CALLBACK API

General information

The backend can automatically forward some events using the «callback» system.
A callback is a custom http request containing your device(s) data, along with other variables, sent to a given server/platform.
The callbacks are triggered when a new device message is received, when a location has been computed, or when a device communication loss has been detected for example.
The configuration of callbacks is done in the device type page.
You can configure a custom callback to forward events directly to your servers (you can find complete documentation below) or configure callbacks for one of the 4 platforms integrated in Sigfox's backend:

AWS IoT, you can find complete documentation about AWS IoT following this link.
AWS Kinesis, you can find complete documentation about AWS Kinesis following this link.
Microsoft Azure Event HUB, you can find complete documentation about Azure Event HUB following this link.
Microsoft Azure IoT HUB, you can find complete documentation about Azure IoT HUB following this link.

To make sure you receive callbacks, you should ensure that you've whitelisted the following backend IP address range. This is in CIDR notation.
185.110.97.0/24
These IPs may change (or be added to) in the future, and we will keep this page up to date.

Custom callback

There are 3 callback types.
You have a set of available variables for each type of callback.
These variables are replaced by their value when a callback is called.
When receiving a callback, the client system must return an HTTP 2xx code within 10 seconds. If the client system fails to process the callback during this time, an automatic retry will be scheduled 1 minute later.

Callback types

Each callback type shares a set of common variables :

`time (int)`: the event timestamp (in seconds since the Unix Epoch)

DATA

You can enter a configuration to define custom variables that will be replaced by the parsed data. You can then use these variables in your callback. To get more information about the format of the configuration, please read the online help.

This callback type defines the reception of a user message from a device. Common variable available for each subtype are :

`device (string)`: device identifier (in hexadecimal – up to 8 characters <=> 4 bytes)
`duplicate (bool)`: «true» if the message is a duplicate one, meaning that the backend has already processed this message from a different base station, «false» otherwise. To receive duplicate messages, you have to check the «send duplicate» box in the callback configuration page
`snr (float)`: the signal to noise ratio (in dB – Float value with two maximum fraction digits)
`rssi (float)`: the RSSI (in dBm – Float value with two maximum fraction digits). If there is no data to be returned, then the value is null.
`avgSnr (float)`: the average signal to noise ratio computed from the last 25 messages (in dB – Float value with two maximum fraction digits) or «N/A». The device must have send at least 15 messages.
`station (string)`: the base station identifier (in hexadecimal – 4 characters <=> 2 bytes)
`data (string)`: the user data (in hexadecimal)
`lat (float)`: the latitude, rounded to the nearest integer, of the base station which received the message
`lng (float)`: the longitude, rounded to the nearest integer, of the base station which received the message
`seqNumber (int)`: the sequence number of the message if available

Here are the subtypes for this type :

UPLINK

This subtype does not define any additional variable.

BIDIR

`ack (bool)`: true if this message needs to be acknowledged, false else.
The client can decide not to send any answer to the device. There are 2 ways to do so :
respond to the callback with the HTTP NO_CONTENT code (204).
respond with a json data containing the `noData` field ex :

```
{ "0CB3" :
    {
        "noData" : true
    }
}
```

SERVICE

This callback type defines the reception of an operational message from a device. Common variables are:

`device (string)`: device identifier (in hexadecimal – up to 8 characters <=> 4 bytes)
`duplicate (bool)`: «true» if the message is a duplicate one, meaning that the backend has already processed this message from a different base station, «false» otherwise. To receive duplicate messages, you have to check the «send duplicate» box in the callback configuration page
`signal (float)`: the signal to noise ratio (in dB – Float value with two maximum fraction digits)
`avgSnr (float)`: the average signal to noise ratio computed from the last 25 messages (in dB – Float value with two maximum fraction digits) or «N/A». The device must have send at least 15 messages.
`station (string)`: the base station identifier (in hexadecimal – 4 characters <=> 2 bytes)
`lat (float)`: the latitude, rounded to the nearest integer, of the base station which received the message
`lng (float)`: the longitude, rounded to the nearest integer, of the base station which received the message

Here are the subtypes for this type :

STATUS

`batt (float)`: the voltage of the battery (in Volt – Float value with two maximum fraction digits)
`temp (float)`: the temperature of the device (in °C – Float value with two maximum fraction digits)
`seqNumber`: the sequence number of the message if available

GEOLOC

`radius (int)`: the radius limit in meter that defines the zone in which the device is located.
`seqNumber (int)`: the sequence number of the message if available
`lat (float)`: the estimated latitude of the device within a circle based on the GPS data or the Sigfox Geolocation service
`lng (float)`: the estimated longitude of the device within a circle based on the GPS data or the Sigfox Geolocation service

ACKNOWLEDGE

`infoCode (int)`: this is the status code of the downlink :
`0 (ACKED)` the station emitted the answer,
`1 (NO_ANSWER)` the client did not give any answer,
`2 (INVALID_PAYLOAD)` the data to send to the device is invalid,
`3 (OVERRUN_ERROR)` the device exceeded its daily downlink quota, so it was blocked because of a lower priority than transmissions for devices that did not exceed their quota,
`4 (NETWORK_ERROR)` it was not possible to transmit the answer,
`5 (PENDING)` not technically a code that is sent in the callback because it is a transient state before the answer is sent,
`6 (NO_DOWNLINK_CONTRACT)` the device asked for an answer but its BSS order does not allow downlink,
`7 (TOO_MANY_REQUESTS)` the device asked for an answer before the expiration of the listening time,
`8 (INVALID_CONFIGURATION)` the device type is configured to get data by callback, but no BIDIR callback was defined
`infoMessage (string)`: a message associated to the code.
`downlinkAck (bool)`: true if the station acknowledged the transmission, false else.
`downlinkOverusage (bool)`: true if the device exceeded its daily quota, false else.

REPEATER

`batt (float)`: the voltage of the battery (in Volt – Float value with two maximum fraction digits)
`whitelistDevices (int)`: the number of devices in the white list. If 0, all messages are repeated
`repeatedMsg (int)`: the number of repeated messages since the last status frame transmission
`unwantedDevices (int)`: the number of unique devices not in the white list received since the last status frame transmission. If 15, means that the number is >14
`unwantedMsg (int)`: the range of sum of number of messages received from all devices not in the white list: not repeated messages since the last status frame transmission: 0:0, 1: <5, 2: <10, 3: <50, 4:<100, 5:<150, 6:<200, 7:<250, 8:<300, 9:<350, 10<400, 11:<450, 12:<500, 13:<800, 14: <1000, 15: >1000
`wrongMsg (int)`: the range of total number of malformed frames received since the last status frame transmission: 0:0, 1: <5, 2: <10, 3: <50, 4:<100, 5:<150, 6:<200, 7:<250, 8:<300, 9:<350, 10<400, 11:<450, 12:<500, 13:<800, 14: <1000, 15: >1000
`overRegulMsg (int)`: the range of number of messages not repeated due to compliance with local regulation since the last status frame transmission: 0:0, 1: <5, 2: <10, 3: <50, 4:<100, 5:<150, 6:<200, 7:<250, 8:<300, 9:<350, 10<400, 11:<450, 12:<500, 13:<800, 14: <1000, 15: >1000

ERROR

This callback type defines the communication loss for a device. Custom variables are:

`device (string[])`: List of device identifiers (in hexadecimal – up to 8 characters <=> 4 bytes) separated by comma
`info (string)`: Information on error, in case of communication loss, contains the last received message date of the device
`severity (string)`: «ERROR» when it's a device problem, «WARN» when the network has experienced some issues that could cause some message loss or delay.

Callback medium

The medium defines the way to forward the event.

EMAIL

You'll need to setup a valid e-mail, a subject and the message body.
The subject and the body can contains arbitrary text with defined markup which are the name of variables surrounded by brackets for this callback type.

HTTP

You'll need to implement a RESTful, web-facing service. There are two kinds of callback:

Simple

Each message is directly forwarded in a single HTTP request. You can use HTTP **GET**, **POST** or **PUT** method although **POST** method is recommended.

Variables

According the type of callback, different variables are available. The available variables list is displayed above the URL field. These variables can be used in 3 places :
URL : as path variables or request parameters.
Headers : as values of a header. Variables cannot be used in header keys because format is standardized.
Body : If you choose to use the POST or PUT method, you can define a body template that contains variables. These variables are replaced the same way as in URL or Headers.

Headers

You can set custom headers in your callbacks. For security reason, we disallow all standards headers except **Authorization**. This header allows you to use other authentication method than Basic. You cannot put twice the same header. As you can put your user info in the URL in the form of http://login:password@yourdomain.com, be careful not to put an Authorization header.

GET method

The variables are automatically replaced in query string.

```
GET http://hostname/path?id={device}&time={time}&key1={data}&key2={signal}
```

POST/PUT method

POST or PUT method allows you to set the content-type and the body of your request You can choose the content-type between 3 :

application/x-www-form-urlencoded
This is the default content type for POST or PUT request. When using this content-type, you can set body in form encoded format :

```
device={device}&data={data}&time={time}
```

If this format is not respected, it will be rejected.
Note that if you put some variables in the query part of the URL, these parameters will be appended to the body, whether it is empty or not. Eg :

```
POST http://hostname/path?id={device}&time={time}&key1={data}&key2={signal}

with a body template:
device={device}&data={data}&time={time}

will result in the following body:

id={device}&time={time}&key1={data}&key2={signal}&device={device}&data={data}&time={time}
```

application/json
You can set a JSON object in the body :

```
{
    "device" : "{device}",
    "data" : "{data}",
    "time" : {time}
}
```

When this content type is chosen, JSON content is validated. If the content does not respect the JSON standard, it will be rejected. Please refer to JSON specification for more details
text/plain
This content type is a free. No validation is performed (except forbidden characters). Eg :

```
device => {device}
data => {data}
time => {time}
```

Batch

Messages are gathered together per callbacks, each line representing a message, then sent in batch using a single HTTP request every second. This avoids a possible peak load that your server can not handle. As the payload contains multiple messages, only HTTP POST method are supported. When using batch callbacks with the duplicate option, there is no guarantee that duplicates of a given message will be grouped in the same batch. It all depends on the exact moment of processing of each of these duplicates.

```
POST http://hostname/path?batch={batch} where batch={device};{data};{signal};...
```

Downlink callbacks

When a message needs to be acknowledged, the callback selected for the downlink data must send data in the response. It must contain the 8 bytes data that will be sent to the device asking for acknowledgment. The data is json formatted, and must be structured as the following :

```
{
    "device_id" : { "downlinkData" : "deadbeefcafebabe"}
}
```

With **device_id** beeing replaced by the corresponding device id, in hexadecimal format, up to 8 digits. The downlink data must be 8 bytes in hexadecimal format.

With batch callbacks, multiple devices may need an acknowledgment. The response must contain the data for all these devices. It will take this form :

```
{
    "device1_id" : { "downlinkData" : "deadbeefcafebabe"},
    "device2_id" : { "downlinkData" : "bebebabab0b0b1b1"}
}
```

If the json contains other data, or data related to devices not asking for acknowledgment, it will be ignored.