CLOUD OF THINGS
REST API GUIDE

Version 1.7.2
Date 13.04.2018

LIFE IS FOR SHARING.

TABLE OF CONTENT

1. INTRODUCTION TO REST

This is a guideline of the REST and SMARTREST protocols. It will explain how to integrate a device into Cloud of Things and how to develop an application for Cloud of Things.

Note: If case of questions please contact cloudofthings@telekom.de.

1.1. OVERVIEW

Cloud of Things employs REST for all external communication. Regardless whether the communication originates from IoT devices, from web applications or from back office IT systems – the communication protocol is always REST.
REST is a very simple and secure protocol based on HTTP(S) and TCP. It is today the de-facto Internet standard supported by all networked programming environments ranging from very simple devices up to large-scale IT. One of the many books introducing REST is RESTful Web Services.

This guide explains how to use Cloud of Things' REST interfaces to:

- Interface devices with Cloud of Things
- Develop applications on top of Cloud of Things
- Integrate other cloud services or IT backend applications with Cloud of Things.

It first shows you how to use the REST interfaces in general, then discusses device integration and finally it describes application development. The description is closely linked to the reference guide, which describes each interface in detail. Relevant chapters in the reference guide are in particular:

- **REST implementation** is the reference for all general concepts
- **Device management library** specifies the data model for device management
- **Sensor library** specifies the data model for sensors and controls.

If you develop using Java ME/SE, JavaScript or C/C++, please check the relevant developer's guides for even more convenient access to Cloud of Things' functionality. Also, if you use any of the supported development boards, see the corresponding "Devices" section for more information.

1.2. USING THE REST INTERFACES

Most programming environments today have particular support for REST-based communication. For experimentation and for understanding REST interfaces, it is helpful to use one of the numerous available command line tools or browser extensions.

For example, many operating systems come pre-installed with the "curl" command. If you want to start browsing the APIs, enter on a command line:

```
$ curl -u <username>/<password> https://<yourURL>.ram.m2m.telekom.com/platform
```

Replace "username" and "password" with the username and password that you used to register to Cloud of Things. Similarly, replace "yourURL" with the URL you used at registration time.

The command will return links to all basic interfaces of Cloud of Things:

```
...
"inventory": {
    "managedObjects": {
        "references": [],
        "self": "https://<yourURL>/inventory/managedObjects"
    },
    "managedObjectsForFragmentType":
"https://<yourURL>/inventory/managedObjects?fragmentType={fragmentType}",
    "managedObjectsForListOfIds":
"https://<yourURL>/inventory/managedObjects?ids={ids}",
    "managedObjectsForType":
"https://<yourURL>/inventory/managedObjects?type={type}",
    "self": "https://<yourURL>/inventory"
},
...
```

To format the output more nicely on a Mac, try "curl ... | python -mjson.tool".

From this point, you can navigate further. For example, display the items in the inventory by following the "managedObjects" link:

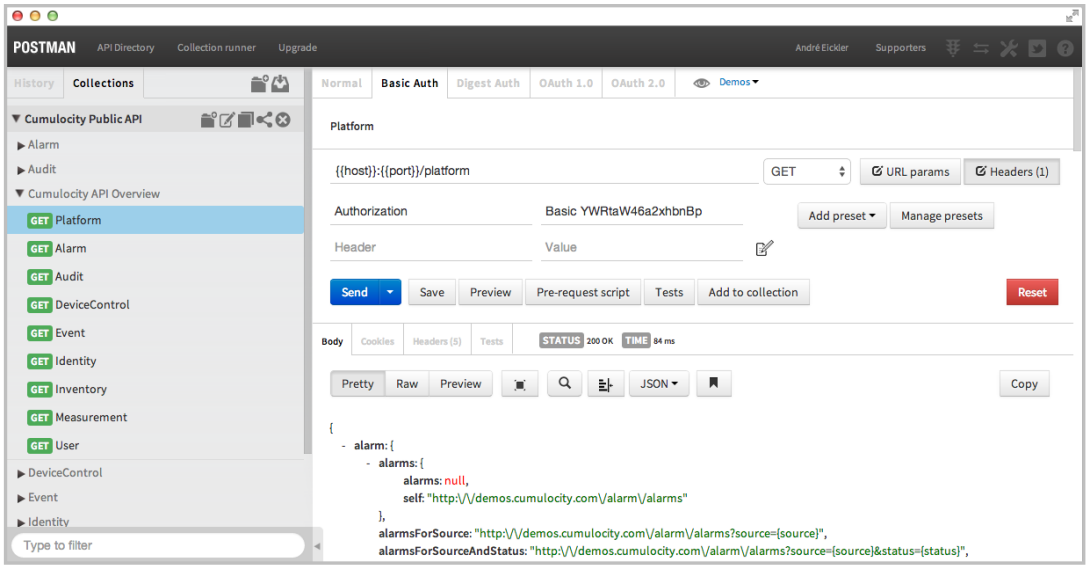
```
$ curl -u <username>/<password> https://<yourURL>.ram.m2m.telekom.
com/inventory/managedObjects
```

You will notice that just a subset of the items in the inventory is actually returned, a so-called "page".

1.3. USING POSTMAN

A convenient way to explore REST interface and the Cloud of Things database content are browser extensions such as Postman or Advanced REST Client for Chrome. If you want to make use of them, download (Link) and install Postman. After starting Postman, you can choose to either create an account or click "Take me straight to the app". Then click the button below and choose the variant of Postman that you have just installed. You may see a browser security prompt asking you whether you actually want to run Postman (on Windows "Electron").

Now, click the "Collections" tab on the top left of Postman. You should see a folder "Cumulocity API" with the examples. Open that folder and the sub-folder "Alarms", then click on "Get collection of alarms".



Here is a shortcut to set up Postman for Cloud of Things:

- Download the Cloud of Things Postman collection and click "**Import collection**" in Postman (or get it from the Postman API directory).
- Click on drop-down menu next to the little "eye" widget to configure your Cloud of Things URL. Click "**Manage environments**" and "**Add**". Then type a name for your tenant and configure a key "**url**" with a value of "https://<yoururl>.ram.m2m.telekom.com". Click "**Submit**".
- Now, you can run REST calls. Click, for example, on "**Cloud of Things API**", "**Cloud of Things API Overview**", "**GET Platform**". By clicking the "**Send**" button, you can send the GET request to Cloud of Things. The first time that you send a request to Cloud of Things, you have to enter your credentials. Click on "**Basic Auth**" and enter your username and password, followed by a click on "**Refresh Headers**".
- To explore the API, click on the links in the responses. Similar to navigate through pages of results, click on the "**next**" link at the bottom of the response. Add, for example, "**?pageSize=100**" to the end of the request URL to get more data than the default five items.

Note: Postman has two issues: it always sends a content type even if you do not specify one. If you see an error, please add the "**Content-Type**" header described in the reference manual. It also sometimes shows "Malformed JSON" as a response, which is a bug in Postman.

2. HELLO REST!

2.1. OVERVIEW

This section gives a very basic example how to create a device representation in Cloud of Things and subsequently how to send related measurement data. All steps are performed by calling REST interfaces. Those REST calls are demonstrated by CURL statements that can be executed on command line. Please have a look on the previous section for a short introduction to CURL.

2.2. PREREQUISITES

In order to follow this tutorial, check if the following prerequisites are fulfilled:

- You have a valid tenant, user and password to access Cloud of Things
- The command line tool CURL is installed on your system.

2.3. DO THE REST CALLS

We will now perform a sequence of just two REST calls, which are described in detail next:

- Step 1: Create a new device in the inventory of Cloud of Things
- Step 2: Transmit measurement data related to that device

In real world those steps are performed by the 'device agent'. Step one is performed just once, when the device is connected to Cloud of Things for the first time. After that, actions related to that device can be performed by referencing the device by an internal ID which is returned when executing this step.

2.3.1. CREATE A NEW DEVICE

To create a new device in the inventory of Cloud of Things the following REST request is needed:

```
POST /inventory/managedObjects HTTP/1.1
Content-Type: application/vnd.com.nsn.cumulocity.managedObject+json;
charset=UTF-8; ver=0.9
Accept: application/vnd.com.nsn.cumulocity.managedObject+json; charset=UTF-
8; ver=0.9
Authorization: Basic <<Base64 encoded credentials <tenant
ID>/<username>:<password> >>
...
{
    "c8y_IsDevice" : {},
    "name" : "HelloWorldDevice"
}
```

This call can be done by executing the following curl statement:

```
curl -v -u <username>:<password> \
   -H 'Accept: application/vnd.com.nsn.cumulocity.managedObject+json;
charset=UTF-8; ver=0.9' \
   -H 'Content-type: application/vnd.com.nsn.cumulocity.managedObject+json;
charset=UTF-8; ver=0.9' \
   -X POST \
```

```
   -d '{"c8y_IsDevice":{},"name":"HelloWorldDevice"}' \
   https://<tenant-ID>.ram.m2m.telekom.com/inventory/managedObjects
```

Please replace <username>, <password> and <tenant-ID> with the appropriate credentials given to you when registering with Cloud of Things. The same credentials used to access the Cloud of Things Web GUI can be used to execute the REST calls.

You will receive a response like that:

```
HTTP/1.1 201 Created
Content-Type: application/vnd.com.nsn.cumulocity.managedObject+json;
charset=UTF-8; ver=0.9
Authorization: Basic <<Base64 encoded credentials <tenant
ID>/<username>:<password> >>
...
{
    "id": "1231234"
    "lastUpdated": "2014-12-15T14:58:26.279+01:00",
    "name": "HelloWorldDevice",
    "owner": "<username>",
    "self": "https://<tenant-
ID>.ram.m2m.telekom.com/inventory/managedObjects/1231234",
    "c8y_IsDevice": {},
    ...
}
```

When creating a device, Cloud of Things generates an ID, which is needed in further calls in order to reference the device. We can find this ID as the "id" attribute-value pair in the response.

2.3.2. TRANSMIT MEASUREMENT DATA

Now the device is created, we can send measurement data. In our case, we will send a temperature measurement in the unit of Celsius which was collected on a certain time:

```
POST /measurement/measurements
Content-Type: application/vnd.com.nsn.cumulocity.measurement+json;
charset=UTF-8; ver=0.9
Accept: application/vnd.com.nsn.cumulocity.measurement+json; charset=UTF-8;
ver=0.9
...
{
    "c8y_TemperatureMeasurement": {
        "T": {
            "value": 21.23,
            "unit":"C"
        }
    },
    "time": "2014-12-15T13:00:00.123+02:00",
    "source": {
        "id": "1231234"
    },
    "type":"c8y_PTCMeasurement"
}
```

Please replace the id value with the appropriate value you received in the first step.

Furthermore, you should update the time value to a recent timestamp in order to make it easy to find back the measurement on Cloud of Things UI later.

```
curl -v -u <username>:<password> \
   -H 'Accept: application/vnd.com.nsn.cumulocity.measurement+json;
charset=UTF-8; ver=0.9' \
   -H 'Content-type: application/vnd.com.nsn.cumulocity.measurement+json;
charset=UTF-8; ver=0.9' \
   -X POST \
   -d
'{"c8y_TemperatureMeasurement":{"T":{"value":21.23,"unit":"C"}},"time":"201
4-12-
15T13:00:00.123+02:00","source":{"id":"1231234"},"type":"c8y_PTCMeasurement
"}' \
   https://<tenant-ID>.ram.m2m.telekom.com/measurement/measurements/
```

The response to that request will look like this:

```
HTTP/1.1 201 Created
Content-Type: application/vnd.com.nsn.cumulocity.measurement+json;
charset=UTF-8; ver=0.9
...
{
    "id": "4711",
    "self": "https://<tenant-
ID>.ram.m2m.telekom.com/measurement/measurements/4711",
    "source": {
        "id": "1231234",
        "self": "https://<tenant-
ID>.ram.m2m.telekom.com/inventory/managedObjects/1231234"
    },
    "time": "2014-12-15T12:00:00.123+01:00",
    "type": "c8y_PTCMeasurement",
    "c8y_TemperatureMeasurement": {
        "T" : {
            "unit" : "C",
            "value" : 21.23
        }
    }
}
```

If you like to, you can repeat sending measurements. Before sending the request again, you should update the timestamp (value of attribute 'time') in order to create a time series.

Now you are done. Enter Cloud of Things Web GUI, select your device on the "**All devices**" tab and move further to the "**Measurements**" tab. Here you can see your measurement data. If not, change the filter setting to e.g. "last week" to include the timestamp you used in your submitted measurement.

2.4. GO FURTHER

The sequence of REST calls demonstrated here is just a shortened procedure of Device Integration. The first step ('create a new device' and 'register device') is part of the 'startup phase', whereas step two ('sending measurements') references to the 'cycle phase'.

Please go further to the **Device Integration** section to get the necessary information required for implementing real-world agents.

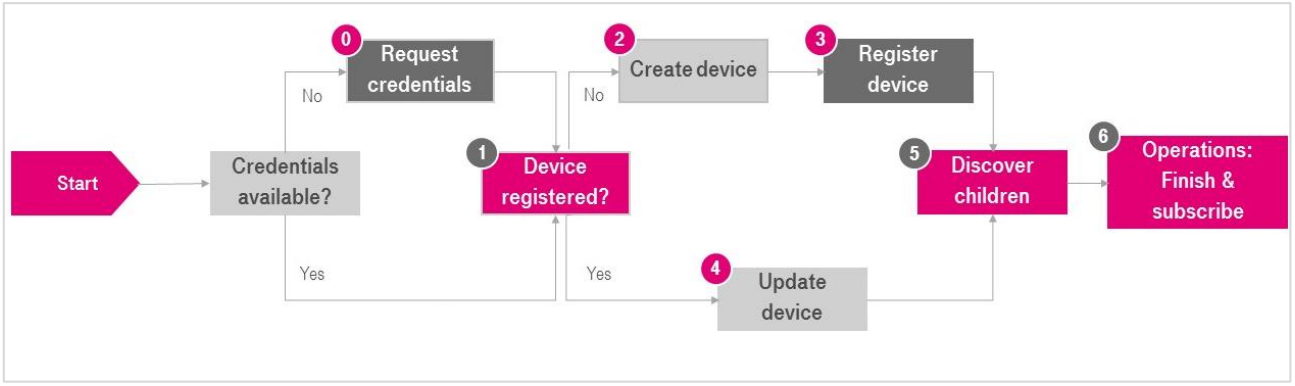
3. DEVICE INTEGRATION

3.1. OVERVIEW

The basic life cycle for integrating devices into Cloud of Things is discussed in Interfacing devices. In this section, we will show how this life cycle is implemented on REST level. The life cycle consists of two phases, a startup phase and a cycle phase.

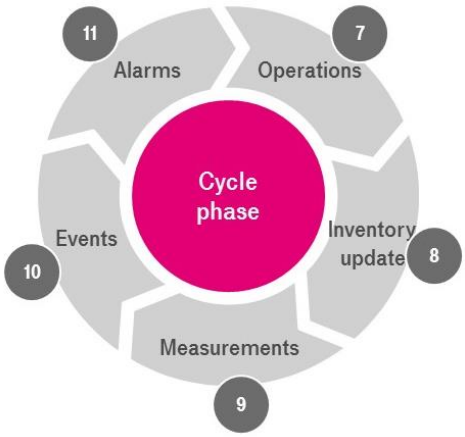
The startup phase is responsible for connecting the device to Cloud of Things and updating the device data in the inventory. It also performs cleanup tasks required for operations. It consists of the following steps:

- Step 0: Request device credentials, if they have not been requested yet.
- Step 1: Check if the device is already registered.
- Step 2: If no, create the device in the inventory and
- Step 3: Register the device.
- Step 4: If yes, update the device in the inventory.
- Step 5: Discover child devices and create or update them in the inventory.
- Step 6: Finish operations that required a restart and subscribe to new operations.



The cycle phase follows. It continuously updates the inventory, writes measurements, alarms and events and executes operations when required. It can be considered to be the "main loop" of the device which is executed until the device shuts down. The loop consists of the following steps:

- Step 7: Execute operations.
- Step 8: Update inventory.
- Step 9: Send measurements.
- Step 10: Send events.
- Step 11: Send alarms.



3.2. STARTUP PHASE

3.2.1. STEP 0: REQUEST DEVICE CREDENTIALS

Since every request to Cloud of Things needs to be authenticated, also requests from devices need to be authenticated. If you want to assign individual credentials to devices, you can use the device credentials API to generate new credentials automatically. To do so, request device credentials at first startup through the API and store them locally on the device for further requests.

The process works as follows:

- Cloud of Things assumes each device to have some form of unique ID. A good device identifier may be the MAC address of the network adapter, the IMEI of a mobile device or a hardware serial number
- When you take a new device into use, you enter this unique ID into "Device registration" in Cloud of Things and start the device.
- The device will connect to Cloud of Things and send its unique ID repeatedly. For this purpose, Cloud of Things provides a static host that can be enquired from registration.cloud-of-things@telekom.de.
- You can accept the connection from the device in "**Device registration**", in which case Cloud of Things sends generated credentials to the device.
- The device will use these credentials for all further requests.

From device perspective, this is a single REST request:

```
POST /devicecontrol/deviceCredentials
Content-Type:
application/vnd.com.nsn.cumulocity.deviceCredentials+json;ver=...
Authorization: Basic ...
{
  "id" : "0000000017b769d5"
}
```

The device issues this request repeatedly. While the user has not yet registered and accepted the device, the request returns "404 Not Found." After the device is accepted, the following response is returned:

```
HTTP/1.1 201 OK
Content-Type:
application/vnd.com.nsn.cumulocity.deviceCredentials+json;ver=...
Content-Length: ...
```

```
{
  "id" : "0000000017b769d5",
  "self" : "<<URL of new request>>",
  "tenantId" : "test",
  "username" : "device_0000000017b769d5",
  "password" : "3rasfst4swfa"
}
```

The device can now connect to Cloud of Things using the tenant ID, username and password.

Note: If the device does not receive credentials, it will not move to step 1 according to the above-mentioned process diagram. For further information please contact m2m-hardware-support@telekom.de.

3.2.2. STEP 1: CHECK IF THE DEVICE IS ALREADY REGISTERED

The unique ID of the device is also used for registering the device in the inventory. The registration is carried out using the Identity API. In the Identity API, each managed object can be associated with multiple identifiers distinguished by type. Types are, for example, "c8y_Serial" for a hardware serial, "c8y_MAC" for a MAC address and "c8y_IMEI" for an IMEI.

To check if a device is already registered, use a GET request on the identity API using the device identifier and its type. The following example shows a check for a Raspberry Pi with hardware serial "0000000017b769d5".

```
GET /identity/externalIds/c8y_Serial/raspi-0000000017b769d5 HTTP/1.1

HTTP/1.1 200 OK
Content-Type: application/vnd.com.nsn.cumulocity.externalId+json;
charset=UTF-8; ver=0.9
...
{
    "externalId": "raspi-0000000017b769d5",
    "managedObject": {
        "id": "2480300",
        "self": "https://.../managedObjects/2480300"
    },
    "self": "https://.../identity/externalIds/c8y_Serial/raspi-
0000000017b769d5",
    "type": "c8y_Serial"
}
```

Note that while MAC addresses are guaranteed to be globally unique, serial numbers for hardware may overlap across different hardwares. Hence, in the above example, we prefixed the serial number with a "raspi-".

In this case, the device is already registered and a status code of 200 is returned. In the response, a URL to the device in the inventory is returned in "managedObject.self". This URL can be used to work with the device lateron.

If a device is not yet registered, a 404 status code and an error message is returned:

```
GET /identity/externalIds/c8y_Serial/raspi-0000000017b769d6 HTTP/1.1

HTTP/1.1 404 Not Found
Content-Type: application/vnd.com.nsn.cumulocity.error+json;charset=UTF-
8;ver=0.9
...
{
    "error": "identity/Not Found",
```

```
    "info": "https://www.cumulocity.com/guides/reference-
guide/#error_reporting",
    "message": "External id not found; external id = ID [type=c8y_Serial,
value=raspi-0000000017b769d6]"
}
```

3.2.3. STEP 2: CREATE THE DEVICE IN THE INVENTORY

If Step 1 above indicated that no managed object representing the device exists, create the managed object in Cloud of Things. The managed object describes the device, both its instance and metadata. Instance data includes hardware and software information, serial numbers, and device configuration data. Metadata describes the capabilities of the devices, including the supported operations.

To create a managed object, issue a POST request on the managed objects collection in the Inventory API. The following example creates a Raspberry Pi using the Linux agent:

```
POST /inventory/managedObjects HTTP/1.1
Content-Type: application/vnd.com.nsn.cumulocity.managedObject+json
Accept: application/vnd.com.nsn.cumulocity.managedObject+json
...
{
    "name": "RaspPi BCM2708 0000000017b769d5",
    "type": "c8y_Linux",
    "c8y_IsDevice": {},
    "com_cumulocity_model_Agent": {},
    "c8y_SupportedOperations": [ "c8y_Restart", "c8y_Configuration",
"c8y_Software", "c8y_Firmware" ],
    "c8y_Hardware": {
        "revision": "000e",
        "model": "RaspPi BCM2708",
        "serialNumber": "0000000017b769d5"
    },
    "c8y_Configuration": {
        "config": "#Fri Aug 30 09:13:56 BST
2013\nc8y.log.eventLevel=INFO\n..."
    },
    "c8y_Mobile": {
         "imei": "861145013087177",
        "cellId": "4904-A496",
        "iccid": "89490200000876620613"
    },
    "c8y_Firmware": {
        "name": "raspberrypi-bootloader",
        "version": "1.20130207-1"
    },
    "c8y_Software": {
        "pi-driver": "pi-driver-3.4.5.jar",
        "pi4j-gpio-extension": "pi4j-gpio-extension-0.0.5.jar",
        ...
    }
}

HTTP/1.1 201 Created
Content-Type:
application/vnd.com.nsn.cumulocity.managedObject+json;charset=UTF-8;ver=0.9
...
{
    "id": "2480300",
    "lastUpdated": "2013-08-30T10:12:24.378+02:00",
```

```
    "name": "RaspPi BCM2708 0000000017b769d5",
    "owner": "admin",
    "self": "https://.../inventory/managedObjects/2480300",
    "type": "c8y_Linux",
    "c8y_IsDevice": {},
    ...
    "assetParents": {
        "references": [],
        "self": "https://.../inventory/managedObjects/2480300/assetParents"
    },
    "childAssets": {
        "references": [],
        "self": "https://.../inventory/managedObjects/2480300/childAssets"
    },
    "childDevices": {
        "references": [],
        "self": "https://.../inventory/managedObjects/2480300/childDevices"
    },
    "deviceParents": {
        "references": [],
        "self":
"https://.../inventory/managedObjects/2480300/deviceParents"
    }
}
```

The example above contains a number of metadata items for the device:

- "c8y_IsDevice" marks devices that can be managed using Cloud of Things' Device Management.
- "com_cumulocity_model_Agent" marks devices running a Cloud of Things agent. Such devices will receive all operations targeted to themselves and their children for routing.
- "c8y_SupportedOperations" states that this device can be restarted and configured. In addition, it can carry out software and firmware updated.

If the device could be successfully created, a status code of 201 is returned. If the original request contains an "**Accept**" header as in the example, the complete created object is returned including the ID and URL to reference the object in future requests. The returned object also includes references to collections of child devices and child assets that can be used to add children to the device (see below).

3.2.4. STEP 3: REGISTER THE DEVICE

After the new device has been created, it can now be associated with its built-in identifier as described in Step 1. This ensures that the device can find itself in Cloud of Things after the next power-up.

Continuing the above example, we would associate the newly created device "2480300" with its hardware serial number:

```
POST /identity/globalIds/2480300/externalIds HTTP/1.1
Content-Type: application/vnd.com.nsn.cumulocity.externalId+json
Accept: application/vnd.com.nsn.cumulocity.externalId+json
...
{
    "type" : "c8y_Serial",
    "externalId" : "raspi-0000000017b769d5"
}

HTTP/1.1 201 Created
```

```
Content-Type:
application/vnd.com.nsn.cumulocity.externalId+json;charset=UTF-8;ver=0.9
...
{
    "externalId": "raspi-0000000017b769d5",
    "managedObject": {
        "id": "2480300",
        "self": "https://.../inventory/managedObjects/2480300"
    },
    "self": "https://.../identity/externalIds/c8y_Serial/raspi-
0000000017b769d5",
    "type": "c8y_Serial"
}
```

3.2.5. STEP 4: UPDATE THE DEVICE IN THE INVENTORY

If Step 1 above returned that the device was previously registered already, we need to make sure that the inventory representation of the device is up to date with respect to the current state of the actual device. For this purpose, a PUT request is sent to the URL of the device in the inventory. Note that only fragments that can actually change need to be transmitted. (See Cloud of Things' domain model for more information on fragments.)

For example, the hardware information of a device will usually not change, but the software installation may change. So it may make sense to bring the software information in the inventory up to the latest state after a reboot of the device:

```
PUT /inventory/managedObjects/2480300 HTTP/1.1
Content-Type: application/vnd.com.nsn.cumulocity.managedObject+json
...
{
    "c8y_Software": {
        "pi-driver": "pi-driver-3.4.6.jar",
        "pi4j-gpio-extension": "pi4j-gpio-extension-0.0.5.jar"
    }
}

HTTP/1.1 200 OK
```

Do not update the name of a device from an agent!

An agent creates a default name for a device so that it can be identified in the inventory, but users should be able to edit this name or update it with information from their asset management.

3.2.6. STEP 5: DISCOVER CHILD DEVICES AND CREATE OR UPDATE THEM IN THE INVENTORY

Depending on the complexity of the sensor network, devices may have child devices associated with them. A good example is home automation: You often have a home automation gateway that installs a multitude of different sensors and controls installed in various rooms of the household. The basic registration of child devices is similar to the registration of the main device up to the fact, that child devices usually do not run an agent instance. To link a device with a child, send a POST request to the child devices URL that was returned when creating the object (see above).

For example, assume a child device with the URL "**https://.../inventory/managedObjects/2543801**" has already been created. To link this device with its parent, issue:

```
POST /inventory/managedObjects/2480300/childDevices HTTP/1.1
```

```
Content-Type:
application/vnd.com.nsn.cumulocity.managedObjectReference+json
{ "managedObject" : { "self" :
"https://.../inventory/managedObjects/2543801" } }

HTTP/1.1 201 Created
```

Finally, devices and references can be deleted by issuing a DELETE request to their URLs. For example, the reference from the parent device to the child device that we just created can be removed by issuing:

```
DELETE /inventory/managedObjects/2480300/childDevices/2543801 HTTP/1.1

HTTP/1.1 204 No Content
```

This does not delete the device itself in the inventory, only the reference. To delete the device, issue:

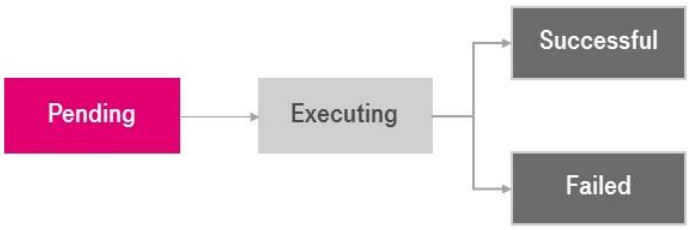
```
DELETE /inventory/managedObjects/2543801 HTTP/1.1

HTTP/1.1 204 No Content
```

This request will also delete all data associated with the device including its registration information, measurements, alarms, events and operations. Usually, it is not recommended to delete devices automatically. For example, if a device has just temporarily lost its connection, you usually do not want to lose all historical information associated with the device.

3.3. WORKING WITH OPERATIONS

Each operation in Cloud of Things is cycled through an execution flow. When an operation is created through a Cloud of Things application, its state is "PENDING", i.e., it has been queued for executing but it hasn't executed yet. When an agent picks up the operation and starts executing it, it marks the operations as "EXECUTING" in Cloud of Things. The agent will then carry out the operation on the device or its children (for examples, it will restart the device, or set a relay). Then it will possibly update the inventory reflecting the new state of the device or its children (e.g., it updates the current state of the relay in the inventory). Then the agent will mark the operation in Cloud of Things as either "SUCCESSFUL" or "FAILED", potentially indicating the error.



The benefit of this execution flow is that it supports devices that are offline and temporarily out of coverage. It also allows devices to support operations that require a restart – such as a firmware upgrade. After the restart, the device needs to know what it previously did and hence needs to query all "EXECUTING" operations and see if they were successful. Also, it needs to listen what new operations may be queued for it.

3.3.1. STEP 6: FINISH OPERATIONS AND SUBSCRIBE

To clean up operations that are still in "EXECUTING" status, query operations by agent ID and status. In our example, the request would be:

```
GET /devicecontrol/operations?agentId=2480300&status=EXECUTING HTTP/1.1

HTTP/1.1 200 OK
Content-Type: application/vnd.com.nsn.cumulocity.operationCollection+json;;
charset=UTF-8; ver=0.9
...
{
    "next":
"https://.../devicecontrol/operations?agentId=2480300&status=EXECUTING",
    "operations": [
        {
            "creationTime": "2013-08-29T19:49:15.239+02:00",
            "deviceId": "2480300",
            "id": "2593101",
            "self": "https://.../devicecontrol/operations/2480300",
            "status": "EXECUTING",
            "c8y_Restart": {
            }
        }
    ],
    "statistics": {
        "currentPage": 1,
        "pageSize": 2000
    },
    "self":
"https://.../devicecontrol/operations?agentId=2480300&status=EXECUTING"
}
```

The restart seems to have executed well -- we are back after all. So let's set the operation to "SUCCESSFUL".

```
PUT /devicecontrol/operations/2480300 HTTP/1.1
Content-Type: application/vnd.com.nsn.cumulocity.operation+json
{
    "status": "SUCCESSFUL"
}

HTTP/1.1 200 OK
```

Then, listen to new operations created in Cloud of Things. The mechanism for listening to real-time data in Cloud of Things is described in Real-time notifications and is based on the standard Bayeux protocol. First, a handshake is required. The handshake tells Cloud of Things what protocols the agent supports for notifications and allocates a client ID to the agent.

```
POST /devicecontrol/notifications HTTP/1.1
Content-Type: application/json
...
[ {
    "id": "1",
    "supportedConnectionTypes": ["long-polling"],
    "channel": "/meta/handshake",
    "version": "1.0"
} ]

HTTP/1.1 200 OK
...
[ {
    "id": "1",
    "supportedConnectionTypes": ["websocket","long-polling"],
    "channel": "/meta/handshake",
    "version": "1.0",
```

```
    "clientId": "139jhm07u1dlry92fdl63rmq2c",
    "minimumVersion": "1.0",
    "successful": true
}]
```

Afterwards, the device respectively the agent needs to subscribe to notifications for operations. This is done using a POST request with the ID of the device as subscription channel. In our example, the Raspberry Pi runs an agent and has ID 2480300:

```
POST /devicecontrol/notifications HTTP/1.1
Content-Type: application/json
...
[ {
    "id": "2",
    "channel": "/meta/subscribe",
    "subscription": "/2480300",
    "clientId":"139jhm07u1dlry92fdl63rmq2c"
}]

HTTP/1.1 200 OK
...
[ {
    "id":"2",
    "channel": "/meta/subscribe",
    "subscription": "/2480300",
    "successful": true,
} ]
```

Finally, the device connects and waits for operations to be sent to it.

```
POST /devicecontrol/notifications HTTP/1.1
Content-Type: application/json
...
[ {
    "id": "3",
    "connectionType": "long-polling",
    "channel": "/meta/connect",
    "clientId": "139jhm07u1dlry92fdl63rmq2c"
} ]
```

This request will hang until an operation is issued, i.e. the HTTP server will not answer immediately, but wait until an operation is available for the device (long polling).

Note that there might have been operations that were pending before we subscribed to new incoming operations. We need to query these still. This is done after the subscription to not miss any operations between query and subscription. The technical handling is just like previously described for "EXECUTING" operations, but using "PENDING" instead:

```
GET /devicecontrol/operations?agentId=2480300&status=PENDING HTTP/1.1
```

3.4. CYCLE PHASE

3.4.1. STEP 7: EXECUTE OPERATIONS

Assume now that an operation is queued for the agent. This will make the long polling request that we issued above return with the operation. Here is an example of a response with a single configuration operation:

```
HTTP/1.1 200 OK
...
[
    {
        "id": "139",
        "data": {
            "creationTime":"2013-09-04T10:53:35.128+02:00",
            "deviceId": "2480300",
            "id": "2546600",
            "self": "https://.../devicecontrol/operations/2546600",
            "status": "PENDING",
            "description": "Configuration update",
            "c8y_Configuration": { "config": "#Wed Sep 04 10:54:06 CEST
2013\n..." }
        },
        "channel": "/2480300"
    }, {
        "id": "3",
        "successful": true,
        "channel": "/meta/connect"
    }
]
```

When the agent picks up the operation, it sets it to "EXECUTING" state in Cloud of Things using a PUT request (see above example for "FAILED"). It carries out the operation on the device and runs possible updates of the Cloud of Things inventory. Finally, it sets the operation to "SUCCESSFUL" or "FAILED" depending on the outcome. Then, it will reconnect again to "/devicecontrol/notifications" as described above and wait for the next operation.

The device should reconnect within ten seconds to the server to not loose queued operations. This is the time that Cloud of Things buffers real-time data. The interval can be specified upon handshake.

3.4.2. STEP 8: UPDATE INVENTORY

The inventory entry of a device usually represents its current state, which may be subject of continuous change. As an example, consider a device with a GPS chip. That device will keep its current location up-to-date in the inventory. At the same time, it will report location updates as well as event to maintain a trace of its locations. Technically, such updates are reported with the same requests as shown in Step 4.

3.4.3. STEP 9: SEND MEASUREMENTS

To create new measurements in Cloud of Things, issue a POST request with the measurement. The example below shows how to create a signal strength measurement.

```
POST /measurement/measurements HTTP/1.1
Content-Type: application/vnd.com.nsn.cumulocity.measurement+json
...
{
    "source": { "id": "2480300" },
    "time": "2013-07-02T16:32:30.152+02:00",
    "type": "huawei_E3131SignalStrength",
```

```
    "c8y_SignalStrength": {
        "rssi": { "value": -53, "unit": "dBm" },
        "ber": { "value": 0.14, "unit": "%" }
    }
}

HTTP/1.1 201 Created
```

3.4.4. STEP 10: SEND EVENTS

Similar, use a POST request for events. The following example shows a location update from a GPS sensor.

```
POST /event/events HTTP/1.1
Content-Type: application/vnd.com.nsn.cumulocity.event+json
...
{
    "source": { "id": "1197500" },
    "text": "Location updated",
    "time": "2013-07-19T09:07:22.598+02:00",
    "type": "c8y_LocationUpdate",
    "c8y_Position": {
        "alt": 73.9,
        "lng": 6.151782,
        "lat": 51.211971
    }
}

HTTP/1.1 201 Created
```

Note that all data types in Cloud of Things can include arbitrary extensions in the form of additional fragments. In this case, the event includes a position, but also self-defined fragments can be added.

3.4.5. STEP 11: SEND ALARMS

Alarms represents events that most likely require human intervention to be solved. For example, if the battery in a device runs out of energy, someone has to visit the device to replace the battery. Creating an alarm is technically very similar to creating an event.

```
POST /alarm/alarms HTTP/1.1
Content-Type: application/vnd.com.nsn.cumulocity.alarm+json
Accept: application/vnd.com.nsn.cumulocity.alarm+json
...
{
    "source": { "id": "10400" },
    "text": "Tracker lost power",
    "time": "2013-08-19T21:31:22.740+02:00",
    "type": "c8y_PowerAlarm",
    "status": "ACTIVE",
    "severity": "MAJOR",
}

HTTP/1.1 201 Created
Content-Type: application/vnd.com.nsn.cumulocity.alarm+json
...
{
    "id": "214600",
    "self": "https://.../alarm/alarms/214600",
```

```
    ...
}
```

However, you most likely should not create an alarm for a device, if there is a similar alarm already active in the system. Creating many alarms may flood the user interface and may require users to manually clear all the alarms. This is an example for finding the active alarms of our Raspberry Pi from above:

```
GET /alarm/alarms?source=2480300&status=ACTIVE HTTP/1.1
```

In contrast to events, alarms can be updated. If an issue is resolved (e.g., the battery was replace, power was restored), the corresponding alarm should be automatically cleared to save manual work. This can be done through a PUT request to the URL of the alarm. In the above example for creating an alarm, we used an "Accept" header to get the URL of the new alarm in the response. We can use this URL to clear the alarm:

```
PUT /alarm/alarms/214600 HTTP/1.1
Content-Type: application/vnd.com.nsn.cumulocity.alarm+json
...
{
    "status": "CLEARED"
}

HTTP/1.1 200 OK
```

If you are uncertain on whether to send an event or raise an alarm, you can simply just raise an event and let the user decide with a CEL rule if they want to convert the event into an alarm.

4. APPLICATION DEVELOPMENT

4.1. OVERVIEW

In this section, we are touching some of the basic use cases in using the Cloud of Things REST APIs for application development. Typically, you need to:

- Register assets
- Link devices to assets
- Synchronize assets with external systems
- Query particular capabilities
- Query readings from sensors
- Send operations to devices
- Listen for events.

4.2. REGISTER ASSETS

Assets are the objects that your business and your application focuses on. For example, assets might be buildings and rooms if your business centers around building management or home automation. Or they might be routes and machines, if your business is about servicing machines.

Assets are stored in the inventory along with the devices, but they often have an own structure independent of devices. You create assets by POSTing them to the collection of managed objects in the inventory. For example, to create a new room in the inventory:

```
POST /inventory/managedObjects HTTP/1.1
Content-Type: application/vnd.com.nsn.cumulocity.managedObject+json
Accept: application/vnd.com.nsn.cumulocity.managedObject+json
...
{
    "name": "Building 043",
    "type": "c8y_Building"
}

HTTP/1.1 201 Created
Content-Type:
application/vnd.com.nsn.cumulocity.managedObject+json;charset=UTF-8;ver=0.9
...
{
    "owner": "admin",
    "id": "2549800",
    "self": "https://.../inventory/managedObjects/2549800",
    "type": "c8y_Building",
    "lastUpdated": "2013-09-05T16:38:31.250+02:00",
    "name": "Building 043",
    "assetParents": {
        "references": [],
        "self": "https://.../inventory/managedObjects/2549800/assetParents"
    },
    "childAssets": {
        "references": [],
        "self": "https://.../inventory/managedObjects/2549800/childAssets"
    },
    "childDevices": {
        "references": [],
        "self": "https://.../inventory/managedObjects/2549800/childDevices"
    },
    "deviceParents": {
        "references": [],
        "self":
"https://.../inventory/managedObjects/2549800/deviceParents"
    }
}
```

If the device could be successfully created, a status code of 201 is returned. If the original request contains an "Accept" header just like in the example, the complete created object is returned including the ID and URL to reference the object in future requests. The returned object also includes references to collections of child devices and child assets that can be used to add children to the device.

For example, assuming that we have also created a room, and that room's "self" property is "https://.../inventory/managedObjects/2549700". To link the room to the building, POST to the child assets collection of the building (see the "self" property of "childAssets" above):

```
POST /inventory/managedObjects/2549800/childAssets HTTP/1.1
Content-Type:
application/vnd.com.nsn.cumulocity.managedObjectReference+json

{ "managedObject" : { "self" :
"https://.../inventory/managedObjects/2549700" } }

HTTP/1.1 201 Created
```

Now querying the building again shows that the room has been registered as child of the building:

```
GET /inventory/managedObjects/2549800 HTTP/1.1

HTTP/1.1 200 OK
Content-Type: application/vnd.com.nsn.cumulocity.managedObject+json;
charset=UTF-8; ver=0.9
...
{
    "owner": "admin",
    "id": "2549800",
    "self": "https://.../inventory/managedObjects/2549800",
    ...
    "childAssets": {
        "references": [
            {
                "managedObject": {
                    "id": "2549700",
                    "name": "Room 042",
                    "self": "https://.../inventory/managedObjects/2549700"
                },
                "self":
"https://.../inventory/managedObjects/2549800/childAssets/2549700"
            }
        ],
        "self": "https://.../inventory/managedObjects/2549800/childAssets"
    }
}
```

4.3. LINK DEVICES TO ASSETS

Just like you link assets to other child assets, you can link assets also to devices that monitor and control the asset. For example, assume that you have a light sensor installed in the room, and that light sensor has the URL "**https://.../inventory/managedObjects/2480500**". POST to the "childDevices" of the room as follows:

```
POST /inventory/managedObjects/2549700/childDevices HTTP/1.1
Content-Type:
application/vnd.com.nsn.cumulocity.managedObjectReference+json

{ "managedObject" : { "self" :
"https://.../inventory/managedObjects/2480500" } }

HTTP/1.1 201 Created
```

4.4. SYNCHRONIZE ASSETS WITH EXTERNAL SYSTEMS

Often, Cloud of Things will not be the only IT system dealing with a company's asset. The technical procedure for synchronizing assets stored in external IT systems is exactly the same as the procedure used for registering devices:

- Use the Identity API to link the asset ID of the external IT system to the asset ID of Cloud of Things
- Use the Inventory API to create or update the assets in Cloud of Things' inventory based on the external system's data.

4.5. QUERY PARTICULAR CAPABILITIES

To decouple applications from the specifics of particular types and makes of devices, applications can use so-called fragments to query the inventory. For example, to find all managed objects having a location, use

```
GET /inventory/managedObjects?fragmentType=c8y_Position&withTotalPages=true
HTTP/1.1

HTTP/1.1 200 OK
Content-Type:
application/vnd.com.nsn.cumulocity.managedObjectCollection+json;
charset=UTF-8; ver=0.9
...
{
    "managedObjects": [
        {
            "id": "2480700",
            "lastUpdated": "2013-08-30T10:15:44.218+02:00",
            "name": "RaspPi BCM2708 0000000017b769d5 Gps eM9",
            "owner": "admin",
            "self": "https://.../inventory/managedObjects/2480700",
            "type": "c8y_TinkerForge_Gps",
            "c8y_Position": {
                "alt": 102.36,
                "lng": 6.769717,
                "lat": 51.267259
            },
            ...
        },
        ...
    ]
    "next":
"https://.../inventory/managedObjects?withTotalPages=true&fragmentType=c8y_
Position&pageSize=5&currentPage=2",
    "statistics": {
        "currentPage": 1,
        "pageSize": 5,
        "totalPages": 4
    },
    "self":
"https://.../inventory/managedObjects?withTotalPages=true&fragmentType=c8y_
Position&pageSize=5&currentPage=1"
}
```

Now, you can, for example, place the object in a map.

Querying the "/platform" resource will show you further possibilities for querying your data (see the Introduction).

Note that queries do not necessarily return all query results at once, but only a "page" of the result. For more information on paging, see the Section Query result paging. The optional parameter "withTotalPages" will make the query contain full page statistics at the expensive of slightly slower performance.

4.6. QUERY READINGS FROM SENSORS

Similar to the inventory, you can also query for particular sensor readings. For example, let's query the light measurements of the past month (from the time of writing this text):

```
GET /measurement/measurements?dateFrom=2013-08-05&dateTo=2013-09-
05&fragmentType=c8y_LightMeasurement HTTP/1.1

HTTP/1.1 200 OK
```

```
Content-Type:
application/vnd.com.nsn.cumulocity.measurementCollection+json; charset=UTF-
8; ver=0.9
...
{
    "measurements": [
        {
            "id": "2480900",
            "self": "https://.../measurement/measurements/2480900",
            "source": {
                "id": "2480500",
                "self": "https://.../inventory/managedObjects/2480500"
            },
            "time": "2013-08-29T21:19:52.321+02:00",
            "type": "c8y_LightMeasurement",
            "c8y_LightMeasurement": {
                "e": { "unit": "lux", "value": 169.2 }
            }
        },
        ...
    ]
    ...
}
```

4.7. SEND OPERATIONS TO DEVICES

To trigger an operation on a device, POST the operation to the Device Control API. The following example restarts the device with the ID "2480300" (which is the Raspberry Pi that we previously integrated):

```
POST /devicecontrol/operations HTTP/1.1
Content-Type: application/vnd.com.nsn.cumulocity.operation+json;
Accept: application/vnd.com.nsn.cumulocity.operation+json;
...
{
"deviceId": "2480300",
"c8y_Restart":{}
}

HTTP/1.1 201 Created
Content-Type: application/vnd.com.nsn.cumulocity.operation+json;
charset=UTF-8; ver=0.9
...
{
"creationTime": "2013-09-05T19:18:16.117+02:00",
"deviceId": "2480300",
"id":"2550200",
"self": "https://.../devicecontrol/operations/2550200",
"status": "PENDING",
"c8y_Restart": {}
}
```

The POST command returns immediately when the operation has been queued for the device. The actual operation executes asynchronously. Since we added the optional "Accept" header in the example request, we will get the full queued operation in the response including its URL in the "self" property. Using a GET on that URL, you can check the current state of execution of the operation:

```
GET /devicecontrol/operations/2550200 HTTP/1.1
```

```
HTTP/1.1 200 OK
Content-Type: application/vnd.com.nsn.cumulocity.operation+json;
charset=UTF-8; ver=0.9
...
{
    "status": "PENDING",
    ...
}
```

A state of "PENDING" means here that the device has not yet picked up the operation. "EXECUTING" means that the device is in the process of executing the operation. Finally, "SUCCESSFUL" or "FAILED" indicate that the operation is completed.

4.8. LISTEN FOR EVENTS

Besides querying the Cloud of Things data store, you can also process and receive events in real-time as described in Real-time processing in Cloud of Things. For example, assume that you would like to display real-time location updates in a map. Use the administration user interface (or the REST API) to create a new rule module "myRule":

```
select *
from EventCreated e
where e.event.type = "c8y_LocationUpdate";
```

If you have a device that sends location updates, you should see them immediately in the user interface. To receive them in your own REST client, you use the Notification API to subscribe to them. The API is based on the Bayeux protocol. First, a handshake is required. The handshake tells Cloud of Things what protocols the client supports for notifications and allocates a client ID to the client.

```
POST /cep/notifications HTTP/1.1
Content-Type: application/json
...
[ {
    "id": "1",
    "supportedConnectionTypes": ["long-polling"],
    "channel": "/meta/handshake",
    "version": "1.0"
} ]
```

```
HTTP/1.1 200 OK
...
[ {
    "id": "1",
    "supportedConnectionTypes": ["websocket","long-polling"],
    "channel": "/meta/handshake",
    "version": "1.0",
    "clientId": "71fjkmy0495rxrkfcmp0mhcev1",
    "minimumVersion": "1.0",
    "successful": true
}]
```

After the handshake, the client needs to subscribe to the output of the above rule. This is done using a POST request with the module name and the statement name as subscription channel. In our example, we used the module name "myRule" and did not give a name to the "select" statement ("@Name(")"), so the subscription channel is "/myRule/*".

```
POST /cep/notifications HTTP/1.1
Content-Type: application/json
...
```

```
[ {
    "id": "2",
    "channel": "/meta/subscribe",
    "subscription": "/myRule/*",
    "clientId":"71fjkmy0495rxrkfcmp0mhcev1"
}]

HTTP/1.1 200 OK
...
[ {
    "id":"2",
    "channel": "/meta/subscribe",
    "subscription": "/myRule/*",
    "successful": true,
} ]
```

Finally, the client connects and waits for events to be sent to it.

```
POST /cep/notifications HTTP/1.1
Content-Type: application/json
...
[ {
    "id": "3",
    "connectionType": "long-polling",
    "channel": "/meta/connect",
    "clientId": "71fjkmy0495rxrkfcmp0mhcev1"
} ]
```

This request will hang until an operation is issued. Here is an example of a response with a single location update:

```
HTTP/1.1 200 OK
...
[
    {
        "id": "139",
        "data": {
            "creationTime": "2013-08-30T09:38:45.551+02:00",
            "id": "2481400",
            "self": "https://.../event/events/2481400",
            "source": {
                "id": "2480700",
                "name": "RaspPi BCM2708 0000000017b769d5 Gps eM9",
                "self": "https://.../inventory/managedObjects/2480700"
            },
            "text": "Location updated",
            "time": "2013-08-29T21:20:01.671+02:00",
            "type": "c8y_LocationUpdate",
            "c8y_Position": {
                "alt": 58.34,
                "lng": 6.769717,
                "lat": 51.267259
            },
            "channel": "/myRule/*"
    }, {
        "id": "3",
        "successful": true,
        "channel": "/meta/connect"
    }
]
```

5. USING SMARTREST

5.1. OVERVIEW

The Cloud of Things REST APIs provide you with a generic Internet of Things (IoT) protocol that is simple to use from most environments. It can be ad-hoc adapted to any IoT use case and uses standard Internet communication and security mechanisms. While this is a great leap forward over tailored IoT protocols with proprietary technologies, it poses some challenges to very constrained environments such as low-end microcontrollers or low-bandwidth communication channels.

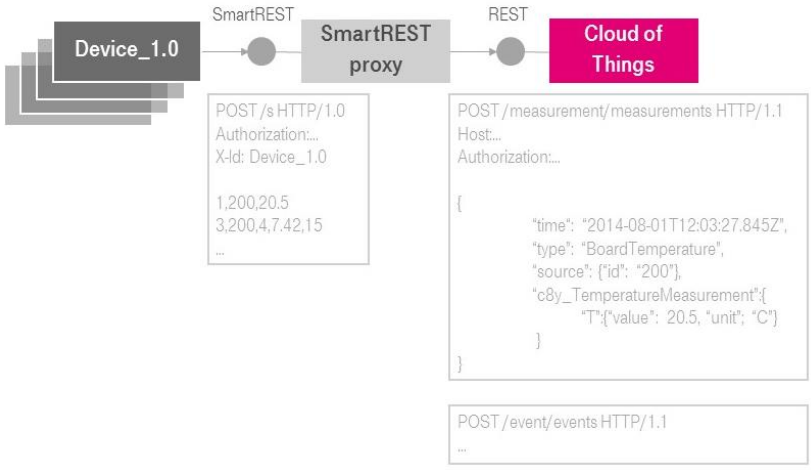
For these environments, Cloud of Things offers the so-called "SmartREST" protocol. SmartREST combines the benefits of standard technology and tailored protocols:

- It continues to work on any network by using standard HTTP technology
- It supports HTTP authentication and encryption
- It still gracefully handles protocol versioning
- Its network traffic usage is close to custom-optimized protocols by transferring pure payload data during normal operation
- It is based on CSV (comma separated values) and hence is easy to handle from C-based environments
- It supports server-generated timestamps for devices without clocks

In the next section, we will discuss the concepts behind SmartREST and the basic protocol that is used. SmartREST is based on separating metadata from payload data by using so-called templates, which are then described. Finally, we show how to send and receive data using SmartREST. For a detailed description of the protocol, see the SmartREST reference.

5.2. HOW DOES SMARTREST WORK?

The image below illustrates how SmartREST works. Devices and other clients connect to a dedicated SmartREST endpoint on Cloud of Things and send their data in rows of comma-separated values. These rows are expanded by Cloud of Things' SmartREST proxy into standard Cloud of Things REST API requests. Similar, responses from Cloud of Things are compressed by the proxy from their original JSON format into comma-separated values before sending them back to the device.

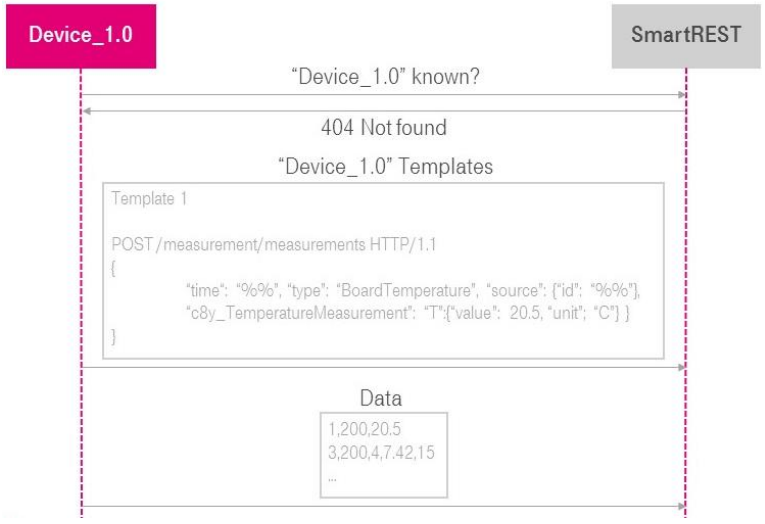


How can Cloud of Things interpret comma-separated values into meaningful REST requests? For that purpose, devices register templates with Cloud of Things. The templates contain the expanded REST requests together with placeholders into

which the Cloud of Things SmartREST proxy consecutively inserts the comma-separate values. For responses, the templates describe which values to pick from the structured REST response to construct comma-separated values.

Templates are associated with software or firmware versions of a device. Usually, a particular implementation of a device or application can only issue a particular set of well-defined types of requests. All devices with the same implementation share the same set of request types. Hence, the templates can be defined at implementation time. To make the templates available to Cloud of Things, the first device with a particular implementation will send its templates and makes them available for usage by all similar devices.

This process is illustrated below. Assume a device with an implementation version "Device_1.0" starts communicating through SmartREST. After retrieving its credentials, the device will ask the SmartREST proxy if its template is already known. If the template is not found on the server, the device will send its template in a single static text request to Cloud of Things. Once this procedure has been carried out, all similar devices using that template can start communicating using SmartREST without re-sending the template to the server.



The example also roughly illustrates the translation process. In "Template 1", "%%" is a placeholder to be filled by the SmartREST proxy. "time" is filled with a server-side timestamp (see below). The remaining placeholders are filled with request data. The line "1,200,20.5" in the example request is interpreted as follows:

- The first column references the template to be used, in this case Template 1.
- "200" refers to the first free placeholder in the template, in this case the ID in the "source" element (the ID of the device that sends the measurement.)
- "20.5" refers to the second free placeholder in the template, here the value of the temperature measurement.

5.3. THE BASIC SMARTREST PROTOCOL

The basic structure of all SmartREST requests is as follows:

- All requests are POST requests to the endpoint "/s", regardless of what the requests finally translate to.
- The standard HTTP "Authorization" header is used to authenticate the client.
- An additional "X-Id:" header is used to identify the implementation of the client, either as device type (such as "Device_1.0") or as an identifier returned by the template registration process.
- A request body contains rows of text in comma-separated value format. Each row corresponds to one request to the standard Cloud of Things REST API.
- The response is always "200 OK".

- The response body again contains rows of comma-separated values. A row corresponds to a response from the Cloud of Things REST API on a particular request.

Using the above example, a SmartREST request would be as follows:

```
POST /s HTTP/1.1
Authorization: Basic ...
X-Id: Device_1.0

1,200,20.5
```

And the corresponding response would be:

```
HTTP/1.1 200 OK
Content-Length: 6

20,0
```

To match the requests and responses, a response line contains, next to the error code, the line of the request that the response answers. In this example, "20" indicates "OK" and "0" refers to the first line of the request.

5.4. HOW ARE TEMPLATES REGISTERED?

As described above, a client using SmartREST will first ask if its SmartREST templates are already known to the server. This is done with an empty SmartREST request:

```
POST /s HTTP/1.1
Authorization: Basic ...
X-Id: Device_1.0
```

If the device implementation is known, the response will return an ID that can be used as "shorthand" in the "X-Id" header of later requests.

```
HTTP/1.1 200 OK

20,<id>
```

If the device implementation is unknown, the response will be:

```
HTTP/1.1 200 OK

40,"No template for this X-ID"
```

In this case, create all templates used in your device implementation.

```
POST /s HTTP/1.1
Authorization: Basic ...
X-Id: Device_1.0

10,1,POST,/measurement/measurements,application/vnd.com.nsn.cumulocity.meas
urement+json,,%%,NOW UNSIGNED NUMBER,{ "time": "%%", "type": ... }
...
```

In this example, "10" refers to a request template (whereas "11" would refer to a response template). The template is number "1", so SmartREST requests using this template have a "1" in their first column. The template refers to a "POST" request to the endpoint "/measurement/measurements" with a content type of "**application/vnd.com.nsn.cumulocity.measurement+json**". The placeholder used in the template is "%%". The placeholders are a time stamp ("NOW"), an unsigned number and a general number. Finally, the last column contains the body of the request to be filled in a sent.

5.5. HOW ARE RESPONSES HANDLED?

The above example illustrated the handling of requests and request templates. For responses, JSONPath expressions translate Cloud of Things REST responses into CSV. Assume, for example, a device has a display and can show a message on the display. An operation to update the message would look like this:

```
{
    "c8y_Message": {
        "text": "Hello, world!"
    },
    "creationTime": "2014-02-25T08:32:45.435+01:00",
    "deviceId": "8789602",
    "status": "PENDING",
    ...
}
```

On the client side, the device mainly needs to know the text to be shown. In JSONPath, the "text" property is extracted using the following syntax:

```
$.c8y_Message.text
```

In this syntax, "$" refers to the root of the data structure and "." selects an element from a data structure. For more options, please consult the JSONPath reference.

A device usually queries for all operations that are associated with it and that are in pending state. The standard Cloud of Things response to such a query is:

```
{
    "operations": [
        {
            "c8y_Message": {
                "text": "Hello, world!"
            },
            "creationTime": "2014-02-25T08:32:45.435+01:00",
            "deviceId": "8789602",
            "status": "PENDING",
            ...
        }, {
            "c8y_Relay": {
                ...
            }
            ...
    ]
]
```

That is, the response contains a list of operations, and these operations can have different types. To work with such a structure, use the following response template:

```
11,2,$.operations,$.c8y_Message,$.c8y_Message.text
```

This means, value by value:

- 11: This is a response template.
- 2: It has Number 2.
- $.operations: The response is a list and the list's property is "operations".
- $.c8y_Message: This template applies to responses with the property "c8y_Message".
- $.c8y_Message.text: The text will be extracted from the message and will be returned.

The SmartREST client will thus get the following response:

```
HTTP/1.1 200 OK

2,0,"Hello, world!"
```

That is, the response was created using Template 2, the template to translate display message operations. The response refer to the first request sent. The actual message to set is "Hello, world!".