NOCCELA

API Documentation

Noccela Oy

December 14, 2023

This document is written for API version **1.3**.

Contents

Introduction

Noccela is a state-of-the-art indoor localization system with infrastructure consisting of beacons installed at a physical site and smart tags which communicate with the said beacons. Beacons are wired with ethernet cables which provides them with both WAN access and operating power using *power over ethernet* (PoE). The tags come in various forms, most common being an EAS tag attached to retail store products (pictured in figure 1). Also available are personnel badges that can be carried by personnel in a lanyard. Different partner use-cases might also call for new form-factors which can be implemented.

Beacons (pictured in figure 2) aggregate and mediate data traffic between cloud and smart tags as well as perform the measurements needed for locating the tags. Positions of smart tags can be arbitrary but the locations of beacons are well-known, allowing later approximation of the tag positions. Two-way communication means that not only can the data flow from tags to the cloud, but the tags can also be configured and updated on the fly with ease. This allows configuring tags for specific environments remotely.

Figure 1: Smart Tag

The main data flowing from the physical site up to cloud is the RF data relevant to localization, but the tags also have various on-board sensors, such as accelerometer and temperature sensors whose data can also be utilized.

The brains of the system is the cloud software situated out of site in remote servers around the world. This cloud software environment consists of a large number of different software components handling tasks ranging from hosting web portals to data analysis and running the actual positioning algorithms. Keeping the on-site software simple allows Noccela and partners utilizing the data to focus on the domain case using high-level languages and have minimal considerations for the low-level software.

The localization is achieved using an unique proprietary algorithm based on *time of arrival* (ToA) run in the cloud. These approximate locations are then stored in database and forwarded straight into real-time components that require them, including the third party applications that connect to the cloud's external API described in this document.

Figure 2: Beacon 3, upcoming smaller version of the beacon

Third parties (partners, evaluators) have two ways to access this location data and other relevant data such as site layout.

1. First and simplest one is to fetch historical data through our HTTP API using simple HTTP requests. This is used to fetch historical data or for applications to which the immediate availability of the location is not crucial.

2. Second and slightly more complex way is to utilize WebSockets to establish a direct TCP socket to our cloud. This socket does away with the HTTP overhead and provides immediate access to location data as the tag location is calculated in the cloud. In this model the cloud pushes data to third parties.

Both of these methods require authentication using a JSON Web Token (JWT) which is obtained from authentication server. This process is described in the next section.

Glossary

Tag Smart tag that is located by the system.

Beacon Internet connected device that communicates with the tags.

Account Owner of one or multiple sites, for example a company or store chain.

Site A physical location with installed beacons and a layout i.e. a store or office.

App Application that has single set of credentials to connect to the Noccela cloud.

Client An instance of an application implementation that has a token or fetches one using credentials and communicates with the API e.g. a web app.

Real-time data Data that is sent to client immediately when it's available e.g. tag locations as soon as they are calculated.

Historical data Historical data from database that is queried by the client. Sometimes aggregated and filtered!

Links

Noccela home page https://www.noccela.com

Customer portal https://my.noccela.io

Partner portal https://partner.noccela.io

API base address https://api.noccela.io/

Auth. server base address https://auth.noccela.io/

Authentication

Overview

To request an access token, user has to create an application in the *partner portal* (https://partner.noccela.io). The number of apps is not limited, so it is best to create an app for each use case instead of using a single one for all. Apps can be given full or subset of the set of privileges the partner has, but can be prevented to access any site or scope not deemed necessary for the app (principle of least privilege). The permissions are granted per site, giving account-wide permissions is not supported from the web UI but can be achieved by contacting Noccela. One can find the app-specific *client id* and *client secret* from the app's details page in partner portal.

Third parties authenticate using OAuth2 authentication standard implemented by the authentication server (authorization server) and Noccela Cloud (resource server). Only OAuth2 flow permitted at the moment is *client credentials* which means the third party application (client) fetches a token from authorization server using client id and client secret (i.e. password). The response contains an access token in the form of a JWT token and the expiration time in seconds. Decoded JWT also contains absolute expiration time as well as time it was issued. This token is attached to HTTP requests sent against the HTTP API or sent through WebSocket. This flow is very simple and no in-depth knowledge of OAuth2 is required.

A very important thing to realize about JWT authentication is that it is self-contained and stateless. It has all relevant permissions coded inside it and if the signature is valid, the backend can trust the information it contains. This also means that if permissions are updated, old tokens don't change and new token has to be requested for the changes to take effect. This might mean that permission changes take at most 24 hours to fully propagate i.e. until all old tokens have expired.

The address of the authentication server is `https://auth.noccela.io`

The address of the token endpoint is `https://auth.noccela.io/connect/token`

The token request is a **POST** request with *Content-Type* of *application/x-www-form-urlencoded* as required by the standard. The body contains three parameters, *client_id*, *client_secret* and *grant_type* with value *client_credentials*. The grant type parameter is mandatory!

Examples

Example raw HTTP request (with some headers omitted)

```
POST /connect/token HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Host: auth.noccela.io

client_id=CLIENT_ID&client_secret=SECRET&grant_type=client_credentials
```

Example in JavaScript:

```
// Build the request body.
const authRequestBody = new URLSearchParams();
authRequestBody.append("client_id", 123);
authRequestBody.append("client_secret", "hunter2");
authRequestBody.append("grant_type", "client_credentials");

// Send request, await response.
const authResponse = await fetch(url, {
    method: "post",
    headers: {
        "Content-Type": "application/x-www-form-urlencoded"
    },
    body: authRequestBody
});

// Extract relevant values from response JSON.
const {
    expires_in: tokenExpiration,
    access_token: accessToken
} = await authResponse.json();
```

Example response body, content type is always *application/json*

```
{
    "access_token": "...",
    "expires_in": 86400,
    "token_type": "Bearer",
    "scope": "global"
}
```

Re-authentication

Response contains the relative expiration time for the token in seconds, starting from the moment the request was processed. When the token expires, calls to API return HTTP status code 403.

Subsequent calls to token endpoint return a *new* token with new expiration time starting from that moment, effectively resetting the expiration. It does not matter if the previous token is still valid.

It is up to third party to handle refreshing the token and it is advisable to request a new one when there's less than half of the time remaining in current token.

The moment of expiration can be calculated by comparing the expiration time to moment of request, but if the state is not available, the decoded JWT also contains the expiration time as Unix timestamp.

HTTP API

Overview

Noccela Cloud exposes HTTP endpoints for third party apps, for accessing historical data such as latest known locations of tags, to request histories in CSV for data analysis and to request and modify site layouts for installation purposes.

The base address to this API is https://api.noccela.io/

When querying historical tag location data, an important thing to know is that this data is filtered! Only states which cause the tag to relocate over 0.5m from the previous stored position is written to database. Real-time updates should be handled over socket.

Devices (tags, beacons) are identified in the system exclusively by their serial number, called "device ID" in the documentation and code. It is printed on their label and can be read from there either as plain text or bar code.

The response has status code reflective of the operation. Some of the error messages also include status text which helps debugging the issue. Some common status codes returned are:

200 OK, with fetched data for GET operations.

204 No content, when data was modified with POST operations.

400 Bad request, when the request is somehow invalid or malformed.

403 Forbidden, when token is invalid or token doesn't grant permissions for the request.

404 Not found, when endpoint is not known or if requested data is not found.

5xx Server error, when there was an error in the server's end.

User authentication happens by including the JWT token in *Authorization* HTTP header with *Bearer* prefix as follows.

```
Authorization: Bearer *TOKEN*
```

Noccela provides a Swagger/OpenAPI specification that is generated from application code and thus is always up to date and comprehensive. Noccela Cloud exposes *Swagger document* and *Swagger UI* for logged-in partner users. The former is a standardized complete JSON description of the API which enables for example automatically generating client SDKs using any of the numerous existing code generators. The latter is a web-based UI for viewing information about the various methods in the API, as well as testing them in browser.

Swagger document https://api.noccela.io/swagger/v1/swagger.json

Swagger UI https://api.noccela.io/swagger

Further documentation about the HTTP API is available in the above link. It can be always assumed that the document is up to date if the latest API schema is selected.

Using Swagger UI

https://swagger.io/tools/swagger-ui/

Swagger UI is a web-based interactive document that provides comprehensive information about the various methods (individual endpoints) in the API, including their descriptions, parameters, response models (schemas) and content types etc. To view the page one must be signed in the the partner portal.

Viewing all of the data is possible without authentication, but to test the API user has to authenticate. This is done by clicking the green "Authorize" button in the top right section of the page, providing the *client id* and *client secret* from app's page and clicking "Authorize". Selecting the "global" scope is not necessary. It is the single OAuth scope that is implicitly given to all clients even if they do not explicitly request it.

After the user has authenticated one can test the endpoints by clicking "Try It Out" on a method, providing the parameters and then clicking "Execute".

It should be noted that the WS endpoint is listed here but cannot be tested using the Swagger API as it requires WebSocket connection. Successful request is listed as returning *101 Switching Protocols* response.

Real-time WebSocket protocol

Overview

If the user's use case requires real-time updates for tags' state and location, the use of WebSocket API is recommended. If the use case requires the location of tag once in a while, like once in an hour or so, using HTTP API might be a better choice.

The WebSocket API works by client sending WS upgrade request at the *realtime* endpoint of the server that currently handles the target site. *The URL should contain account and site IDs as parameters.*

The domain of the server can be requested from an HTTP endpoint listed in OpenAPI document. This domain is then used when constructing the socket endpoint URL.

For example, request to `https://api.noccela.io/realtime/domain?account=1&site=1` returns domain of `api.n1.fi1.noccela.io`. The endpoint into which the WebSocket request should be made for this site is thus `https://api.n1.fi1.noccela.io/realtime?account=1&site=1`.

The authentication does not happen through HTTP headers due to most implementations not supporting additional headers in upgrade request (even though not strictly forbidden by WS specification).

Authenticating happens by sending the JWT token after the connection has been opened. It must be the first message sent to server, without any extra formatting. If the authentication fails the socket is closed, if it is successful the server sends a JSON message with type *authSuccess* and token *nbf* and *exp* fields (issuance and expiration timestamps).

The API has three types of messages as follows.

1. Ping messages, which are empty messages. They must be answered with pong message which is **"1"**. If ping is not answered, the connection times out in two (2) minutes.

2. Requests from client to server. These have following form. *All properties are mandatory.* Empty payload is represented by empty JSON object.

```
{
    "uniqueId": 123, // A unique ID, used to match response
    "action": "...", // Action name as string
    "payload": {} // Action specific payload. No payload -> empty object!
}
```

These requests are responded by server with responses of following form.

```
{
    "uniqueId": "123", // String ID that matches the response to the request.
    "status": "ok", // Status message, such as 'ok', 'badRequest'
                    // , 'forbidden', 'serverError'..
    "payload": {} // Response payload
}
```

All requests to server generate a response such as above, this can be used to verify that event registration was successful etc.

3. Server-sent messages without matching request. This can be for example location or tag status update the client has subscribed to. These have the following form

```
{
    "action": "...", // The type of the message
    "status": "ok", // Status message
    "payload": {} // Depending on the event type, either JSON payload or
                  // BASE64-encoded MessagePack string
}
```

Decoding encoded message

To save bandwidth, some large message types (namely, initial tag status and initial alert state) are encoded in binary format called *MessagePack* which compresses JSON significantly (the larger the JSON, the larger the compression ratio). These initial state usually contain the complete state of every tag on the site which causes the messages to be largeish.

Check https://msgpack.org/ for more information on the format.

The binary message is then encoded in BASE64 representation and set as the *payload* of the message. Other types of messages are in plain JSON.

In MessagePack some dictionaries (like tag state) are converted into arrays without property names, so this has to be converted back to dictionary using the known indices of properties.

Following example shows how to decode the message into JSON in JavaScript using an npm library `"@ygoe/msgpack"`.

```
// BASE64 -> byte array.
const bytes = atob(baseMsg);

const arrayBuffer = new ArrayBuffer(bytes.length);
const intArray = new Uint8Array(arrayBuffer);

for (let i = 0; i < bytes.length; i++) {
    intArray[i] = bytes.charCodeAt(i);
}

// Call deserialize from MsgPack implementation.
const payload = deserialize(intArray);

// Resulting data is plain Javascript object.
//...
```

To see how to get the original property names check the message-specific notes below.

Message types

This chapter describes each relevant message type in the WS API and the form of the related requests and responses. Read the descriptions carefully as different actions might have some peculiarities in their implementation. Short list of these same messages can be found in appendix E.

1. Site information

Fetch comprehensive site information, including its layout.

Request

```
{
    "uniqueId": str,
    "action": "getSite"
}
```

Response

```
{
    "uniqueId": str,
    "status": "ok",
    "payload": {
        // Site details.
    }
}
```

2. Initial tag state

Fetch initial tag state for *all* tags on the given site. The API works by first fetching the initial state, then updating the incrementally by registering to *registerTagDiffStream*.

This request will result in response being sent immediately and only once, it does not register any event. To get subsequent updates, check diff request. If the socket is broken for a period of time, some of the incremental updates might have been lost and thus this request should be invoked again after connection has been re-established.

Request

```
{
    "uniqueId": str,
    "action": "initialTagState",
    "payload": {
        "deviceIds": int64[] | null // Device IDs for tags, if omitted
                                    // or null then all tags from site
                                    // are registered.
    }
}
```

Response

This is the response to request.

```
{
    "uniqueId": str,
    "status": "ok"
}
```

Server sent event

This is the actual payload event sent immediately.

```
{
    "action": "initialTagState",
    "status": "ok",
    "payload": str // "AAAQ2..."
}
```

The response is BASE64-encoded MessagePack-encoded JSON which is to be decoded like described in previous section. The result is an object of status arrays, with fixed indexes for different properties. The keys of the outermost object are deviceIds of the tags. This is decoded to semantic property names in the following JavaScript example.

For the explanation of the properties, refer to appendix C.

```
const payload = parseMsgPack(msg);
const result = {};
const tagMacs = Object.keys(payload);

for (const [deviceId, tagData] of Object.entries(payload)) {
    const tagObj = {
        deviceId: +deviceId // Keys are strings.
    };
    tagObj["name"] = tagData[1];
    tagObj["batteryVoltage"] = tagData[2];
    tagObj["batteryStatus"] = tagData[3];
    tagObj["status"] = tagData[4];
    tagObj["areas"] = tagData[5];
    tagObj["wire"] = tagData[6];
    tagObj["reed"] = tagData[7];
    tagObj["isOnline"] = tagData[8];
    tagObj["timestamp"] = tagData[9];
    tagObj["x"] = tagData[10];
    tagObj["y"] = tagData[11];
    tagObj["accelerometer"] = tagData[12];
    tagObj["floorId"] = tagData[13];
    tagObj["signalLost"] = tagData[14];
    tagObj["powerSave"] = tagData[15];
    tagObj["deviceModel"] = tagData[16];
    tagObj["fwVersion"] = tagData[17];
    tagObj["strokeCount"] = tagData[18];
    tagObj["z"] = tagData[19];
    tagObj["uncertaintyDistance"] = tagData[20];

    result[tagObj["deviceId"]] = tagObj;
}

return result;
```

3. Incremental updates to tag state

To get the latest complete state of tags on a site, request the initial state described previously.

This event registers a hook for status updates on tag state. These status updates include any changes in tag's state, including location but many other properties as well such as battery status. Messages contain only the updated values (except "isOnline" which is sent every time), other values can be assumed to have stayed the same.

Register request

```
{
    "uniqueId": str,
    "action": "registerTagDiffStream"
    "payload": {
        "deviceIds": int64[] | null // Device IDs for tags to receive updates, if
                                    // omitted or null then all tags from site
                                    // are registered.
```

```
    }
}
```

Registration request has also matching unregister request, when client no longer needs updates but wants to keep the socket open. This is not necessary if the socket is closed.

Unregistration works by default by removing an event ONLY if the filters match i.e. account, site and deviceIds are the same. Use the same filters to unregister that you used to register.

Unregister request

```
{
    "uniqueId": str,
    "action": "unregisterTagDiffStream",
    "payload": {
        "deviceIds": int64[]
    }
}
```

Server sent event

```
{
    "action": "tagDiffStream"
    "status": "ok",
    "payload": {
        "tags": {
            "123": { // Dictionary indexed by device ID.
                "...": "..." // Changed properties.
            }
        },
        "removedTags": int64[] // Tags removed from site.
                               // Not present if empty.
    }
}
```

4. Initial alert states

Querying initial alert state works in much the same way as initial tag state. The response is MessagePack-encoded. One must provide date ranges for the alert search. The dates should be given as string, preferably as ISO 8601 compliant date or datetime strings.

Request

```
{
    "uniqueId": str,
    "action": "initialAlertState",
    "payload": {
        "dateRanges": [
            // Multiple date ranges may be provided.
            // However prefer only one date range per request.
            {
                "start": str, // Any ISO standard compliant date should do
                              // "2019-01-01" etc.
                "end": str // "2020-01-01T00:01:00Z" Z postfix means UTC time
            }
        ]
    }
}
```

Response

```
{
    "uniqueId": str,
    "status": "ok",
}
```

Server sent event

```
{
    "action": "initialAlertState",
    "status": "ok",
    "payload": str // "AAA..."
}
```

Response payload will be BASE64 encoded binary data encoded in MessagePack. Check previous sections to see how to decode it.

5. Incremental updates to alert state

Registering to alert change stream will cause the server to send new alerts created on the site and updates to existing alerts to the client.

Register request

```
{
    "uniqueId": str,
    "action": "registerAlertDiffStream"
}
```

Unregister request

```
{
    "uniqueId": str,
    "action": "unregisterAlertDiffStream"
}
```

Server sent event

```
{
    "action": "alertDiffStream",
    "status": "ok",
    "payload": {
        "alerts": {
            // Indexed by alert id. Note that the key is string!
            "123": {
                // Alert data.
            }
        },
        "removedAlerts": int64[] // Removed alert ids.
    }
}
```

6. Alert details

Request tag's path before the alert was created.

Request

```
{
    "uniqueId": str,
    "action": "alert",
    "payload": {
        "alertId": int64
    }
}
```

Response

```
{
    "uniqueId": str,
    "status": "ok",
    "payload": {
        "alertId": int64,
        "points": [
            // List of tag coordinates leading to the alert.
            {
                "x": int32,
                "y": int32,
                "timestamp": int64,
                "floorId": int32
            },
            {
                "x": int32,
                "y": int32,
                "timestamp": int64,
                "floorId": int32
            }
            // ...
        ]
    }
}
```

7. Location updates

Register to location updates, containing only coordinates and floor ID. If only coordinates are needed this is the preferred action to get tag location in real-time and for many applications the only action needed!

Register request

```
{
    "uniqueId": str,
    "action": "registerTagLocation",
    "payload": {
        // Devices to get updates for. If null all devices
        // from the site are registered.
        "deviceIds": int64[] | null
    }
}
```

Unregister request

```
{
    "uniqueId": str,
    "action": "unregisterTagLocation",
    "payload": {
        // Devices to unregister.
        // If null all from the site are unregistered.
        "deviceIds": int64[] | null
    }
}
```

Server sent event

```
{
    "action": "locationUpdate",
    "payload": {
        // Dictionary indexed by device ID.
        "123": {
            "x": int32,
            "y": int32,
            "z": int32,
            // If the site has a single floor, omitted.
            "floor": int32,
            // Timestamp when position engine has calculated position,
            //as ms since epoch.
            "timestamp": int64,
            // Timestamp when distance measurements has been conducted,
            //as ms since epoch.
            "twrTimestamp": int64,
            //If defined, then tags real position lies within this radius
            //from tags position
            "uncertaintyDistance": int32
```

```
        }
    }
}
```

8. Historical contact traces

Query which returns the contact traces from the given timespan and tags. Can be used to initialize the state for a contact tracing system. Differs from ”initial tag state” in some ways, the response is not encoded and it is sent directly as the response to the request and not as separate server sent event.

Request

```
{
    "uniqueId": str,
    "action": "initialContactTracingState",
    "payload": {
        // Serial numbers for tags to limit the scope of the request,
        // or null for all tags.
        "deviceIds": int64[],
        // Search range start timestamp as Unix timestamp in milliseconds
        // (milliseconds since epoch).
        "start": int64,
        // Search range stop timestamp as Unix timestamp in milliseconds.
        // Can be omitted for current moment.
        "stop": int64
    }
}
```

Response

```
{
    "uniqueId": str,
    "status": "ok",
    "payload": [
        // Same format as for contact trace updates, refer to the
        // next section.
    ]
}
```

9. Contact trace / Safe zone violation updates

Register to receive contact traces as they happen. A contact trace represents a contact, i.e. a period of time two tag-carrying individuals (or their tags, specifically) have spent closer than the specified safety distance.

Register request

```
{
    "uniqueId": str,
    "action": "registerContactTracingStream",
    "payload": {
        // Devices to get updates for. If null all devices
        // from the site are registered.
        "deviceIds": int64[] | null
    }
}
```

Unregister request

```
{
    "uniqueId": str,
    "action": "unregisterContactTracingStream",
    "payload": {
        // Devices to unregister.
        // If null all tags from the site are unregistered.
        "deviceIds": int64[] | null
    }
}
```

Server sent event

```
{
    "action": "contactTracingUpdate",
    "payload": [
        {
            // Unix timestamp (ms) for the moment the trace was registered
            // in cloud.
            "cloudTimestamp": int64,
            // Unix timestamp (ms) for the tag's timestamp for contact
            // in P2P mode.
            "tagTimestamp": int64,
            // First tag's serial number.
            "tag1": int64,
            // Second tag's serial number.
            "tag2": int64,
            // Serial number for the beacon the tag uploaded data to
            // in P2P mode.
            "beacon": int64,
            // Closest distance the tags were to each other, in cm.
            "distance": int32,
            // Duration of the contact in seconds.
            "duration": int32,
            // Unix timestamp (ms) for the contact start.
            "start": int64,
            // Unix timestamp (ms) for the contact end.
            "stop": int64,
            // The system the event originated from.
            "source": "RTLS" | "P2P",
            // The type of the contact, two tiered with different
            // configurable safe radii.
            "level": "Major" | "Minor"
        }
    ]
}
```

Note that properties that do not have a value are omitted. The contact traces can originate from two different systems, RTLS (full locationing system) or P2P (peer-to-peer, tags work independently). These work differently and some properties are relevant only for either one of the systems, for example *tagTimestamp* represents the moment tag recorded the contact, based on its internal clock. *Beacon* is the serial number for the beacon P2P tag uploaded its contact traces to.

10. P2P tag distance update

Register to receive distance measurements between two tags. Tags do peer-to-peer distance measuring only in environments where it has been explicitly enabled, which is mostly environments where contact tracing is used.

Register request

```
{
    "uniqueId": str,
    "action": "registerP2PDistanceStream",
    "payload": {
        // Devices to get updates for. If null all devices
        // from the site are registered.
        "deviceIds": int64[] | null
    }
}
```

Unregister request

```
{
    "uniqueId": str,
    "action": "unregisterP2PDistanceStream",
    "payload": {
        // Devices to unregister.
        // If null all tags from the site are unregistered.
        "deviceIds": int64[] | null
    }
}
```

Server sent event

```
{
    "action": "p2pDistanceUpdate",
    "payload": [
        {
            // First tag participating in the measurement.
            "tag1": int64,
            // Second tag participating in the measurement.
            "tag2": int64,
            // Beacon that sent the data i.e. the one tag uploaded
            // the packet into.
            "beacon": int64,
            // Measured distance between the tags in mm.
            "distance": int32,
            // Unix timestamp of the measurement.
            "timestamp": int64
        }
    ]
}
```

11. TWR-distance update

Register to receive distance measurements between a beacon and a tag. By default the creation of these events is disabled and in order to receive these events it has to be agreed with Noccela first.

Register request

```
{
    "uniqueId": str,
    "action": "registerTwrStream",
    "payload": {
        // Tag id's to get updates from. If null all devices
        // from the site are registered.
        "deviceIds": int64[] | null
    }
}
```

Unregister request

```
{
    "uniqueId": str,
    "action": "unregisterTwrStream",
    "payload": {
        // Devices to unregister.
        // If null all tags from the site are unregistered.
        "deviceIds": int64[] | null
    }
}
```

Server sent event

```
{
    "action": "twrStreamData",
    "payload": [
        {
            // The tag participating in the measurement.
            "tId": int64,
            // The beacon participating in the measurement.
            "bId": int64,
            // Unix timestamp of the measurement.
            "t": int64,
            // Measured distance between the tag and the beacon in mm.
            "d": int32
        }
    ]
}
```

12. Get blueprint

Fetch the blueprint image data. Layout information needs to be fetched before this to obtain the needed fileId(s).

Request

```
{
    "uniqueId": str,
    "action": "getBlueprint",
    "payload": {
        "fileId": int32
    }
}
```

Response

```
{
    "uniqueId": str,
    "action": "getBlueprint",
    "payload": {
        "type": str, // file type
        "name": str, // name of the file
        "data": str //Image data as base64 string
    }
}
```

13. Send signal

Send signaling request to Noccela's signal device.

Request

```
{
    "uniqueId": str,
    "action": "sendSignal",
    "payload": {
        "deviceId": int64,
        // Module indices 0 and 1 can be requested in this array
        "modules": [{
                "index": uint16, // index of the module.
                "value": uint16 // value for the module.
        }]
    }
}
```

Response

```
{
    "uniqueId": str,
    "action": "sendSignal",
    "payload": {} // No payload
}
```

14. Play tag buzzer

Request tag(s) to play their buzzer and/or blink led.

Request

```
{
    "uniqueId": str,
    "action": "playTagBuzzer",
    "payload": {
        // Device id's for the tags.
        // If null or omitted => all tags on the site get the request!
        "devices": int64[] | null,
        // If tag should play loud buzzer.
        //  If omitted => lower volume gets played.
        "alertSound": boolean,
        // Seconds to play buzzer.
        // If only led is wanted, then zero should pe put here.
        //  If null or omitted => default value of 15 is used.
        "buzzerSeconds": uint16 | null,
        // Seconds to blink led.
        // If only buzzer is wanted, then zero should pe put here.
        // If null or omitted => default value of 15 is used.
        "ledSeconds": uint16 | null,
        // Buzzers on interval in steps of 50 ms.
        // Value 5 means that buzzer will be on for 250 ms on each interval.
        // If null or omitted => default value of 5 is used.
        "buzzerOnInterval" uint16 | null,
         // Buzzers off interval in steps of 50 ms.
        // Value 5 means that buzzer will be off for 250 ms on each interval.
        // If null or omitted => default value of 5 is used.
        "buzzerOffInterval" uint16 | null,
         // Color of the tags led.
        // If null or omitted => default value of green is used.
        "ledColor": "green" | "red" | "yellow" | null,
        // Led blinking frequency. 0 = no blink, 1 = 4 Hz, 2 = 2 Hz, 3 = 1 Hz.
        // If null or omitted => default value of 2 is used.
        "ledBlinkFrequency": 0 | 1 | 2 | 3,
        // In order to play tags buzzer/led the tag has to be alive.
        // If this is set to true and the tag is not alive,
        //  the request will be sent to the tag when it becomes alive again.
        // If this is set to false and tag is not alive, the request is ignored.
        // If omitted =>default value false is used.
        "playWithDelay": boolean
    }
}
```

Response

```
{
    "uniqueId": str,
    "action": "playTagBuzzer",
    "payload": {} // No payload
}
```

15. Site layout update

Register to receive events when site's layout gets adjusted.

Register request

```
{
    "uniqueId": str,
    "action": "registerLayoutChanges"
}
```

Unregister request

```
{
    "uniqueId": str,
    "action": "unregisterLayoutChanges"
}
```

Server sent event

```
{
    "action": "siteLayoutChanged",
    "payload": [
        {
            // MinorId of the new layout.
            "minorId": int32
        }
    ]
}
```

16. Refresh token

Refresh jwt-token.

Request

```
{
    "uniqueId": str,
    "action": "refreshToken",
    "payload": {
        "token": str
        }
}
```

Response

```
{
    "uniqueId": str,
    "action": "refreshToken",
    "payload": {
        "tokenExpiration": int64,
        "tokenIssued": int64,
        "cloudVersion": str
        }
}
```

17. Initial beacon state

Fetch initial beacon state for *all* beacons on the given site. This can be mainly used for monitoring beacons' online statuses and in some cases also charging status and battery voltage. The API works by first fetching the initial state, then updating the incrementally by registering to *registerBeaconDiffStream*.

This request will result in response being sent immediately and only once, it does not register any event. To get subsequent updates, check diff request. If the socket is broken for a period of time, some of the incremental updates might have been lost and thus this request should be invoked again after connection has been re-established.

Request

```
{
    "uniqueId": str,
    "action": "getInitialBeaconState",
    "payload": {
        "useCompactData": bool        //if omitted or false, compact mode is used.
                                        //otherwise returns json
        }
}
```

Response

```
{
    "uniqueId": str,
    "action": "initialBeaconState",
    "payload": msgpack str or json
}
```

Parse compact response

```
{
   const payload = parseMsgPack(msg);
    const result = {};

    for (const [deviceId, beaconData] of Object.entries(payload)) {
        const did: number = +deviceId;
        const beaconObj = {
            online: beaconData[0],
            charging : beaconData[1],
            voltage : beaconData[2],
        };
        result[did] = beaconObj;
    }
    return result;
}
```

JSON-response

Null values in charging or voltage means those aren't available for current hw.

```
{
          {
                "deviceId1": {
                        "online": bool,
                        "charging": bool | null,
                        "voltage": int32 | null
                },...
        }
}
```

18. Register beacon diff stream

Register to receive events when site's beacons status gets changed.

Register request

```
{
    "uniqueId": str,
    "action": "registerToBeaconChangeStream",
    "payload": {
        "useCompactData": bool        //if omitted or false, compact mode is used.
                                        //otherwise returns json
        }
}
```

Unregister request

```
{
    "uniqueId": str,
    "action": "unregisterBeaconChangeStream"
}
```

Server sent event

```
{
    "action": "beaconDiffStream",
    "payload": msgpack str or json
   //responses are the same as in initial state
  //only changed values are sent, others omitted!
}
```

HTTP Push API

The usual way to get information from an external system is to request it from the server when it is needed. The HTTP API described previously in this document is an example of this. In many cases, especially when the interesting information or events are infrequent this is inefficient and adds latency as the client has to continuously poll the resource. Socket might also be overkill for sparse events for it requires maintaining a constant connection to the source. An alternative way is to invert the direction of the queries and let the data source *push* the updates to the client. This is what the push API, also referred to as *event API*, is for.

The basic operating principle of the API is as follows:

1. The client defines its own HTTP interface (single endpoint) that can receive updates from the cloud
2. Client registers this *endpoint* with the cloud
3. The endpoint is *validated* by the cloud
4. Client subscribes to the events it wants to receive
5. As events matching the client-provided criteria happen, they are sent in a HTTP request to the client

Note: Request examples in this section should be up-to-date if you have the most recent document at hand, but as they are also part of the HTTP API the generated documentation is also available online in the OpenAPI/Swagger document and related UI. They can also be used directly through the web-based UI so sending the requests manually is not required to test the API.

The following lists in more detail the different steps required to use the event API. Requests to unregister endpoints or event subscriptions and to query their details are not listed here. They can be found in HTTP documentation.

1. Creating an endpoint

To create an endpoint you need to give it a non-null *name* and have its *absolute URL* at hand. Create the endpoint by sending a HTTP **POST** request at **/eventapi/endpoint**. The body content type must be **application/json** with all requests in this section. The response code is **200** for a successful request and the response contains the generated ID for the endpoint that can then be used in subsequent queries. When an endpoint is created, it still needs to be registered/validated before it can be used.

Request

```
POST /eventapi/endpoint HTTP/1.1
Content-Type: application/json
Host: api.noccela.io

{
    "name": "MyEndpoint",
    "url": "https://api.myapp.com/endpoint"
}
```

Response

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "id": 123
}
```

2. Registering an endpoint

After the endpoint has been created and its ID received, one needs to register it. If one tries to register events for a non-validated endpoint it results in an error message. The registration is performed by sending a **POST** request from cloud to the specified client URL with *type* property value **"challenge"** and *challenge* property with a randomly generated token string. The endpoint must respond to this request with status code **200** and body set to the received challenge string (and nothing else, no JSON or other formatting). The content type would preferably be **text/plain** although it is not required. If the validation was a success the original request to cloud is fulfilled with no body and status code **204**. After this the endpoint is in *registered* state and now its ID can be used to subscribe to events.

Request

```
POST /eventapi/registerendpoint HTTP/1.1
Content-Type: application/json
Host: api.noccela.io

{
    "id": 123
}
```

Response

```
HTTP/1.1 204 No Content
```

Challenge request (from cloud to client endpoint)

```
POST /yourapiurl HTTP/1.1
Content-Type: application/json
Host: your.host

{
    "type": "challenge",
    "challenge": "AAABBB"
}
```

Challenge response (from client endpoint back to cloud)

```
HTTP/1.1 200 OK
Content-Type: text/plain

AAABBB
```

3. Registering events

Event subscriptions are referred to as *registrations* and also have their associated unique ID which can be used to refer to the specific registration without the endpoint ID. An event registration is linked to a single endpoint and has optionally event-specific parameters which can be used to control the event terms e.g. cooldown and tresholds. Two different subscriptions to the same event and same endpoint are considered individual subscriptions and have their own unique IDs, both have to be unregistered to stop the events. One can also register multiple events of same type for the same endpoint with different parameters e.g. to receive events for different devices. The request requires ID for the endpoint the events will be sent to, the *type* of the event as a string (refer to following section) and optionally the parameters of the subscription, depending on the event type.

Request

```
POST /eventapi/event HTTP/1.1
Content-Type: application/json
Host: api.noccela.io

{
    "endpointId": 123,
    "type": "RoomChange",
    "parameters: {}
}
```

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "id": 123
}
```

4. Receiving events

Once the event has been registered, whenever a matching event occurs in the cloud an event is dispatched as a HTTP request to the client's endpoint. This request should be responded with status code **200 OK** or **204 No Content** to acknowledge it. In case the event dispatch fails (request times out, invalid status code etc.) the cloud employs retry and circuit breaker patterns, so the event is resent for a given amount of times until it's discarded. If the client's system is offline the circuit breaker patterns defers any communication with the endpoint. This might result in some delay before receiving events after the client's system is back online.

The event messages all adhere to following pattern:

```
{
    "type": str,    // The type of the event as a string
    "ts": int64,    // Timestamp of the *message* as Unix timestamp
    "regId": int64, // The registration/subscription ID for the event
    "ev": [
        // Array of one or multiple events of same type.
        {
            "ts": int64, // Timestamp of the *event* as Unix timestamp
            "p": {}      // Payload of the event, containing event
                         // specific information
        }
    ]
}
```

Event types

This section lists all events available in the API, available parameters and the format of their payloads.

Room Change

This event is created when a target (tag) leaves or enters a given area. Areas can be created using the web-based site designer tool available for partners in the partner portal. Subscribing requires access to area ID which can be retrieved from HTTP endpoint for site layout. The sensibility of the event can be controlled with parameters.

Target is considered to enter or leave an area when they have stayed inside its boundaries (with an extra boundary of 0.5 meters) in such a way that both `locationCountTreshold` and `timeInAreaTreshold` are met. In normal conditions they should be fulfilled at the same time, but in case of positioning issues all of the target's locations might not be accepted. Location count condition exists for this reason, so that the state is not changed without sufficient data. Both or one of the filters can be disabled by setting them to 0.

Additional *cooldown* exists so that multiple events are not created in rapid succession. This is controlled by `cooldownSeconds` property and defines the time span in which the state cannot change and counters towards that such as previously described are not increased (the timer starts after the cooldown has ended). Cooldown concerns *entering* an area, not leaving it. Thus, an event for exiting the area can be created anytime after entering it. Cooldown is counted from the moment the event for *exiting* the area has been created. It can be disabled by setting it to 0.

The `timeSpent` in response is always calculated from the moment the target has entered the area, cooldown or not. The event's timestamp (`ts` property of the event object in array) reflects the moment the target has entered/exited the area, excluding the timer, cooldown and other possible wait periods. It always reflects the actual moment in time when the change happened.

Type: `RoomChange`

Parameters

```
{
    // Account ID for the site.
    "accountId": int64,

    // Site ID for the site.
    "siteId": int64,

    // Array of tag device IDs to track or null
    // for all tags at given site.
    "deviceIds": int64[]?,

    // Area ID of the area/room.
    "areaId": int64,

    // Seconds to wait before another event can be created.
    // DEFAULT: 60 seconds
    "cooldownSeconds": int64?,

    // How many seconds the target has to spend inside/outside the
    // area for change to be registered.
    // DEFAULT: 5 seconds
    "timeInAreaTreshold": int32?,

    // How many times the tag must be positioned inside the area.
    // This condition is combined with `timeInArea` and both must match.
    // DEFAULT: 5
    "locationCountTreshold": int32?
}
```

Payload

```
{
    // Device which triggered the event.
    "deviceId": int64,

    // The current status of the device
    // "inside" means tag has entered the area
    // "outside" means tag has left the area
    "status": "inside" | "outside",

    // How many seconds tag spent inside the
    // area in case it just left it.
    "timeSpent": int32?
}
```

Additional

The outer (root) message has an additional property `areaId` which is the area this event concerns.

Stroke Count

This event is created when tags stroke counter gets increased. Stroke counter has to be explicitly enabled and is mainly used in industrial environments.

Type: `StrokeCount`

Parameters

```
{
    // Account ID for the site.
    "accountId": int64,

    // Site ID for the site.
    "siteId": int64,

    // Array of tag device IDs to track or null
    // for all tags at given site.
    "deviceIds": int64[]?,

    // Event gets sent every time stroke count mod update frequency = 0.
    // DEFAULT: 1
    "updateFrequency": uint32,

}
```

Payload

```
{
    // Device which triggered the event.
    "deviceId": int64,

    // Amount of strokes
    "strokeCount": int64
}
```

Libraries

This section iterates available libraries that make it easier to integrate with Noccela systems.

HTTP API

It is advised to use a code generator to generate a client SDK for the HTTP API, as it will be always up-to-date and correct if you target the correct version of the API document. Noccela does not provide such libraries as of yet, but there exists a code generator targeting virtually every programming language in general use today.

https://swagger.io/tools/swagger-codegen/

Javascript WS API library

Noccela provides a JavaScript client library that abstracts away any complexity in the WebSocket API into a few easy-to-use functions. Maintaining the socket, reconnecting, filtering responses etc. are handled implicitly within the library. Using the library can make integration with JS apps hassle-free and the libraries will be updated to match the newest versions of the API.

Library can be included in browsers in minified form or imported as ES module with Webpack or NodeJs. Check repository README for further details.

The repository is available in the following URL, check the repository README for further details. Note that the repository is private and the user has to be added as guest by Noccela before accessing the SDK repositories. If you don't have account with access, please contact Noccela.

https://github.com/Noccela/ncc-cloud-integration-js

Appendix

Appendix A: Battery Statuses

```
{
    "Ok": 1,
    "Low": 2,
    "Empty": 3
}
```

Appendix B: Tag Statuses

```
{
    // Unknown state.
    "NotSupported": 0,

    // Tag was just loaded to site and not yet ready.
    "FetchingDetails": 1,

    // Tag is online but doesn't create alerts.
    "AlarmUnarmed": 2,

    // Tag is online and can create alerts. The "normal" state
    // for EAS tag.
    "AlarmArmed": 4,

    // Tag does not belong to the site it is currently physically present in.
    "ForeignTag": 5,

    // Tag is call tag.
    "AssistanceRequestTag": 7,

    // Tag is a "badge" tag.
    "BackupRequestTag": 8
}
```

Appendix C: Tag State Message

Following JSON contains the properties and explanations for both initial tag state and tag diff messages. Tag diff messages contain the changed values.

```
{
    // Tag's serial number. Used to identify it through the API
    // and also printed on its physical label.
    "deviceId": int64,

    // Name given to the tag.
    "name": string,
```

```
// Current battery voltage in mV.
"batteryVoltage": int16,

// State of the battery as in appendix A.
"batteryStatus": string,

// Tag's current status.
"tagStatus": "Tag's status as in appendix B.",

// Array of tag's current areas by their ID. Can be combined
// with layout information.
"areas": int32[],

// Boolean indicating whether the wire is connected between
// the two contacts.
"wire": boolean,

// Boolean indicating whether the reed sensor (magnet field)
// is activated.
"reed": boolean,

// Boolean indicating whether this is recent or historical
// information.
"isOnline": boolean,

// Status timestamp in ms since epoch.
"timestamp": int64,

// X coordinate.
"x": int32,

// Y coordinate.
"y": int32,

// Z coordinate.
"z": int32,

// Boolean indicating if tag's accelerometer is active
// i.e. it's moving.
"accelerometer": boolean,

// Id of the current floor the tag is in. Combine with
// layout data.
"floorId": int,

// Boolean indicating whether the tag's signal is lost
// i.e. it didn't go to sleep normally but disappeared.
"signalLost": boolean,

// Boolean indicating whether tag is going to sleep. True
// just before going to sleep and false when it awakes.
"powerSave": boolean,
```

```
    // Tags device model.
    "deviceModel": int32,

    // Tags firmware version.
    "fwVersion": string,

    // Tags stroke count. Used only if explicitly enabled.
    "strokeCount": int64,
}
```

Appendix D: Permission Levels

Following lists all permission levels that currently exists withing the context of API with short explanations.

SiteInformation Fetch data about site e.g. the layout.

SiteModifications Modify site-related data e.g. the layout or alert areas.

DeviceInformation Fetch data about a device (tags, beacons) e.g. model, type, location...

DeviceModifications Modify device data e.g. the name of a tag or blink LEDs etc.

HistoricalData Query historical data from databases.

Designer Access to site designer on partner portal.

EventApi Ability to create ”hooks” in the cloud.

RealTimeApi Connect to the WebSocket API for real-time updates.

InstallationTool Plan and calibrate beacon positions.

Appendix E: Socket API commands listing

Following lists all publicly available *actions* in socket API. More detailed explanations and JSON schemas can be found in preceding chapter.

`site`	Fetch site's layout.
`initialTagState`	Tags' most recent full state.
`registerTagLocation`	Register for updates to tags' coordinates.
`unregisterTagLocation`	Unregister from updates to tags' coordinates.
`registerTagDiffStream`	Register for incremental updates to tags' state.
`unregisterTagDiffStream`	Unregister from incremental updates to tags' state.
`initialAlertState`	Full alert states from a given time span.
`alert`	Tags' path leading to a given alert.
`registerAlertDiffStream`	Register for incremental updates to alerts' state.
`unregisterAlertDiffStream`	Unregister from incremental updates to alerts' state.
`initialContactTracingState`	Fetch historical contact tracing information.
`registerContactTracingStream`	Register for contact tracing events.
`unregisterContactTracingStream`	Unregister from contact tracing events.
`registerTwrStream`	Register for twr-distance updates.
`unregisterTwrStream`	Unregister from twr-distance updates.
`getBlueprint`	Fetch sites blueprint image data.
`sendSignal`	Send signaling request to signal device.
`playTagBuzzer`	Request tags to play their buzzer and/or led.

Appendix F: Device Models

```
{
    "Unknown tag": 512,
    "Noccela Dual2 Tag White": 518,
```

```
    "Noccela Dual2 Tag Black": 519,
    "Noccela Dual Tag v2b Safer": 520,
    "Noccela Dual2B Tag": 521,
    "Noccela Dual2B Badge Tag": 528,
    "Noccela Dual2B Call Button Tag": 529,
    "Noccela Badge Tag": 530,
    "Noccela Dual2B2 Tag": 531,
    "Noccela Call Button Tag": 532,
    "Noccela Badge 3A0 PU Black": 533,
    "Noccela Badge 3A1 PU Black": 534,
    "Noccela Badge 3A2 PU Black": 535,
    "Noccela Badge 3A1 U Black": 536,
    "Noccela Badge 3A2 U Black": 537,
    "Noccela Call Button Tag 2A0 White": 544,
    "Noccela Industrial Tag 0A0": 545,
    "Noccela Industrial Tag 0A0 LED": 546,
    "Noccela Industrial Tag 1A0": 547,
    "Noccela Industrial Tag 1A0 LED": 548,
    "Noccela Industrial Tag 1A1": 549,
    "Noccela Industrial Tag 1A1 LED": 550,
    "Noccela Sport Tag 0A0 Black": 551,
    "Noccela Sport Tag 0A0 White": 552,
    "Noccela Sport Tag 0A0 Flanges Black": 553,
    "Noccela Sport Tag 0A0 Flanges White": 560,
    "Noccela Sport Tag 1A0 Black": 561,
    "Noccela Sport Tag 1A0 White": 562,
    "Noccela Sport Tag 1A0 Flanges Black": 563,
    "Noccela Sport Tag 1A0 Flanges White": 564,
    "Noccela Sport Tag 1A1 Black": 565,
    "Noccela Sport Tag 1A1 White": 566,
    "Noccela Sport Tag 1A1 Flanges Black": 567,
    "Noccela Sport Tag 1A1 Flanges White": 568,
}
```