dojot Documentation

Release 0.4.3

Matheus Magalhaes

Sep 04, 2020

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.

CHAPTER 1

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.

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.

Table of Contents

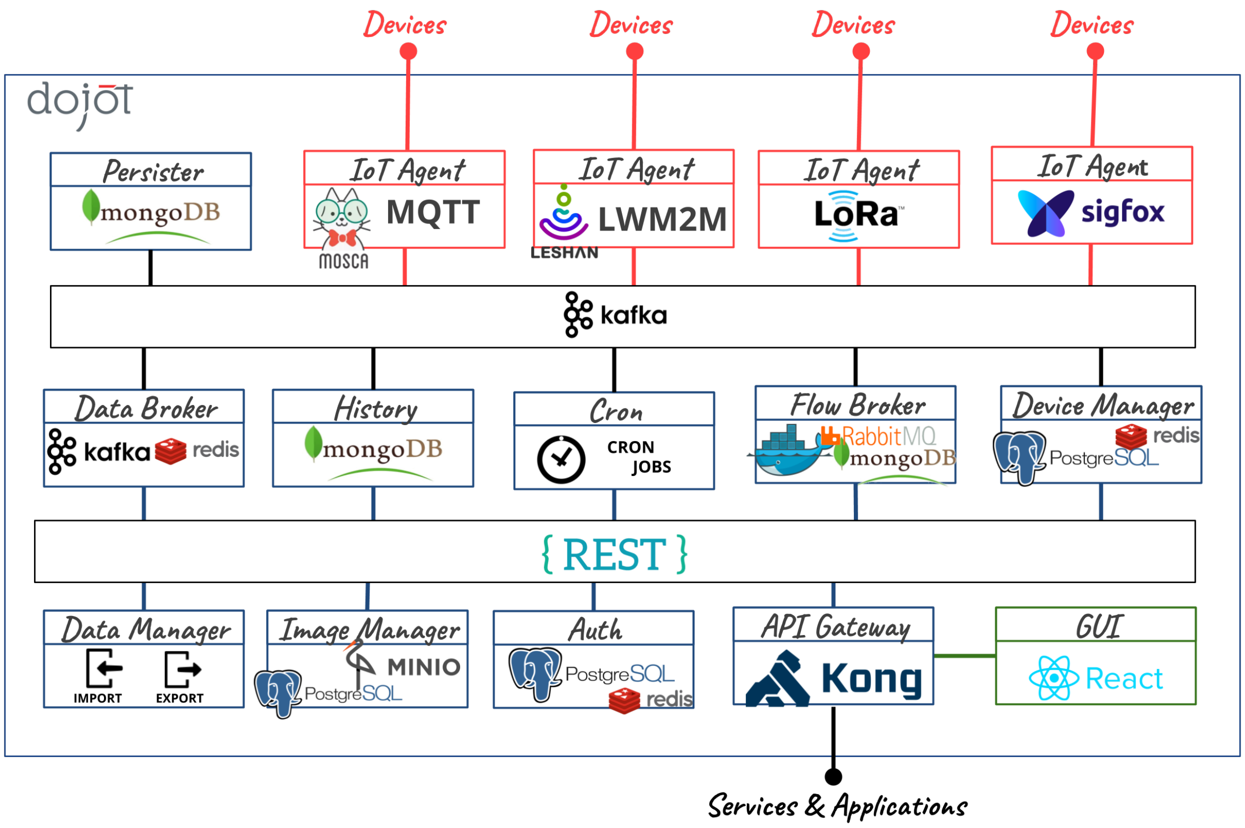


Fig. 1.1: : The microservice architecture of dojot platform.

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 save all data generated by a device onto an external database. As devices start sending their readings to dojot, a user can:

- receive these readings in real time by socket.io 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 user contexts are isolated and there are no data sharing, the access credentials are validated by the authorization service 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 between different users 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 is also 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 users 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.

It is also responsible to ensure that it communicates with devices using secure channels.

1.1.4 User Authorization Service

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

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

1.1.5 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).

1.1.6 Data Manager

This service manages the dojot's data configuration, making possible to import and export configuration.

1.1.7 Cron

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

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 Kong API Gateway

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

1.1.10 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)

1.1.11 Image manager

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

1.2 Infrastructure

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

- 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.

CHAPTER 2

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 sends 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 preprocess 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 is underlying protocol, it might be possible to do such thing in a 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 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

At start, all IoT agents (in fact, all services that need to receive or send messages related to devices) must know the list of configured tenants. This is the most basic piece of information that IoT agent needs to know in order to work properly. The request that should be sent to Auth service is this (all requests sent from dojot services to its own services should use the “dojot-management” user):

Host: Auth	
Endpoint: /admin/tenants	Method: GET
Request	
Headers	Authorization: Bearer ${JWT}
Response	
Headers	Content-Type: application/json

Body format	`tenants => *tenant`
	`  tenant => string`

A sample response for this request is:

```
{
  "tenants": [
    "admin",
    "users",
    "system"
  ]
}
```

With this list, the IoT agent can request topics for receiving device and tenant lifecycle events and for publishing new device attribute data. This is done by sending 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": "c9b2c688-9e40-4032-877a-3d262acba9d0"
}
```

Some subjects are “tenant-sensitive” (a different topic will be returned for different tenants) and some are not (the same topic will be returned regardless the tenant). DataBroker will use the tenant contained in the authorization token when dealing with tenant-sensitive subjects.

The following subjects should be used by IoT agents:

- dojot.tenancy
- dojot.device-manager.device-template
- dojot.device-manager.device
- device-data

Each one will be detailed in the following sections

dojot.tenancy

The topic related to this subject will be used to receive tenant lifecycle events. Whenever a new tenant is created or delete, the following message will be published:

Subject: dojot.tenancy	

Body format (JSON)	`type="CREATE"/"DELETE"`
	`tenant=>string`

This subject is not tenant-sensitive. A sample message received by this topic is:

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

dojot.device-manager.device-template

Subject: dojot.device-manager.device-template	

Body format (JSON)	`event => "create"`
	`data => id label attrs`
	`  id => string`
	`  label => string`
	`  attrs => [*template_attrs]`

dojot.device-manager.devices

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 label templates attrs created`
	`  id => string`
	`  label => string`
	`  templates => *number`
	`  attrs => [*template_attrs]`
	`  created => iso_date`

Body format (JSON)	`event => "remove"`
	`meta => service`
	`  service => string`
	`data => id`
	`  id => string`

Body format (JSON)	`event => "actuate"`
	`meta => service`
	`  service => string`
	`data => id`
	`  id => string`
	`  attrs => *device_attrs`

The template_attrs is a simple key/value JSON with template ID as key and the following structure as value:

```
{
  "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
}
```

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

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

This subject is tenant-sensitive.

A sample message received by this topic is:

```
{
  "event": "create",
  "meta": {
    "service": "admin"
  },
  "data": {
    "id": "efac",
```

(continues on next page)

(continued from previous page)

```
    "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 recv_time`
	`  deviceid => string`
	`  tenant => string`
	`  timestamp => unix_timestamp`
	`  recv_time => unix_timestamp`
	`attrs => *device_attrs`

This subject is tenant-sensitive. The timestamp is associated to when the attribute values were gathered by the device (this could be done by the device itself or by the IoT agent, if no timestamp was defined by the device). The recv_time attribute indicates when the message was received.

A sample message received by this topic is:

```
{
  "metadata": {
    "deviceid": "c6ea4b",
    "tenant": "admin",
    "timestamp": 1528226137452,
    "recv_time": 1528226137462
  },
  "attrs": {
    "humidity": 60
  }
}
```

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 in node.js **recommended** (https://github.com/dojot/iotagent-nodejs) and java (https://github.com/dojot/iotagent-java) to facilitate the development of an iotAgent.

CHAPTER 3

Concepts

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

Table of Contents

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.

In the next section, there is an explanation of a few basic entities in dojot: devices, templates and flows. With these concepts in mind, we present a small tutorial to how to use them in dojot - it only covers API access. There a GUI oriented tutorial in *Using web interface* tutorial.

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

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

- User identity
- Validation data
- Token expiration date

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

3.1.2 Devices and templates

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

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

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

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

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

DeviceManager documentation explains in more depth all the available operations.

3.1.3 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.

CHAPTER 4

Components and APIs

4.1 Components

Table 4.1: Components

Component	Repository / Main site	Documentation for Component	Component API Documentation
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.	
Auth	GitHub - auth	Auth doc.	API - auth
History	GitHub - history		API - history
Device Manager	GitHub - DeviceManager	DeviceManager doc.	API - DeviceManager
Image Manager	GitHub - image-manager		API - image-manager
GUI	GitHub - GUI		
Flowbroker	GitHub - flowbroker		API - flowbroker
Databroker	GitHub - data-broker		API - data-broker
Iotagent Mosca (MQTT)	GitHub - iotagent-mosca		
EJBCA-REST	GitHub - EJBCA-REST		API - EJBCA-REST
Data Manager	GitHub - Data Manager		API - Data Manager
Cron	GitHub - Cron		API - Cron

4.2 Exposed APIs (API Gateway)

The API gateway used in dojot reroutes some of endpoints from component. The following table shows which **endpoint exposed 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 API in Using API interface.

Table 4.2: Exposed endpoints

Component	Endpoint exposed by the API gateway	Component Endpoint	Component Endpoint Documentation	Needs Authentication
GUI	/	/		No
Device Manager	/device	/device	API - DeviceManager	Yes
Device Manager	/template	/template	API - DeviceManager	Yes
Flowbroker	**/flows**	/	API - flowbroker	Yes
Auth	**/auth**	/	API - auth	No
Auth	**/auth**/revoke	/revoke		No
Auth	**/auth**/user	/user	API - auth	Yes
Auth	**/auth**/pap	/pap	API - auth	Yes
History	**/history**	/	API - history	Yes
EJBCA REST	/sign	/sign	API - EJBCA-REST	Yes
EJBCA REST	/ca	/ca	API - EJBCA-REST	Yes
EJBCA REST	/user	/user	API - EJBCA-REST	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		Yes
Data Broker	/subscription	/subscription		Yes
Data Broker	/stream	/stream		Yes
Data Broker	/socket.io	/socket.io	API - data-broker	No

NOTE: Some of the endpoints from component 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/
Device Manager	host:5000/device	host:8000/device
Device Manager	host:5000/template	host:8000/template
Flowbroker	host:80/	host:8000/flows
Auth	host:5000/	host:8000/auth
Auth	host:5000/auth/revoke	host:8000/auth/revoke
Auth	host:5000/user	host:8000/auth/user
Auth	host:5000/pap	host:8000/auth/pap
History	host:8000/	host:8000/history
EJBCA REST	host:5583/sign	host:8000/sign
EJBCA REST	host:5583/ca	host:8000/ca
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

4.3 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 4.4: Original endpoints

Component	Message	Subject
DeviceManager	Device CRUD (Messages - DeviceManager)	`dojot.device-manager.device`
iotagent-mosca	Device data update (Messages - iotagent-mosca)	`device-data`
auth	Tenants creation/removal (Messages - auth)	`dojot.tenancy`

CHAPTER 5

Internal communication

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

5.1 Components

The main components current in dojot are shown in Fig. 5.1.

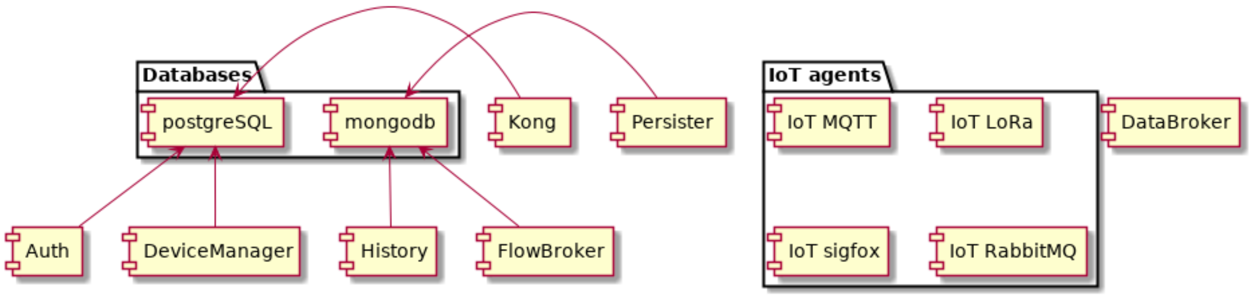


Fig. 5.1: dojot components

They are:

- Auth: authentication mechanism
- 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.

Each service 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 create an access token, described here. There is no further considerations beyond following the API description associated to each service. This can be seen in figure Fig. 5.2. Note that all interactions depicted here are abstractions of the actual ones. Also, it should be noted that these interactions are valid only for internal components. Any external service should use Kong as entrypoint.

In this figure, a client retrieves an access token for user *admin* whose password is *p4ssw0rd*. After that, a user can send a request to HTTP APIs using it. This is shown in Fig. 5.3. Note: the actual authorization mechanism is detailed in *Auth + API gateway (Kong)*.

In this figure, a client creates a new device using the token retrieved in Fig. 5.2. This request is analyzed by Kong, which will invoke Auth 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. 5.4;

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. DataBroker will invoke Redis to check whether this topic is already created and check whether dojot administrator had created a profile to this particular tuple `{tenant, subject}`.

The two profile schemes available are shown in Fig. 5.5 and Fig. 5.6.

The automatic scheme set the number of Kafka partitions to be used to the topic being created, as well as the replication factor (how many replicas will be there for each topic partition). It's up to Kafka to decide which partition and replica will be assigned to which broker instance. You can check Kafka partitions and replicas in order to know a bit more about partition and replicas. Of course you can check Kafka's official documentation.

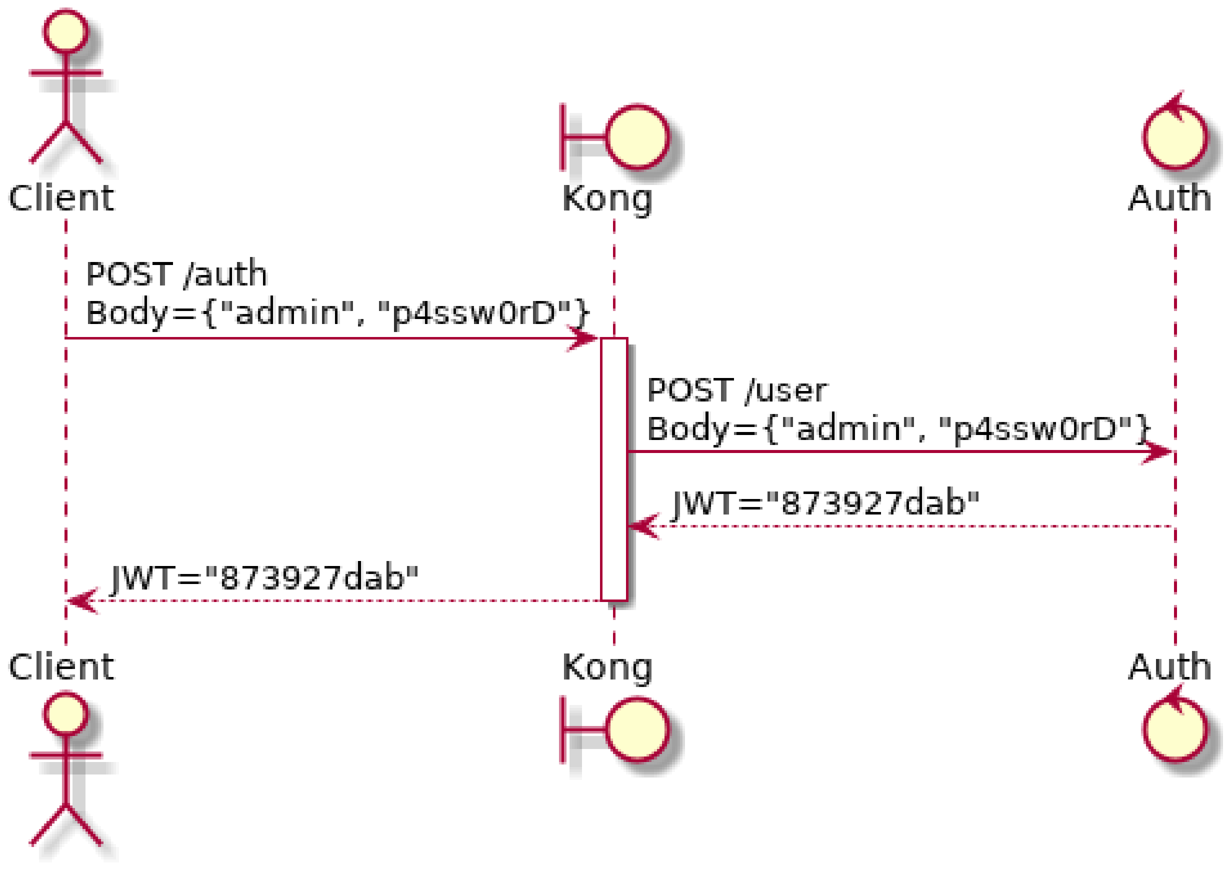


Fig. 5.2: Initial authentication

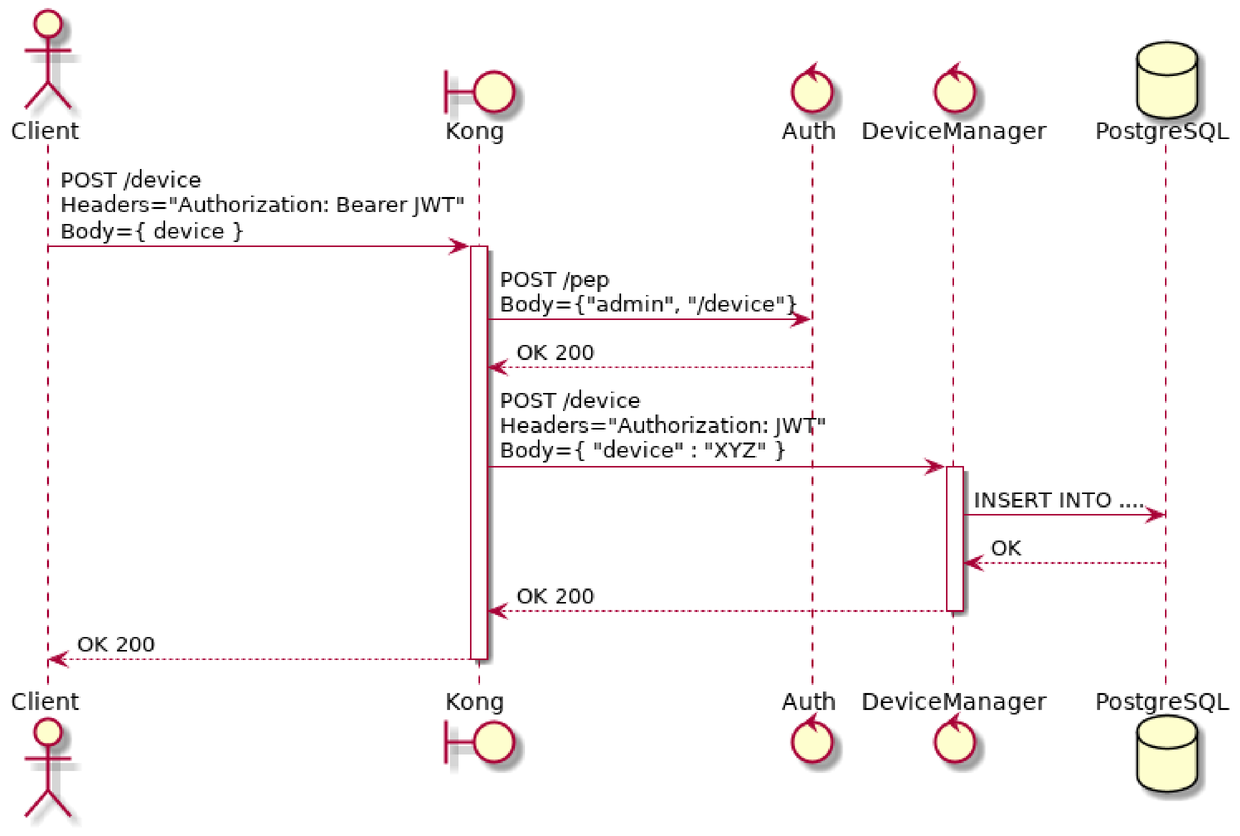


Fig. 5.3: Sending messages to HTTP API

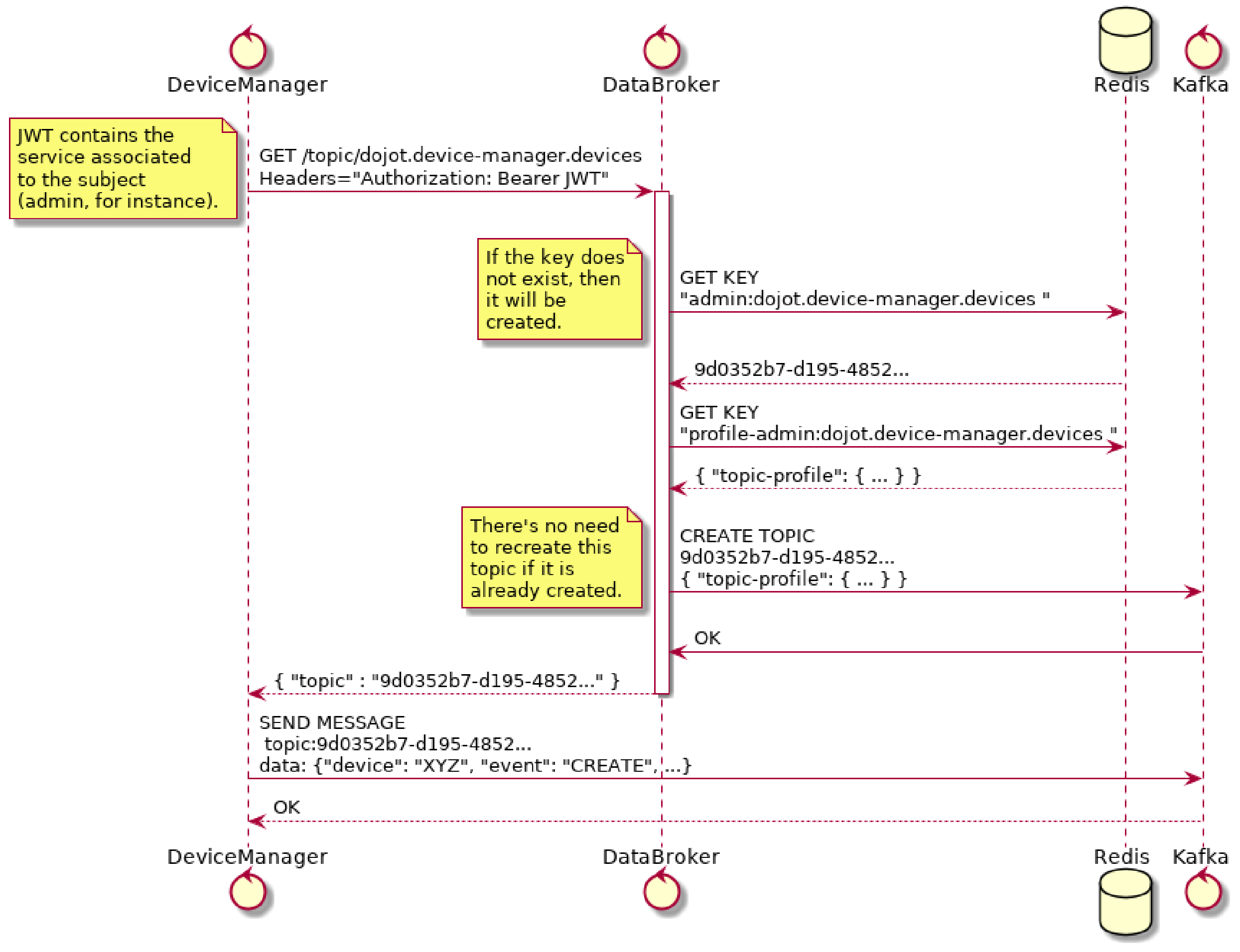

DeviceManager
DataBroker
Redis
Kafka
JWT contains the service associated to the subject (admin, for instance).
GET /topic/dojot.device-manager.devices
Headers="Authorization: Bearer JWT"
If the key does not exist, then it will be created.
GET KEY
"admin:dojot.device-manager.devices "
9d0352b7-d195-4852...
GET KEY
"profile-admin:dojot.device-manager.devices "
{ "topic-profile": { ... } }
There's no need to recreate this topic if it is already created.
CREATE TOPIC
9d0352b7-d195-4852...
{ "topic-profile": { ... } }
OK
{ "topic" : "9d0352b7-d195-4852..." }
SEND MESSAGE
topic:9d0352b7-d195-4852...
data: {"device": "XYZ", "event": "CREATE", ...}
OK

Fig. 5.4: Retrieving Kafka topics

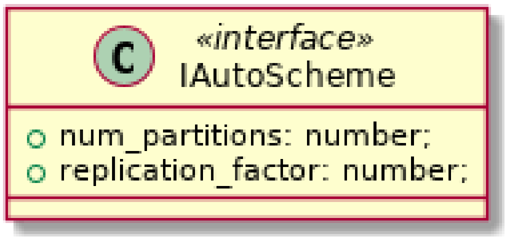


Fig. 5.5: Automatic scheme profile

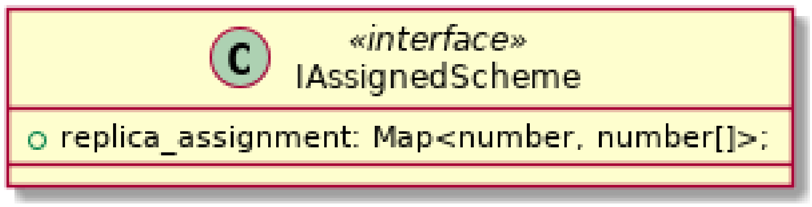


Fig. 5.6: Assigned scheme profile

The assigned scheme indicates which partition will be allocated to which Kafka instance. This includes also replicas (partitions with more than one associated Kafka instance).

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. This is done in two phases: component boot time and component runtime.

In the first phase, a component asks Auth in order to retrieve all currently configured tenants. It is interested, let's say, in consuming messages from *device-data* and *dojot.device-manager.devices* subjects. Therefore, it will request DataBroker a topic for each tenant for each subject. With that list of topics, it can create Producers and Consumers to send and receives messages through those topics. This is shown by Fig. 5.7.

The second phase starts after startup and its purpose is to process all messages received through Kafka. This will include any tenant that is created after all services are up and running. Fig. 5.8 shows how to deal with these messages.

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

5.3 Auth + API gateway (Kong)

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

5.3.1 Kong configuration

There are two configuration procedures when starting Kong within dojot:

1. Migrating existing data
2. Registering API endpoints and plugins.

The first task is performed by simply invoking Kong with a special flag.

The second one is performed by executing a configuration script *kong.config.sh*. Its only purpose is to register endpoints in Kong, such as:

```
(curl -o /dev/null ${kong}/apis -sS -X POST \
    --header "Content-Type: application/json" \
    -d @- ) <<PAYLOAD
{
    "name": "data-broker",
    "uris": ["/device/(.*)/latest", "/subscription"],
    "strip_uri": false,
    "upstream_url": "http://data-broker:80"
}
PAYLOAD
```

This command will register the endpoint */device/*/latest* and */subscription* and all requests to it are going to be forwarded to *http//data-broker:80*. You can check the documentation on how to add endpoints in Kong's documentation.

For some of its registered endpoints, *kong.config.sh* will add two plugins to selected endpoints:

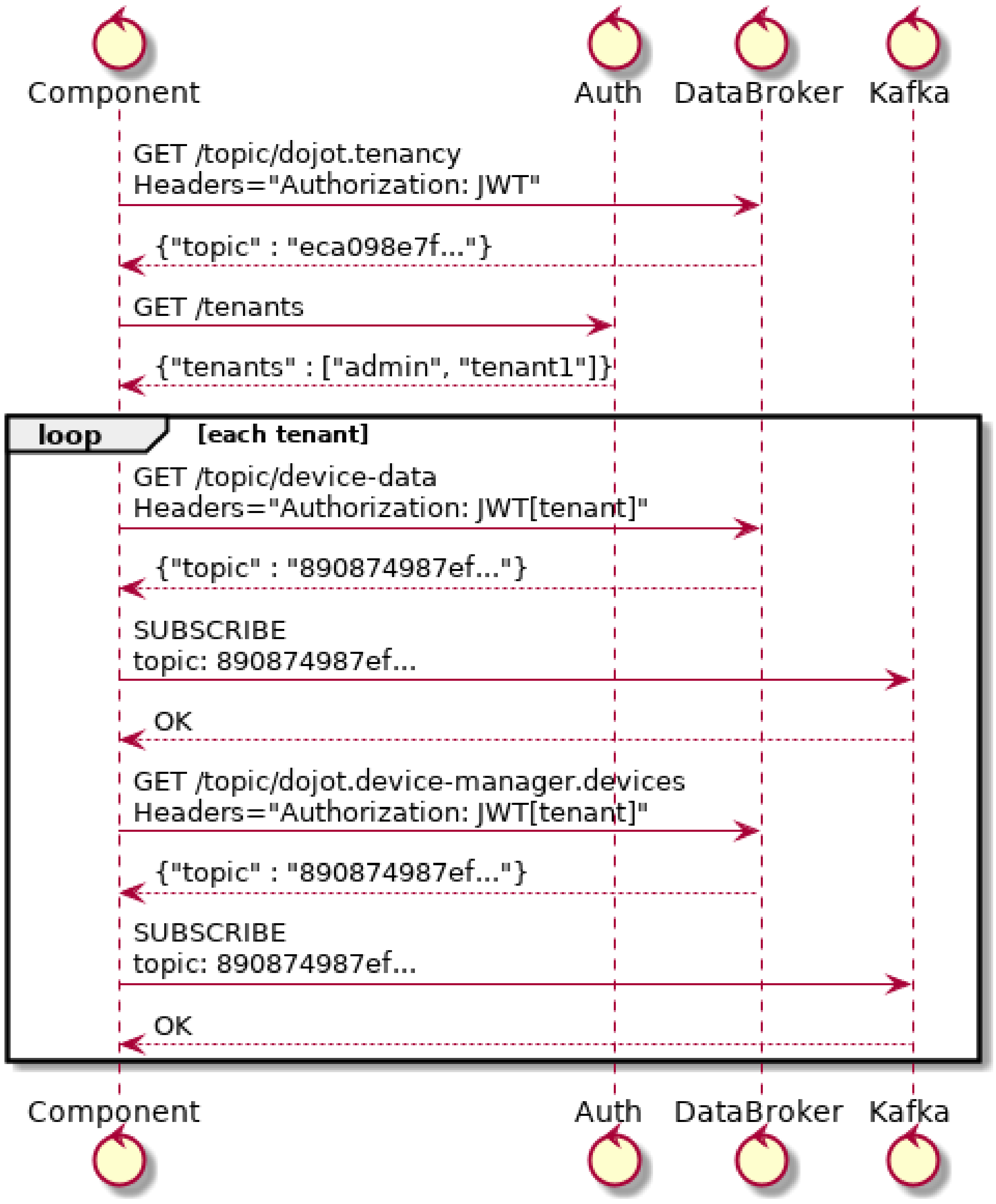

Component
Auth
DataBroker
Kafka
GET /topic/dojot.tenancy
Headers="Authorization: JWT"
{"topic" : "eca098e7f..."}
GET /tenants
{"tenants" : ["admin", "tenant1"]}
loop
[each tenant]
GET /topic/device-data
Headers="Authorization: JWT[tenant]"
{"topic" : "890874987ef..."}
SUBSCRIBE
topic: 890874987ef...
OK
GET /topic/dojot.device-manager.devices
Headers="Authorization: JWT[tenant]"
{"topic" : "890874987ef..."}
SUBSCRIBE
topic: 890874987ef...
OK

Fig. 5.7: Tenant bootstrapping at startup

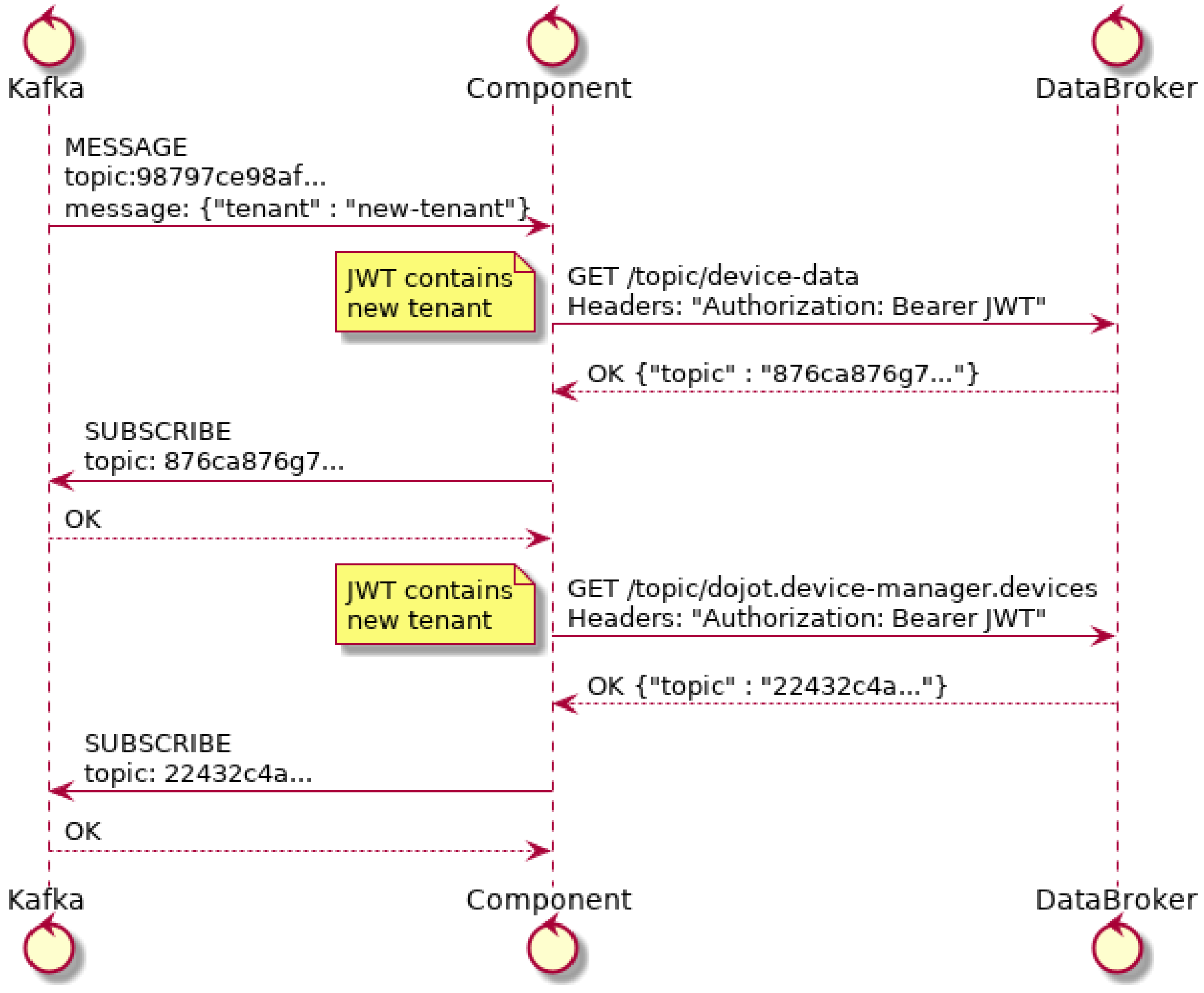

Kafka
Component
DataBroker
MESSAGE
topic:98797ce98af...
message: {"tenant" : "new-tenant"}
JWT contains new tenant
GET /topic/device-data
Headers: "Authorization: Bearer JWT"
OK {"topic" : "876ca876g7..."}
SUBSCRIBE
topic: 876ca876g7...
OK
JWT contains new tenant
GET /topic/dojot.device-manager.devices
Headers: "Authorization: Bearer JWT"
OK {"topic" : "22432c4a..."}
SUBSCRIBE
topic: 22432c4a...
OK
Kafka
Component
DataBroker

Fig. 5.8: Tenant bootstrapping

1. JWT generation. The documentation for this plugin is available at Kong JWT plugin page.
2. Configuration a plugin which will forward all policies requests to Auth. will invoke Auth in order to authenticate requests. This plugin is available in PEP-Kong repository.

The following request install these two plugins in data-broker API:

```
curl -o /dev/null -sS -X POST ${kong}/apis/data-broker/plugins -d "name=jwt"
curl -o /dev/null -sS -X POST ${kong}/apis/data-broker/plugins -d "name=pepkong" -d
↪"config.pdpUrl=http://auth:5000/pdp"
```

Emitted messages

Auth will emit just one message via Kafka for tenant creation:

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

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.

All messages published by Device Manager to Kafka can be seen in Device Manager messages.

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. 5.9.

The data sent by IoT agent has the structure shown in Fig. 5.10.

Such message would be:

```
{
    "metadata": {
        "deviceid": "c6ea4b",
        "tenant": "admin",
        "timestamp": 1528226137452
    },
    "attrs": {
        "humidity": 60,
        "temperature" : 23
```

(continues on next page)

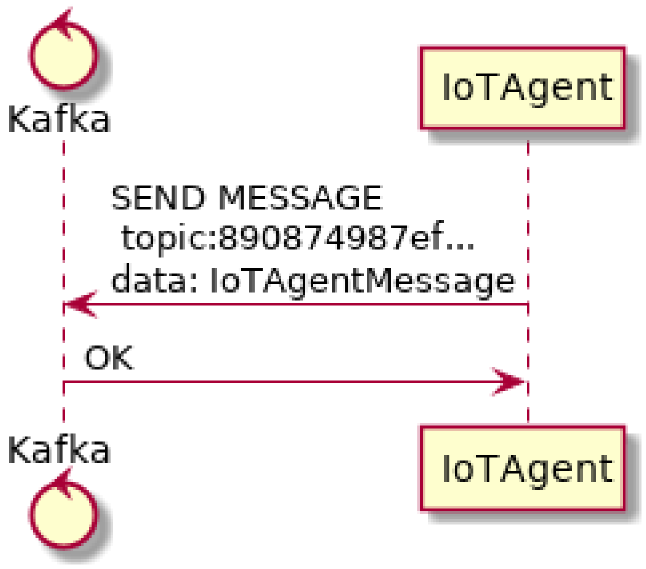


Fig. 5.9: IoT agent message to Kafka

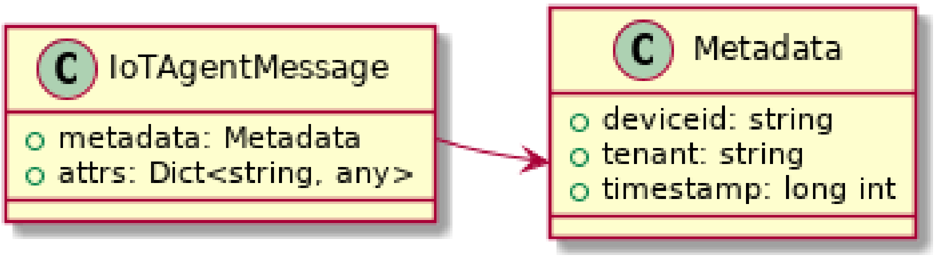


Fig. 5.10: IoT agent message structure

(continued from previous page)

```
    }
}
```

5.6 Persister

Persister is a very simple service which only purpose is to receive messages from devices (using `device-data` subject) and store them into MongoDB. For that, the bootstrapping procedure (detailed in *Bootstrapping tenants*) is performed and, whenever a new message is received, it will create a new Mongo document and store it into the device's collection. This is shown in Fig. 5.11.

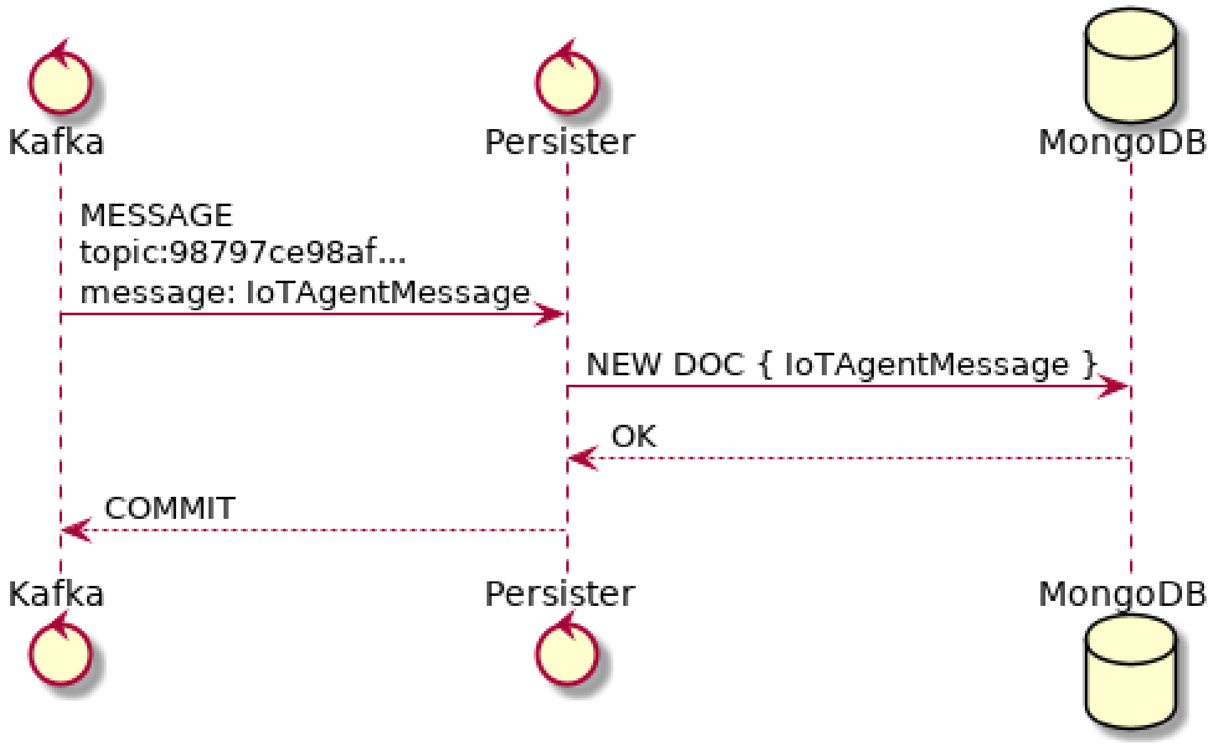


Fig. 5.11: Persister

This service is simple as it is by design.

5.7 History

History is also a very simple service: whenever a user or application sends a request to it, it will query MongoDB and build a proper message to send back to the user/application. This is shown in Fig. 5.12.

5.8 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*

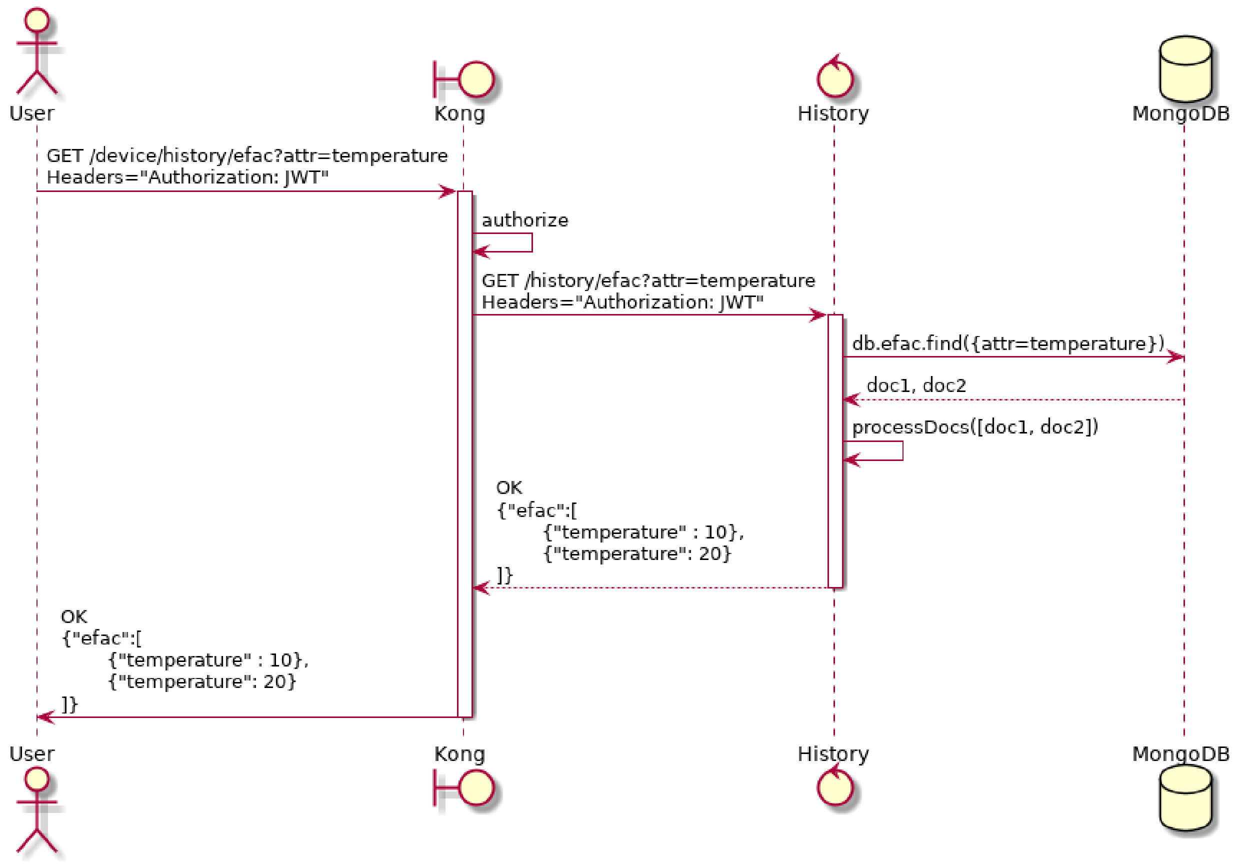

User
Kong
History
MongoDB
GET /device/history/efac?attr=temperature
Headers="Authorization: JWT"
authorize
GET /history/efac?attr=temperature
Headers="Authorization: JWT"
db.efac.find({attr=temperature})
doc1, doc2
processDocs([doc1, doc2])
OK
{"efac":[
{"temperature" : 10},
{"temperature": 20}
]}
OK
{"efac":[
{"temperature" : 10},
{"temperature": 20}
]}
User
Kong
History
MongoDB

Fig. 5.12: History

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.

CHAPTER 6

Installation Guide

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

Table of Contents

6.1 Hardware requirements

The estimated hardware requirements for 500 devices with updates every 15s are:

Table 6.1: Hardware requirements for 500 devices

Deployment		CPU	RAM	Free disk space
Docker-compose		4 Cores	4GB	10GB
Kubernetes	Master	2 Cores	2GB	2GB
Kubernetes	Worker	4 Cores	4GB	10GB

In addition, you need:

- **Network access**
- **The following ports should be opened:**
  - **TCP**: 8000 *(web interface access)*; 1883 *(MQTT, If you are going to use MQTT)*; 5896 *(LW2M, If you are going to use LW2M file server via HTTP instead of coap, UDP)*.
  - **TLS**: 8883 *(MQTTS, If you are going to use MQTT with TLS, in secure mode.)*.
  - **UDP**: 5683 and 5693 *(LW2M, If you are going to use LW2M)*; 5684 and 5694 *(LW2M, If you are going to use LW2M with DTLS)*.

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

6.2 Docker compose

In this video tutorial above, version v0.4.2 is used, but the same video is valid for the current version, it is only necessary to change to version v0.4.3.

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

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

This guide has been checked on an Ubuntu 16.04 and 18.04 LTS environment.

The following sections describe all Docker compose dependencies.

6.2.1 Docker engine

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

https://docs.docker.com/engine/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 docker-compose commands should be run by the super user (root), or as sudo.

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

6.2.2 Docker Compose

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

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

6.2.3 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 tagname has to be replaced):

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

git checkout tag_name -b branch_name
```

For instance:

```
git checkout v0.4.3 -b v0.4.3
```

That done, the environment can be brought up by:

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

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

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

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

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

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

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

6.2.4 Usage

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

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

6.3 Kubernetes

In this video tutorial above, version v0.4.2 is used, but the same video is valid for the current version, it is only necessary to change to version v0.4.3

This section provides instructions on how to create a dojot deployment on a multi-node environment, using Kubernetes as the orchestration platform.

This deployment option when properly configured can be used for creating production environments.

The following sections describe all dependencies and steps required for this deployment.

6.3.1 Kubernetes Cluster

For this guide it is advised that you already have a working K8s cluster.

If you need to build a Kubernetes cluster from scratch, up to date information and installation procedures can be found at Kubernetes setup documentation.

6.3.2 Kubernetes Requirements

- The Kubernetes supported version is between **v1.11** and **v1.15** .
- Access to Docker Hub repositories
- (optional) a storage class that will be used for persistent storage

6.3.3 dojot Deployment

To deploy dojot to Kubernetes it is advised the use of ansible playbooks developed for dojot. The playbooks and all the related code can be found on the repository Ansible dojot.

The following steps will describe how to use this repository and its playbooks.

1. Cloning the repository

The first deployment step is cloning the repository. To do so, execute the command:

```
git clone -b v0.4.3 https://github.com/dojot/ansible-dojot
```

2. Installing dependencies

The next step is installing the dependencies for running the ansible playbook, this dependencies include ansible itself with other modules that will be used to parse templates and communicate with kubernetes.

Enter the folder where the repository was downloaded and install the pip packages with the following commands:

```
cd ansible-dojot
pip install -r requirements.txt
```

3. Configuring the inventory

For deploying kubernetes with ansible, it is necessary to model your desired environment on an ansible inventory.

In the repository there is an '*inventory*' folder containing an example inventory called '*example_local*' that can be used as the starting point to creating the real environment inventory.

The first file that requires changes is the hosts.yaml. This file describes the nodes that will be accessed by ansible to perform the deployment. As the dojot deployment is done directly to K8s, only a node with access to the kubernetes cluster is actually required.

The node that will access the cluster might be a kubernetes cluster node that is accessible via SSH or event your local machine if it can reach the kubernetes cluster with a configuration file.

On the example file, the access is done via a local node, where the ansible script is executed. This node is described as localhost in the hosts item of the group **all**.

These same nodes must be added as children of the group dojot-k8s.

To configure a local access on the hosts file, follow the example below:

```
---
all:
  hosts:
    localhost:
      ansible_connection: local
      ansible_python.version.major: 3
  children:
    dojot-k8s:
      hosts:
        localhost:
```

To configure remote access via ssh to a node of the cluster, follow this other example:

```
---
all:
  hosts:
    NODE_NAME:
      ansible_host: NODE_IP
  children:
    dojot-k8s:
      hosts:
        NODE_NAME:
```

The next step is configuring the mandatory and optional variables required for deploying dojot.

There is a document describing each of the variables that can be configured at Ansible dojot variables.

This variables must be set for the group '*dojot-k8s*', to do so set their values on the file dojot.yaml on the folder '**group_vars/dojot-k8s/**'

4. Executing the deployment playbook

Now that the inventory is set, the next step is executing the deployment playbook.

To do so, run the following command:

```
ansible-playbook -K -k -i inventories/YOUR_INVENTORY deploy.yaml
```

Wait for the playbook execution to finish without errors.

5. Accessing the deployed dojot environment

Dojot access will be set using NodePorts, to view the proper ports to access the environment it is necessary to check service configuration.

```
kubectl get service -n dojot kong iotagent-mosca
```

This command will return the port used for external access to both the REST API and GUI via kong and the MQTT port via iotagent-mosca.

CHAPTER 7

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.

It takes a role as an enabler platform with:

- Open APIs which makes the access to the platform resources easy.
- Capacity to store large volumes of data in different formats.
- Connectors to different types of devices.
- Graphical user interface with flow builder to quicly 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 a whole, such as architectural enhancements.
- https://github.com/dojot/docker-compose: repository for Docker compose files and configurations. This is what we would recommend to use to start with.
- https://github.com/dojot/ansible-dojot: repository for ansible (kubernetes) files and configurations.

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 nice 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-rc.1
  v0.4.3-rc.2
  v0.4.3


$ git checkout v0.4.2
```

Once in a while we'll release new versions for dojot components. They might be independently released (although we tend to synchronize all of them). Once we end up with a stable set of component versions, we'll update the docker-compose repository.

2 Deploy the latest docker images. This command might need `sudo`.

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

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

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.

In order to send data to dojot via MQTT (using iotagent-mosca), there are some things to keep in mind:

- The topic should look like `/<tenant>/<device-id>/attrs` (for instance: `/admin/efac/attrs`). Depending on how IoT agent MQTT was started (more strict), the client ID must also be set to "<tenant>:<deviceid>", such as "admin:efac".
- MQTT payload must be a JSON with each key being an attribute of the dojot device, such as:

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

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` and `Text`: 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 in Device-Manager how-to - sending actuation 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 I saved an attribute, but it disappeared from the device. Is it a bug?

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

7.3.12 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",
```

(continues on next page)

(continued from previous page)

```
    "value": 25.76,
    "attr": "temperature"
  }
]
```

There are more operators that could be used to filter entries. Check History API 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 email.
- Notify through 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?

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

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.

Another thing that you could do is to build a flowbroker node to deal with contexts, which can be used to store and retrieve data related to a flow or node.

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?

And that's it! In the https://github.com/dojot/flowbroker/tree/master/lib, there is two examples of how to build a Docker image that could be added to flow node menu.

7.5 Applications

7.5.1 What APIs are available for applications?

You can check all available APIs in the API Listing page

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 two ways that your application could be integrated with dojot:

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

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

CHAPTER 8

Release history

8.1 carate - 2019.09.11

- IoT agents:
  - Support for LWM2M devices
- GUI:
  - New interface for devices management
  - New interface for templates management
  - Attribute Metadata Management
  - Import and export
  - Firmware update
  - Profiles
  - Notifications
  - Internationalization
  - View details with Actuator Attributes
  - Devices Filter on Map
- Flows:
  - New node Event Device In
  - New node Event Template Device
  - New node FTP
  - New node Multi Device Out
  - New node Notification
  - New node Multi Actuate

  - New node Cron
  - New node Cron Batch
  - New node Cumulative Sum
  - New node Merge data
  - Node email has been removed
  - Internacionalization
  - Support to handlebars template on template node
- History:
  - New endpoint to query notifications
- ImageManager:
  - Improvements to support Firmware Update
- DataBroker:
  - Support notifications in socket-io
- DataManager:
  - It's new a dojot's microservice that manages the dojot's data configuration, making possible to import and export configuration.
- Cron:
  - It's new a dojot's microservice that allows you to schedule events to be emitted to other microservices.
- New libraries:
  - To accelerate development: Dojot Module Java and Dojot Module Python
  - HealthCheck: HealthCheck Python e Healthcheck NodeJs
  - IoTAgent: IoTAgent Java

CHAPTER 9

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 and firmware update.

Note:

- Who is this for: entry-level users
- Level: basic
- Reading time: 20m

9.1 Device management

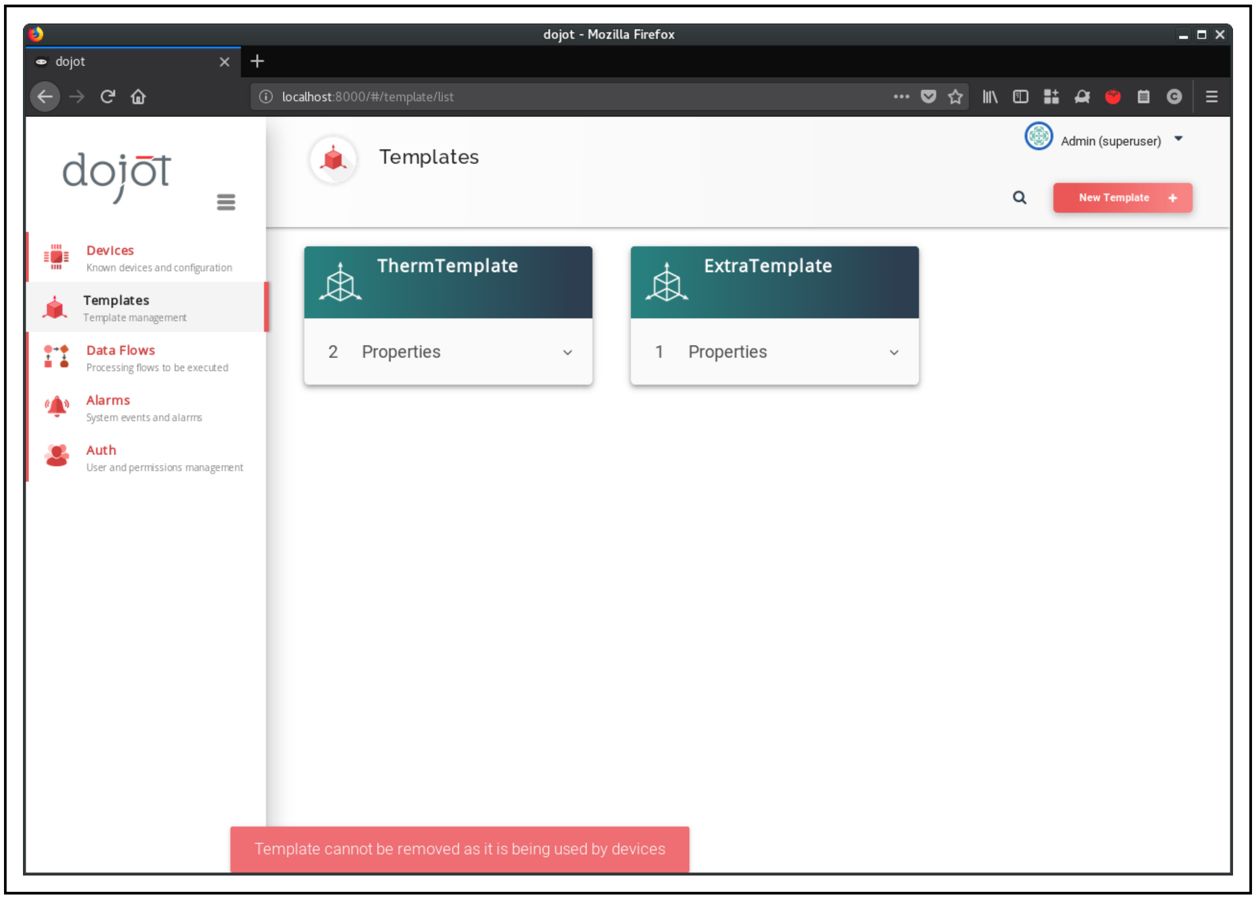
This section will show how to manage device. For this tutorial we will show how to add two thermometers and a virtual device that will represent an alarm system that will monitor both sensors.

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.

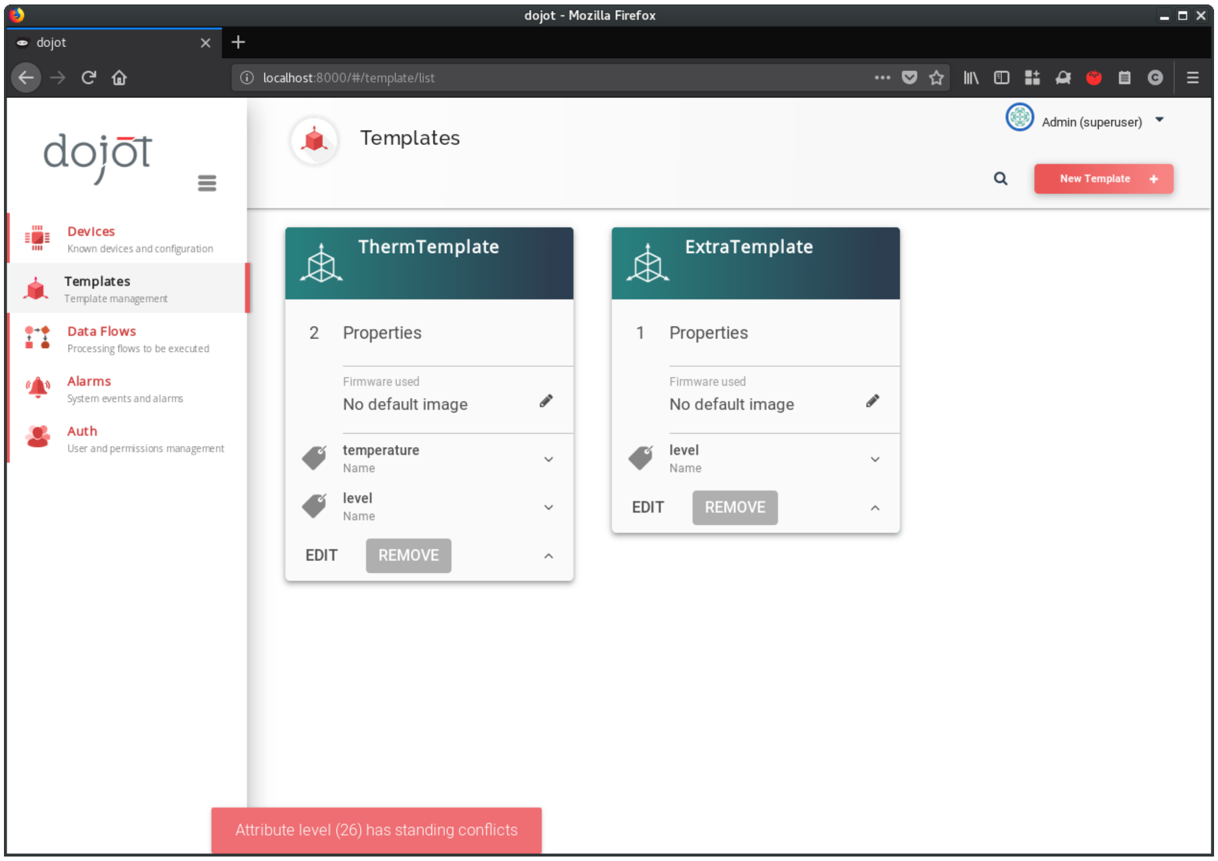
Note that, when you select the template in the right panel at device creation screen, all attributes are inherited from that device. You could add more templates as needed, keeping in mind that templates used to compose a device must not share an attribute with the same name.

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:



Template cannot be removed as it is being used by devices

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 card 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`.

Just for the sake of simplicity, we'll emulate one device using mosquitto_pub tool. We set the `client-id` parameter by using the `-i` flag of mosquitto_pub. See more about in *Using API interface* at topic Sending Messages. Besides that this examples are using MQTT without TLS, we recommend *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.

9.2 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 notification of severity 4 (mildly important) and a message indicating that the kitchen is OK. Otherwise, if the temperature is higher than 40, then a notification is sent with severity 1 (highest severity) and a message indicating that the kitchen is on fire. This is done as shown belown.

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.

9.3 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!

9.4 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.

CHAPTER 10

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

10.1 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
```

10.2 Getting access token

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

```
JWT=$(curl -s -X POST http://localhost:8000/auth \
-H 'Content-Type:application/json' \
-d '{"username": "admin", "passwd" : "admin"}' | jq -r ".jwt")
```

To check:

```
echo $JWT
```

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

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

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

10.3 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 how-to for more detailed instructions.

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

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

```
{
  "pagination": {
    "has_next": false,
    "next_page": null,
    "total": 1,
    "page": 1
  },
  "devices": [
    {
      "templates": [
        1
      ],
      "created": "2018-01-25T12:36:29.353958+00:00",
      "attrs": {
        "1": [
          {
            "template_id": "1",
            "created": "2018-01-25T12:30:42.167126+00:00",
            "label": "temperature",
            "value_type": "float",
            "type": "dynamic",
            "id": 1
          },
          {
            "template_id": "1",
            "created": "2018-01-25T12:30:42.167126+00:00",
            "label": "fan",
            "value_type": "actuator",
            "type": "float",
            "id": 2
          }
        ]
      },
      "id": "0998",
      "label": "device_0"
    }
  ]
}
```

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

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 MQTT broker is not running before starting dojot (by running commands like `ps aux | grep mosquitto`) to avoid port conflicts.

Note: To run *mosquitto_pub* and *mosquitto_sub* without using TLS, as in the examples below, you must set the environment variable ALLOW_UNSECURED_MODE with the value *'true'* to the *iotagent-mqtt* service, that is, *ALLOW_UNSECURED_MODE='true'*. You can change this value in the dojot *docker-compose.yml* file and then kill and up the docker-compose again. **By default this value is already 'true'.**

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
}
```

Let's send this message to dojot:

```
mosquitto_pub -h localhost -t /admin/0998/attrs -p 1883 -i admin:0998 -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 the mosquitto_sub:

```
mosquitto_sub -h localhost -p 1883  -i admin:0998 -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}}'
```

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: `<service>:<deviceid>`, 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 `/<service-id>/<device-id>/attrs` (for instance: `/admin/efac/attrs`).
- The topic to subscrine should assume the pattern `/<service-id>/<device-id>/config` (for instance: `/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 }
```

This examples are using MQTT without TLS, we recommend *Using MQTT with security (TLS)*.

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

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.

CHAPTER 11

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

11.1 Dojot nodes

11.1.1 Device Event in

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:

```
{
    "data": {
        "attrs": {
            "temperature": 10,
            "some-static-attr": "efac"
        }
    }
}
```

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

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

To configure the device in node, a window like Fig. 11.1 will be displayed.

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.

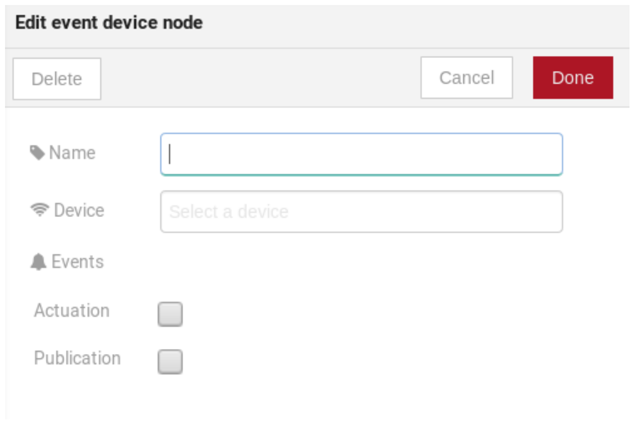
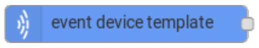


Fig. 11.1: : Device in configuration window

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

11.1.2 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.

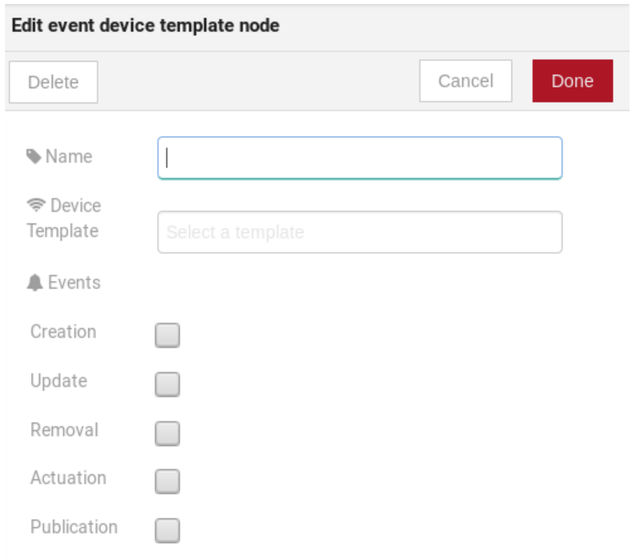


Fig. 11.2: : Device template in configuration window

Fields:

- **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.

11.1.3 Multi device out

Device out will determine wich 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*.

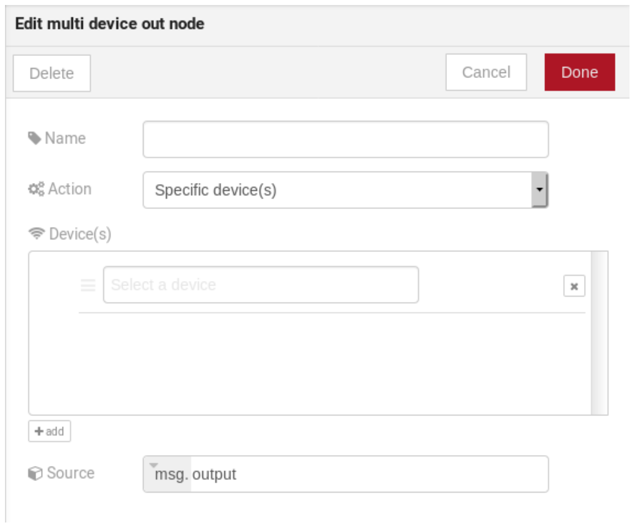


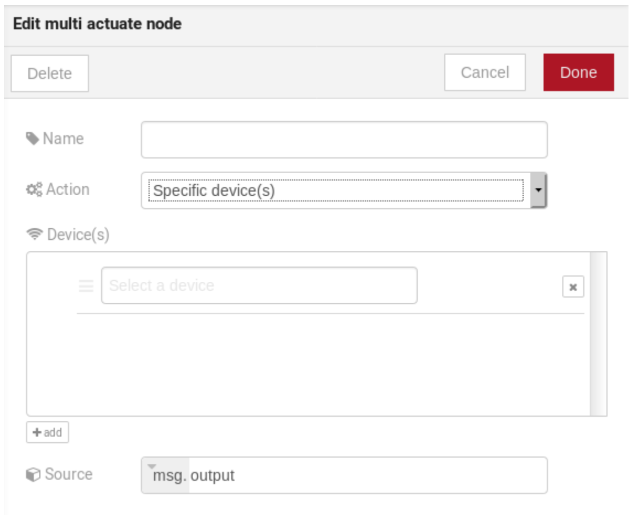
Fig. 11.3: : Device out config window

Fields:

- **Name** *(optional)*: Name of the node.
- **Action** ***(required)*: Which node 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 nodes that will receive the update.
  - *Device(s) defined during the flow*: which nodes 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

11.1.4 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.



Name

Action: Specific device(s)

Device(s): Select a device

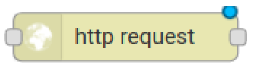
Source: msg. output

Fig. 11.4: : Actuate configuration

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 node the message will be sent to.
  - *Device(s) defined during the flow*: which nodes 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

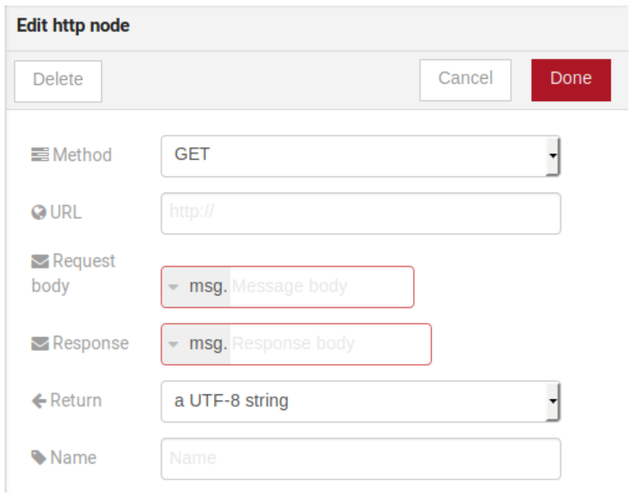
11.1.5 HTTP request



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

Fields:

- **Method** *(required)*: The http method (GET, POST, etc...).
- **URL** *(required)*: The URL that will receive the http request


Edit http node

Delete | Cancel | Done

Method: GET

URL: http://

Request body: msg. Message body

Response: msg. Response body

Return: a UTF-8 string

Name: Name

Fig. 11.5: : Http Request configuration

- **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.

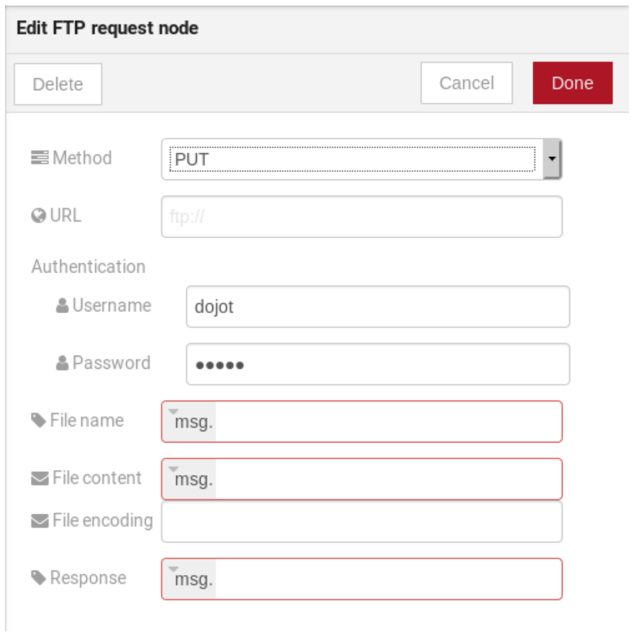
11.1.6 Ftp request

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

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.

11.1.7 Notification


Edit FTP request node

Delete Cancel Done

Method: PUT

URL: ftp://

Authentication

Username: dojot

Password: •••••

File name: msg.

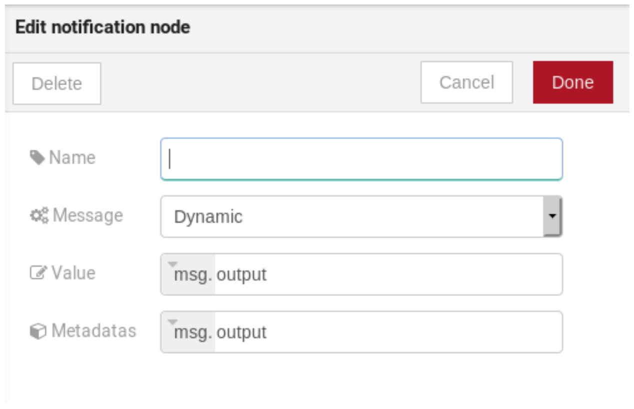
File content: msg.

File encoding:

Response: msg.

Fig. 11.6: : Device template in configuration window

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.


Edit notification node

Delete Cancel Done

Name:

Message: Dynamic

Value: msg. output

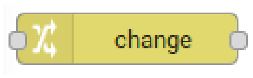
Metadatas: msg. output

Fig. 11.7: : 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

11.1.8 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.

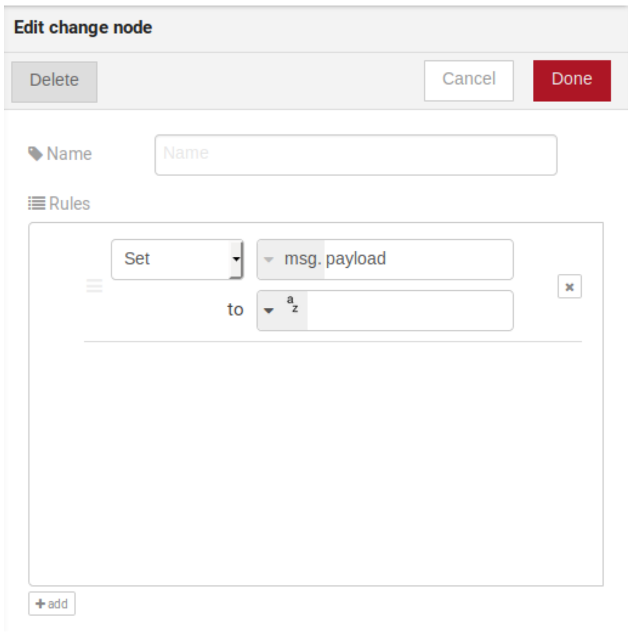


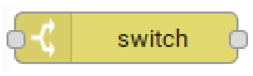
Fig. 11.8: : Change configuration

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.

11.1.9 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

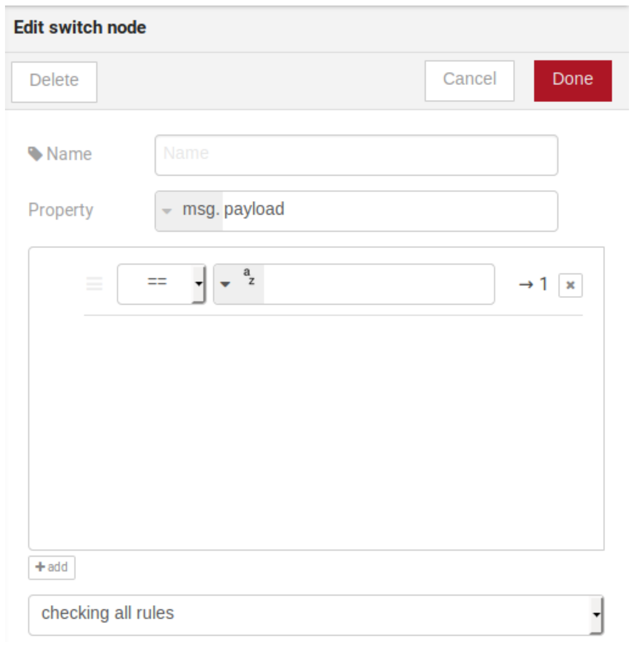


Fig. 11.9: : Switch configuration

- **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…

11.1.10 Template

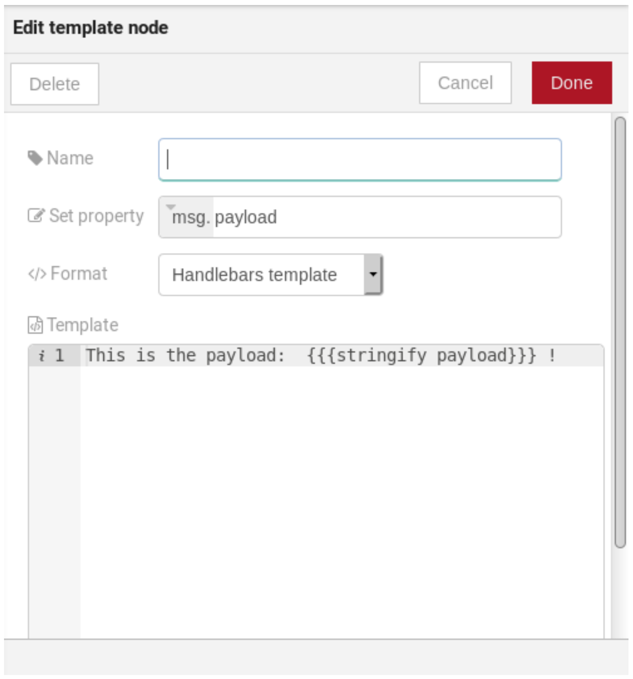
Note: Despite the name, this node has nothing to do with dojot templates

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. 11.10 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



Set property: msg. payload

Format: Handlebars template

```
This is the payload:  {{{stringify payload}}} !
```

Fig. 11.10: : Template configuration

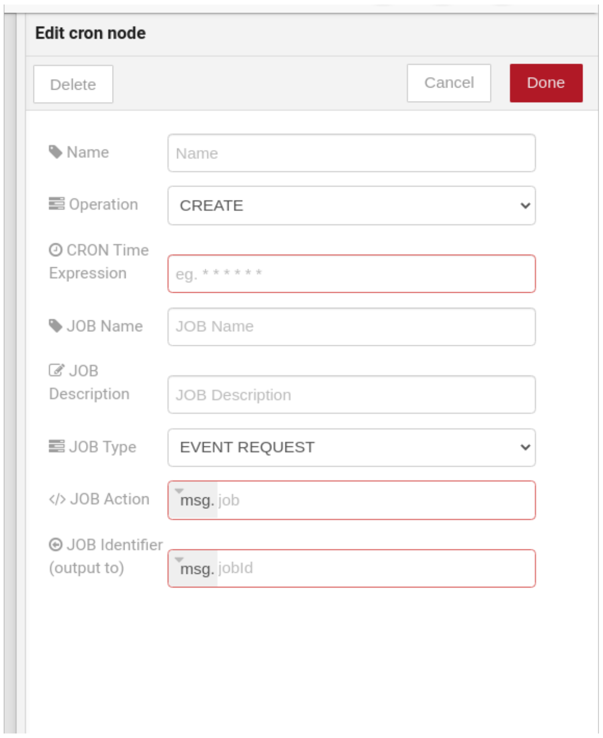
- **Format** *(required)*: Format template will be writen
- **Template** *(required)*: Value that will be assigned to the target variable set on **Set property**
- **Output as** *(required)*: The format of the output

11.1.11 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


Edit cron node

Name: Name

Operation: CREATE

CRON Time Expression: eg. * * * * * *

JOB Name: JOB Name

JOB Description: JOB Description

JOB Type: EVENT REQUEST

JOB Action: msg. job

JOB Identifier (output to): msg. jobId

Fig. 11.11: : Cron configuration

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

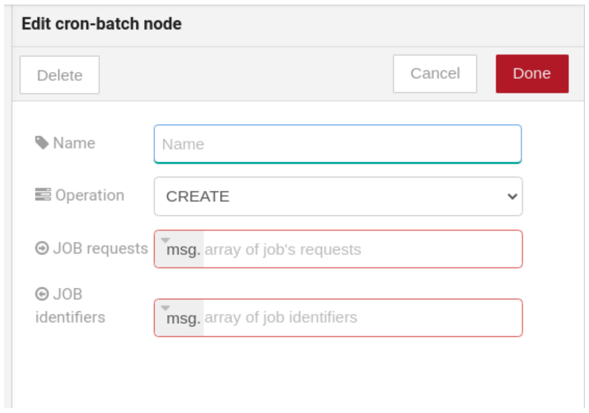
```
{
    "method": "PUT",
    "headers": {
                    "Authorization": "Bearer ${JWT}",
                    "Content-Type": "application/json"
                  },
    "url": "http://device-manager:5000/device/${DEVICE_ID}/actuate",
    "body": {
                  "attrs": {"message": "keepalive"}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

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"}
                  }
}
```

11.1.12 Cron batch

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



Name: Name
Operation: CREATE
JOB requests: msg. array of job's requests
JOB identifiers: msg. array of job identifiers

Fig. 11.12: : 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

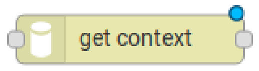
11.1.13 Geofence

Select an interest area to determine wich devices will activate the flow

Fields:

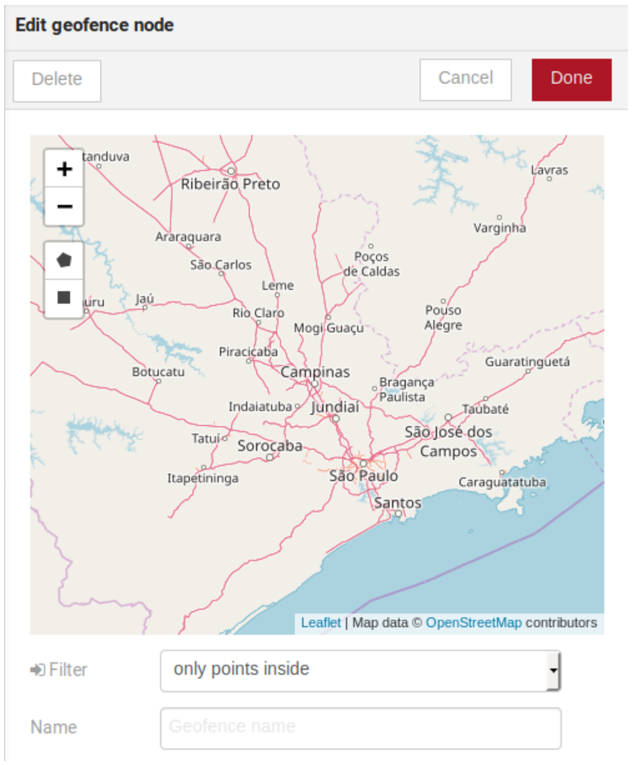
- **Area** *(required)*: Area that will be selected. It can be chosen with an square or with a pentagon.
- **Filter** *(required)*: Which side of the area will be picked: inside or outside the marked area in the field above.
- **Name** *(optional)*: Name of the node

11.1.14 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.

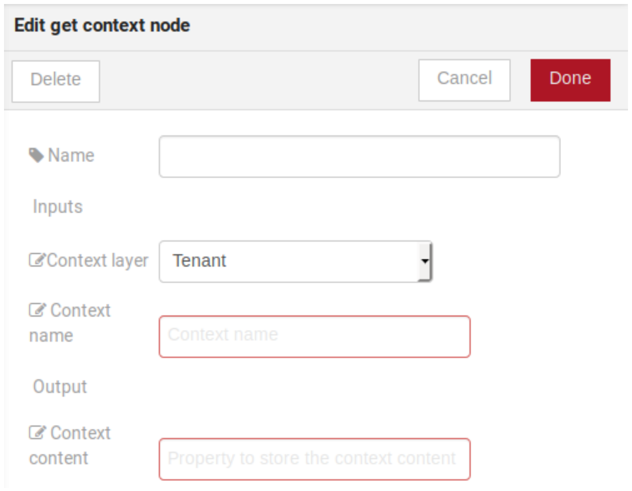
Note: Context is a mechanism that allows a given set of data to persist beyond the life of the event, thus making it possible to store a state for the elements of the solution.



Filter: only points inside

Name: Geofence name

Fig. 11.13: : Geofence configuration



Name:

Inputs

Context layer: Tenant

Context name: Context name

Output

Context content: Property to store the context content

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

11.1.15 Merge data

This node allows objects to be merged in the context.

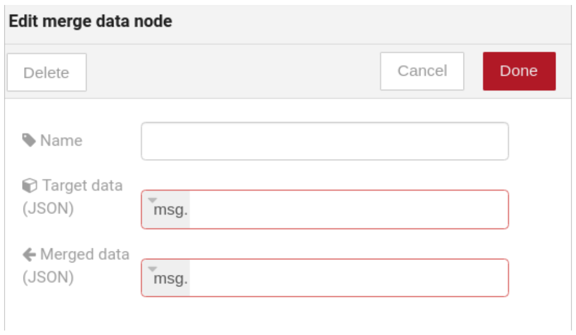
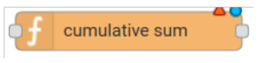

Edit merge data node

Fig. 11.14: : 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

11.1.16 Cumulative sum



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

Fields:

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

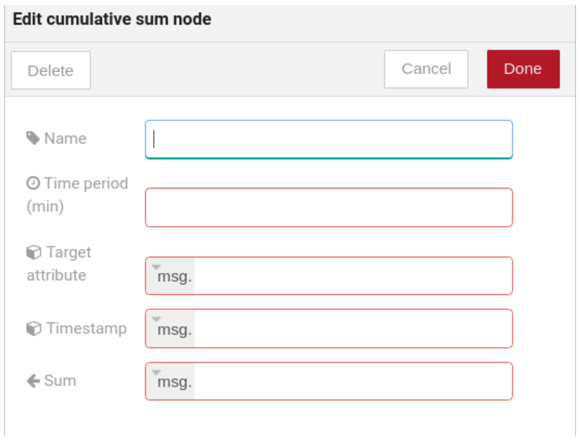


Fig. 11.15: : Cumulative sum configuration

11.1.17 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. 11.16 will be displayed.

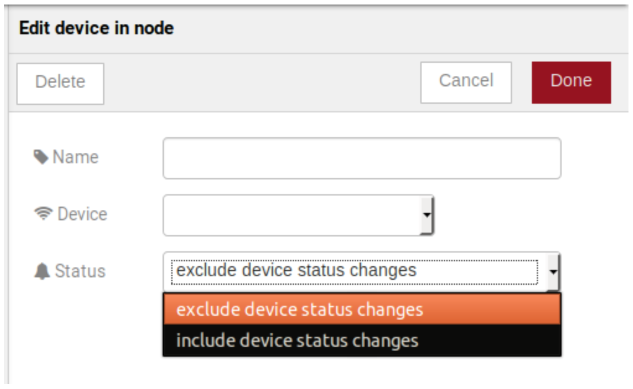

Edit device in node
Delete
Cancel
Done
Name
Device
Status
exclude device status changes
exclude device status changes
include device status changes

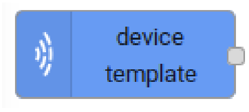
Fig. 11.16: : Device in configuration window

Fields:

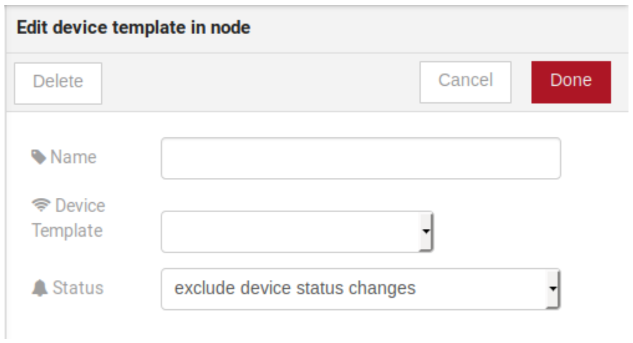
- **Name** *(optional)*: Name of the node
- **Device** *(required)*: The *dojot* device that will trigger the flow
- **Status** *(required)*: *exclude device status changes* will not use device status changes (online, offline) to trigger the flow. On the other hand, *include devices status changes* will use these status to trigger the flow.

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.



Name

Device Template

Status: exclude device status changes

Fig. 11.17: : 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 wich 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*.

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

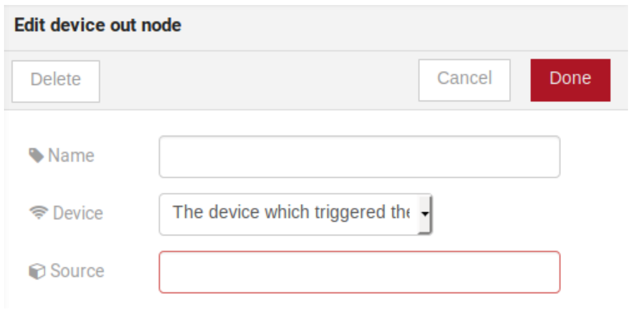


Fig. 11.18: : Device out config window

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.

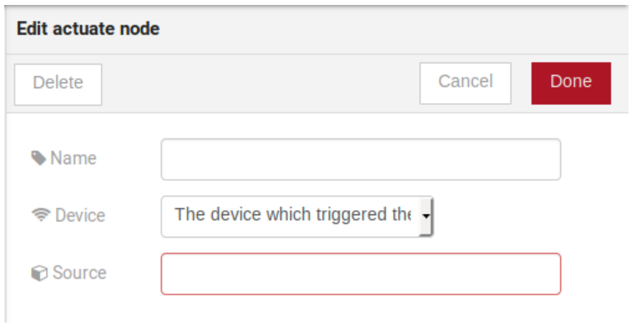


Fig. 11.19: : Actuate configuration

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

11.2 Learn by examples

11.2.1 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.

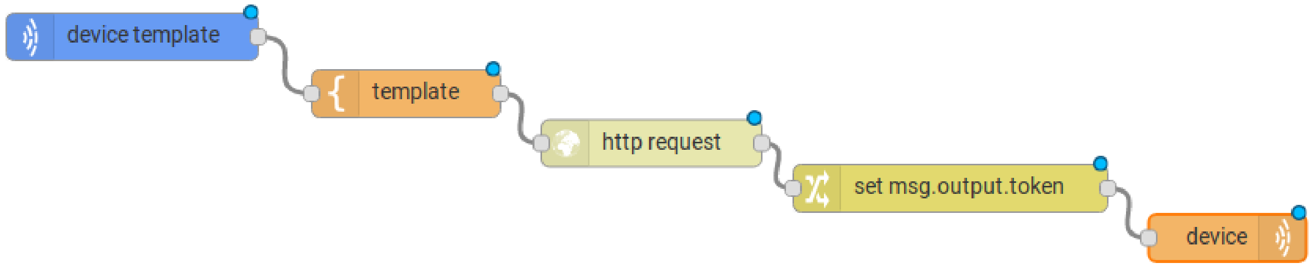


Fig. 11.20: : Flow used to explain http node

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. 11.23 the token that came in the response is assigned to the attribute token of the virtual device.

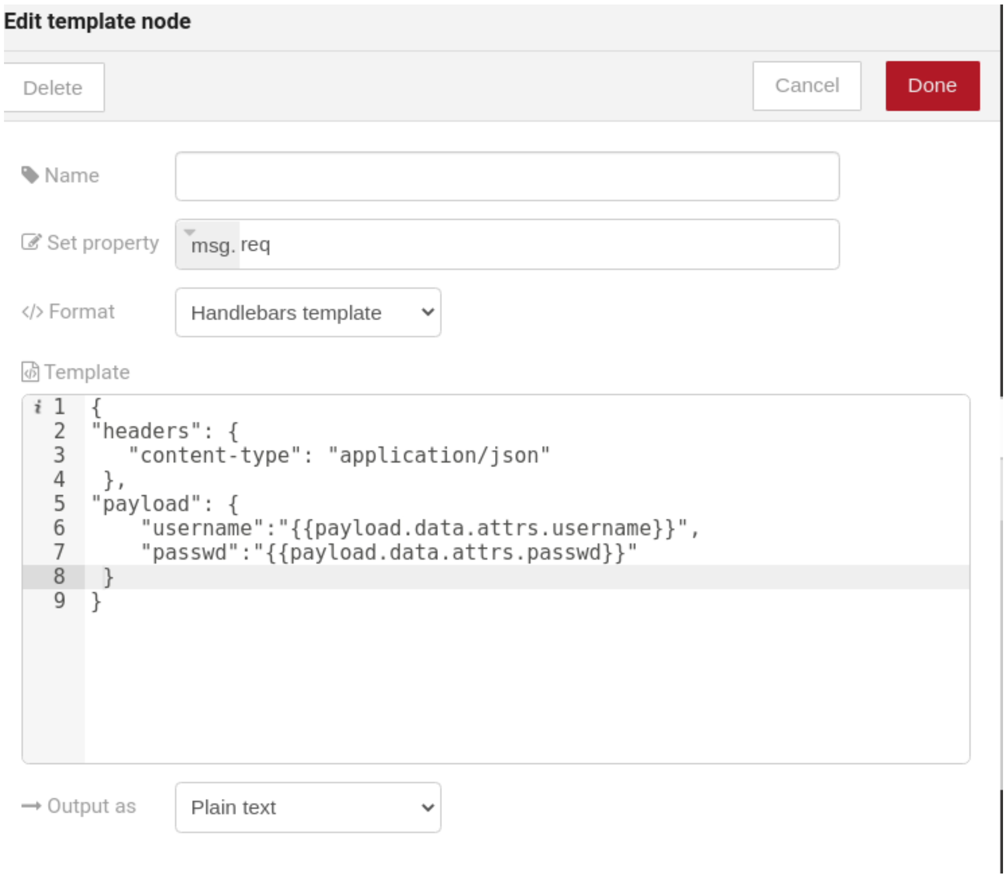
Then, the result of the flow is the attribute *token* of the virtual device be updated with the token that came in the response of the http request:

11.2.2 Using geofence node

A good example to learn how geofence node works ia studying the flow below:

The geofence node named *in area* is set like seem in Fig. 11.27. 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.



Name

Set property: msg. req

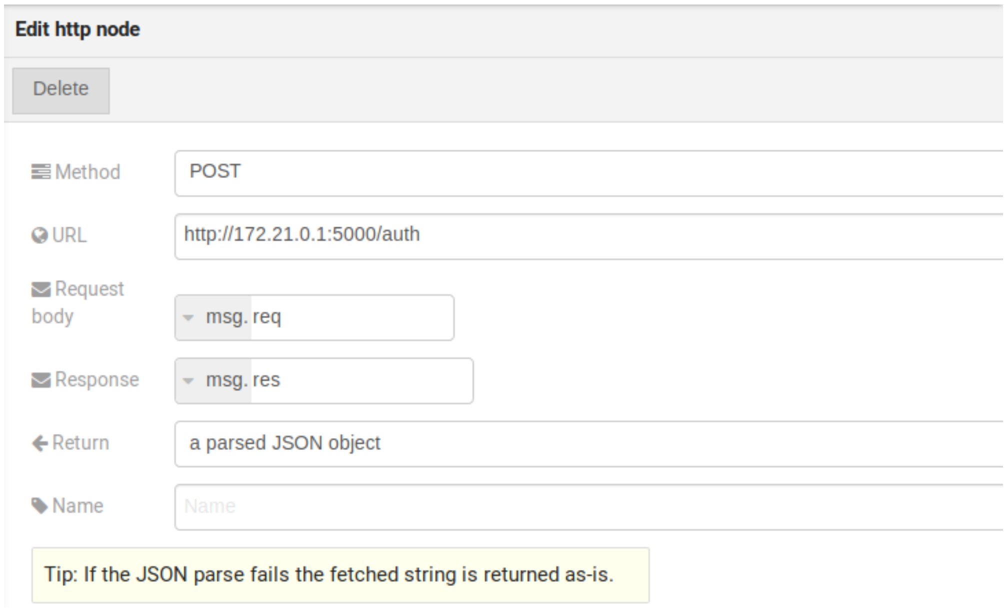
Format: Handlebars template

Template

```
{
"headers": {
   "content-type": "application/json"
 },
"payload": {
    "username":"{{payload.data.attrs.username}}",
    "passwd":"{{payload.data.attrs.passwd}}"
 }
}
```

Output as: Plain text

Fig. 11.21: : Template node configuration



Method: POST

URL: http://172.21.0.1:5000/auth

Request body: msg. req

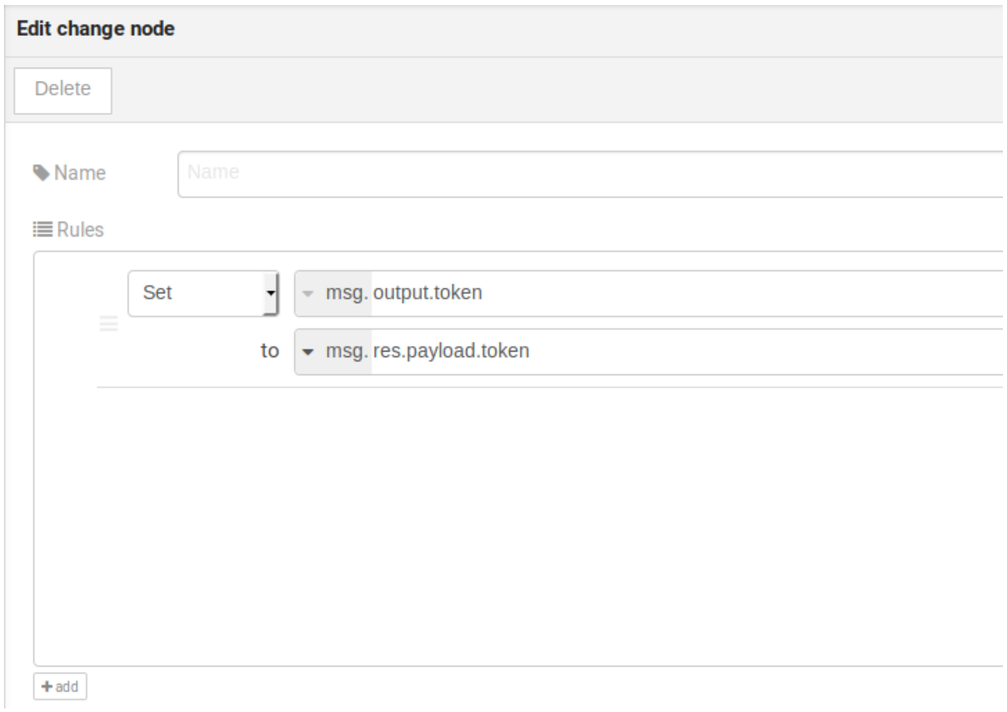
Response: msg. res

Return: a parsed JSON object

Name: Name

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

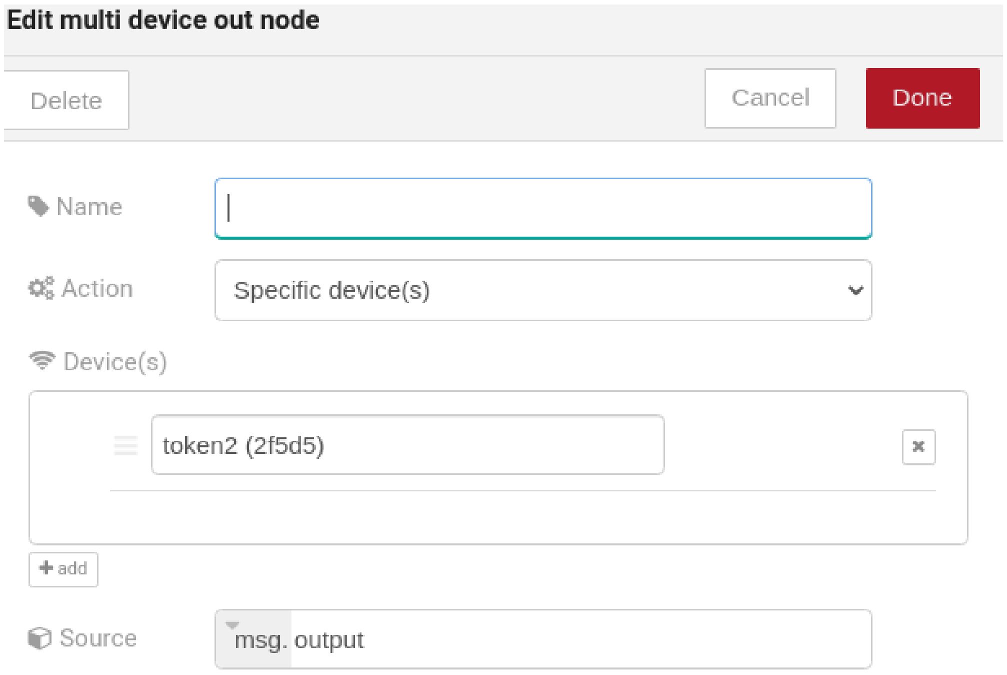
Fig. 11.22: : Template node configuration



Name: Name

Set msg. output.token
to msg. res.payload.token

Fig. 11.23: : Template node configuration



Name:

Action: Specific device(s)

Device(s): token2 (2f5d5)

Source: msg. output

Fig. 11.24: : Device out configuration

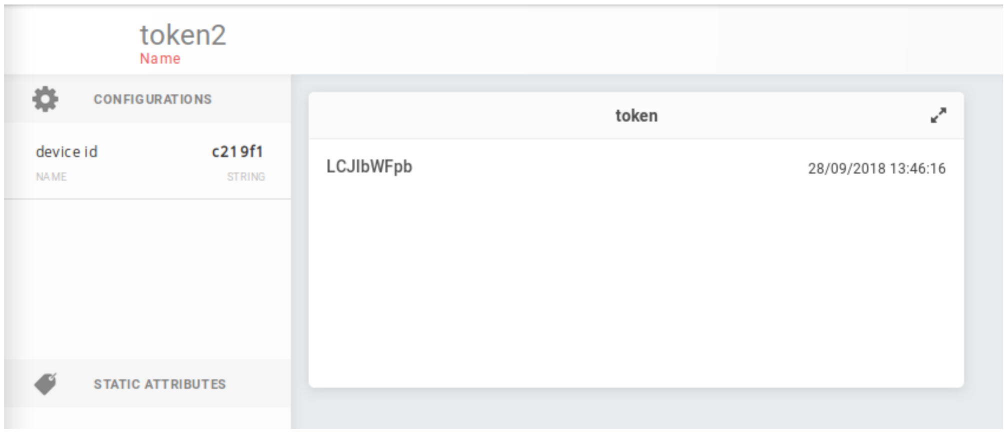

token2
Name
CONFIGURATIONS
device id c219f1
NAME STRING
STATIC ATTRIBUTES
token
LCJIbWFpb 28/09/2018 13:46:16

Fig. 11.25: : Device updated

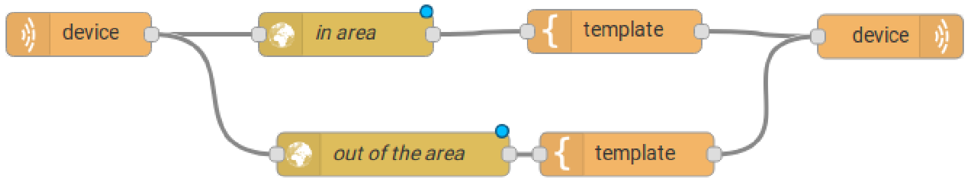


Fig. 11.26: : Flow using geofence

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"

11.2.3 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.

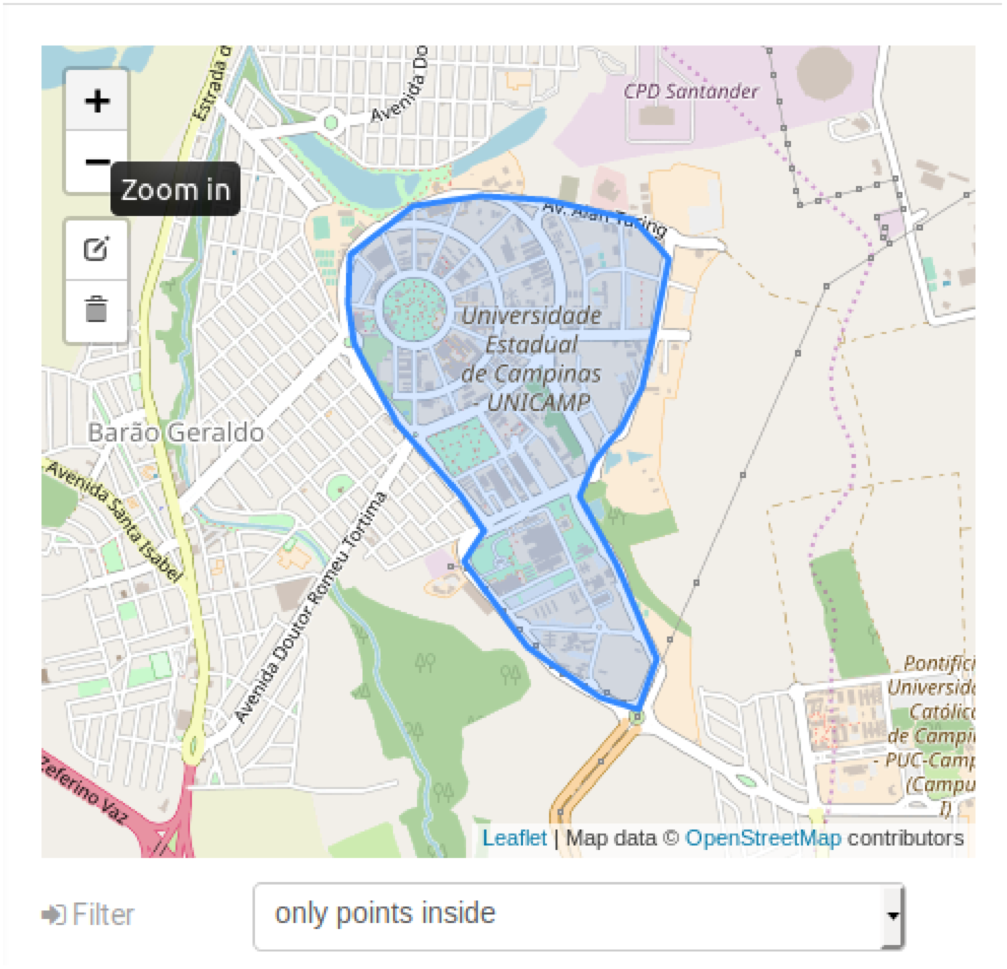
In the *cumulative sum* node, we will accumulate the value of rain (*Target attribute*) in the 60 minute time window (*Time period*) and we will set this sum in a new attribute called *payload.data. attrs.rain60Min* (*Sum*). The *Timestamp* setting refers to the timestamp from the device or dojot, most of the time it can be set with *payload.metadata.timestamp*. See more in Fig. 11.31 .

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. 11.32.

Now, if our value is greater than 100 we need to generate the notification, for that we will use an auxiliary node before, the *template* node. In the template node we will create the message that will appear in the notification and define its metadata, Fig. 11.33.

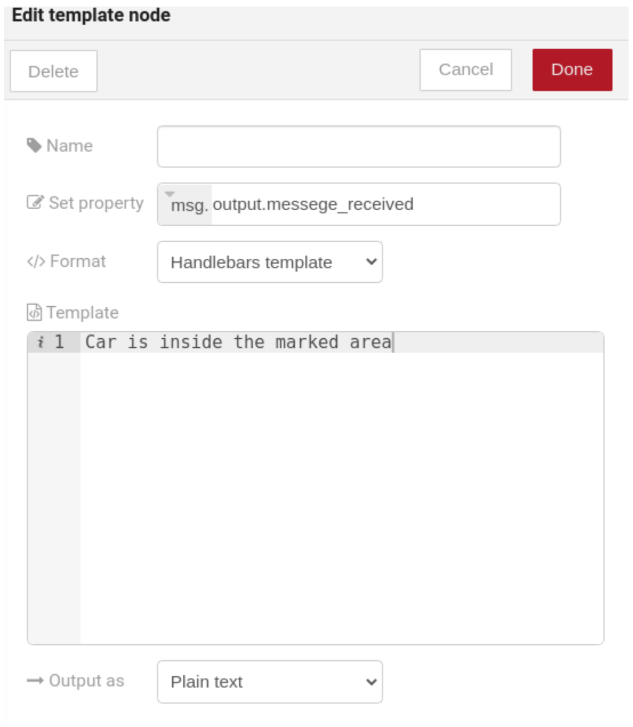
Finally, we will configure the notification node, as in image Fig. 11.34.

So, if the weather station (device set in the event device node with publication checked) sends several messages like


Zoom in
Universidade Estadual de Campinas - UNICAMP
Barão Geraldo
CPD Santander
Av. Alan Turing
Avenida Santa Isabel
Avenida Doutor Romeu Tortima
Leaflet | Map data © OpenStreetMap contributors

Filter: only points inside

Fig. 11.27: : Geofence node configuration

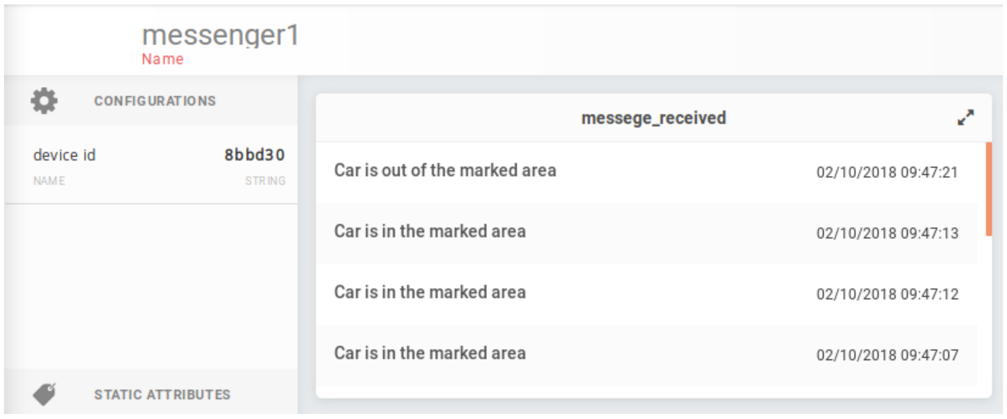

Edit template node

Name:
Set property: msg. output.messege_received
Format: Handlebars template
Template:

```
Car is inside the marked area
```

Output as: Plain text

Fig. 11.28: : Template node configuration if the car is in the marked area


messenger1
Name

device id 8bbd30
NAME STRING

messege_received

Car is out of the marked area	02/10/2018 09:47:21
Car is in the marked area	02/10/2018 09:47:13
Car is in the marked area	02/10/2018 09:47:12
Car is in the marked area	02/10/2018 09:47:07

Fig. 11.29: : Output in device out

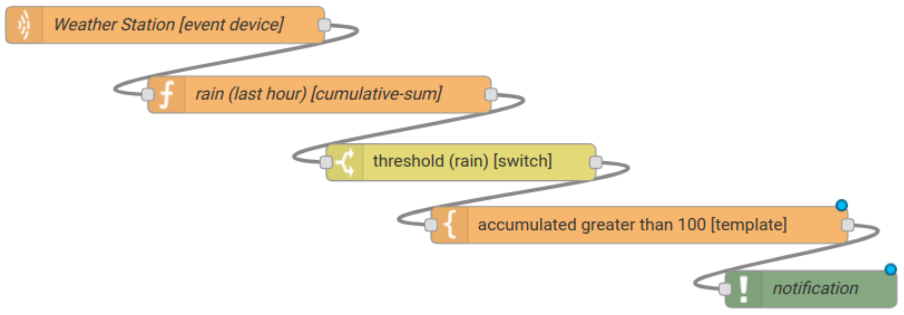
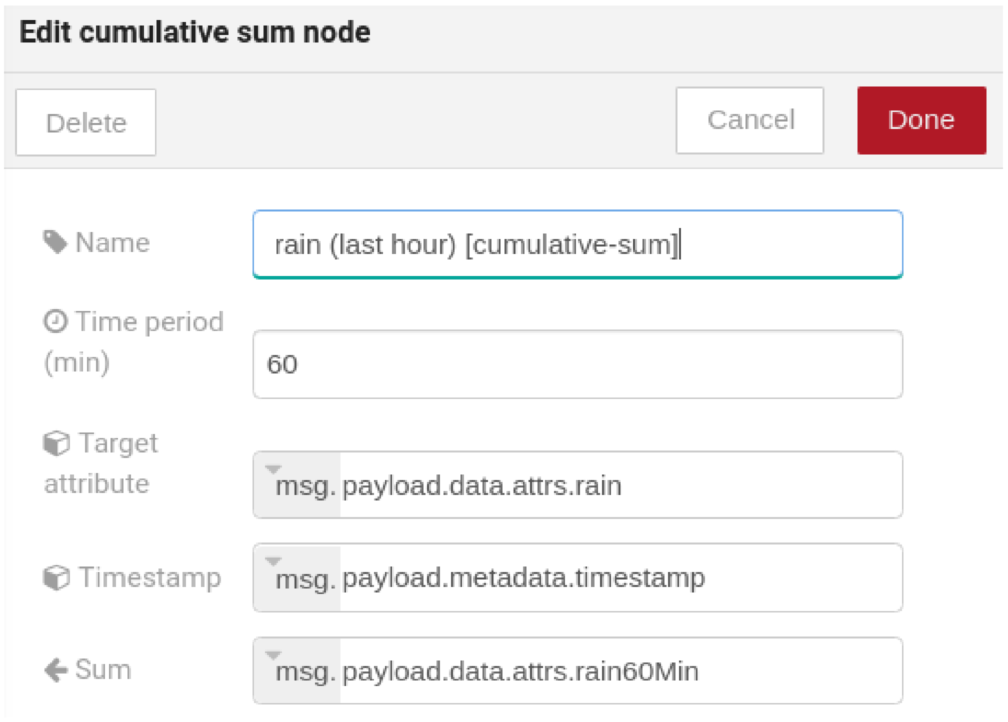

Weather Station [event device]
rain (last hour) [cumulative-sum]
threshold (rain) [switch]
accumulated greater than 100 [template]
notification

Fig. 11.30: : Flow using cumulative sum, switch and notification



Name: rain (last hour) [cumulative-sum]

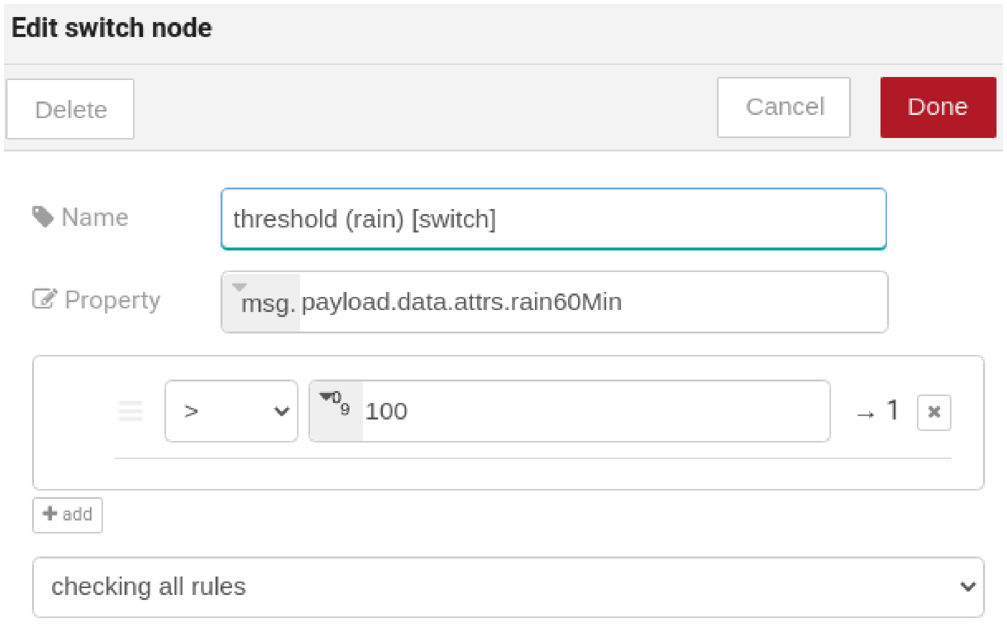
Time period (min): 60

Target attribute: msg. payload.data.attrs.rain

Timestamp: msg. payload.metadata.timestamp

Sum: msg. payload.data.attrs.rain60Min

Fig. 11.31: : Cumulative sum node configuration



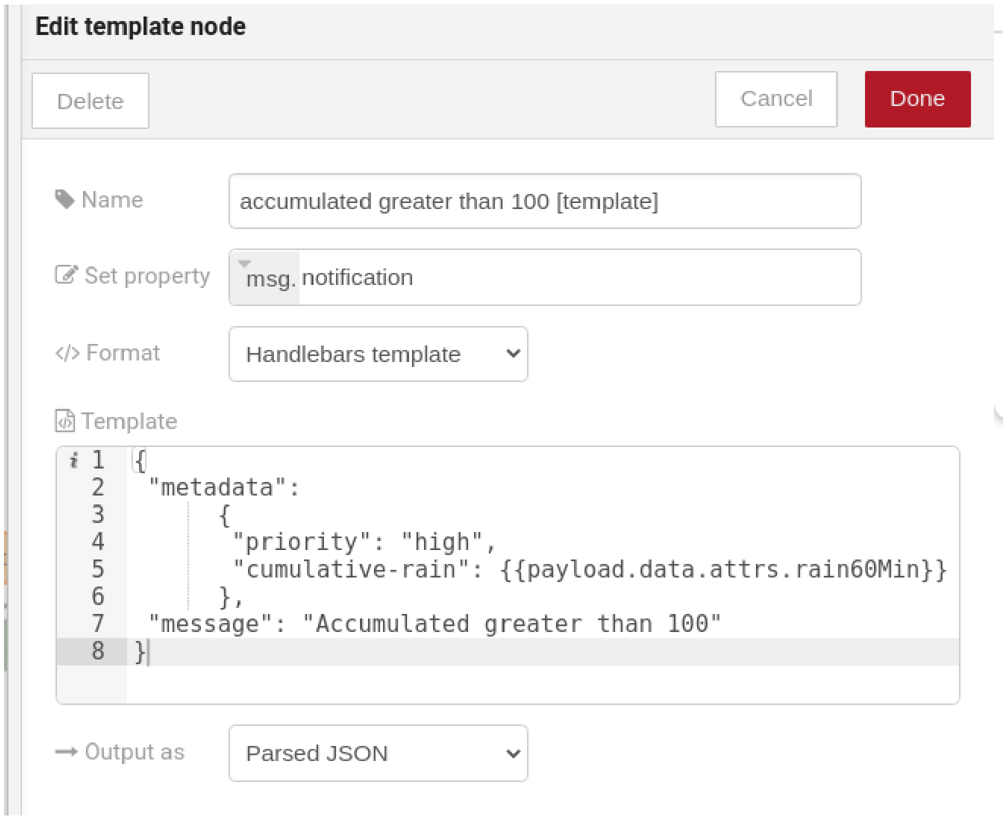
Name: threshold (rain) [switch]

Property: msg. payload.data.attrs.rain60Min

> 100 → 1

checking all rules

Fig. 11.32: : Switch node configuration



Name: accumulated greater than 100 [template]

Set property: msg. notification

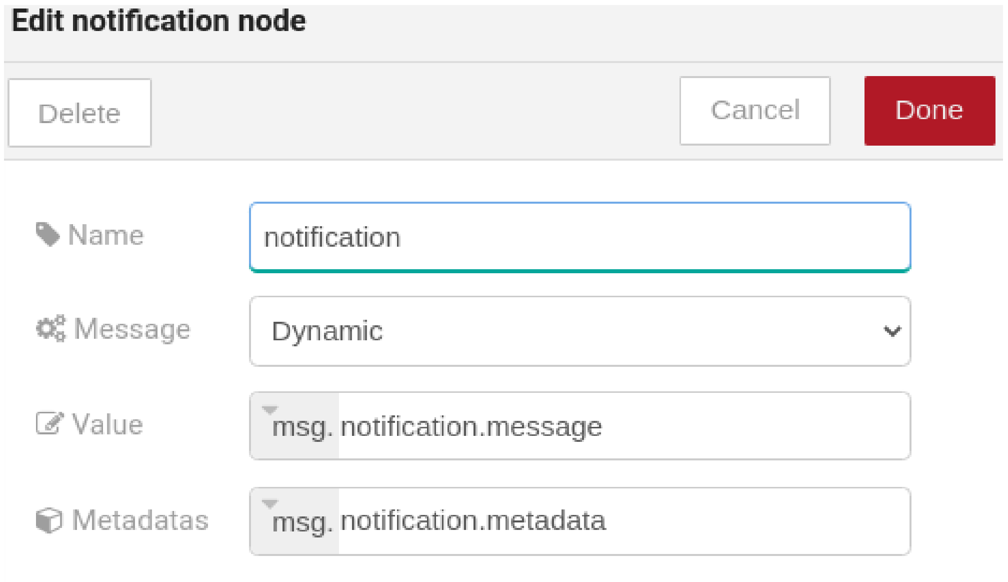
Format: Handlebars template

Template

```
{
 "metadata":
      {
       "priority": "high",
       "cumulative-rain": {{payload.data.attrs.rain60Min}}
      },
 "message": "Accumulated greater than 100"
}
```

Output as: Parsed JSON

Fig. 11.33: : Template node configuration


Edit notification node

Name: notification

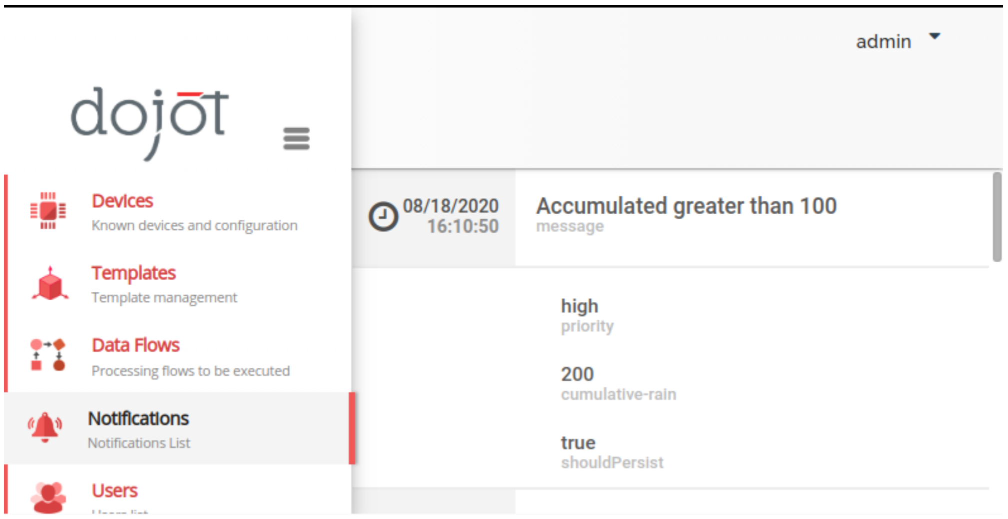
Message: Dynamic

Value: msg. notification.message

Metadatas: msg. notification.metadata

Fig. 11.34: : Notification node configuration

{" 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. 11.35.



08/18/2020 16:10:50

Accumulated greater than 100
message

high
priority

200
cumulative-rain

true
shouldPersist

Fig. 11.35: : Notification

CHAPTER 12

Using MQTT with security (TLS)

Note:

- Audience: administrators
- Level: intermediate
- Reading time: 15 m

This document describes how to configure dojot to use MQTT over TLS when using the microservice IotAgent-Mosca

Table of Contents

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

- A key pair (.key file);
- A certificate signed by a Certificate Authority (CA) trusted by Mosca (.crt file);

- The certificate of this CA (.crt file);

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

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

12.1 Components

12.1.1 EJBCA-REST

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

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

What is a certificate?

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

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

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

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

So, how does EJBCA work in dojot?

When creating a new device, an associated end entity is created in EJBCA. Its name will be the device's ID (like 'f60c28') and its password will be always 'dojot'.

A certificate can be signed by sending a HTTP POST request to host:8000/sign/<cname>/pkcs10. CName is the end entity's name (or device). The payload sent with this request should be a JSON containing the end entity password and a CSR file (certificate intention) in base64 format.

Note that the URL is routed by the API gateway. As in other APIs in dojot, a JWT is needed. You can find how to generate and how to use such token in *Using API interface*.

In order to create the CSR file and ask for a certificate signature, a user can use a helper script called 'Certificate Retriever', which is detailed in *Certificate retriever* section.

12.1.2 Mosca

Mosca is a node.js mqtt broker. To using Mosca you need do some configurations by environment variable:

- MOSCA_TLS_DNS_LIST: TLS DNS list, Servers hostnames, the host to connect external (separated by a comma). Example: localhost, mydomain.com

All the certificates will be created automatically, not needing to configure manually the certificates into the broker.

Note: To use Mosca without TLS too, you need set the environment variable ALLOW_UNSECURED_MODE to 'true' and to use 1883 port. **It is not recommended!**

12.2 Certificate retriever

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

A user can use it by executing:

```
sudo apt install python3-pip #run on Debian-based Linux distributions, if necessary

pip3 install crypto #or pip install crypto, run if necessary
pip3 install pyOpenSSL #or pip install pyOpenSSL, run if necessary
pip3 install requests #or pip install requests, run if necessary

mkdir -p certs
```

And to finally get the certificate for the device:

```
python3 generateLoginPwd.py  ${DOJOT_HOST} ${DEVICE_ID} IOTmidCA #run every time
```

The mandatory parameters are:

- ${DOJOT_HOST}: where dojot is (No / at the end). Example: http://localhost:8000
- ${DEVICE_ID}: device id that will get a new certificate. Example: f60c28

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

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

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

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

12.3 Simulating a device with mosquitto

To publish and subscribe using the appropriated certificates, you must need to be with the Mosca Broker and the EJBCA running. After creating the dojot environment, the templates and the devices, use the mosquitto emulate a device and to publish and subscribe in the desired topics:

Before install mosquitto_pub and mosquitto_sub (from package *mosquitto-clients* on Debian-based Linux distributions) and access the folder certs, if necessary:

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 MQTT broker is not running before starting dojot (by running commands like `ps aux | grep mosquitto`) to avoid port conflicts.

```
sudo apt-get install mosquitto-clients   #if necessary on Debian-based Linux␣
↪distributions
cd certs  #if necessary
```

How to publish:

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

How to subscribe:

```
mosquitto_sub  -h localhost -p 8883 -t /<tenant>/<deviceId>/config -i <tenant>:
↪<deviceId> --cert <your .crt file> --key <your .key file> --cafile IOTmidCA.crt
```

The <your .crt file>, <your .key file> and the cafile can be created with the Certificate Retriever GitHub repository script. Where <tenant> is a context identifier into dojot and <deviceId> is a identifier for the device in the corresponding context.

Note: In this case, the message is a publication with an attribute, this attribute has the label *attr_example* and a new value 10, you need to change this for your case.

12.4 Important Notes

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

12.4.1 Debugging

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

How to read a certificate

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

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