iotagent-json Documentation

Release 0.2.0a1

Matheus Magalhaes

Oct 02, 2018

Contents:

IoT agents are responsible for receiving messages from physical devices (directly or through a gateway) and sending them commands in order to configure them. This iotagent-json, in particular, receives messages via MQTT with JSON payloads.

CHAPTER 1

Concepts

1.1 MQTT

MQTT is a somewhat simple protocol: it follows a publish/subscriber paradigm and messages are exchanged using topics. These topics are simple strings such as `/admin/cafe/attrs`. A publisher can, well, publish messages by sending them to a MQTT broker using a particular topic and all the subscribers that are listening to that topic will receive a copy of the message.

Subscribers can listen not only to specific topics, but also to topics with wildcards. For instance, one could use a '+' to indicate that any token will match the subscribed topic, such as `/admin/+/attrs` - messages sent to both `/admin/cafe/attrs` and `/admin/4593/attrs`, for instance, will be received by this subscriber. Another possibility is to create a subscription to all remainder tokens in the topic, such as `/admin/#`. All messages sent to topics beginning with `/admin/` will be received by this subscriber.

1.2 Kafka

Kafka is, in fact, a project from the Apache Foundation. It is a messaging system that is similar to MQTT in the sense that both are based on publisher/subscriber. Kafka is way more complex and robust - it deals with multiple subscribers belonging to the same group (and performs load-balancing between them), stores and replays messages, and so on. The side effect is that its clients are not that simple, which could be a heavy burden for tiny devices.

CHAPTER 2

Operation

2.1 Configuration

iotagent-json configuration is pretty simple. The main and only configuration file is `config.json`, placed at the repository root directory. For instance, the default configuration file looks like:

```
{
  "mqtt": {
    "host": "mqtt",
    "port" : 1883,
    "protocolId": "MQIsdp",
    "protocolVersion": 3,
    "secure": false,
    "tls": {
      "key": "certs/iotagent.key",
      "cert": "certs/iotagent.crt",
      "ca": [ "certs/ca.crt" ],
      "version": "TLSv1_2_method"
    }
  },
  "broker": {
    "host": "zookeeper:2181",
    "type": "kafka",
    "subject": "device-data",
    "contextBroker": "http://data-broker"
  },
  "device_manager": {
    "consumerOptions": {
      "kafkaHost" : "kafka:9092",
      "sessionTimeout": 15000,
      "groupId": "iotagent"
    },
    "inputSubject": "dojot.device-manager.device"
  },
```

(continues on next page)

(continued from previous page)

```
  "tenancy": {
    "manager": "http://auth:5000",
    "subject": "dojot.tenancy",
    "consumerOptions": {
      "kafkaHost" : "kafka:9092",
      "sessionTimeout": 15000,
      "groupId": "iotagent"
    }
  }
}
```

There are four things to configure:

- MQTT: where the device messages will come from.
- MQTT Security: if used (and you should be using), these are the things that must be configured. They are related to the communication between iotagent-json and the physical device.
- Data broker: where to send device information updates. There is support for Kafka (sending a message to every component that is interested in device updates) and for Orion (context broker from Fiware project).
- Device manager access: how the device manager will send device notifications to iotagent (creation, update and removal).
- Tenancy: how iotagent-json will get tenant-related information, such as which are the tenants currently configured in dojot.

Check dojot documentation if you don't know or don't remember all the components and how and why they communicate to each other.

2.2 Receiving messages from DeviceManager via Kafka

Messages containing device operations should be in this format:

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

    }
  }
}
```

These messages are related to device creation, update, removal and actuation. For creation and update operations, it contains the device data model to be added or updated. For removal operation, it will contain only the device ID being removed. The actuation operation will contain all attributes previously created with their respective values.

The documentation related to this message can be found in DeviceManager Messages.

2.2.1 Device configuration for iotagent-json

The following device attributes are considered by iotagent-json. All these attributes are of `meta` type (with the exception of translator instructions, with type is `meta-translator` and their values are in `static_value` attribute property.

Table 2.1: Device attributes for iotagent-json

Attribute	Description	Example
topic	Topic to which the device will publish messages.	`/admin/efac/attrs`
topic-config	Topic from which the device will accept actuation messages.	`/admin/efac/configuration`
id-location	Where can the physical device identifier be located.	Check *ID-location structure table*.
translator	Instructions to transform the message sent by the device to a simple key-value JSON structure.	`{ "op": "move", "from": "/data/Coils/e/1/bv", "path": "/temperature", "optional": true }` Keep in mind that this JSON should be "stringified", i.e., all special caracters should be escaped. This follows the JSON patch definitions with one important difference: if the patch can't be applied (because the message has no such attribute), the procedure won't fail - that's the purpose of the `optional` attribute. Also, check the definition of a JSON pointer to understand how to reference items inside a list.

The translator described in the table would move the value from `/data/Coils/e/1/bv` to `/temperature`, transforming the message published by the device:

```
{
  "data" : {
    "Coils" : {
      "e": [
        { "bv" : 0.5 },
        { "bv" : 27.5 }
      ]
    }
  }
}
```

into:

```
{
  "temperature" : 27.5
}
```

If the device can't be updated to send messages using the identifier specified by dojot, iotagent-json can be configured to detect whatever "physical" ID (let's call it as *physical device ID*) this device has in order to properly map it to the dojot one (let's call it *dojot device ID*). This configuration is done by the `id-location` device attribute, which is described by the table below. If none is specified, then iotagent-json will assume a default behavior, which considers the ID as the second token of MQTT topic, such as: `/admin/efac/attrs` with physical device ID being `efac`.

Table 2.2: ID-location structure

Attribute	Description	Example
type	Where does the device physical ID can be found.	Possible values are: • `mqtt-topic`: The physical device ID is in MQTT topic, such as /mqtt/admin/**efac**/attrs • `message-attribute`: The physical device ID is somewhere in the message which will be sent from the device. An example would be: – `{"attr1" : 10, "device-id" : "efac"}`
attribute_name	Which attribute has the physical device ID, if `id-location` is `mqtt-message`.	`device-id`, for a message like `{"attr1" : 10, "device-id" : "efac"}`
regexp	Regular expression applied to MQTT topic or selected attribute in order to extract physical device ID.	`\/.*?\/(.*?)\/.*` which extracts `efac` from `/admin/efac/attrs`
id	The physical device ID	BAFE88420 (any identifier specific to a device)
xid	Any message attribute that maps directly to these device ID resolution instructions.	`/c/devices/mqtt/` (the topic used by all devices)

The `xid` attribute should be understood as "I have these instructions for locating the device ID, but I don't know which one to use for this message - thus I'll test the `xid` attribute from each one of them against it". Currently, the `xid` is the MQTT topic used to publish the message.

Example

The following message serves as an example of a device with all attributes used by iotagent-json.

```
{
  "label": "Thermometer Template",
  "attrs": [
    {
      "label": "translator",
      "type": "meta-translator",
      "value_type": "string",
      "static_value": "{ \"op\": \"move\", \"from\": \"/data/Coils/e/1/bv\", \"path\
↪": \"/temperature\", \"optional\": true }"
    },
```

(continues on next page)

(continued from previous page)

```
    {
      "label": "id-location",
      "type": "meta",
      "value_type": "string",
      "static_value": "{\"xid\":\"/agent/main/000BABC80F4A/devinfo\",\"id\":\
↪"000BABC80F4A\",\"type\":\"mqtt-topic\",\"regexp\":\"\\/.*?\\/.*?\\/(.*?)\\/.*\"}"
    },
    {
      "label": "topic",
      "type": "meta",
      "value_type": "string",
      "static_value": "/agent/main/000BABC80F4A/devinfo"
    },
    {
      "label": "topic-config",
      "type": "meta",
      "value_type": "string",
      "static_value": "/agent/main/000BABC80F4A/config"
    },
    {
      "label": "temperature",
      "type": "dynamic",
      "value_type": "float"
    },
    {
      "label": "reset",
      "type": "actuator",
      "value_type": "boolean"
    }
  ]
}
```

For the sake of readability, below are both values for translator and id-location, with no escape characters.

translator:

```
{
  "op": "move",
  "from": "/data/Coils/e/1/bv",
  "path": "/temperature",
  "optional": true
}
```

id-location:

```
{
  "xid": "/agent/main/000BABC80F4A/devinfo",
  "id": "000BABC80F4A",
  "type": "mqtt-topic",
  "regexp": "\\/.*?\\/.*?\\/(.*?)\\/.*"
}
```

These configurations indicate that:

- The device will publish its messages to `/agent/main/000BABC80F4A/devinfo` topic;
- The device will receive commands via MQTT from topic `/agent/main/000BABC80F4A/config`

- Its ID is in MQTT topic, which can be extracted using the regular expression `\/.*?\/.*?\/(.*?)\/.*` and its ID should match 000BABC80F4A.
- The message should be transformed from:

```
{
  "data" : {
    "Coils" : {
      "e": [
        { "bv" : 0.5 },
        { "bv" : 27.5 }
      ]
    }
  }
}
```

into:

```
{
  "temperature" : 27.5
}
```

- These instructions should be applied whenever a message to the topic `/agent/main/000BABC80F4A/devinfo` is received.

2.3 Sending messages to other components via Kafka

When iotagent-json receives a new message from a particular device, it must publish the new data to other components. The default subject used to publish this information is "device-data". Check data-broker documentation to check how these subjects are translated into Kafka topics.

The message sent by iotagent-json is like this one:

```
{
  "metadata": {
    "deviceid": "efac",
    "protocol": "mqtt",
    "payload": "json"
  },
  "attrs": {
  }
}
```

As previously stated, the "attrs" attribute is the same as the one from DeviceManager Messages.

2.4 Receiving messages from devices via MQTT

Any message payload sent to iotagent-json must be in JSON format. Preferably, they should follow a simple key-value structure, such as:

```
{
  "speed": 100.0,
  "weight": 50.2,
```

(continues on next page)

(continued from previous page)

```
  "id": "truck-001"
}
```

If not possible, you could make use of `translator` attributes so that you get more flexibility on device message formats.

2.4.1 Example

This example uses `mosquitto_pub` tool, available with `mosquitto_clients` package. To send a message to iotagent-json via MQTT, just execute this command:

```
mosquitto_pub -h localhost -t /admin/efac/attrs -m '{"speed" : 10}'
```

This command will send the message containing one value for attribute `speed`. The device ID is `efac`. `-t` flag sets the topic to which this message will be published.

This command assumes that you are running iotagent-json in your machine (it also works if you use dojot's docker-compose).

CHAPTER 3

How to build/update/translate documentation

If you have a local clone of this repository and you want to change the documentation, then you should follow this simple guide.

3.1 Build

The readable version of this documentation can be generated by means of sphinx. In order to do so, please follow the steps below. Those are actually based off ReadTheDocs documentation.

```
pip install sphinx sphinx-autobuild sphinx_rtd_theme sphinx-intl
make html
```

For that to work, you must have pip installed on the machine used to build the documentation. To install pip on an Ubuntu machine:

```
sudo apt-get install python-pip
```

To build the documentation in Brazilian Portuguese language, run the following extra commands:

```
sphinx-intl -c conf.py build -d locale
make html BUILDDIR=build/html-pt_BR O='-d build/doctrees/ -D language=pt_BR'
```

3.2 Update workflow

To update the documentation, follow the steps below:

1. Update the source files for the english version
2. Extract translatable messages from the english version

```
make gettext
```

3. Update the message catalog (PO Files) for pt_BR language

```
sphinx-intl -c conf.py update -p build/gettext -l pt_BR
```

4. Translate the messages in the pt_BR language PO files

This workflow is based on the Sphinx i18n guide.

CHAPTER 4

How does it work

iotagent-json depends on two things: a Kafka broker, so that it can receive messages informing it about new devices (and, in extension, about their updates and removals), and a MQTT broker, so that it can receive messages from the devices. It waits for messages sent through these two elements: from the device manager with a management operation on a device and from the MQTT broker with a message sent by a device.

CHAPTER 5

How to build

As this is a npm-based project, building it is as simple as

```
npm install
npm run-script build
```

If everything runs fine, the generated code should be in `./build` folder.

CHAPTER 6

How to run

As simple as:

```
npm run-script start ./config.json
```

Remember that you should already have a Kafka node (with a zookeeper instance) and a MQTT broker (such as Eclipse Mosquitto)

6.1 How do I know if it is working properly?

Simply put: you won't. In fact you can implement a simple Kafka publisher to emulate the behaviour of a device manager instance and a listener to check what messages it is generating. But it seems easier to get the real components - they are not that hard to start and to use (given that you use dojot's docker-compose). Check also DeviceManager documentation for further information about how to create a new device.