DeviceManager Documentation

Release 0.2.0

Matheus Magalhaes

Oct 02, 2018

Contents:

The DeviceManager handles all operations related to creation, retrieval, update and deletion of devices in dojot. For more information on that, check DeviceManager Concepts page.

CHAPTER 1

DeviceManager concepts

Here are the main concepts needed to correctly use DeviceManager. They are not hard to understand, but they are essential to operate not only DeviceManager, but the dojot platform as well.

1.1 Device

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 temperature and humidity sensors; it can be represented into 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 associated physical ones, 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.

The information model used for both "real" and virtual devices is as following:

Table 1.1: Device structure

Attribute	Type and mode	Description
id	String (read only)	This is the identifier that will be used when referring to this device.
label	String (read-write, required)	An user label to identify this device more easily
created	DateTime (read-only)	Device creation date
updated	DateTime (read-only)	Device update date
templates	[String (template ID)] (read-write)	List of template IDs to "assemble" this device (more on this on 'Template' section)
attrs	[Attributes] (read-only)	Map of attributes currently set to this device.

The *attrs* attribute is, in fact, a map associating a template ID with an attribute, such as:

```
{
  "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
      }
    ],
    "2": [
      {
        "template_id": "2",
        "created": "2018-01-05T15:47:02.995541+00:00",
        "label": "this-is-another-sample-attribute",
        "value_type": "string",
        "type": "dynamic",
        "id": 4
      }
    ]
  }
}
```


This structure indicates that there are two attributes: one called `this-is-a-sample-attribute` from template ID `1` and another one called `this-is-another-sample-attribute` from template ID `2`.

Table 1.2: Attribute structure

Attribute	Type and mode	Description
id	integer (read-write)	Attribute ID (automatically generated)
label	string (read-write, required)	User label for this attribute
created	DateTime (read-only)	Attribute creation date
updated	DateTime (read-only)	Attribute update date
type	string (read-write, required)	Attribute type (“static”, “dynamic”, “actuator”)
value_type	string (read-write, required)	Attribute value type (“string”, “float”, “integer”, “geo”)
static_value	string (read-write)	If this is a static attribute, which is its static value
template_id	string (read-write)	From which template did this attribute come from.

All attributes that are read/write can be used when creating or updating the device. All of them are returned (if that makes sense - for instance, static_value won’t be returned when no value is set to it) when retrieving device data.

An example of such structure would be:


```
{
  "templates": [
    1,
    2
  ],
  "created": "2018-01-05T17:33:31.605748+00:00",
  "attrs": {
    "1": [
      {
        "template_id": "1",
        "created": "2018-01-05T15:41:54.840116+00:00",
```


(continues on next page)

(continued from previous page)


```
        "label": "temperature",
        "value_type": "float",
        "type": "dynamic",
        "id": 1
      }
      {
        "static_value": "SuperTemplate Rev01",
        "created": "2018-01-05T15:41:54.883507+00:00",
        "label": "model",
        "value_type": "string",
        "type": "static",
        "id": 3,
        "template_id": "1"
      }
    ],
    "2": [
      {
        "static_value": "/admin/efac/attrs",
        "template_id": "2",
        "created": "2018-01-05T15:47:02.995541+00:00",
        "label": "mqtt-topic",
        "value_type": "string",
        "type": "meta",
        "id": 4
      }
    ]
  },
  "id": "b7bd",
  "label": "device"
}
```


1.2 Template

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 themometers 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 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 information model used for templates is:

Table 1.3: Template structure

Attribute	Type and mode	Description
id	string (read-write)	This is the identifier that will be used when referring to this template
label	string (read-write, required)	An user label to identify this template more easily
created	DateTime (read-only)	Template creation date
updated	DateTime (read-only)	Template update date
attrs	[Attributes] (read-write)	List of attributes currently set to this template - it's the same as *attributes* from Device section.

An example of such structure would be:


```
{
  "created": "2018-01-05T15:41:54.803052+00:00",
  "attrs": [
    {
      "template_id": "1",
      "created": "2018-01-05T15:41:54.840116+00:00",
      "label": "temperature",
      "value_type": "float",
      "type": "dynamic",
      "id": 1
    },
    {
      "template_id": "1",
      "created": "2018-01-05T15:41:54.882169+00:00",
      "label": "pressure",
      "value_type": "float",
      "type": "dynamic",
      "id": 2
    },
    {
      "static_value": "SuperTemplate Rev01",
      "created": "2018-01-05T15:41:54.883507+00:00",
      "label": "model",
      "value_type": "string",
      "type": "static",
      "id": 3,
      "template_id": "1"
    }
  ],
  "id": 1,
  "label": "Sample Template"
}
```


All attributes that are read/write can be used when creating or updating the template. All of them are returned (if that makes sense - for instance, static_value won't be returned when no value is set to it) when retrieving device data.

CHAPTER 2

Using DeviceManager

Using DeviceManager is indeed simple: create a template with attributes and then create devices using that template. That's it. This page will show how to do that.

All examples in this page consider that all dojot's components are up and running (check the documentation for how to do that). All request will include a `${JWT}` variable - this was retrieved from auth component.

2.1 Creating templates and devices

Right off the bat, let's retrieve a token from `auth`:

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

```
{
  "jwt": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIU..."
}
```

This token will be stored in `bash ${JWT}` bash variable, referenced in all requests.

Attention: Every request made with this token will be valid only for the tenant (user "service") associated with this token. For instance, listing created devices will return only those devices which were created using this tenant.

A template is, simply put, a model from which devices can be created. They can be merged to build a single device (or a set of devices). It is created by sending a HTTP request to DeviceManager:

```
curl -X POST http://localhost:8000/template \
-H "Authorization: Bearer ${JWT}" \
-H 'Content-Type:application/json' \
-d ' {
  "label": "SuperTemplate",
  "attrs": [
    {
      "label": "temperature",
      "type": "dynamic",
      "value_type": "float"
    },
    {
      "label": "pressure",
      "type": "dynamic",
      "value_type": "float"
    },
    {
      "label": "model",
      "type": "static",
      "value_type" : "string",
      "static_value" : "SuperTemplate Rev01"
    }
  ]
}'
```

Supported `type` values are “dynamic”, “static” and “meta”. Supported `value_types` are “float”, “geo” (for geo-referenced data), “string”, “integer”.

The answer is:

```
{
  "result": "ok",
  "template": {
    "created": "2018-01-05T15:41:54.803052+00:00",
    "attrs": [
      {
        "template_id": "1",
        "created": "2018-01-05T15:41:54.840116+00:00",
        "label": "temperature",
        "value_type": "float",
        "type": "dynamic",
        "id": 1
      },
      {
        "template_id": "1",
        "created": "2018-01-05T15:41:54.882169+00:00",
        "label": "pressure",
        "value_type": "float",
        "type": "dynamic",
        "id": 2
      },
      {
        "static_value": "SuperTemplate Rev01",
        "created": "2018-01-05T15:41:54.883507+00:00",
        "label": "model",
        "value_type": "string",
        "type": "static",
        "id": 3,
```

(continues on next page)

(continued from previous page)

```
        "template_id": "1"
      }
    ],
    "id": 1,
    "label": "SuperTemplate"
  }
}
```

Let's create one more template, so that we can see what happens when two templates are merged.

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

Which results in:

```
{
  "result": "ok",
  "template": {
    "created": "2018-01-05T15:47:02.993965+00:00",
    "attrs": [
      {
        "template_id": "2",
        "created": "2018-01-05T15:47:02.995541+00:00",
        "label": "gps",
        "value_type": "geo",
        "type": "dynamic",
        "id": 4
      }
    ],
    "id": 2,
    "label": "ExtraTemplate"
  }
}
```

Let's check all templates we've created so far.

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

```
{
  "templates": [
    {
      "created": "2018-01-05T15:41:54.803052+00:00",
      "attrs": [
        {
          "template_id": "1",
```

(continues on next page)

(continued from previous page)


```
          "created": "2018-01-05T15:41:54.840116+00:00",
          "label": "temperature",
          "value_type": "float",
          "type": "dynamic",
          "id": 1
        },
        {
          "template_id": "1",
          "created": "2018-01-05T15:41:54.882169+00:00",
          "label": "pressure",
          "value_type": "float",
          "type": "dynamic",
          "id": 2
        },
        {
          "static_value": "SuperTemplate Rev01",
          "created": "2018-01-05T15:41:54.883507+00:00",
          "label": "model",
          "value_type": "string",
          "type": "static",
          "id": 3,
          "template_id": "1"
        }
      ],
      "id": 1,
      "label": "SuperTemplate"
    },
    {
      "created": "2018-01-05T15:47:02.993965+00:00",
      "attrs": [
        {
          "template_id": "2",
          "created": "2018-01-05T15:47:02.995541+00:00",
          "label": "gps",
          "value_type": "geo",
          "type": "dynamic",
          "id": 4
        }
      ],
      "id": 2,
      "label": "ExtraTemplate"
    }
  ],
  "pagination": {
    "has_next": false,
    "next_page": null,
    "total": 1,
    "page": 1
  }
}
```


Now devices can be created using these two templates. Such request would be:

```
curl -X POST http://localhost:8000/device \
-H "Authorization: Bearer ${JWT}" \
-H 'Content-Type:application/json' \
-d ' {
```

(continues on next page)

(continued from previous page)

```
  "templates": [
    "1",
    "2"
  ],
  "label": "device"
}'
```

The result is:


```
{
  "device": {
    "templates": [
      1,
      2
    ],
    "created": "2018-01-05T17:33:31.605748+00:00",
    "attrs": {
      "1": [
        {
          "template_id": "1",
          "created": "2018-01-05T15:41:54.840116+00:00",
          "label": "temperature",
          "value_type": "float",
          "type": "dynamic",
          "id": 1
        },
        {
          "template_id": "1",
          "created": "2018-01-05T15:41:54.882169+00:00",
          "label": "pressure",
          "value_type": "float",
          "type": "dynamic",
          "id": 2
        },
        {
          "static_value": "SuperTemplate Rev01",
          "created": "2018-01-05T15:41:54.883507+00:00",
          "label": "model",
          "value_type": "string",
          "type": "static",
          "id": 3,
          "template_id": "1"
        }
      ],
      "2": [
        {
          "template_id": "2",
          "created": "2018-01-05T15:47:02.995541+00:00",
          "label": "gps",
          "value_type": "geo",
          "type": "dynamic",
          "id": 4
        }
      ]
    },
    "id": "b7bd",
    "label": "device"
```


(continues on next page)

(continued from previous page)

```
  },
  "message": "device created"
}
```

Notice how the resulting device is structured: it has a list of related templates (`template` attribute) and each of its attributes are separated by template ID: `temperature`, `pressure` and `model` are inside attribute `1` (ID of the first created template) and `gps` is inside attribute `2` (ID of the second template). The new device ID can be found in the `id` attribute, which is `b7bd`.

A few considerations must be made:

- If the templates used to compose this new device had attributes with the same name, an error would be generated and the device would not be created.
- If any of the related templates are removed, all its attributes will also be removed from the devices that were created using it. So be careful.

Let's retrieve this new device:

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

This request will list all created devices for the tenant.

```
{
  "pagination": {
    "has_next": false,
    "next_page": null,
    "total": 1,
    "page": 1
  },
  "devices": [
    {
      "templates": [
        1,
        2
      ],
      "created": "2018-01-05T17:33:31.605748+00:00",
      "attrs": {
        "1": [
          {
            "template_id": "1",
            "created": "2018-01-05T15:41:54.840116+00:00",
            "label": "temperature",
            "value_type": "float",
            "type": "dynamic",
            "id": 1
          },
          {
            "template_id": "1",
            "created": "2018-01-05T15:41:54.882169+00:00",
            "label": "pressure",
            "value_type": "float",
            "type": "dynamic",
            "id": 2
          },
          {
            "static_value": "SuperTemplate Rev01",
            "created": "2018-01-05T15:41:54.883507+00:00",
```

(continues on next page)

(continued from previous page)


```
            "label": "model",
            "value_type": "string",
            "type": "static",
            "id": 3,
            "template_id": "1"
          }
        ],
        "2": [
          {
            "template_id": "2",
            "created": "2018-01-05T15:47:02.995541+00:00",
            "label": "gps",
            "value_type": "geo",
            "type": "dynamic",
            "id": 4
          }
        ]
      },
      "id": "b7bd",
      "label": "device"
    }
  ]
}
```


2.2 Removing templates and devices

Removing templates and devices is also very simple. Let's remove the device created previously:

```
curl -X DELETE http://localhost:8000/device/b7bd -H "Authorization: Bearer ${JWT}"
```


```
{
  "removed_device": {
    "templates": [
      1,
      2
    ],
    "created": "2018-01-05T17:33:31.605748+00:00",
    "attrs": {
      "1": [
        {
          "template_id": "1",
          "created": "2018-01-05T15:41:54.840116+00:00",
          "label": "temperature",
          "value_type": "float",
          "type": "dynamic",
          "id": 1
        },
        {
          "template_id": "1",
          "created": "2018-01-05T15:41:54.882169+00:00",
          "label": "pressure",
          "value_type": "float",
          "type": "dynamic",
          "id": 2
```


(continues on next page)

(continued from previous page)


```
        },
        {
          "static_value": "SuperTemplate Rev01",
          "created": "2018-01-05T15:41:54.883507+00:00",
          "label": "model",
          "value_type": "string",
          "type": "static",
          "id": 3,
          "template_id": "1"
        }
      ],
      "2": [
        {
          "template_id": "2",
          "created": "2018-01-05T15:47:02.995541+00:00",
          "label": "gps",
          "value_type": "geo",
          "type": "dynamic",
          "id": 4
        }
      ]
    },
    "id": "b7bd",
    "label": "device"
  },
  "result": "ok"
}
```


Removing templates is also simple:

```
curl -X DELETE http://localhost:8000/template/1 -H "Authorization: Bearer ${JWT}"
```


```
{
  "removed": {
    "created": "2018-01-05T15:41:54.803052+00:00",
    "attrs": [
      {
        "template_id": "1",
        "created": "2018-01-05T15:41:54.840116+00:00",
        "label": "temperature",
        "value_type": "float",
        "type": "dynamic",
        "id": 1
      },
      {
        "template_id": "1",
        "created": "2018-01-05T15:41:54.882169+00:00",
        "label": "pressure",
        "value_type": "float",
        "type": "dynamic",
        "id": 2
      },
      {
        "static_value": "SuperTemplate Rev01",
        "created": "2018-01-05T15:41:54.883507+00:00",
        "label": "model",
```


(continues on next page)

(continued from previous page)

```
        "value_type": "string",
        "type": "static",
        "id": 3,
        "template_id": "1"
      }
    ],
    "id": 1,
    "label": "SuperTemplate"
  },
  "result": "ok"
}
```

These are the very basic operations performed by DeviceManager. All operations can be found in API documentation.

2.3 Sending actuation messages to devices

You can invoke any device actuation via DeviceManager. In order to do so, you have to create some "actuator" attributes in a template. They represent a function exposed by the physical device, such as setting the target temperature, making a step-motor move a bit, resetting the device, etc. Let's create a very similar template from *Creating templates and devices* section and call it a 'Thermostat':

```
curl -X POST http://localhost:8000/template \
-H "Authorization: Bearer ${JWT}" \
-H 'Content-Type:application/json' \
-d ' {
  "label": "Thermostat",
  "attrs": [
    {
      "label": "temperature",
      "type": "dynamic",
      "value_type": "float"
    },
    {
      "label": "pressure",
      "type": "dynamic",
      "value_type": "float"
    },
    {
      "label": "model",
      "type": "static",
      "value_type" : "string",
      "static_value" : "Thermostat Rev01"
    },
    {
      "label": "target_temperature",
      "type": "actuator",
      "value_type": "float"
    }
  ]
}'
```

Note that we have one more attribute - `target_temperature` - to which we will send messages to set the target temperature. This attribute could also have the same name as `temperature` with no side-effects whatsoever. If an actuation request is received by dojot, only `actuator`-type attribute are considered.

This request should give an answer like this:


```
{
  "result": "ok",
  "template": {
    "created": "2018-01-30T12:16:51.423705+00:00",
    "label": "Thermostat",
    "attrs": [
      {
        "template_id": "1",
        "created": "2018-01-30T12:16:51.427113+00:00",
        "label": "temperature",
        "value_type": "float",
        "type": "dynamic",
        "id": 1
      },
      {
        "template_id": "1",
        "created": "2018-01-30T12:16:51.429224+00:00",
        "label": "pressure",
        "value_type": "float",
        "type": "dynamic",
        "id": 2
      },
      {
        "static_value": "Thermostat Rev01",
        "created": "2018-01-30T12:16:51.430194+00:00",
        "label": "model",
        "value_type": "string",
        "type": "static",
        "id": 3,
        "template_id": "1"
      },
      {
        "template_id": "1",
        "created": "2018-01-30T12:16:51.430870+00:00",
        "label": "target_temperature",
        "value_type": "float",
        "type": "actuator",
        "id": 4
      }
    ],
    "id": 1
  }
}
```


Creating a device based on it is no different than before:

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

This gives back the following data:

```
{
  "message": "devices created",
  "devices": [
    {
      "id": "356d",
      "label": "device"
    }
  ]
}
```

To send a configuration message to the device, you should send a request like this:

```
curl -X PUT http://localhost:8000/device/356d/actuate \
-H "Authorization: Bearer ${JWT}" \
-H 'Content-Type:application/json' \
-d ' {
    "attrs": {
        "target_temperature" : 10.6
    }
}'
```

The request payload contains only the following attribute:

- attrs: All the attributes and their respective values that will be configured on the device. Each value can be as simple as a float or a string, or it could hold a more complex structure, such as an object.

Remember that the attribute must be an actuator for this request to succeed. If not, a message like the following one is returned:

```
{
  "status": "some of the attributes are not configurable",
  "attrs": [
    "pressure"
  ]
}
```

The request will be published via Kafka. All elements that are interested in device notifications (such as IoT agents), will received it. What should be done with it is up to the component that processes this message. Check the documentation of each component (in particular, from IoT agents) to check what is done with it.

CHAPTER 3

REST API

All APIs are available in Github pages API description, which is automatically generated from `this file`.

CHAPTER 4

Internal messages

There are some messages that are published by DeviceManager through Kafka. These messages are notifications of device management operations, and they can be consumed by any component interested in them, such as IoT agents.

Table 4.1: Kafka messages

Event	Service	Message type
Device creation	dojot.device-manager.device	*Creation message*
Device update	dojot.device-manager.device	*Update message*
Device removal	dojot.device-manager.device	*Removal message*
Device actuation	dojot.device-manager.device	*Actuation message*

4.1 Creation message

This message is published whenever a new device is created. Its payload is a simple JSON:

```
{
  "event": "create",
  "meta": {
    "service": "admin"
  },
  "data": {
    "id": "efac",
    "label" : "Device 1",
    "templates" : [ 1, 2, 3],
    "attrs" : {

    },
    "created" : "2018-02-06T10:43:40.890330+00:00"
  }
}
```

And its attributes are:

- *event* (string): “create”
- *meta*: Meta information about the message
  - *service* (string): Tenant associated to this device
- *data*: device data structure
  - *id* (string): Device ID
  - *attrs*: Device attributes. This field is as described in *DeviceManager concepts*

4.2 Update message

This message is published whenever a new device is updated. Its payload looks very similar to device creation:

```
{
  "event": "update",
  "meta": {
    "service": "admin"
  },
  "data": {
    "id": "efac",
    "label" : "Device 1",
    "templates" : [ 1, 2, 3],
    "attrs" : {

    },
    "created" : "2018-02-06T10:43:40.890330+00:00"
  }
}
```

- *event* (string): “update”
- *meta*: Meta information about the message
  - *service* (string): Tenant associated to this device
- *data*: device new data structure
  - *id* (string): ID of the device being updated
  - *attrs*: Device attributes. This field is as described in *DeviceManager concepts*

4.3 Removal message

This message is published whenever a device is removed. Its payload is:

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

- *event* (string): “remove”
- *meta*: Meta information about the message
  - *service* (string): Tenant associated to this device
- *data*: device data
  - *id* (string): ID of the device being removed

4.4 Actuation message

This message is published whenever a device must be configured. The payload is:

```
{
  "event": "actuate",
  "meta": {
    "service": "admin"
  },
  "data" : {
    "id" : "efac",
    "attrs": {
      "reset" : 1,
      "step-motor" : "+45"
    }
  }
}
```

- *event* (string): “actuate”
- *meta*: Meta information about the message
  - *service* (string): Tenant associated to this device

This message should be forwarded to the device. It can contain more attributes than the ones specified by DeviceManager. For instance, a thermostat could be configured with the following message:

```
{
  "event": "actuate",
  "meta": {
    "service": "admin"
  },
  "data" : {
    "id" : "efac",
    "attrs": {
      "target_temperature" : 23.5
    }
  }
}
```

The attribute actually used by the device would be “target_temperature” so that it can adjust correctly the temperature. It’s up to the receiver of this message (an IoT agent, for instance) to properly send the configuration to the device.

CHAPTER 5

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.

5.1 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
export READTHEDOCS_VERSION=latest
make html
```

The `READTHEDOCS_VERSION` environment variable should be set to the component version being built, such as `latest` or `0.2.0`. In the automated build process from readthedocs, this exact variable will be set as the name of the branch/tag being built.

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'
```

5.2 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 6

Dependencies

DeviceManager has the following dependencies:

- flask (including flask_sqlalchemy)
- psycopg2
- marshmallow
- requests
- gunicorn
- gevent
- json-logging-py
- kakfa-python

But you won't need to worry about installing any of these - they are automatically installed when starting DeviceManager. There must be, though, a postgres instance accessible by DeviceManager.

CHAPTER 7

How to run

If you really need to run DeviceManager as a standalone process (without dojot's wonderful docker-compose), you can execute these commands:

```
python setup.py develop
gunicorn device-manager.app:app
```

Keep in mind that running a standalone instance of DeviceManager misses a lot of security checks (such as user identity checks, proper multi-tenancy validations, and so on). In particular, every request sent to DeviceManager needs an access token, which should be retrived from auth component.

CHAPTER 8

How to use it

There are a few concepts that must be understood to properly use DeviceManager. Visit DeviceManager Concepts page to check them out.

This component listens to HTTP requests at port 5000 - all its endpoints are documented in the API page.

IMPORTANT: If you are using all dojot's components (for instance, using a deploy based on docker-compose), it is recommended to visit dojot documentation to check the endpoints for all services (including DeviceManager's)**