Sigfox API usage

External Use

Contents

1. Sigfox data interfaces 2
2. API Introduction 2
3. API Information and versioning 2
4. How to use API 3
   - Credentials generation 3
   - API documentation 3
   - API usage 4
   - Authentication and Credential Renewal 4
   - Paging 5
   - Paging limits 6
   - API Result handling and interpretation 8
     - **Error cases** 8
     - **Successful requests** 8
   - Too many requests 9
5. Example: get coverage information 10
   - Environment requisites: API scope 10
   - Preparing API request 10
   - Implementation 12
     - **With RESTlet plugin:** 12
     - **With curl command line:** 12
     - **With python:** 12
     - **With java:** 12
6. Best practices 13
   - When API should be used 13
   - When API must **not** be used 13
   - Sample use case, mixing all interfaces 13
   - Sample use case involving APIs pooling 14
7. APIs that are worth a look 14
   - Missed messages 14
   - Downlink selection 16
8. Sigfox API Version 2 16

1. Sigfox data interfaces

Sigfox provides 3 different interfaces to access data:

- Graphical User Interface (GUI) to access and alter data in a visual way, for human operators. All available data are accessible through GUI.
- Callbacks to receive automatic notification upon specific event, as for example new messages from devices, newly activated device… The target of callbacks are computers running an HTTP server. Only event-based data are eligible to callbacks.
- API, to request, create or alter a specific data through HTTP request. The originator of the request is a computer running a computer program, to do specific tasks. Most of the data operations are available through API, such as group creation, device registration, callback creation, etc.

2. API Introduction

Most of the features and data found on Sigfox backend website can also be accessed using a webservice API in a programmatic way. API requests are easy to generate and handled by multiple software and languages such as C, Java, Python, Curl, browser extensions, etc. The return data is in JSON format which is easily parse-able and compatible with all programming languages.

The current API is the second generation of Sigfox API (v2), which was publicly released in October 2018.

The API v2 base URL is https://api.sigfox.com/v2/

3. API Information and versioning

API changes are, as all Sigfox backend functions, exposed in the backend release note. You can also request, in addition, to receive it directly to your inbox, which is highly recommended for developers. To do so, click on the profile icon (first icon upper left), then check the box beside “Subscribe to release email”.

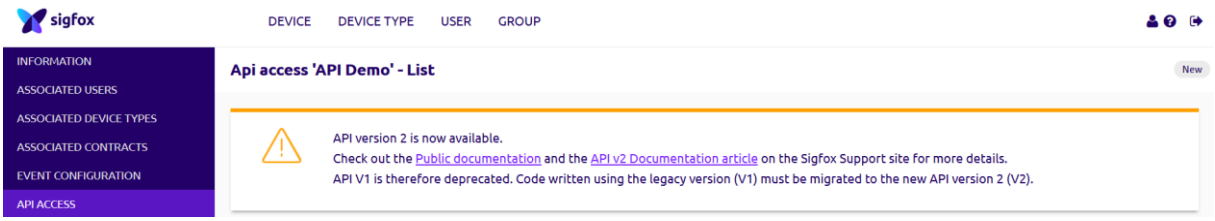
The API changes complies with the API versioning rules. Basically, added features (new end points and new key / value pairs) are added in the current API version, whereas modification and depreciations (removing end point, removing key / value pair, renaming of key, additions in closed lists…) are inserted exclusively in newer API versions. All those changes will be notified in the release notes.
You can consult the detailed specification of the Semantic Versioning in the following document: https://support.sigfox.com/docs/api-versioning

4. How to use API

As APIs access is restricted to authenticated API user, the first step to access API is to generate API credentials.

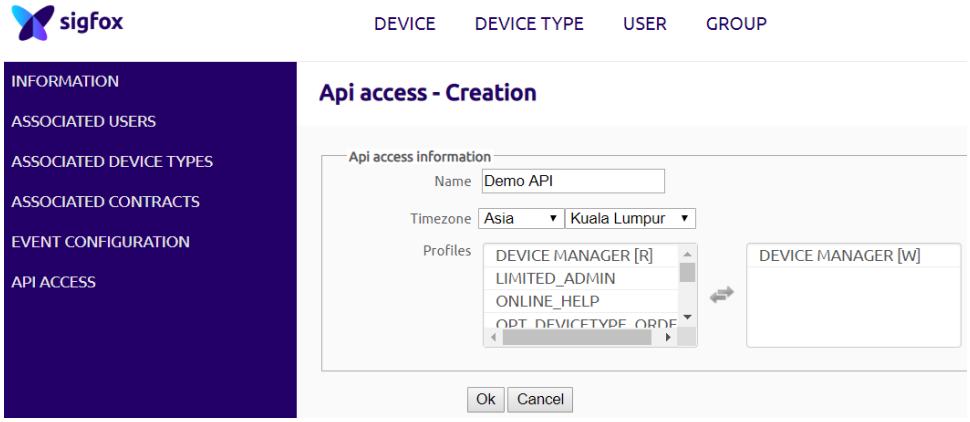
Credentials generation

To generate API credentials, you must go to the group that will support the API, then choose API Access in the submenu and hit the 'New' button to generate new credentials.



The interface will then prompt for the API name (descriptive info) and applicable time zone and, the most important part, the accessible roles for this new API. Accessible API methods and output response will be determined by the roles which are set for API credential. For example, with a read only role, the API won't allow any creation or update, but only access data reading.

API with Device Manger[W] is one of useful API for device management which allows registration or moving devices, creating and editing callbacks and accessing a device's PAC, etc.



API documentation

The full API documentation details can be retrieved through API with appropriate API user credential. Public documentation based on Device Manager[W] role is directly available through support.sigfox.com (https://support.sigfox.com/docs/apidocs) and "API v2 documentation" article can also be accessible in Sigfox Resources portal (https://support.sigfox.com/document/api-documentation).

API usage

Following API documentation, the REST principle (usage of POST, GET, DELETE, PUT HTTP requests) is used for Sigfox API: any access to APIs is done through an authenticated HTTPS request, with the URL shown in the documentation, and the potential parameters as explained in the documentation.

The PUT request is the only request used to edit an existing entity. You need to specify every value of the parameters to be updated in the body of the request. To remove non-mandatory parameters, set their values to “null” inside the body of the request.

The result code of the HTTP request will be used to check the correct behavior (20x result code indicate success, such as 200 or 204), and the return data will give the optional data (such as job ID, requested data…).
On failures (40x HTTP result codes, such as 403), the associated content will describe the reason of the issue.

You can refer to the following document to have detailed meanings on HTTP status and error codes: https://support.sigfox.com/docs/api-response-code-references

Highlight:

- No pooling message and device sync status through API
- Reasonable API calls according to fleet
- Retrieve message and device status by Callback
- Reasonable callbacks per Device type

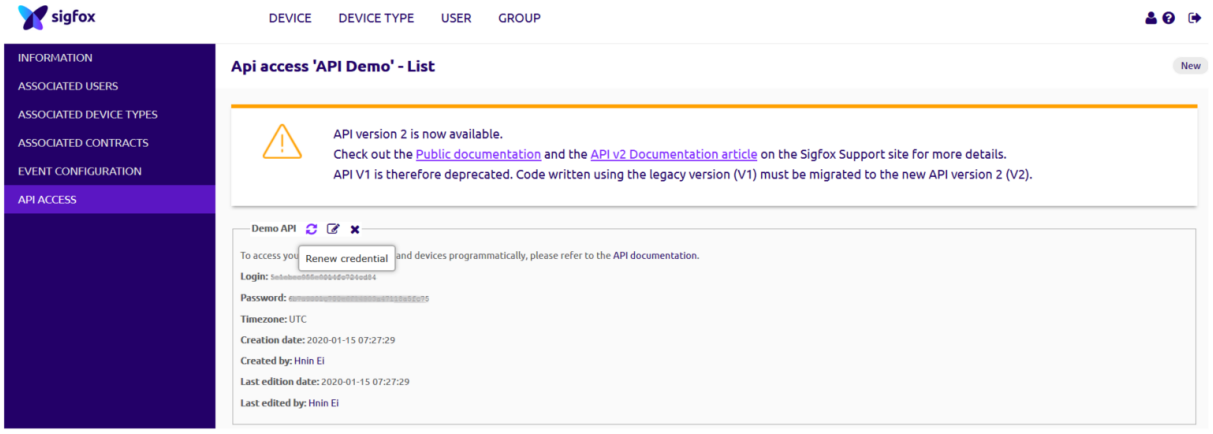
Authentication and Credential Renewal

The API URL is only accessible using secure version of HTTP protocol, HTTPS, with a valid login and password, API credentials. Authentication credentials are associated to a group, which define all accessible data that will be available with those credentials (groups visibility, devices and devices types, operations allowed…).

When created, a group does not have any default associated API credentials, but you can create them, with required profiles, under API access. If the API credentials ever get compromised, new password can be generated at any moment, invalidating the previous one.

To renew API credentials, either to change the login password in preventive mode or after an attack, you must go to the group that will support the API, then choose API Access in the submenu and click on the ‘Renew’ button to generate new credentials with the same scope.

As a best practice, if multiple systems use APIs calls, it is highly recommended to have at least one different credentials pair per system. If one of them is compromised, it will be easier to renew the credentials (using the renew button) for this specific system and change its credentials rather than changing all the credentials for other systems.

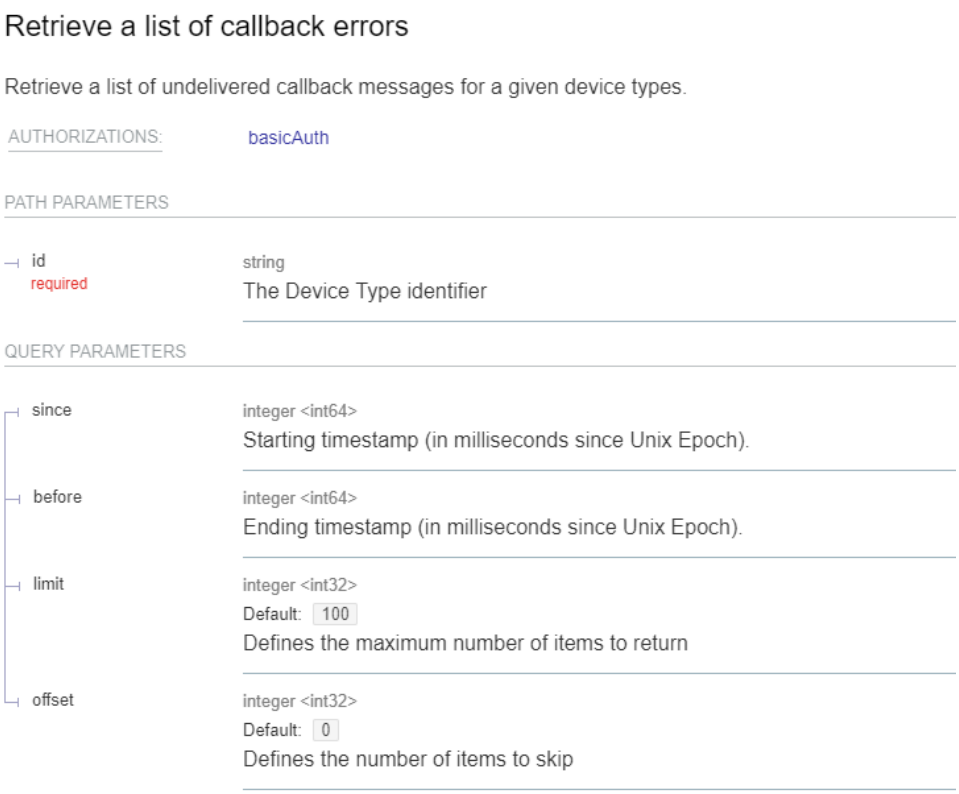


Note that after renewing credentials, all systems that used the previous credentials must be updated with the newest login and password values.

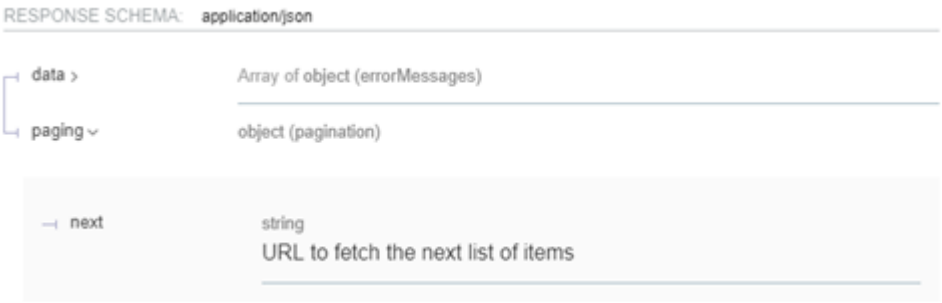
Paging

If a large amount of data is expected from an API request (above 100 objects), the reply will be sliced. In this case, every reply will contain a link to the next slice in the “paging” section.

If available in the concerned API, you might also use the offset parameter to access directly one of the next slice part instead of starting from the first one (for example, to list the device types from a group omitting the first 20 ones would result in sending the device type list request with an offset parameter set to 20).



In any case, if the data must be retrieved via multiple requests, any further request will be specified as an URL in the response, in the “paging” section.



Note: if on a paged request an HTTP status 429 is received, it indicates that this HTTP request was too close to the previous request. You need then to wait for a guard delay prior reiterating your request. You can read the following *Too many requests* section for better understanding.

Paging limits

Even if paging allows requesting more than 100 objects, it will fail to retrieve very large number of objects. For example, if you want to retrieve all your devices hand have hundreds of thousand devices, using only paging will fail. The reason is that, when your initial API request is processed, the full set of objects is retrieved. But with hundreds of thousand objects, the full set will be too large to fit the allowed memory usage per request.
In such cases, you should rely on filters and sort combination to retrieve the desired data. In our example, using devices end point, you might then use the sort on device ids in combination with the minimal id or maximum id depending of the sort direction. For example, with ascending sort on id, you should use the minimal id, with the last id retrieved plus one on each request.

Retrieve a list of devices

Retrieve a list of devices according to visibility permissions and request filters.

AUTHORIZATIONS: basicAuth

QUERY PARAMETERS

id	string The device's identifier (hexadecimal format)
groupIds	Array of string Returns all devices under the given groups (included sub-groups if the parameter deep is equals to true)
deep	boolean Default: false if true, we search by groups and subgroups through the parameter 'groupIds'
deviceTypeId	string Returns all devices of the given device type
sort	string Default: "name" Enum: "id" "-id" "name" "-name" "lastCom" "-lastCom" The field on which the list will be sorted. (field to sort ascending or -field to sort descending)
minId	string The minimal id of the filtered range, only availble when sort parameter is set to "id" or "-id"
maxId	string The maximal id of the filtered range, only availble when sort parameter is set to "id" or "-id"

The same applies for example with messages, with filter based on date.

API Result handling and interpretation

The first check to be done once the request has been sent is the return code.
If the return code is not 20x, then there is an issue with the request.

Error cases

For example, in the case of coverage information request (see coverage example below), if the radius parameter is set to 10, then the return code would be 400. In that case, the body in the HTTP reply is helpful to understand the issue:

If another request targets, for example, a device that is out of the API scope, then an HTTP 403 return code will be sent.

You might face a 429 error code in case of over-usage of API calls. See below the Too many requests section to understand the reason leading to this error message and how to design a compliant application that will fix the situation.

The API documentation can be then helpful to understand the result code, but the body text associated with the response will be the most relevant information about the error for a human being.

Note however that this message might be changed without warning. Thus, it should be avoided to interpret it in a programmatic way. That's also the reason why the error message value is not listed in the API documentation.

Successful requests

If the result is a 20x, the request has been successful. Once the content has been retrieved, the result can then be interpreted according to the documentation.

In our coding example (see Implementation section below), with 3 positives, non-zero values, we have 3 or more base stations covering our targeted surface of 200 meters radius.

For a different position returning only 2 strictly positive values, then the targeted surface is only covered by two stations and, for another position that do not return positives values, then the targeted surface isn't covered.
Depending on the objective in this example, the API request result can be used to build:

- A binary, either covered or not position (Covered or not)
- A result of redundancy at this specific position (0,1,2 or 3 and more base stations)

- A useful information on the margin for this specific position (39dB margin)
- A simulation of coverage knowing the attenuation due to environment (indoor: 20dB margin is usually considered) or device used (1U device has a up to 7dB penalty compared to 0U devices)
- A note built from all those considerations and the application (for static objects, reliability and resilience will come from margin and a bit from redundancy, whereas moving objects will greater benefit from redundancy than pure margin…, for example 87%)

All those results interpretations can (and should) be done to present an information adapted to the user.

Too many requests

Since July 2019, Sigfox has implemented a rate limit policy to avoid potential harmful situation coming from API misusage. Whenever API requests are made to the same base URL, the Sigfox cloud will now block requests in over usage (too many requests within the same period).

In such case, the HTTP return code will be 429, *Too many requests*, regardless of the validity of the request.

If you face this situation, you should think about inserting a delay in your request or, better, rethink your use.

In most cases, the limit delay between requests is function of the base URL called. For example, device information might have a bigger delay than devices messages.

For example, if you are facing *Too many requests* status requesting through API device information, you might have to consider caching information: device information will unlikely change between two close requests. Indeed, once the device is registered, the only changes will be activation time (that can be detected on messages reception), token expiration (that can occur once per year per device) and others administrative events. Thus, requesting device information on multiple time within a same second is not efficient. It would produce, in quite all cases, the very same result.

On another case, if you face *Too many requests* status requesting through API device messages, you should consider using callbacks instead of API calls to recover messages from devices. The latest method is all but efficient, as a device should send 140 messages per day maximum. Thus, requesting for new messages through API on each second will result in an efficiency rate below 0.1% (86400 requests per day to find out 140 pertinent results). This situation cannot scale, because after a few hundred of devices (and may be before), you'll face a resource issue. In comparison, the callbacks have a 100% efficiency in this case.

If you want to know more on API usage limits and rules, please consult the following document: https://support.sigfox.com/docs/api-rate-limiting

5. Example: get coverage information

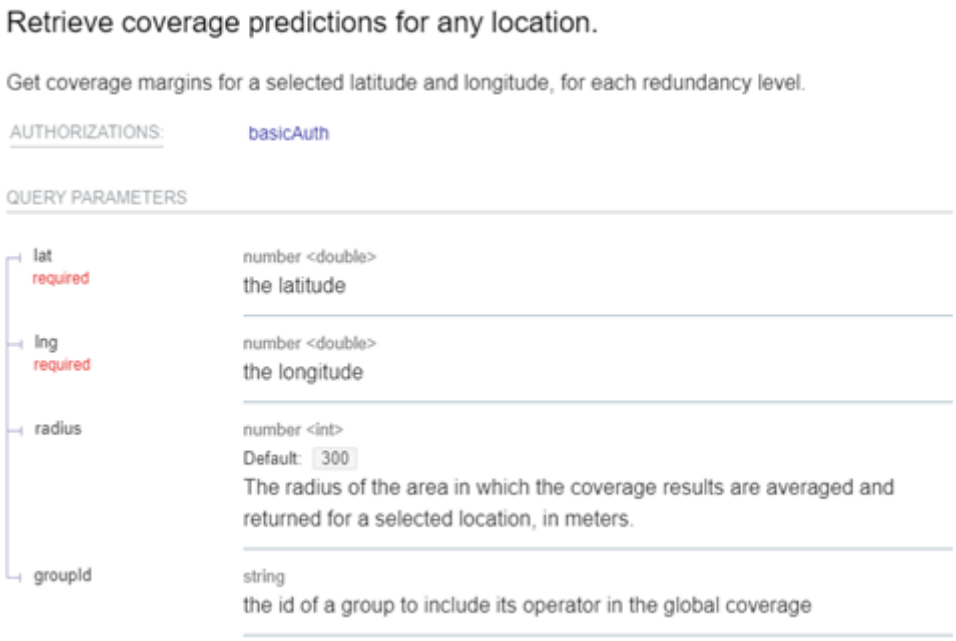
Environment requisites: API scope

To access coverage information, an API must be created with read roles: Device Manager Read or Customer Read for example is fully enough to access this API endpoint. On the other end, user creation will require a Customer Write or Device Manager Write role.

As for this example involving initial API creation, especially for production APIs, it is highly recommended to use strict, less permissive roles: in case of security breach, having low scope APIs role exposed will contain the breach.

Preparing API request

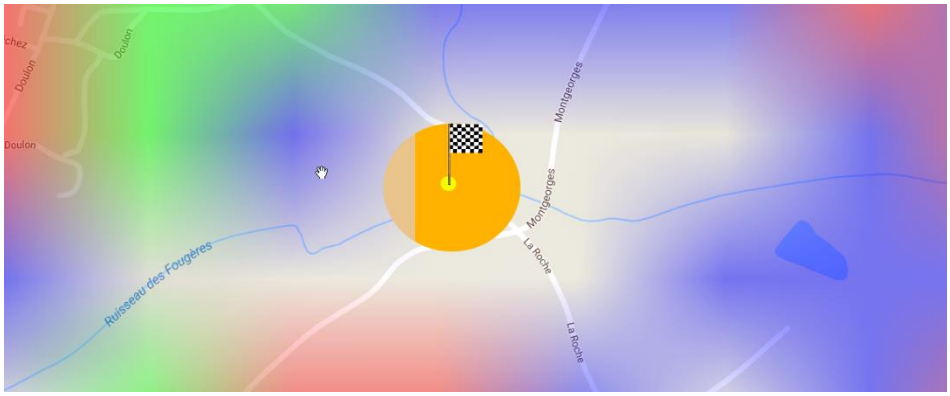
Below is an example of Global Coverage API (single point) to get coverage levels for any location. The API description shows that two mandatory parameters must be provided, and two being optional.



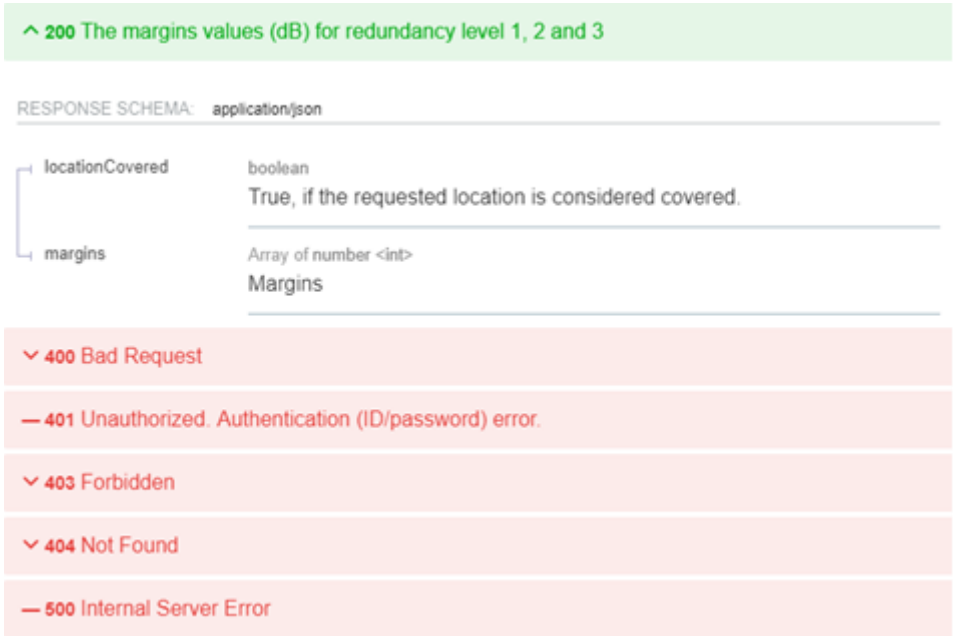
The latitude and the longitude are mandatory. The radius and groupId are optional. Note that if the input data is a postal address, corresponding latitude and longitude must be retrieved using distinct API, either using google, mapbox, opencage, openaddresses or any others geocoding APIs.
The radius, the optional parameter, will define the surface around the test point to consider. As described in the documentation, 300 meters is the default value, but we can set a value down to 200 meters.

Note that the coverage simulation accuracy doesn't go down to 1-meter precision (two nearby coordinates will have same the coverage value, coverage "definition" is above several dozen of meters), thus the global coverage doesn't allow values below 200 meters.

Considering 200 meters radius will avoid most mixed results from two adjacent simulated surfaces that might lead to average value between a non-covered and a covered zone. For example, with the image bellow, when using the orange radius, there will be a mix between a covered zone (at the left of the flag represented by a pastel-orange surface) and a non-covered zone (full orange surface). On the opposite, with the smaller yellow radius, there will be no coverage for the same coordinates. Thus, with the same position, with an orange -bigger-radius, the coverage will exist (with low coverage values), whereas with the yellow -smaller-radius there will be no coverage.

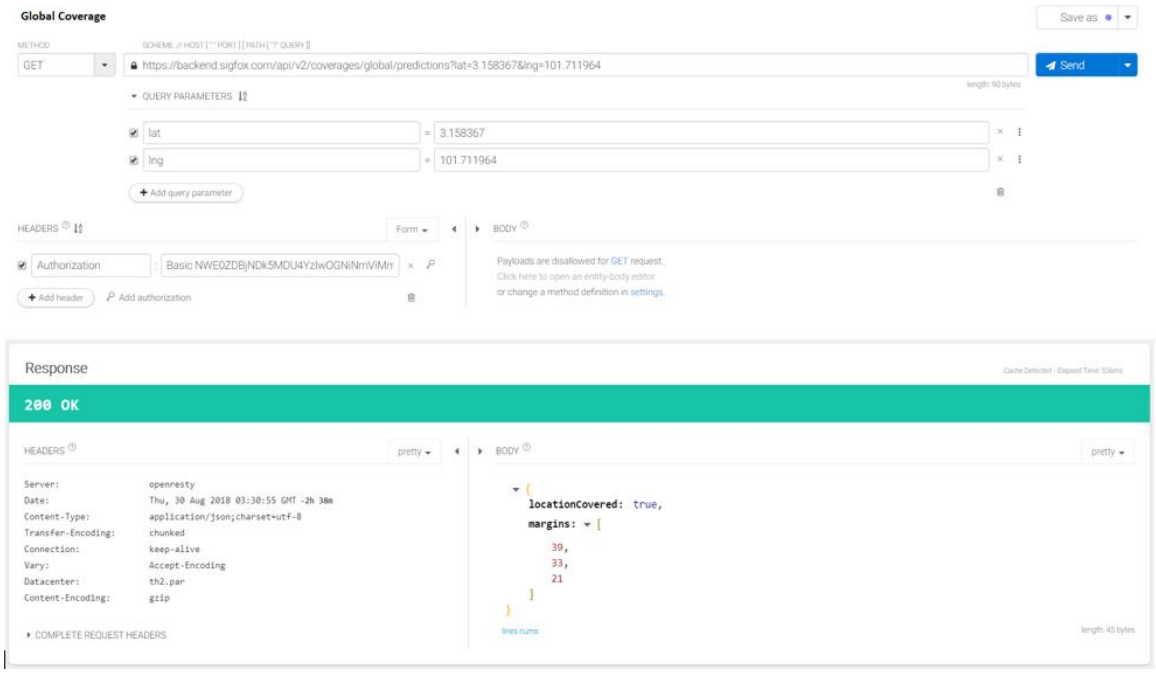


Once longitude, latitude and optionally radius have been set, the request can be sent to the API endpoint. For example, to test the API call, you can use a browser plugin such as RESTlet client for Chrome, a command line command such as curl with Linux, or a programing language such as python, java or others. The response will come as described in the documentation, in JSON format.



Implementation

With RESTlet plugin:



Tip: Enter the URL and click on “Query parameters” to add the desired parameters, and on “Add authorization” for the credentials before hitting the “Send” button

With curl command line:

```
curl  --get --data lat=43.52 --data lng=1.55 --data radius=200
      --user 5***6:2***3
      https://backend.sigfox.com/api/v2/coverages/global/predictions
…
* Connection #0 to host backend.sigfox.com left intact
{"margins":[31,27,25]}
```

With python:

```
import requests
parameters = {"lat": 43.52, "lng": 1.55, "radius": 200}
login = "5***6"
password = "2***3"
authentication = (login, password)
response =
requests.get("https://backend.sigfox.com/api/v2/coverages/global/predictions",
                        auth=authentication,
                        params=parameters)
# The variable response contains the response from the server
```

With java:

```
URIBuilder builder = new URIBuilder();
builder.setScheme("http").setHost("https://backend.sigfox.com")
                         .setPath("/api/v2/coverages/global/predictions")
                         .setParameter("lat", 43.52)
                         .setParameter("lng", 1.55)
                         .setParameter("radius", 200);
URI uri = builder.build();
String user = "5***6";
String pwd = "2***3";
HttpGet httpget = new HttpGet(uri);
httpGet.addHeader("Authorization",
                  "Basic " + Base64.encodeToString(
                                 (user + ":" + pwd).getBytes(),
                                 Base64.NO_WRAP));
response = httpget.getURI();
// The variable response contains the response from the server
```

6. Best practices

When API should be used

APIs should be used in any recurring task that have rare occurrence, for example:

- Perform a device move from one device type to another upon end customer subscription
- Disengage sequence number of a device after receiving event callback warning
- Retrieve missed callback after event callback send error or user's database maintenance
- Replacement of a failed device by a working one
- Synchronize the group organization

Other data retrieval task might benefit from API calls:

- End user exposition of coverage on a specific address
- Monthly check of token duration associated with devices
- PAC retrieval prior to moving a device

When API must **not** be used

APIs shouldn't be used to target event-based data: they aren't efficient with pooling (see Too many requests session). The right way to deal with event-based data is using callbacks. They can be used on conjunction with APIs (and should), and APIs might then be used to go further on processing this single event (for example, moving a device to a "verified" device-type after receiving a specific message…).

APIs shouldn't be used to retrieve messages or status data, nor refresh user database periodically, especially if the pooling frequency is above once per day.

Sample use case, mixing all interfaces

For example, if a user wants to build and keep its own copy of Sigfox backend data, such as the groups and device type organization, callbacks definition, registered devices list and all the messages received, then synchronization between Sigfox and user servers' database should use the following principles:

- Messages synchronization should be done through callbacks (push mechanism),
- status of devices should be synchronized through events (push mechanism),
- Group organization, device types, callbacks and settings (such as email, URL…) through APIs

Note that the latest data (such as group organization, device types, callbacks and setting, etc.) retrieved by API, are usually manipulated by a user. In such cases, the user server should provide its own GUI to manipulate groups, device type, callbacks and user. Thus, all

interactions will be done on user's server instead of Sigfox GUI, with perfect integration and without unnecessary pooling.
In this example, the only task that involves Sigfox GUI is the original API creation itself.

Sample use case involving APIs pooling

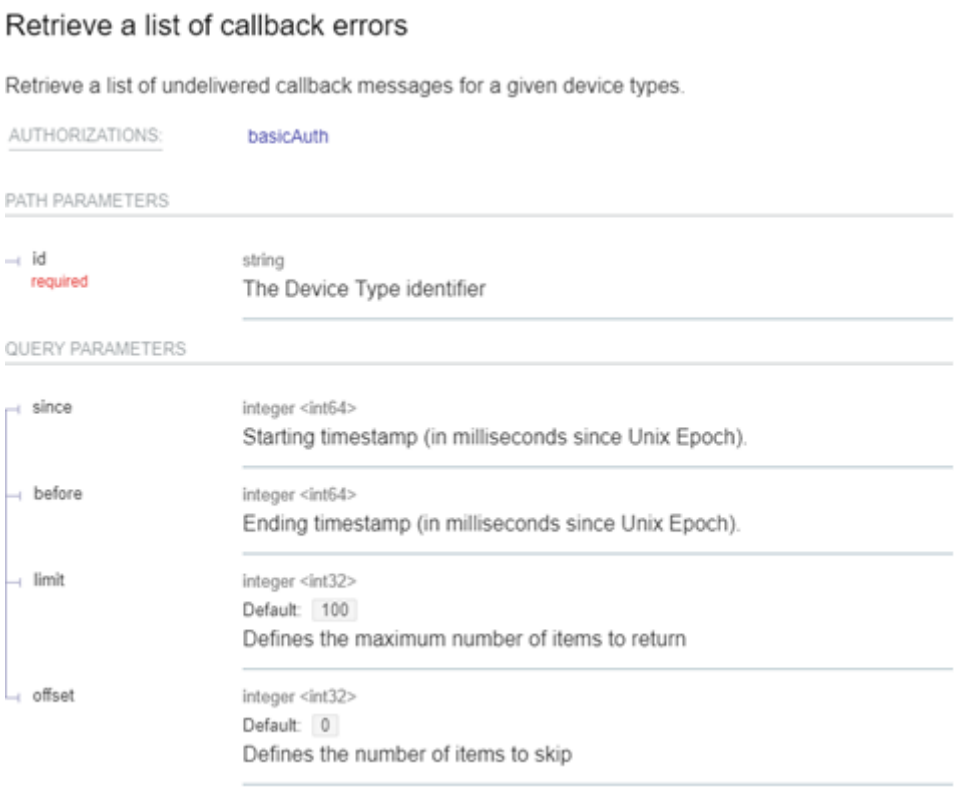
In example above, if a user wants to keep using the Sigfox GUI for specific -rare- tasks at one-point but still synchronize the organization with its own servers, then periodically or manually, a status refresh must be done. It will mostly use APIs queries to pull and report the changes made on Sigfox GUI to the user's servers. In such case, pooling through APIs may be used, although it cannot be as efficient as directly using APIs to handle those rare tasks.

When APIs pooling is required, the pooling frequency must match the probable usage. For example, if the only part relying on Sigfox GUI is the group organization, and if the occurrence of group creation / update / deletion is one per month, pooling automatically changes through API should not be done on a minute base: reasonable number of requests will likely show a change. **Either a monthly based timing or a manual triggering should be preferred.**

7. APIs that are worth a look

Missed messages

One of the main usages of Sigfox is to get data from devices using callbacks. But what's happened when a callback is missed? How to get information back to the server handling callbacks?

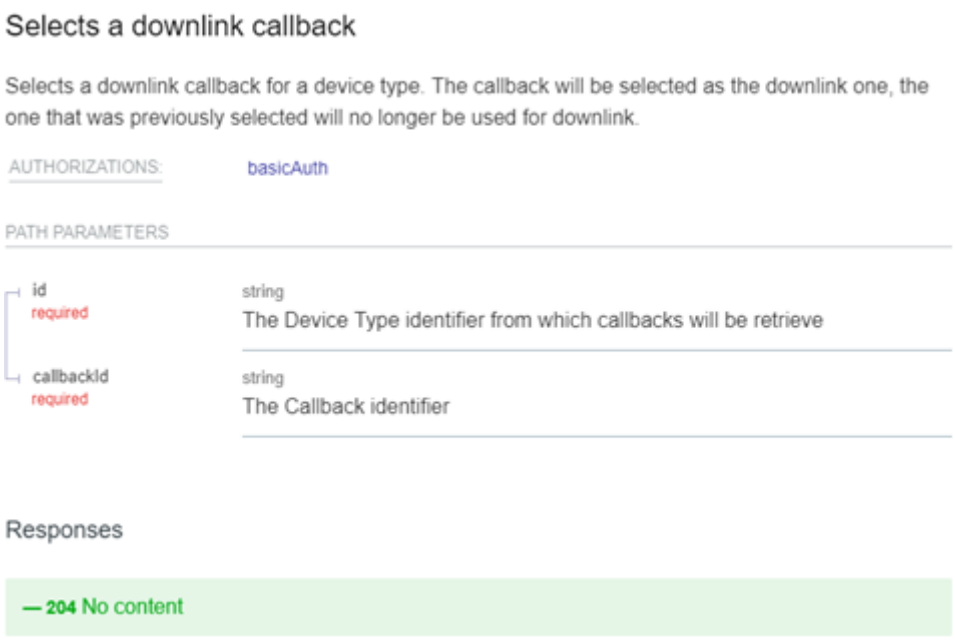


After a server downtime or after receiving an event callback for missed callback, you should use the API addressing missed messages.

The response will include all information regarding the missed callback and, the more important info, the content of the callback as it was produced by Sigfox systems in the “callback” object.

Downlink selection

When bidirectional mode is used with messages, you might want to change the downlink URL to redirect the requests to another server while working on a planned maintenance on the main server. The downlink selection can help to define the right URL to get downlink answer from.



8. Sigfox API Version 2

The current API version, called version 2, presents a more consistent grammar, with improved respect of RESTful principles, more complete documentation as well as uniformization of outputs (both single info and grouped -lists- info will have the same format).

For example, to get the list of all devices' last communication date from a group containing sub groups by using with the current API version 2, it is now possible to get the information directly from a single GET request on devices' end-point, with usage of "groupId" filter and "deep" set to true. Additionally, you can request the API to sort the devices list on the last communication date, using "sort" parameter with value "-lastCom" or "lastCom" (depending on the sorting direction).

Now if for a specific reason you want to also have the device type name, with the addition of "fields" parameter with value "deviceType(name)", you can request it to be added to the output result.

With the usage of filters and additional fields, most of the complexity that might exist in previous version of APIs, for example to retrieve a specific information on all targeted devices, groups, contracts or device types, is now efficiently available through one simple request. Thus, if the existing code is based on loops to get the same information from a list of objects, a redesign using the power and simplicity of the API version 2 is greatly profitable.

Do not hesitate to use online API documentation, ask.sigfox.com or support.sigfox.com to get help on this latest version and request assistance.