
dojot Documentation

Release 0.2.0

Matheus Magalhaes

Oct 03, 2018

Contents:

1	Architecture	3
1.1	Components	4
1.2	Infrastructure	7
1.3	Communications	7
2	Concepts	9
2.1	dojot basics	9
3	Components and APIs	13
3.1	Components	13
3.2	Exposed APIs	14
3.3	Kafka messages	14
4	Installation Guide	15
4.1	Hardware requirements	15
4.2	Docker compose	16
4.3	Kubernetes	17
5	Frequently Asked Questions	21
5.1	General	22
5.2	Usage	23
5.3	Devices	24
5.4	Data Flows	26
5.5	Applications	28
6	Release history	31
6.1	battojutsu - 2018.10.03	31
7	Using web interface	33
7.1	Device management	33
7.2	Flow configuration	36
8	Using API interface	37
8.1	Getting access token	37
8.2	Device creation	38
8.3	Sending messages	40
8.4	Checking historical data	40

9	Using flow builder	43
9.1	Dojot nodes	43
9.2	Learn by examples	52

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.

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

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

Table of Contents

- *Components*
 - *Kafka + data-broker + NGSI*
 - *DeviceManager*
 - *IoT Agent*
 - *User Authorization Service*
 - *flowbroker*
 - *History*
 - *Logging and Auditing Service*
 - *Kong API Gateway*
 - *GUI*
 - *Elastic Service Controller*
 - *Alarm Management*
 - *Image manager*
- *Infrastructure*
- *Communications*

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. This architecture is described on [Fig. 1.1](#).

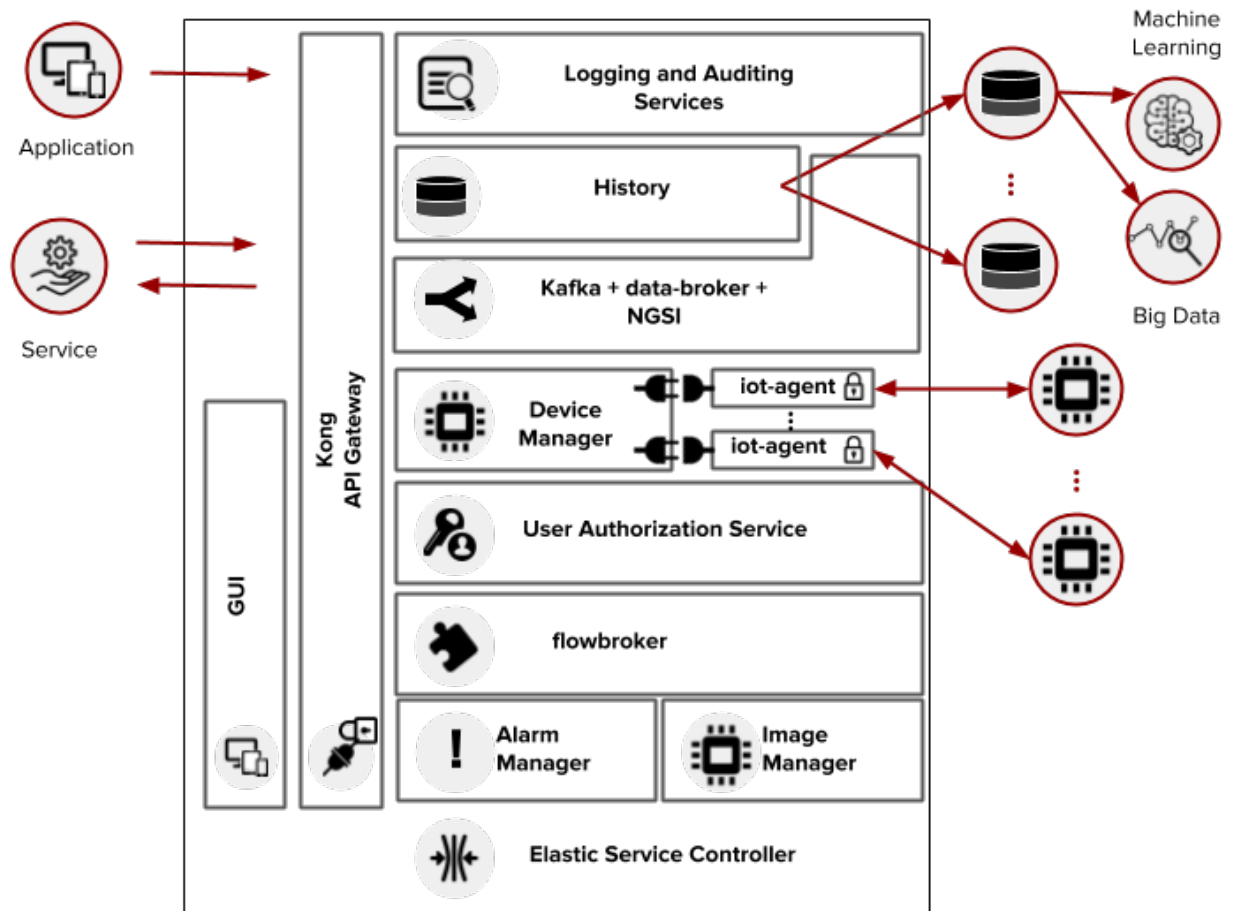


Fig. 1.1: Current Architecture

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, the user might want to receive these readings via notifications generated by subscriptions, 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 is no data sharing, the access credentials are validated by the authorization service for each and every operation (API Request). 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 context broker for internal distribution, reaching, for instance, the history service so it can persist the data on a database. If certain conditions are matched when rules are being processed, a new event is generated and sent to the broker service to be redistributed to

the interested services.

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 + data-broker + NGSI

Apache Kafka is a distributed messaging platform that can be used by applications which need to stream data or consume/produce data pipelines. In 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 flow-broker service makes use of an in-memory database for efficiency. It adds context to Apache Kafka, making it possible that internal or even external services are able to subscribe or query data based on context. Flow-broker is also a distributed service to avoid it being a single point of failure or even a bottleneck for the architecture.

To keep a certain level of compatibility with NGSI-compatible components, it is possible to build an element that offers a NGSI interface for such components.

1.1.2 DeviceManager

DeviceManager is a core entity which is responsible for keeping device and templates data models. It is also responsible for publishing any updates to all interested components (namely IoT agents, history and subscription manager) 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 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

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 History

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

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

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

1.1.7 Logging and Auditing Service

All the services that are part of the *dojot* platform can generate usage metrics of its resources that can be used by a logging and auditing service, which process this registers and summarize then based on users and applications.

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

Such components are currently in development.

1.1.8 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.

1.1.9 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
- **Applications Management:** Creation, Visualization, Edition and Deletion Operations
- **Device Models Management:** Creation, Visualization, Edition and Deletion Operations
- **Devices Management:** Creation, Visualization (real time data), Edition and Deletion Operations
- **Processing Flows Management:** Creation, Visualization, Edition and Deletion Operations

1.1.10 Elastic Service Controller

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

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

This component is currently scheduled for development.

1.1.11 Alarm Management

This component is responsible for handling alarms generated by dojot's internal components, such as IoT agents, Device Manager, and so on.

1.1.12 Image manager

This component is responsible for device image storage and retrieval.

1.2 Infrastructure

A few extra components are used in dojot that were not shown in [Fig. 1.1](#). 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

Concepts

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

Table of Contents

- *dojot basics*
 - *User authentication*
 - *Devices and templates*
 - *Flows*

Note:

- **Audience**
 - Users that want to take a look at how dojot works;
 - Application developers.
 - Level: basic
-

2.1 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 tutorials/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.

2.1.1 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](#).

2.1.2 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.

2.1.3 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](#).

Components and APIs

3.1 Components

Table 3.1: Components

Component	GitHub repository	Documentation
mongodb		mongodb documentation
postgres		postgres documentation
Kong API gateway		Kong documentation
redis		Redis documentation
zookeeper		Zookeeper documentation
Kafka		Kafka documentation
auth	GitHub - auth	readthedocs - auth
History	GitHub - history	
DeviceManager	GitHub - DeviceManager	readthedocs - DeviceManager
GUI	GitHub - GUI	
Flow broker	GitHub - flowbroker	
Data broker	GitHub - data-broker	
iotagent-mosca	GitHub - iotagent-mosca	
EJBCA-REST	GitHub - EJBCA-REST	

3.2 Exposed APIs

Table 3.2: APIs :header-rows: 1

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

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

Table 3.3: Original endpoints

Service	Original endpoint	Endpoint
DeviceManager	host:5000/device	host:8000/device
DeviceManager	host:5000/template	host:8000/template
flowbroker	host:3000/	host:8000/flows
auth	host:5000/	host:8000/auth
auth	host:5000/auth/revoke	host:8000/auth/revoke
auth	host:5000/user	host:8000/auth/user
STH	host:8666/	host:8000/history
Data-Broker	host:1026/	host:8000/metric
GUI	host/	host:8000/gui
ejbca	host:5583/sign	host:8000/sign
ejbca	host:5583/ca	host:8000/ca

3.3 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 3.4: Original endpoints

Component	Message	Subject
DeviceManager	Device CRUD (Messages - DeviceManager)	<code>dojot.device-manager.device</code>
iotagent-mosca	Device data update (Messages - iotagent-mosca)	<code>device-data</code>

This page contains information about how to deploy dojot using Docker compose. Kubernetes and Google Cloud Platform support is on track to be implemented.

Table of Contents

- *Hardware requirements*
- *Docker compose*
 - *Docker engine*
 - *Docker Compose*
 - *Installation*
 - *Usage*
- *Kubernetes*
 - *Kubernetes Cluster*
 - *Persistent Storage*
 - *Kubernetes Client*
 - *Deployment*

4.1 Hardware requirements

In order to properly run dojot, the minimum hardware requirements are:

- 4GB of RAM
- 10GB of free disk space
- Network access

- The following ports should be opened:

- TCP (incoming connections): 1883 (MQTT), 8883 (Secure MQTT if used), 8000 (web interface access)
- TCP (outgoing connections): 25 (if send e-mail node is used in a flow)

4.2 Docker compose

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

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

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

The following sections describe all Docker compose dependencies.

4.2.1 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/installation/>

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

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

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

4.2.2 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/>

4.2.3 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 0.2.0 -b 0.2.0
```

Or if you're brave enough:

```
git checkout master
```

After the repository is cloned, and a release (or branch) has been selected, there are still a few external modules that must be gathered before using the platform. These modules can be retrieved by executing the following command:

```
git submodule update --init --recursive
```

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/>

4.2.4 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 `tutorials/using-api-interface` and `tutorials/using-web-interface` for more information about how to interact with the platform.

4.3 Kubernetes

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

This deployment option as presented in this document is best suited for testing and platform assessment. With appropriate changes, this option can be also be used in production environments.

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

The following sections describe all Kubernetes dependencies.

4.3.1 Kubernetes Cluster

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

If you desire to prepare a Kubernetes cluster from scratch, up to date information and installation procedures can be found at [Kubernetes setup documentation](#).

4.3.2 Persistent Storage

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

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

Information about the currently supported persistent storage for Kubernetes can be found at [persistent-volumes](#) page.

4.3.3 Kubernetes Client

To install the Kubernetes client on your machine before proceeding with this guide, follow the proper instructions as presented on the [Kubernetes documentation](#).

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

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

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

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

4.3.4 Deployment

To deploy dojot to a Kubernetes environment, we provide a script for clusters with Ceph as storage solution.

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

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

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

This repository contains all the scripts and deployment files necessary to properly setup dojot's containers. There is one file that must be changed: `config.yaml`, which contains all the parameters used by these scripts. An example of such file is this:

```

1  ---
2  version: 0.2.0-nightly20180319
3  namespace: dojot
4  storage:
5    type: ceph
6    cephMonitors:
7      - '10.0.0.1:6789'
8      - '10.0.0.2:6789'
9      - '10.0.0.3:6789'
10   cephAdminId: admin
11   cephAdminKey: AQD85Z5a/wnlJBAARNISUDpC6RHc8g/UkUcDLA==
12   cephUserId: admin
13   cephUserKey: AQD85Z5a/wnlJBAARNISUDpC6RHc8g/UkUcDLA==
14   cephPoolName: kube
15  externalAccess:
16    type: publicIP
17    ips:
18      - '10.0.0.1'
19      - '10.0.0.2'
20      - '10.0.0.3'
21    ports:
22      httpPort: 80
23      httpsPort: 443
24      mqttPort: 1883
25      mqttSecurePort: 8883
26  services:
27    zookeeper:
28      clusterSize: 3
29    postgres:
30      clusterSize: 3
31    mongodb:
32      replicas: 2
33    kafka:
34      clusterSize: 3
35    auth:
36      emailHost: 'smtp.gmail.com'
37      emailUser: 'test@test.com'
38    emailPassword: 'password'

```

From line 5 to 14, we have Ceph configuration parameters. The `cephMonitors` attribute specifies how many monitors are going to be used and by which address they can be accessed. For more information about this element, check [ceph monitors documentation](#). `cephAdminId`, `cephAdminKey`, `cephUserId` and `cephUserKey` attributes refers to user information. These values are set/generated in user creation.

In `externalAccess` section we have what addresses and ports should be exposed for external access. In `services` section, we can configure how many replicas we want to each service and a few other parameters to configure that service (for instance, `auth` takes an `emailHost` and `emailUser` parameters).

To configure and start the kubernetes cluster, just install all python requirements and start the `deploy.py` script:

```

pip install -r ./requirements.txt
python ./deploy.py

```

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

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

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

5.1 General

5.1.1 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 quickly prototype IoT solutions.
- Real time event processing with customizable rules.

5.1.2 Where can I get it?

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

5.1.3 Which repository is the main one?

There are two main ones:

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

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

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

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

5.2 Usage

5.2.1 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.

5.2.2 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](#).

5.2.3 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 cutting-edge version (beware the bug)

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

If you need a more stable 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-aikido

$ git checkout 0.2.0-aikido -b 0.2.0
```

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.

5.3 Devices

5.3.1 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*.

5.3.2 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.

5.3.3 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.

5.3.4 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.

5.3.5 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 `/<service-id>/<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 }
```

5.3.6 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.

5.3.7 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. The requirements for such integration is described in the question [How can I send MQTT data to dojot so that it appears on the dashboard?](#).

5.3.8 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?](#).

5.3.9 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](#).

5.3.10 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 connector similar to the existing ones for MQTT, CoAP and HTTP.

5.3.11 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.

5.3.12 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",
    "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.

5.4 Data Flows

5.4.1 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.

5.4.2 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.

5.4.3 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.

5.4.4 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.

5.4.5 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.

5.4.6 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.

5.4.7 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.

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

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

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

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

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

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

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

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

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

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

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

It’s pretty easy, actually, although it needs a few commands in bash. To add a new node, you should send the following request:

```
curl -H "Authorization: Bearer ${JWT}" http://localhost:8000/flows/v1/node
-H "content-type: application/json" -d '{"image": "mmagr/kelvin:latest",
"id":"kelvin"}'
```

This will add a new node called ‘kelvin’ which is implemented by a docker image located at “mmagr/kelvin”. There’s only one caveat: you should pull this image in your target system (where dojot is installed) before adding it to the flow menu.

If you don’t want this node anymore, you could delete it:

```
curl -X DELETE -H "Authorization: Bearer ${JWT}"
"http://localhost:8000/flows/v1/node/kelvin"
```

And that’s it! In the [flowbroker](#) repository, there is an example of how to build a Docker image that could be added to flow node menu.

5.5 Applications

5.5.1 What APIs are available for applications?

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

5.5.2 How can I use them?

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

5.5.3 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, by e-mail or stored in a virtual device (which can be used to generate notifications as previously described).

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

6.1 battojutsu - 2018.10.03

- IoT agents:
 - Support for [sigfox devices](#)
 - Support for [LoRa devices](#) to be used with EveryNet networks.
 - Many improvements for IoT agent MQTT - performance, stability and documentation.
- GUI:
 - Map overlays
 - Pin color configuration on maps
 - Support for more screen resolutions
 - Filters (devices, templates)
 - Improved pagination
- Flows:
 - Support for global contexts: a new service, called ContextManager, was created to deal with contexts within a flow. They can be thought as data chunks that can be stored and retrieved by ContextManager when invoked within a flow node. They are split into four different access levels: tenant, flow, node and node instance. Check [flowbroker node library](#) to check how to use context within nodes or check [flowbroker's get-context node](#) to use it directly from flowbroker GUI you could just open the new flowbroker UI and check it in the node palette)
 - New configuration options for device actuation: send actuation message to the same device that triggered the flow or set which is the targeted device dynamically, set while the flow is being processed.
 - Support for device information caching (improving performance)
- History:

- Support for queries that retrieve all attributes from a particular device (without explicitly selecting which one should be returned). Check [history API](#) for more information.
- DeviceManager:
 - Now DeviceManager is able to generate a random key for devices (PSK)
- New libraries:
 - New library for [dojot modules](#) to accelerate development.
 - New [log library](#) to standardize all service logs.

Using web interface

This tutorial will show how to do basic operations in doJot, such as creating devices, checking its attributes and creating flows.

Note:

- Who is this for: entry-level users
 - Level: basic
 - Reading time: 15m
-

7.1 Device management

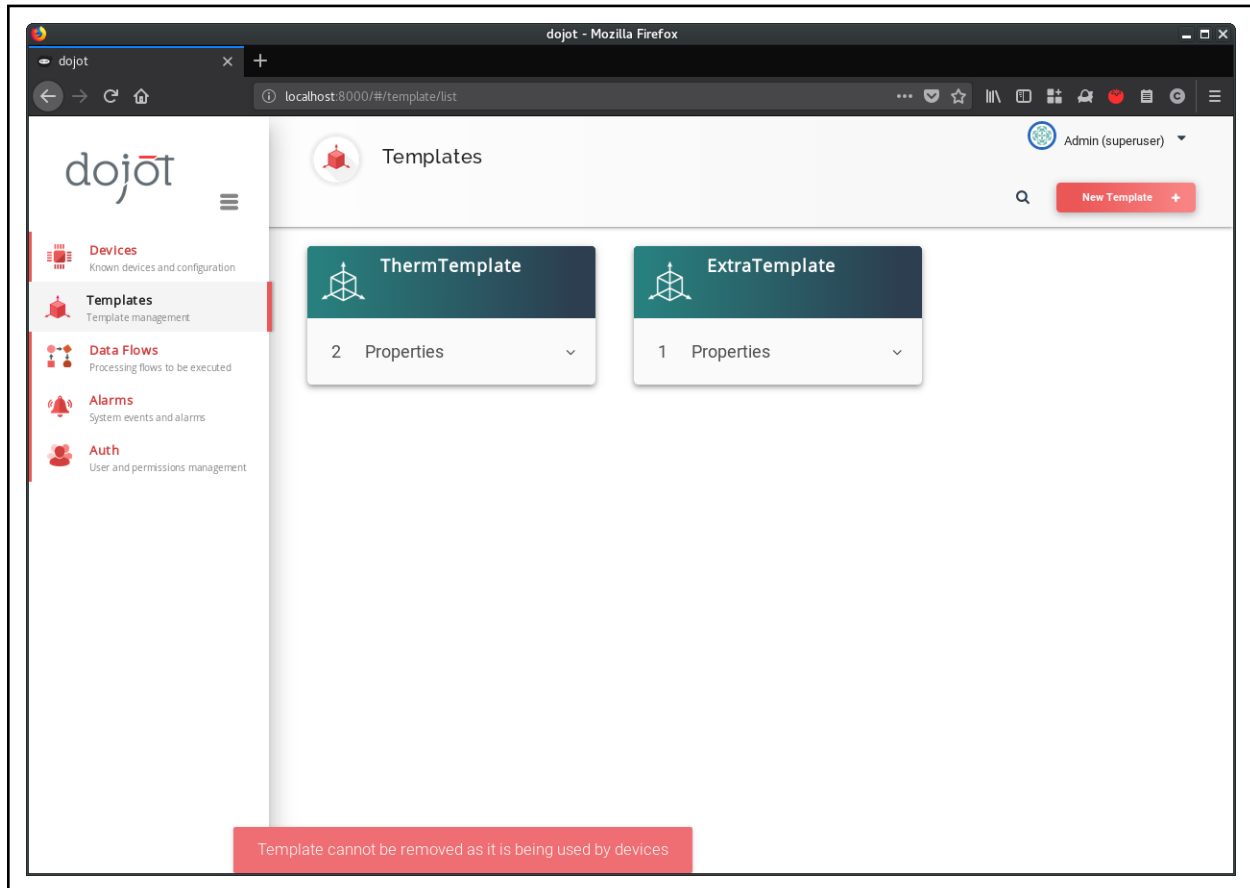
This section will show how to manage device. For this tutorial we will show how to add two thermometers and a virtual device that will represent an alarm system that will monitor both sensors.

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.

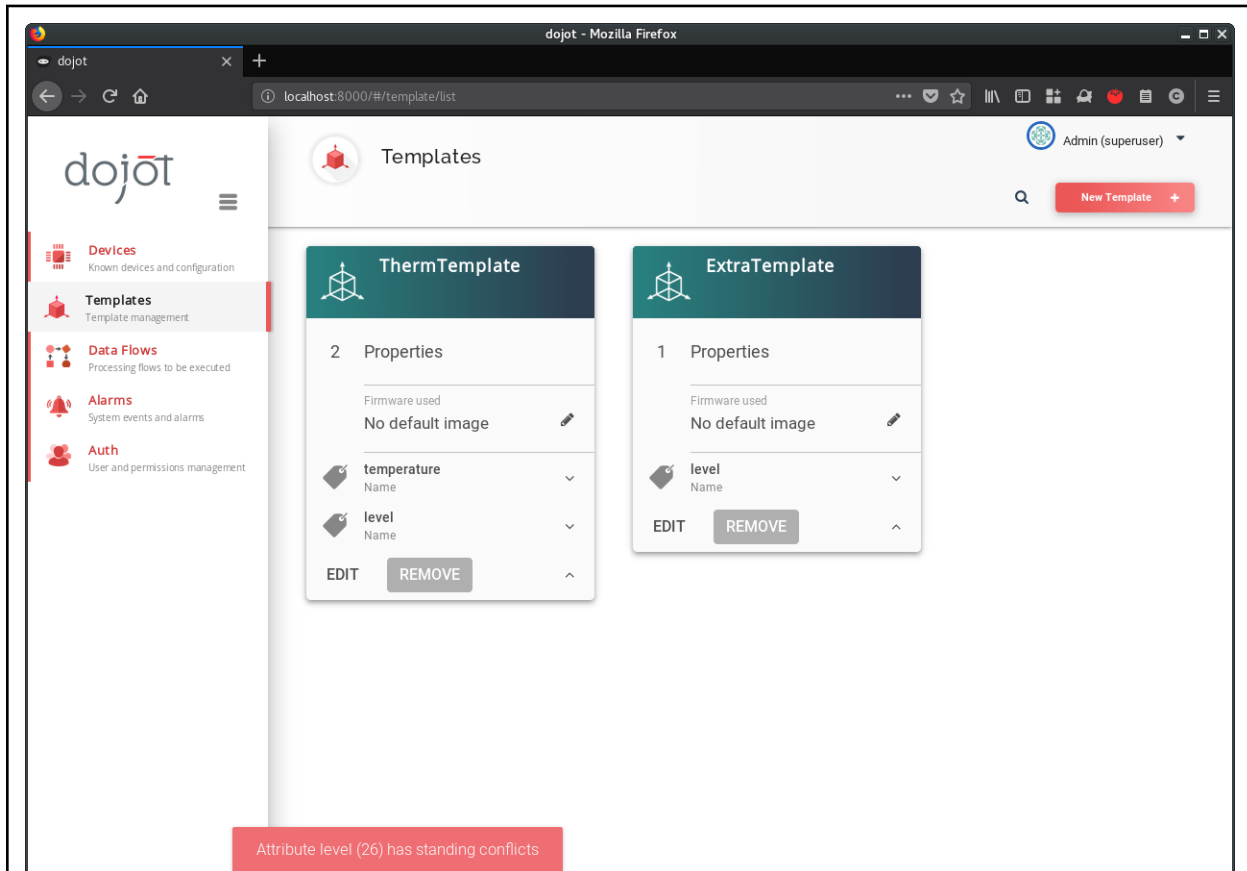
Now we have one template from which devices can be “instantiated”. All devices based on it will accept messages via MQTT that are sent to “/devices/thermometers” topic. To create new devices, you should go back to the devices tab and create a new device, selecting the templates it will be based on, as shown below.

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

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



Attention: You can add and remove attributes from templates and they will be immediately available to devices. In case of new attributes being added, though, you should keep in mind that there must not be any device with templates which have attributes with same name. If such thing happens, the following message will appear:



This snapshot was generated by creating a new template (ExtraTemplate) with one attribute, called `level`. Then a new device based on both templates was created and, afterwards a new attribute also called `level` was added to ThermTemplate.

When this happens, no modification is applied to the template (no attribute named “level” related to the “ThermTemplate” is created). However, it remains in the template 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: as we defined the MQTT topic (all devices will send to `/devices/thermometer` topic), the devices must identify themselves using the `client-id` parameter from MQTT protocol. Another way of doing that is to just use the default topic scheme (which is `/ {SERVICE} / {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`.

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.

7.2 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 below.

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

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

If you are interested on how to use the data generated by these devices in your application, check the Building an application tutorial.

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
-

8.1 Getting access token

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

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

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

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

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

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

8.2 Device creation

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

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

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

This request should give back this message:

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

(continues on next page)

(continued from previous page)

```

28     }
29 }

```

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

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

```

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

```

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

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

Which should give back:

```

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

```

8.3 Sending messages

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

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

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

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

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`).
- If you do define a topic in device template, then your device should publish its data to it and set the `client-id` parameter.
- MQTT payload must be a JSON with each key being an attribute of the dojot device, such as:

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

For more information on how dojot deals with data sent from devices, check the [integrating-physical-devices](#) tutorial. There you can find how to deal with devices that don't publish messages in such format and how to translate them.

8.4 Checking historical data

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

```
mosquitto_pub -t /admin/3bb9/attrs -m '{"temperature": 36.5}'  
mosquitto_pub -t /admin/3bb9/attrs -m '{"temperature": 15.6}'  
mosquitto_pub -t /admin/3bb9/attrs -m '{"temperature": 10.6}'
```

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

```
curl -X GET \
-H 'Authorization: Bearer eyJhbGciOiJIUzI1NiIsIn... ' \
"http://localhost:8000/history/device/3bb9/history?lastN=3&attr=temperature"
```

The history endpoint is built from these values:

- `.../device/3bb9/...`: the device ID is 3bb9 - 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. More operators are available in [history APIs](#).

The request should result in the following message:

```
[
  {
    "device_id": "3bb9",
    "ts": "2018-03-22T13:47:07.050000Z",
    "value": 10.6,
    "attr": "temperature"
  },
  {
    "device_id": "3bb9",
    "ts": "2018-03-22T13:46:42.455000Z",
    "value": 15.6,
    "attr": "temperature"
  },
  {
    "device_id": "3bb9",
    "ts": "2018-03-22T13:46:21.535000Z",
    "value": 36.5,
    "attr": "temperature"
  }
]
```

This message contains all previously sent values.

CHAPTER 9

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: 10 min
-

9.1 Dojot nodes

- *Device in*
- *Device template in*
- *http*
- *Device out*
- *Actuate*
- *Change*
- *Switch*
- *Template*
- *Email*
- *Geofence*
- *Get Context*

9.1.1 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. 9.1](#) will be

displayed.

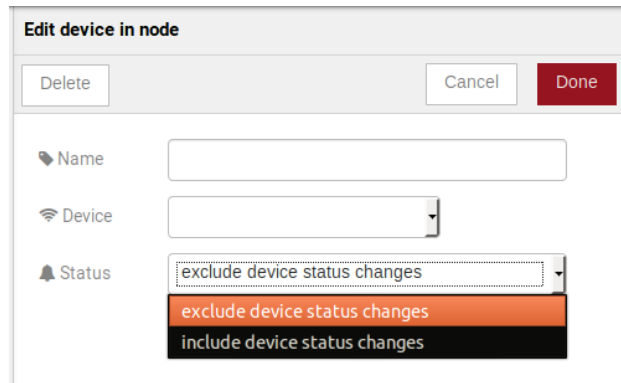
The 'Edit device in node' window has a title bar with the same text. Below the title bar are three buttons: 'Delete' (light grey), 'Cancel' (light grey), and 'Done' (red). The main area contains three fields: 'Name' with a text input, 'Device' with a dropdown menu, and 'Status' with a dropdown menu. The 'Status' dropdown is open, showing three options: 'exclude device status changes' (white), 'exclude device status changes' (orange), and 'include device status changes' (black).

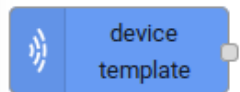
Fig. 9.1: : Device in configuration window

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.

9.1.2 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.

Fields:

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

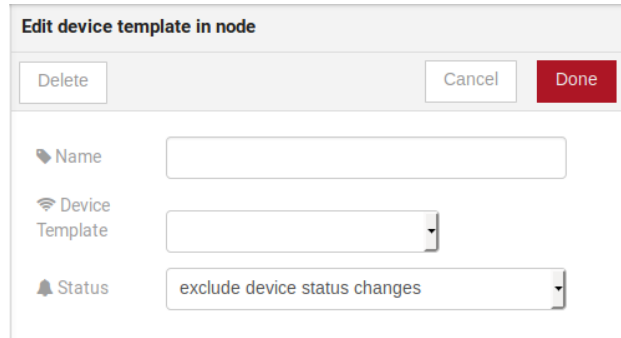
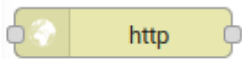


Fig. 9.2: : Device template in configuration window

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

9.1.3 http



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

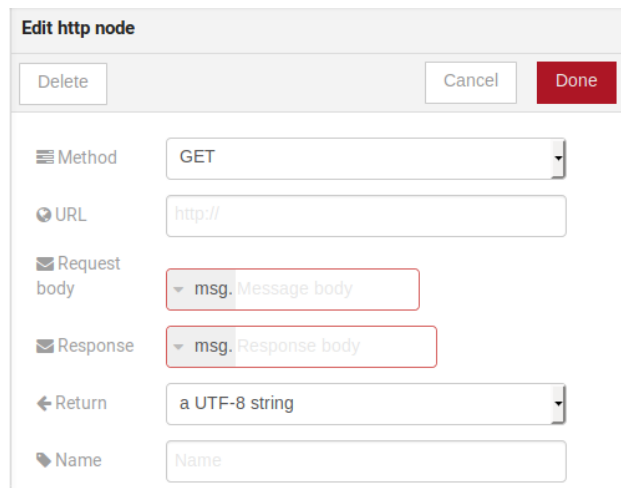


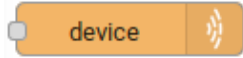
Fig. 9.3: : Device template in configuration window

Fields:

- **Method** (*required*): The http method (GET, POST, etc...).
- **URL** (*required*): The URL that will receive the http request
- **Request body** (*required*): Variable that contains the request body. This value can be assigned to the variable using the **template node**, for example.
- **Response** (*required*): Variable that will receive the http response.
- **Return** (*required*): Type of the return.

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

9.1.4 Device out



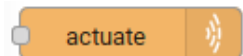
Device out will determine which device will have its attributes updated on *dojot* according to the result of the flow. Bear in mind that this node doesn't send messages to your device, it will only update the attributes on the platform. Normally, the chosen device out is a *virtual device*, which is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

Fig. 9.4: : Device out config window

Fields:

- **Name** (*optional*): Name of the node.
- **Device** (*required*): Select “The device that triggered the flow” will make the device that was the entry-point be the end-point of the flow. “Specific device” any chosen device will 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

9.1.5 Actuate



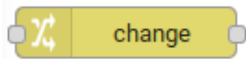
Actuate node is, basically, the same thing of **device out** node. But, it can send messages to a real device, like telling a lamp to turn the light off and etc...

Fig. 9.5: : 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

9.1.6 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

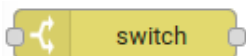
Fig. 9.6: : Change configuration

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.

9.1.7 Switch



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

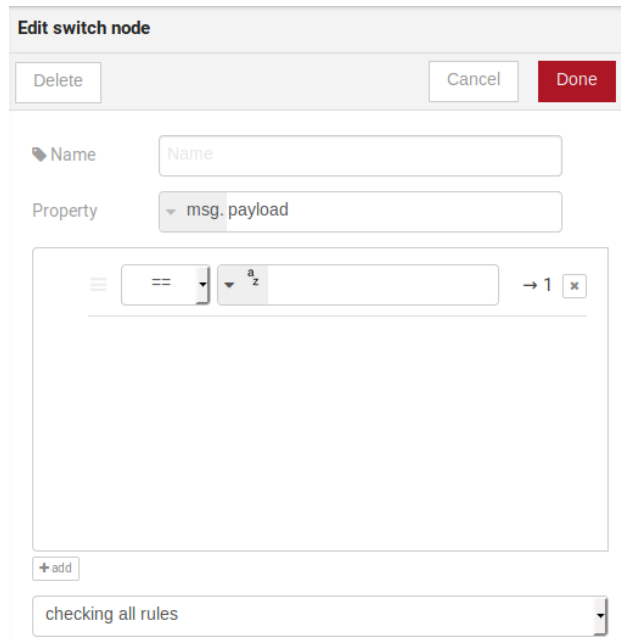


Fig. 9.7: : Switch configuration

Fields:

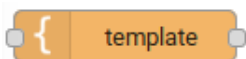
- **Name** (*optional*): Name of the node
- **Property** (*required*): Variable that will be evaluated
- **Rule box** (*required*): Rules that will determine the output branch of the node. Also, it can be configured to stop checking rules when it finds one that matches other or check all the rules and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

Note:

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

9.1.8 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. . .

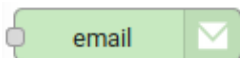
It uses the [mustache](#) template language. Check [Fig. 9.8](#) as example: the field **a** of payload will be replaced with the value of the **payload.b**

Fig. 9.8: : Template configuration

Fields:

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

9.1.9 Email



Sends an e-mail for a given address.

Fields:

- **From** (*required*): The source email.
- **To** (*required*): Destination email.
- **Server** (*required*): The server of the email destination.
- **Subject** (*required*): Subject of the email.

Fig. 9.9: : Email configuration

- **Body** (*required*): Message on the email. The message can be written in a variable using the **template node**, for example.
- **Name** (*optional*): Name of the node.

9.1.10 Geofence

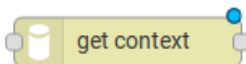


Select an interest area to determine wich devices will activate the flow

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

9.1.11 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

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

Edit geofence node

Delete Cancel Done



Leaflet | Map data © OpenStreetMap contributors

Filter only points inside

Name Geofence name

The figure shows a map of the São Paulo metropolitan area with various cities labeled. A geofence is configured on the map. The configuration options include a filter set to 'only points inside' and a name field labeled 'Geofence name'.

Fig. 9.10: : Geofence configuration

Edit get context node

Delete Cancel Done

Name

Inputs

Context layer Tenant

Context name Context name

Output

Context content Property to store the context content

The figure shows the configuration options for a 'get context' node. It includes a name field, a context layer dropdown set to 'Tenant', a context name field, and a context content field labeled 'Property to store the context content'.

9.2 Learn by examples

- *Using template and email nodes*
- *Using http node*
- *Using geofence node*

9.2.1 Using template and email nodes

To explain these nodes, the flow below will be used:

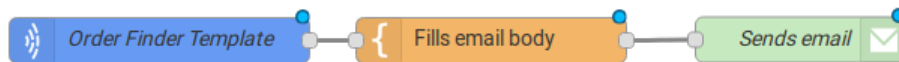


Fig. 9.11: : Flow using template and email nodes

Wonder a system that sends an email to somebody when an order arrive at his mail box. The email would be sent with the name of the sender, his phone number and the content of the order. A device with the order finder template has the attributes: *sender*, *phone* and *content*.

The template node will fill the message with the attributes that came in the message. The attributes sent by the entry-point device can be accessed on the variable **payload**. So, using the *mustache* template language, the node configuration would be like Fig. 9.12.

Then, the email body on the email node should be assigned to the variable that is on the field *Set property* on Fig. 9.12:

Then, the result of the flow, is an email arrive, probably at the spam box, to the destination address:

9.2.2 Using http node

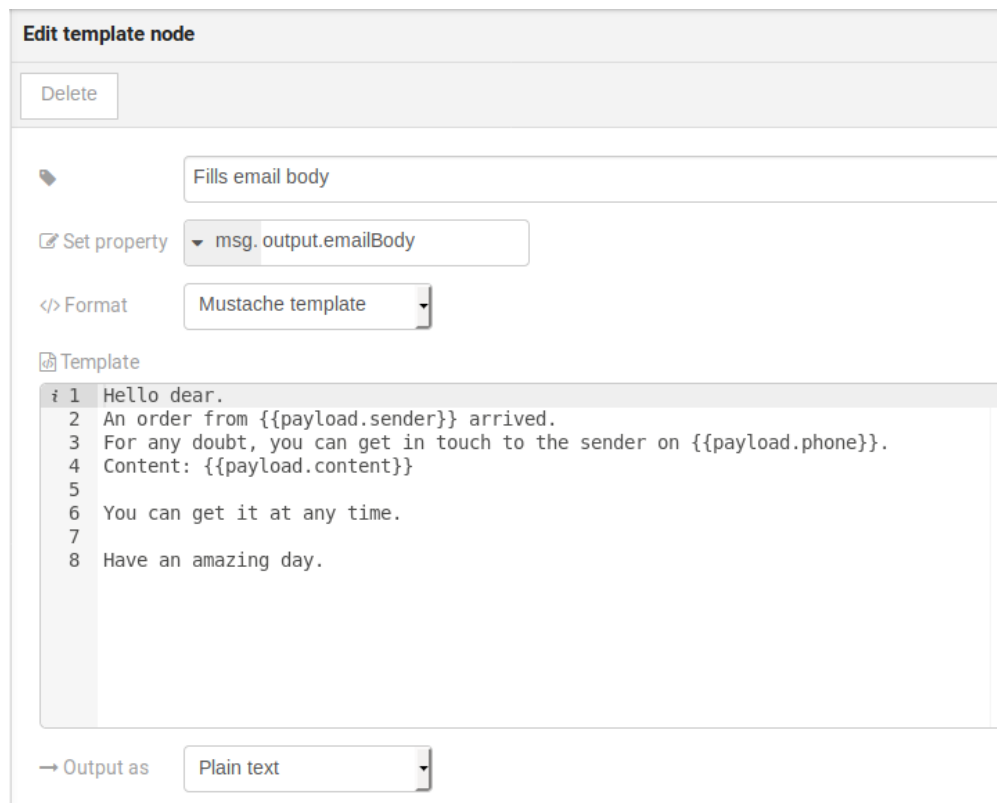
Imagine this scenario: a device sends an *username* and a *password*, and from these attrs, the flow will request to a server an authentication token that will be sent to a virtual device that has a *token* attribute.

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



Edit template node

Delete

Fills email body

Set property msg.output.emailBody

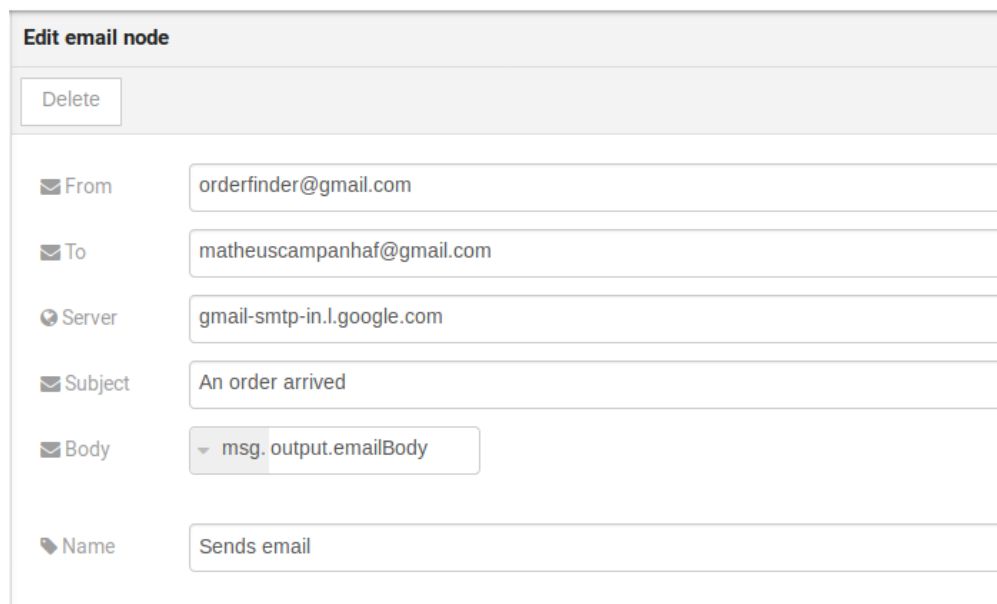
Format Mustache template

Template

```
1 Hello dear.  
2 An order from {{payload.sender}} arrived.  
3 For any doubt, you can get in touch to the sender on {{payload.phone}}.  
4 Content: {{payload.content}}  
5  
6 You can get it at any time.  
7  
8 Have an amazing day.
```

Output as Plain text

Fig. 9.12: : Template configuration



Edit email node

Delete

From orderfinder@gmail.com

To matheuscampanhaf@gmail.com

Server gmail-smtp-in.l.google.com

Subject An order arrived

Body msg.output.emailBody

Name Sends email

Fig. 9.13: : Email node configuration

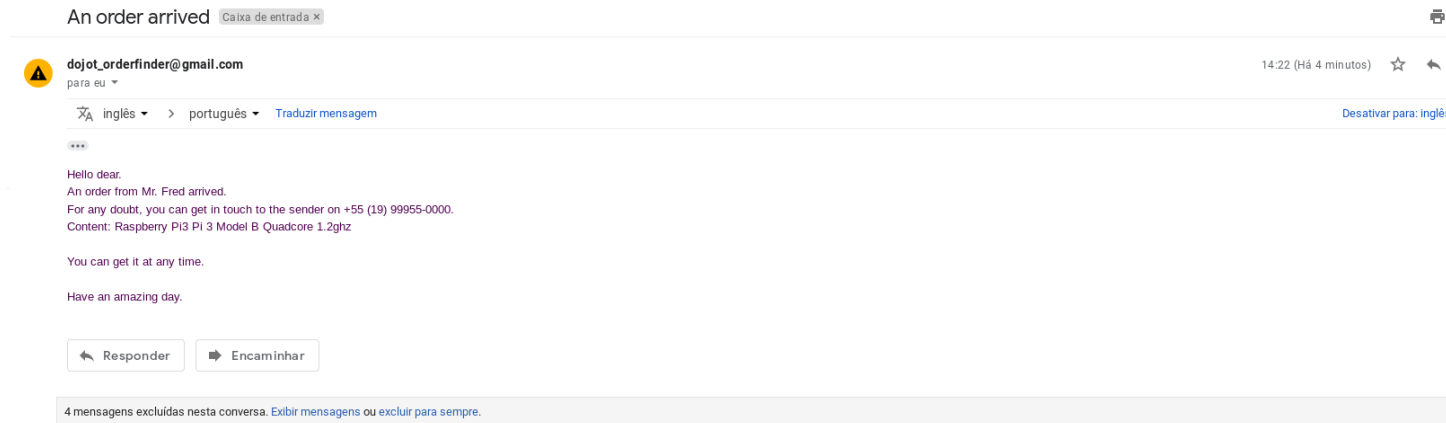


Fig. 9.14: : Sent email

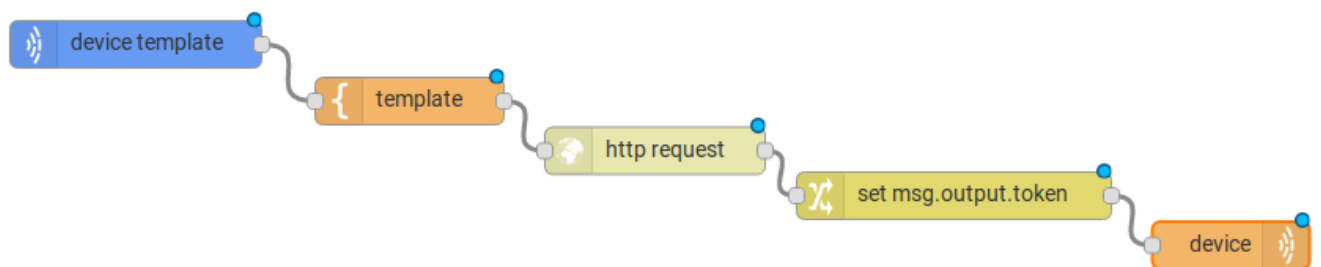
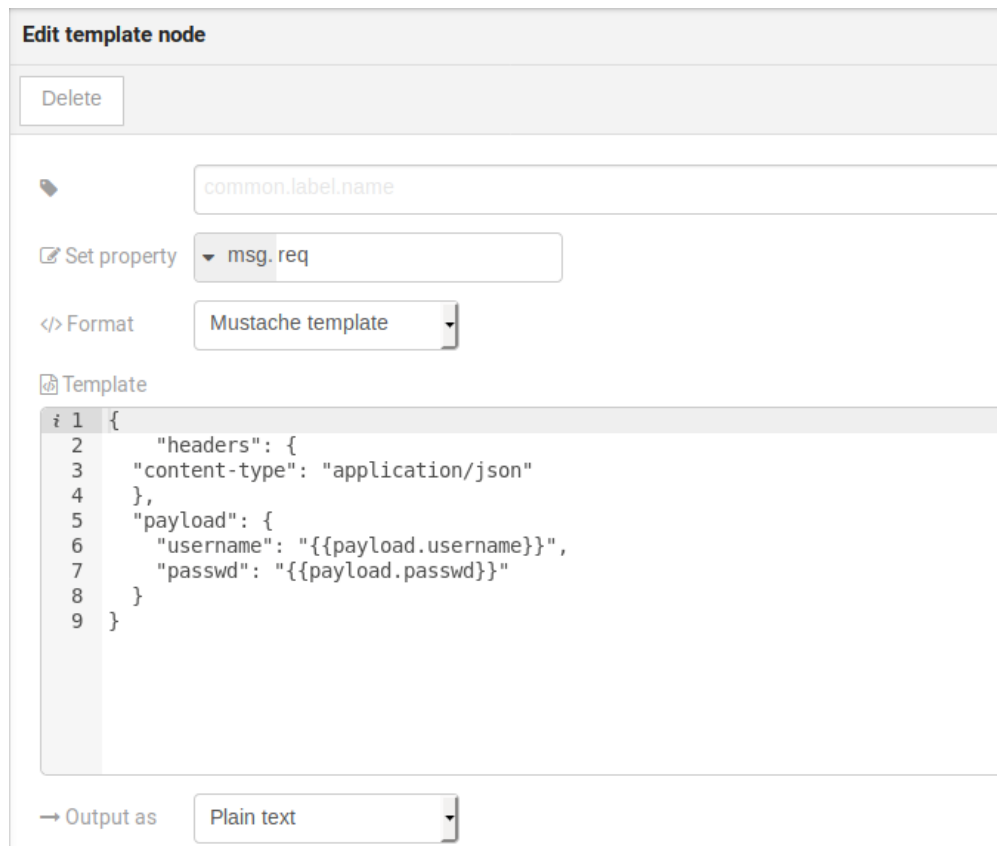


Fig. 9.15: : Flow used to explain http node



Edit template node

Delete

common.label.name

Set property ▼ msg.req

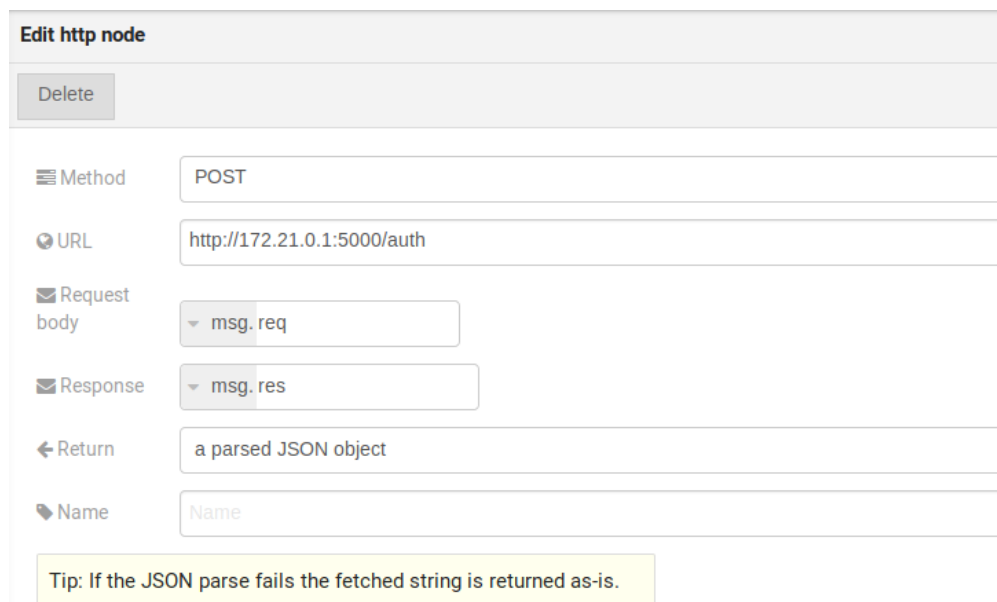
</> Format Mustache template

Template

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

→ Output as Plain text

Fig. 9.16: : Template node configuration



Edit http node

Delete

Method POST

URL http://172.21.0.1:5000/auth

Request body ▼ msg.req

Response ▼ msg.res

Return a parsed JSON object

Name Name

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

Fig. 9.17: : Template node configuration

The screenshot shows the 'Edit change node' configuration window. At the top, there is a 'Delete' button. Below it is a 'Name' field with a placeholder 'Name'. Under the 'Rules' section, there is a configuration for a 'Set' operation. The 'Set' operation is configured to set the value of 'msg.output.token' to 'msg.res.payload.token'. At the bottom left, there is a '+ add' button.

Fig. 9.18: : Template node configuration

The screenshot shows the 'Edit device out node' configuration window. At the top, there is a 'Delete' button. Below it is a 'Name' field. Under the 'Device' section, there is a dropdown menu with the selected value 'A specific device'. Below the device selection, there is a text field containing 'token2 (c219f1)'. At the bottom, there is a 'Source' section with a dropdown menu showing 'output'.

Fig. 9.19: : Device out configuration

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:

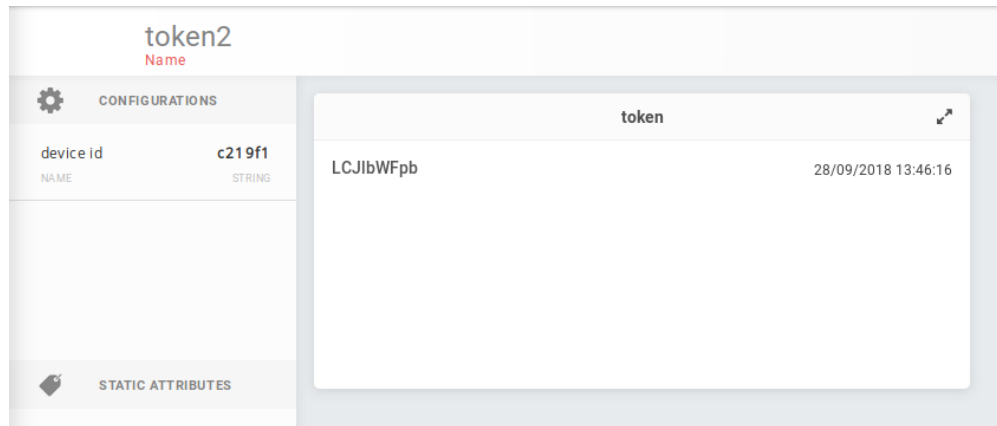


Fig. 9.20: : Device updated

9.2.3 Using geofence node

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

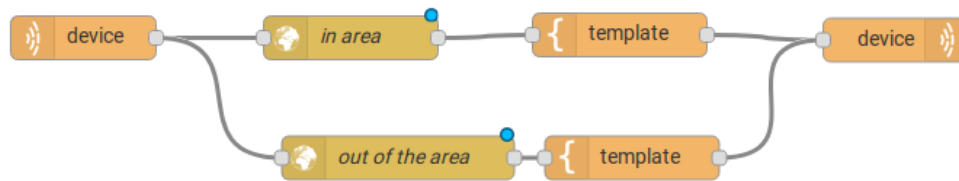


Fig. 9.21: : Flow using geofence

The geofence node named *in area* is set like seen in Fig. 9.22. The only thing that differs 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 message with a geo attribute the geofence node will evaluate the geo point according to its rule and if it matches the rule, the node forwards the information to the next node and, if not, the execution of the branch, which has the geofence that the rule didn't match, stops.

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

Back to the example, if the car sends a message that he is in the marked area, like { "position": "-22.820156, -47.2682535" }, the message received in device out will be "Car is inside the marked area", and, if it sends { "position": "0, 0" } device out will receive "Car is out of the marked area"

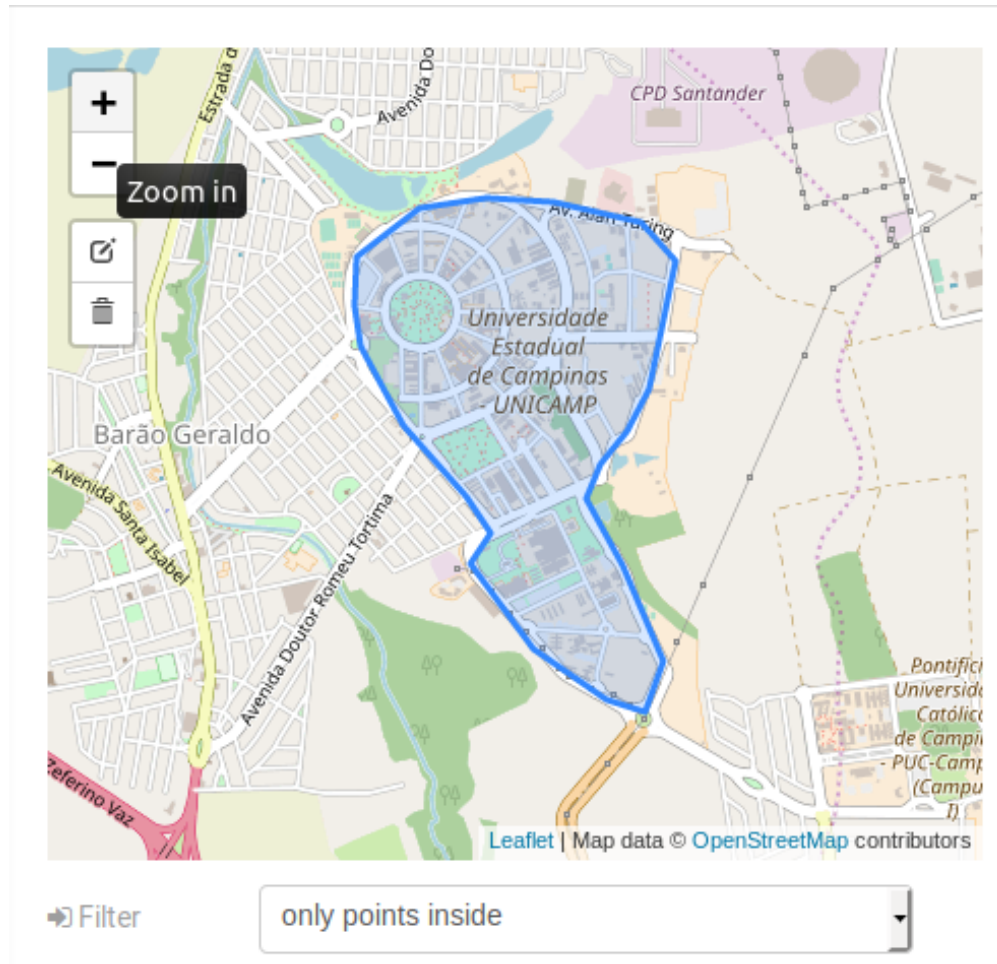


Fig. 9.22: : Geofence node configuration

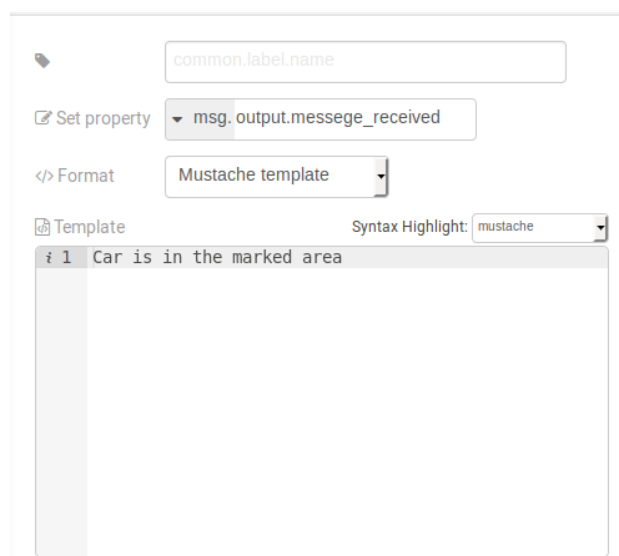


Fig. 9.23: : Template node configuration if the car is in the marked area

The screenshot displays the dojot interface for a device named 'messenger1'. The interface is divided into two main sections: 'CONFIGURATIONS' on the left and 'messege_received' on the right. The 'CONFIGURATIONS' section shows the 'device id' as '8bbd30' with a 'NAME' and 'STRING' type. The 'messege_received' section shows a list of messages received by the device, with a vertical orange bar on the right indicating the scroll position.

messenger1	
Name	
CONFIGURATIONS	
device id	8bbd30
NAME	STRING
STATIC ATTRIBUTES	
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. 9.24: : Output in device out