dojot Documentation

Release 0.2.0

Matheus Magalhaes

Apr 13, 2018

Contents:

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

CHAPTER 1

Architecture

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

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

Table of Contents

1.1 Components

dojot was designed to make fast solution prototyping possible, providing a platform that's easy to use, scalable and robust. Its internal architecture makes use of many well-known open-source components with others designed and implemented by dojot team. This architecture is described on Fig. 1.1.

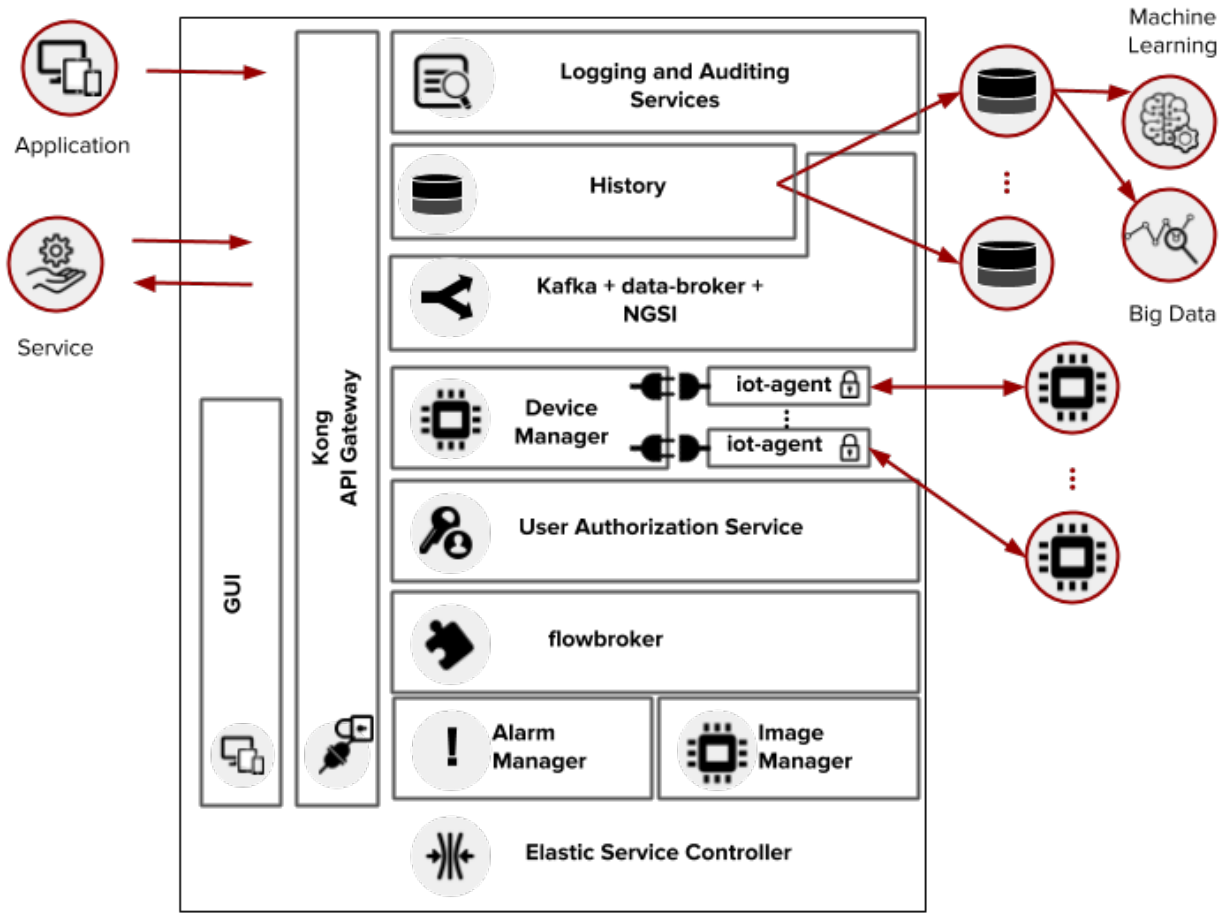

Fig. 1.1: Current Architecture

Using dojot is as follows: a user configures IoT devices through the GUI or directly using the REST APIs provided by the API Gateway. Data processing flows might be also configured - these entities can perform a variety of actions, such as generate notifications when a particular device attribute reaches a certain threshold or save all data generated by a device onto an external database. As devices start sending their readings to dojot, the user might want to receive these readings via notifications generated by subscriptions, consolidate all data into virtual devices, gather all data from historical database, and so on. These features can be used through REST APIs - these are the basic building blocks that any application based on dojot should use. dojot GUI provides an easy way to perform management operations for all entities related to the platform (users, devices, templates and flows) and can also be used to check if everything is working fine.

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

the interested services.

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

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

1.1.1 Kafka + data-broker + NGSI

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

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

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

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

1.1.2 DeviceManager

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

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

1.1.3 IoT Agent

An IoT agent is an adaptation service between physical devices and *dojot's* core components. It could be understood as a *device driver* for a set of devices. The *dojot* platform can have multiple iot-agents, each one of them being specialized in a specific protocol like, for instance, MQTT/JSON, CoAP/LWM2M and HTTP/JSON.

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

1.1.4 User Authorization Service

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

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

1.1.5 flowbroker

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

1.1.6 History

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

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

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

1.1.7 Logging and Auditing Service

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

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

Such components are currently in development.

1.1.8 Kong API Gateway

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

1.1.9 GUI

The Graphical User Interface in *dojot* is responsible for providing responsive interfaces to manage the platform, including functionalities like:

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

1.1.10 Elastic Service Controller

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

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

This component is currently scheduled for development.

1.1.11 Alarm Management

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

1.1.12 Image manager

This component is responsible for device image storage and retrieval.

1.2 Infrastructure

A few extra components are used in dojot that were not shown in Fig. 1.1. They are:

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

1.3 Communications

All components communicate with each other in two ways:

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

CHAPTER 2

User Guide

This document provides information on how to use dojot from a device developer or application developer point of view.

Table of Contents

2.1 Who should read this

- Users that want a deeper look at how dojot works;
- Application developers.

2.2 Getting Started

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

2.3 dojot basics

Before using dojot, you should be familiar with some basic operations and concepts. They are very simple to understand and use, but without them, all operations might become obscure and senseless.

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.

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

2.3.1 User authentication

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

- User identity
- Validation data
- Token expiration date

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

2.3.2 Devices and templates

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

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

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

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

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

let's say *TexasInstr882*, and the 'thermometer' template. The user would have also to add translation instructions (implemented in terms of data flows, build in flowbuilder) in order to map the temperature reading that will be sent from the device to a "temperature" attribute.

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

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

2.3.3 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.

2.3.4 Step-by-step device management

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

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

Getting access token

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

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

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

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

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

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

Device creation

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

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

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

This request should give back this message:

```
{
  "result": "ok",
  "template": {
    "created": "2018-01-25T12:30:42.164695+00:00",
    "data_attrs": [
      {
        "template_id": "1",
        "created": "2018-01-25T12:30:42.167126+00:00",
        "label": "temperature",
        "value_type": "float",
        "type": "dynamic",
        "id": 1
      }
    ],
    "label": "Thermometer Template",
    "config_attrs": [],
    "attrs": [
      {
        "template_id": "1",
        "created": "2018-01-25T12:30:42.167126+00:00",
        "label": "temperature",
        "value_type": "float",
        "type": "dynamic",
        "id": 1
      }
    ],
    "id": 1
  }
}
```

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

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

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

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

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

Which should give back:

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

Sending messages

So far we got an access token and created a template and a device based on it. In an actual deployment, the physical device would send messages to dojot with all its attributes and their current values. For this tutorial we will send

MQTT messages by hand to the platform, emulating such physical device. For that, we will use mosquitto_pub from Mosquitto project.

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

The 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 FAQ, there are some considerations regarding MQTT topics:

- If you don't define any topic in device template, it will assume the pattern `/<service-id>/<device-id>/attrs` (for instance: `/admin/efac/attrs`). This should be the topic to which the device will publish its information to.
- If you do define a topic in device template, then your device should publish its data to it and set the client-id parameter. It should follow the following pattern: `<service>:<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 }
```

For more information on how dojot deals with data sent from devices, check the *Integrating physical devices* section.

Checking historical data

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

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

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

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

The history endpoint is built from these values:

- `.../device/3bb9/...`: the device ID is `3bb9` - this is retrieved from the `id` attribute from the device
- `.../history?lastN=3&attr=temperature`: the requested attribute is temperature and it should get the last 3 values. More operators are available in history APIs.

The request should result in the following message:


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


This message contains all previously sent values.

2.4 Integrating physical devices

If you want to integrate your device within dojot, it must be able to send messages to the platform. There are two ways to do that:

- Use one of the available IoT agents: currently, there is support for MQTT-based devices. If your project is using (or allows changing to) this protocol, then it would suffice to check if the device is sending its data using a simple key/value JSON. If it isn't, then you might want to use iotagent-mosca (check iotagent-mosca documentation to check out how to do that). If it is indeed sending key/value JSON messages, then it can send its messages to dojot's broker and it will be recognized by the platform.
- Create a new IoT agent to support the protocol used by the device: if your device is using another protocol that is not yet supported, then it might be a good idea to implement a new IoT agent. It's not that hard, but there are a few details that must be taken into account. To help developers to do such thing, there is the iotagent-nodejs library which deals with most internal mechanisms and messages - check its documentation to know more.

After your device is able to communicate with dojot, you can start using it as described in *Step-by-step device management*.

CHAPTER 3

Components and APIs

3.1 Components

Table 3.1: Components

Component	GitHub repository	Documentation
mongodb		mongodb documentation
postgres		postgres documentation
Kong API gateway		Kong documentation
redis		Redis documentation
zookeeper		Zookeeper documentation
Kafka		Kafka documentation
auth	GitHub - auth	readthedocs - auth
History	GitHub - history-ws	
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	

3.2 Exposed APIs

Table 3.2: APIs :header-rows: 1

Endpoint	Purpose	Component API	Repository
/device	Device management	API - DeviceManager	GitHub - DeviceManager
/template	Template management	API - DeviceManager	GitHub - DeviceManager
/flows	Flow management	API - flowbroker	GitHub - flowbroker
/auth	User authentication	API - auth	GitHub - auth
/auth/revoke	User authentication	API - auth	GitHub - auth
/auth/user	User authentication	API - auth	GitHub - auth
/history	Device historical data	API - history-ws	GitHub - history-ws
/metric	Context broker	API - data-broker	GitHub - data-broker
/gui	Graphical User Interface		GitHub - GUI
/sign	Public key signing	API - EJBCA-REST	GitHub - EJBCA-REST
/ca	Certification-Auth. functions	API - EJBCA-REST	GitHub - EJBCA-REST
/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 3.3: 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
Data-Broker	host:1026/	host:8000/metric
GUI	host/	host:8000/gui
ejbca	host:5583/sign	host:8000/sign
ejbca	host:5583/ca	host:8000/ca

3.3 Kafka messages

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

Table 3.4: Original endpoints

Component	Message	Subject
DeviceManager	Device CRUD (Messages - DeviceManager)	`dojot.device-manager.device`
iotagent-mosca	Device data update (Messages - iotagent-mosca)	`device-data`

CHAPTER 4

Installation Guide

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

4.1 Installation - 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.

4.1.1 Dependencies

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

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/

Docker Compose

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

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

4.1.2 Installation

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

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

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

or as git clone command::

```
git clone 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
```

For instance:

```
git checkout 0.1.0-dojot
```

Or if you're brave enough:

```
git checkout master
```

That done, the environment can be brought up by:

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

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

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

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

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

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

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

4.1.3 Usage

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

Read the *User Guide* for more information about how to interact with the platform.

CHAPTER 5

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

5.1 General

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

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

It takes a role as an enabler platform with:

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

5.1.2 Where can I get it?

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

5.1.3 Which repository is the main one?

There are two main ones:

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

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

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

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

5.2 Usage

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

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

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

And that's it.

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

REST APIs are explained in the *Applications* section.

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

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

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

You need to follow some steps:

1 Update the docker-compose repository to the cutting-edge version (beware the bugs though)

```
$ 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
```

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

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

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

5.3 Devices

5.3.1 What are *devices* for dojot?

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

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

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

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

It is as simple as it seems: the *regular device* for dojot is a mirror (digital twin) of your actual device. You can choose which attributes are available for applications and other components by adding each one of them at the device creation interface.

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

Regular devices are created to serve as a mirror (digital twin) for the actual devices and sensors. A *virtual device* is an abstraction that models things that are not feasible in the real world. For instance, let's say that a user has few smoke detectors in a laboratory, each one with different attributes.

Wouldn't it be nice if we had one device called *Laboratory* that has one attribute *isOnFire*? Therefore, the applications could rely only on this attribute to take an action.

Another difference is how virtual devices are populated. Regular ones will be filled with information sent by devices or gateways to the platform and virtual ones will be filled by flows or by applications.

5.3.4 And what are *templates*?

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

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

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

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

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

- If you don't define any topic in device template, it will assume the pattern `/<service-id>/<device-id>/attrs` (for instance: `/admin/efac/attrs`). This should be the topic to which the device will publish its information to.
- If you do define a topic in device template, then your device should publish its data to it and set the client-id parameter. It should follow the following pattern: `<service>:<deviceid>`, such as `admin:efac`.
- MQTT payload must be a JSON with each key being an attribute of the dojot device, such as:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
[
  {
    "device_id": "3bb9",
    "ts": "2018-03-22T13:47:07.050000Z",
    "value": 29.76,
    "attr": "temperature"
  },
  {
    "device_id": "3bb9",
    "ts": "2018-03-22T13:46:42.455000Z",
    "value": 23.76,
    "attr": "temperature"
  },
  {
    "device_id": "3bb9",
    "ts": "2018-03-22T13:46:21.535000Z",
    "value": 25.76,
    "attr": "temperature"
  }
]
```

There are more operators that could be used to filter entries. Check history-ws API documentation to check out all possible operators.

5.4 Data Flows

5.4.1 What is data flow?

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

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

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

5.4.3 Why should I use it?

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

5.4.4 What can it do, exactly?

You can do things such as:

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

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

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

5.4.5 So, how can I use it?

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

5.4.6 Can I apply the same flow to multiple devices?

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5.5 Applications

5.5.1 What APIs are available for applications?

You can check all available APIs in the API Listing page

5.5.2 How can I use them?

There is a very quick and useful tutorial in the User Guide.

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

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

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

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