Sparkplug

Sparkplug is an adapter program, licensed under Apache License 2.0 for communicating with Quva Flow. Sparkplug currently supports sending messages as JSON and XML objects to our REST API. Sparkplug features a full suite of routines for validating the contents of the messages prior to sending them.

Installation

Obtain sparkplug from the GitHub repository:

```
git clone https://github.com/Quva/sparkplug.git
```

Choose a tag of your liking, e.g. 1.4.3, and grab that:

```
git checkout tags/1.4.3
```

Install prerequisites using pip:

```
pip install -r requirements.txt
```

after which go ahead and install sparkplug:

```
make clean build install
```

or, if you are missing `make`, call `setup.py` directly:

```
python setup.py clean build install
```

API Documentation

For now, there are two types of messages: Variables and Event. The former is used for declaring variables and their meta data, and the latter is used for declaring events. Both message types are currently supported by the Quva analytics service, but more will be added when needed.

Message Container

Each message is enclosed in a container, the Message Container. The Message Container has the following fields:

key	type	required	comment
message_header	Object	YES	Header of the message
message_body	Object	YES	Body of the message

The Message Container in JSON is expressed as:

```
{
  "message_header": {
    ...
  },
  "message_body": {
    ...
  }
}
```

In XML the Message Container is expressed as:

```
<message>
  <message_header>
    ...
  </message_header>
  <message_body>
    ...
  </message_body>
</message>
```

Message Header

Every Message Container contains Message Header with the following fields:

key	type	required	comment
message_type	Enum	YES	Indicates which message it is. Supported types currently are: “variables”, “event”
message_sender_id	String	YES	An ID that identifies the sender
message_recipient_id	String	YES	An ID that identifies recipient (Quva)
message_id	String	NO	An ID that uniquely identifies the message
message_reply	Object	NO	Used if the message needs to be replied

The Message Header takes the following form as JSON:

```
{
  "message_header": {
    "message_type": "<myevent>",
    "message_sender_id": "<mysenderid>",
    "message_recipient_id": "Quva",
    "message_id": "<myuniquemessageid>"
    "message_reply": {
      ...
    }
  },
  "message_body": {
    ...
  }
}
```

Message Reply field

Message Reply field inside the Message Header contains information about the topic the reply is sent to:

key	type	required	comment
reply_to_topic	String	YES	Specify the topic to which the reply is sent

The Message Reply field takes the following form:

```
    ...
    "message_reply": {
      "reply_to_topic": "<topicid>"
    }
    ...
```

Variables Message

Variables Message is contained inside the message body of the container that has type "variables" like so:

```
{
  "message_header": {
    "message_type": "variables",
    "message_sender_id": "<mysenderid>",
    "message_recipient_id": "Quva",
    "message_id": "<myuniquemessageid>"
  },
  "message_body": {
    "variables": [...]
  }
}
```

The list inside the "variables" field contains a list of objects with the following fields:

key	type	required	comment
variable_source_id	String	YES	Source identifier. For example //
variable_name	String	YES	Human-readable name for the variable. Does not have to be unique, i.e. multiple sources can share the same variable names.
variable_unit	String	NO	Scientific unit (for example SI) for the variable
variable_is_txt	Boolean	YES	Flag to denote whether the the variable should be treated as text or number
variable_properties	Map	NO	map of properties listed per variable, such as: origin table, site id, machine id, sensor id, etc. Can store at most 100 keys.

Variables Message should be sent just once to the service so as to register them. Without registering the variables they are not stored in the database and thus cannot be surfaced in the frontend nor used by analytics applications. The message contains all the meta data for all the variables that are of interest regarding analysis. Below is an example how the JSON containing the aforementioned fields should be formatted:

```
{
  "message_header": {
    "message_type": "variables",
    "message_sender_id": "<mysenderid>",
    "message_recipient_id": "Quva",
    "message_id": "<myuniquemessageid>"
  },
  "message_body": {
    "variables": [
      {
        "variable_unit": null,
        "variable_is_txt": true,
        "variable_source_id": "<country>/<site>/<unit>",
        "variable_name": "<localname>"
        "variable_properties": {
          "source_table_field": "<fieldname>",
          "source_table": "<tablename>"
        },
      },
      {
      ...
      }
    ]
  }
}
```

Variable identifier consists of two pieces of information: the name (`variable_name`) and source (`variable_source_id`). A variable that has a specific name can come from multiple sources. This convention makes it possible to pool together data for a single variable coming from different sources, which may be beneficial for analytics.

The current interface supports at most 1 million variables.

Event Message

Event Message is contained inside the message body of the container that has type "event" like so:

```
{
  "message_header": {
    "message_type": "event",
    "message_sender_id": "<mysenderid>",
    "message_recipient_id": "Quva",
    "message_id": "<myuniquemessageid>"
  },
  "message_body": {
    "event": {...},
    "measurements": [...]
  }
}
```

Event Messages are sent when a new event happens, or an old one gets updated. The service can identify whether the event is new or re-entered based on `event_id`. General event information is stored in the field `event` and has the following fields in it:

key	type	required	comment
event_id	String	YES	Unique string for every event
event_type	String	YES	Groups similar events together
event_start_time	Date	YES	What is the start time of the event
event_stop_time	Date	YES	What is the stop time of the event
event_properties	Map	NO	Map of properties for the event. Can store at most 100 keys.

Along with the event information comes the measurements, given in a separate field `measurements`. Inside `measurements` there is a list of objects with the following fields:

key	type	required	comment
variable_name	String	YES	What is the name of the variable
variable_source_id	String	YES	What is the source of the variable
measurement_time	Date	YES	When was the measurement taken
measurement_num_value	Double	NO	What was the measured value. Needs to be set if variable_is_txt is False.
measurement_txt_value	String	NO	What was the measured value. Needs to be set if variable_is_txt is True.
measurement_properties	Map	NO	Map of the properties of the measurement. Can store at most 10 keys.

Of these, `measurement_num_value` and `measurement_txt_value` are mutually exclusive and should be used according to how the variables are set in the Variables message (see `variable_is_txt` flag). Below is an example Event message in JSON format:

```
{
  "message_header": {
    "message_type": "event",
    "message_sender_id": "<mysenderid>",
    "message_recipient_id": "Quva",
    "message_id": "<myuniquemessageid>"
  },
```

```
  "message_body": {
    "measurements": [
      {
        "measurement_time": "2014-12-30 00:00:00+0200",
        "variable_source_id": "<country>/<site>/<unit>",
        "measurement_txt_value": "YES",
        "variable_name": "Is sensor active?"
      },
      {
      ...
      }
    ],
    "event": {
      "event_id": "<myeventid>",
      "event_stop_time": "yyyy-mm-dd HH:MM:SS+ZZZZ",
      "event_start_time": "yyyy-mm-dd HH:MM:SS+ZZZZ",
      "event_type": "<myeventtype>",
      "event_properties": {
        ...
      }
    }
  }
}
```

Feedback Message

Feedback Message is returned only if reply information is given and reply is requested. Quva Flow will return a Feedback Message on two occassions: * Upon retrieving and parsing a message. The Feedback Message informs whether retrieval, parsing, and action were successful or not. * Upon finishing analysis that triggers an alarm. The Feedback Message then contains information about the source of alarm.

Feedback Message has the has the usual top-level fields for Message Header and Message Body. Message Body inside the Feedback Message has fields

key	type	required	comment
analysis_result	Object	YES	Contains a list of variables that caused the alarm
event	Object	YES	Generic information

A Feedback Message could look like follows:

```
  "message_body": {
    "analysis_result": {
      [AlarmVariable1, AlarmVariable2, ...]
    },
    "event": {
      "original_event_id": "<myeventid>",
      "event_type": "QUALITY_FEEDBACK",
      "event_properties": {
        "OK_MESSAGE": "Everything OK",
        "ERROR_MESSAGE": "",
        "ERROR_URL": "",
        "ERROR_CODE": "0"
      }
    }
  }
```

where each Alarm Variable in the list has the fields

key	type	required	comment
variable_description	String	YES	Description of the variable. Same description as specified in the Variables Message
variable_group	String	YES	Looked up based on the specified variable property that defines the group.
alarm_description	String	YES	Description of the alarm.
measurement_num_value	Double	YES	Mean value of the samples in the event in which the variable raised the alarm.
min_measurement_specific_num_value	Double	YES	Measurement-specific min-threshold
max_measurement_specific_num_value	Double	YES	Measurement-specific max-threshold
min_empirical_threshold_num_value	Double	YES	SPC-based min-threshold
max_empirical_threshold_num_value	Double	YES	SPC-based max-threshold

and looks like as JSON:

```
"alarm_variable": {
  "variable_description": "<myvariabledescription>",
  "variable_group": "<myvariablegroup>",
  "alarm_description": "<myalarmdescription>",
  "measurement_num_value": 10.1,
  "min_measurement_specific_num_value": 3.5,
  "max_measurement_specific_num_value": 4.1,
  "min_empirical_threshold_num_value": 2.5,
  "max_empirical_threshold_num_value": 11.0
}
```

Sending messages to Quva Flow

Sending the message can be done using your favorite method that supports POST commands to REST API.

Via cURL

An example command to send message with cURL is:

```
curl \
     -u $USERNAME:$PASSWORD \
     -H "Content-Type: application/json" \
     -X POST \
     -d "@message.json" \
     -k \
     https://aiko.quva.fi:8162/<path/to/application>/ImportQueue?senderID=<mysenderid>
```

Note that `message_sender_id` is also provided in the request header, which helps sending notifications in case of malformed messages.

One can also pass XML message to cURL:

```
curl \
     -u $USERNAME:$PASSWORD \
     -H "Content-Type: text/xml" \
     -X POST \
     -d "@message.xml" \
     -k \
     https://aiko.quva.fi:8162/<path/to/application>/ImportQueue?senderID=<mysenderid>
```

Via sparkplug

Alternatively, one can use the sparkplug program that performs input validation before sending the data:

```
sparkplug \
      --payload message.json \
      --url https://aiko.quva.fi:8162/<path/to/application>/ImportQueue?senderID=<mysenderid> \
      --username $USERNAME \
      --password $PASSWORD
```

If one is only interested in input validation, a.k.a dryrun, the following will do:

```
sparkplug \
      --payload message.json \
      --isDryrun
```

The XML message is sent similarly (sparkplug infers the format of the message with the suffix):

```
sparkplug \
      --payload message.xml \
      --url https://aiko.quva.fi:8162/<path/to/application>/ImportQueue?senderID=<mysenderid> \
      --username $USERNAME \
      --password $PASSWORD
```