
dojot Documentation

Release 0.3.0

Matheus Magalhaes

Apr 01, 2019

Contents:

1	Architecture	3
1.1	Components	4
1.2	Infrastructure	7
1.3	Communications	7
2	IoT Agent architecture	9
2.1	TODO	9
2.2	Who should read this?	9
2.3	Introduction	9
2.4	Device security	10
2.5	Information context separation	12
2.6	IoT agent information and management	12
2.7	IoT agent operation	13
2.8	Behavior	17
3	Concepts	19
3.1	dojot basics	19
4	Components and APIs	23
4.1	Components	23
4.2	Exposed APIs	24
4.3	Kafka messages	24
5	Internal communication	25
5.1	Components	25
5.2	Messaging and authentication	26
5.3	Auth + API gateway (Kong)	30
5.4	Device Manager	33
5.5	IoT agent	33
5.6	Persister	35
5.7	History	35
5.8	Data Broker	35
5.9	Flowbroker	37
6	Installation Guide	39
6.1	Hardware requirements	40
6.2	Docker compose	40

6.3	Kubernetes	42
7	Frequently Asked Questions	45
7.1	General	46
7.2	Usage	47
7.3	Devices	48
7.4	Data Flows	50
7.5	Applications	52
8	Release history	55
8.1	battojutsu - 2018.10.03	55
9	Using web interface	57
9.1	Device management	57
9.2	Flow configuration	60
10	Using API interface	61
10.1	Getting access token	61
10.2	Device creation	62
10.3	Sending messages	64
10.4	Checking historical data	64
11	Using flow builder	67
11.1	Dojot nodes	67
11.2	Learn by examples	74

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 + DataBroker*
 - *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](#).

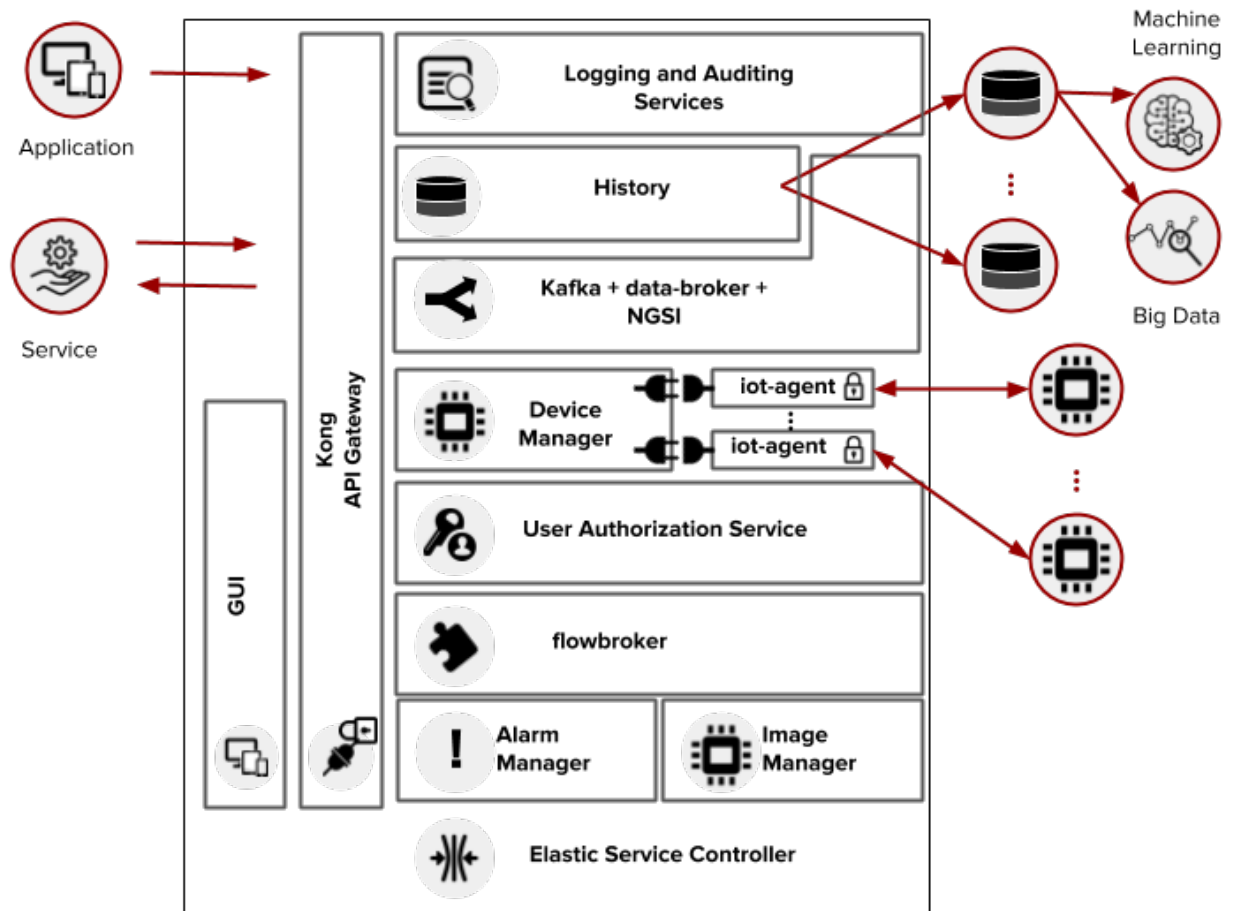


Fig. 1.1: Current Architecture

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, a user can:

- 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 context are isolated and there is 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 history 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 subscribe or query 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 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.

This component is also scheduled for development.

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.

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 TODO

- Sanitize “device” and “physical device” terms. Sometimes they are used interchangeably.
- Build a list of terms (so that I can just say “blingblong” meaning “physical devices or its representative elements”)
- Find a better term for “representative elements”
- Fill the Device identity section
- Select what SHOULD be done, what MAY be implemented or SHALL be presented

2.2 Who should read this?

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

2.3 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.

2.4 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.

2.4.1 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.

2.4.2 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.

2.4.3 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.

2.5 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.

2.6 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. The messages used to publish its device template is detailed in **'Device Template message'** section.
- **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.

2.7 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. This is shown in the figure below.

<< a figure should be here >>

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.

2.7.1 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	<pre>tenants => *tenant tenant => string</pre>

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	<pre>topic => string</pre>

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: <code>dojot.tenancy</code>	
Body format (JSON)	<pre>type="CREATE"/"DELETE" tenant=>string</pre>

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: <code>dojot.device-manager.device-template</code>	
Body format (JSON)	<pre>event => "create" data => id label attrs id => string label => string attrs => [*template_attrs]</pre>

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)	<pre> 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 </pre>
Body format (JSON)	<pre> event => "remove" meta => service service => string data => id id => string </pre>
Body format (JSON)	<pre> event => "actuate" meta => service service => string data => id id => string attrs => *device_attrs </pre>

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)	<pre> metadata => deviceid tenant timestamp_ ↪recv_time deviceid => string tenant => string timestamp => unix_timestamp recv_time => unix_timestamp attrs => *device_attrs </pre>

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
  }
}

```

2.7.2 Firmware update

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

The firmware update process should be:

-

2.8 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.

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
-

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](#).

Components and APIs

4.1 Components

Table 4.1: Components

Component	Repository / Main site	Documentation
MongoDB	MongoDB official site	MongoDB documentation
postgres	PostgreSQL official site	PostgreSQL documentation
Kong API gateway (Community Edition)	Kong official site	Kong documentation
redis	Redis official site	Redis documentation
zookeeper	Zookeeper official site	Zookeeper documentation
Kafka	Kafka official site	Kafka documentation
auth	GitHub - auth	readthedocs - auth
History	GitHub - history	
DeviceManager	GitHub - DeviceManager	readthedocs - DeviceManager
Image manager	GitHub - image-manager	
GUI	GitHub - GUI	
Flow broker	GitHub - flowbroker	
Data broker	GitHub - data-broker	
iotagent-mosca	GitHub - iotagent-mosca	
EJBCA-REST	GitHub - EJBCA-REST	
Alarm manager	GitHub - alarm-manager	

4.2 Exposed APIs

Table 4.2: 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
/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
/image	Device image management	API - image-manager	GitHub - image-manager

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

Table 4.3: Original endpoints

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

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

Internal communication

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

5.1 Components

Current dojoy components are shown in Fig. 5.1.

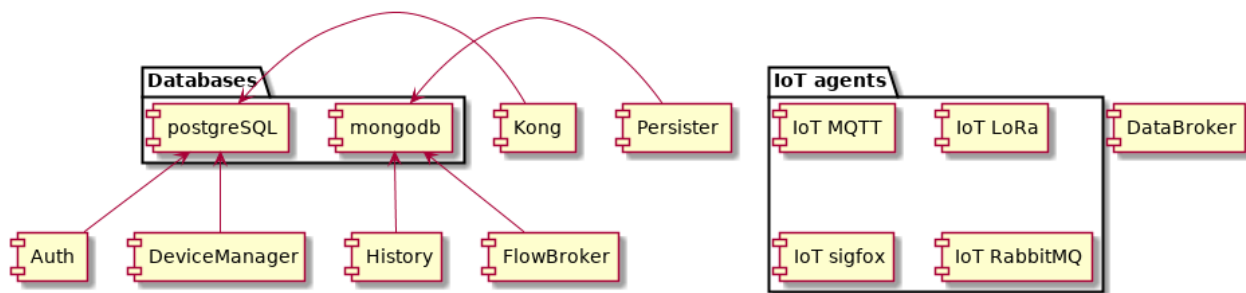


Fig. 5.1: dojoy components

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 endpoint.

This is a test

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](#).

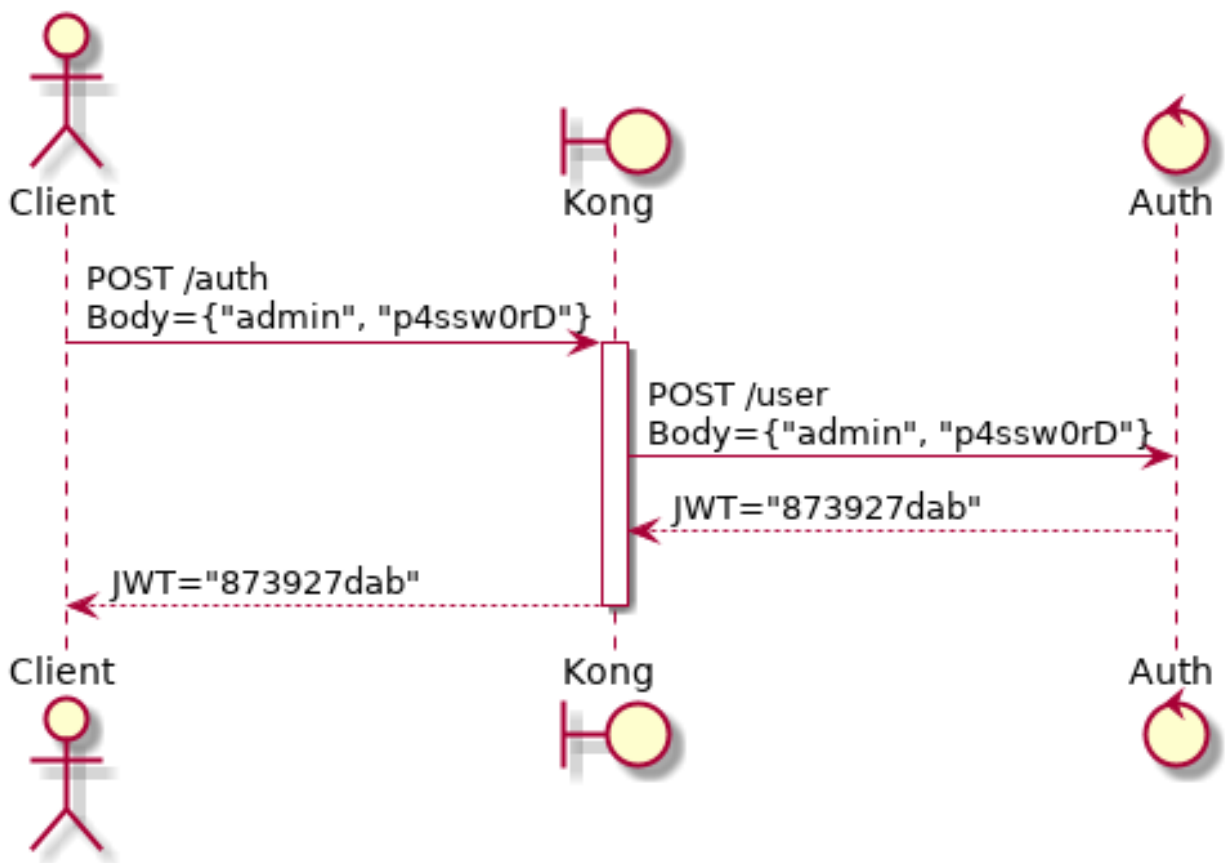


Fig. 5.2: Initial authentication

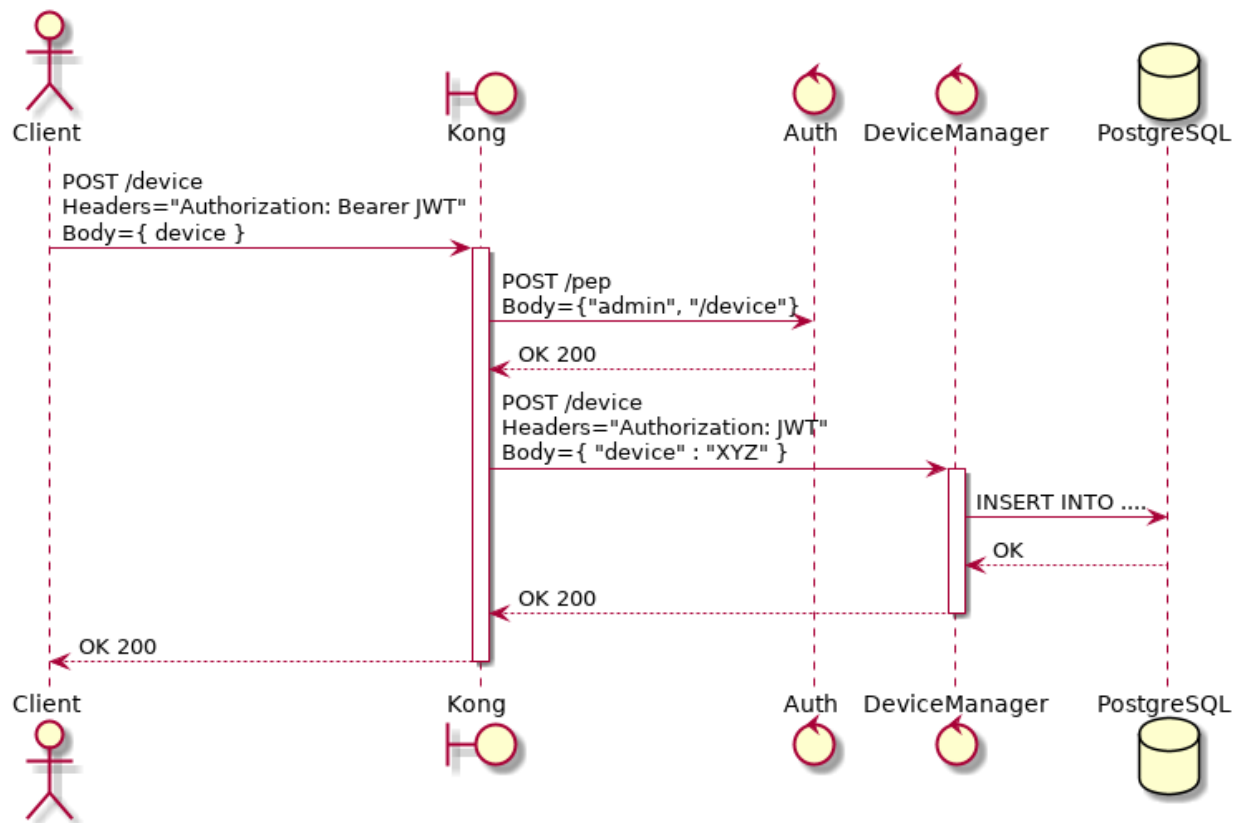


Fig. 5.3: Sending messages to HTTP API

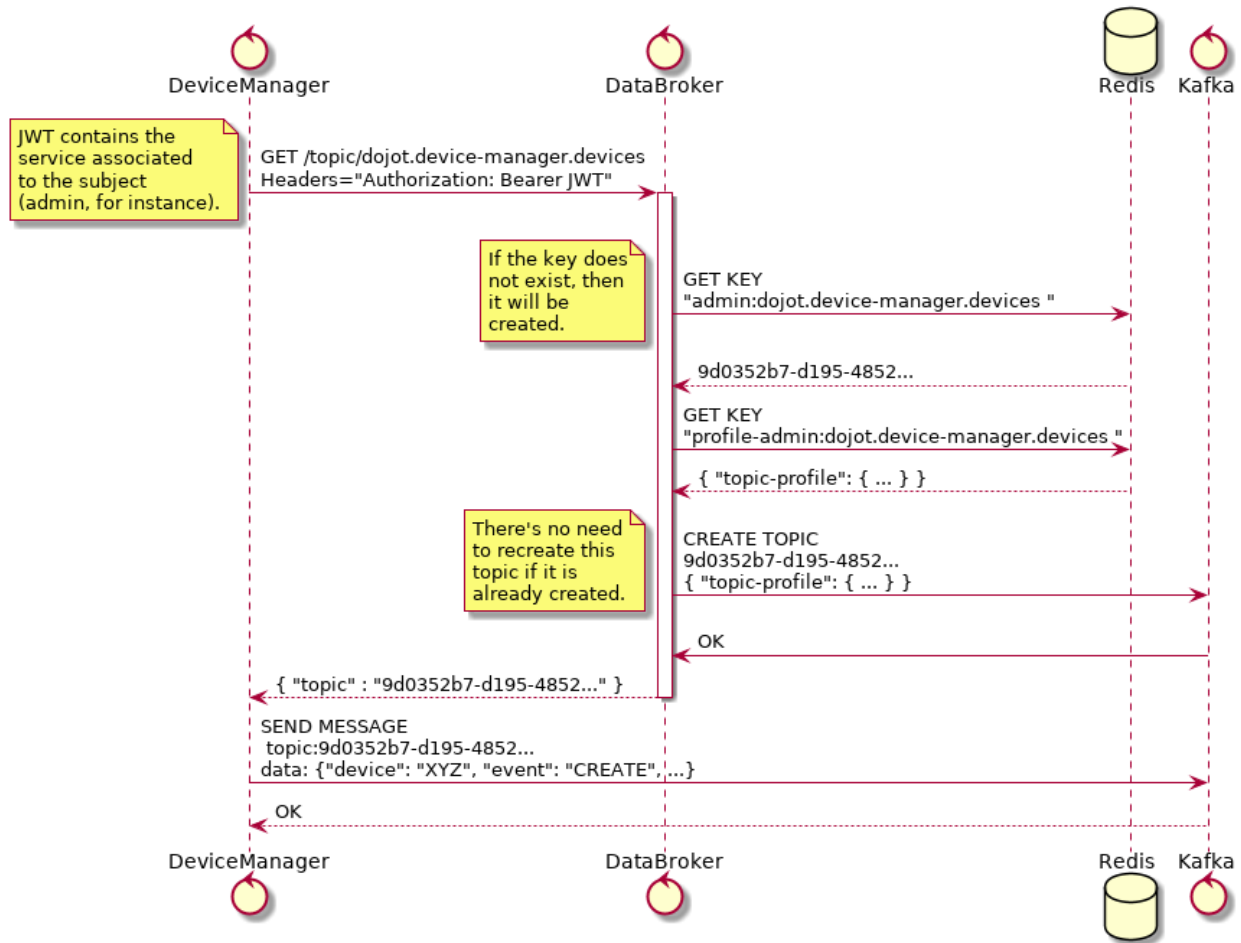


Fig. 5.4: Retrieving Kafka topics

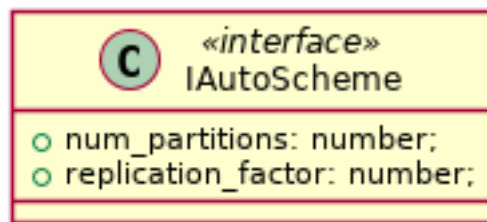


Fig. 5.5: Automatic scheme profile

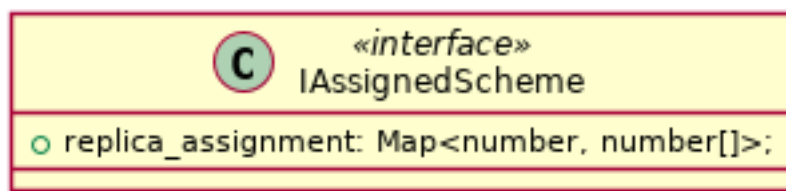


Fig. 5.6: Assigned scheme profile

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:

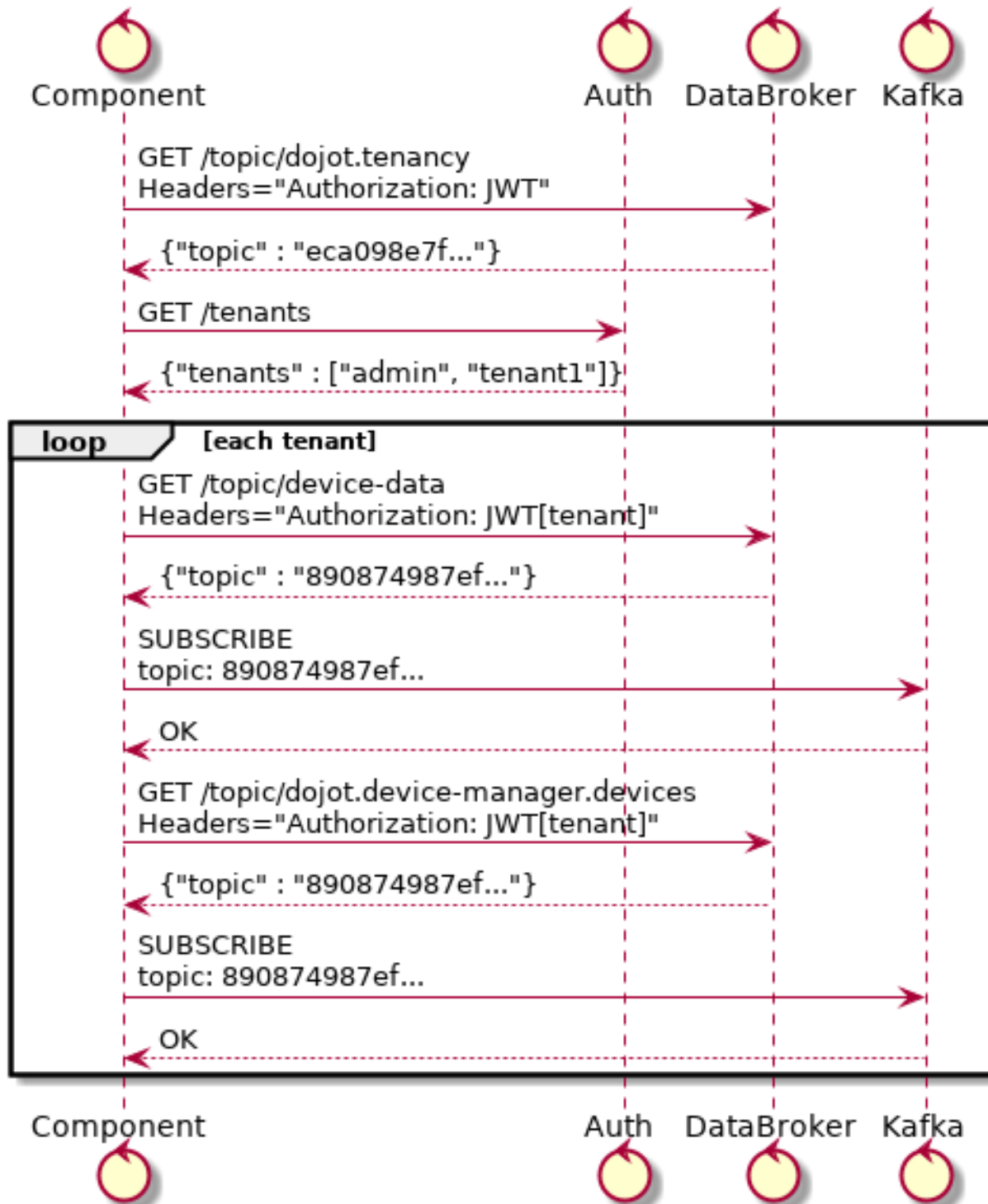


Fig. 5.7: Tenant bootstrapping at startup

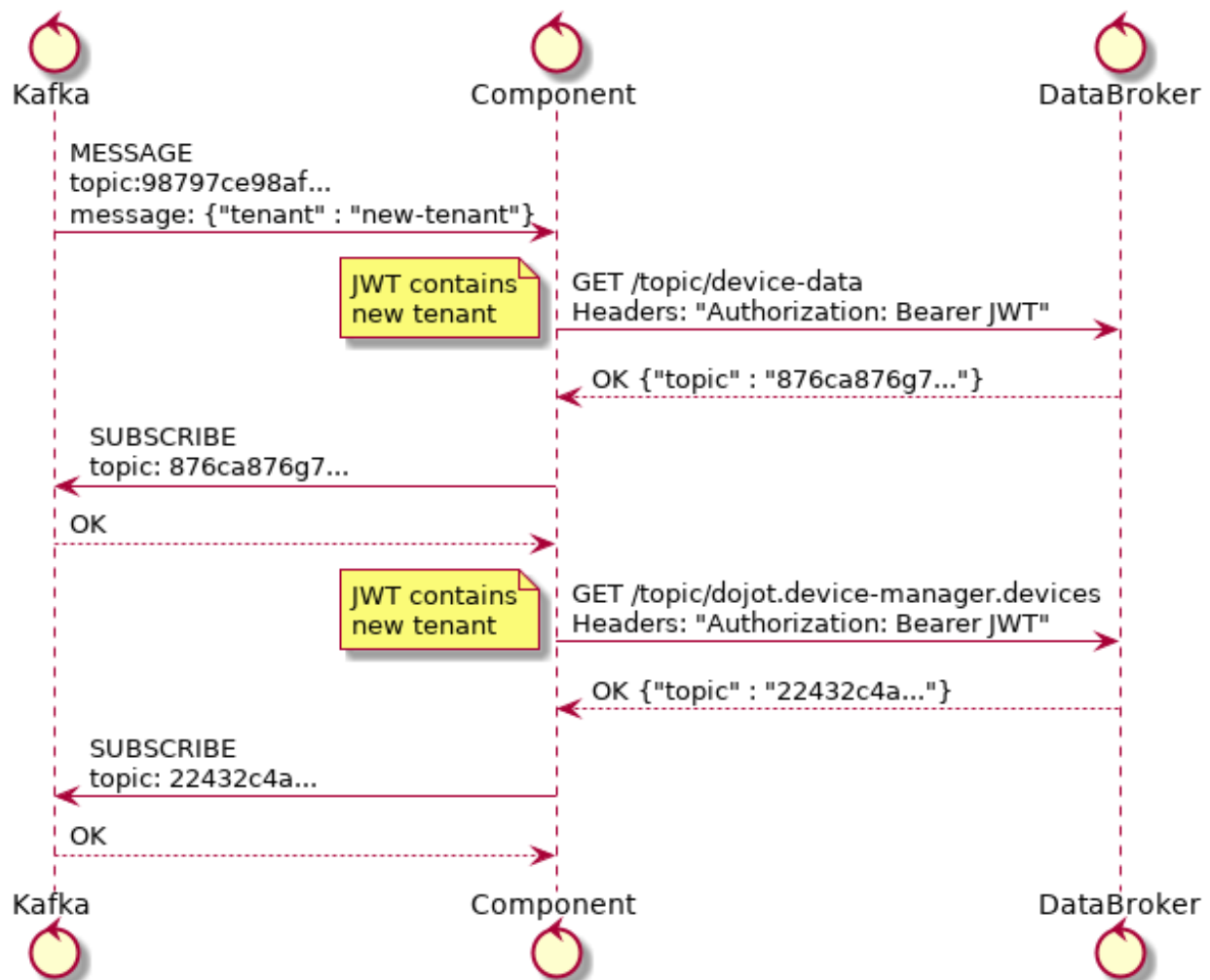


Fig. 5.8: Tenant bootstrapping

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)

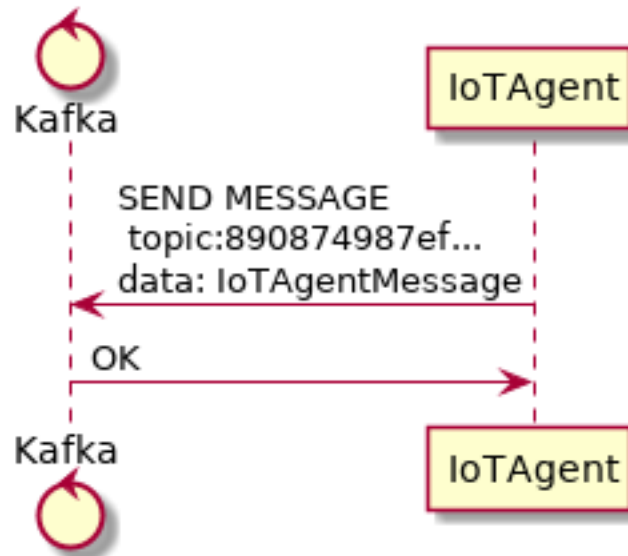


Fig. 5.9: IoT agent message to Kafka

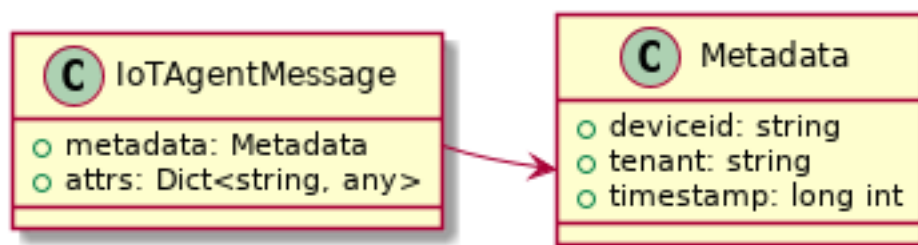


Fig. 5.10: IoT agent message structure

(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](#).

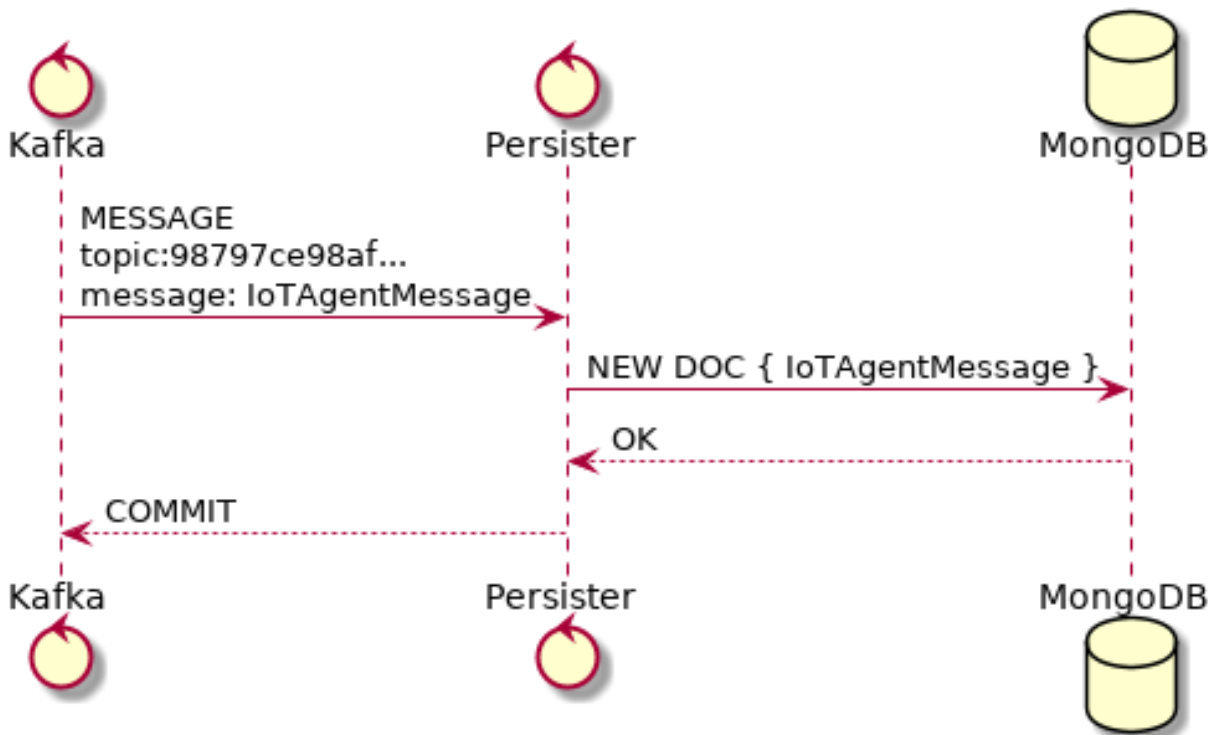


Fig. 5.11: Persister

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*

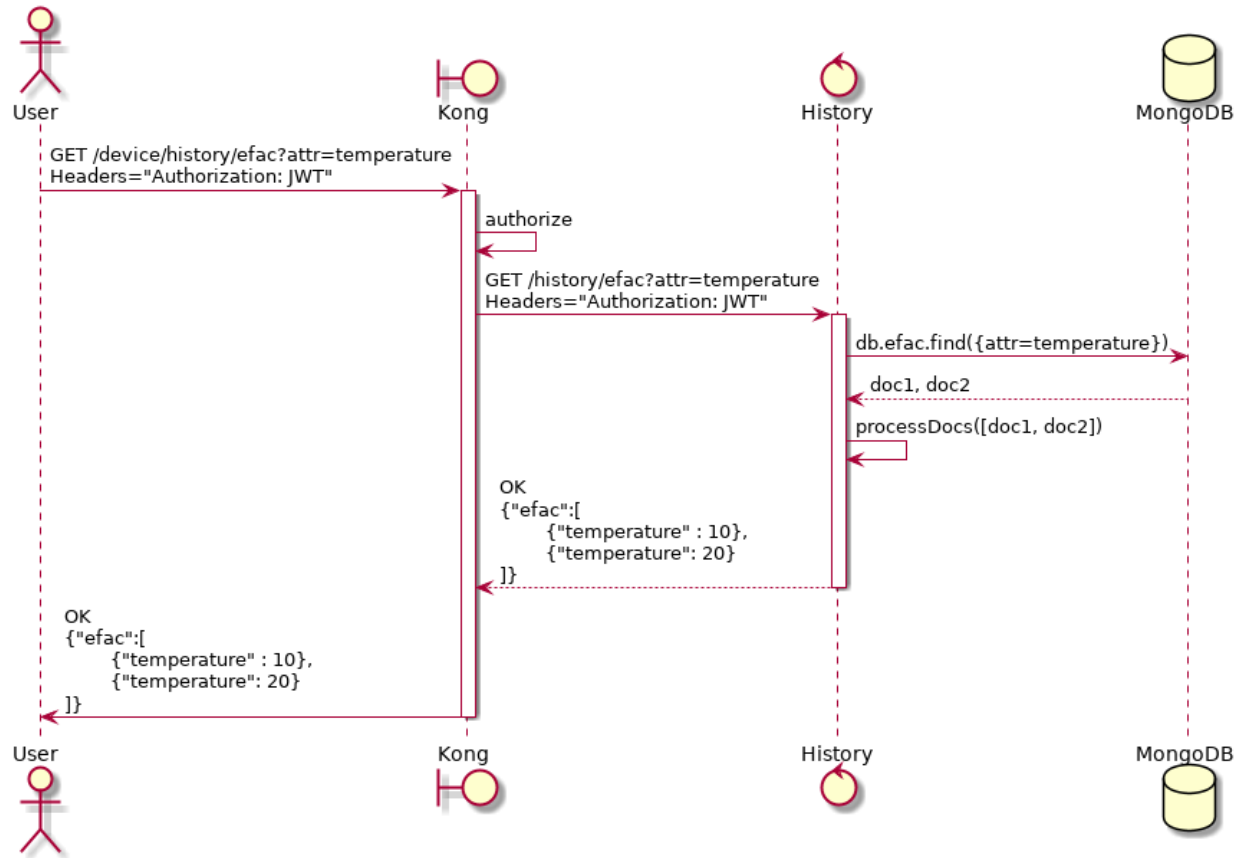


Fig. 5.12: History

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](#).

5.9 Flowbroker

TODO!

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*
 - *Kubernetes Requirements*
 - *dojot Deployment*
 - * *1. Cloning the repository*
 - * *2. Installing dependencies*
 - * *3. Configuring the inventory*
 - * *4. Executing the deployment playbook*
 - * *5. Accessing the deployed dojot environment*

6.1 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)

6.2 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.

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/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/>

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 v0.3.1 -b v0.3.1
```

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/>

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

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 minimum Kubernetes supported version is **v1.11**.
- 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 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.

```
kubect1 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.

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?*

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

7.1.2 Where can I get it?

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

7.1.3 Which repository is the main one?

There are 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.

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 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.

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 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.

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 `/<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 }
```

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. 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?](#).

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

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 eyJhbGciOiJIUzI1NiIsInR5cGU6IjY9/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.

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 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.

7.4.9 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.

7.4.10 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.

7.4.11 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.

7.4.12 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.

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, 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?*.

8.1 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
-

9.1 Device management

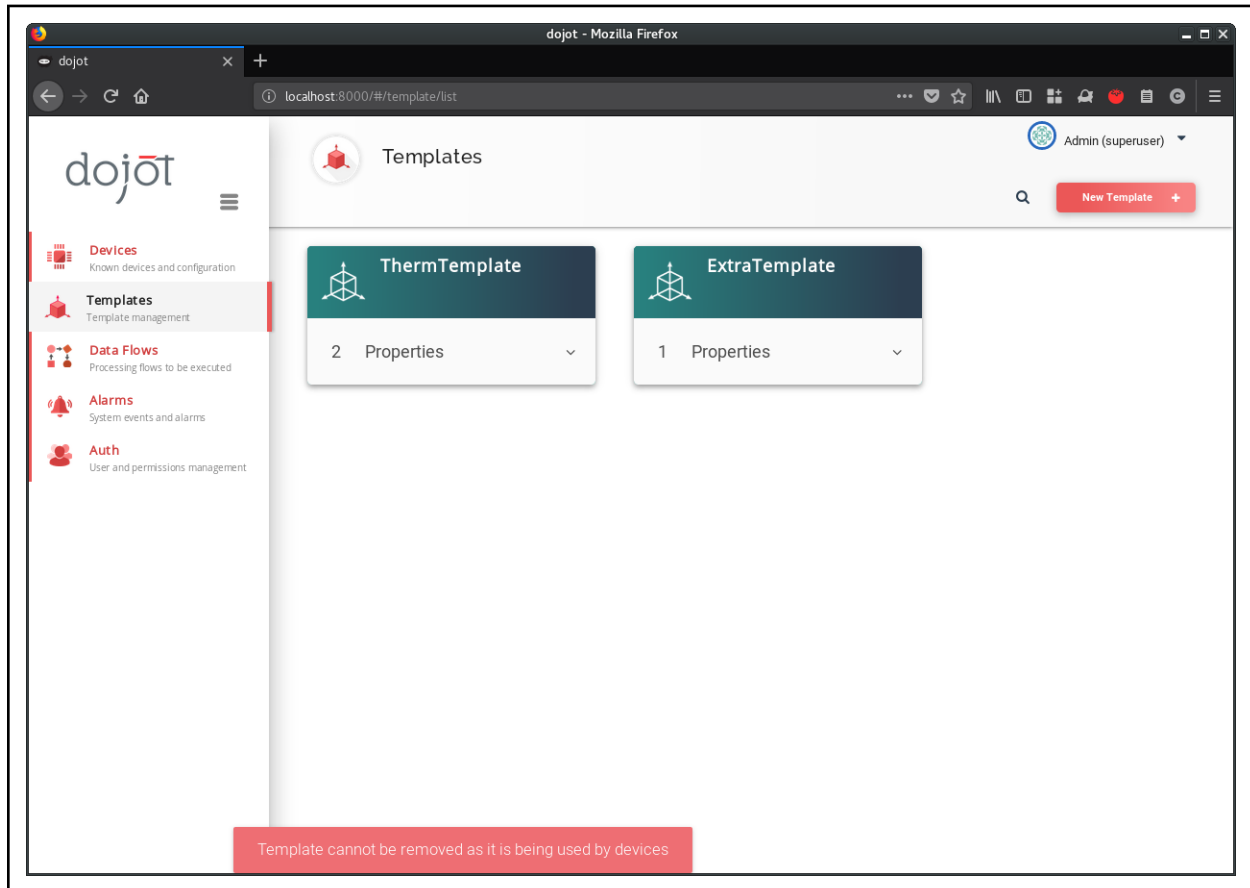
This section will show how to manage device. For this tutorial we will show how to add two thermometers and a virtual device that will represent an alarm system that will monitor both sensors.

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.

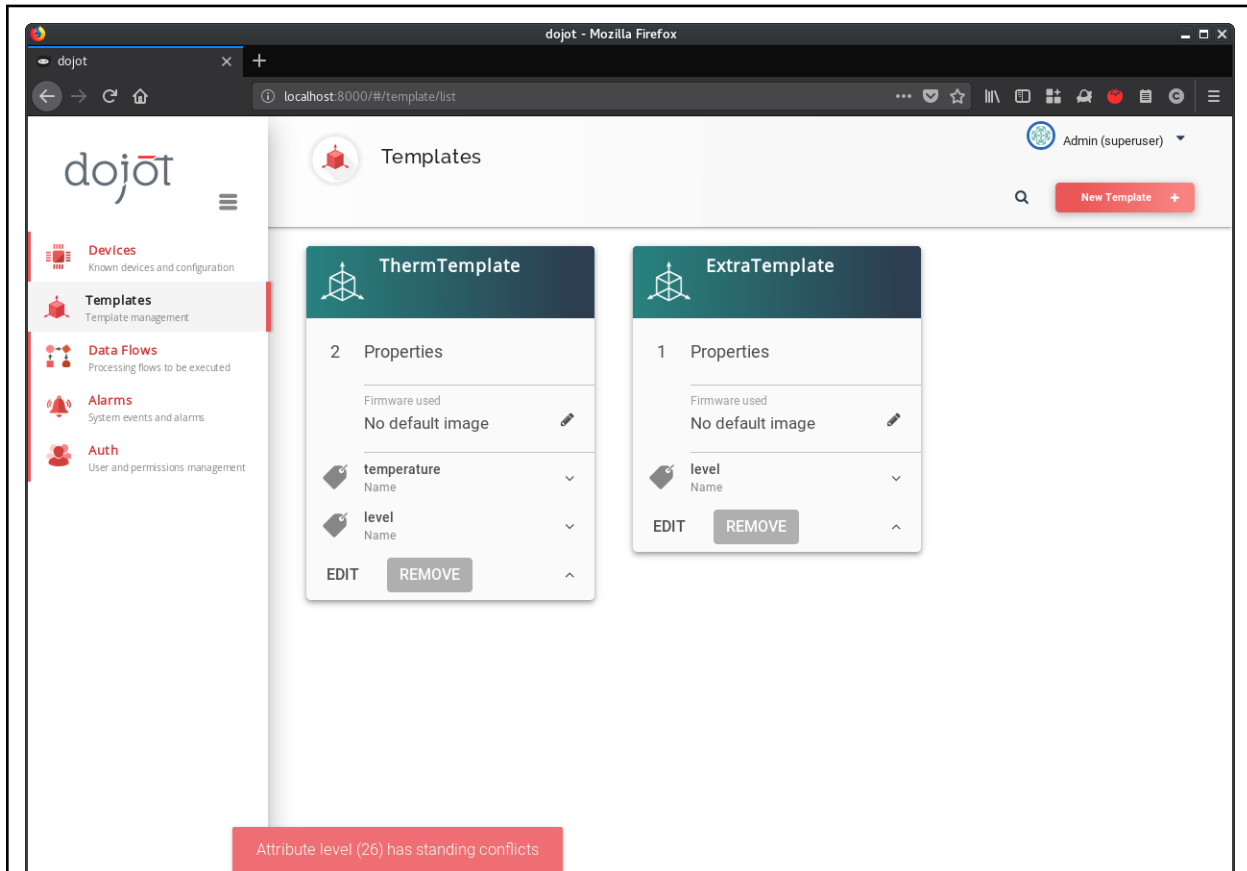
Now we have one template from which devices can be “instantiated”. All devices based on it will accept messages via MQTT that are sent to “/devices/thermometers” topic. To create new devices, you should go back to the devices tab and create a new device, selecting the templates it will be based on, as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

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 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.

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 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 `${JWT}`, which contains the token got from auth component.

10.2 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"
    }
  ]
}
```

10.3 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`).
- 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.

10.4 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/0998/attrs -m '{"temperature": 36.5}'  
mosquitto_pub -t /admin/0998/attrs -m '{"temperature": 15.6}'  
mosquitto_pub -t /admin/0998/attrs -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. More operators are available in [history APIs](#).

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: 10 min
-

11.1 Dojot nodes

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

11.1.1 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. 11.1 will be displayed.

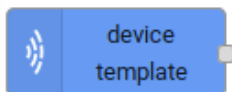
Fig. 11.1: : Device in configuration window

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.

11.1.2 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.
- **Device** (*required*): The *dojot* device that will trigger the flow.
- **Status** (*required*): Choose if devices status changes will trigger or not the flow.

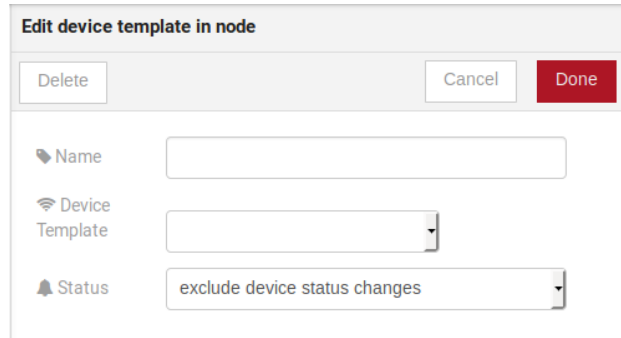
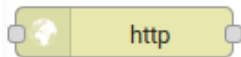


Fig. 11.2: : Device template in configuration window

11.1.3 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.

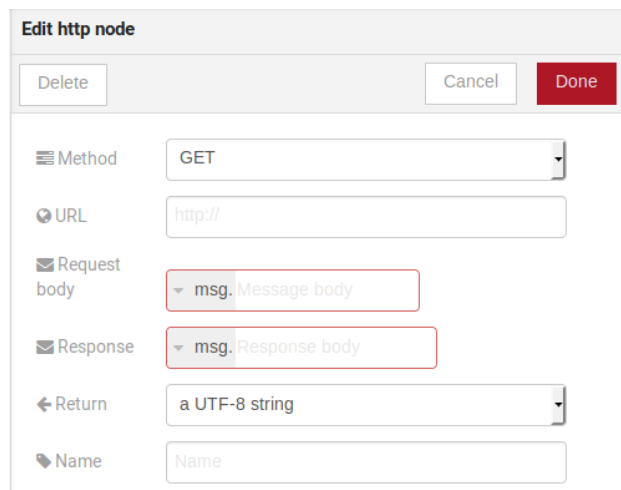
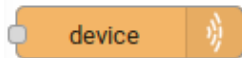


Fig. 11.3: : Device template in configuration window

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.

11.1.4 Device out



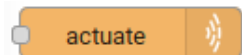
Device out will determine which device will have its attributes updated on *dojot* according to the result of the flow. Bear in mind that this node doesn't send messages to your device, it will only update the attributes on the platform. Normally, the chosen device out is a *virtual device*, which is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

Fig. 11.4: : 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

11.1.5 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. 11.5: : 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

11.1.6 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.

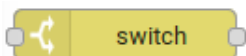
Fig. 11.6: : Change configuration

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.

11.1.7 Switch



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

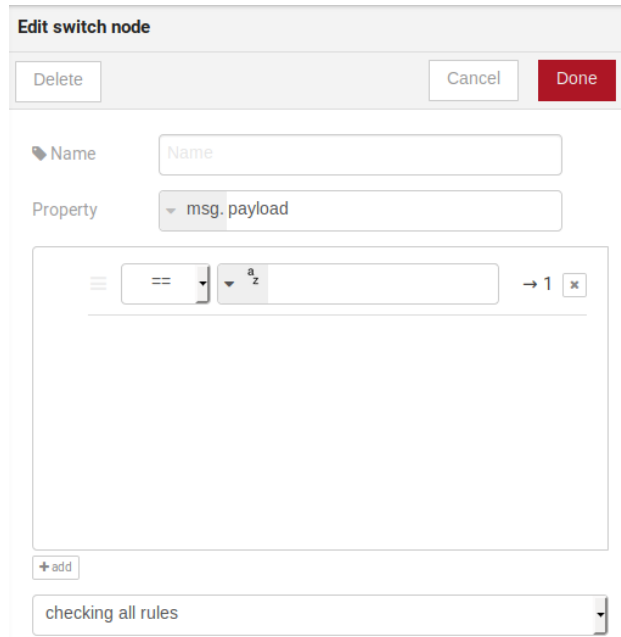


Fig. 11.7: : Switch configuration

Fields:

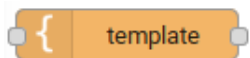
- **Name** (*optional*): Name of the node
- **Property** (*required*): Variable that will be evaluated
- **Rule box** (*required*): Rules that will determine the output branch of the node. Also, it can be configured to stop checking rules when it finds one that matches other or check all the rules and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

11.1.8 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...

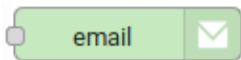
It uses the [mustache](#) template language. Check [Fig. 11.8](#) as example: the field **a** of payload will be replaced with the value of the **payload.b**

Fig. 11.8: : 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

11.1.9 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.
- **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.

The screenshot shows a dialog box titled "Edit email node". At the top, there are three buttons: "Delete", "Cancel", and "Done". Below these are several input fields with email-related icons:

- From:** email@address.com
- To:** email@address.com
- Server:** gmail-smtp-in.l.google.com
- Subject:** Message title
- Body:** msg.Message body
- Name:** Name

Fig. 11.9: : Email configuration

11.1.10 Geofence

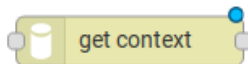


Select an interest area to determine wich devices will activate the flow

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

11.1.11 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

11.2 Learn by examples

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. 11.10: : 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 list of inputs with a checked checkbox for 'Context layer' set to 'Tenant', a checked checkbox for 'Context name' with a text field 'Context name', and a checked checkbox for 'Context content' with a text field 'Property to store the context content'.

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

11.2.1 Using template and email nodes

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

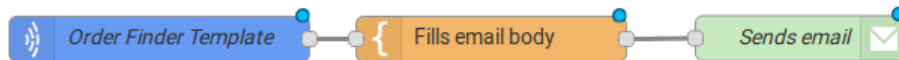


Fig. 11.11: : Flow using template and email nodes

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. 11.12.

Then, the email body on the email node should be assigned to the variable that is on the field *Set property* on Fig. 11.12:

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

11.2.2 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. 11.18 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:

The screenshot shows the 'Edit template node' configuration window. At the top, there is a 'Delete' button. Below it, a description field contains 'Fills email body'. The 'Set property' section has a dropdown menu showing 'msg. output.emailBody'. The 'Format' section has a dropdown menu showing 'Mustache template'. The 'Template' section contains a text area with the following content:

```
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.
```

At the bottom, the 'Output as' section has a dropdown menu showing 'Plain text'.

Fig. 11.12: : Template configuration

The screenshot shows the 'Edit email node' configuration window. At the top, there is a 'Delete' button. Below it, several fields are configured:

- 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. 11.13: : Email node configuration

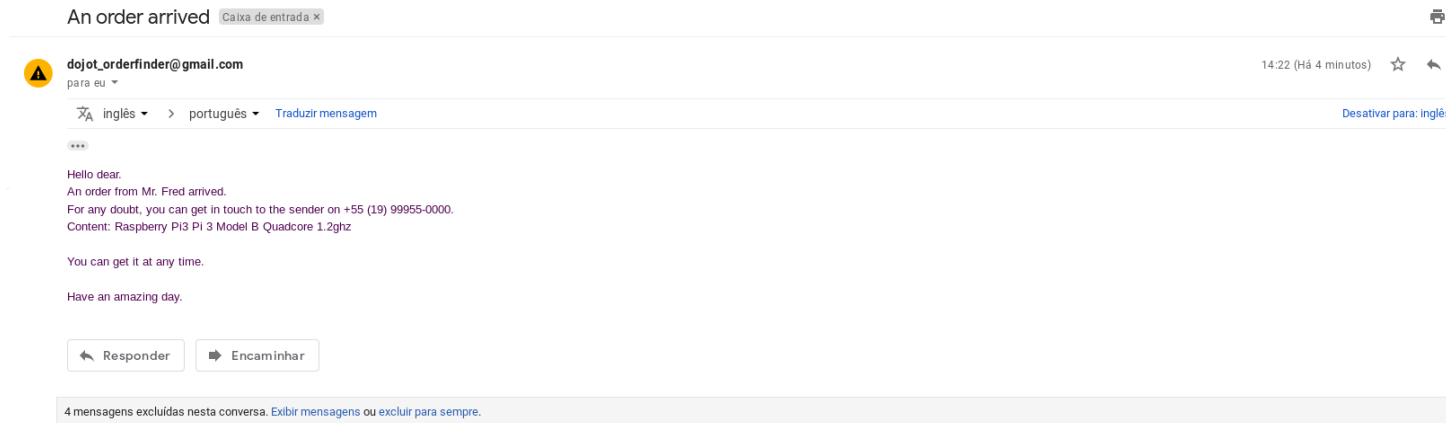


Fig. 11.14: : Sent email

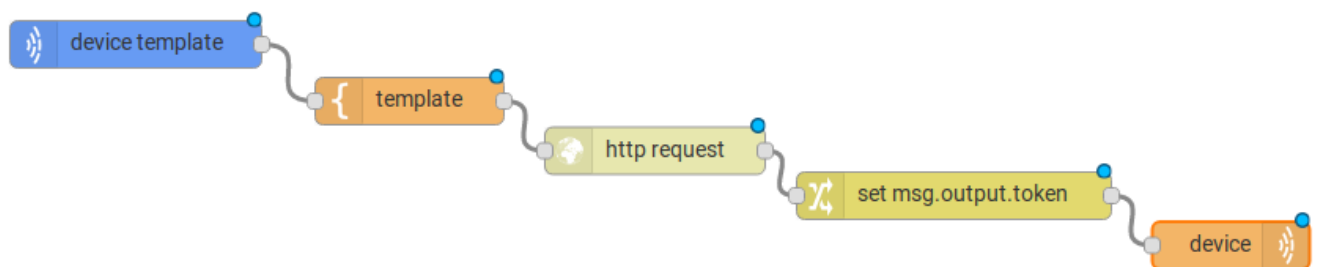
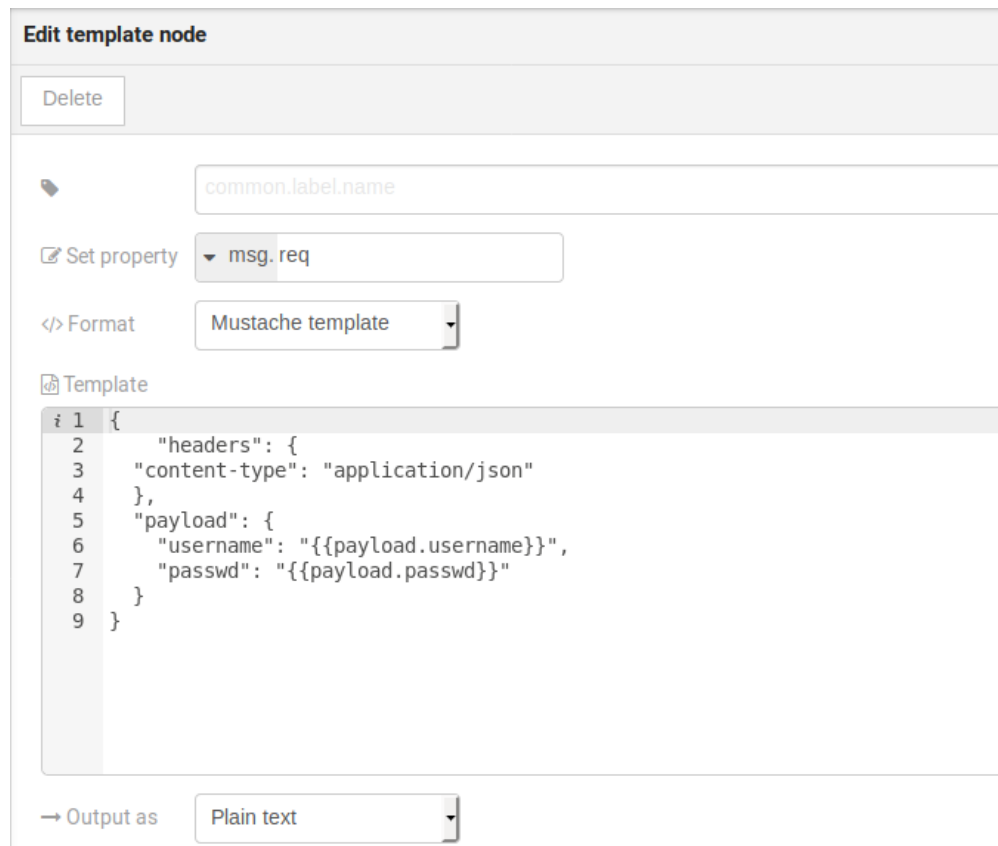


Fig. 11.15: : Flow used to explain http node



Edit template node

Delete

common.label.name

Set property ▼ msg.req

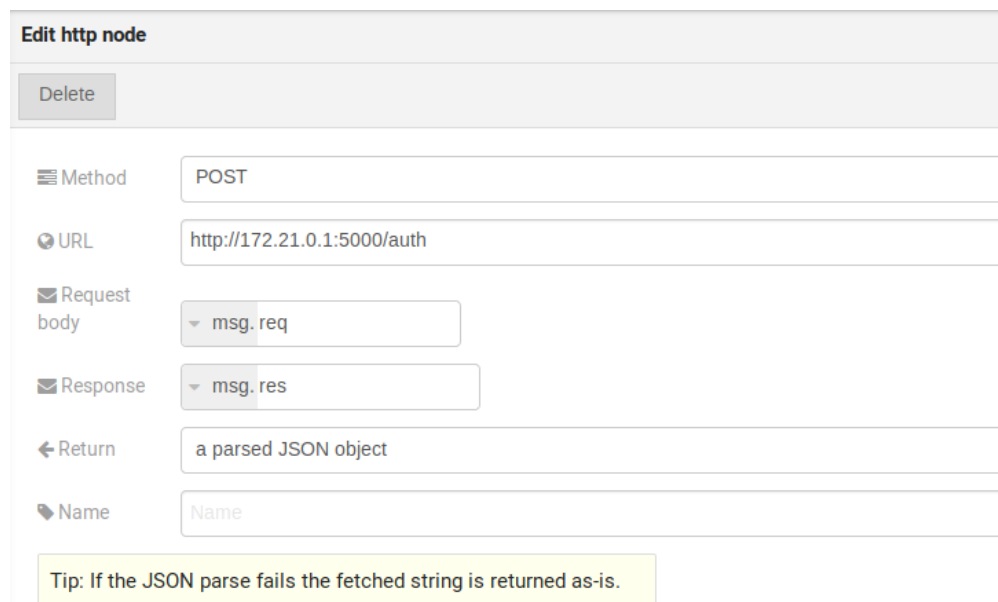
</> Format Mustache template

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. 11.16: : 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. 11.17: : Template node configuration

The screenshot shows the 'Edit change node' configuration window. At the top 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 of the rules area is a '+ add' button.

Fig. 11.18: : Template node configuration

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

Fig. 11.19: : Device out configuration

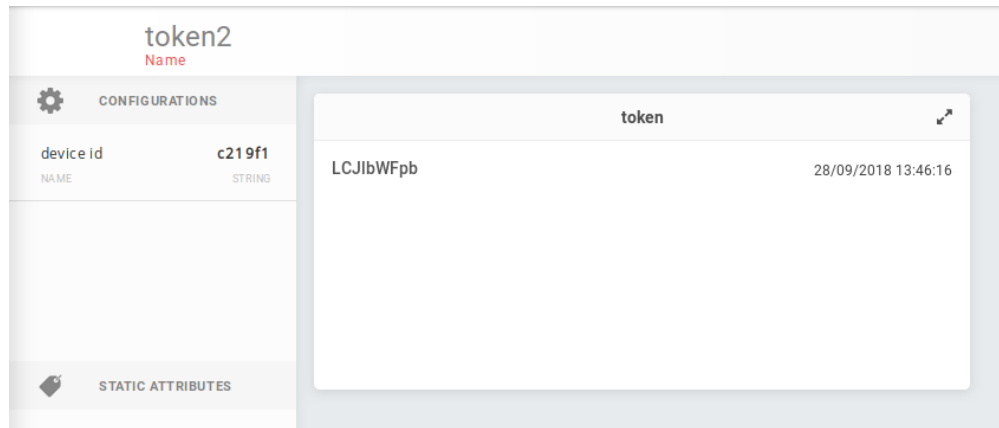


Fig. 11.20: : Device updated

11.2.3 Using geofence node

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

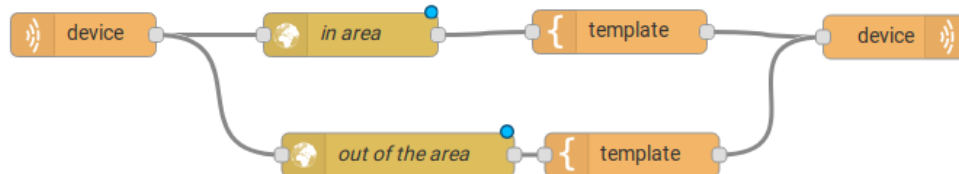


Fig. 11.21: : Flow using geofence

The geofence node named *in area* is set like seen in Fig. 11.22. 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"

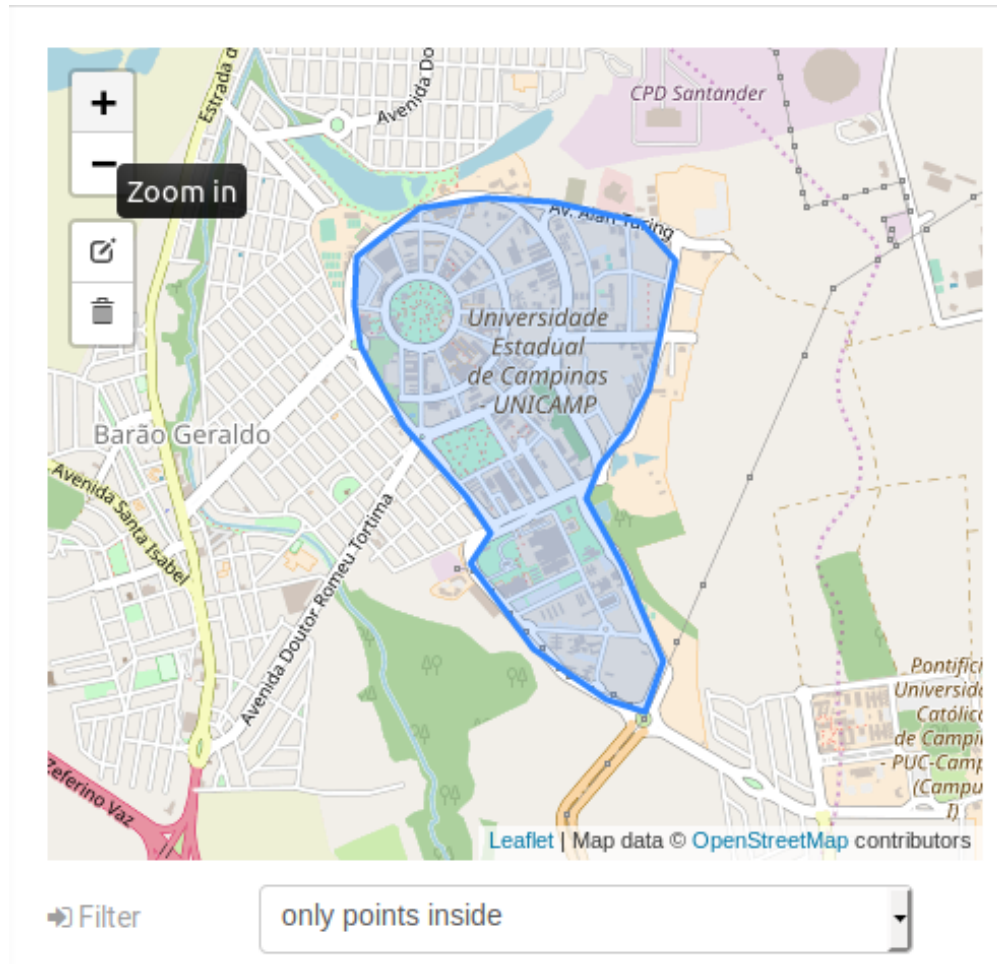


Fig. 11.22: : Geofence node configuration

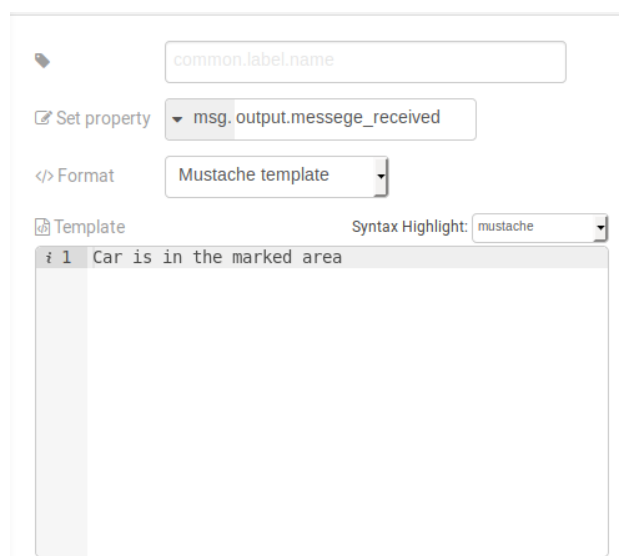
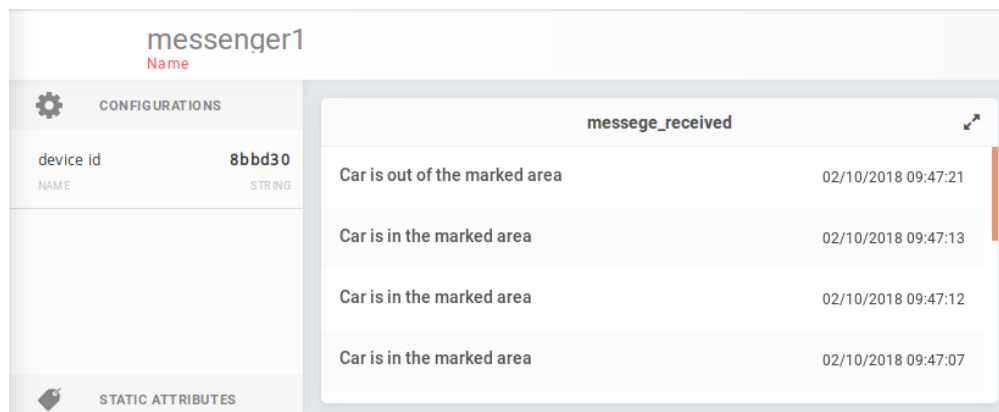


Fig. 11.23: : Template node configuration if the car is in the marked area



The screenshot displays the DojoT interface for a device named 'messenger1'. The interface is divided into two main sections: 'CONFIGURATIONS' on the left and 'STATIC ATTRIBUTES' on the right. The 'CONFIGURATIONS' section shows a table with one row: 'device id' with the value '8bbd30'. The 'STATIC ATTRIBUTES' section shows a table with one row: 'messege_received' with a value of 'Car is out of the marked area' and a timestamp of '02/10/2018 09:47:21'. Below this, there are three more rows with the value 'Car is in the marked area' and timestamps '02/10/2018 09:47:13', '02/10/2018 09:47:12', and '02/10/2018 09:47:07'.

messenger1	
Name	
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. 11.24: : Output in device out