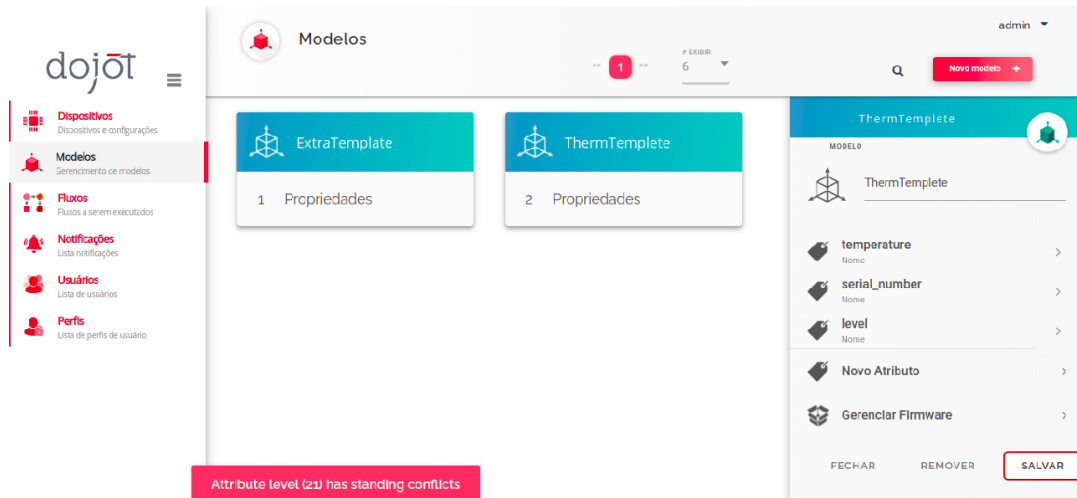
Now the physical devices can send messages to dojot. There are few things to pay attention to: the MQTT topic is `<tenant>:<device-id>/attrs`.

For simplicity's sake, we'll emulate one device using mosquitto_pub tool. We set the `username` parameter by using the `-u` flag of mosquitto_pub. See more about in *Using API interface* at topic Sending Messages.

> **Attention:** In the videos we are using Mosca as the MQTT broker, so the messages are being sent to topics in the old format. If you are using VerneMQ, change them according to the forementioned pattern. Check *Frequently Asked Questions* for more info about the differences between them.

Note: The examples are using insecure MQTT. The recommended approach is to use TLS. Check the section *Using MQTT with security (TLS)*.

Now that we've created the sensors, let's create a virtual one. This will be the representation of a alarm system that will be triggered whenever something bad is detected to these sensors. Let's say they are installed in a kitchen. So it is expected that their temperature readings will be no more than 40C. If it is more than that, our simple detection system will conclude that the kitchen is on fire. This alarm representation will have two attributes: one for a severity level for a particular alarm and another one for a textual message, so that the user is properly informed of what's happening.

Just as for "regular devices", virtual devices also are based on templates. So, let's create one, as shown below.

10.2 Flow configuration

Once we've created the virtual device, we can add a flow to implement the logic behind the alarm generation. The idea is: if the temperature reading is less than 40, then the alarm system will be updated with a message of severity 4 (mildly important) and a message indicating that the kitchen is OK. Otherwise, if the temperature is higher than 40, then a message is sent with severity 1 (highest severity) and a message indicating that the kitchen is on fire. This is done as shown below.

Note that the "change" nodes have a reference to an "output" entity. This can be thought as a simple data structure - it will have a `message` and a `severity` attributes that match those from the virtual device. This "object" is referenced in the output node as a data source for the device to be updated (in this case, the virtual device we've created). In other words, you can think of this as a piece of information carried from "change" nodes to the "virtual device" with names "msg.output.message" and "msg.output.severity", where "message" and "severity" are the virtual device attributes.

So, let's send a few more messages and see what will happen to that virtual device.

10.3 Import and Export

This section shows how to use the Import and Export features. These options allow your configuration data to be saved to a file, for Export, and loaded to dojot, for Import. This file has the JSON format and it contains the data from templates, devices, flows, remote nodes, and scheduling tasks that were entered in your tenant. To perform data configuration export procedure, expand the menu at the top right of the page, click "Import / Export" and then "Export" as shown below:

The exported file can be stored as a backup and later imported back into Dojot.

To perform data configuration import procedure, expand the menu in the upper right corner of the page, click "Import / Export" and then "Import." In the window that appears it is possible to drag and drop your file or browse to the destination folder and select it. It is only allowed to add a JSON extension file, in the expected format, as illustrated in the following video:

> **Attention:** When performing the import procedure all current tenant configuration, such as: devices, templates, flows, remote nodes and scheduling tasks, will be permanently deleted, so that new ones are created. History data is not part of importing and exporting!

10.4 Firmware update

During the lifetime of a device, you may need to update control software (firmware) to correct some issues you encounter while using it, or even add new features. Dojot currently supports the firmware upgrade procedure via the LwM2M communication protocol. For details regarding the procedure for integrating with your device please check the LwM2M protocol specification. If your device communicates via this protocol and has the firmware update procedure in place, you can follow the steps below to update your device version.

The firmware upgrade process consists of three steps:

- image management;
- image transfer to device;
- image application on device;

The details of their implementation are as follows.

In order to enable the firmware management you must create a template and, once saved, enable the firmware manager. After that, you can upload the firmware images to the dojot repository that are associated with this template. Attention: the image extension must be ".hex".

Note that when Firmware Manager is enabled, five attributes are assigned to the template. They are used to support image updating. Attribute names can be edited as required by the application. The attributes are:

- Device State:
    - Current state of firmware update
- Result of apply version:
    - Contains the result of downloading or updating the firmware
- Sets which version to transfer:
    - Indicates to the IoT agent responsible for the device, what is the name and version of the firmware image to be downloaded and updated on the device
- Trigger version update:
    - Actuator used to initiate firmware update procedure
- Current version of the image:
    - Current version of the firmware image, provided by the device

After you create the template with the Firmware management option enabled, you can associate it with a device. So, you can then transfer an image and apply it to the device, as shown in the video below:

Note that in each step, the status and result of image processing are shown.

10.5 Generating certificates for devices

This section will show how to generate x509 certificates for a device, so that dojot can communicate with devices securely via TLS. To be able to send a publication, it is necessary to download the three files, the private key "admin 4302d4.key", the device certificate "admin 4302d4.crt" and the CA certificate "ca.crt".

> **Attention:** The generation of certificates via the graphical interface (GUI) only works in deployments where it is possible to access the GUI via *HTTPS* or *localhost*.

To be able to execute the command of the example video it is necessary to be in the same directory as the 3 files attached. Command used in the example:

```
mosquitto_pub -h localhost -p 8883 -t admin: 4302d4/attrs -m '{"humidity": 7}' --cert
→"admin 4302d4.crt" --key "admin 4302d4.key" --cafile ca.crt
```

10.6 Generating device history report

This section will demonstrate how to generate a device history report. The report is able to show data for one or more attributes of the respective device. For this, it is necessary to select the desired attributes, define the period and click on "generate".

10.7 Performing access to the Dashboard

The dashboard is part of GUI-V2, to access it you need to use another URL, in the current version just add to the end of the URL "/v2" in relation to the interface used in the previous items, for example, in the case of localhost it would be http://localhost:8000/v2, see more at *Components and APIs*. The login and password credentials are the same as those used in the rest of dojot. After logging in, a new screen will open and an ADD button will appear in the upper right corner, which will give you the options for various types of viewing. At this moment, in the first "General" screen it will be necessary to add a name for the visualization and optionally a description. The list of devices will appear on the next screen, if you do not find the desired device, you can search by name. After selecting the device, the attributes that are linked to it will be listed, and a color can be chosen to display each attribute, it is also possible to add a caption for each attribute. In "retrieve records by:" it will be possible to configure some filters, you can select the type of historical data filter as the "last records", in "order" (minute, hours, days and months), and you can also choose an interval time, in addition you can still view the records in "Real time", as new data is received they will be displayed in the chosen view. After everything is configured, a summary will be shown with the chosen name and attributes. When accessing the views, it will be possible to change the size, fix (which will disable the option to change the size) and still delete the views.

> **Note:** Since v0.8 there have been some changes to the login screen, but they are not present in the video below.

ELEVEN

USING API INTERFACE

This section provides a complete step-by-step tutorial of how to create, update, send messages to and check historical data of a device. Also, this tutorial assumes that you are using docker-compose, which has all the necessary components to properly run dojot.

Note:

- Audience: developers

- Level: basic

- Reading time: 15 m

11.1 Prerequisites

It uses:

- curl to access the dojot platform APIs;

- jq to process the return JSON from the dojot platform APIs.

- mosquitto to publish and subscribe in iotagent-mosca (or iotagent-mqtt) via MQTT.

In Debian-based Linux distributions you can run:

```
sudo apt-get install curl
sudo apt-get install jq
sudo apt-get install mosquitto-clients
```

11.2 Getting access token

All requests must contain a valid access token. See ways to get a token in *Getting Access Token (Access Token JWT)*.

To check the token:

```
echo $JWT
```

The token ("eyJ0eXAiOiJKV1QiL. . .") inside $JWT 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 an environment variable called `${JWT}`, which contains the token got from auth component.

11.3 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 documentation 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"
    },
    {
      "label": "fan",
      "type": "actuator",
      "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            "template_id": "1",
28            "created": "2018-01-25T12:30:42.167126+00:00",
29            "label": "fan",
30            "type": "actuator",
31            "value_type": "float",
32            "id": 2
33          }
34        ],
35        "id": 1
36      }
37    }
```

Note that the template ID is 1 (line 35), if you have already created another template this id will be different.

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

```
curl -X POST http://localhost:8000/device \
-H "Authorization: Bearer ${JWT}" \
-H 'Content-Type:application/json' \
-d ' {
  "templates": [
    "1"
  ],
  "label": "device"
}'
```

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:

```
1    {
2      "pagination": {
3        "has_next": false,
4        "next_page": null,
5        "total": 1,
6        "page": 1
7      },
8      "devices": [
9        {
10          "templates": [
11            1
12          ],
13          "created": "2018-01-25T12:36:29.353958+00:00",
14          "attrs": {
15            "1": [
16              {
17                "template_id": "1",
18                "created": "2018-01-25T12:30:42.167126+00:00",
19                "label": "temperature",
20                "value_type": "float",
21                "type": "dynamic",
```

```
22              "id": 1
23            },
24            {
25              "template_id": "1",
26              "created": "2018-01-25T12:30:42.167126+00:00",
27              "label": "fan",
28              "value_type": "actuator",
29              "type": "float",
30              "id": 2
31            }
32          ]
33        },
34        "id": "0998", # <-- this is the device-id
35        "label": "device_0"
36      }
37    ]
38  }
```

The *device-id* used in the next steps must be changed as returned in the creation of the device. In the above run the *id*
returned was *0998* on line 34. Therefore, all places in the next steps with references to *0998* must be changed.

11.4 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 publish messages to dojot with all its attributes and their current values. For this tutorial we will publish
MQTT messages by hand to the platform, emulating such physical device. For that, we will use mosquitto_pub and
mosquitto_sub from mosquitto.

The default message format used by dojot is a simple key-value JSON (you could translate any message format to this
scheme using flows, though), such as:

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

> **Attention:** Some Linux distributions, Debian-based ones 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 too. In this tutorial, **only the tools from package `mosquitto-
> clients` on Debian-based Linux distributions are going to be used**. Please check if another MQTT broker **is
> not running** before starting dojot (by running commands like ps aux | grep mosquitto) to avoid port
> conflicts.

For simplicity's sake, we are not using TLS in the examples below. Check *Using MQTT with security (TLS)* for more
information on its usage.

> **Note:** To run *mosquitto_pub* and *mosquitto_sub* without using TLS as in the examples below, you need to config-
> ure some settings (or for how to disable the mode without TLS). For more details on this topic, please refer to the
> *Unsecured mode MQTT (without TLS)* page.

Since version **v0.5.2**, you can choose the between two MQTT brokers: Mosca or VerneMQ. By default, VerneMQ is
used, but you can use Mosca too. Check the *Installation Guide* for more information.

11.4.1 Using VerneMQ

As noted in the *Frequently Asked Questions*, there are some considerations regarding MQTT topics:

- You must set the username that originates the message using the `username` MQTT parameter. It should follow the following pattern: `<tenant>:<device-id>`, such as `admin:efac`. It must match the tenant and device ID set in the topic.

- The topic to publish messages has the pattern `<tenant>:<device-id>/attrs` (e.g.: `admin:efac/attrs`).

- The topic to subscribe should has the pattern `<tenant>:<device-id>/config` (e.g.: `admin:efac/config`).

- MQTT payload must be a JSON with each key being an attribute of the dojot device, such as:

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

Let's publish the following message:

```
mosquitto_pub -h localhost -p 1883 -u admin:0998 -t admin:0998/attrs -m '{"temperature
→": 10.6}' -q 1
```

If there is no output, the message was sent to MQTT broker.

Note that we publish a message with the parameter `-q 1`. This means that the message will use QoS 1, i.e., the message is guaranteed to be send at least one time.

Also you can send a configuration message from dojot to the device to change some of its attributes. The target attribute must be of type "actuator".

To simulate receiving the message on a device, we can use `mosquitto_sub`:

```
mosquitto_sub -h localhost -p 1883 -u admin:0998 -t admin:0998/config -q 1
```

Triggering the sending of the message from the dojot to the device.

```
curl -X PUT \
    http://localhost:8000/device/0998/actuate \
    -H "Authorization: Bearer ${JWT}" \
    -H 'Content-Type:application/json' \
    -d '{"attrs": {"fan" : 100}}'
```

11.4.2 Using Mosca (legacy)

As noted in the *Frequently Asked Questions*, there are some considerations regarding MQTT topics:

- You can set the device ID that originates the message using the `client-id` MQTT parameter. It should follow the following pattern: `<tenant>:<device-id>`, such as `admin:efac`.

- If you can't do such thing, then the device should set its ID using the topic used to publish messages. The topic should assume the pattern `/<tenant>/<device-id>/attrs` (e.g.: `/admin/efac/attrs`).

- The topic to subscribe should assume the pattern `/<tenant>/<device-id>/config` (e.g.: `/admin/efac/config`).

- MQTT payload must be a JSON with each key being an attribute of the dojot device, such as:

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

> **Attention:** VerneMQ is the new default MQTT broker. Support for Mosca will be eventually dropped, so use VerneMQ if possible!

Let's send a message to dojot:

```
mosquitto_pub -h localhost -t /admin/0998/attrs -p 1883 -m '{"temperature": 10.6}'
```

If there is no output, the message was sent to MQTT broker.

Also you can send a configuration message from dojot to the device to change some of its attributes. The target attribute must be of type "actuator".

To simulate receiving the message on a device, we can use `mosquitto_sub`:

```
mosquitto_sub -h localhost -p 1883 -t /admin/0998/config
```

Triggering the sending of the message from the dojot to the device.

```
curl -X PUT \
    http://localhost:8000/device/0998/actuate \
    -H "Authorization: Bearer ${JWT}" \
    -H 'Content-Type:application/json' \
    -d '{"attrs": {"fan" : 100}}'
```

> **Note:** For the rest of the tutorial we will treat as if you are using VerneMQ.

11.5 Checking historical data

In order to check all values that were sent from a device for a particular attribute, you could use the history api, see more in *Components and APIs*. Let's first send a few other values to dojot so we can get a few more interesting results:

```
mosquitto_pub -h localhost -p 1883 -u admin:0998 -t admin:0998/attrs -m '{"temperature
↪": 36.5}' -q 1
mosquitto_pub -h localhost -p 1883 -u admin:0998 -t admin:0998/attrs -m '{"temperature
↪": 15.6}' -q 1
mosquitto_pub -h localhost -p 1883 -u admin:0998 -t admin:0998/attrs -m '{"temperature
↪": 10.6}' -q 1
```

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

```
curl -X GET \
  -H "Authorization: Bearer ${JWT}" \
  "http://localhost:8000/history/device/0998/history?lastN=3&attr=temperature"
```

The history endpoint is built from these values:

- `.../device/0998/...`: the device ID is `0998` - this is retrieved from the `id` attribute from the device

- `.../history?lastN=3&attr=temperature`: the requested attribute is temperature and it should get the last 3 values.

  The request should result in the following message:

```
[
  {
    "device_id": "0998",
    "ts": "2018-03-22T13:47:07.050000Z",
    "value": 10.6,
    "attr": "temperature"
  },
  {
    "device_id": "0998",
    "ts": "2018-03-22T13:46:42.455000Z",
    "value": 15.6,
    "attr": "temperature"
  },
  {
    "device_id": "0998",
    "ts": "2018-03-22T13:46:21.535000Z",
    "value": 36.5,
    "attr": "temperature"
  }
]
```

This message contains all previously sent values.

USING FLOW BUILDER

This tutorial will show how to properly use flow builder to process messages and events generated by devices.

Note:

- Who is this for: entry-level users

- Level: basic

- Reading time: 20 min

Table of Contents

12.1 Dojot nodes

- *Geofence*

- *Get Context*

- *Merge data*

- *Cumulative sum*

- *Publish in FTP topic*

- *Deprecated nodes*

    - *Device in*

    - *Device template in*

    - *Device out*

    - *Actuate*

12.1.1 Device Event in

event device
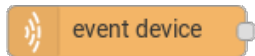
This node specifies that messages received from or sent to a particular device. The message created by this node is somewhat different than the one created by DeviceIn node.

To configure the device in node, a window like Fig. 12.1 will be displayed.

Fig. 12.1: : Device in configuration window

Fields:

- **Name** *(optional)*: Name of the node

- **Device** *(required)*: The *dojot* device that will trigger the flow

- **Events** *(required)*: Select which events will trigger this flow. The *Actuation* option will select actuation messages (those sent to the device) and *Publication* will select all messages published by the device.

Messages examples generated by this node:

For a `Publication` event:

```json
{
    "data": {
        "attrs": {
            "temperature": 10,
            "static_example": "static_example_value"
        },
        "id":"9face8"
    },
    "metadata": {
        "timestamp":1623163659936,
        "tenant":"admin"
    },
    "event": "publish"
}
```

For a `Actuation` event:

```json
{
    "data": {
        "attrs": {
            "temperature": 10,
            "static_example": "static_example_value"
        },
        "id":"9face8"
    },
    "metadata": {
        "timestamp":1623163659936,
        "tenant":"admin"
    },
    "event": "configure"
}
```

Note: In the *Publication* and *Actuation* events the static attributes will also always be a key in json inside the `data.attrs` key with its respective **label** and **value**

This structure can be referenced in nodes like *Template* and others such as:

```
Sample message {{payload.data.attrs.temperature}}
```

Note: If the the device that triggers a flow is removed, the flow won't work anymore.

12.1.2 Device template event in

This node will specifies that messages from devices composed by a particular template will trigger this flow. For instance, if the device template set in this node is template A, all devices that are composed with template A will trigger the flow. For example: *device1* is composed by templates [A,B], *device2* by template A and *device3* by template B. Then, in that scenario, only messages from *device1* and *device2* will initiate the flow, because template A is one of the templates that compose those devices.

Fields:

Fig. 12.2: : Device template in configuration window

- **Name** *(optional)*: Name of the node.

- **Device** *(required)*: The *dojot* device that will trigger the flow.

- **Events** *(required)*: Select which event will trigger this flow. *Creation*, *Update*, *Removal* are related to device management operations. *Actuation* will trigger this flow in case of sending actuation messages to the device and *Publication* will trigger this flow whenever a device publishes a message to dojot.

Messages examples generated by this node:

For a `Publication` event:

```
{
    "data": {
        "attrs": {
            "temperature": 10,
            "static_example": "static_example_value"
        },
        "id":"9face8"
    },
    "metadata": {
        "timestamp":1623163659936,
        "tenant":"admin"
    },
    "event": "publish"
}
```

For a `Actuation` event:

```
{
    "data": {
        "attrs": {
            "temperature": 10,
            "static_example": "static_example_value"
        },
```

(continues on next page)

```
        "id":"9face8"
    },
    "metadata": {
        "timestamp":1623163659936,
        "tenant":"admin"
    },
    "event": "configure"
}
```

For a `Creation` event (creating a device using that template):

```
{
    "data": {
        "label":"template_name"
        "id":"9face8"
        "templates":["1"]
        "created":"2021-06-08T14:46:27.008321+00:00"
        "attrs": {
            "temperature": {
                "id":1,
                "value_type":"float",
                "static_value":"",
                "type":"dynamic",
                "template_id":"1",
                "created":"2021-06-08T14:33:24.330779+00:00",
                "is_static_overridden":false
            }
        },
    },
    "metadata": {
        "tenant":"admin"
    },
    "event": "create"
}
```

For a `Update` event (Generated by changing a template or adding a template to an existing device):

```
{
    "data": {
        "label":"template_name"
        "id":"9face8"
        "templates":["1"]
        "created":"2021-06-08T14:46:27.008321+00:00"
        "updated":"2021-06-08T14:51:16.003916+00:00"
        "attrs": {
            "temperature": {
                "id":1,
                "value_type":"float",
                "static_value":"",
                "type":"dynamic",
                "template_id":"1",
                "created":"2021-06-08T14:33:24.330779+00:00",
                "is_static_overridden":false
            }
        },
    },
    "metadata": {
```

```
        "tenant":"admin"
    },
    "event": "update"
}
```

For a `Removal` event (When removing a template from a device):

```
{
    "data": {
        "label":"template_name"
        "id":"9face8"
        "templates":["1"]
        "created":"2021-06-08T14:46:27.008321+00:00"
        "updated":"2021-06-08T14:51:16.003916+00:00"
        "attrs": {
            "temperature": {
                    "id":1,
            },
        },
    },
    "metadata": {
        "tenant":"admin"
    },
    "event": "remove"
}
```

Note: To better understand the JSON within the *data* key for the *Creation*, *Update*, *Removal* events see the *device-manager* documentation.

Note: In the *Publication* and *Actuation* events the static attributes will also always be a key in json inside the `data.attrs` key with its respective **label** and **value**

This structure can be referenced in nodes like *Template* and others such as:

```
Sample message {{payload.data.attrs.temperature}}
```

12.1.3 Multi device out

Device out will determine which device (or devices) will have its attributes updated on *dojot* according to the result of the flow. Bear in mind that this node doesn't send messages to your device, it will only update the attributes on the platform. Normally, the chosen device out is a *virtual device*, which is a device that exists only on *dojot*.

Fields:

- **Name** *(optional)*: Name of the node.

- **Action** *(required)*: **Which device will receive the update. Options are:**

    - *The device that triggered the flow*: this will update the same device that sent the message which triggered this flow.

    - *Specific device(s)*: which device(s) that will receive the update.

Fig. 12.3: : Device out config window

- *Device(s) defined during the flow*: which device(s) that will receive the update. This is referenced by a list of values, just as with output values (msg.list_of_devices).

- **Device** *(required)*: Select "The device that triggered the flow" will make the device that was the entry-point be the end-point of the flow. "Specific device" any chosen device wil be the output of the flow and "a device defined during the flow" will make a device that the flow selected during the execution the endpoint.

- **Source** *(required)*: Data structure that will be mapped as message to device out
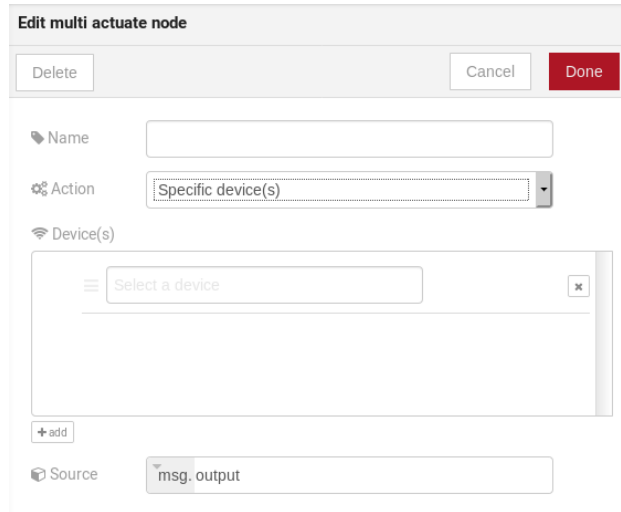
12.1.4 Multi actuate
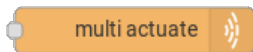
Actuate node is, basically, the same thing of **device out** node. But, it can send messages to a real device, like telling a lamp to turn the light off and etc.

Fields:

- **Name** *(optional)*: Name of the node.

- **Action** *(required)*: **which device a message will be sent to. Options are:**

    - *The device that triggered the flow*: this will send a message to the same device that sent the message which triggered this flow.

    - *Specific device(s)*: which device(s) the message will be sent to.

    - *Device(s) defined during the flow*: which device(s) the message will be sent to. This is referenced by a list of values, just as with output values (msg.list_of_devices).

- **Device** *(required)*: Select "The device that triggered the flow" will make the device that was the entry-point be the end-point of the flow. "Specific device" any chosen device wil be the output of the flow and "a device defined during the flow" will make a device that the flow selected during the execution the endpoint.

- **Source** *(required)*: Data structure that will be mapped as message to device out

Fig. 12.4: : Actuate configuration

12.1.5 HTTP request
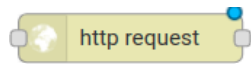
This node sends an http request to a given address, and, then, it can forward the response to the next node in the flow.

Fig. 12.5: : Http Request configuration

Fields:

- **Method** *(required)*: The http method (GET, POST, etc. . .).

- **URL** *(required)*: The URL that will receive the http request

- **Request body** *(required)*: Variable that contains the request body. This value can be assigned to the variable using the **template node**, for example.

- **Response** *(required)*: Variable that will receive the http response.

- **Return** *(required)*: Type of the return.

- **Name** *(optional)*: Name of the node.

12.1.6 Ftp request
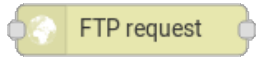
This node sends a file to a FTP server. When uploading a file, its name can be set by the "Filename" field in the same way as other output variables (it should refer to a variable set in the flow). The file encoding can also be set to, for example, "base64" or "utf-8".

Fig. 12.6: : Device template in configuration window

Fields:

- **Method** *(required)*: The FTP action to be taken (PUT).
- **URL** *(required)*: the FTP server
- **Authentication** *(required)*: Username and password to access this server.
- **File name** *(required)*: Variable containing the file name to be uploaded.
- **File content** *(required)*: This variable should hold the file content.
- **File encoding** *(required)*: How the file is encoded
- **Response** *(optinal)*: Variable that will receive the FTP response
- **Name** *(optional)*: Name of the node.

12.1.7 Notification

This node sends a user notification to other services in dojot. This might be useful to generate application notifications that could be consumed by a front-end application. The user can set a static message to be sent or, as other output nodes, configure a variable set in a previous node which will be resolved at runtime. Also, metadata can be added to the message: it can be a simple key-value object containing arbitrary data.

Fig. 12.7: : Device template in configuration window

Fields:

- **Name** *(optional)*: Name of the node

- **Message** *(required)*: Static, if the notification should contain a static text, or dynamic, which will enable a variable to be set as output to this node. This variable will be substituted in runtime.

- **Value** *(required)*: message content (either static text or a variable reference).

- **Metadata** *(required)*: variable reference containing a simple dictionary (key-value pairs) containing the metadata to be added to the message

12.1.8 Change
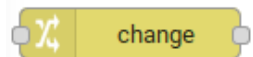
Change node is used to copy or assign values to an output, i. e., copy values of a message attributes to a dictionary that will be assigned to virtual device.

Fields:

- **Name** *(optional)*: Name of the node

- **msg** *(required)*: Definition of the data structure that will be sent to the next node and will receive the value set on the *to* field

- **to** *(required)*: Assignment or copy of values

Note: More than one rule can be assign by clicking on +*add* below the rules box.

Fig. 12.8: : Change configuration

12.1.9 Switch
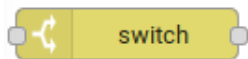
The Switch node allows messages to be routed to different branches of a flow by evaluating a set of rules against each message.

Fields:

- **Name** *(optional)*: Name of the node
- **Property** *(required)*: Variable that will be evaluated
- **Rule box** *(required)*: Rules that will determine the output branch of the node. Also, it can be configured to stop checking rules when it finds one that matches other or check all the rules and route the message to the corresponding output.

Note:

- More than one rule can be assign by clicking on *+add* below the rules box.
- The rules are mapped one-to-one to the output conectors. Then the first rule is related to the first output, the second rule to the second output and etc...
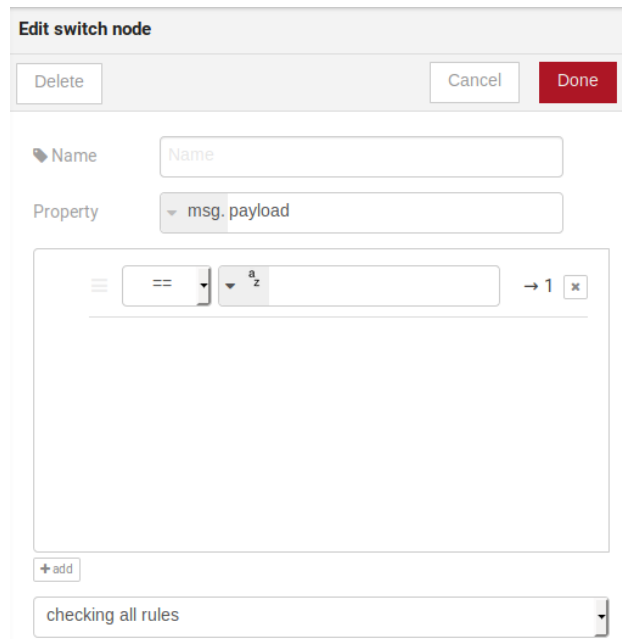
Fig. 12.9: : Switch configuration

12.1.10 Template

Note: Despite the name, this node has nothing to do with dojot templates

This node will assign a value to a target variable. This value can be a constant, the value of an attribute that came from the entry device and etc. . .

It uses the mustache template language. Check Fig. 12.10 as example: the field **a** of payload will be replaced with the value of the **payload.b**

Fields:

- **Name** *(optional)*: Name of the node
- **Set Property** *(required)*: Variable that will receive the value
- **Format** *(required)*: Format template will be writen
- **Template** *(required)*: Value that will be assigned to the target variable set on **Set property**
- **Output as** *(required)*: The format of the output

Fig. 12.10: : Template configuration

12.1.11 Cron

Processing node to create/remove cron jobs. Cron allowing to schedule tasks to: send events to the data broker or execute a http request.

Fields:

- **Operation** *(required)*: Defines the type of processing if creating or removing cron jobs (CREATE, REMOVE).

- **CRON Time Expression** *(required)*: CRON Time Expression, eg. * * * * * *. Required when using CREATE type operation.

- **JOB Name** *(optional)*: Name of Job.

- **JOB Description** *(optional)*: Description of Job.

- **JOB Type** *(required)*: Options are EVENT REQUEST or HTTP REQUEST.

- **JOB Action** *(required)*: Variable that contains the JSON to JOB Action. This value can be assigned to the variable using the template node, for example.

- **JOB Identifier (output to)** *(required)*: Variable that will receive the JOB Identifier.

- **Name** *(optional)*: Name of the node

Example of *JOB Action* when *JOB Type* is **HTTP REQUEST**:

```
{
    "method": "PUT",
    "headers": {
            "Authorization": "Bearer ${JWT}",
```

(continues on next page)

Fig. 12.11: : Cron configuration

```
                "Content-Type": "application/json"
            },
    "url": "http://device-manager:5000/device/${DEVICE_ID}/actuate",
    "body": {
                "attrs": {"message": "keepalive"}
            }
}
```

Example of *JOB Action* when *JOB Type* is **EVENT REQUEST**:

```
{
    "subject": "dojot.device-manager.device",
    "message": {
                "event": "configure",
                "data": { "attrs": { "message": "keepalive"},
                        "id": "6a1213"
                    },
                "meta": { "service": "admin"}
            }
}
```

12.1.12 Cron batch

It works like the *cron node*, but here you can use a batch of schedules.

Fig. 12.12: : Cron batch configuration

Fields:

- **Operation** *(required)*: Defines the types of processings if creating or removing cron jobs (CREATE, REMOVE).
- **JOB requests** *(required)*: Variable that contains the array of JSONs to JOB Actions.
- **JOB identifiers** *(required)*: Variable that will receive the array of job identifiers.
- **Name** *(optional)*: Name of the node

12.1.13 Geofence
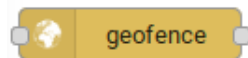
Select an interest area to determine which devices will activate the flow

Fields:

- **Area** *(required)*: Area that will be selected. It can be chosen with an square or with a pentagon.
- **Filter** *(required)*: Which side of the area will be picked: inside or outside the marked area in the field above.
- **Name** *(optional)*: Name of the node
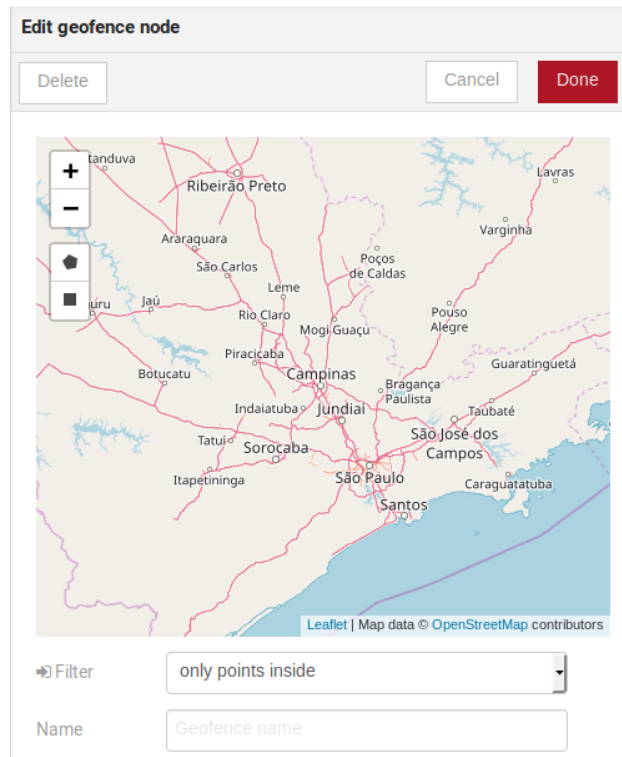
Fig. 12.13: : Geofence configuration

12.1.14 Get Context
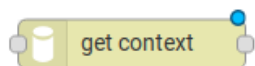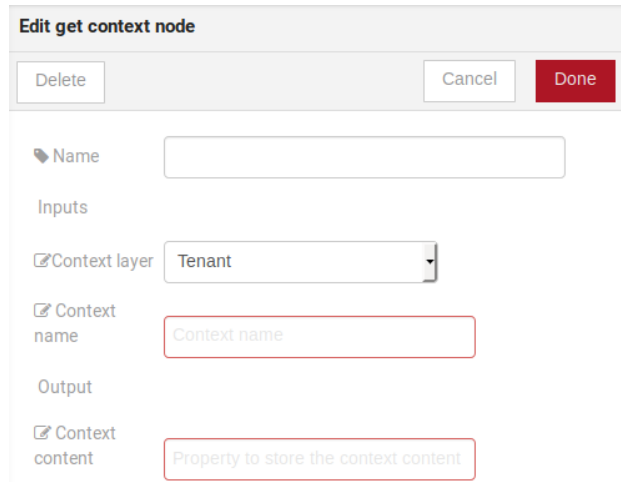
This node is used to get a variable that is in the context and assign its value to a variable that will be used in the flow.

Note: Context is a mechanism that allows a given set of data to persist beyond the life of the event, thus making it possible to store a state for the elements of the solution.

Fields:

- **Name** *(optional)**: Name of the node
- **Context layer** *(required)**: The layer of the context that que variable is at
- **Context name** *(required)**: The variable that is in the context
- **Context content** *(required)**: The variable in the flow that will receive the value of the context
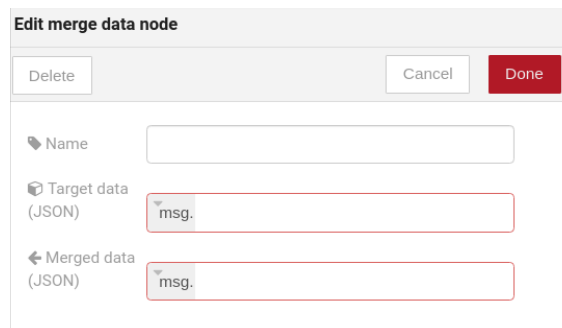
12.1.15 Merge data
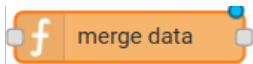
This node allows objects to be merged in the **flow** context.

Fig. 12.14: : Merge data configuration

Fields:

- **Target data (JSON)** *(required)*: Variable that contains the data to be merged.
- **Merged data (JSON)** *(required)*: Variable that will receive the new data merged with your existing data.
- **Name** *(optional)*: Name of the node

12.1.16 Cumulative sum
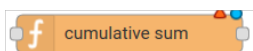
The cumulative sum node accumulates the data for an attribute in a temporal window and keeping this in the **flow** context.

Fields:

- **Time period (min)** *(required)*: Time in minutes to keep the sum.

Fig. 12.15: : Cumulative sum configuration

- **Target attribute** *(required)*: Variable that contains the value to be sum.
- **Timestamp** *(required)*: Variable containing the timestamp from the device or dojot. Most of the time it can be set with *payload.metadata.timestamp*.
- **Sum** *(required)*: Variable that will receive the sum.
- **Name** *(optional)*: Name of the node

12.1.17 Publish in FTP topic
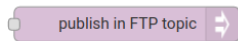
Node to forward messages to Apache Kafka FTP topic.

It publishes to the `tenant.dojot.ftp` topic (*tenant* is defined by which tenant the flow belongs to) in which the messages are produced with informations about the file name, encoding format and file content.

Fig. 12.16: : *Publish in FTP topic* configuration

Fields:

- **Encoding** *(required)*: The encoding that the file to be sent uses. Valid values are: ascii, base64, hex, utf16le, utf8 and binary.

- **Filename** *(required)*: Variable with the name of the file to be sent.

- **Content** *(required)*: Variable with the file contents to be sent.

- **Name** *(optional)*: Name of the node

Example of a message sent by this node:

```
{
    "metadata": {
        "msgId": "33846252-659f-42cc-8831-e2ccb923a702",
        "ts": 1571858674,
        "service": "flowbroker",
        "contentType": "application/vnd.dojot.ftp+json"
    },
    "data": {
        "filename": "filename.jpg",
        "encoding": "base64",
        "content": "..."
    }
}
```

Where the keys above are:

- msgId: Value of type uuidv4 used to uniquely identify the message in dojot's context.

- ts: Timestamp in Unix Timestamp (ms) format from the moment the message was produced.

- service: Name of the service that generated the message.

- contentType: Type of encoding used by the file.

- filename: Name of the file to be sent to the FTP server.

- encoding: Encoding the contents of the file. Valid values are: ascii, base64, hex, utf16le, utf8 and binary.

- content: File contents.

This can be used with the kafka2ftp component. See more in *Components and APIs*.

12.1.18 Deprecated nodes

These nodes are scheduled to be removed in future versions. They will work with no problems with current flows.

Device in

This node determine an especific device to be the entry-point of a flow. To configure the device in node, a window like Fig. 12.17 will be displayed.

Fields:

- **Name** *(optional)*: Name of the node

- **Device** *(required)*: The *dojot* device that will trigger the flow

- **Status** *(required)*: *exclude device status changes* will not use device status changes (online, offline) to trigger the flow. On the other hand, *include devices status changes* will use these status to trigger the flow.

Fig. 12.17: : Device in configuration window

Note: If the the device that triggers a flow is removed, the flow becomes invalid.

Device template in
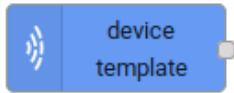
This node will make that a flow get triggered by devices that are composed by a certain template. If the device template that is configured in **device template in** node is template A, all devices that are composed with template A will trigger the flow. For example: *device1* is composed by templates [A,B], *device2* by template A and *device3* by template B. Then, in that scenario, only messages from *device1* and *device2* will initiate the flow, because template A is one of the templates that compose those devices.

Fig. 12.18: : Device template in configuration window

Fields:

- **Name** *(optional)*: Name of the node.
- **Device** *(required)*: The *dojot* device that will trigger the flow.
- **Status** *(required)*: Choose if devices status changes will trigger or not the flow.

Device out
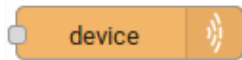
Device out will determine which device will have its attributes updated on *dojot* according to the result of the flow. Bear in mind that this node doesn't send messages to your device, it will only update the attributes on the platform. Normally, the chosen device out is a *virtual device*, which is a device that exists only on *dojot*.

Fig. 12.19: : Device out config window

Fields:

- **Name** *(optional)*: Name of the node.

- **Device** *(required)*: Select "The device that triggered the flow" will make the device that was the entry-point be the end-point of the flow. "Specific device" any chosen device wil be the output of the flow and "a device defined during the flow" will make a device that the flow selected during the execution the endpoint.

- **Source** *(required)*: Data structure that will be mapped as message to device out

Actuate
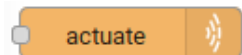
Actuate node is, basically, the same thing of **device out** node. But, it can send messages to a real device, like telling a lamp to turn the light off and etc.

Fig. 12.20: : Actuate configuration

Fields:

- **Name** *(optional)*: Name of the node.

- **Device** *(required)*: A real device on dojot
- **Source** *(required)*: Data structure that will be mapped as message to device out

12.2 Learn by examples

- *Using http node*
- *Using geofence node*
- *Using cumulative sum, switch and notification node*

12.2.1 Using http node

Imagine this scenario: a device sends an *username* and a *password*, and from these attrs, the flow will request to a server an authentication token that will be sent to a virtual device that has a *token* attribute.
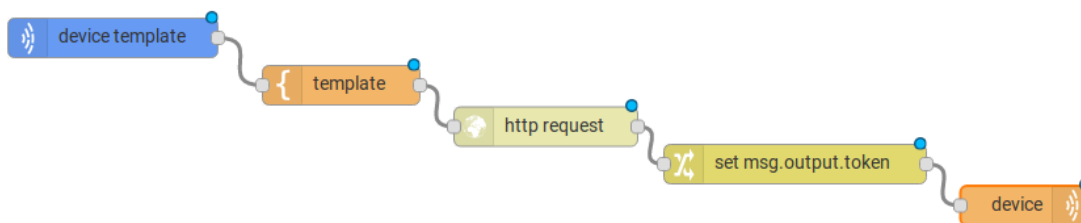
Fig. 12.21: : Flow used to explain http node

To send that request to the server, the http method should be a POST and the parameters should be within the requisition. So, in the template node, a JSON object will be assigned to a variable. The body (parameters *username* and *password*) of the requisition will be assigned to the **payload** key of the JSON object. And, if needed, this object can have a *headers* key as well.

Then, on the http node, the Requisition field will receive the value of the object created at the template node. And, the response will be assigned to any variable, in this case, this is *msg.res* .

Note: If UTF-8 String buffer is chosen in the return field, the body of the response body will be a string. If JSON object is chosen, the body will be an object.

As seen, the response of the server is *req.res* and the response body can be accessed on **msg.res.payload**. So, the keys of the object that came on the responsy can be accessed by: **msg.res.payload.key**. On figure Fig. 12.24 the token that came in the response is assigned to the attribute token of the virtual device.

Then, the result of the flow is the attribute *token* of the virtual device be updated with the token that came in the response of the http request:

Edit template node

Delete   Cancel   Done

🏷 Name   [   ]

✏ Set property   [▾ msg. req   ]

</> Format   [Handlebars template   ▾]

▣ Template
```
i 1  {
  2  "headers": {
  3      "content-type": "application/json"
  4    },
  5  "payload": {
  6      "username":"{{payload.data.attrs.username}}",
  7      "passwd":"{{payload.data.attrs.passwd}}"
  8    }
  9  }
```

→ Output as   [Plain text   ▾]

Fig. 12.22: : Template node configuration

Edit http node

Delete

≡ Method   [POST   ]

◉ URL   [http://172.21.0.1:5000/auth   ]

✉ Request
body   [▾ msg. req   ]

✉ Response   [▾ msg. res   ]

← Return   [a parsed JSON object   ]

🏷 Name   [Name   ]

Tip: If the JSON parse fails the fetched string is returned as-is.

Fig. 12.23: : Template node configuration

Fig. 12.24: : Template node configuration

Fig. 12.25: : Device out configuration
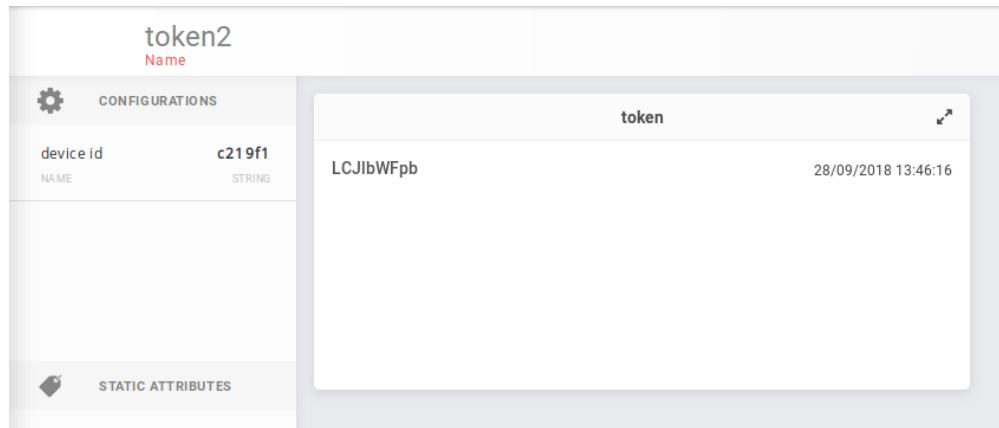
Fig. 12.26: : Device updated

12.2.2 Using geofence node

A good example to learn how geofence node works ia studying the flow below:
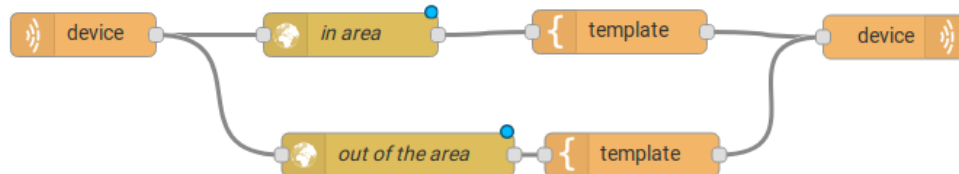
Fig. 12.27: : Flow using geofence

The geofence node named *in area* is set like seem in Fig. 12.28. The only thing that diffs the geofence nodes *in area* from *out of the area* is the field **Filter** that, in the first, is configured to *only points inside* and *only points outside* in the second, respectively.

Then, if the device that is set as *device in* sends a messagem with a geo attribute the geofence node will evaluate the geo point acoording to its rule and if it matches the rule, the node forward the information to the next node and, if not, the execution of the branch, which has the geofence that the rule didn't match, stops.

Note: To geofence node work, the messege received **should** have a geo attribute, if not, the branches of the flow will stop at the geofence nodes.

Back to the example, if the car sends a message that he is in the marked area, like `{ "position": "-22. 820156,-47.2682535" }`, the message received in device out will be "Car is inside the marked area", and, if it sends `{"position": "0,0"}` device out will receive "Car is out of the marked area"

Fig. 12.28: : Geofence node configuration

Fig. 12.29: : Template node configuration if the car is in the marked area

Fig. 12.30: : Output in device out

12.2.3 Using cumulative sum, switch and notification node

Imagine this scenario: a device sends the level of rain, we want to generate a notification if the accumulated, sum, of the rains in the last hour is greater than 100.
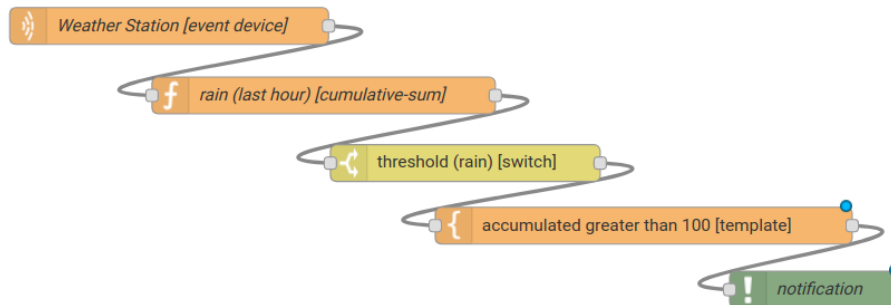
Fig. 12.31: : Flow using cumulative sum, switch and notification

In the *cumulative sum* node, we will accumulate the value of rain (*Target attribute*) in the 60 minute time window (*Time period*) and we will set this sum in a new attribute called *payload.data. attrs.rain60Min* (*Sum*). The *Timestamp* setting refers to the timestamp from the device or dojot, most of the time it can be set with *payload.metadata.timestamp*. See more in Fig. 12.32 .

Fig. 12.32: : Cumulative sum node configuration

We want the notification to be triggered only if the accumulated rain value is greater than 100, for that we will use the switch node. As in image Fig. 12.33.

Now, if our value is greater than 100 we need to generate the notification, for that we will use an auxiliary node before,

Fig. 12.33: : Switch node configuration

the *template* node. In the template node we will create the message that will appear in the notification and define its metadata, Fig. 12.34.

Finally, we will configure the notification node, as in image Fig. 12.35.

So, if the weather station (device set in the event device node with publication checked) sends several messages like *{" rain ": 5}* during the last hour and one of these times the sum exceeds 100, the notification will be generated. Note: Multiple notifications can be generated, as long as the accumulated value is greater than 100. See image Fig. 12.36.

Fig. 12.34: : Template node configuration

Fig. 12.35: : Notification node configuration
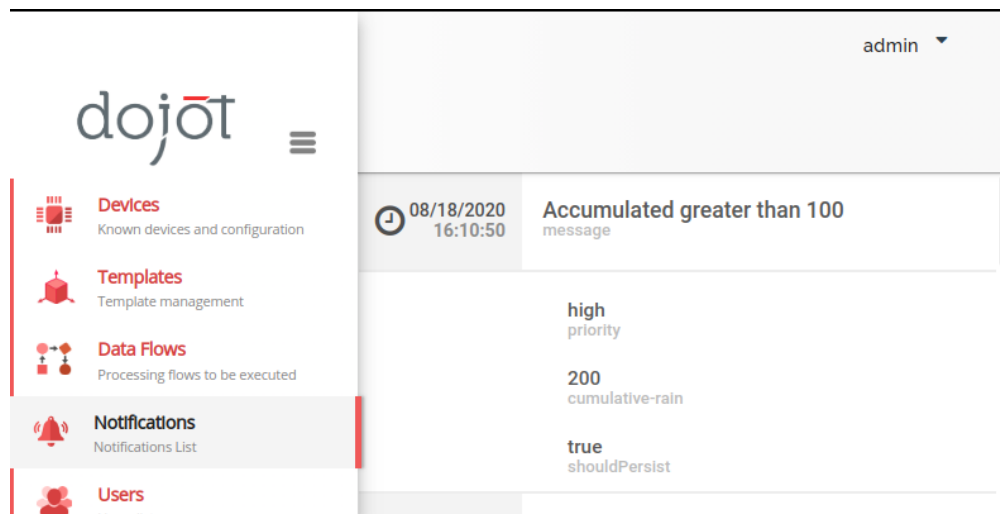
Fig. 12.36: : Notification

USING KEYCLOAK IN DOJOT

As of version v0.8.0, dojot started using keycloak and removed the use of the old *auth* service. Keycloak is an open source identity and access management solution.

In this tutorial, we will explain some important points of setting up and using keycloak with dojot.

Note:

- Audience: users and administrators
- Level: basic to intermediate
- Reading time: 20 m

Table of Contents

Attention: A dojot *tenant* is equivalent to a keycloak *realm*.

13.1 Required

- If you are starting a *deployment*, installing a dojot, see *Setting Passwords (Required) via Environment Variables*.

13.2 Recommendations

- Always use HTTPs.
- Use strong passwords, by default the minimum required are passwords with 8 characters, one number, one uppercase letter, one lowercase letter and one special character.
- Configure SMTP

13.3 Accessing the Administrative Panel

Replace <url> and <realm> for your case

13.3.1 Using the master realm

> **Attention:** You can only access this page using **localhost** or **https**.

Using a user who belongs to the Realm master and can control all realms, go to `<url>/auth/`, eg `https://dojot.com.br/auth`.

13.3.2 Using a user with role of *admin*

Using a user who belongs to a realm can only control this realm, go to `<url>/auth/admin/<realm>/console`, eg `https://dojot.com.br/auth/admin/admin/console`.

13.3.3 Checking selected realm

After accessing the keycloak administrative interface using one of the paths above, make sure you have the realm you want to use selected, in the left sidebar you can select the realm of interest, as highlighted in the image below:
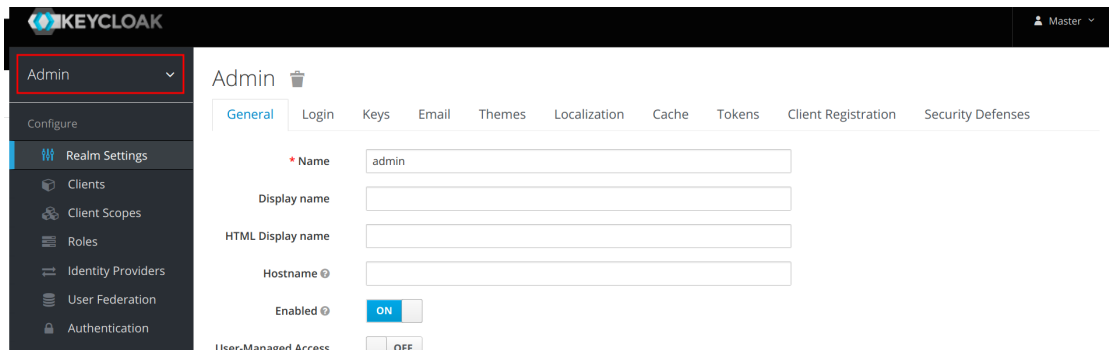
13.4 Accessing an Account Settings

Replace <url> and <realm> for your case

To access your account and make changes like enable 2-factor authentication, change password, change email, go to the `<url>/auth/realms/<realm>/account`, eg `https://dojot.com.br/auth/realms/admin/account`.

13.5 Creating a new user

Access the Administrative panel as explained in the topic *Accessing the Administrative Panel*.

In the left bar menu select the *Users* option and then click *Add User*.

Now let's create the user, first fill in **Username**. It is recommended to fill in the **Email** field and in **Required User Actions** Add the `Update Password` and `Verify Email` options, but for `Verify Email` to make sense it is necessary to have the SMTP configured for the Realm of interest. If you do not want to configure SMTP do not put `Verify Email`. Another point is that if an SMTP has not been configured, it is necessary to create a temporary password and provide the user, to do so, access the **Credentials** tab after creating in **Save**.

Furthermore, it is important to define the **Role** of this user in your Realm, without this it cannot be used. After clicking `Save` one of the tabs will be **Role Mapping**, click on it to define the role of the user. In the **Role Mapping** tab in the **Available Roles** box there will be the `admin` and `user` options (if no other *Roles* are created). A user with role `admin` has access to all APIs and also the Administrator Panel of your realm while a user with Role `user` has access to the API but cannot access the Administrator Panel. Select the Role that fits this new user and add it to the **Assigned Roles** Box.

13.6 Configuring SMTP through the graphical interface

Access the Administrative panel as explained in the topic *Accessing the Administrative Panel*.

In the menu on the left bar select the option **Realm Settings** and then select the tab **Email** and make the necessary settings.

13.7 Setting HTTPs as required by the graphical interface

Note: You must be using a *deployment* with HTTPs configured.

Access the Administrative panel as explained in the topic *Accessing the Administrative Panel*.

In the menu on the left bar select the **Realm Settings** option and then select the **Login** tab, under **Require SSL** select the `external requests` option and click **Save**.

13.8 Creating a new realm (tenant in the dojot)

Access the administrative panel as explained in the topic *Accessing the Administrative Panel*, but necessarily **using the master realm**.

On the left side menu, hover over the current Realm name and an **Add realm** option will appear, click on it.

Note: The realm created here will be based on the file passed as the base for the keycloak, see more in *Base keycloak configuration file for creating realms*.

13.9 Getting Access Token (Access Token JWT)

13.9.1 From *username* and *password* in a realm (Not recommended)

We have a *client* created in keycloak configured and disabled with the name `dev-test-cli` which allows getting a JWT from a login and password in a realm.

> **Attention:** For security reasons it is disabled by default, after use it is recommended to disable it again.

How to enable and disable client `dev-test-cli`

Access the Administrative panel as explained in the topic *Accessing the Administrative Panel*.

Select the option **Clients** in the left side menu, it will load a new screen, in it there will be the list of clients. Look for the `dev-test-cli` client, click on it, it will open a new screen with several options, one of them will be the **Enabled** option which will have the value `OFF` change to the value `ON`, go to the end of the page and click `Save`.

Remember to disable this client after use by setting the **Enabled** value to `OFF` and saving, for security reasons.

> **Attention:** To use the command below to obtain the JWT you must already have access through the graphical interface the keycloak once and defined a password. If you haven't done this yet, please follow the previous topic *Accessing an Account Settings* before continuing. Also it is necessary to have curl and jq. In Debian-based Linux distributions you can run: `sudo apt-get install curl jq`

After enabling the `dev-test-cli` client it will be possible to obtain a token using login and password. As in the curl command below using the keycloak API, change the values of `DOJOT_URL`, `REALM`, `USERNAME` and `PASSWORD` according to your case:

```
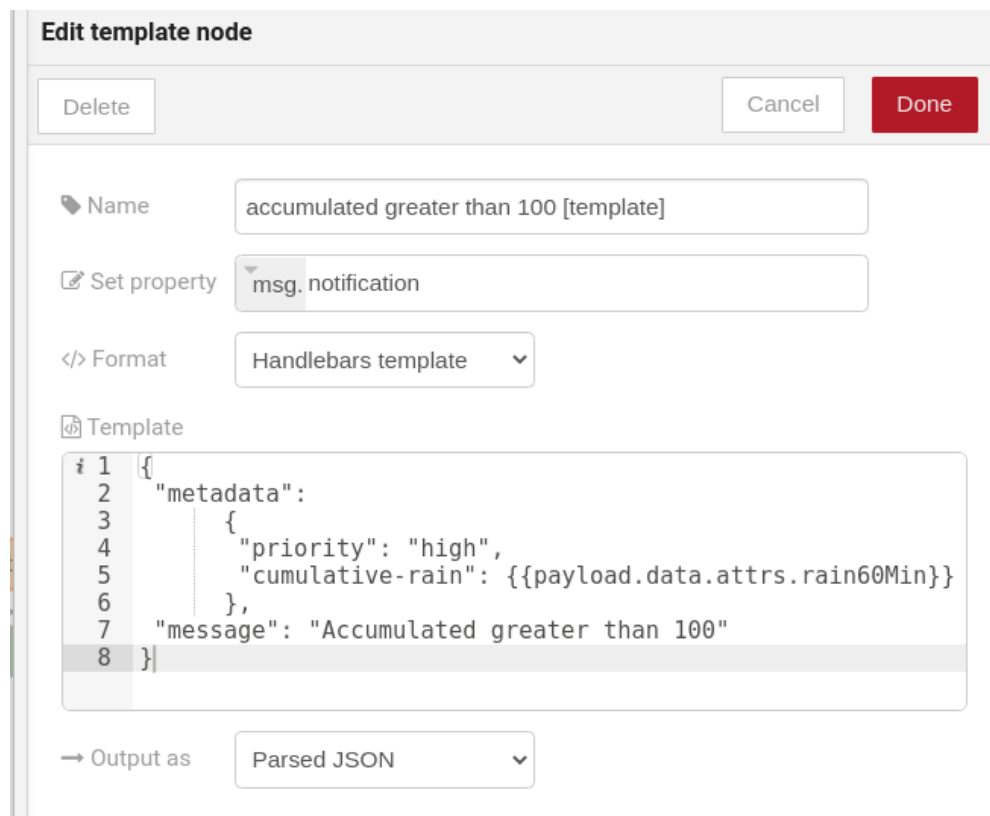DOJOT_URL=http://localhost:8000
REALM=admin
USERNAME=admin
PASSWORD=password

JWT=$(curl --location --request POST ${DOJOT_URL}/auth/realms/${REALM}/protocol/
↪openid-connect/token \
--data-urlencode "username=${USERNAME}" \
--data-urlencode "password=${PASSWORD}" \
--data-urlencode "client_id=dev-test-cli" \
--data-urlencode "grant_type=password" 2>/dev/null | jq -r ".access_token")
```

If everything works out, the token will be available in the JWT variable, to get the variable's value use the command below:

```
echo $JWT
```

> **Attention:** The lifetime of this access token is not very long, this process returns a *refresh token* which can be used to get a new access token.

13.9.2 For a backend application

To use the Access Token in a backend application it is recommended to create a new *Client* for the realm of interest or the realms of interest, remember to configure them in **Client Protocol** with the value `openid-connect`, **Access Type** with the value `Confidential` and using the *Secret* obtained from the **Credentials** tab, there are several possible configurations. It is important to follow the currently established security standards for OAuth 2 and OpenIDConnect. There are several keycloak libraries in various languages that can help with this development.

13.9.3 For a frontend application

We have a microservice in order to help the development to ensure more security for using keycloak with dojot, see more about Backstage. You can use keycloak directly with PKCE and OpenID Connect as well. It is important to follow the currently established security standards for OAuth 2 and OpenIDConnect.

13.9.4 For a mobile application

We don't have use cases, but it's important to follow the currently established security standards for OAuth 2 and OpenIDConnect. Some starting points to pay attention to would be to create a new Client with the Access Type *public* and use PKCE.

13.10 Deployment Settings

13.10.1 Setting Passwords (Required) via Environment Variables

The dojot deployments (**Docker compose and Ansible - kubernetes**) do not come with passwords set for the `admin` and `master` users and it is necessary to set these passwords so that the *deployments* will start up correctly, if not configured services will not start and dojot will be unavailable. But beware these values are only applied when the keycloak starts for the first time and the *realm* are created, it will not affect the existing realms.

> **Attention:** When configuring use strong passwords, by default the minimum required are passwords with 8 characters, one number, one uppercase letter, one lowercase letter and one special character.

Docker compose

You need to set a password value in the *.env* file for the `KEYCLOAK_MASTER_PASSWORD` and `KEYCLOAK_ADMIN_PASSWORD_TEMP` variables. The `KEYCLOAK_ADMIN_PASSWORD_TEMP` value will be the *admin* user password of all realms when created. See more at settings required.

Ansible (kubernetes)

Work in progress

13.10.2 Setting SMTP via environment variables

It is possible to configure SMTP via environment variables, but beware these values are only applied when the *realm* is created, it will not affect the existing realms. To configure a *realm* that already exists, see *Configuring SMTP through the graphical interface*.

Docker compose

See more at configuring SMTP.

Ansible (kubernetes)

Work in progress

13.10.3 Setting HTTPs as required via environment variables

Note: These values are only applied when the *realm* is created, it will not affect existing realms. For existing realms see *Setting HTTPs as required by the graphical interface*.

Docker compose

You need to set the **EXTERNAL** value in the *.env* file to the `KEYCLOAK_REALM_SSL_MODE` variable. See more at configuring HTTPs.

Ansible (kubernetes)

Work in progress

13.10.4 Base keycloak configuration file for creating realms

Configuration file used as the basis for creating realms in keycloak. It is the file exported from a realm by the full keycloak. This file is configured using the `DOJOT_CUSTOM_REALM_REP_FILE` environment variable in the keycloak service. When creating a new realm the identifiers (*ids*) and realm name in this file are ignored. However, it is important to point out that our *deployments* for both *docker-compose* and *ansible (kubernetes)* already have a file with the necessary settings to use dojot.

Docker compose

In the case of docker-compose this file is in *keycloak/customRealmRepresentation.json*

Ansible (kubernetes)

Work in progress

13.10.5 Old libraries

To maintain compatibility with dojot it is necessary to provide the login and password of the master or a user who can get the list of realms available in the older libraries, namely dojot-module-nodejs, iotagent-nodejs, dojot-module-python, dojot-module-java, iotagent-java, see more in their respective documentation. However, it is important to point out that our *deployments* for both *docker-compose* and *ansible (kubernetes)* are already prepared for this and it is only necessary to configure the passwords as described in the topic *Setting Passwords (Required) via Environment Variables*.

13.10.6 Configuring new routes, services, resources,on keycloak to use authorization

In keycloak settings example there is a basic case of configuration example, you can use it as basis for new settings.

13.10.7 Configuring keycloak with Nginx

In how to use keycloak with a nginx proxy there is an example with the necessary settings.

FOURTEEN

USING MQTT WITH SECURITY (TLS)

Note:

- Audience: users

- Level: intermediate

- Reading time: 15 m

This document describes how to configure the dojot and publish/subscribe using MQTT over TLS (MQTTS) when using the microservice Broker MQTT IotAgent-VerneMQ or Legacy Broker MQTT IotAgent-Mosca.

14.1 Components

14.1.1 X.509 Identity Management

The purpose of this component is to provide x.509 identification for final dojot entities, that is, IoT devices that communicate with the dojot IoT platform. See more about in X.509 Identity Management.

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.

14.1.2 MQTT Broker

IotAgent VerneMQ

The IotAgent VerneMQ extends VerneMQ with some features and services for dojot case, see more in IotAgent-VerneMQ. This is the **default** Broker MQTT in dojot deployments.

All the certificates will be automatically created, so there is no need to manually configure the brokers' ones.

> **Attention:** In order for TLS to work correctly it is necessary to configure the domain in which the `deployment` is running. In *ansible (Kubernetes)* this can be set in the `dojot_domain_name` variable and in *docker-compose* in the `DOJOT_DOMAIN_NAME` variable in *.env*. In *ansible (Kubernetes)* you can disable the *unsecured mode* by changing the `dojot_insecure_mode` variable to `false` and in *docker-compose* unmapping to port 1883 in the *iotagent-mqtt* service. Check the *Installation Guide* for more info.

14.2 How to connect a device with the IotAgent VerneMQ

> **Attention:** First of all, you will need a running dojot's environment. Then you should create a new device (or use an existing one if you prefer) and retrieve its ID.

14.2.1 Retrieving certificate for a device

For a device to connect using MQTT over TLS (MQTTS), it must possess:

- A key pair (.key file);
- A certificate signed by a Certificate Authority (CA) trusted by dojot (.crt file);
- The certificate of this CA (.crt file);

The objective when retrieving the certificate for a device is to obtain these three files: the device's certificate, the device's key pair and the CA certificate.

There are two tools to facilitate obtaining certificates from the dojot platform:

- The CertReq script.
- GUI's embedded certificate generation utility (more details in *Generating certificates for devices*).

In addition, you can use OpenSSL to create certificates and sign using the API - x509-identity-mgmt, see more at X.509 Identity Management.

A brief explanation of how to use CertReq

As prerequisites this uses git, OpenSSL, curl and jq .

Note: You need to enable the *dev-test-cli* client in the keycloak. For security reasons it is disabled by default, after use it is recommended to disable it again *How to enable and disable client dev-test-cli*.

On Debian-based Linux distributions, you can install these prerequisites by running:

```
sudo apt install git curl jq openssl
```

Download CertReq on your machine directly from dojot repository and switch to the corresponding version of your dojot environment:

```
git clone https://github.com/dojot/dojot.git
cd dojot
git checkout v0.8.0
```

Enter in `certreq` directory:

```
cd tools/certreq
```

Finally, you can run the script to generate certificates and keys as follows:

```
./bin/certreq.sh \
    -h http://localhost \
    -p 8000 \
    -i 'a1998e' \
    -t 'admin' \
    -u 'admin' \
    -s 'admin'
```

Given a *username* `admin`, *password* `admin` and *tenant* `admin`, this command will request a certificate with *device ID (identifier)* `a1998e` for the dojot platform *host* `http://localhost` on *port* `8000`.

Note: For more details about the CertReq parameters, check the CertReq - Parameters document. Other useful resources for this matter are the How to connect a device with the IoTAgent-VerneMQ with mutual TLS tutorial and the CertReq documentation.

And in the end this tool will create the directories `./ca` and `./cert_{DEVICE_ID}` to store the certificates and public/private keys.

14.2.2 Simulating a device with mosquitto

We are going to use *mosquitto* to simulate a device; it will publish and subscribe in dojot via MQTT.

Before continuing, install *mosquitto_pub* and *mosquitto_sub* from the *mosquitto-clients* package on Debian-based Linux distributions:

> **Attention:** Some Linux distributions, Debian-based Linux distributions 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 too. In this tutorial, only the tools from package *mosquitto-clients* on Debian-based Linux distributions are going to be used. Please check if another MQTT broker is not running before starting dojot (by running commands like `ps aux | grep mosquitto`) to avoid port conflicts.

On Debian-based Linux distributions you can install `mosquitto-clients` running:

```
sudo apt-get install mosquitto-clients
```

IotAgent VerneMQ (Default)

To publish and subscribe using the appropriate certificates, you must have IotAgent VerneMQ Broker, V2K Bridge, K2V Bridge and the X.509 Identity Management running, see more in IotAgent-VerneMQ.

Simulating a device publishing with mosquitto

```
mosquitto_pub  -h localhost -p 8883 -t <tenant>:<deviceId>/attrs  -m '{"attr_example
→": 10}' --cert <device .crt file> --key <device .key file> --cafile <ca .crt file>
```

An example of publication with the certificates and keys generated in the previous topic with CertReq tool.

```
mosquitto_pub \
-h localhost \
-p 8883 \
-t admin:a1998e/attrs \
-m '{"attr_example": 10 }' \
--cert './cert_a1998e/cert.pem' \
--key './cert_a1998e/private.key' \
--cafile './ca/ca.pem'
```

Simulating a device subscribing with mosquitto

```
mosquitto_sub  -h localhost -p 8883 -t <tenant>:<deviceId>/config  --cert <device .
→crt file> --key <device .key file> --cafile <ca .crt file>
```

For more details about simulate a device see in Simulating a device with mosquitto and more about simulate a device with security in Simulating a device with mosquitto with security.

IotAgent Mosca (Legacy)

To publish and subscribe using the appropriated certificates, you must need to be with the IotAgent Mosca Broker and the X.509 Identity Management running, see more in IotAgent-Mosca. In addition, you need to use a **different topic** from VerneMQ and pass the identifier to publish and subscribe, as follows:

How to publish:

```
mosquitto_pub  -h localhost -p 8883 -t /<tenant>/<deviceId>/attrs -i <tenant>:
→<deviceId> -m '{"attr_example": 10}' --cert <device .crt file> --key <device .key
→file> --cafile <ca .crt file>
```

How to subscribe:

```
mosquitto_sub  -h localhost -p 8883 -t /<tenant>/<deviceId>/config -i <tenant>:
→<deviceId> --cert <device .crt file> --key <device .key file> --cafile <ca .crt
→file>
```

Note: In these examples, the published message has the attribute *attrs_example*. You need to change its name to comply to your device's attribute.

14.2.3 Unsecured mode MQTT (without TLS)

Attention: MQTT without security is not recommended, use this for testing only.

14.3 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:

Reading certificate:

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

Getting certificate fingerprint:

```
openssl x509 -in certFile.crt -noout -fingerprint -sha256
```

LOAD TESTING DOJOT PLATFORM

In this tutorial, will be shown how to run a load test using the dojot's Locust implementation.

> **Attention:** Locust was created to work with VerneMQ. Tests with Mosca are not guaranteed to work.

Table of Contents

15.1 Setting the environment up

Note: You need to enable the *dev-test-cli* client in the keycloak. For security reasons it is disabled by default, after use it is recommended to disable it again *How to enable and disable client dev-test-cli*.

First of all, you will need a running dojot environment. Check the *Installation Guide* for more info.

To access the Locust implementation, download the dojot repository on your machine and switch to the same version as your current environment:

```
git clone https://github.com/dojot/dojot.git
cd dojot
git checkout v0.8.0
```

Enter in Locust directory:

```
cd connector/mqtt/locust
```

15.2 Running a simple test

In this section, it will be shown how to configure and run Locust, and also how to generate certificates with the `generate_certs` script in order to execute the load test. This is a simple test where will be created 100 clients to send messages to dojot.

15.2.1 Configuration

Note: Make sure to check the README included in Locust directory to learn more about the architecture and configurations. In this tutorial we will only cover the most important configurations.

Before running the tests, there must be changed some configurations in Locust docker compose files.

Open `Docker/docker-compose-master.yml` and change the following environment variables:

```
# The location of your dojot installation
DOJOT_URL: "http://1.2.3.4"
# Usually it's the dojot address too (if you don't know for sure, keep the same
↪address as dojot)
DOJOT_MQTT_HOST: "1.2.3.4"
# The default MQTTS port
DOJOT_MQTT_PORT: "8883"
```

Open `Docker/docker-compose-slave.yml` and change the following environment variables:

```
# The location of your dojot installation
DOJOT_URL: "http://1.2.3.4"
# Usually it's the dojot address too (if you don't know for sure, keep the same
↪address as dojot)
DOJOT_MQTT_HOST: "1.2.3.4"
# The default MQTTS port
DOJOT_MQTT_PORT: "8883"
```

Note: We are assuming you are running the master and the slave in the same machine, i.e. in *non-distributed mode*. Later on will be shown how to distribute slaves between multiple machines.

Open `Docker/scripts/generate_certs/docker-compose.yml` and change the following environment variables:

```
# The location of your dojot installation
DOJOT_URL: "http://1.2.3.4"
```

15.2.2 Generating the certificates

As said before, the communication from Locust to dojot is secure, then, it is necessary to use certificates.

There are two ways of simulating devices: you can create fake devices (will not show up in dojot's GUI) or real devices. In this part of the tutorial, we will create real devices, so you can check the sent messages in the GUI.

Before running the script, we need to initialize the Locust master. Inside Locust repository, run:

```
docker-compose -f Docker/docker-compose-master.yml up
```

After its initialization, run the `generate_certs` script docker compose and enter in it:

```
docker-compose -f Docker/scripts/generate_certs/docker-compose.yml up -d
docker-compose -f Docker/scripts/generate_certs/docker-compose.yml exec generate-
↪certs bash
```

Create the devices in dojot:

```
generate_certs dojot create --devices 100
```

You can now check that the devices are created in dojot.

Fig. 15.1: : Some of the devices that `generate_certs` created in dojot.

Generate the certificates for them:

Fig. 15.2: : The template used by `generate_certs` to create devices.

```
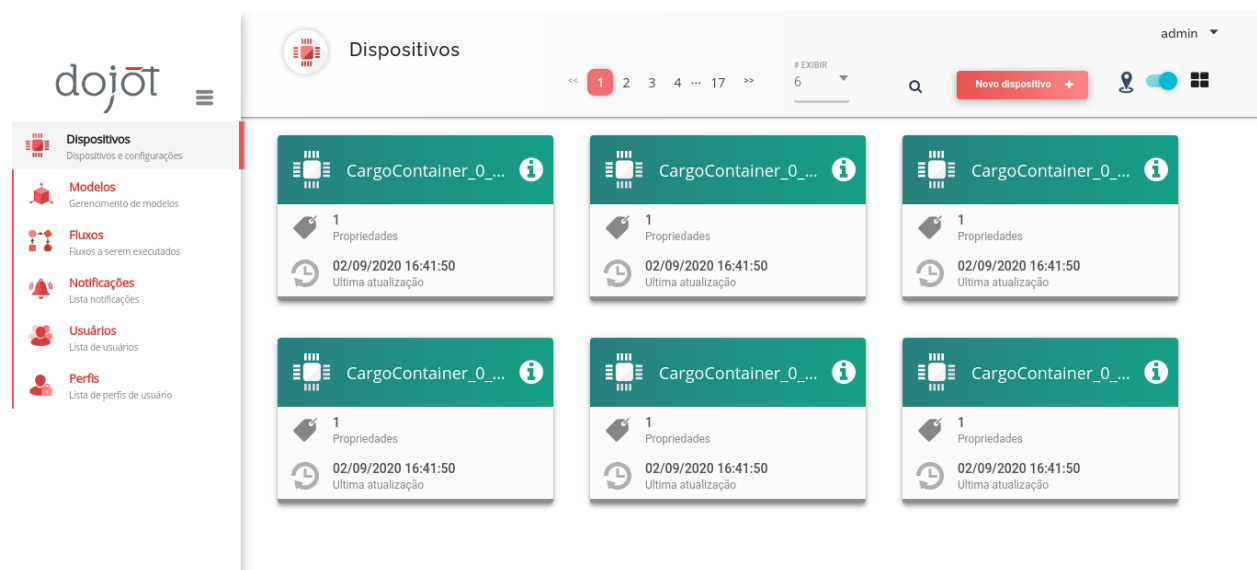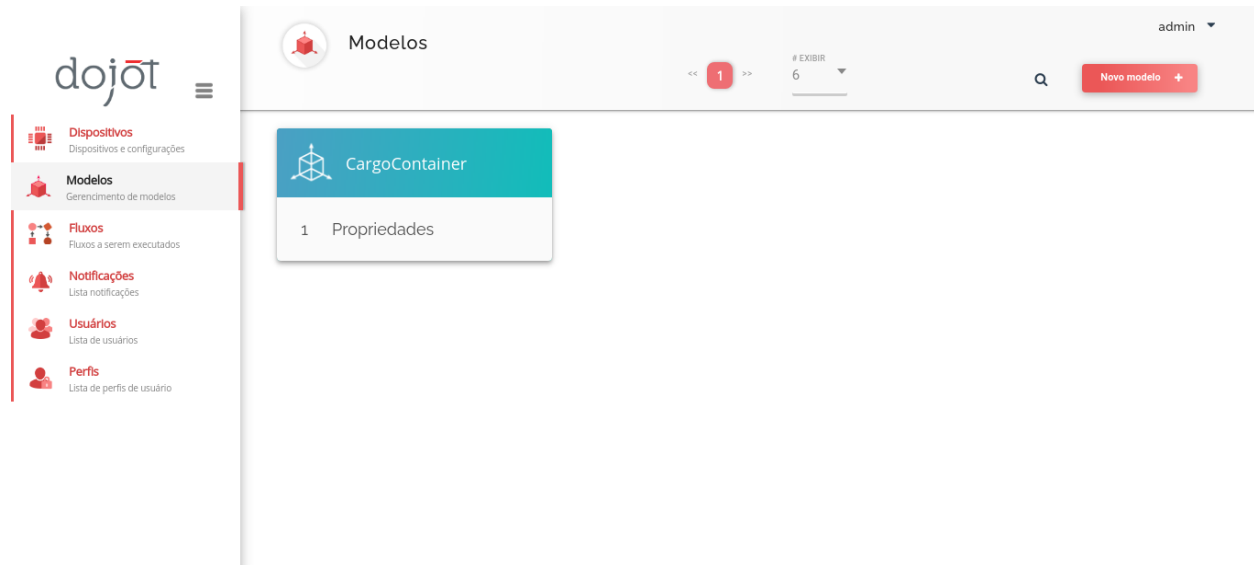generate_certs cert --dojot
```

The certificates are exported to the `cert` directory. Now the test can be initialized!

15.2.3 Initializing the slaves

Locust master does nothing by itself. The one who does all the work of sending requests is the Locust slave container. Initialize it by running:

```
docker-compose -f Docker/docker-compose-slave.yml up
```

You should see a message in the Locust master log saying that one slave has connected to him.

15.2.4 Running the test

You are all set to begin the test. To run it, you need to access the Locust interface in your browser in `localhost:8089` (supposing you are running Locust master in `localhost`).

Type `100` in `Number of users to simulate`, `10` in `Hatch rate` and click in `Start swarming`. This tells Locust to run 100 clients, creating 10 of them per second.

The default configuration is for every client to send a message each 30 seconds, so you'll have to wait a moment for the messages to arrive at dojot.

You can go to dojot and see that the messages are arriving there.
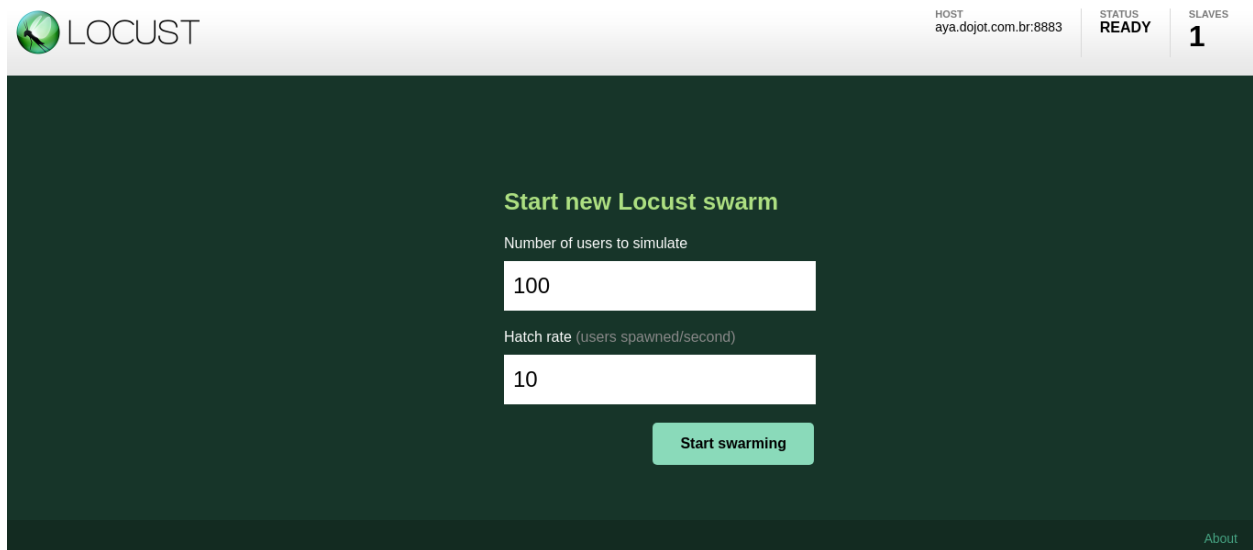
Fig. 15.3: : Configuring Locust to run the clients.
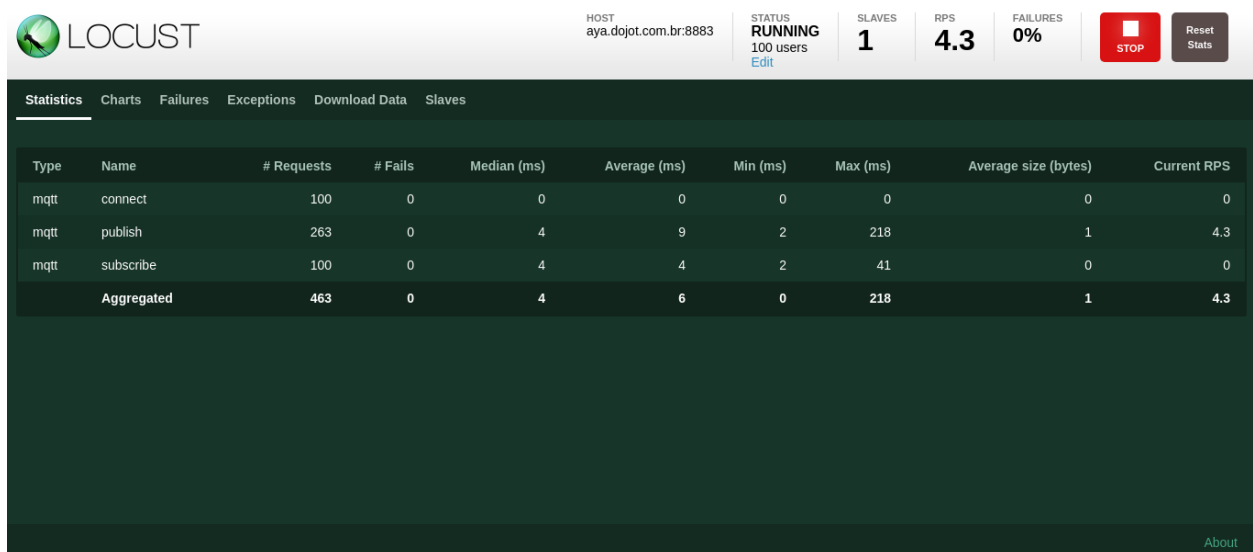
Fig. 15.4: : Locust statistics after running for a few minutes.
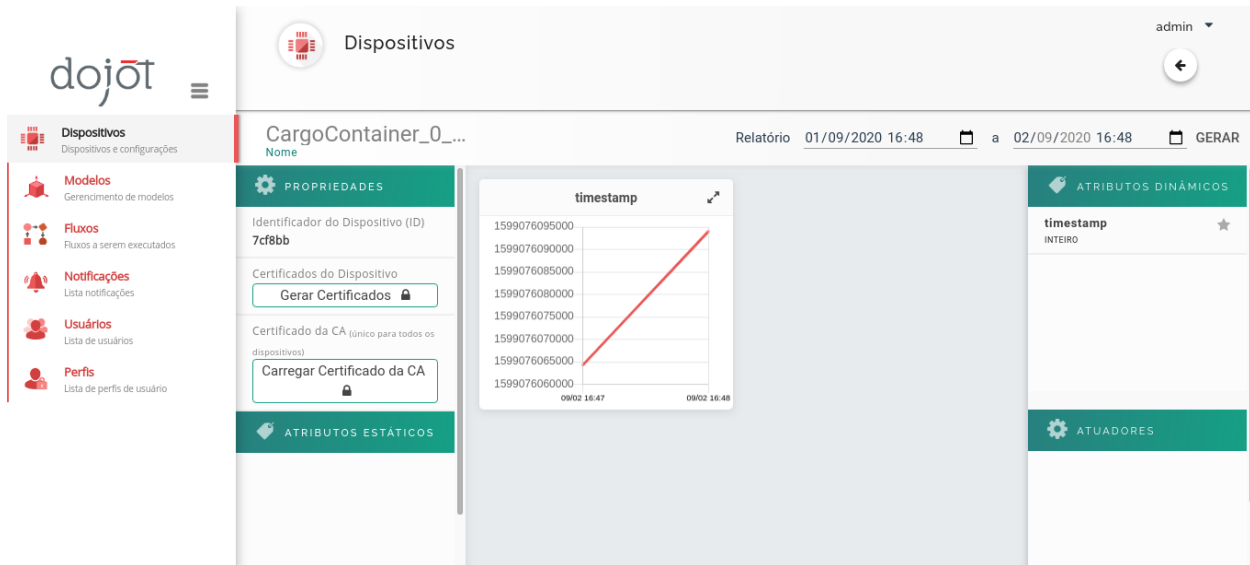
Fig. 15.5: : An example device receiving messages.

15.3 Running a distributed test

For small tests, the forementioned procedure might be sufficient, but if you really want to force dojot, you might encounter some barriers when using only one slave and/or one machine. As a solution for this problem, Locust has a **distributed mode**, permitting you to initialize multiple slaves in multiple machines, limiting Locust performance to the amount of processing power (and budget!) you got.

We are going to use two virtual machines to run 4 slaves (2 in each VM) and 1 master (in one of them) to create 1,000 fake devices. We will refer to the machine with the master as **primary** and the other as **secondary**.

> **Attention:** There is no correlation between the chosen numbers: the amount of clients that each slave supports depends on various variables, such as the number of available CPU cores.

> **Tip:** We advice you to run 1 slave per CPU core you have in the machine, i.e. if you have a VM with 4 CPUs, run 4 slaves.

15.3.1 Configuration for the distributed case

Before running the tests, there must be changes to be made in some Locust docker compose files. Note that you must clone the dojot repository in each machine that you will use for Locust.

> **Tip:** Since the configurations can be the same for the slaves and the generate_certs script, you can share them between machines by using `sshfs`. Assuming you are in the Locust directory in the secondary machine, run the following command:

```
sshfs -o allow_other -o nonempty <user>@<ip>:/path/to/dojot/connector/mqtt/locust/
↪Docker Docker
```

Check the sshfs documentation for more details.

Open `Docker/docker-compose-master.yml` in the primary machine and change the following environment variables:

```
# The location of your dojot installation
DOJOT_URL: "http://1.2.3.4"
# Usually it's the dojot address too (if you don't know for sure, keep the same
↪address as dojot)
DOJOT_MQTT_HOST: "1.2.3.4"
# The default MQTTS port
DOJOT_MQTT_PORT: "8883"
```

Open `Docker/docker-compose-slave.yml` and change the following environment variables:

```
# The location of your dojot installation
DOJOT_URL: "http://1.2.3.4"
# Usually it's the dojot address too (if you don't know for sure, keep the same
↪address as dojot)
DOJOT_MQTT_HOST: "1.2.3.4"
# The default MQTTS port
DOJOT_MQTT_PORT: "8883"


# If it's in the same machine as the master, you can leave as it is
LOCUST_MASTER_HOST: "locust-master"


# If it's in the same machine as the master, you can leave as it is
REDIS_HOST: "redis"
# Change to 6380 if the master is in another machine
REDIS_PORT: "6379"
```

Open `Docker/scripts/generate_certs/docker-compose.yml` and change the following environment variables:

```
# The location of your dojot installation
DOJOT_URL: "http://1.2.3.4"

# If it's in the same machine as the master, you can leave as it is
REDIS_HOST: "redis"
# Change to 6380 if the master is in another machine
REDIS_PORT: "6379"
```

As you can see, the configurations have changed a little bit, with the changes being only about the master and Redis location.

15.3.2 Generating certificates

In this part of the tutorial, we will create fake devices that, unlike in the simple test, won't appear in dojot GUI.

Before running the script, we need to initialize the Locust master. Inside Locust directory in your **primary** machine, run:

```
docker-compose -f Docker/docker-compose-master.yml up
```

After its initialization, initialize the docker compose `generate_certs` script in the **primary** machine and enter in it:

```
docker-compose -f Docker/scripts/generate_certs/docker-compose.yml up -d
docker-compose -f Docker/scripts/generate_certs/docker-compose.yml exec generate-
↪certs bash
```

Create the certificates:

```
generate_certs cert --devices 1000
```

Note: The fake devices are simulated as certificates.

Now go to your **secondary** machine, initialize the generate_certs and inside it, run:

```
generate_certs redis --export
```

Since the certificates are stored in Redis, you can simply export them with the shown command in any machine, preventing the tedious job of copying in each VM the cert directory with the certificates.

15.3.3 Slave initialization

Run in your **primary** and **secondary** machines:

```
docker-compose -f Docker/docker-compose-slave.yml up --scale locust-slave=2
```

This command creates two Locust slave containers in each machine. You should see in the Locust master log a message for each slave that connects to it.

15.3.4 Running the distributed test

We are all set to begin the test. To run it, you need to access the Locust interface in your browser in the Locust master location, e.g.: localhost:8089.

Type 1000 in Number of users to simulate, 10 in Hatch rate and click in Start swarming. This tells Locust to run 1,000 clients, creating 10 of them per second.

15.4 Using Grafana's Locust dashboard

The Locust web interface is easy and simple to use, but there are some downsides. The major one is the persistence: The history data will be deleted as soon as you close or refresh the page.

To solve this problem, we chose to add the Locust Exporter image to the master docker compose file, allowing us to export all of its metrics in a Prometheus-compatible format. That way we can persist this information in Prometheus and centralize all the dashboards in Grafana. Unfortunately, we still need the Locust web interface to initialize tests.

Attention: Since the Grafana/Prometheus stack is available only in the Kubernetes installation, this part is not applicable to docker compose installations. We encourage you to check the *Installation Guide* for more information on dojot's installation methods.

Fig. 15.6: : Configuring Locust to run the clients.

Fig. 15.7: : Locust statistics after running for a few minutes.

15.4.1 Ansible configuration

You need to decide where your Locust master will be beforehand to be able to initialize the Ansible playbook. The Ansible configurations that you need to change to link Locust Exporter to Prometheus are:

```
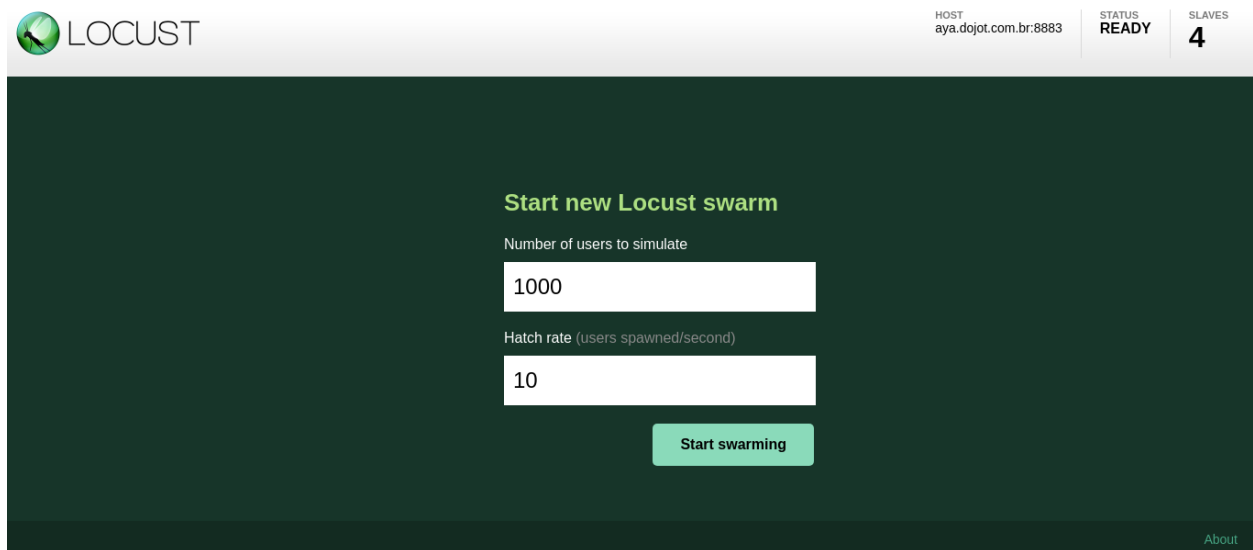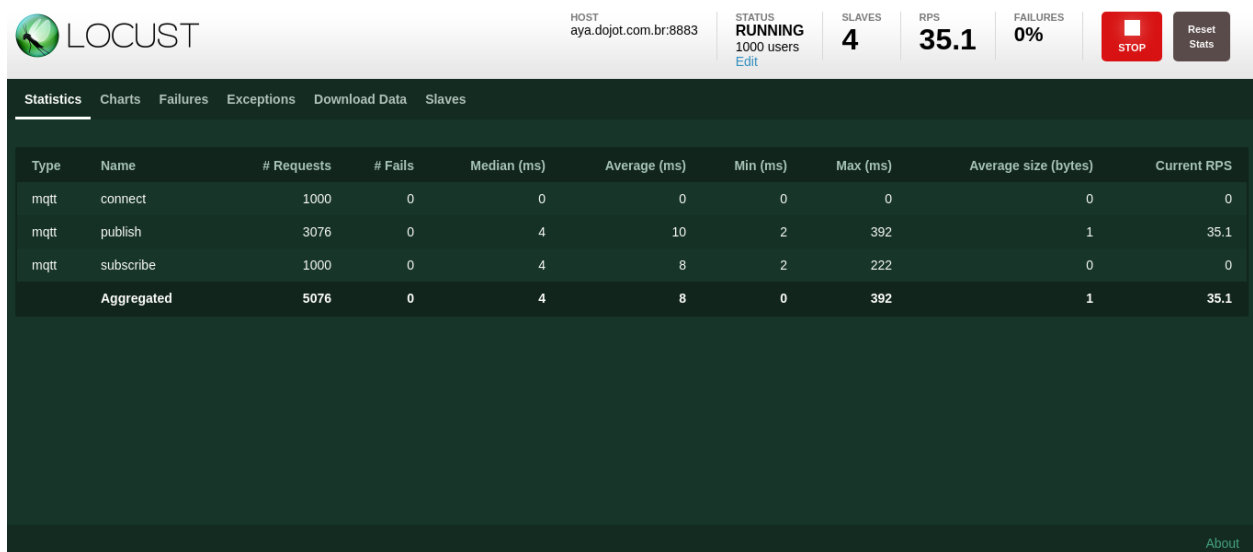dojot_enable_locust_exporter: true
dojot_locust_exporter:
  ip: 1.2.3.4
```

Change the IP to the Locust master one and run the playbook. Now you can initialize a (distributed or not) test as shown in the previous sections and you should see the Locust data being sent to Grafana's Locust dashboard.
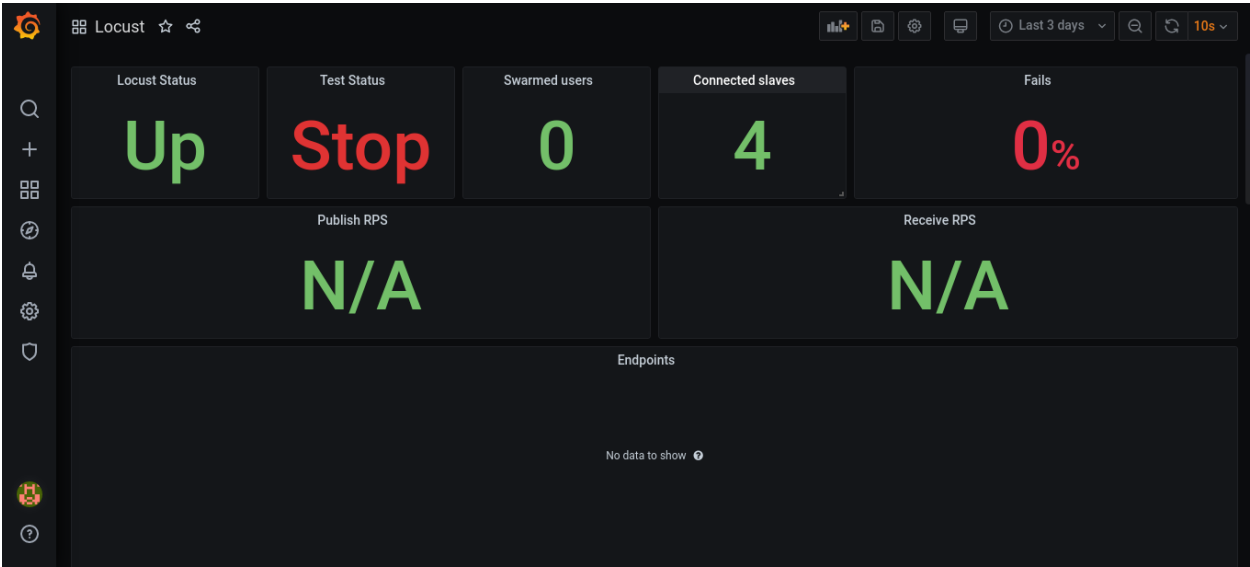
Fig. 15.8: : Locust statistics in Grafana - before beginning the test.
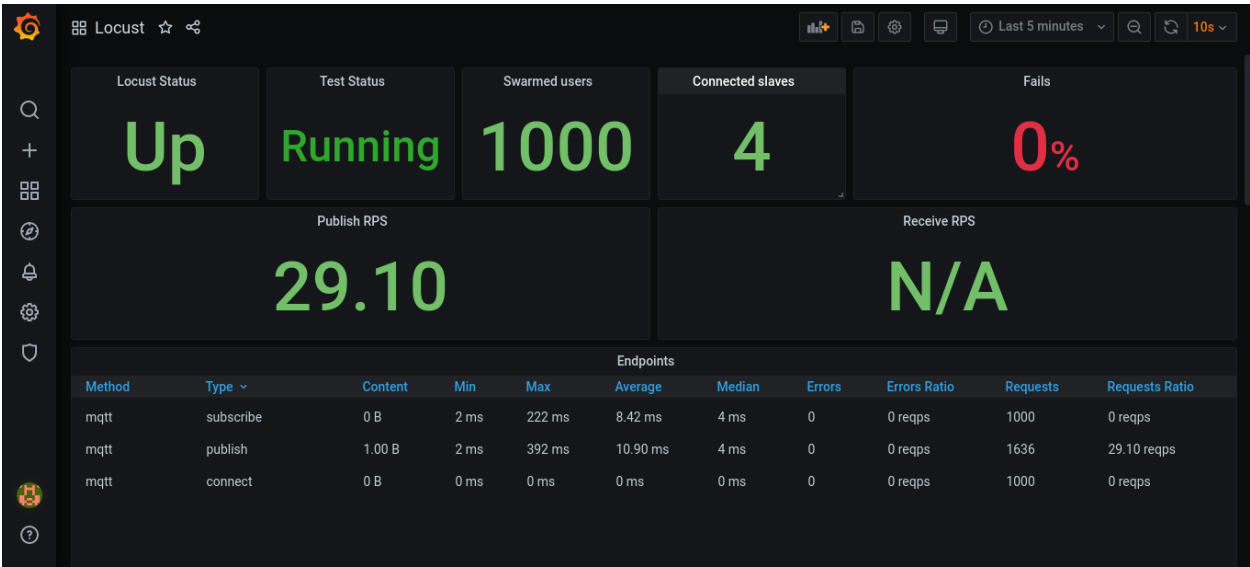
Fig. 15.9: : Locust statistics in Grafana - after the test has begun.

15.5 Requisites for a 100,000 clients test

Now that you know how to run distributed tests using Locust, you are able to execute a 100,000 clients test. For this, you will need a lot of computational power and multiple machines, both for dojot and for Locust. The goal is to reach 100,000 simultaneous MQTTS connections with a rate of ~3,333 RPS (for both publish and receive), i.e. a message each 30 seconds for each connected client.

Since this is only a special case of the distributed test, we will only cover the specifications and some tips for the test, since the procedure to configure it is the same as we've already done.

Note: As you might already know, this test is only possible in the Kubernetes installation of dojot.

15.5.1 Hardware specifications

For dojot platform:

Machine name	Hosted services	CPU	RAM
dojot-verne-1	VerneMQ, K2V and V2K	8	8GB
dojot-verne-2	VerneMQ, K2V and V2K	8	8GB
dojot-verne-3	VerneMQ, K2V and V2K	8	8GB
dojot-x509	x509 identity manager	4	4GB
dojot-kafka	Kafka and Zookeeper	6	6GB
dojot-dojot	The rest of the services	4	4GB
haproxy	Load balancer	4	4GB

For Locust, we will use 5 replicas of the same machine, with **14GB** of RAM and **9** CPUs.

15.5.2 General tips for the test

- Ansible has the `100k` tag to prepare the environment for a 100,000 clients test. It runs a minified version of dojot without some services. This was made because not all services are ready yet to support such a load.

- By sharing a persistent volume between EJBCA pods, you can scale them up to increase the certificate creation throughput.

- The certificate creation can be distributed between all the Locust machines. In our example - with 5 Locust machines - we can generate 20,000 certificates in each machine. This can greatly increase the certificate throughput if EJBCA has been scaled too.

- After generating the certificates, make sure that all machines have all the certificates. You can export them by running `generate_certs redis --export` inside `generate_certs` container.

- To check the number of certificates, run:

```
ls cert | wc -l
```

  The returned value should be `200,003`. This number includes a key and a certificate for each device, the CA certificate and the `renew` and `revoke` directories.

- It is strongly recommended to run one slave per CPU core, totaling 45 slaves in this example.

- Since the Locust web interface does not persist any data, use Grafana's Locust dashboard to keep track of your test. Check the previous section for more info on how to configure the Locust exporter.

- You can also run the test with `revoke` and `renew`. Check the repository's README for more configurations' details.