English

Asema IoT Central

JSON API 1.0

ASEMA

Table of Contents

1. Introduction

Asema IoT Central is an event driven, property-oriented, object management system. The JSON API offers methods to plug externally into that system and drive the events and properties from various other systems. So to give some more insight into how the API works, let's first explain what the first definition means in practice:

- **Event driven**. Things move through the system core as events and their signals. Whenever something changes, moves, is stored, is measured, and so forth, an event and a corresponding signal is created. Logical components of the system capture those signals and then act according to their specific role in the system. For instance if a new measurement value is received, an event for measurement change is sent to the system. An example of a system component that hears this signal is the database driver. When it gets the signal it stores the value into the database.

- **Object management**. In Asema IoT Central, all "things", such as a lamp you may have configured as something to control, is an object. Each object has a set of capabilities and properties that are used and manipulated by the system. The objects are also sources and sinks of events. So if, say, the lamp changes its color, the property of color in that lamp will change and an event is created. If the lamp has a capability of changing a real physical object, the driver for the capability will hear the changing property and will do the actual change in the physical object. Further, some other object may be listening to the events of the first object. A logic rule object may for instance listen to the event, process it and create another event that changes the color of some other lamp.

- **Property oriented**. Changes in the system and the corresponding physical world are modeled as properties of objects. So if you would want to change the color of a lamp, you'd change the property `color` of the lamp. If you'd like to change the lamp on or off, you'd change the `actuator_state` property of the object. The names of the properties are defined in the schemas of the objects and those schemas can be queried through the API to figure out what each object can actually do. Some common property changes also have shortcuts in the API, such as `turn_object` which will turn the actuator state to ON or OFF depending on the parameter given or `toggle_object` that will change the actuator state to the opposite of what it was.

Because Asema IoT Central is primarily designed for monitoring and controlling some physical objects which are something, do something and are somewhere, most of the core tasks are very similar for all the objects. Consequently, there is sort of a basic "toolkit" of functions that help in performing most the work programmers need to address in applications. This includes the following methods

- `get_object_meta`. Tells you what the object is. For instance its name, type classification, etc.

- `get_object_capabilities`. Tells you what the object can do. Can it measure and control something for instance.

- `get_object_position`. Tells you where the object is (as coordinates).

- `get_object_velocity`. Tells you where the object is going to (if it moves).

- `get_object_status`. Tells you whether the object is properly connected, powered and online.

- `get_object_property_names`. Tells you what properties the object has.

The rest of automation then in the majority of cases involves changing those properties and/or monitoring their values. For this task there are two very straighforward methods: `get_object_property` and `set_object_property`.

When you want to know the property values and their changes, Asema IoT Central offers you functions for getting values both as push and pull. So you can either poll the value by getting the property at (ir)regular intervals or subscribe to it and the system will send you the changes as they occur.

2. Using the JSON API

2.1. Calling the API

2.1.1. Calling with HTTP POST

To call the JSON API using HTTP POST, create a POST message and send it to the path `/json` of the server and port where Asema IoT Central runs. For instance, if it is at port 8080 on IP 192.168.1.1, the URL to call would be http://192.168.1.1:8080/json.

The calls in Asema IoT Central POST handler use the JSON RPC 2.0 standard. You can create the body of the call either automatically with a JSON-RPC client or just manually. JSON RPC is so simple that manual handling of the payloads is usually sufficient. As specified in the standard, payload format is a dictionary with id, method and params sections. Whhen you make the call, params is always a dictionary (named arguments), **except** with the notable exception of the subscription methods where it is a list. E.g. to call the method `get_object_property`, the payload would be

```
{
 "id": 1,
 "method": "get_object_property",
 "params": {
  "gid": "the_gid_of_the_object_to_address",
  "property": "some_property_to_fetch"
 }
}
```

Likewise, the call with a list parameters, in this case `add_to_subscription` would be

```
{
 "id": 2,
 "method": "add_to_subscription",
 "params": [10, "abcd"]
}
```

In the example above, 10 is assumed to be the stream number and "abcd" a GID of some object (replace these with actual values in your code).

2.1.2. Calling with HTTP GET

You can also use HTTP GET to call the API, although using POST is recommended as plugging in the parameters to the GET URL is somewhat cumbersome in long calls and GET always has the security flaw of leaving traces of the call parameters into various webserver logs.

If you do use HTTP GET, form a URL using as the base the path `/json` of the server and port where Asema IoT Central runs. For instance, if it is at port 8080 on IP 192.168.1.1, the URL to call would start with http://192.168.1.1:8080/json.

The parameters of the call should then be placed into the query paramters of the URL. E.g. to call the method `get_object_property`, the URL would be for example http://192.168.1.1:8080/json?method=get_object_property&gid=the_gid_of_the_object_to_address&property=some_property_to_fetch.

2.1.3. JSON API call example in Python

Below is an example Python script that demonstrates the use of the JSON API. It requests a couple of objects from Asema IoT Central (a TV and a thermometer) and then toggles the TV state and asks for

the temperature from the thermometer. Note that for this code to work, those objects naturally need to be in the system. Modify the code accordingly to test in your local installation.

Requests to the server are made using the `make_json_post_request` function which is a generic HTTP POST function that creates a JSON format message, handles sequential request numbering, adds the required headers, and parses the response from the server. Understanding the workings of this method basically covers everything needed to do the call.

```
#!/usr/bin/python

from simplejson import loads, dumps
import urllib, urllib2

SERVER_IP = "127.0.0.1"
SERVER_PORT = 8080

requestId = 0


def make_json_post_request(url, command, params):
    global requestId

    requestId += 1
    payload = { 'id': requestId, 'method': command, 'params': params }
    result = None
    headers = { 'content-type' : 'application/json' }
    try:
        req = urllib2.Request(url, dumps(payload), headers)

        # use a 5s timeout
        filehandle = urllib2.urlopen(req, timeout = 5)
        if filehandle is not None:
            data = filehandle.read()
            result = loads(data)
    except:
        print "Failed in contacting", url
    finally:
        return result


def findByType(url, type):
    response = make_json_post_request(url, 'find_objects_by_type', { "type": type })
    return response['result']['objects']

def findByName(url, name):
    response = make_json_post_request(url, 'find_objects_by_name', { "name": name })
    return response['result']['objects']

def getObject(url, gid):
    response = make_json_post_request(url, 'get_object',  { "gid": gid }])
    return response['result']['object_data']

def toggleByGid(url, gid):
    make_json_post_request(url, 'toggle_object',  { "gid": gid })

def getSensorValue(url, gid, property):
    response = make_json_post_request(url, 'get_property_value', { "gid": gid, "property": property })
    return response['result']

def main():
    url = "http://" + SERVER_IP + ":" + str(SERVER_PORT) + "/json"

    objs = findByType(url, "Appliance");
    print "Appliances:", objs
    print
    objs = findByName(url, "TV");
    tv = objs[0]
    print "TVs (sought by name):", objs
    print

    objs = findByName(url, "Thermometer");
    thermo = objs[0]
    print "Thermometers (sought by name):", objs
    print

    toggleByGid(url, tv['gid']);
    print "Toggled TV"
    print
    v = getSensorValue(url, thermo['gid'], "temperature_celsius")
    print "Current temperature:", v

if __name__ == '__main__':
    main()
```

3. Object API methods

3.1. Property value subscription

Subscription methods let you obtain a handle to and HTTP Push stream of value updates for properties. A basic subscription with `subscribe` simply opens the stream, it does not yet send data of any particular object. Once you have the stream handle, you add the objects into the stream by calling `add_to_subscription`. Once done, whenever some property of any of the objects associated with the stream changes, a notification will be pushed through the stream.

> **Important**
> Note that HTTP Push is one of three possible methods for receiving proprty notifications, the two others being MQTT and WebSockets. The `subscribe` method described here only applies to HTTP Push. For other methods, please refer to corresponding manuals.

> **Important**
> Note that unlike all other methods of the JSON API, the subscribe methods take their parameters in the form of a list. This is for backwards compatibility with other Asema software and devices. So when you call a subscription method, instead of creating a dictionary with parameter names and values, create a list instead and just put the parameter values into the list in correct order (the same order as listed in the function documentation below).

`subscribe`(*`uri`*);

```
string uri mandatory;
```

Subscribes to a stream of notifications. The return value is an integer of that is the number of the stream. Add this number to add objects into the stream and to unsubscribe from the stream. The sole argument is the URI of the recipient. The system will call this exact URI with an HTTP POST to send the notification.

`add_to_subscription`(*`streamId, gid`*);

```
int streamId mandatory;
string gid mandatory;
```

Adds an object to the stream. After addition, the property changes of that particular object will be sent as notifications. Note that you can repeat the method multiple times to add more objects.

`unsubscribe`(*`streamId`*);

```
int streamId mandatory;
```

Unsubscribe from the stream i.e. make the system stop sending notifications to it.

3.2. Retrieving objects and their properties

`get_all_objects`

Returns a list of dictionaries representing the data of all objects in the system.

`get_object`(*`gid`*);

```
string gid mandatory;
```

Returns a dictionary representing the data of the object represented by the GID.

get_object_capabilities(*gid*);

```
string gid mandatory;
```

Returns a dictionary representing the capabilities (such as control or measurement) of the object represented by the GID.

get_object_meta(*gid*);

```
string gid mandatory;
```

Returns a dictionary of metadata of the particular object.

set_object_property(*gid*, *property*, *value*);

```
string gid mandatory;
string property mandatory;
variant value mandatory;
```

Sets the latest recorded value of a particular property of a given object. Note that if the property has a physical meaning (as is linked and specified in the object settings), the system will try to also set this physical property.

get_object_property(*gid*, *property*);

```
string gid mandatory;
string property mandatory;
```

Returns the latest recorded value of a particular property of a given object.

get_object_property_history(*gid*, *property*, *start*, *end*, *interval*);

```
string gid mandatory;
string property mandatory;
string[yyyy-MM-ddTHH:mm:ss] start mandatory;
string[yyyy-MM-ddTHH:mm:ss] end mandatory;
string interval optional;
```

Returns a timeseries (list of time-value pairs) of a given property of the object identified by the GID from start time to end time, both ends inclusive.

get_object_property_timeseries(*gid*, *property*, *start*, *end*, *interval*);

```
string gid mandatory;
string property mandatory;
string[yyyy-MM-ddTHH:mm:ss] start mandatory;
string[yyyy-MM-ddTHH:mm:ss] end mandatory;
string interval optional;
```

A synonym (equal call) to `get_object_property_history`.

add_object_property(*gid*, *property*);

```
string gid mandatory;
string property mandatory;
```

Adds a property to the map of properties of an object, if does not exist. Note that the effect is similar to `set_object_property` with the exception that the value remains unset (invalid) until it is actually set by some method later on.

get_object_property_names(*gid*);

string *gid* mandatory;

Return a list of all the names of the properties the object has.

get_object_events(*gid*);

string *gid* mandatory;

Return a list of all events associated with an object.

add_object_event(*gid*, *title*, *start*, *end*);

string *gid* mandatory;
string *title* mandatory;
string[yyyy-MM-ddTHH:mm:ss] *start* mandatory;
string[yyyy-MM-ddTHH:mm:ss] *end* optional;

Add a new event to the object.

get_object_status(*gid*);

string *gid* mandatory;

Return a dictionary of status values of the object e.g. whether it is online or not.

get_object_position(*gid*);

string *gid* mandatory;

Return the latitude, longitude and altitude of the object, in WGS-84 decimal format.

get_object_orientation(*gid*);

string *gid* mandatory;

Return the pitch, yaw and tilt of the object, in radians.

get_object_velocity(*gid*);

string *gid* mandatory;

Return the speed of the object, relative to ground plane.

3.3. Searching for objects

find_objects_by_category(*category*);

int *category* mandatory;

Return a list of names and GIDs of all objects whose category matches the search criterion.

find_objects_by_type(*type*);

string *type* mandatory;

Return a list of names and GIDs of all objects whose set type matches the search criterion.

find_objects_by_name(*name*);

string *name* mandatory;

Return a list of names and GIDs of all objects whose name matches the search criterion.

filter_objects(*filter*);

```
dict filter mandatory;
```

Return a list of names and GIDs of all objects characteristics match the values set in the filter criteria.

3.4. Invoking object capabilities and toggling actuators

use_object(*gid*, *capability*, *value*);

```
string gid mandatory;
string[({"ON", "OFF", "SET")] capability mandatory;
int value optional;
```

If the object represents an actuator, change the state of that actuator (or set its value) to the set value. E.g. setting capability to "ON" will set the actuator on. Defining the capability as "SET" and supplying the value parameter, will attempt to set the actuator to that value, if the actuator happens to support such an action.

turn_object(*gid*, *direction*);

```
string gid mandatory;
bool direction mandatory;
```

If the object represents an actuator, will try to find a capability that represents the switching of that actuator (i.e. ON or OFF). Then turns that capability according to the direction, true for ON, false for OFF.

toggle_object(*gid*);

```
string gid mandatory;
```

If the object represents an actuator and that actuator has a two way control such as a relay, will try to it toggle (i.e. change to the opposite direction).

use_action_by_gid(*gid*);

```
string gid mandatory;
```

Invoke a predefined action script. Note that because actions are assumed to be self-contained scripts, this method simply starts that script with no further paramters.

use_capability_by_gid(*gid*, *value*);

```
string gid mandatory;
variant value optionak;
```

If an object has a capability that can be recognized with a GID, will invoke that capability. In case the capability takes a value, will use the value supplies as a parameter.

3.5. Accessing system settings values

get_system_property(*property_name*);

```
string property_name mandatory;
```

get_device_property(*property_name*);

```
string property_name mandatory;
```

`get_mqtt_topics`

Lists all MQTT topic identifiers entered into the system.

`get_mqtt_topic_strings`

Lists all the actual MQTT topic strings entered into the system.