Parking APIs Introduction

Nwave provides four APIs to its clients. There are two main types of APIs – Push and Pull.

1. Data is retrieved by a client from the source in Pull APIs.
2. Data is sent by the source to a client in Push APIs.

Nwave's APIs are based on three technologies

1. HTTP
2. AMQP
3. GraphQL

Type	Technology	API	Advantages	Disadvantages	Typical applications
Push	HTTP	HTTP Caller	• Simple • Common • Real-time updates	• Limited performance at scale • Transformation of occupancy information to parking sessions is necessary • Unordered message delivery	• Raw sensor data transfer between backend systems • It is good for quick proof of concept demo integrations and tests
	AMQP	RabbitMQ RTA & Sessions	• Fast • Reliable • Scalable • Real-time updates • FIFO message delivery	• Requires setup and configuration of the RabbitMQ Server	• Robust message bus between high load backend systems • Commercial billing information / SDI
Pull	HTTP	REST Occupancy	• Simple • Fast response (<1s) • Quick setup	• Does not support real-time occupancy status updates	• Query-based method to get data when it is required, e.g. loading a page about a parking space, group or zone occupancy
		Parking Analytics API	• Simple integration • Flexible reports • Reports for the period of time • Response data is ready to showing on charts	• More complex than REST Occupancy API • Flexibility of reports leads to slower request processing	• Allows to build your own parking analytics dashbords with wide range of filtration and grouping abilities
Push & Pull	GraphQL	GraphQL Occupancy	• Flexible • Traffic-Efficient • Real-time updates	• Relatively new and less common	• Modern and large user-based web and mobile apps • Real-time dashboards

HTTP Caller

Diagram:

HTTP Caller API is one of the simplest types of API but provides little functionality to users. This API sends HTTP requests to your configured endpoints. HTTP requests are formed based on raw sensor events. During downtime, Nwave's cloud will make at up to 100 retries for each request which can lead to high traffic spikes.

More details about this API can be found here HTTP Caller.

RabbitMQ RTA & Sessions

Diagram:

Rabbit MQ is an enterprise-grade message bus that separates the application from transport layers. For example, RabbitMQ lets you set up messaging politics according to your preferences (e.g. you can configure a period and volume of message retention on a RabbitMQ server in case your service is offline).

Nwave provides two types of parking data through RabbitMQ:

RabbitMQ RTA (Real-Time Availability) gives you enriched parking occupancy information about every parking event. This lets you receive comprehensive application occupancy data for unmarked bays (when one car can occupy more than one sensor).	**RabbitMQ Parking Session Logging** saves your time and resources on developing and maintaining the code for storing occupancy history. RabbitMQ session is logging data which is already enriched by Nwave cloud (for marked and unmarked bays): session start, end time, session restoration in case of partial message loss and SDI data.

More details about these API can be found here:RabbitMQ.

REST Occupancy

REST Occupancy API is the API for retrieving real-time parking availability through simple HTTP requests.

This API provides a wide spectrum of query filters. You can use this API directly in your Web and Mobile Apps to:

- Display occupancy around a user on the map
- Recommend the nearest available parking spaces

More details about this API can be found here: REST Occupancy API.

GraphQL Occupancy

GraphQL Occupancy API provides the same functionality as Occupancy REST API (see above). But there are a few differences, which are very significant for Mobile Apps and other low-latency applications.

1. GraphQL API provides real-time occupancy updates on end-user devices. The application is able to subscribe to occupancy changes within a specified geospatial area.
2. GraphQL API allows you to request specific fields of an object which can significantly reduce traffic and increase speed.

More details about this API can be found here: GraphQL Occupancy API.

Overview

Introduction

To send data over HTTP, device messages have to be transformed into an HTTP request. The blueprint for transforming a device message into an HTTP request is known as a template. Sensors send different types of messages with different sets of fields, therefore a Template has to be defined for each message type. Templates are grouped together into Template Suites and in order for messages to be sent, template suites have to be connected to zones.

Template Suites

Template suites:

- Store a collection of templates
- Connect to zones with devices
- Used for monitoring HTTP calls

Template

A template is a blueprint for building an HTTP request for a single message type.

Each template has the following fields:

Field	Description
Name	A unique reference for a template in a template suite
Message Type	Type of message that this template will be applied to.
Method	HTTP method that will be used for in a request
URL	URL to which a request is made
Header	Request headers, typically used for authorization
Body	Request body defined as a JSON string

Authorization Methods

1. Basic HTTP(s) Authentication using authorization header in the format of "Basic " + Base64 (username: password)
2. API-Key

Message Types

Different message types have a set of common and distinct attributes that can be used in the template.

Message types have two main categories – **Direct** and **Smart**.

The main difference between these two categories is that Smart message types have a pre-defined template body that cannot be changed. The resulting message structure is identical to RabbitMQ protocols (RTA & Parking Sessions) and supports Parking Session messages that are crucial for unmarked parking bays and useful for payment applications. It also allows for an easier transition to RabbitMQ.

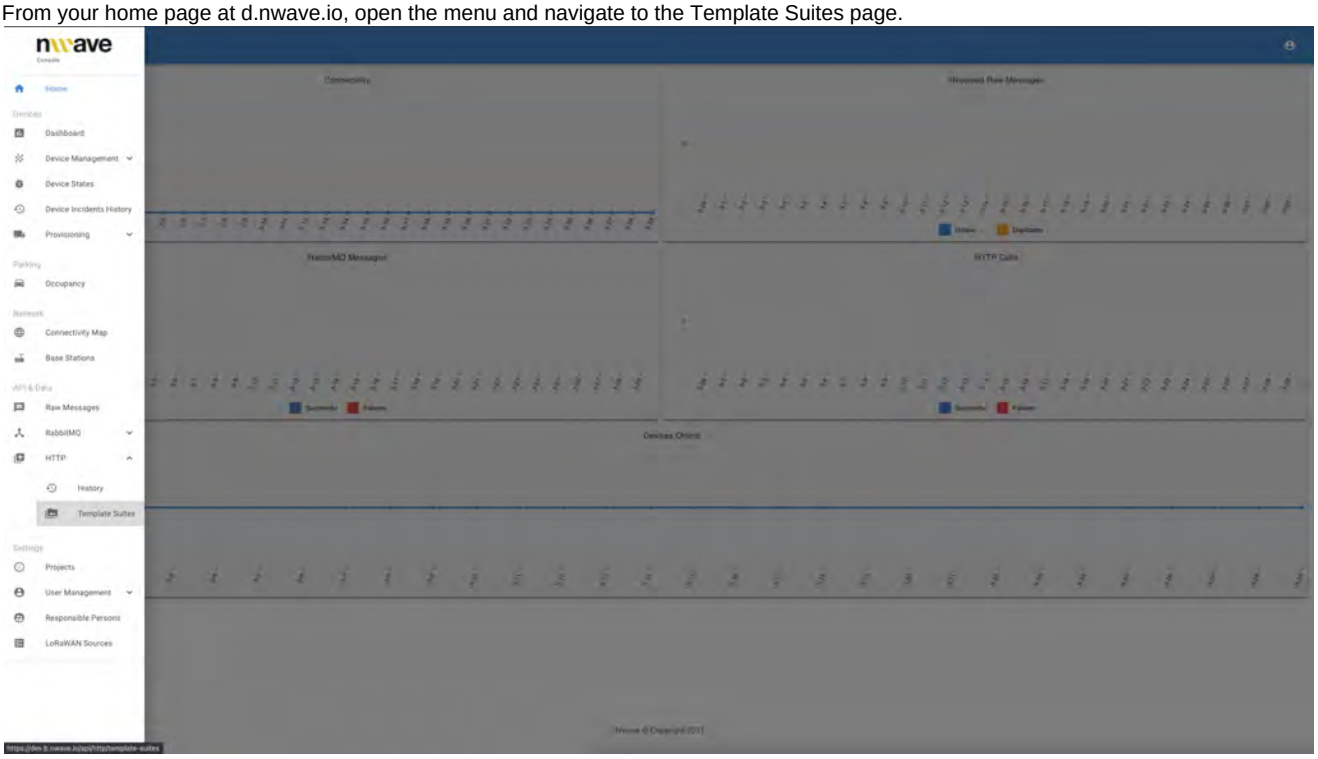
Category	Message Type	
Direct	Status Change	status change message is sent from a sensor when its value has been changed (occupancy state for parking sensor or number of detected cars for car counters
	Heartbeat	heartbeat message is sent after every constant time period of unchanging sensor state
	User Registration	user registration message is sent when a user is authorized on the sensor using a Bluetooth-tag
Smart	Group Availability	status change message in a group format with group summary

	Parking Sessions	smart parking session messages formed by Nwave’s backend, with auto-correction for lost messages

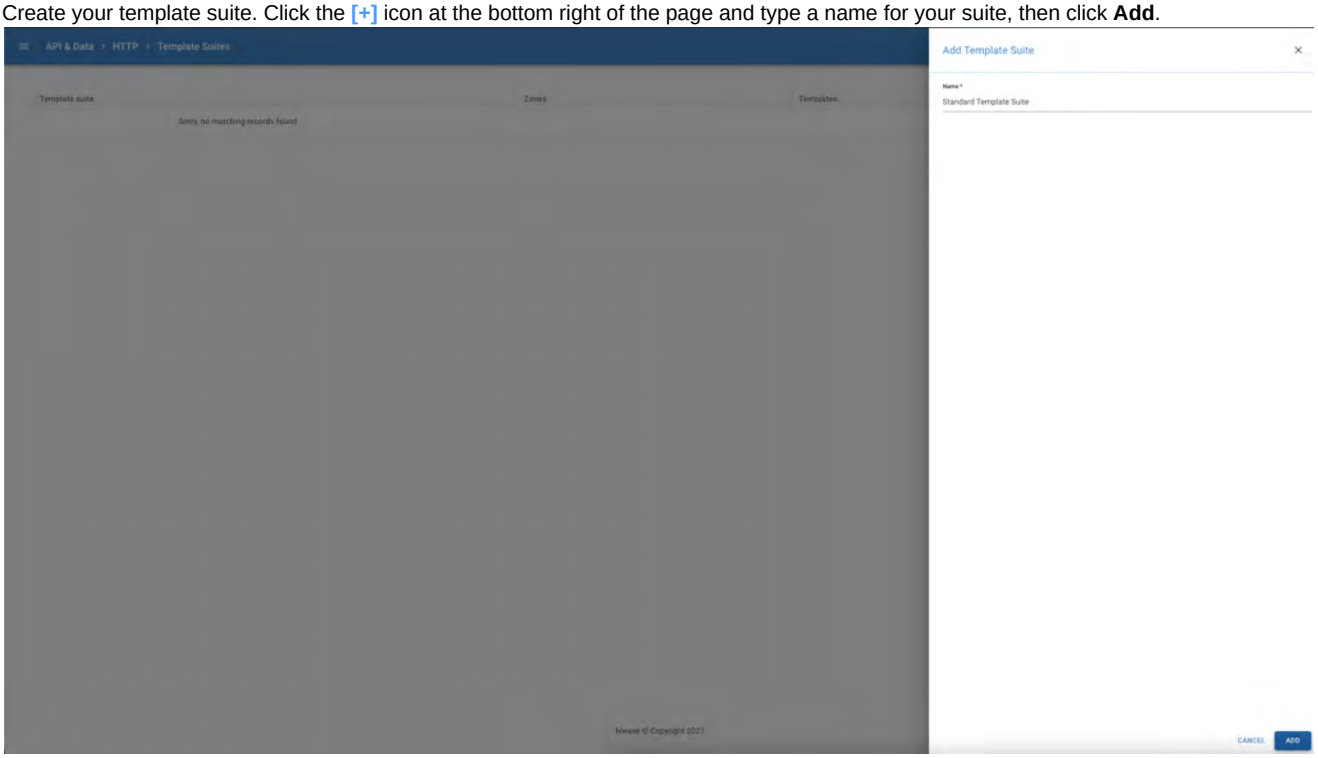
Quick Start Guide

To start receiving Status Change messages to your backend over HTTP, you need to perform the following steps:

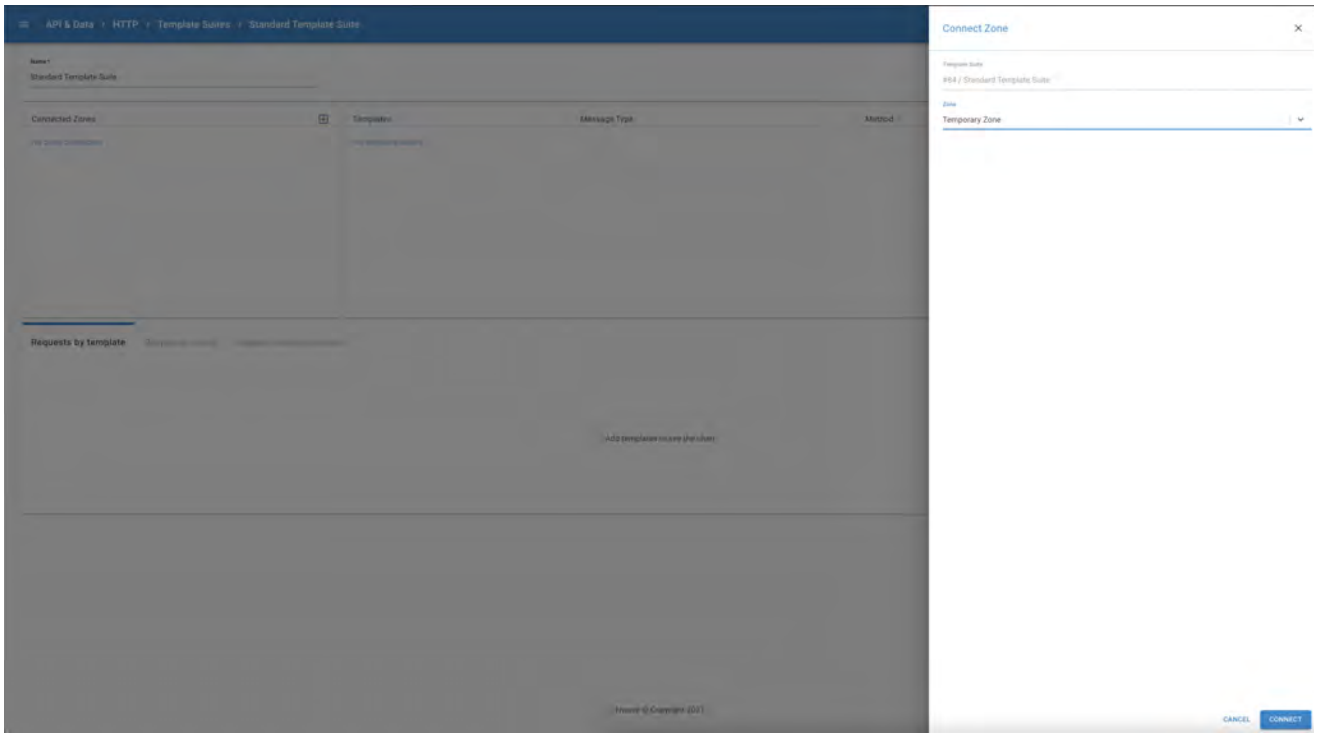
1. From your home page at d.nwave.io, open the menu and navigate to the Template Suites page.



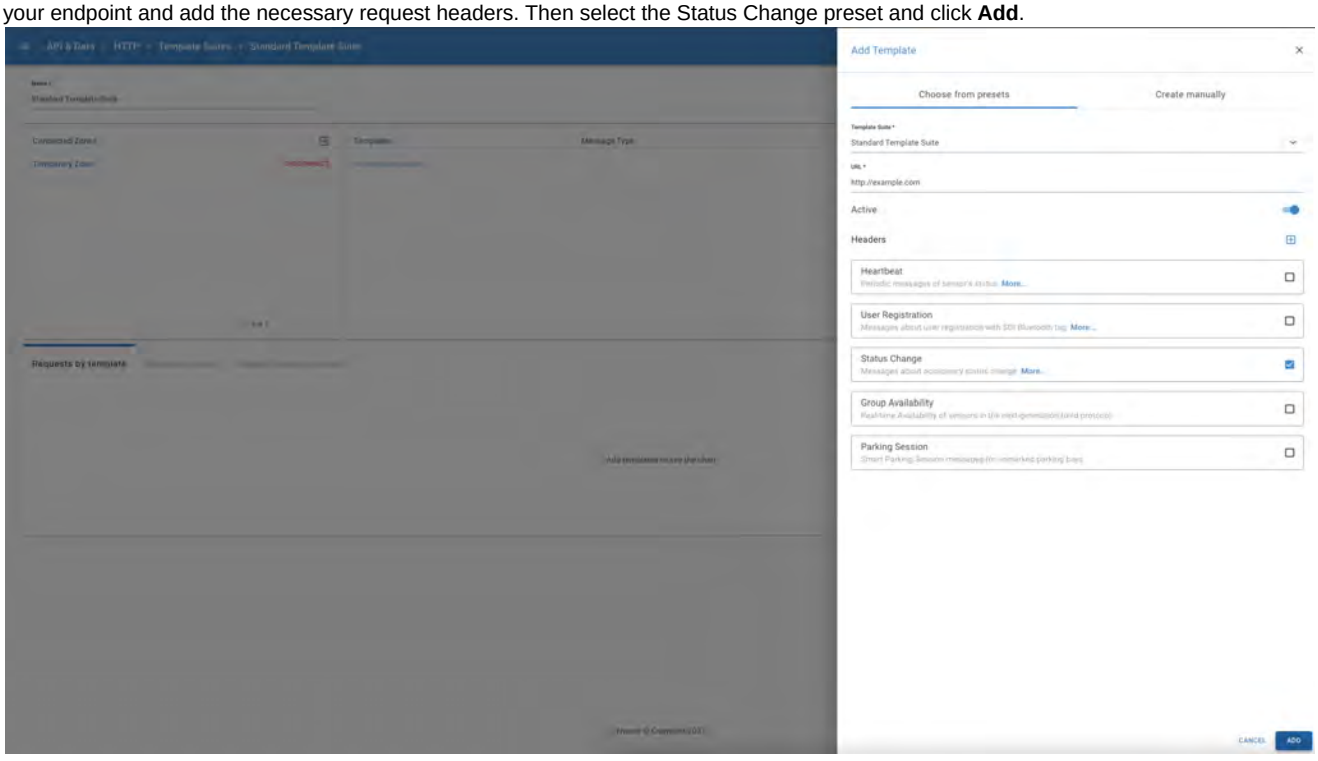
2. Create your template suite. Click the [+] icon at the bottom right of the page and type a name for your suite, then click **Add**.



3. Connect a zone from which you wish to receive messages. Click on the [+] icon in the Connected Zones section, select your zone and click **Connect**.



4. Add Status Change Template to the Suite. Click on the [+] icon in the Templates section, under Choose from presets tab, enter the URL of your endpoint and add the necessary request headers. Then select the Status Change preset and click **Add**.



You should start receiving messages at your specified endpoint. You can check the HTTP History Page for the status of HTTP calls and call history.

If you don’t have positioned sensors and a functional Base Station, you can use Virtual Devices for testing your HTTP Suites.

Standard Templates (Direct & Smart)

Direct vs Smart message types

Direct Message Types are actually raw messages received from sensors, which are formatted into JSON. You are able to configure a message of this types of processing scenario and set up your own HTTP-body format.

Smart message types have a pre-defined template body that cannot be changed. These messages are the results of the Nwave Parking Analytics system. They support the following features:

- Automatic parking session correction and parking session integrity in the event of message loss
- Occupancy event deduplication in the event of two occupancies (unmarked bays only)
- Filtering of extremely short parking sessions (unmarked bays only)
- Advanced handling of unmarked bay occupancies
- Easy transition to RabbitMQ protocols (RTA & Parking Sessions) due to identical structure

Smart message types are **recommended** for payment applications

Category	Message Type	Description
Direct	Status Change	status change message is sent from a sensor when its value has been changed (occupancy state for parking sensor or number of detected cars for car counters
	Heartbeat	heartbeat message is sent after every constant time period of unchanging sensor state
	User Registration	user registration message is sent when a user is authorized on the sensor using a Bluetooth-tag
Smart	Group Occupancy	status change message in a group format with group summary
	Parking Sessions	smart parking session messages formed by Nwave's backend, with auto-correction for lost messages

Preset Templates for Direct Message Types (Editable format)

These are template bodies of the template presets available at your console. You can create custom templates for each **direct** message type and the templates below can be used as a starting point.

Message Type	Template Body

Status Change

```
{
  "device_id": "{device_id}",
  "position":{
      "network_id": "{network_id}",
      "custom_id": "{custom_id}",
      "latitude": {latitude},
      "longitude": {longitude},
      "group_inner_id": {group_inner_id},
      "group":{
          "id": {group_id},
          "name": "{group_name}",
          "zone_id": {zone_id}
      }
  },
  "message_type": "status_change",
  "occupied": "{"occupied" if parsed
["occupation_status"] is True else "free"}",
  "previous_occupancy_status_duration_min": {parsed
["previous_occupancy_status_duration_min"]},
  "voltage_V": {parsed["voltage_V"]}
}
```

- **message_type** - message type;
- **message_trace_id** - system message-id;
- **occupied** - parking occupancy status ("occupied" or "free");
- **previous_status_duration_min** - duration of previous sensor status;
- **voltage_V** - device's voltage


```
{
  "device_id": "10000",
  "position":{
      "network_id": "908db095-e113-4248-998b-
694c33850bbe",
      "custom_id": "B03",
      "latitude": 1.01,
      "longitude": -3.732,
      "group_inner_id": 1,
      "group":{
          "id": 1,
          "name": "Group Name",
          "zone_id": 1
      }
  },
  "message_type": "status_change",
  "occupied": "occupied",
  "previous_occupancy_status_duration_min": 15,
  "voltage_V": 3.1
}
```

User Registration	{
	"device_id": "{device_id}",
	"position":{
	"network_id": "{network_id}",
	"custom_id": "{custom_id}",
	"latitude": {latitude},
	"longitude": {longitude},
	"group_inner_id": {group_inner_id},
	"group":{
	"id": {group_id},
	"name": "{group_name}",
	"zone_id": {zone_id}
	}
	},
	"message_type": "user_registration",
	"occupied": "{"occupied" if parsed
	["occupation_status"] is True else "free"}",
	"voltage_V": {parsed["voltage_V"]},
	"auth_ble_tag": {
	"tag_id": "{parsed["user_ID"]}",
	"event_time": "{message_time}"
	}
	}

- **message_type** - message type;
- **message_trace_id** - system message-id;
- **parking_session_iterator** - short serial number of parking session. Iterator (number) is incrementing when new parking session starts (0-7);
- **occupied**- parking occupancy status ("occupied" or "free");
- **voltage_V** - device's voltage;
- **auth_ble_tag**
  - **tag_id** - an ID of Bluetooth tag which was used for authorization;
  - **event_time** - message reception time.


```
{
  "device_id": "10000",
  "position":{
      "network_id": "908db095-e113-4248-998b-
694c33850bbe",
      "custom_id": "B03",
      "latitude": 1.01,
      "longitude": -3.732,
      "group_inner_id": 1,
      "group":{
          "id": 1,
          "name": "Group Name",
          "zone_id": 1
      }
  },
  "message_type": "user_registration",
  "occupied": "occupied",
  "voltage_V": 3.1,
  "auth_ble_tag": {
    "tag_id": "123ABC00",
    "event_time": "2021-01-01T00:00:00.000000+00:00"
  }
}
```

Heartbeat	<pre>{
	  "device_id": "{device_id}",
	  "position":{
	  "network_id": "{network_id}",
	  "custom_id": "{custom_id}",
	  "latitude": {latitude},
	  "longitude": {longitude},
	  "group_inner_id": {group_inner_id},
	  "group":{
	  "id": {group_id},
	  "name": "{group_name}",
	  "zone_id": {zone_id}
	  }
	  },
	  "message_type": "heartbeat",
	  "occupied": "{"occupied" if parsed
	["occupation_status"] is True else "free"}",
	  "heartbeat_message_counter": {parsed
	["heartbeat_message_counter"]},
	  "voltage_V": {parsed["voltage_V"]}
	}</pre>

- **message_type** - message type;
- **message_trace_id** - system message-id;
- **occupied** - parking occupancy status ("occupied" or "free");
- **heartbeat_message_counter** - the value increases for every following heartbeat during single occupancy state (0-11);
- **voltage_V** - device's voltage

```
{
  "device_id": "10000",
  "position":{
      "network_id": "908db095-e113-4248-998b-
694c33850bbe",
      "custom_id": "B03",
      "latitude": 1.01,
      "longitude": -3.732,
      "group_inner_id": 1,
      "group":{
          "id": 1,
          "name": "Group Name",
          "zone_id": 1
      }
  },
  "message_type": "heartbeat",
  "occupied": occupied",
  "heartbeat_message_counter": 2,
  "voltage_V": 3.1
}
```

Preset Templates for Smart Message Types (Fixed format)

These are examples of HTTP request bodies produced by smart templates. They **cannot be modified** but you can use the example requests as reference.

Message Type	Example Request Body
Group Availability	

```
{
   "timestamp": "2021-01-01T00:00:00.000000+00:00",
   "group_id": 1,
   "project_id": 1,
   "group_custom_id": "Group Custom ID",
   "level_id": null,
   "floor_number": null,
   "positions_availability":[
      {
         "position":{
            "id": 1,
            "network_id":"00000000-0000-0000-0000-
000000000001",
```

```
            "custom_id": "Custom ID 1",
            "group_inner_id": 1,
            "lat": 52.406063006389,
            "lon": -1.5157277658969
         },
         "occupation_status":"occupied"
      },
      {
         "position":{
            "id": 2,
            "network_id":"00000000-0000-0000-0000-
000000000002",
            "custom_id": "Custom ID 2",
            "group_inner_id": 2,
            "lat": 52.406022747069,
            "lon": -1.5157359155385
         },
         "occupation_status":"occupied"
      },
      {
         "position":{
            "id": 3,
            "network_id":"00000000-0000-0000-0000-
000000000003",
            "custom_id": "Custom ID 3",
            "group_inner_id": 3,
            "lat": 52.406101689254,
            "lon": -1.515721405201
         },
         "occupation_status":"n/a"
      }
   ],
   "summary":{
      "total": 3,
      "occupied": 2,
      "available": 0,
      "undefined": 1
   }
}
```


Parking Session


```
{
   "parking_session_uuid": "d8f7d4b3-a26c-4921-a488-
489de273bcf6",
   "involved_devices":[
      {
         "device_id": "100AA",
         "hardware_type": "Sparkit Surface V3.9",
```

```
         "position":{
            "network_id":"00000000-0000-0000-0000-
0000000100aa",
            "custom_id": null,
            "latitude": 52.40602,
            "longitude": -1.5157359,
            "group":{
               "id": 1,
               "type": "marked_spaces",
               "name": "Group Name 1",
               "custom_id": "Group Custom ID",
               "zone_id": 1,
               "level_id": null,
               "floor_number": null,
               "zone":{
                  "id": 1,
                  "project_id": 1
               }
            },
            "group_inner_id":1
         }
      }
   ],
   "correction_counter": 0,
   "session_start":{
      "event_time":"2021-01-01T00:00:00.000000+00:00",
      "delta_time_sec": 0,
      "message_trace_ids":[
         "d8cd1146-21f7-3906-21e4-8f55534f6573"
      ]
   },
  "partial_end":{
        "event_time": "string" // timestamptz, yyyy-MM-
dd'T'HH:mm:ss.SSSXXX (2019-06-13T16:16:51.000+00:00)
        "delta_time_sec": "integer",
        "message_trace_ids":["strings"],
        "network_id": "string", // unexpectedly
released position
        "custom_id": "string"
    },
   "session_end": {
      "event_time": "2021-01-01T01:00:00.000000+00:00",
      "delta_time_sec": 0,
      "message_trace_ids": [
          "71bfb22e-3569-d7c1-26fd-a2a0d0febb7a"
      ],
    },
    "auth_ble_tag": {
        "tag_id": "string",
        "event_time": "string"
```

```
    },
    "auth_mobile": {
        "session_id": "string",
        "event_time": "string"
    }
}
```

Custom Templates

Custom Templates are possible for Direct Message Types. You can read more about message types and standard templates here. If you need more data than provides a standard template, you are able to create your own template. For doing this you need to use a special Templating Language, which is being processed by Templating Engine.

Templating Engine

The HTTP template language is a language of transforming data based on the python 3 language syntax.

There are two main functions of a template language:

- getting the source data attribute values;
- operations over the source data attribute values.

For retrieving a value of any attribute from source data you need to use the source data attribute name surrounded by braces. This rule works for all fields except for **parsed data** fields**.** For retrieving any value from parsed data you need to use the following format: `{parsed['<field name>']}`.

Examples:

- `{modem_id}` - gets device ID hex;
- `{data}` - gets full raw message;
- `{message_time}` - gets a UNIX timestamp;
- `{parsed['voltage_V']}` - gets battery voltage information.

More complex example:

This URL template creates a URL that contains a position's network ID and occupancy status:

```
http://example.com/parking?id={network_id}&occupied={parsed['occupation_status']}
```

Nwave supports some operations over the source data values if pure values are not usable or can be converted to a more convenient format.

Operations:

- IF-THEN-ELSE operator in Python style `{<result if true> if <condition> else <result if false>}`;
- `pow(a, b)` - returns a value of x to the ***power*** of y (x^y);
- `str(obj)` - stringifies an object;
- `hex(int)` - returns a **hexadecimal** representation of an integer value;
- `int(str, base)` - returns an integer representation of a string;
- `len(str)` - returns the length of a string argument;
- `iso8601(ts)` - returns ISO8601 representation of a timestamp;
- `<obj>[index]` - accesses string characters by index;
- `obj[a:b:c]` - slice operator which works the same as the Pyhton language;
- `+, I, *, /, *, //, **, %` - standard Python language operators.

Example:

This body example returns JSON which contains a position network id, occupation status in the format "occupied"/"free" and message receiving timestamp in ISO8601 format:

```
{
  "timestamp": "{iso8601(message_time)}",
  "space_network_address": "{network_id}",
  "new_status": "{"occupied" if parsed["occupation_status"] is True
else "free"}"
}
```

Using the template functionality, you can even create custom formatting request, where the format is changed depends on source data values. The following snippet shows how to create an empty request if a raw message is too short, but if the data is too long, the template will add a message payload tail as a query argument:

```
http://example.com{(('?tail=%s' % data[6:]) if (len(data) >= 6) else '')}
```

You can use templates in URLs and request bodies.

Common Attributes

The following attributes are available for all message types:

String attributes should be wrapped in quotes.

Attribute	Template Usage	Type	Description
message_trace_id	`"{message_trace_id}"`	str	message trace id UUID format
message_time	`"{message_time}"`	str	time of receiving a message from a base station
received_time	`"{received_time}"`	str	time of receiving a message by the Nwave cloud
device_id	`"{device_id}"`	str	device ID in hex format
device_id_dec	`{device_id_dec}`	int	device ID in decimal format

signal	`{signal}`	float	message signal level
data	`"{data}"`	str	raw message payload in hex format
station_id	`{station_id}`	int	ID of a station that got the message
custom_id	`"{custom_id}"`	str	bound position custom ID or '?' if a custom ID was not set
latitude	`{latitude}`	float	latitude parameter of a position
longitude	`{longitude}`	float	longitude parameter of a position
floor_number	`{floor_number}`	int	position's floor number
network_id	`"{network_id}"`	str	network ID of a position
zone_id	`{zone_id}`	int	decimal zone ID of a position
group_id	`{group_id}`	int	decimal group ID of a position

group_name	`"{group_name}"`	str	group name which a position belongs to
group_inner_id	`{group_inner_id}`	int	inner ID of a position in a group

Distinct Attributes

Attributes that are different for each message type are accessible under the **parsed** key.

Message Type	Attribute	Template Usage	Type	Description
Status Change	voltage_V	`{parsed["voltage_V"]}`	float	device's battery voltage
	parking_session_iterator	`{parsed["parking_session_iterator"]}`	int	a number which is incremented after every occupancy Not available for LoRa sensors
	occupation_status	`{parsed["occupation_status"]}`	boolean	"true" if a sensor is occupied, otherwise - "false"
	previous_occupancy_status_duration_min	`{parsed["previous_occupancy_status_duration_min"]}`	int	duration of a sensor previous state, this field can help if the previous message was not received by any station
Heartbeat	voltage_V	`{parsed["voltage_V"]}`	float	device's battery voltage

	parking_session_iterator	`{parsed ["parking_session_iterator"]}`	int	a number which is incremented after every occupancy Not available for LoRa sensors
	occupation_status	`{parsed ["occupation_status"]}`	boolean	“true” if a sensor is occupied, otherwise - “false”
	heartbeat_message_counter	`{parsed ["heartbeat_message_counter"]}`	int	number of heartbeat messages sent during an unchanged occupancy state
User Registration	voltage_V	`{parsed ["voltage_V"]}`	float	device’s battery voltage
	parking_session_iterator	`{parsed ["parking_session_iterator"]}`	int	a number which is incremented after every occupancy Not available for LoRa sensors
	occupation_status	`{parsed ["occupation_status"]}`	boolean	“true” if a sensor is occupied, otherwise - “false”
	user_ID	`"{parsed ["user_ID"]}"`	str	an ID of Bluetooth tag which was used for authorization

HTTP Templates and Suites

HTTP Templates

HTTP Template is a rule of building an outgoing HTTP request. A template specified a rule of building request method, URL, HTTP headers, and request body. A template is applied to an incoming message from devices, which has been parsed and enriched by information about the device. For the convenient building of templates, Nwave created a special template language which will be explained further in this document.

HTTP template may be applied to all messages coming from a device or only to one type of message. Nwave supports the following messages types:

- **All** - apply a template to all types of messages;
- **Status Change** - status change message is sent from a sensor when its value has been changed (occupancy state for parking sensor or number of detected cars for car counters);
- **Heartbeat** - heartbeat message is sent after every constant time period of unchanging sensor state;
- **Calibration** - this type of message is sent after a sensor calibration performed using Nwave mobile apps;
- **User Registration** - user registration message is sent when a user is authorized on the sensor using a Bluetooth-tag;
- **Error** - this type of message is sent only by car counters when some hardware, software or environment problem has been detected by a sensor;
- **General** - general type is used for data coming from devices flashed by custom firmware.

HTTP request source data

HTTP request service builds requests based on a user-defined template and incoming data which contains the following fields:

- **message_trace_id** - a message trace id;
- **message_time** - a time of receiving a message from a base station;
- **received_time** - a time of receiving a message by the Nwave cloud;
- **device_id** - a device ID in hex format;
- **device_id_dec** - a device ID in decimal format;
- **signal** - a message signal level;
- **data** - a raw message payload in hex format;
- **station_id** - an ID of a station that got the message;
- **custom_id** - a bound position custom ID or '?' if a custom ID was not set;
- **latitude** - a latitude parameter of a position;
- **longitude** - a longitude parameter of a position;
- **level** - position's level (e.g. ground, 1st);
- **network_id** - a network ID of a position;
- **zone_id** - a decimal zone ID of a position;
- **group_id** - a decimal group ID of a position;
- **group_name** - a group name which a position belongs to;
- **group_inner_id** - an inner ID of a position in a group;
- **parsed** - a parsed message. It contains different data, depending on device type, firmware type, and message type.

Parking sensor parsed message objects

Status Change

- **voltage_V** - device’s battery voltage;
- **parking_session_iterator** - a number which is incremented after every occupancy;
- **occupation_status** - “true” if a sensor is occupied, otherwise - “false”;
- **previous_occupancy_status_duration_min** - duration of a sensor previous state, this field can help if the previous message was not received by any station.

Heartbeat

- **voltage_V** - device’s battery voltage;
- **parking_session_iterator** - a number which is incremented after every occupancy;
- **occupation_status** - “true” if a sensor is occupied, otherwise - “false”;
- **heartbeat_message_counter** - number of heartbeat messages sent during an unchanged occupancy state.

Calibration

- **voltage_V** - device’s battery voltage;
- **hardware_reset** - “true” if a message caused by a device’s periodical self-reset;
- **noisy_environment** - “true” if an environment is too noisy for the calibration process

User Registration

- **voltage_V** - device's battery voltage;
- **parking_session_iterator** - a number which is incremented after every occupancy;
- **occupation_status** - "true" if a sensor is occupied, otherwise - "false";
- **user_ID** - an ID of Bluetooth tag which was used for authorization.

Car Counter parsed message object

Status Change

- **voltage_V** - device's battery voltage;
- **car_counter** - number of detected cars.

Error

- **voltage_V** - device's battery voltage;
- **first_calibration** - "true" if a message is caused by calibration via mobile app;
- **magnetometer_calibration_error** - an error while magnetometer calibration or self-calibration;
- **magnetometer_calibration_timeout** - "true" if hardware error occured;
- **reading_error** - "true" if there are some problems with onboard sensor's data reading;
- **sensor_power_supply_too_low** - "true" if battery level is too low;
- **hardware_reset** - "true" if a message caused by a device periodical self-reset.

HTTP template language

The HTTP template language is a language of transforming data based on the python 3 language syntax.

There are two main functions of a template language:

- getting the source data field values;
- operations over the source data field values.

For retrieving a value of any field from source data you need to use source data field name surrounded by braces. This rule works for all fields except for **parsed data** fields**.** For retrieving any value from parsed data you need to use the following format: `{parsed['<field name>']}`.

Examples:

- `{modem_id}` - gets device ID hex;
- `{data}` - gets full raw message;
- `{message_time}` - gets a UNIX timestamp;
- `{parsed['voltage_V']}` - gets battery voltage information.

More complex example:

This URL template creates a URL that contains a position's network ID and occupancy status:

```
http://example.com/parking?id={network_id}&occupied={parsed['occupation_status']}
```

Nwave supports some operations over the source data values if pure values are not usable or can be converted to a more convenient format.

Operations:

- IF-THEN-ELSE operator in Python style `{<result if true> if <condition> else <result if false>}`;
- `pow(a, b)` - returns a value of x to the ***power*** of y (x^y);
- `str(obj)` - stringifies an object;
- `hex(int)` - returns a **hexadecimal** representation of an integer value;
- `int(str, base)` - returns an integer representation of a string;
- `len(str)` - returns the length of a string argument;
- `iso8601(ts)` - returns ISO8601 representation of a timestamp;
- `<obj>[index]` - accesses string characters by index;
- `obj[a:b:c]` - slice operator which works the same as the Pyhton language;
- `+, I, *, /, *, //, **, %` - standard Python language operators.

Example:

This body example returns JSON which contains a position network id, occupation status in the format “occupied”/”free” and message receiving timestamp in ISO8601 format:

```
{
  "timestamp": "{iso8601(message_time)}",
  "space_network_address": "{network_id}",
  "new_status": "{"occupied" if parsed["occupation_status"] is True
else "free"}"
}
```

Using the template functionality, you can even create custom formatting request, where the format is changed depends on source data values. The following snippet shows how to create an empty request if a raw message is too short, but if the data is too long, the template will add a message payload tail as a query argument:

```
http://example.com{(('?tail=%s' % data[6:]) if (len(data) >= 6) else '')}
```

You can use templates in URLs and request bodies.

HTTP Template suites

HTTP template suites are used for uniting HTTP templates, which are applied to different (or the same) message types. Also, HTTP Template Suites are data routing entities. You can configure data routing from a device’s zone or a group only to an HTTP Template Suite, but not to an individual HTTP Template.

Rabbit MQ Broker AWS Setup

This is a step-by-step guide on how to set up a RabbitMQ broker in the Amazon MQ service.

However, you can also choose to set up and manage the RabbitMQ broker on your own server.

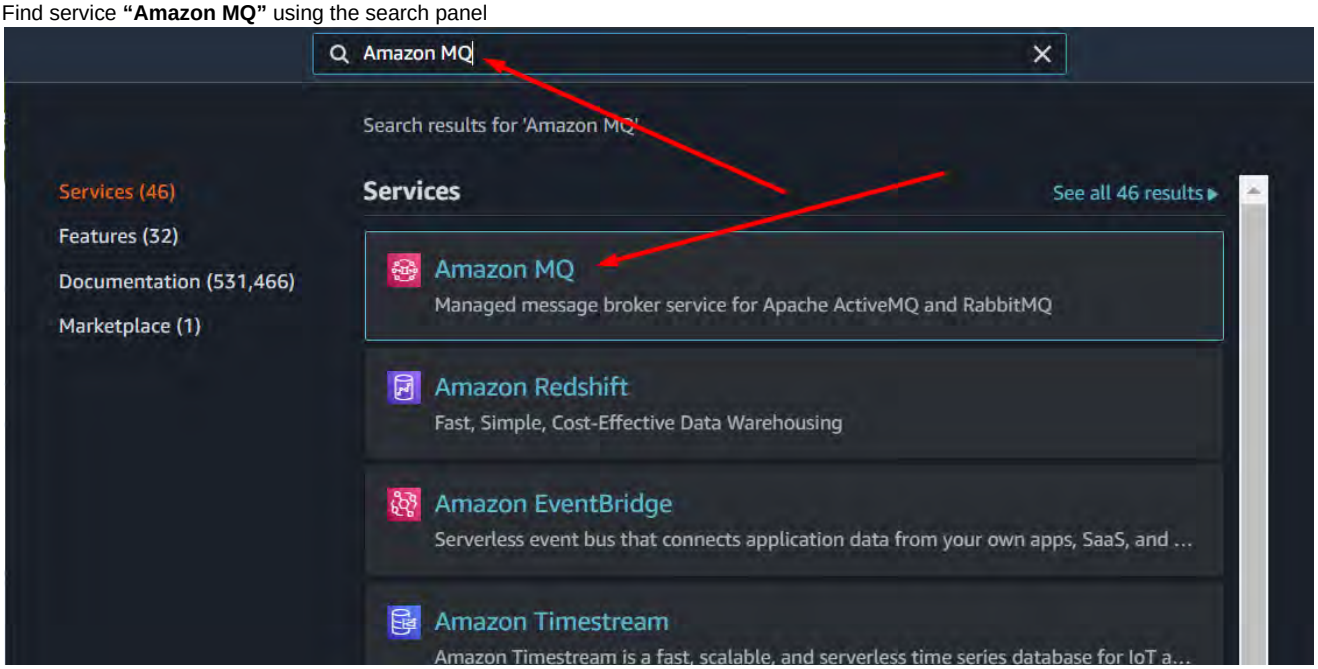
Detailed information about the Amazon MQ service can be found in the official AWS documentation.

The configuration of a RabbitMQ broker consists of 2 steps:

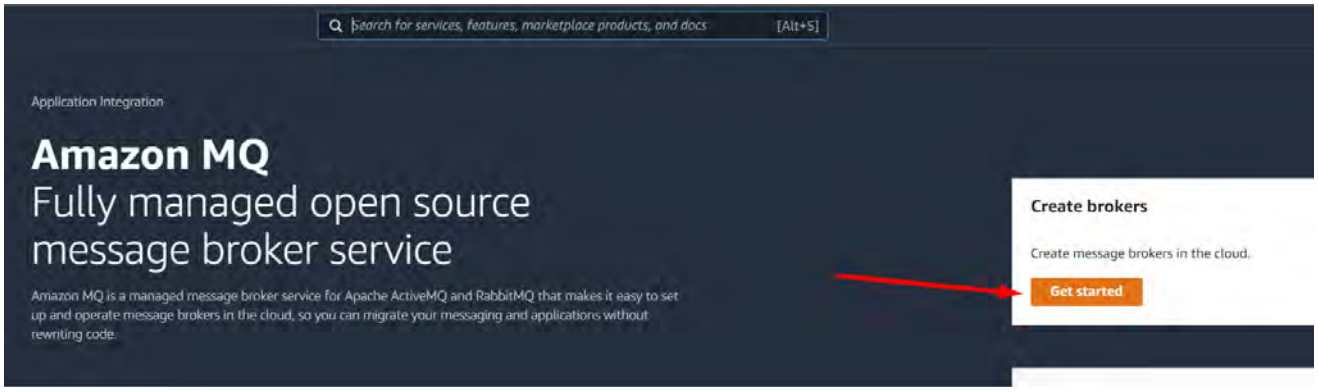
1. Broker creation
2. Broker configuration

Broker creation

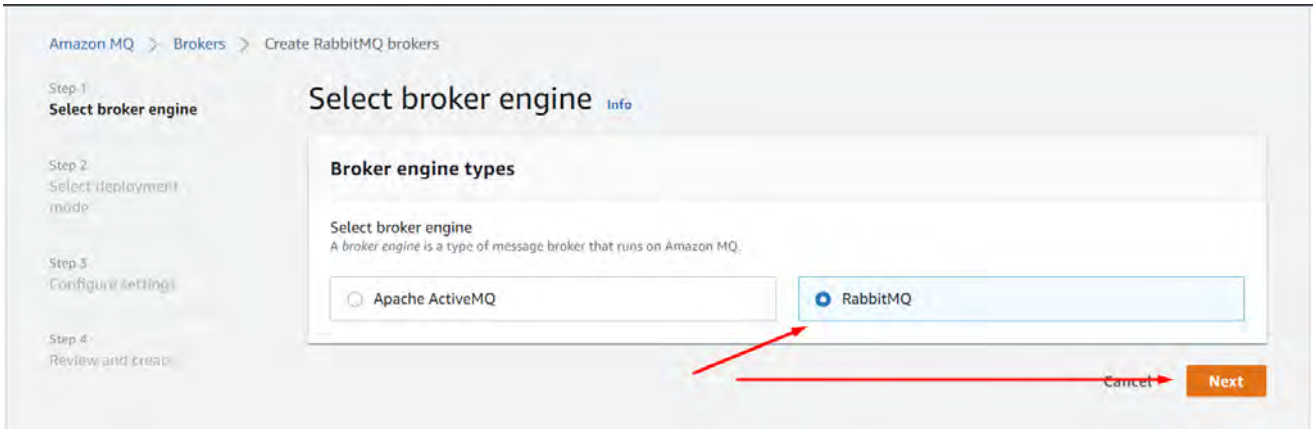
1. Find service **“Amazon MQ”** using the search panel



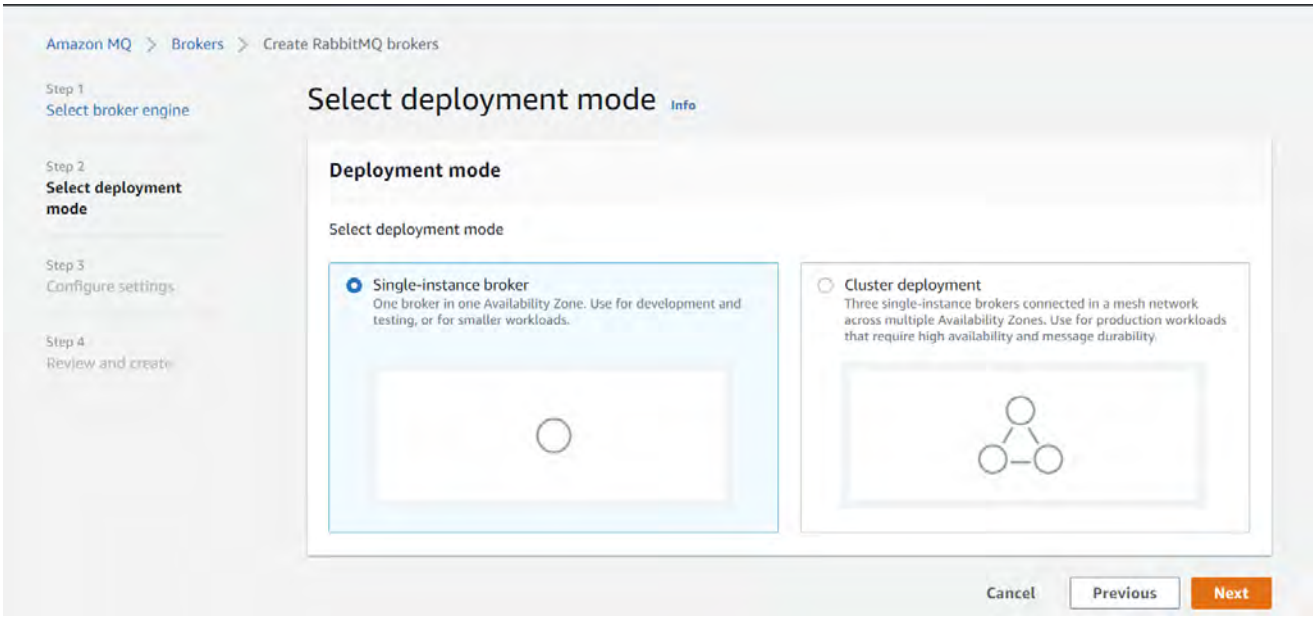
2. Click on the **“Get started”** button



3. Select “Rabbit MQ” engine and click **“Next”**



4. For simplicity choose the **single-instance broker**. If you wish to set up a more reliable cluster deployment please refer to the official documentation.

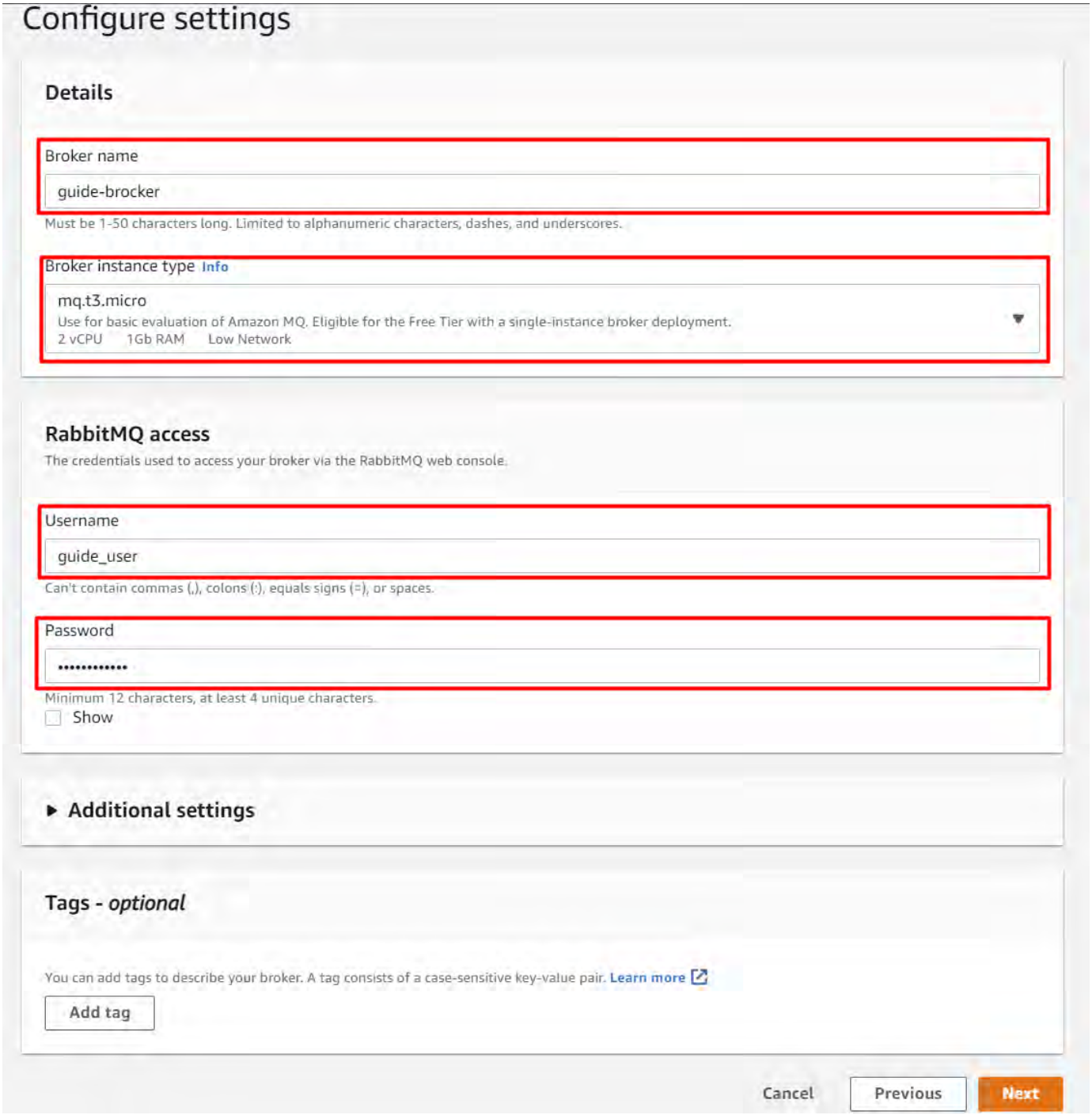


5. Configure the settings.

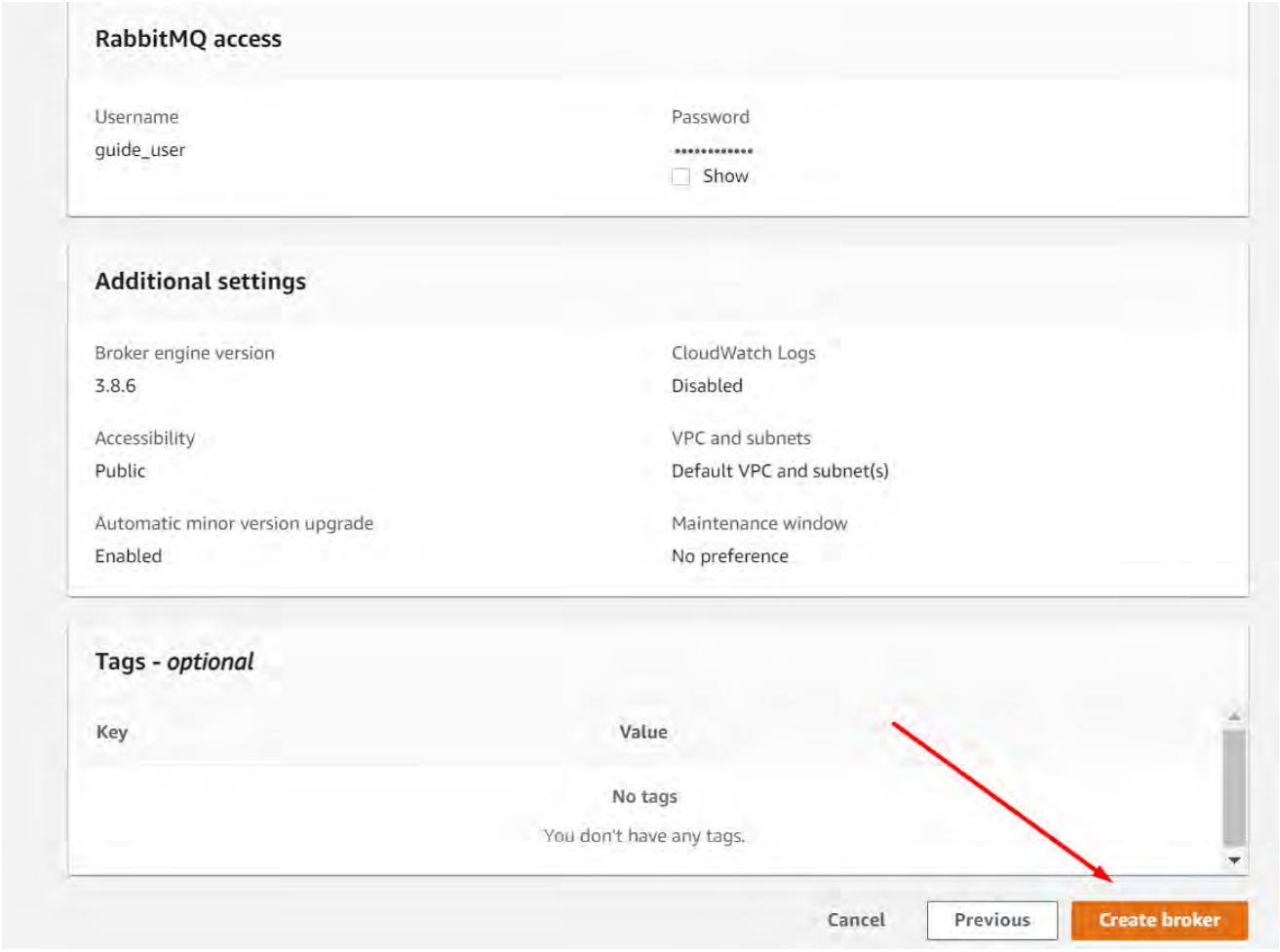
Brocker name	guide-broker	broker identifier in the Amazon MQ service
Brocker instance type	mq.t3.micro	You can select the smallest instance for now Larger instances will be required for over 100,000 devices
Username	guide_user	
Password	<your password>	

Please **save** your username and password as it will be required for broker configuration and integration.

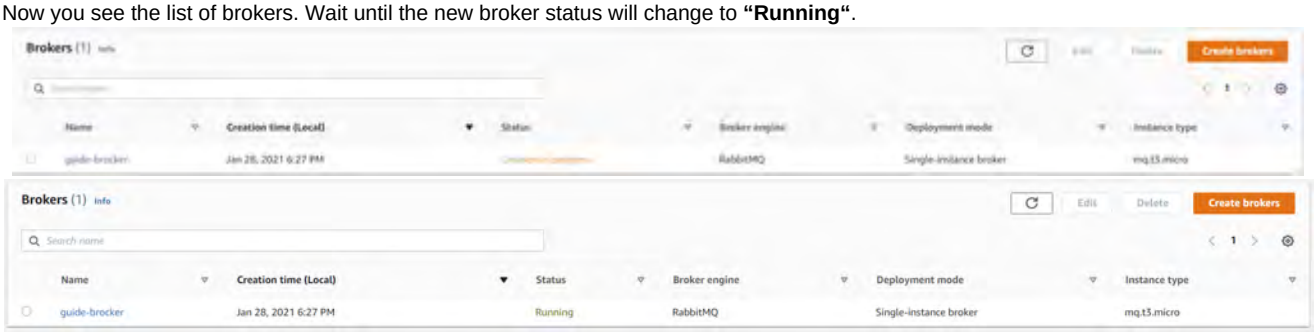
Fill out the form and click **"Next"**:



1. You can review your broker configuration on the last screen. Click **"Create broker"** to finalize the creation procedure.



2. Now you see the list of brokers. Wait until the new broker status will change to **"Running"**.

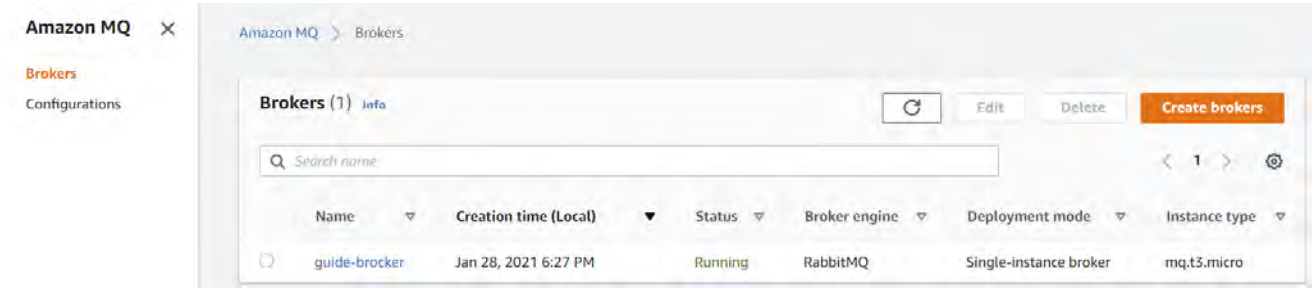


The broker creation is complete. Now you can proceed to the next step.

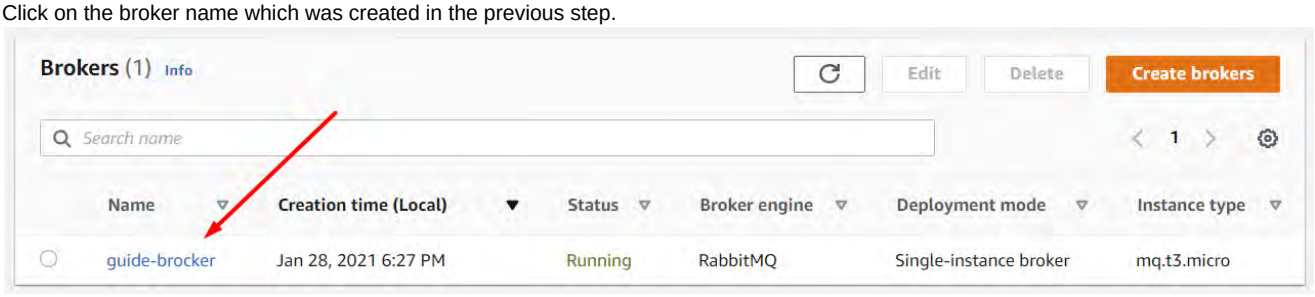
Broker configuration

The broker configuration step explains how to configure RabbitMQ Queue, RabbitMQ Exchange and bind the queue to the exchange.

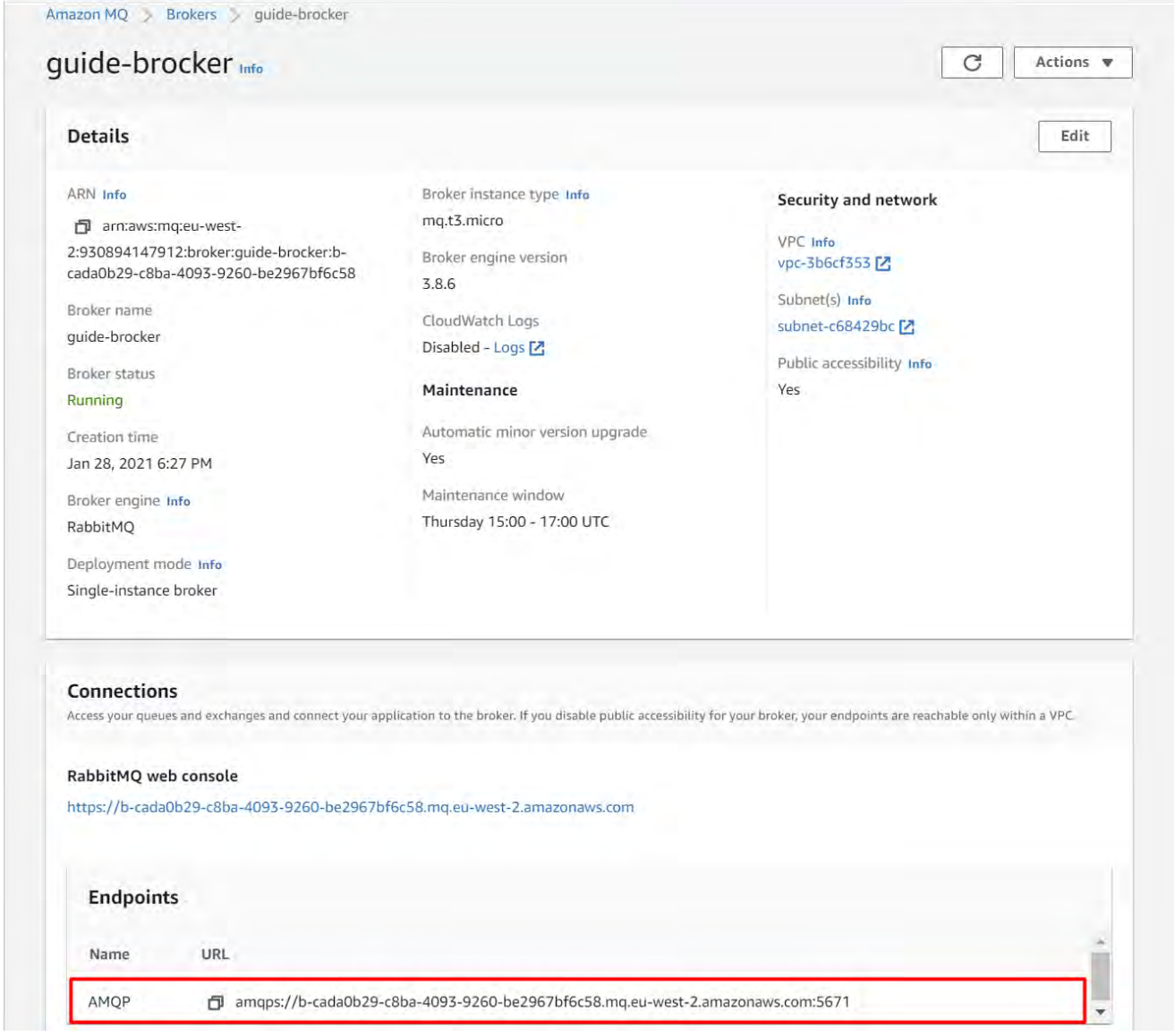
1. Go to Amazon MQ Brockers page



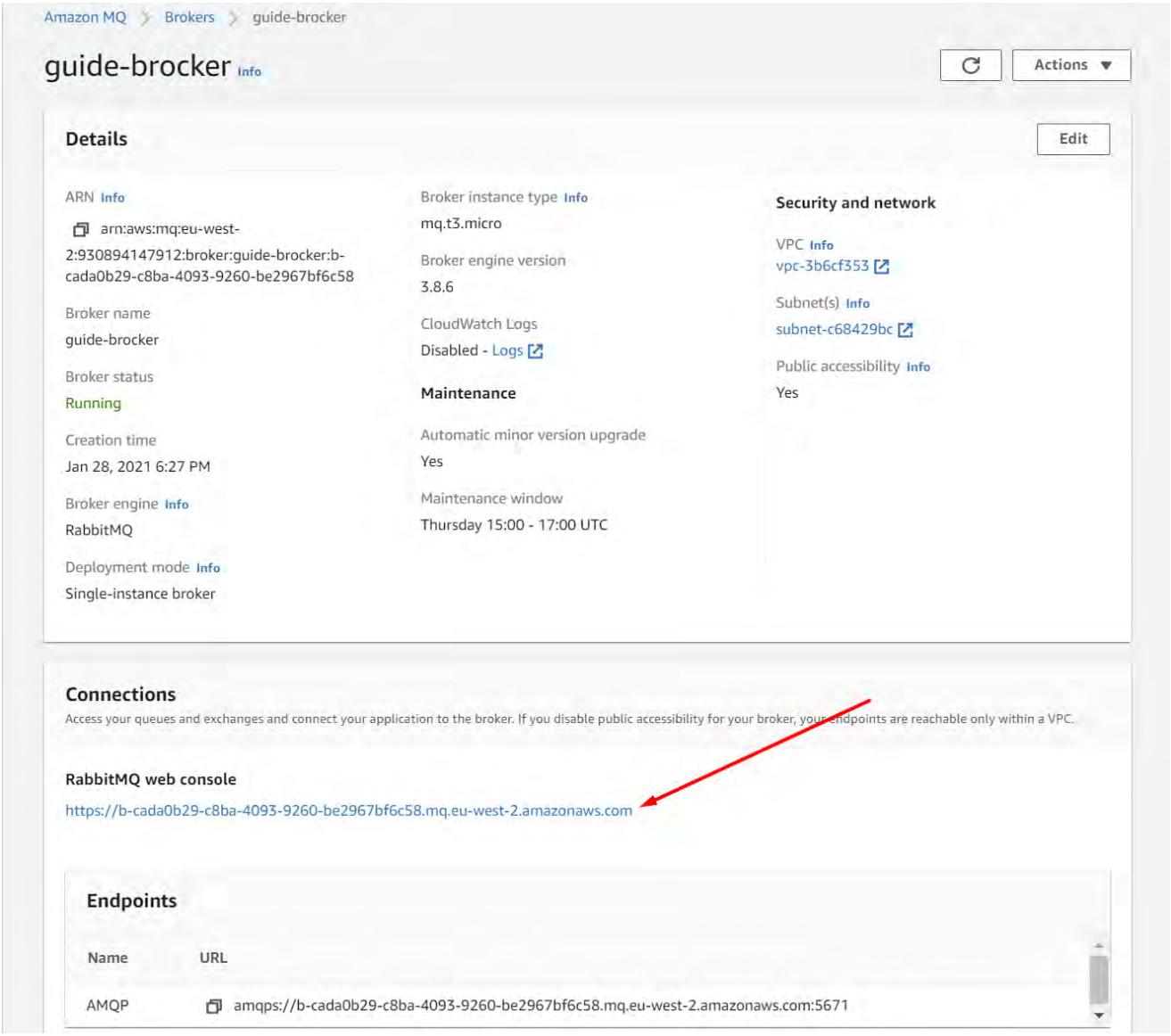
2. Click on the broker name which was created in the previous step.



3. In the Connections section, you can find the **Endpoint URL**. You will need this URL to configure the integration Nwave.

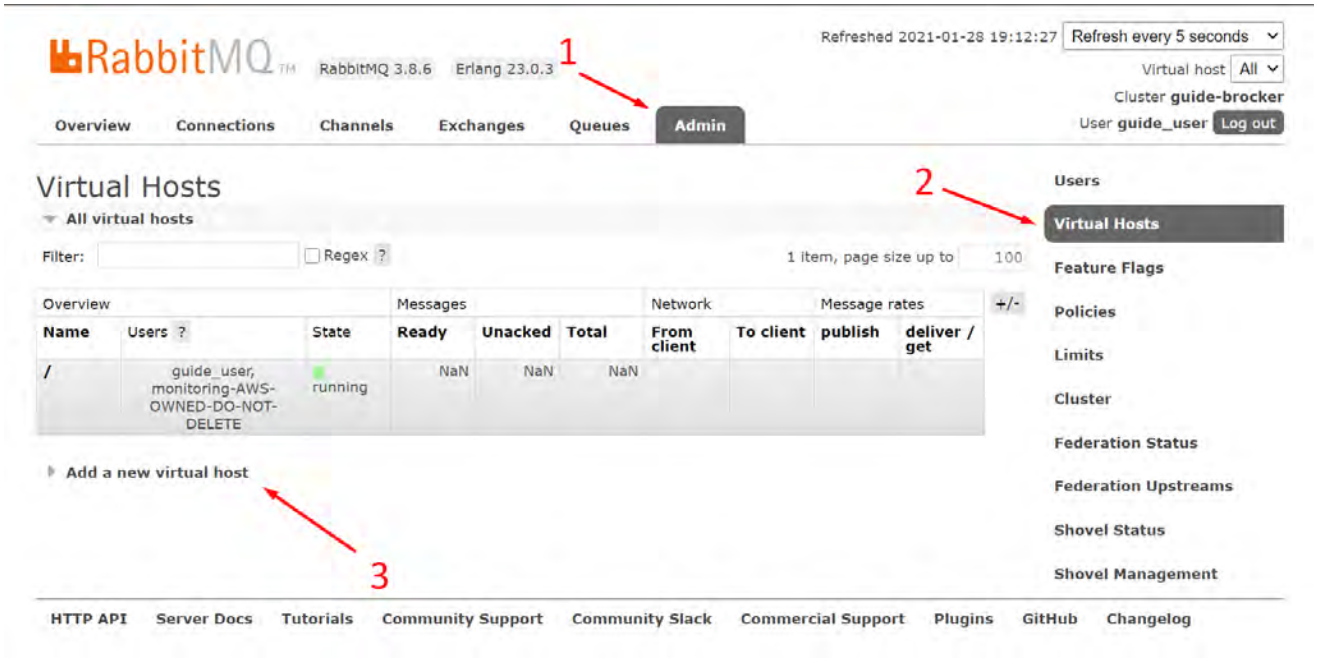


4. Click on the “RabbitMQ web console” link to continue broker configuration

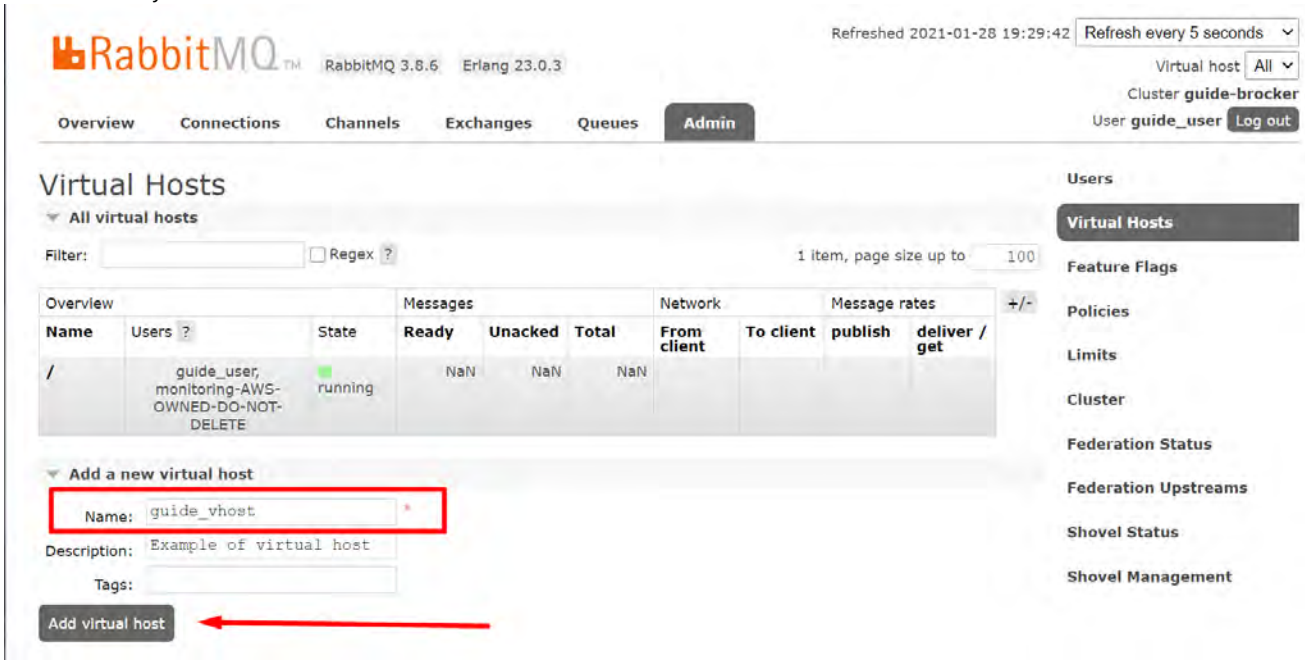


5. In the newly opened tab, you must enter the username and password used during broker creation in the previous step

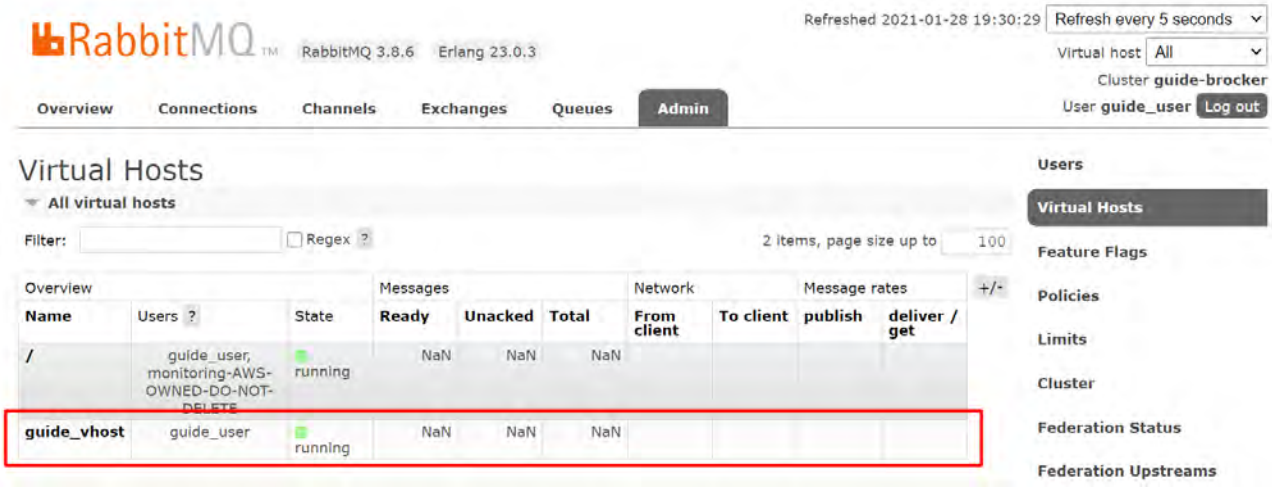
6. Now you have to register a new Virtual Host. Go to **“Admin Page”,** select **“Virtual Hosts”** menu on the right and click on expanding form **“Add a new virtual host”**



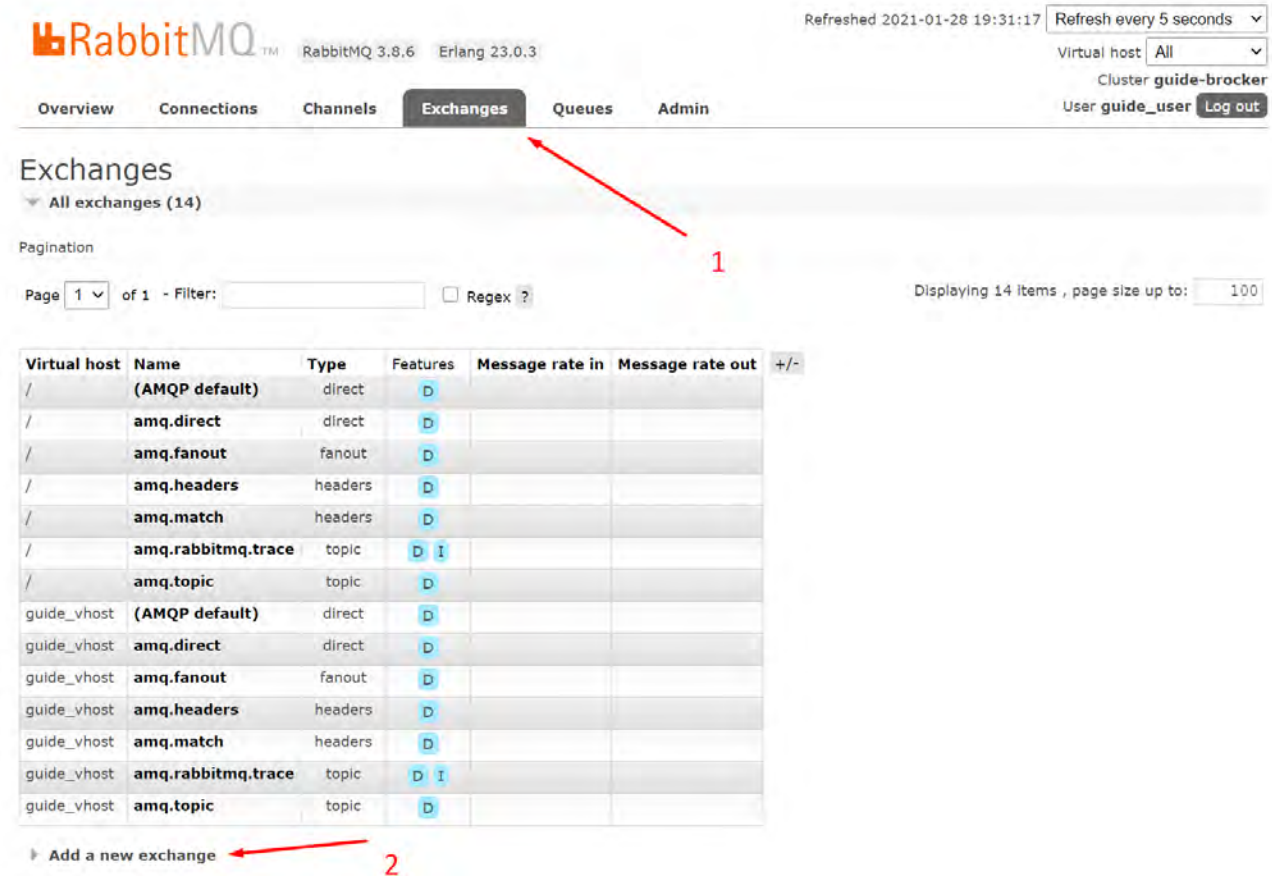
7. Give a name to your new virtual host and click on the **"Add virtual host"** button.



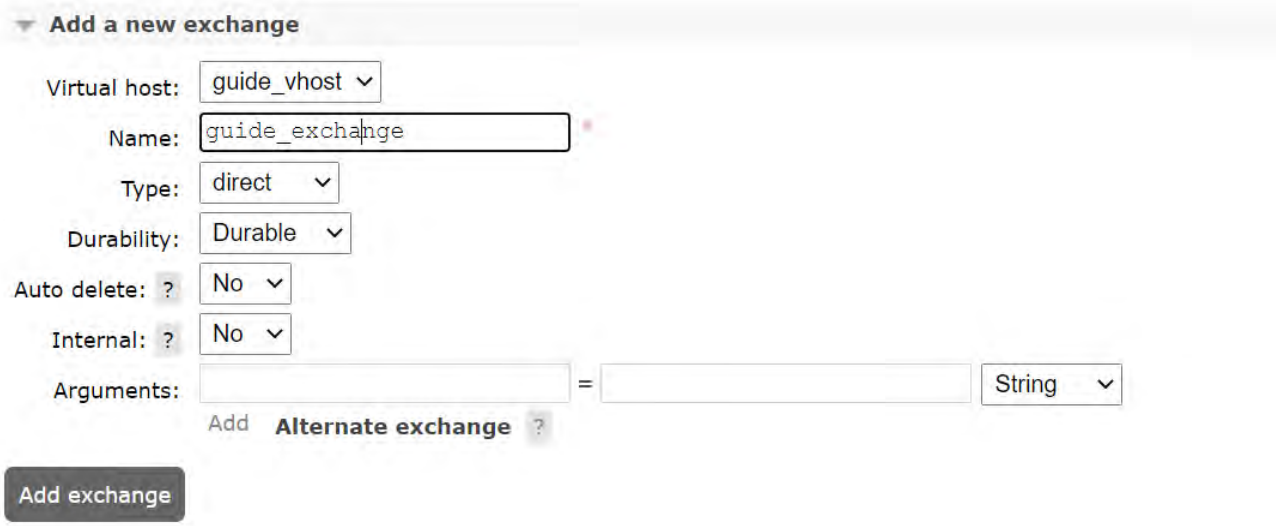
8. Now you can see the create virtual host in the table



9. Now you have to create a new exchange. Go to the **"Exchanges"** menu and click on the expanding form **"Add a new exchange"**



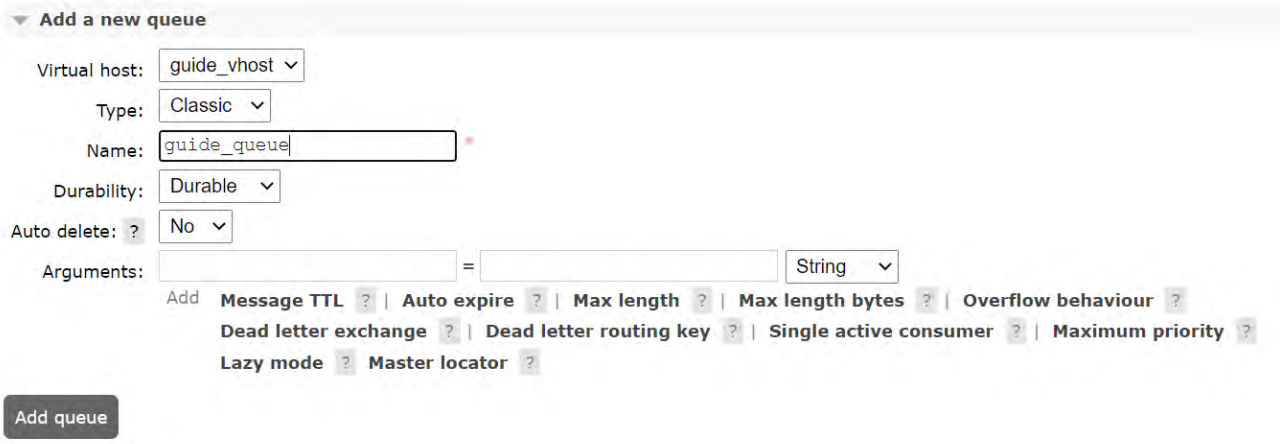
10. Fill the exchange creating form. You should select the virtual host created in the previous step and give a new **name** to the exchange. The **exchange name** is needed for integration with Nwave Cloud. Fill all other options as on the screenshot:



Click on the button “Add exchange” to finish the exchange registration.

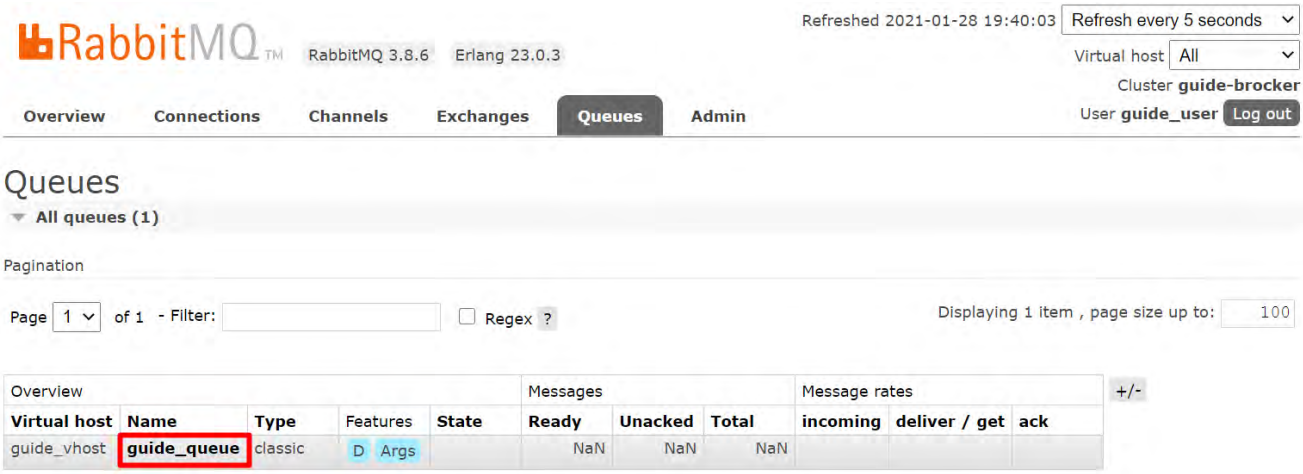
11. Now you have to create a new Queue. Go to the menu **“Queues”** and click on expanding form “**Add a new queue”**

12. Fill the form. You should select previously create Virtual Host and give a name to your queue. Remember the **queue name**. It is needed for Nwave Cloud integration.

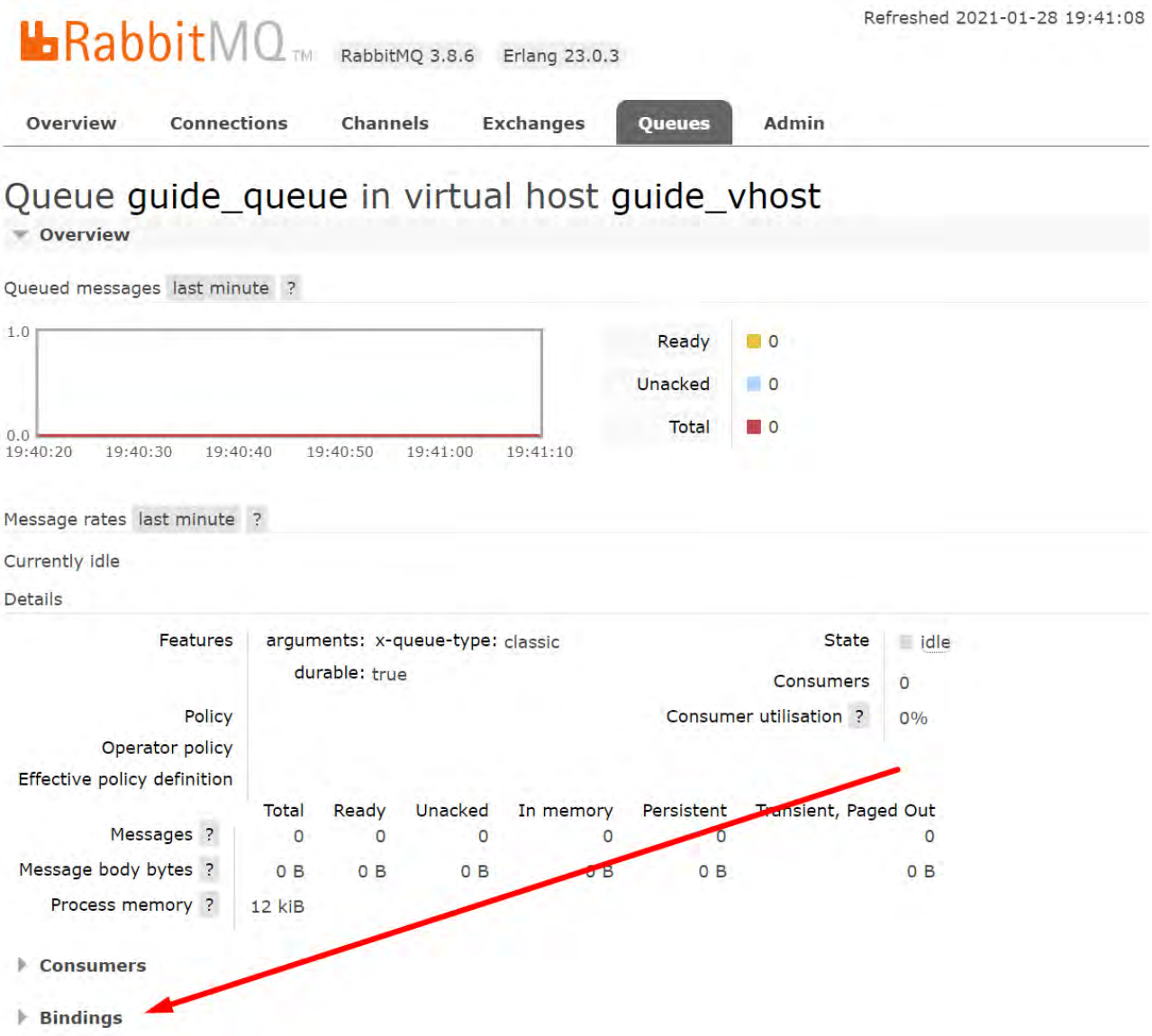


Click on the button **"Add queue"** to finish.

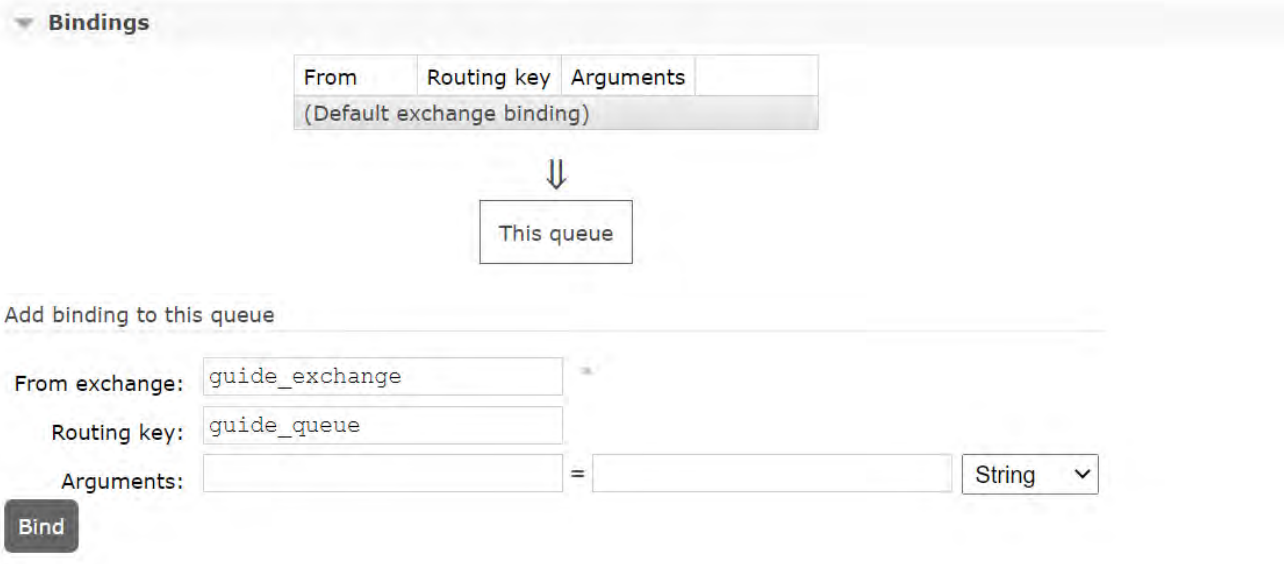
13. Now click on the Queue name in the table



14. Open the form **"Bindings"**



15. Fill in the previously registered exchange name and queue name



Finally, press the button **"Bind"**.

Integration credentials

Here is the list of all credentials than you need to proceed integration with Nwave Cloud:

- Broker URL
- Username
- Password
- Virtual Host
- Exchange name
- Queue name

Now you can proceed to the integration with Nwave Cloud.

Integration with Nwave Cloud

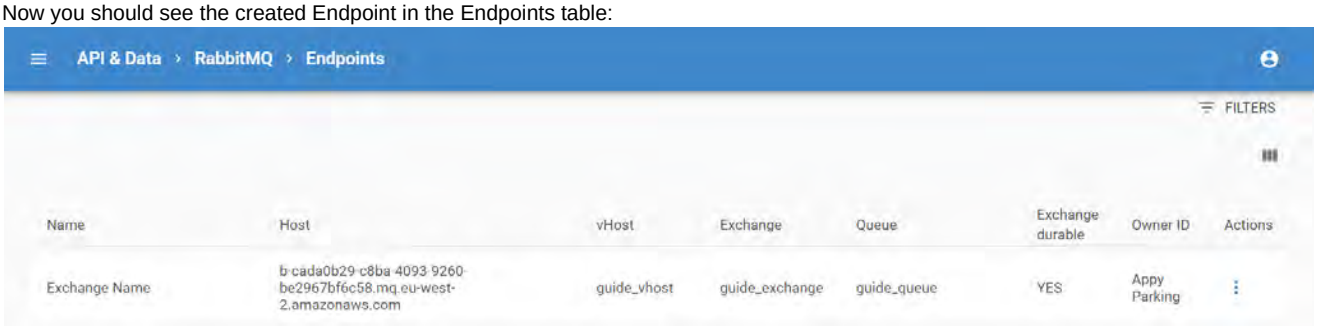
1. Go to the main menu

2. Go to the RabbitMQ Endpoints page

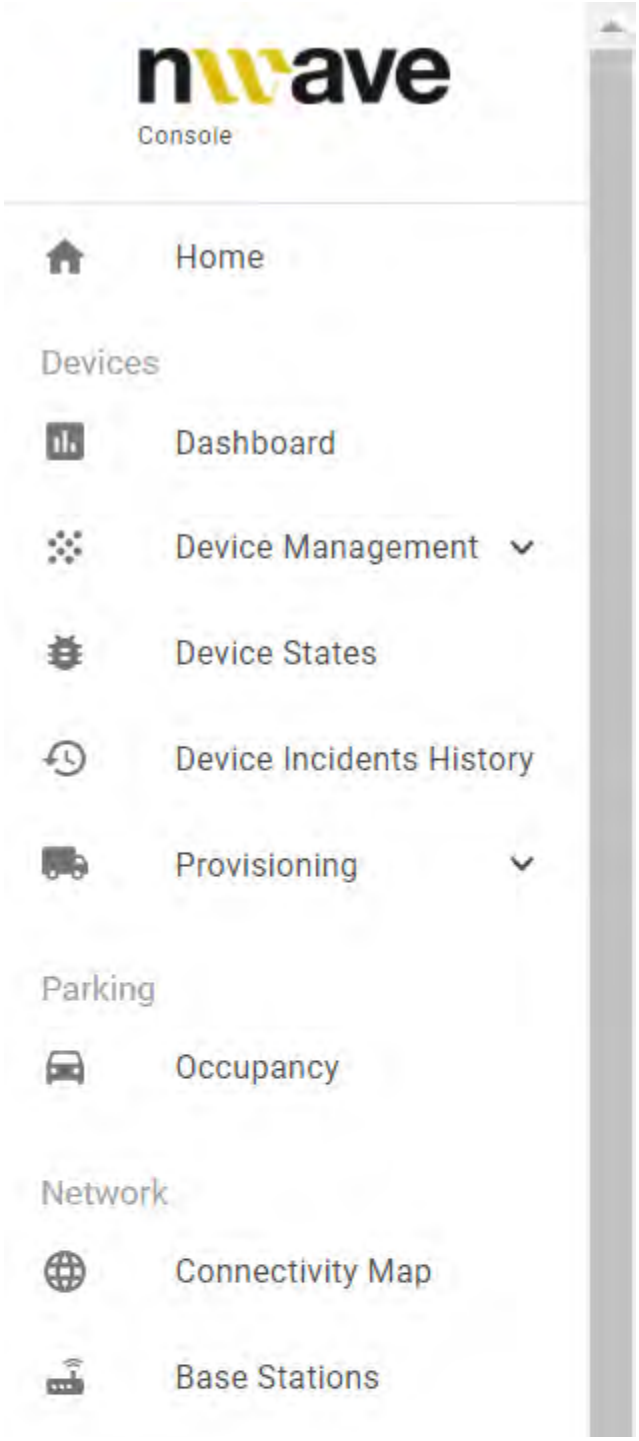
3. Click on the ⊕ button at the bottom right
4. Fill out the “New endpoint” form:
    - Name - user-defined
    - Host - the endpoint URL excluding **“amqps://”** prefix and excluding the **“:port“** postfix
    - Port - 5671
    - VHost - your virtual hostname
    - Exchange - your exchange name
    - Exchange is durable - yes, because this option was selected during the exchange registration
    - Exchange type - Direct, because this option was selected during the exchange registration
    - Queue - your queue name
    - Login - your broker’s username
    - Password/Repeat password - your broker’s password

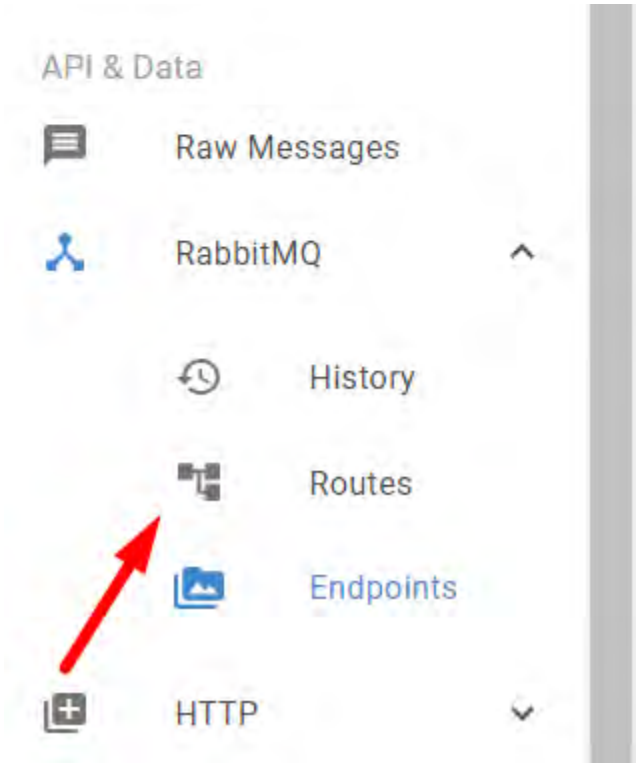
Click on the “Add” button.

5. Now you should see the created Endpoint in the Endpoints table:

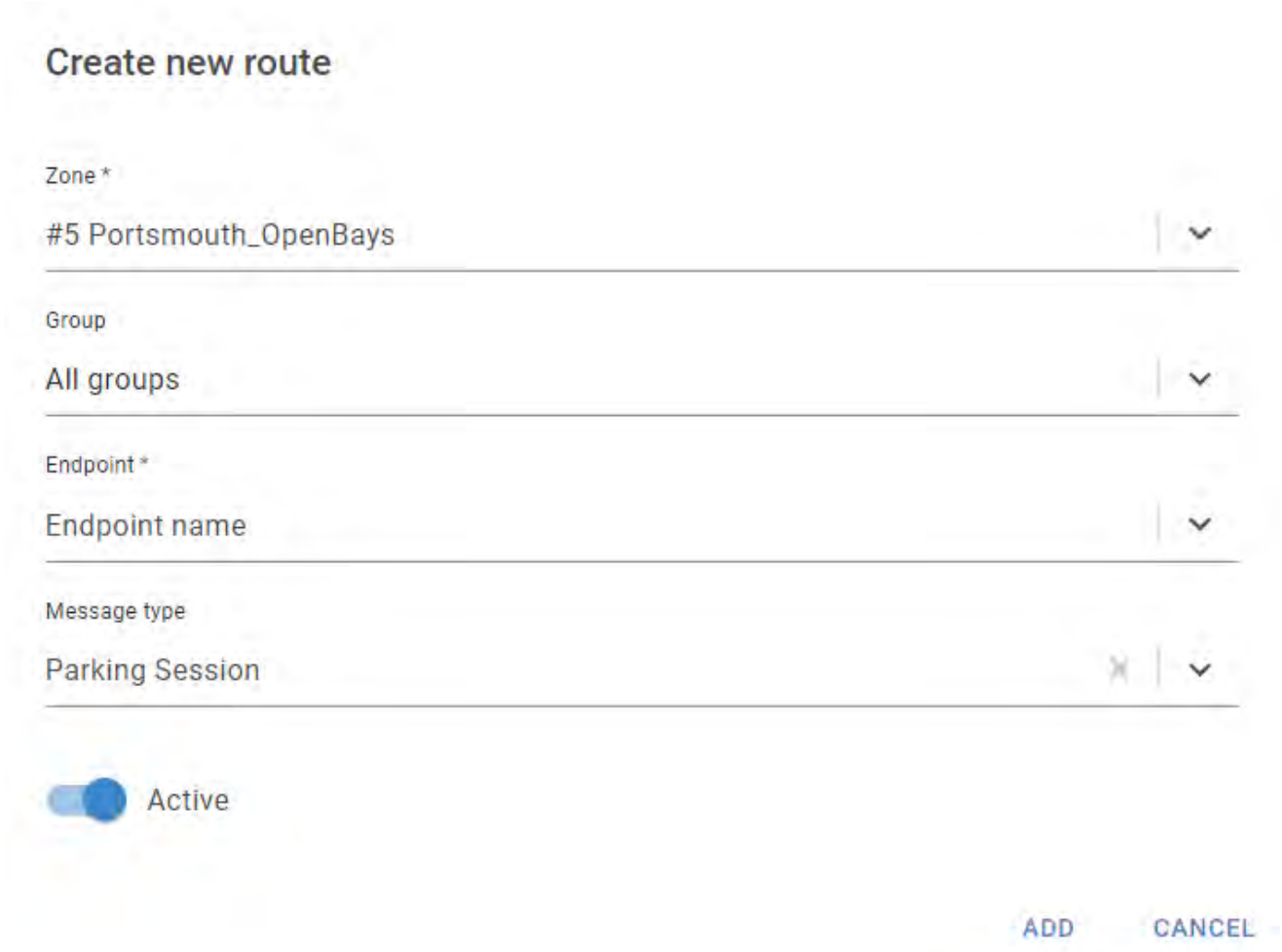


6. Go to the Routes menu to continue the integration





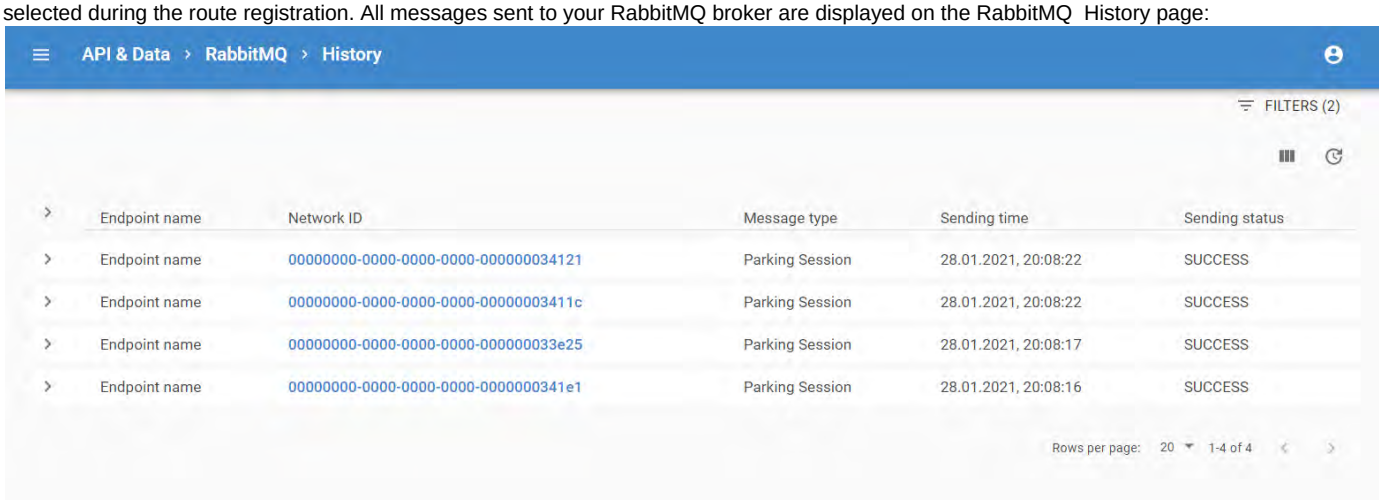
7. Click on the button in the bottom right
8. Fill the form. Select Zone for sending data to RabbitMQ, your previously created Endpoint and message type. Toggle **Active** option if you want to start sending data after the route is created:



Press the button **“Add”** to finish the route registration.

Integration testing

To test the integration you should wait until your sensor sends at least one message. The sensor should be positioned in the zone, which was selected during the route registration. All messages sent to your RabbitMQ broker are displayed on the RabbitMQ History page:



Also, you can check if your RabbitMQ broker is receiving messages. Go to the RabbitMQ broker administration page, find your queue on the Queues page and click on the name of your queue. You should see message processing statistics:

RabbitMQ™ RabbitMQ 3.8.6 Erlang 23.0.3

Refreshed 2021-01-28 20:12:02 Refresh every 5 seconds

Virtual host All

Cluster **guide-brocker**

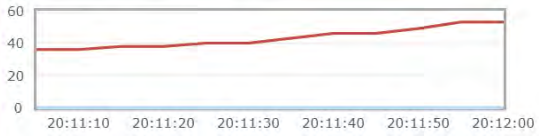
User **guide_user** Log out

Overview | Connections | Channels | Exchanges | Queues | Admin

Queue guide_queue in virtual host guide_vhost

Overview

Queued messages last minute ?



Ready 55

Unacked 0

Total 55

Message rates last minute ?

Publish 0.20/s

Details

RabbitMQ Group Availability

RabbitMQ Group Availability API is capable of sending updates about parking occupancy with a calculated summary. When you use this API, you are getting complete occupancy info about parking. Nwave cloud calculates the number of occupied and vacant spaces according to parking type, parking sensors health and other secondary data.

Group Availability message structure

```
{
  "timestamp": string ISO8601,
  "group_id": integer,
  "group_custom_id": string,
  "level_id": int,
  "floor_number": int,
  "positions_availability": [
    {
      "position": {
        "id": int,
        "network_id": UUID,
        "custom_id": string,
        "group_inner_id": int,
        "lat": real,
        "lon": real
      },
      "occupation_status": string //'occupied' / 'free' / 'n/a'
    }
    ...
  ],
  "summary": {
      "total": int,
      "occupied": int,
      "available": int,
      "undefined": int
  }
}
```

where:

- `timestamp` - message generation time
- `group_id` - parking group id
- `group_custom_id` - user-defined ID of a parking group
- `level_id` - identified of a level object which describes parking floor
- `floor_number` - parking floor number
- `positions_availability` - list of parking spaces in a parking group with their IDs, geo-locations and occupancy statuses
- `summary` - an object which describes a number of parking spaces inside of parking and a number of occupied and vacant spaces

RabbitMQ Parking Sessions

Parking Session Logging

The second generation of Parking Session Logging API is designed for comprehensive *parking session* representation.

It provides robust session integrity control and auto-correction service in the case of partial sensor message loss.

This API allows customers to focus on business applications of *parking sessions* instead of postprocessing of individual sensor occupancies.

This API includes sensors' positioning, grouping and zoning attributes so that your data receiving service can be stateless.

General Info

The API can be used with all types of parking lots: marked spaces, open bays, off-street parking and garages.

Please, watch the video at the end of this document to learn how to set up RabbitMQ Endpoint in the Nwave Console.

API details

Data objects

Every parking session is described by the following data structure:

```
{
    "parking_session_uuid": "string",
    "correction_counter": integer,
    "session_start":{
        "event_time": "string" // timestamptz, yyyy-MM-dd'T'HH:mm:ss.
SSSXXX (2019-06-13T16:16:51.000+00:00)
        "delta_time_sec": integer,
        "message_trace_ids":["strings""]
    },
    "partial_end":{
        "event_time": "string" // timestamptz, yyyy-MM-dd'T'HH:mm:ss.
SSSXXX (2019-06-13T16:16:51.000+00:00)
        "delta_time_sec": "integer",
        "message_trace_ids":["strings"],
        "network_id": "string", // unexpectedly released position
        "custom_id": "string"
    },
    "session_end":{
        "event_time": "string" // timestamptz, yyyy-MM-dd'T'HH:mm:ss.
```

```
SSSXXX (2019-06-13T16:16:51.000+00:00)
        "delta_time_sec": integer,
        "message_trace_ids":["strings"]
    },
    "involved_devices":[
        {
             "serial_id": "string", # deprecated
             "device_id": "string",
             "hardware_type": "string",
             "position":{
                 "network_id": "string",
                 "custom_id": "string",
                 "latitude": float,
                 "longitude": float,
                 "group":{
                     "id": int,
                     "type": "string",
                     "name": "string",
                     "custom_id": "string",
                     "zone_id": int,
                     "level_id": int,
                     "level_name": "string",
                     "floor_number": int
                 },
                 "group_inner_id": int,
             }
        },
        ...
    ],
    "auth_ble_tag": {
        "tag_id": "string",
        "event_time": "string"
    },
    "auth_mobile": {
        "session_id": "string",
        "event_time": "string"
    }
}
```

Objects description

`parking_session_uuid` - a unique parking session-id;

`correction_counter` - total number of corrections.

```
"session_start":{
    "event_time":string // timestamptz, yyyy-MM-dd'T'HH:mm:ss.SSSXXX
(2019-06-13T16:16:51.000+00:00)
    "delta_time_sec":integer,
    "message_trace_ids":[strings]
}
```

The object `session_start` **is sent to open a parking occupancy session:**

`event_time` - a session start timestamp;

`delta_time_sec` - an error of session start time in seconds;

`message_trace_ids` - tracing identifiers of raw device messages related to the session start

---

```
"session_end":{
    "event_time":string // timestamptz, yyyy-MM-dd'T'HH:mm:ss.SSSXXX
(2019-06-13T16:16:51.000+00:00)
    "delta_time_sec":integer,
    "message_trace_ids":[strings]
}
```

The object `session_end` **is sent to close a parking occupancy session:**

`event_time` - a session end timestamp;

`delta_time_sec` - an error of session end time in seconds;

`message_trace_ids` - tracing identifiers of raw device related to the session end.

```
"partial_end":{
    "event_time":string // timestamptz, yyyy-MM-dd'T'HH:mm:ss.SSSXXX
(2019-06-13T16:16:51.000+00:00)
    "delta_time_sec":integer,
    "message_trace_ids":[strings],
    "network_id":string,
    "custom_id":string
}
```

`partial_end` **object can be received only for Open Bay parking type**

A partial end is an event where two sensors were occupied in an open bay session but only one sensor was released. This is a temporary state of parking sessions. A `session_end` object will always be sent when the second sensor is released in that session.

`event_time` - a timestamp of unexpected sensor release;

`delta_time_sec` - a timestamp error in seconds;

`message_trace_ids` - tracing identifiers of messages which describe unexpected release;

`network_id` - network_id of a sensor's position that was unexpectedly released;

`custom_id`- a user-defined identifier of a sensor's position that was unexpectedly released.

```
"involved_devices":[
  {
        "serial_id":string,, # deprecated
        "device_id": string,
        "hardware_type":string,
        "position":{
            "network_id":string,
            "custom_id":string,
            "latitude":float,
            "longitude":float,
            "group":{
                "id":int,
                "type":string,
                "name":string,
                "custom_id":string,
                "zone_id":int,
                "level_id": int,
                "level_name": str,
                "floor_number": int
            },
            "group_inner_id":int,
        }
    },
    ...
]
```

`involved_devices` - a list which contains a description of parking sensors involved in a parking session;

`serial_id` - a parking sensor serial ID (deprecated);

`device_id` - a parking sensor serial ID;

`hardware_type` - a parking sensor hardware model;

`position.network_id` - a position network_id;

`position.custom_id`- a position user-defined identifier;

`position.latitude`/`longitude` - a position geolocation coordinates;

`position.group.id`- a position group ID;

`position.group.type` - a position group type. Possible values: `general`, `unmarked_parking_bay`, `marked_parking_bay`;

`position.group.name` - a position group name;

`position.group.custom_id` - a position group user-defined ID;

`position.group.zone_id` - an ID of a zone which contains positions group;

`position.group.level_id` - an ID of level (floor) which contains positions group;

`position.group.level_name` - a Name of level (floor) which contains positions group;

`position.group.floor_number` - a floor number which contains positions group;

`position.group_inner_id` - an index number of a position in positions group.

```
"auth_ble_tag": {
    "tag_id": str,
    "event_time": string
}
```

The object `auth_ble_tag` contains user authorization information if a user authorized on parking space using Nwave's BLE-Tag;

`tag_id` - ID of BLE-Tag;

`event_time`- a timestamp of user authorization.

```
"auth_mobile": {
    "session_id": string,
    "event_time": string
}
```

The object `auth_mobile`contains user authorization information if a user authorized on a parking space using mobile application connected to the Nwave One-Click Check-in service;

`one_click_session_id` - an ID of One-Click Check-in session;

`event_time`- a timestamp of user authorization.

Examples

Session start message

When a parking session starts, the Nwave cloud builds a message that contains only data about the sensor and time of parking session beginning.

It can contain one or two described devices depending on the device's group type.

```
{
 "parking_session_uuid": "e9fcc95e-b9cd-4e7f-b275-092a62daf61d",
 "involved_devices": [
   {
     "device_id": "a33b47",
     "hardware_type": "Sparkit Surface V3.9",
     "position": {
       "network_id": "23f3e949-2dd3-47ca-b00f-c3310d4ce418",
       "custom_id": "684d395c-e875-422a-904d-c095ad981cc6",
       "latitude": 50.793682,
       "longitude": -1.0986286,
       "group": {
         "id": 1545,
         "type": "unmarked_parking_bay",
         "name": "cambridgepark32",
         "custom_id": "684d395c-e875-422a-904d-c095ad981cc6",
         "zone_id": 5,
         "level_id": null,
         "floor_number": null
       },
       "group_inner_id": 1
     }
   }
 ],
 "correction_counter": 0,
 "session_start": {
   "event_time": "2021-01-26T07:48:40.121000+00:00",
   "delta_time_sec": 60,
   "message_trace_ids":
   [
     "d1b6d450-cc5f-83d4-adbe-c9f5383fb0d8"
   ]
 }
}
```


as you can see, the value of field `delta_time_sec` is 60. It means, that real parking session start time is between `2021-01-26T07:48:40 - 60 sec` and `2021-01-26T07:48:40 + 60 sec`.

Session end message

```
{
  "parking_session_uuid": "e9fcc95e-b9cd-4e7f-b275-092a62daf61d",
  "involved_devices": [
    {
      "device_id": "a33b47",
      "hardware_type": "Sparkit Surface V3.9",
      "position": {
        "network_id": "23f3e949-2dd3-47ca-b00f-c3310d4ce418",
        "custom_id": "684d395c-e875-422a-904d-c095ad981cc6",
        "latitude": 50.793682,
        "longitude": -1.0986286,
        "group": {
          "id": 1545,
          "type": "unmarked_parking_bay",
          "name": "cambridgepark32",
          "custom_id": "684d395c-e875-422a-904d-c095ad981cc6",
          "zone_id": 5,
          "level_id": null,
          "floor_number": null
        },
        "group_inner_id": 1
      }
    }
  ],
  "correction_counter": 0,
  "session_start": {
    "event_time": "2021-01-26T07:48:40.121000+00:00",
    "delta_time_sec": 60,
    "message_trace_ids": [
      "d1b6d450-cc5f-83d4-adbe-c9f5383fb0d8"
    ]
  },
  "session_end": {
    "event_time": "2021-01-26T07:51:26.466000+00:00",
    "delta_time_sec": 0,
    "message_trace_ids": [
      "06f75246-20e5-9d48-404b-fa965bbdefe7"
    ]
  }
}
```


Partial-end message

Partial-end is possible only for unmarked group type when a car parks on 2 neighbouring sensors. When a car releases only a single sensor, the partial-end message is sent. When the car releases the second sensor, the session end message is sent. This event can happen when a car tries to join traffic on a busy road.

Here we show an example of the partial-end message:

```
{
  "parking_session_uuid": "da2ecb68-efb1-458e-ba5d-e5ba80b87f6e",
  "involved_devices": [
    {
      "device_id": "33c83",
      "hardware_type": "Sparkit Surface V3.9",
      "position": {
        "network_id": "00000000-0000-0000-0000-000000033c83",
        "custom_id": "cd4206ec-6876-4eca-b859-d5735dd97386",
        "latitude": 50.780323,
        "longitude": -1.0682689,
        "group": {
          "id": 1729,
          "type": "unmarked_parking_bay",
          "name": "5EastneyEsplanade",
          "custom_id": "cd4206ec-6876-4eca-b859-d5735dd97386",
          "zone_id": 5,
          "level_id": null,
          "floor_number": null
        },
        "group_inner_id": 54
      }
    }, {
      "device_id": "33e5f",
      "hardware_type": "Sparkit Surface V3.9",
      "position": {
        "network_id": "00000000-0000-0000-0000-000000033e5f",
        "custom_id": "cd4206ec-6876-4eca-b859-d5735dd97386",
        "latitude": 50.78033,
        "longitude": -1.0682275,
        "group": {
          "id": 1729,
          "type": "unmarked_parking_bay",
          "name": "5EastneyEsplanade",
          "custom_id": "cd4206ec-6876-4eca-b859-d5735dd97386",
          "zone_id": 5,
          "level_id": null,
          "floor_number": null
        },
        "group_inner_id": 55
      }
    }
  ],
  "correction_counter": 0,
  "session_start": {
    "event_time": "2021-01-26T09:54:08.287000+00:00",
    "delta_time_sec": 0,
    "message_trace_ids": [
      "65ab2a63-e7c2-8674-aa3d-7047cb428a31"
    ]
```

```
  },
  "partial_end": {
    "event_time": "2021-01-26T10:03:23.900000+00:00",
    "delta_time_sec": 0,
    "message_trace_ids": [
      "56d9035a-e4dc-9661-0804-a968d9a92d0b"
    ],
    "network_id": "00000000-0000-0000-0000-000000033e5f",
    "custom_id": "cd4206ec-6876-4eca-b859-d5735dd97386"
  }
}
```


In the example above you can see that object “partial-end” contains only one sensor.

When the second sensor in the session is released, the Nwave cloud will add a “session-end” object to the end of the partial-end message.

Click here to expand...


```
{
 "parking_session_uuid": "da2ecb68-efb1-458e-ba5d-e5ba80b87f6e",
 "involved_devices": [
   {
     "device_id": "33c83",
     "hardware_type": "Sparkit Surface V3.9",
     "position": {
       "network_id": "00000000-0000-0000-0000-000000033c83",
       "custom_id": "cd4206ec-6876-4eca-b859-d5735dd97386",
       "latitude": 50.780323,
       "longitude": -1.0682689,
       "group": {
         "id": 1729,
         "type": "unmarked_parking_bay",
         "name": "5EastneyEsplanade",
         "custom_id": "cd4206ec-6876-4eca-b859-d5735dd97386",
         "zone_id": 5,
         "level_id": null,
         "floor_number": null
       },
       "group_inner_id": 54
     }
   }, {
     "device_id": "33e5f",
     "hardware_type": "Sparkit Surface V3.9",
     "position": {
       "network_id": "00000000-0000-0000-0000-000000033e5f",
       "custom_id": "cd4206ec-6876-4eca-b859-d5735dd97386",
       "latitude": 50.78033,
       "longitude": -1.0682275,
       "group": {
         "id": 1729,
         "type": "unmarked_parking_bay",
```

```
            "name": "5EastneyEsplanade",
            "custom_id": "cd4206ec-6876-4eca-b859-d5735dd97386",
            "zone_id": 5,
            "level_id": null,
            "floor_number": null
          },
          "group_inner_id": 55
        }
      }
    ],
    "correction_counter": 0,
    "session_start": {
      "event_time": "2021-01-26T09:54:08.287000+00:00",
      "delta_time_sec": 0,
      "message_trace_ids": [
        "65ab2a63-e7c2-8674-aa3d-7047cb428a31"
      ]
    },
    "partial_end": {
      "event_time": "2021-01-26T10:03:23.900000+00:00",
      "delta_time_sec": 0,
      "message_trace_ids": [
        "56d9035a-e4dc-9661-0804-a968d9a92d0b"
      ],
      "network_id": "00000000-0000-0000-0000-000000033e5f",
      "custom_id": "cd4206ec-6876-4eca-b859-d5735dd97386"
    },
    "session_end": {
      "event_time": "2021-01-26T10:16:10.116000+00:00",
      "delta_time_sec": 0,
      "message_trace_ids": [
        "8182f7b8-f5bf-2084-aac8-0b3412965a84"
      ]
    }
  }
```


Session edges correction

As you can see in the examples above, all the messages contain the `correction_counter` field. This field and the field `parking_session_uid` allows you to apply corrections to parking sessions in the right order.

Parking session correction is an important functionality of Parking Session Logging API. The wireless data reception is not ideal, so some messages may be lost. To minimize this effect, Nwave recommends to use at least 2 stations in the same area, but message loss is still possible. Nwave Cloud analyzes additional data from parking sensors and recovers most of the lost messages. When a message is recovered, the Nwave Cloud analyzes occupancy history and makes a correction. It may correct the session's start or end time as well as create new parking sessions.

For example, let's consider 2 messages describing the parking session starting:

```
{
  "parking_session_uuid": "da2ecb68-efb1-458e-ba5d-e5ba80b87f6e",
  "involved_devices": [
    {
      ...
    }
  ],
  "correction_counter": 0,
  "session_start": {
    "event_time": "2021-01-26T09:54:08.287000+00:00",
    "delta_time_sec": 120,
    "message_trace_ids": [
      "65ab2a63-e7c2-8674-aa3d-7047cb428a31"
    ]
  },
  ...
}
```

The message above contains the `session_start` object with `delta_time_sec` field equal to 0 and `correction_counter` field equal to 0. The following message corrects the previous one and decreases session start time delta.

```
{
  "parking_session_uuid": "da2ecb68-efb1-458e-ba5d-e5ba80b87f6e",
  "involved_devices": [
    {
      ...
    }
  ],
  "correction_counter": 1,
  "session_start": {
    "event_time": "2021-01-26T09:55:32.944000+00:00",
    "delta_time_sec": 0,
    "message_trace_ids": [
      "857e7628-38d4-4497-9229-c25d78096d52"
    ]
  },
  ...
}
```

As you can see, the correction message contains the same `parking_session_uuid`, incremented `correction_counter` and more accurate `event_time`. The second message contains different “message”trace_ids”

Endpoint & Route Setup

RMQ Endpoint & Routes Video.mp4

RabbitMQ Consumer Code Examples

Here you can find consumers examples using different programming languages.

The official RabbitMQ site contains a very useful tutorial with code examples. But if you want to run your consuming application right now, you can use examples on this page

Connection parameters

All of these examples use the following connection parameters:

- Protocol:
  - AMQP - if you don't use SSL connection
  - AMQPS - if you use SSL connection. This protocol is used by default on AWS managed RabbitMQ.
- Hostname
- Username
- Password
- Port (usually it is 5671 or 5672)
- Virtual hostname (vHost)
- Exchange name
- Queue name

All our examples use exchange type "direct" and "durable" exchanges and queues.

Examples

JavaScript

Java script example code uses package `amqplib`. Please, install the package before proceeding with an example:

```
npm install amqplib
```

Now you are able to fill in your connection parameters into the script and start consuming:

```
#!/usr/bin/env node

var amqp = require('amqplib/callback_api');

var broker_url = 'amqps://<username>:<password>@<host>:<port>/<vhost>'
var exchange = '<exchange_name>'
var queue = '<queue_name>'


amqp.connect(broker_url, function(error0, connection) {
  if (error0) {
    throw error0;
  }
  connection.createChannel(function(error1, channel) {
    if (error1) {
      throw error1;
    }

    channel.assertExchange(exchange, 'direct', {
      durable: true
    });

    channel.assertQueue(queue, {
      durable: true
      }, function(error2, q) {
        if (error2) {
          throw error2;
        }

      channel.bindQueue(q.queue, exchange, queue);

      console.log(' [*] Waiting for data. To exit press CTRL+C');
      channel.consume(q.queue, function(msg) {
        console.log(" [x] %s: '%s'", msg.fields.routingKey, msg.content.
toString());
      }, {
        noAck: true
      });
    });
  });
});
```

Python

Python example uses package `pika`:

```
pip install pika
```

This example code is able to run with using Python 3.6 or higher. You can remove f-string using and run the code on earlier Python3 version.

```
#!/usr/bin/env python3.8

import pika
import sys
import os


RMQ_QUEUE = '<queue_name>'
RMQ_HOST = '<host>'
RMQ_PORT = <port>
RMQ_VHOST = '<vhost>'
RMQ_LOGIN = '<username>'
RMQ_PASS = '<password>''
RMQ_EXCHANGE = '<exchange_name>'


URL = f'amqp://{RMQ_LOGIN}:{RMQ_PASS}@{RMQ_HOST}:{RMQ_PORT}/{RMQ_VHOST}'

def main():

    parameters = pika.URLParameters(URL)
    rmq_connection = pika.BlockingConnection(parameters)
    rmq_channel = rmq_connection.channel()
    rmq_channel.exchange_declare(
        exchange=RMQ_EXCHANGE,
        exchange_type='direct',
        durable=True
    )
    rmq_channel.queue_declare(
        queue=RMQ_QUEUE,
        durable=True
    )

    def callback(ch, method, properties, body):
        print(" [x] Received %r" % body)

    rmq_channel.basic_consume(
        queue=RMQ_QUEUE,
        on_message_callback=callback,
        auto_ack=True
    )

    print(' [*] Waiting for messages. To exit press CTRL+C')
```

```
    rmq_channel.start_consuming()


if __name__ == "__main__":
    try:
        main()
    except KeyboardInterrupt:
        print('Interrupted')
        try:
            sys.exit(0)
        except SystemExit:
            os._exit(0)
```

RabbitMQ Car Counter

Car Counting service

Car counting service processes data from car counters. Car counters are devices that have special car counting firmware.

NB: During installation and setup process Car counters should also be bound to positions in groups of “*Stand-alone Car Counter*” type.

This service can correct raw sensor data and provides *total* count of cars that crossed the sensor over. This provides a protection against partial message loss, each message has the full data by a given timestamp.

In order to provide a better balance balance between the frequency of reporting and battery life counters have two modes of operation - Standard (max delay 20min) and Faster updates (max delay 5 minutes).

Please refer to the tables below for more details on the updates schedule in each mode:

Standard updates

Number of events	Minimum time between previous and new counter updates
1..2	20 minutes
3..4	10 minutes
5..9	5 minutes
10 and more	3 minutes

Faster updates

Number of events	Minimum time between previous and new counter updates
1..2	5 minutes
3..4	4 minutes
5..6	3 minutes
7..9	2 minutes
10 and more	1 minute

API details

Every car counting data object has the following format:

```
{
  "type": "sa_car_counter",
  "sensor_id": "31777",
  "timestamp": 1615211864,
  "counter": 652,
  "errors": null,
  "msg_version": 2,
  "trace_id": "8d386f23-1172-27bb-55d9-5389a5fbf72e"
}
```

Fields description:

- `"type": "sa_car_counter"` - always has the same value
- `"sensor_id"` - sensor hardware ID in hex format
- `"timestamp"` - Unix-timestamp of event
- `"counter"` - number of detected cars. The maximum value is greater than 2 billion. This should be enough for most cases.
- `"errors"` - list of errors or null. Supported error list can be different for different firmwares
- `"msg_version"` - data protocol version
- `"trace_id"` - message trace ID which can be used for development and debugging

So in the most simple case with one Entry (ingress) and one Exit (egress) lane the service receives two incrementing counters and subtracts Exit counter from Entry counter to calculate the number of vehicles in the parking perimeter.

REST Occupancy API

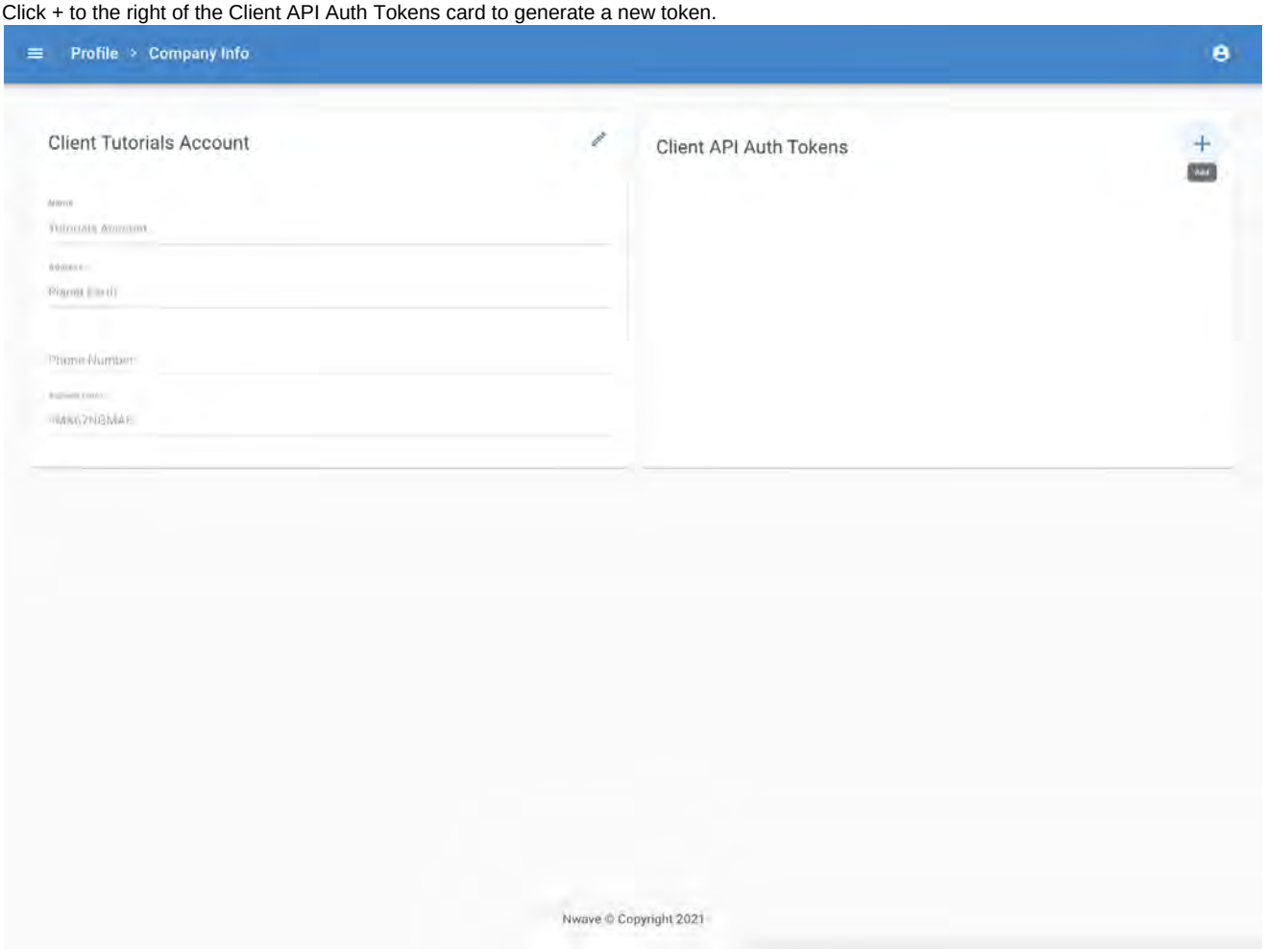
Authorization

To retrieve occupancy information the client needs to be authorised. To successfully authorize with this API, the client is required to provide a valid authorization token in the **x-Auth-Token** header of the HTTP request.

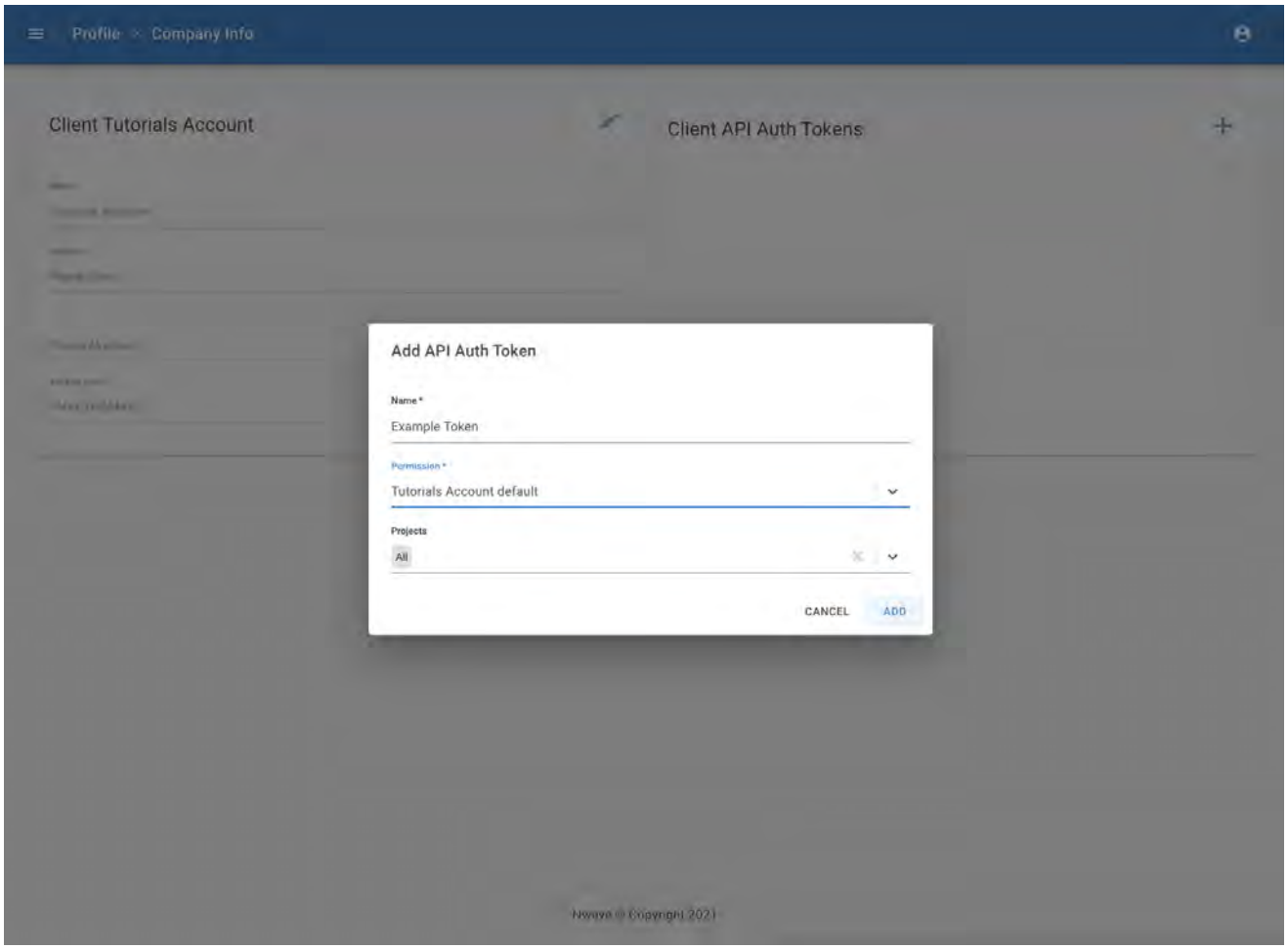
The authorization token can be obtained from the Nwave's console.

1. Click on the user icon in the top right corner and select Company Info.

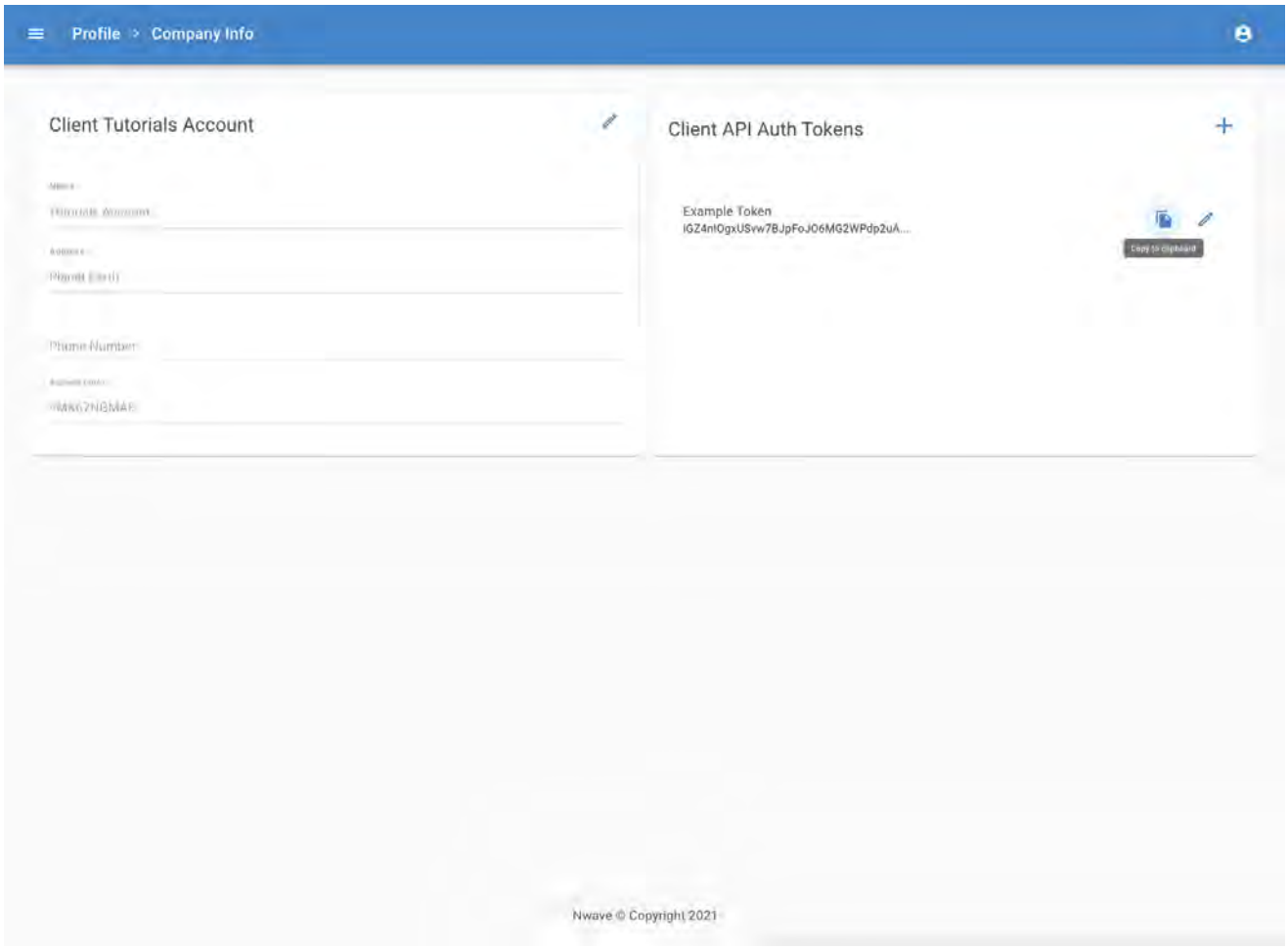
2. Click + to the right of the Client API Auth Tokens card to generate a new token.



3. Enter a name, select permissions and project scope for the new token and click Add.



4. Copy the generated token.



Endpoints Overview

/group/find/short_info

This endpoint retrieves a list of groups and their occupancy summary per specified filters. Each group info has geolocation location data and filters may be geospatial.

> Useful for:
>
> 1. **Quick** display of all the parking groups and their aggregate status per specified filters.
> 2. Displaying occupancy summary on **digital signage**.

Available filters:

- project_id
- zone_id
- group_id
- level_id or floor_number
- geospatial filter

More detailed info can be found in Swagger documentation in the end of this document.

/group/{group_id}/status

This endpoint retrieves **detailed** information for a single group. It will return a list of all parking positions, their location and occupancy status. It will also return an occupancy summary for this group.

More detailed info can be found in Swagger documentation in the end of this document.

Useful for **quickly** displaying all the parking spots in a chosen location. e.g. when a user selects a parking group in a mobile app.

/positions/states/find

This endpoint retrieves **comprehensive** information for all of the positions with extensive filtering capabilities.

More detailed info can be found in Swagger documentation in the end of this document.

Useful for finding **overstays** in a particular zone as it is capable of filtering by occupancy and status change time.

Thi endpoint can be **slower** in comparison to /group/find/short_info that returns less data.

/level/find/short_info

This endpoint retrieves level details and occupancy summary for each level.

More detailed info can be found in Swagger documentation in the end of this document.

Useful for displaying occupancy per level on digital signage.

/level/{level_id}/status

This endpoint retrieves **detailed** information for a single level. It will return level details as well as occupancy and location details of every position on that level.

More detailed info can be found in Swagger documentation in the end of this document.

Useful for displaying occupancy of all positions on a single level in a multi-storey car park.

Quick Start Guide

Sending Requests

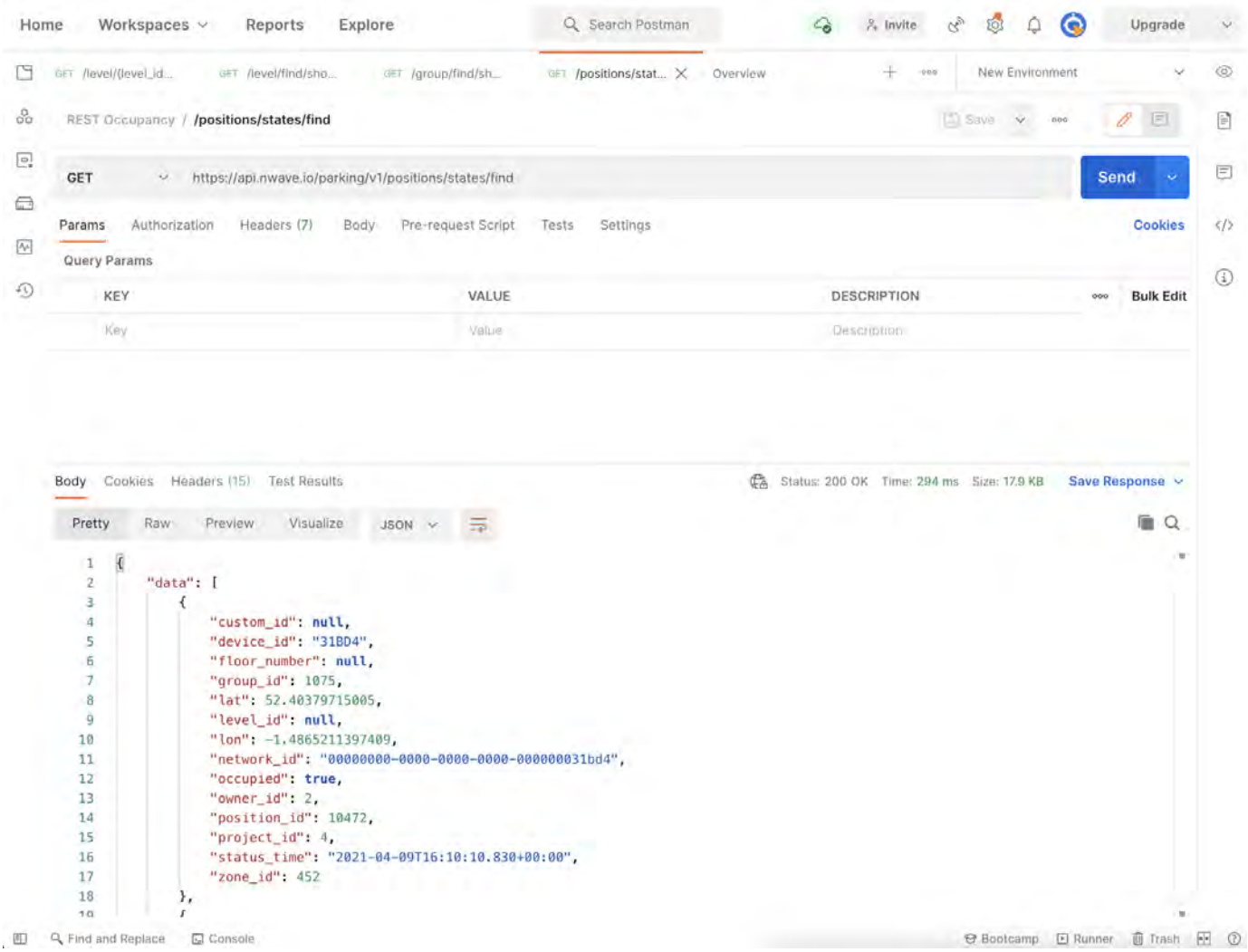
cURL

Run the following command in your terminal window.

```
curl -v -H 'x-Auth-Token: <your_token>' 'https://api.nwave.io/parking/v1
/positions/states/find'
```

Postman

1. Enter the URL and select the GET method.
2. Fill out the key-value pair under the headers tab.
3. Click Send.



Use Cases

Get Group Information & Occupancy Summary on the 2nd floor of MSCP

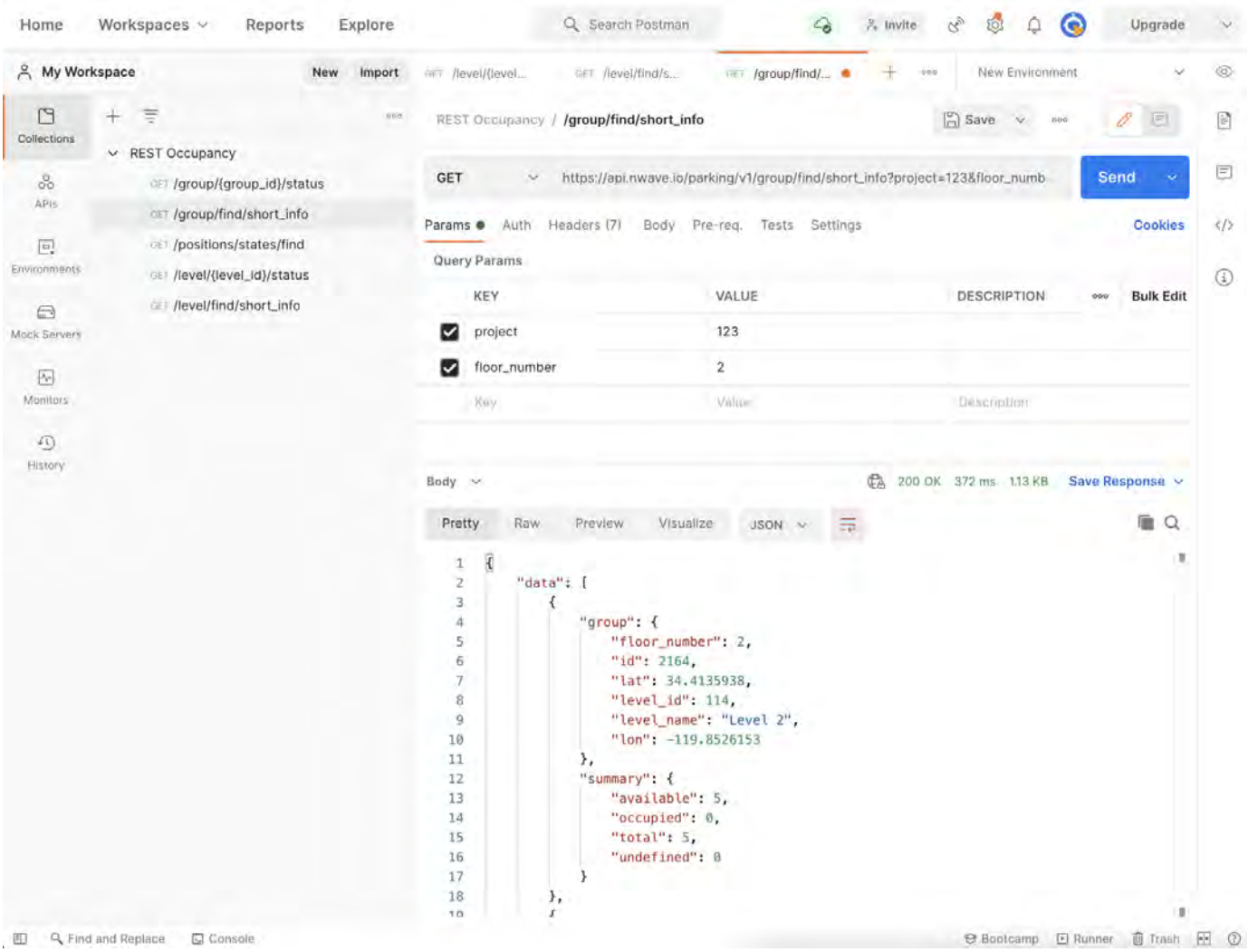
Get the list of parking groups objects (including occupancy summary) on the 2nd level of multi-storey car parking (project ID = 123).

> This request will return all groups on the 2nd floor and the summary for each group. If you want a summary for the whole floor use **level** endpoints.

cURL Example

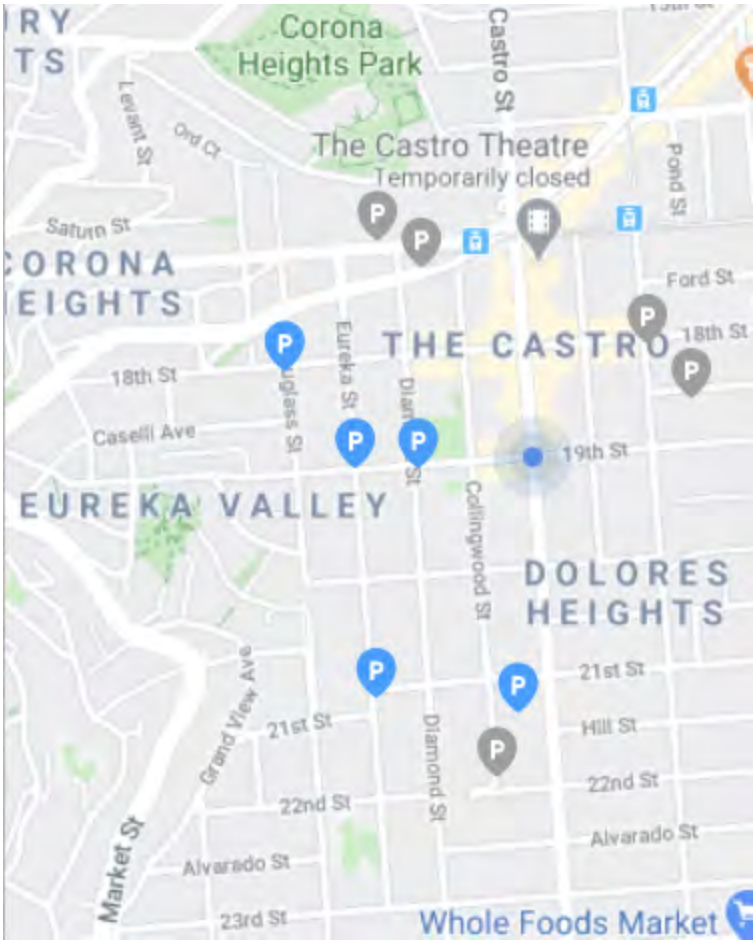
```
curl -v -H 'x-Auth-Token: <Your API Key>' 'https://api.nwave.io/parking
/v1/group/find/short_info?project=123&floor_number=2'
```

Postman Example



Get Group Information & Occupancy Summary within a search radius

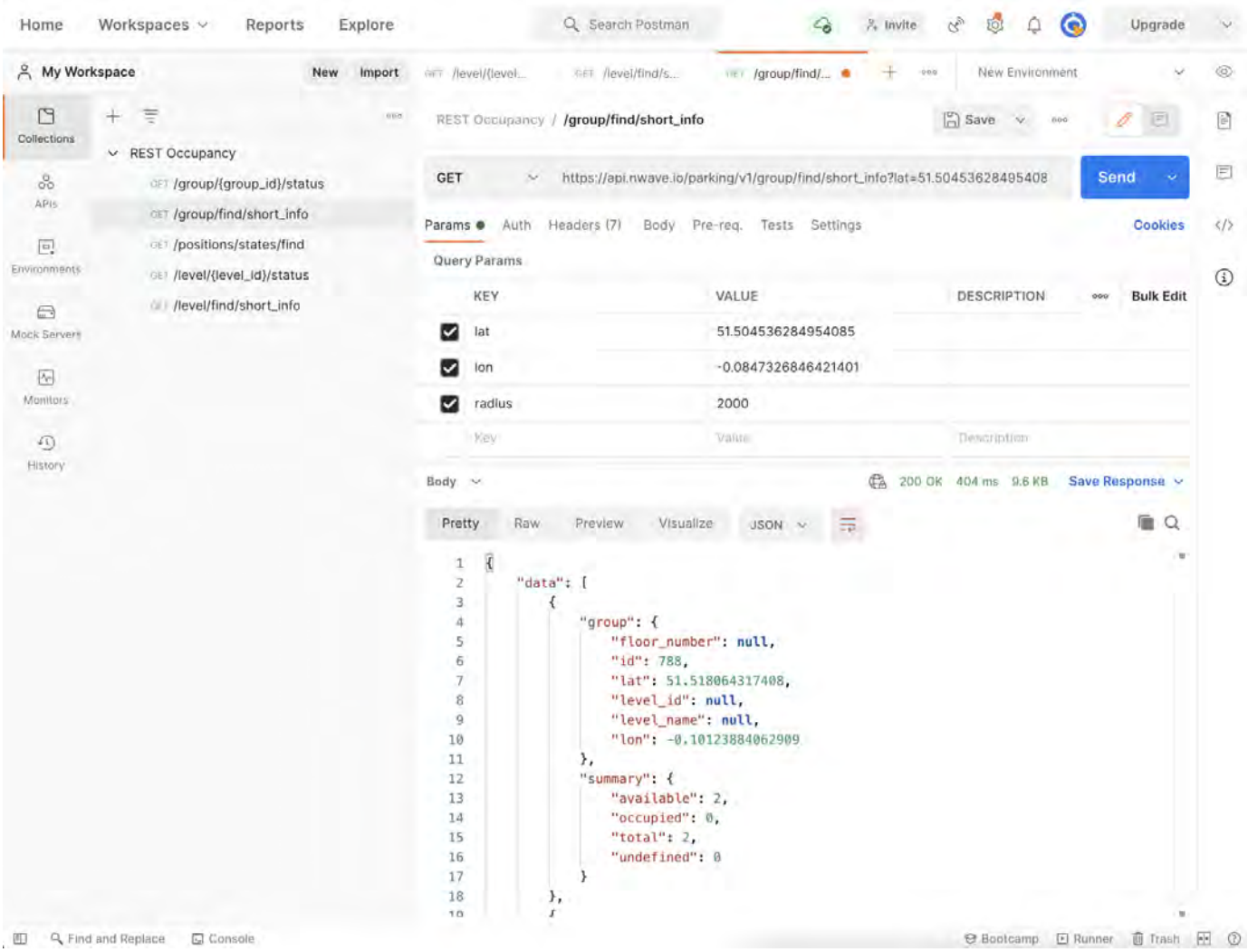
Get parking groups within 2km of the user's coordinates.



cURL example

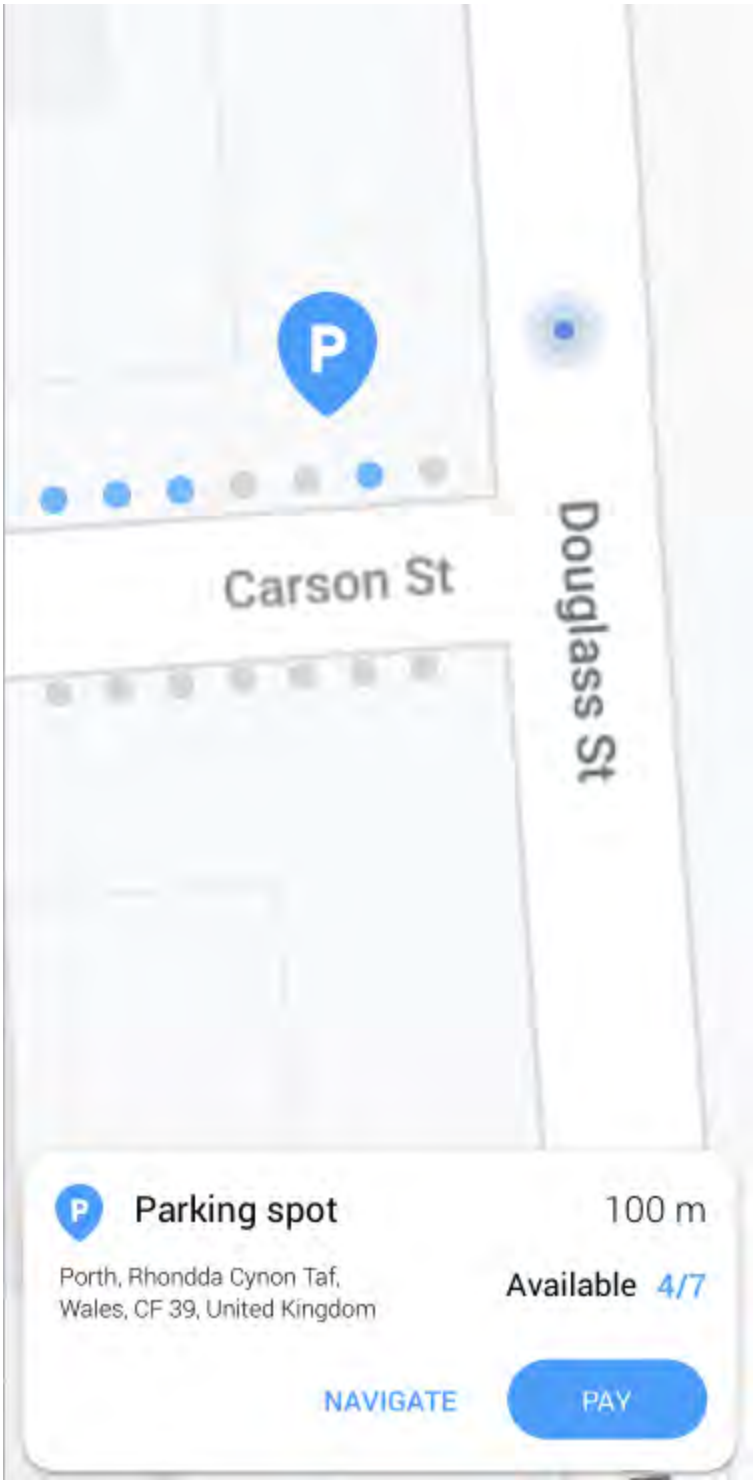
```
curl -v -H 'x-Auth-Token: <Your API Key>' 'https://api.nwave.io/parking
/v1/group/find/short_info?lat=51.504536284954085&lon=-0.0847326846421401
&radius=2000'
```

Postman example



Get individual position occupancy & summary for group X

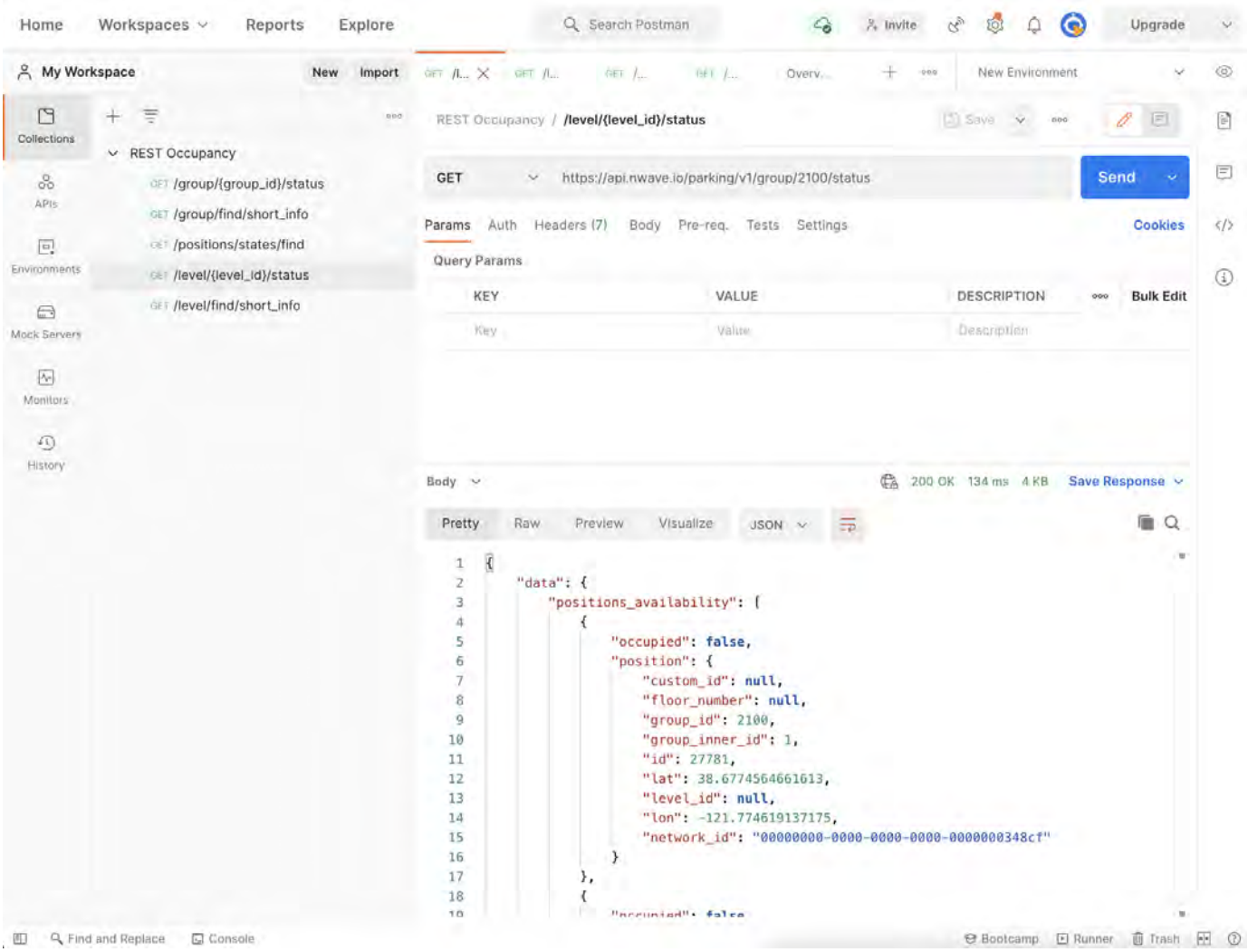
Get detailed group information for group 2100 when a user chooses a group from the map.



cURL example

```
curl -v -H 'x-Auth-Token: <Your API Key>' 'https://api.nwave.io/parking
/v1/group/2100/status'
```

Postman example

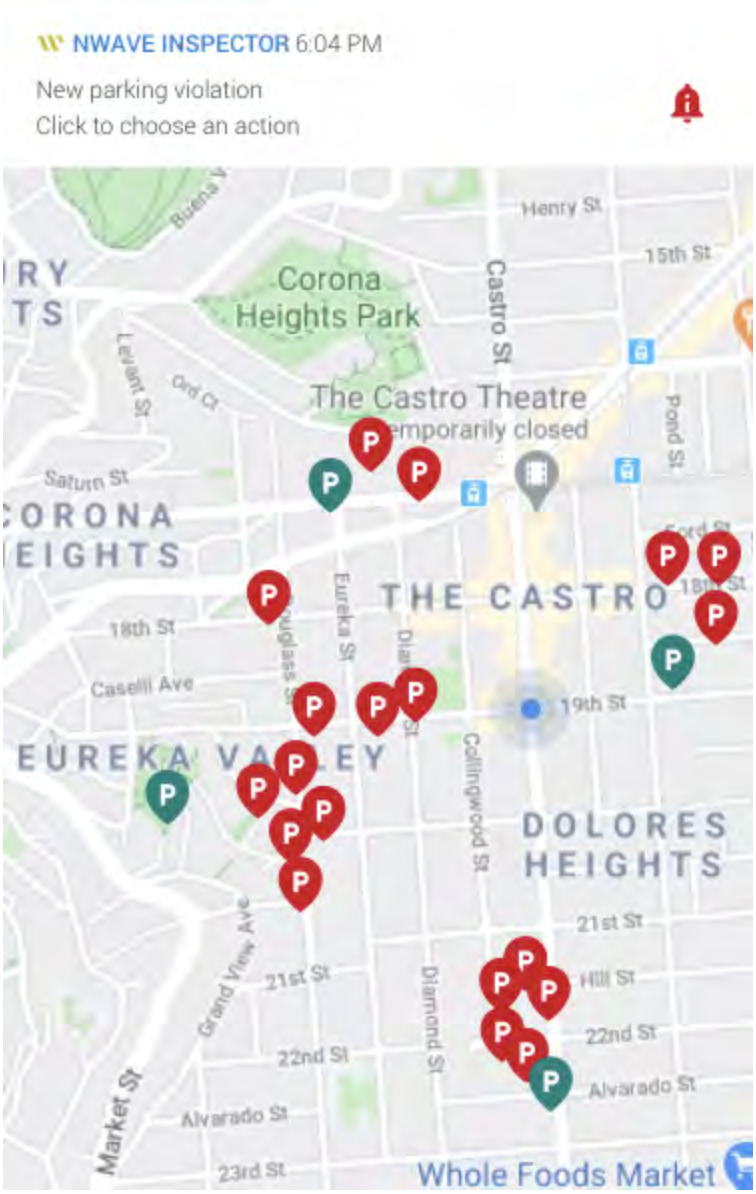


Find occupancies longer than X minutes

If the current time is 2021-02-01 15:24:00, in order to find all positions that have been occupied for over 2 hours in zone 419, we should use the following query string parameters:

1. state: Occupied
2. time_to: 2021-04-09T13:24:00 (current time minus 120 minutes)
3. zone_id: 419

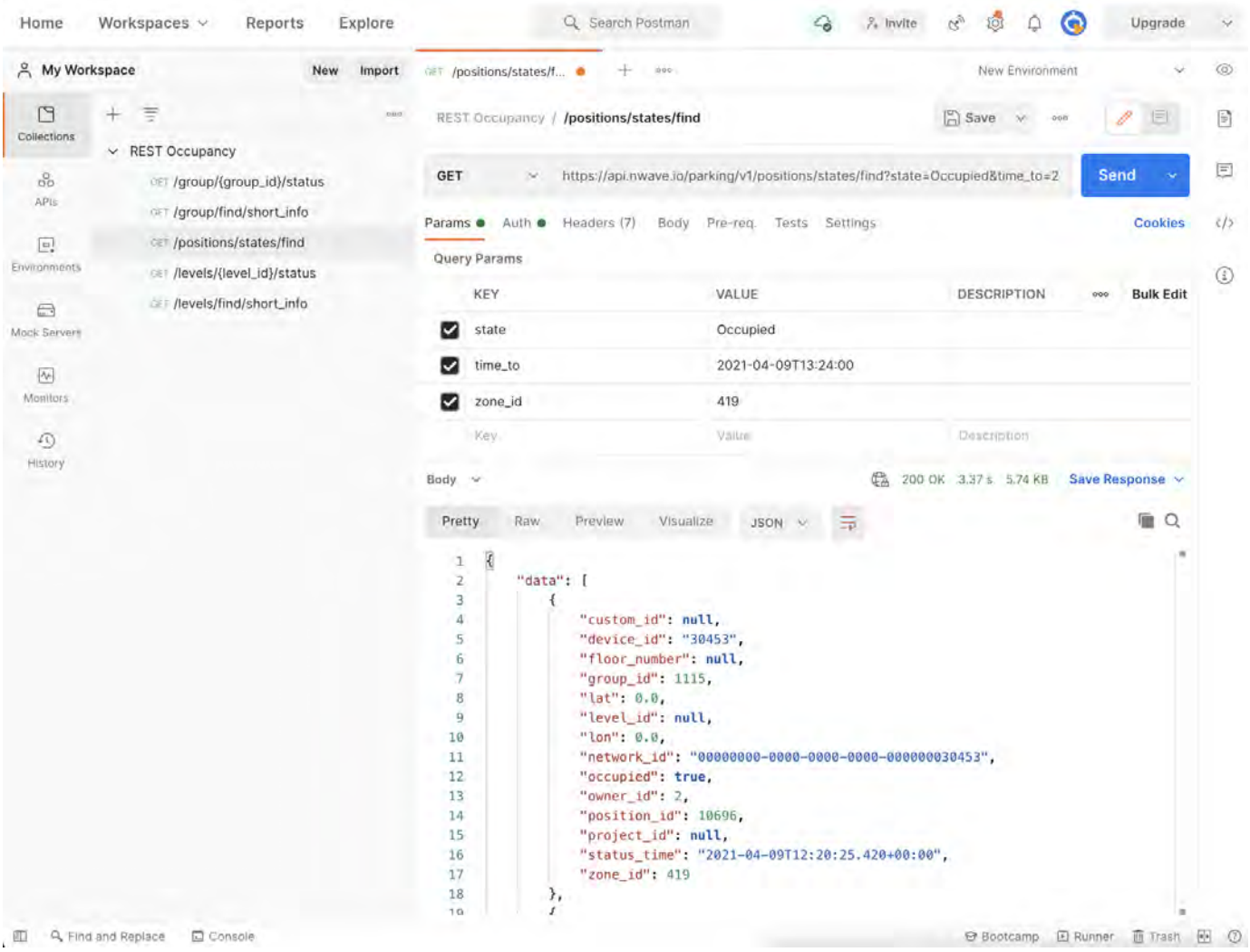
zone id is available on the individual zone page



cURL example

```
curl -v -H 'x-Auth-Token: <Your API Key>' 'https://api.nwave.io/parking
/v1/positions/states/find?state=Occupied&time_to=2021-02-01T16:00:
00&zone_id=123'
```

Postman example



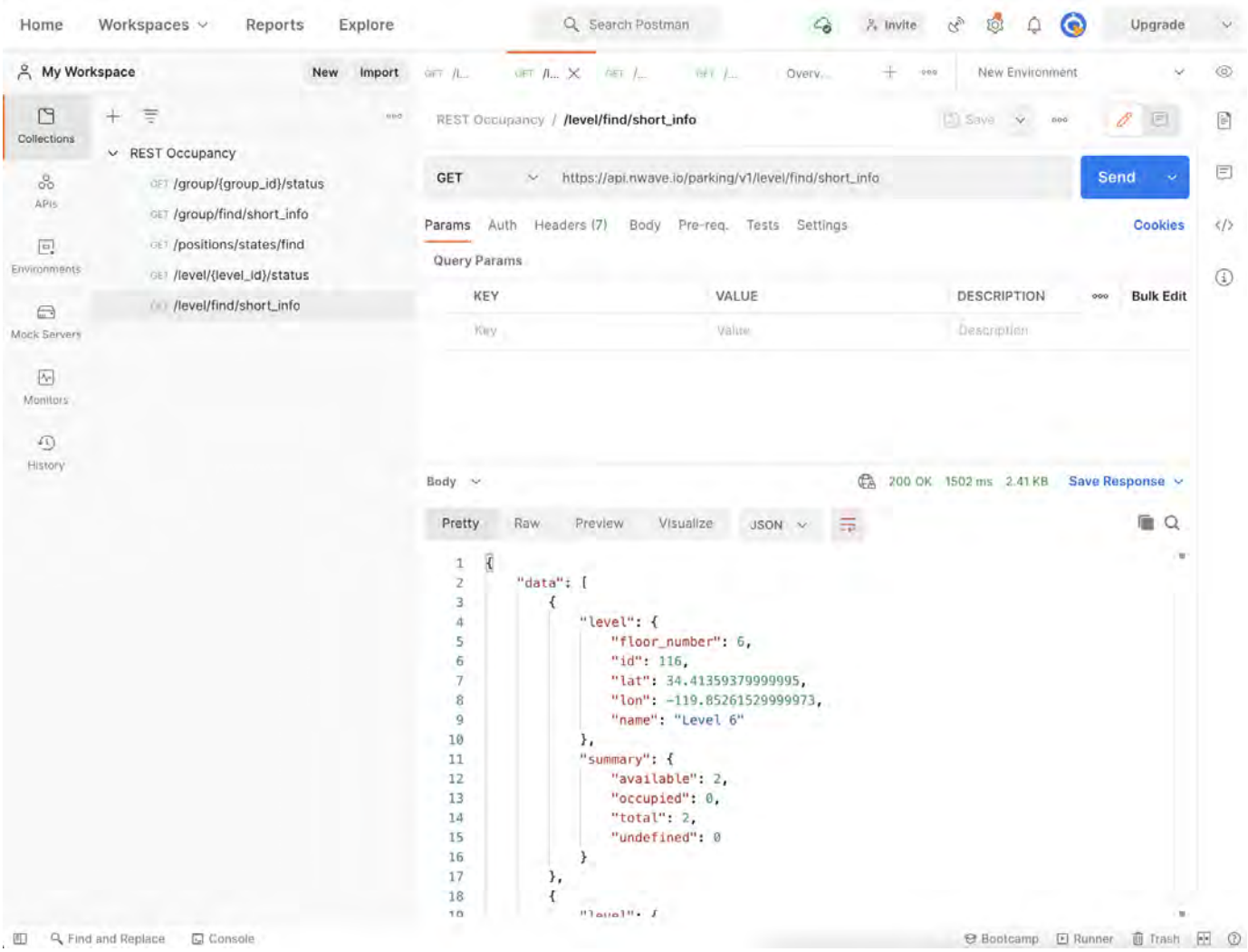
Get occupancy summary per level (Digital signage)

To retrieve occupancy summaries for all levels and display them on a digital sign use the following request.

cURL example

```
curl -v -H 'x-Auth-Token: <Your API Key>' 'https://api.nwave.io/parking
/v1/level/find/short_info?zone_id=123'
```

Postman example



Get position information and summary for level X (MSCP occupancy map)

Retrieving all of the positions information and summary for a level can be useful for displaying availability map for a single floor in a multi-storey car park.

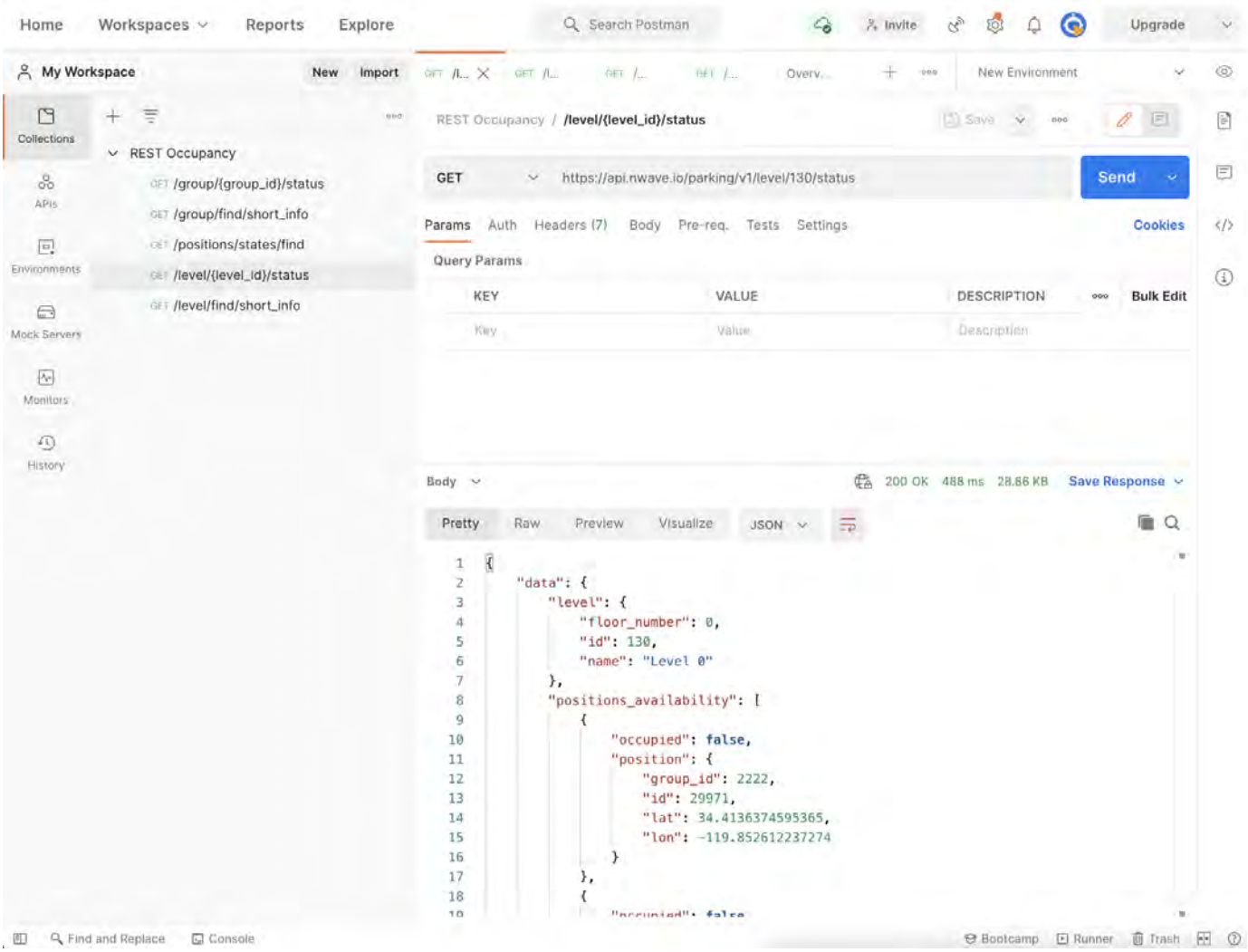
This request will retrieve:

1. level information
2. positions locations on a level
3. positions' occupancies
4. occupancy summary for that level

cURL example

```
curl -v -H 'x-Auth-Token: <Your API Key>' 'https://api.nwave.io/parking/v1/level/130/status'
```

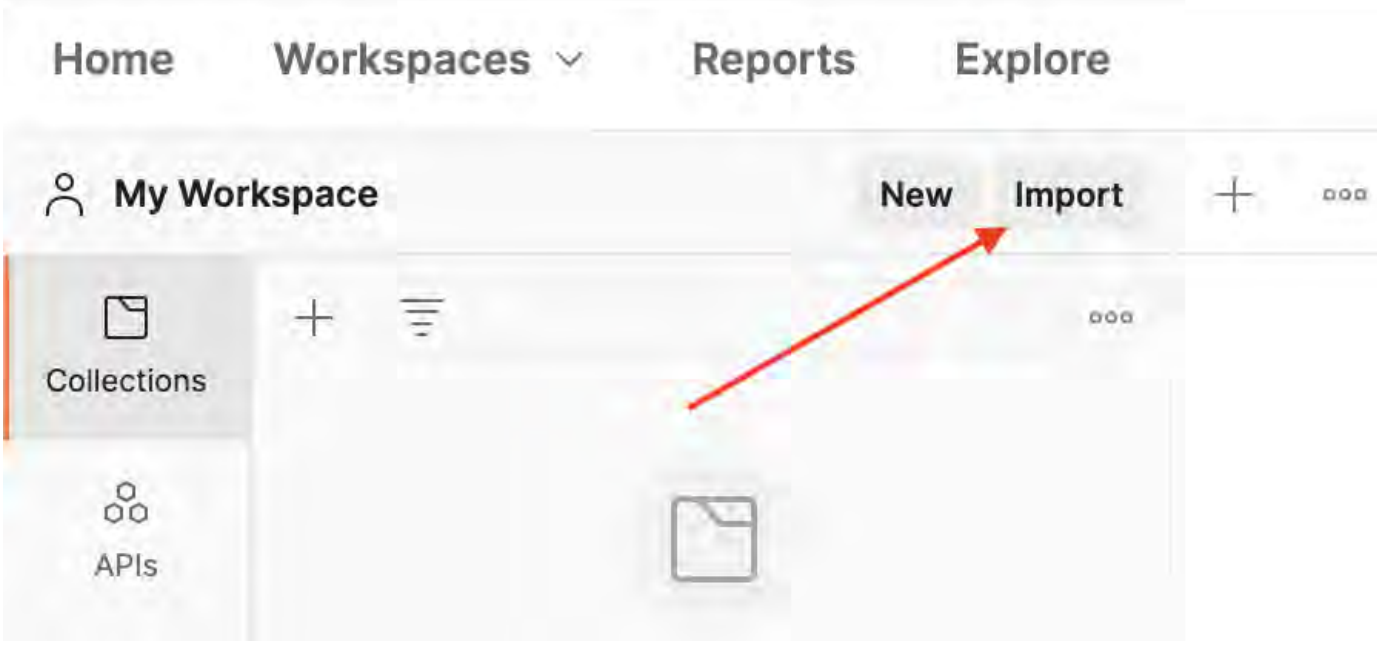
Postman example



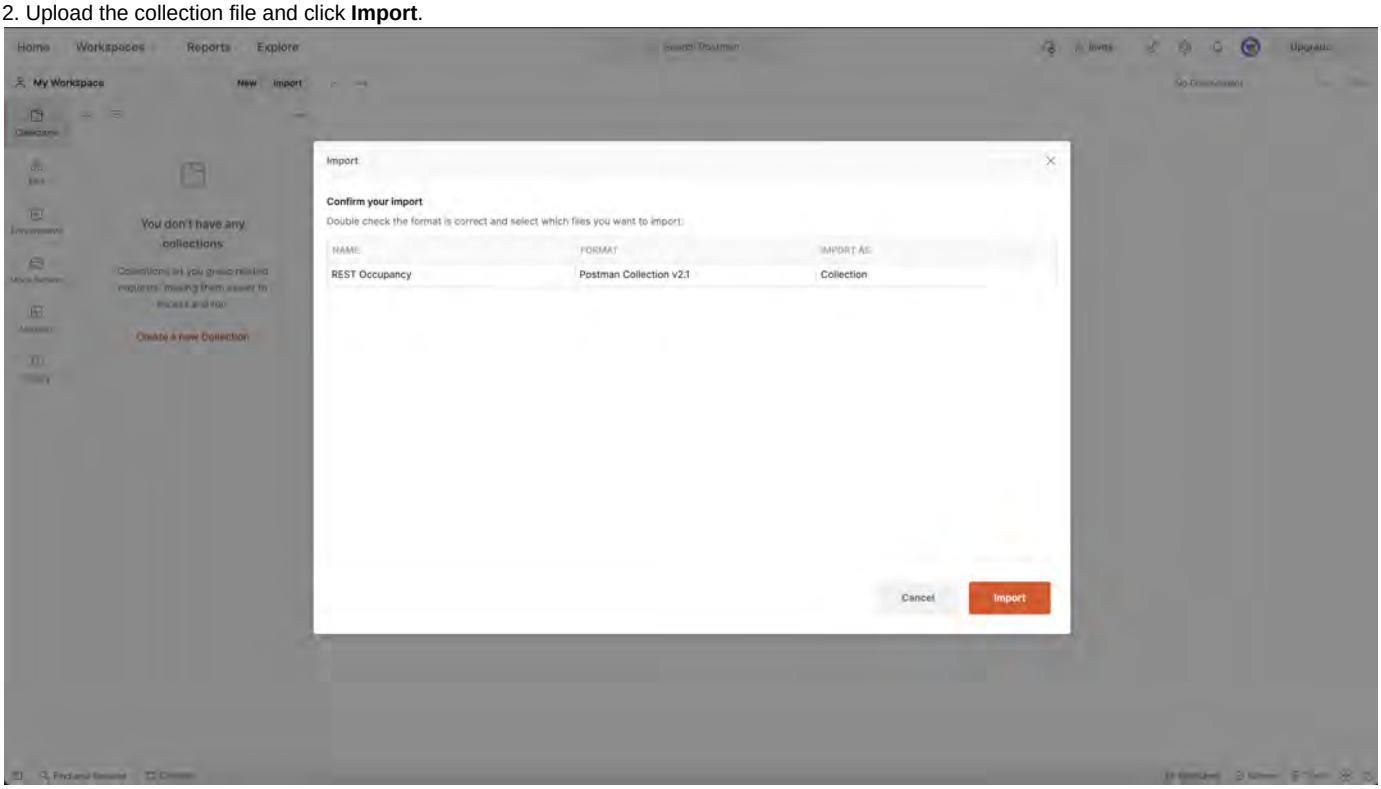
Postman Collection

Importing Collection

1. Click **import** in the My Workspace section.



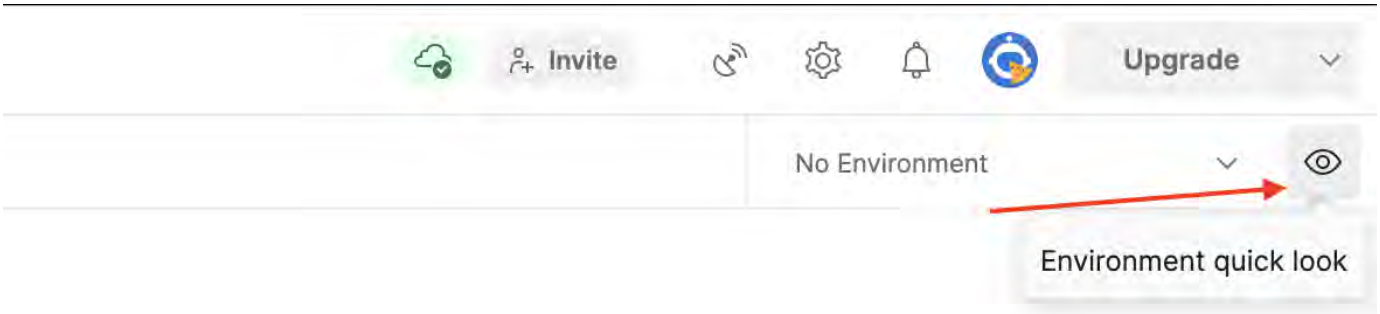
2. Upload the collection file and click **Import**.



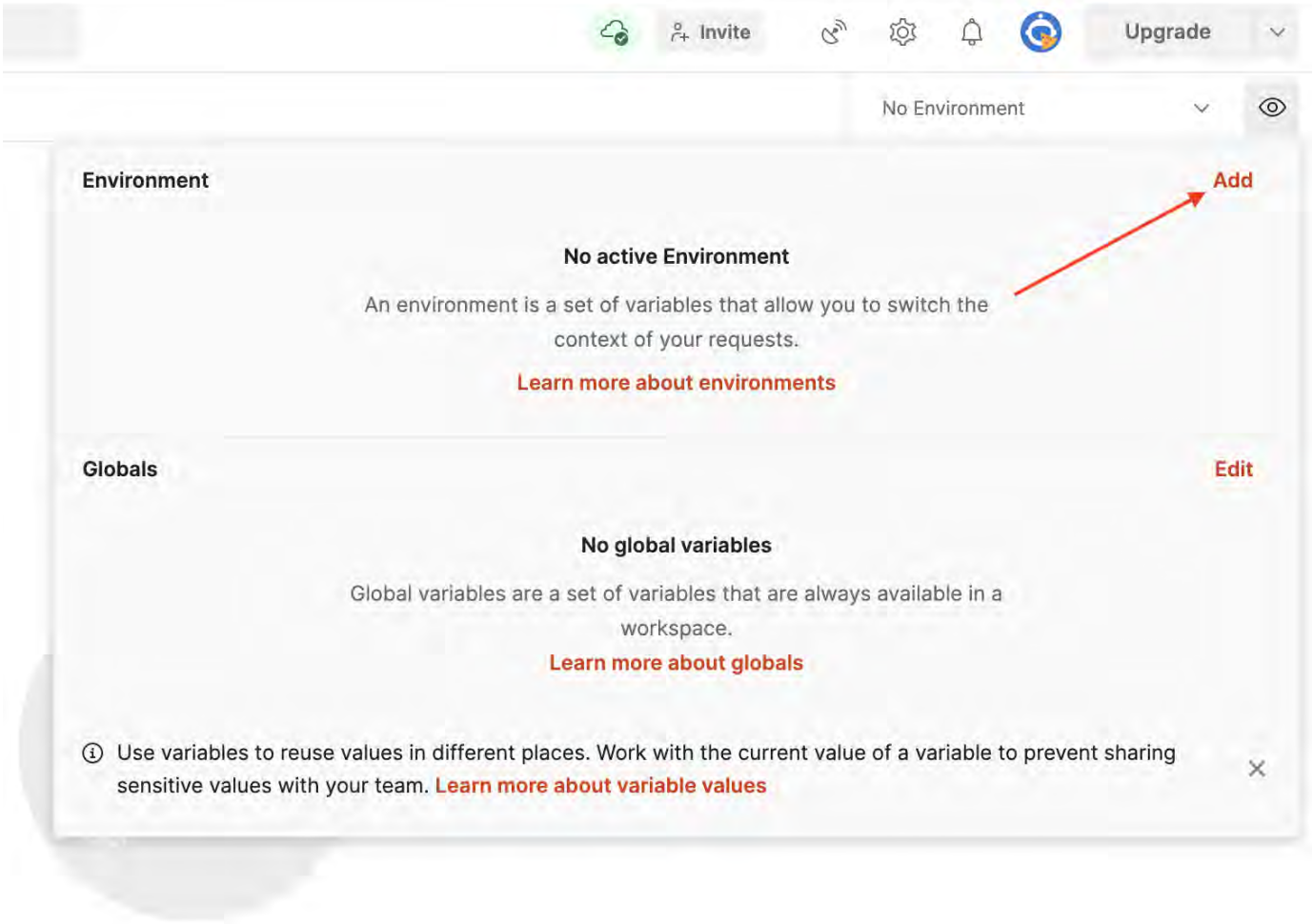
Adding API key to Postman Environment

This collection uses the **api_key** variable to add an authorization token to the **x-Auth-Token** header.

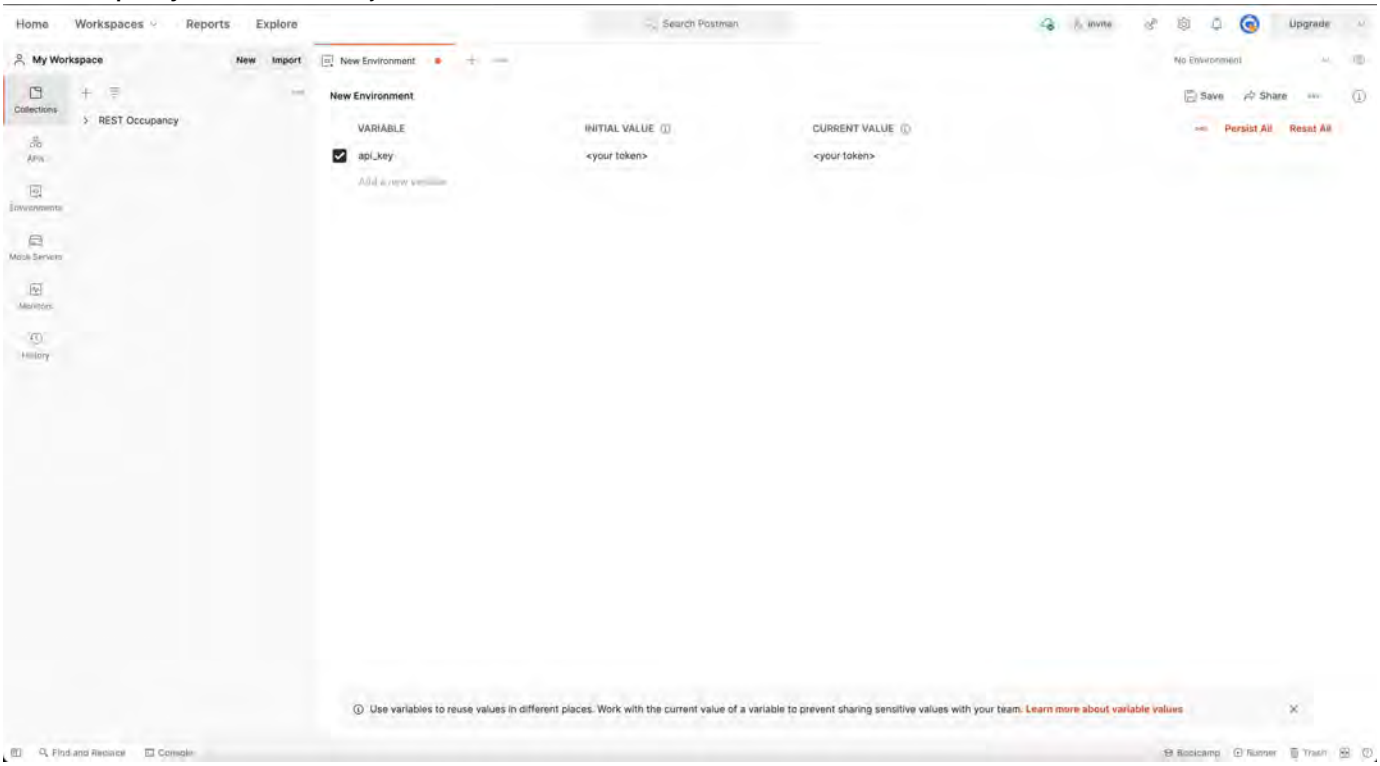
1. To create a new environment click on the **eye icon** near the top right corner.



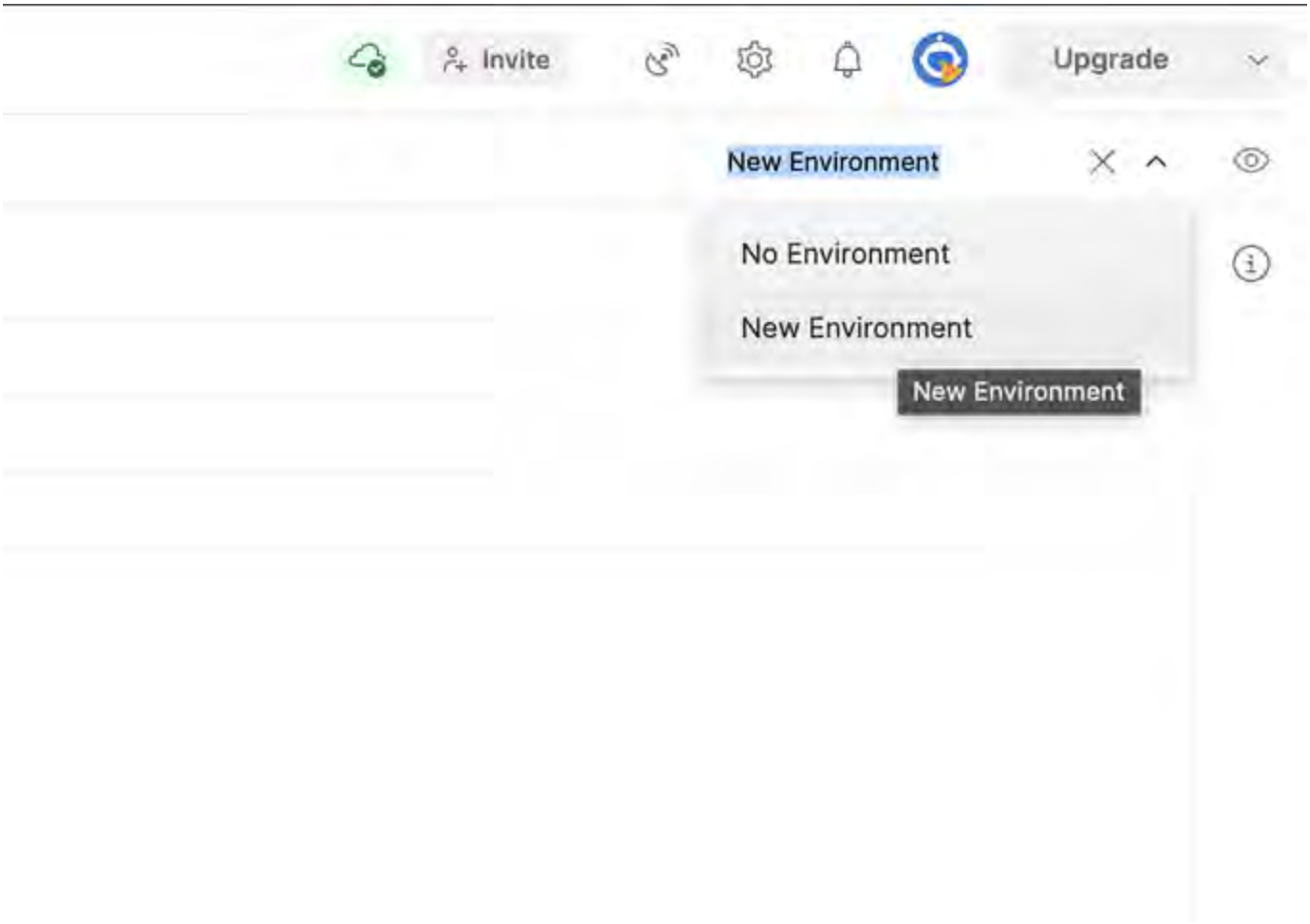
2. Click **Add** to add a new environment.



3. Add the **api_key** variable name and your token in the initial value.



4. Select the New Environment from the list.



5. You can now test the requests in the REST Occupancy Collection.

OpenAPI Documentation

Visualize OpenAPI (Swagger) documentation app

Export to PDF of the OpenAPI specification is not supported. See interactive documentation online.

GraphQL Occupancy API

GraphQL Occupancy API offers similar functionality to REST Occupancy API, however, there are **additional benefits** to using GraphQL.

You can **select the fields** you want to receive from the API. By requesting only the required fields from the API you can decrease traffic and consequently increase load speed.

Additionally, GraphQL Occupancy API supports subscriptions, which allows you to receive **real-time** occupancy updates.

Authorization

In order to authorize with this API, you need to add the following header to your request:

Key	Value
Authorization	<your token>

Your token can be obtained from the Company Info section of the Nwave’s console.

GraphQL Schema

The following GraphQL schema describes data types and operations that can be performed.

schema.graphql

```
type GroupOccupancy @aws_api_key
@aws_lambda {
        id: ID!
        zoneId: Int
        levelId: Int
        name: String!
        groupType: String!
        customId: String
```

```
        location: Location
        positionsOccupancy: [PositionOccupancy]
        summary: OccupancySummary
}

input GroupOccupancyInput {
        id: ID!
        zoneId: Int
        levelId: Int
        name: String!
        groupType: String!
        customId: String
        location: LocationInput!
        positionsOccupancy: [PositionOccupancyInput!]!
        summary: OccupancySummaryInput!
}

type LevelOccupancy @aws_api_key
@aws_lambda {
        id: Int
        name: String
        zoneId: Int!
        floorNumber: Int
        summary: OccupancySummary
}

input LevelOccupancyInput {
        id: Int
        zoneId: Int!
        floorNumber: Int
        name: String
        summary: OccupancySummaryInput!
}

type Location @aws_api_key
@aws_lambda {
        lat: Float!
        lon: Float!
}

input LocationInput {
        lat: Float!
        lon: Float!
}

type Mutation {
        createSubscriptionArea(
                lat: Float,
                lon: Float,
                radius: Int,
```

```
                zoneId: [Int],
                levelId: [Int],
                floorNumber: [Int],
                groupId: [Int],
                labels: [String],
                granularity: SubscriptionGranularity
        ): SubscriptionAreaNoUpdates
                @aws_api_key
@aws_lambda
        updateSubscriptionArea(
                id: ID!,
                lat: Float,
                lon: Float,
                radius: Int,
                zoneId: [Int],
                levelId: [Int],
                floorNumber: [Int],
                groupId: [Int],
                labels: [String],
                granularity: SubscriptionGranularity
        ): SubscriptionAreaNoUpdates
                @aws_api_key
@aws_lambda
        updatePositionOccupancy(positionId: Int!, occupancyStatus:
OccupancyStatus!, timestamp: String!): PositionOccupancy
                @aws_api_key
        pushPositionUpdates(
                id: ID!,
                location: LocationInput,
                radius: Int,
                zoneId: [Int],
                levelId: [Int],
                floorNumber: [Int],
                groupId: [Int],
                labels: [String],
                granularity: SubscriptionGranularity!,
                expiresOn: AWSDateTime!,
                updates: [PositionOccupancyInput!]!,
                updateTime: String!
        ): SubscriptionAreaWithUpdates
                @aws_api_key
        pushGroupUpdates(
                id: ID!,
                location: LocationInput,
                radius: Int,
                zoneId: [Int],
                levelId: [Int],
                floorNumber: [Int],
                groupId: [Int],
                labels: [String],
```

```
                granularity: SubscriptionGranularity!,
                expiresOn: AWSDateTime!,
                updates: [GroupOccupancyInput!]!,
                updateTime: String!
        ): SubscriptionAreaWithUpdates
                @aws_api_key
        pushLevelUpdates(
                id: ID!,
                location: LocationInput,
                radius: Int,
                zoneId: [Int],
                levelId: [Int],
                floorNumber: [Int],
                groupId: [Int],
                labels: [String],
                granularity: SubscriptionGranularity!,
                expiresOn: AWSDateTime!,
                updates: [LevelOccupancyInput!]!,
                updateTime: String!
        ): SubscriptionAreaWithUpdates
                @aws_api_key
        pushZoneUpdates(
                id: ID!,
                location: LocationInput,
                radius: Int,
                zoneId: [Int],
                levelId: [Int],
                floorNumber: [Int],
                groupId: [Int],
                labels: [String],
                granularity: SubscriptionGranularity!,
                expiresOn: AWSDateTime!,
                updates: [ZoneOccupancyInput!]!,
                updateTime: String!
        ): SubscriptionAreaWithUpdates
                @aws_api_key
}

enum OccupancyStatus {
        Occupied
        Free
}

type OccupancySummary @aws_api_key
@aws_lambda {
        total: Int!
        occupied: Int!
        available: Int!
        undefined: Int!
}
```

```
input OccupancySummaryInput {
        total: Int!
        occupied: Int!
        available: Int!
        undefined: Int!
}

type PositionOccupancy @aws_api_key
@aws_lambda {
        id: ID!
        customId: String
        groupId: Int
        occupancyStatus: OccupancyStatus
        statusChangeTime: AWSDateTime
        location: Location!
}

input PositionOccupancyInput {
        id: ID!
        customId: String
        groupId: Int!
        occupancyStatus: OccupancyStatus!
        statusChangeTime: AWSDateTime!
        location: LocationInput!
}

type Query {
        groupOccupancy(id: ID!): GroupOccupancy
                @aws_api_key
@aws_lambda
        findGroupOccupancies(
                ids: [Int],
                lat: Float,
                lon: Float,
                radius: Int,
                levelId: [Int!],
                labels: [String!],
                floorNumber: [Int!],
                zoneId: [Int!],
                projectId: Int,
                groupCustomId: String,
                limit: Int,
                offset: Int
        ): [GroupOccupancy]
                @aws_api_key
@aws_lambda
        findPositionOccupancies(
                ids: [Int],
                lat: Float,
```

```
                lon: Float,
                radius: Int,
                groupId: [Int!],
                levelId: [Int!],
                labels: [String!],
                floorNumber: [Int!],
                zoneId: [Int!],
                projectId: Int,
                groupCustomId: String,
                limit: Int,
                offset: Int
        ): [PositionOccupancy]
                @aws_api_key
@aws_lambda
}

union RtaUpdateObject = ZoneOccupancy | LevelOccupancy |
GroupOccupancy | PositionOccupancy

type Subscription {
        onSubscriptionAreaUpdates(id: ID!):
SubscriptionAreaWithUpdates
                @aws_api_key
@aws_lambda
@aws_subscribe(mutations: ["pushPositionUpdates","pushGroupUpdates","
pushLevelUpdates","pushZoneUpdates"])
}

interface SubscriptionArea {
        id: ID!
        location: Location
        radius: Int
        zoneId: [Int]
        levelId: [Int]
        floorNumber: [Int]
        groupId: [Int]
        labels: [String]
        granularity: SubscriptionGranularity!
        expiresOn: AWSDateTime!
}

type SubscriptionAreaNoUpdates implements SubscriptionArea
@aws_api_key
@aws_lambda {
        id: ID!
        location: Location
        radius: Int
        zoneId: [Int]
        levelId: [Int]
        floorNumber: [Int]
```

```
        groupId: [Int]
        labels: [String]
        granularity: SubscriptionGranularity!
        expiresOn: AWSDateTime!
}

type SubscriptionAreaWithUpdates implements SubscriptionArea
@aws_api_key
@aws_lambda {
        id: ID!
        location: Location
        radius: Int
        zoneId: [Int]
        levelId: [Int]
        floorNumber: [Int]
        groupId: [Int]
        labels: [String]
        granularity: SubscriptionGranularity!
        expiresOn: AWSDateTime!
        updates: [RtaUpdateObject]
        updateTime: AWSDateTime
}

enum SubscriptionGranularity {
        Position
        Level
        Group
        Zone
}

type ZoneOccupancy @aws_api_key
@aws_lambda {
        id: ID!
        name: String!
        projectId: Int
        summary: OccupancySummary
}

input ZoneOccupancyInput {
        id: ID!
        name: String!
        projectId: Int
        summary: OccupancySummaryInput!
}

# @api_key (admin) auth and @aws_lambda (user) auth are required for
all types apart from:
##   updatePositionOccupancy
##   pushPositionUpdates
##   pushGroupUpdates
```

```
##   pushLevelUpdates
##   pushZoneUpdates
## These are for internal subscription notification and should have
exclusive admin access
schema {
        query: Query
        mutation: Mutation
        subscription: Subscription
}
```

Subscription Areas

A subscription area is an object that has two main functions:

1. Filtering of the incoming position updates
2. Defining the format of updates that is received by subscribers

There are two types defined in the GraphQL schema for Subscription Areas:

`SubscriptionAreaNoUpdates` - this type is returned when you create or update a Subscription Area. It does not have 'updates' & 'updateTime' fields as updates are only returned to active subscriptions.

Subscription areas expiry 5 minutes after creation. Any update of a subscription area, including an empty one, will extend the expiration time by 5 minutes from the time of update.

`SubscriptionAreaWithUpdates` - this type is returned to subscribers and as the name suggests it contains the 'updates' & 'updateTime' fields.

Subscription Area Filters

Subscription Areas filter the incoming updates and only notify the subscribers if the incoming position update matches all of the filters. Subscription Area filters can also be split into 2 categories:

1. Hierarchical Filters: zoneId, groupId, levelId, floorNumber, labels
2. Geospatial Filters: lat, lon, radius

A valid subscription area must have at least one hierarchical filter and/or all of the geospatial filters.

A match is when all position attributes are a **subset ()** of the Subscription Area filters and it implies that a given position is inside a Subscription Area.

Subscription Area filter with **null** value matches everything.

The following example is a valid match between Subscription Area and Position Update.

Position

1. zoneId: **1**
2. groupId: **2**
3. levelId: **3**
4. floorNumber: **4**
5. labels: [**'EV', 'Disabled'**]
6. lat: **0.0**
7. lon: **0.0**

Subscription Area

1. zoneId: [**1**, 2, 3]
2. groupId: [1, **2**, 3]
3. levelId: [**3**]
4. floorNumber: **null**
5. lables: [**'EV', 'Disabled'**, 'VIP']
6. lat: **0.0**
7. lon: **0.0**
8. radius: **100**

Geospatial filters create a circular search area on the map, and there can be cases when a **group** of devices **partially** falls into the search area. (e.g. Half a group is within the search area and half is outside of the search area).

Updates for positions that do not fall into the search area would still notify the subscribers.

Subscription Area Updates

The updates field of a Subscription Area can contain a list of 4 different types. All types within the updates list are the same. The type you will receive in the updates field depends on the granularity you choose.

Granularity	Updates Type
Zone	ZoneOccupancy
Level	LevelOccupancy
Group	GroupOccupancy
Position	PositionOccupancy

Updates field will always contain only the object that has changed and therfore the list should always have a length of 1.

Object & Field Descriptions

SubscirptionArea

- id - subscription area id
- location - center coordinates of the search area
- radius - radius from the center in meeters that creates the search area
- zoneId - list of zone ids to match
- levelId - list of level ids to match
- floorNumber - list of floor numbers to match
- groupId - list of group ids to match
- labels - list of labels to match
- granularity - defines the updates type
- expiresOn - timestamp of subscription area expiration
- udpates - occupancy updates for a subscription area
- updateTime - timestamp of the occupancy update

ZoneOccupancy

- id - zone id
- name - zone name
- projectId - project id
- summary - summary of occupancies in the Zone

LevelOccupancy

- id - level id
- name - level name
- zoneId - zone id
- floorNumber - floor number of a level
- summary - summary of occupancies on a Level

GroupOccupancy

- id - group id
- zoneId - zone id
- levelId - level id
- name - group name
- groupType - group type e.g. marked_bay
- customId - user defined group id
- location - enter coordinates of a group
- positionOccupancy - list of PositionOccupancy objects for a group
- summary - summary of occupancies in the Group

PositionOccupancy

- id - position id

- customId - custom id
- groupId - group id
- occupancyStatus - ‘occupied’, ‘free’ or null
- statusChangeTime - timestamp of last occupancyStatus change
- location: coordinates of a position

The summary object should be used to display availability for all parking group types as it handles unmarked bay occupancies

Operations

The following operations can be performed using either Postman or curl command except for subscription operations.

groupOccupancy

This query will return the occupancy of a single group.

Query

```
query MyQuery {
  groupOccupancy(id: 123) {
    id
    customId
    name
    location {
      lat
      lon
    }
    positionsOccupancy {
      customId
      id
      groupId
      location {
        lat
        lon
      }
      occupancyStatus
      statusChangeTime
    }
  }
}
```

Response

```
{
  "data": {
    "groupOccupancy": {
      "id": 3544,
      "customId": null,
      "name": "Foo",
      "location": {
        "lat":
51.493601937687224,
        "lon":
-0.12852029611716095
      },
      "positionsOccupancy": [
        {
          "customId": "",
          "id": 1,
          "groupId": 123,
          "location": {
            "lat":
51.49360068522545,
            "lon":
-0.1286061268139623
          },
          "occupancyStatus":
"Free",
          "statusChangeTime":
"2021-01-27T19:22:47.548+00:
00"
        },
        {
          "customId": "",
          "id": 2,
          "groupId": 123,
          "location": {
            "lat":
51.493601937687224,
            "lon":
-0.12852029611716098
          },
          "occupancyStatus":
"Occupoed",
          "statusChangeTime":
"2021-01-27T18:56:53.502+00:
00"
        }
      ]
    }
  }
}
```

findGroupOccupancies

This query will return a list of group occupancies.

Query

```
query MyQuery {
  findGroupOccupancies(ids:
[1234]) {
    id
    location {
      lat
      lon
    }
    summary {
      available
      occupied
      total
      undefined
    }
  }
}
```

Response

```
{
  "data": {
    "findGroupOccupancies": [
      {
        "id": 1234,
        "location": {
          "lat":
50.780416909504915,
          "lon":
-1.0914407438684124
        },
        "summary": {
          "available": 34,
          "occupied": 13,
          "total": 47,
          "undefined": 0
        }
      }
    ]
  }
}
```

findPositionOccupancies

This query will return a list of position occupancies.

Query

```
query MyQuery {
  findPositionOccupancies
(ids: [1]) {
    id
    location {
      lat
      lon
    }
    occupancyStatus
    statusChangeTime
    groupId
    customId
  }
}
```

Response

```
{
  "data": {

"findPositionOccupancies": [
      {
        "id": 12345,
        "location": {
          "lat":
51.49360068522545,
          "lon":
-0.1286061268139623
        },
        "occupancyStatus":
"Free",
        "statusChangeTime":
"2021-01-27T19:22:47.548+00:
00",
        "groupId": 123,
        "customId": ""
      }
    ]
  }
}
```


createSubscriptionArea

This mutation will create a new subscription area.

Subscription Area Expiration

By default, subscription areas expire 5 minutes after creation. Once a subscription area is expired, it cannot be extended and subscribers will stop receiving updates.

Mutation

Response

```
mutation MyMutation {
  createSubscriptionArea(
    granularity: Group,
    lat: 51.493930,
    lon: -0.129030,
    radius: 100
  ) {
    expiresOn
    granularity
    id
    location {
      lat
      lon
    }
    radius
  }
}
```

```
{
  "data": {
    "createSubscriptionArea":
{
      "expiresOn": "2021-01-
28T23:20:06.232+00:00",
      "granularity": "Group",
      "id": 1,
      "location": {
        "lat": 51.49393,
        "lon": -0.12903
      },
      "radius": 100
    }
  }
}
```

onSubscriptionAreaUpdates

This subscription will subscribe to occupancy updates inside an area.

Selected updates object type must correspond to the granularity of the created search area:

1. granularity: Position updates { … on PositionOccupancy }
2. granularity: Group updates { … on GroupOccupancy }

If you are unsure of the granularity for your subscription area, you can specify both types inside the updates.

Subscription

Update

```
{
  "data": {

"onSubscriptionAreaUpdates": {
      "id": 1,
      "expiresOn": "2021-01-
28T23:20:06.232+00:00",
      "granularity": "Group",
      "location": {
        "lat": 51.49393,
        "lon": -0.12903
      },
      "radius": 1000,
      "updateTime": "2021-01-
28T23:17:38.400+00:00",
      "updates": [
        {
          "id": 123,
          "name": "Foo",
```

```
subscription MySubscription {
  onSubscriptionAreaUpdates
(id: 1) {
    id
    expiresOn
    granularity
    location {
      lat
      lon
    }
    radius
    updateTime
    updates {
      ... on GroupOccupancy {
        id
        name
        customId
        location {
          lat
          lon
        }
        positionsOccupancy {
          customId
          groupId
          id
          location {
            lat
            lon
          }
          occupancyStatus
          statusChangeTime
        }
        summary {
          undefined
          total
          occupied
          available
        }
      }
    }
  }
}
```


```
          "customId": null,
          "location": {
            "lat":
51.493601937687224,
            "lon":
-0.12852029611716095
          },

"positionsOccupancy": [
            {
              "customId": "",
              "groupId": 123,
              "id": 1,
              "location": {
                "lat":
51.49360068522545,
                "lon":
-0.1286061268139623
              },

"occupancyStatus": "Occupied",

"statusChangeTime": "2021-01-
28T23:17:38.400+00:00"
            }
          ],
          "summary": {
            "undefined": 0,
            "total": 1,
            "occupied": 1,
            "available": 0
          }
        }
      ]
    }
  }
}
```


updateSubscriptionArea

This mutation will update an existing subscription area.

All subscription area updates will extend expiration by 5 minutes from the current time.

You can also send empty mutation to only update the expiration.

Mutation

```
mutation MyMutation {
  updateSubscriptionArea(
    id: 1,
    granularity: Position,
    radius: 500
  ) {
    expiresOn
    id
    granularity
    location {
      lat
      lon
    }
    radius
  }
}
```

Response

```
{
  "data": {
    "updateSubscriptionArea":
{
      "expiresOn": "2021-01-
28T23:20:06.232134+00:00",
      "id": 1,
      "granularity":
"Position",
      "location": {
        "lat": 51.49393,
        "lon": -0.12903
      },
      "radius": 500
    }
  }
}
```

```
mutation ExtendSubscription {
  updateSubscriptionArea(
    id: 1,
  ) {
    expiresOn
    id
  }
}
```

```
{
  "data": {
    "updateSubscriptionArea":
{
      "expiresOn": "2021-01-
28T23:25:06.232134+00:00",
      "id": 1,
    }
  }
}
```

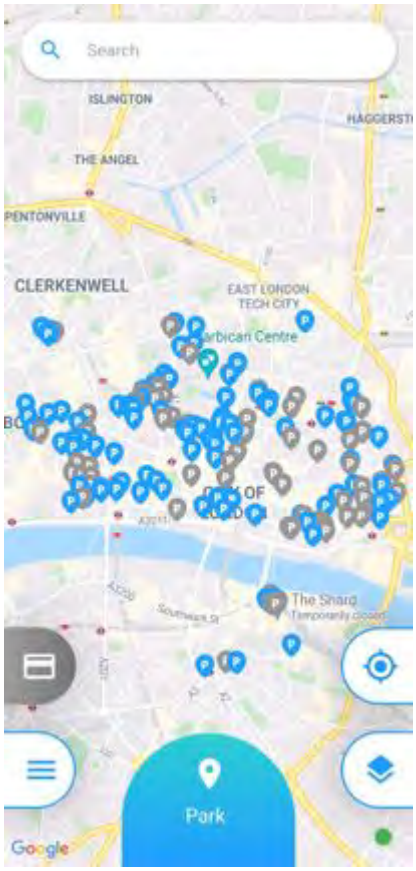
Use cases

Query	Use case

```
query MyQuery {
  findGroupOccupancies
(
    lat: 0.0,
    lon: 0.0,
    radius: 2000
  ) {
    location {
      lat
      lon
    }
    summary {
      available
    }
  }
}
```

The summary object should be used to display availability for all parking group types

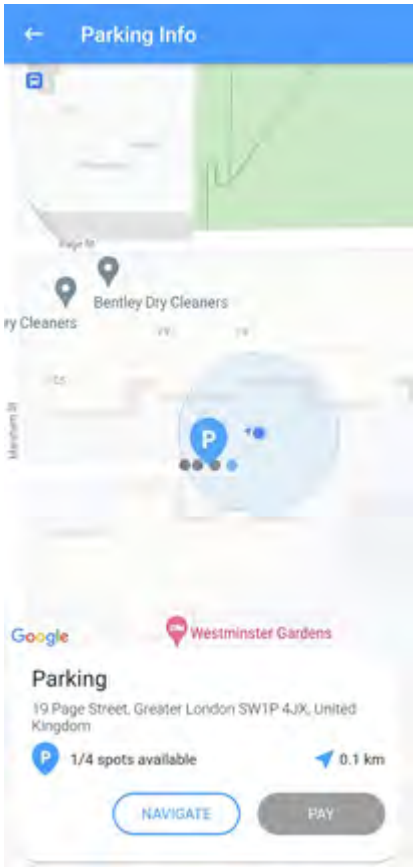
Displaying all parking groups within 2km in a mobile app



```
query MyQuery {
  groupOccupancy(id:
1) {

positionsOccupancy {
      location {
        lat
        lon
      }
      occupancyStatus
    }
  }
}
```

Displaying individual parking positions when a user approaches his desired parking location




```
query MyQuery {

findPositionOccupancie
s(
    lat: 0.0,
    lon: 0.0,
    radius: 2000,
    labels:
['Disabled']
  ) {
    id
    location {
      lat
      lon
    }
    occupancyStatus
  }
}
```

Displaying occupancy statuses of positions with label “Disabled” within 2km radius

```
mutation MyMutation {

createSubscriptionArea
(
    granularity:
Position,
    lat: 1.5,
    lon: 1.5,
    radius: 2000
  ) {
    id
  }
}
subscription
MySubscription {

onSubscriptionAreaUpda
tes(
  id: <id from
previous mutation>
  )
}
```

Displaying live parking statuses within 2km radius

```
mutation
LevelOccuppancyMutatio
n {

createSubscriptionArea
(
    zoneId: 3,
    granularity: Level
  ) {
    id
  }
}
subscription
MySubscription {

onSubscriptionAreaUpda
tes(
  id: <id from
previous mutation>
  )
}
```

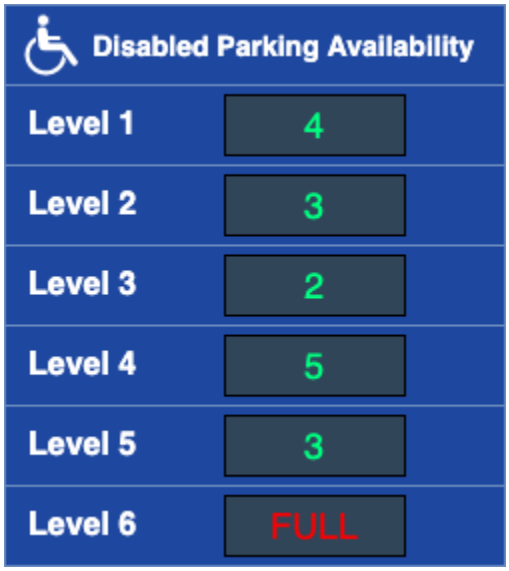
Displaying live occupancies per level in a zone (multistory car park)

```
mutation
LevelOccuppancyDisable
dMutation {

createSubscriptionArea
(
    zoneId: 3,
    labels:
"Disabled",
    granularity: Level
  ) {
    id
  }
}
subscription
MySubscription {

onSubscriptionAreaUpda
tes(
  id: <id from
previous mutation>
  )
}
```

Displaying live occupancies of disabled spaces per level in a zone




```
mutation MyMutation {

updateSubscriptionArea
(
    id: 1,
  ) {
    expiresOn
    id
  }
}
```

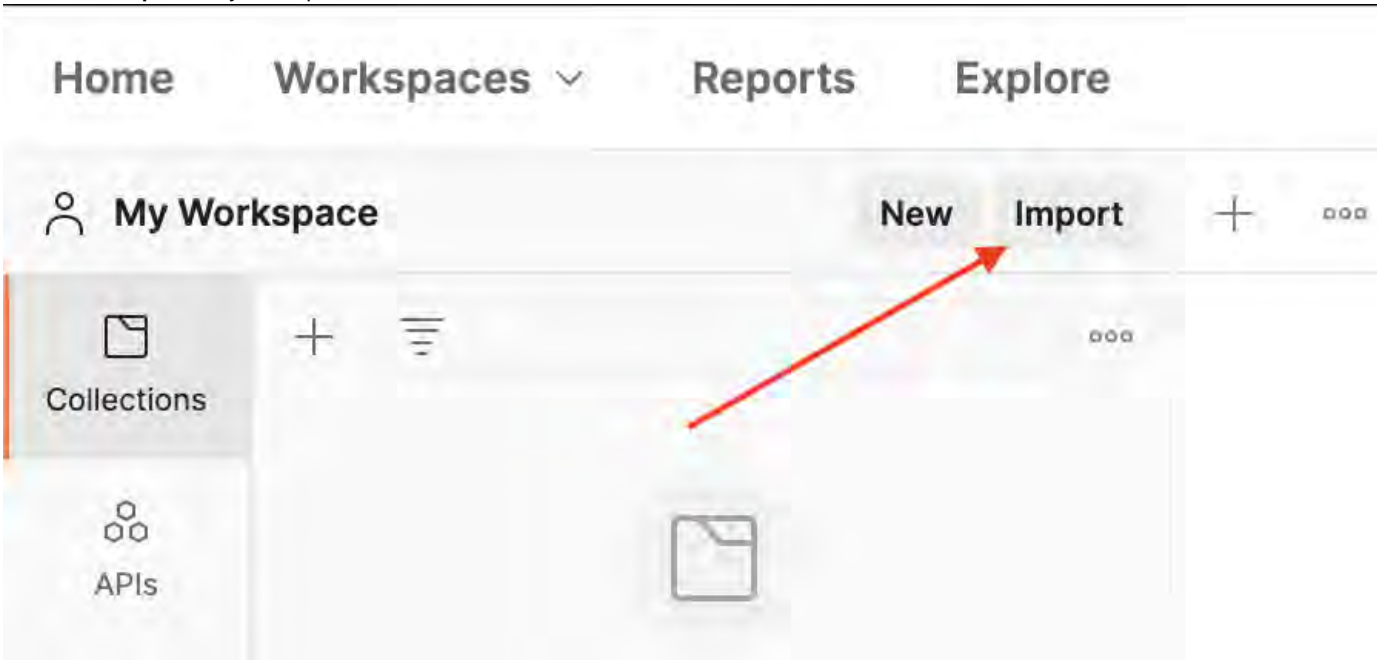
Extend subscription area expiration

Postman Collection

Download Postman Collection here:

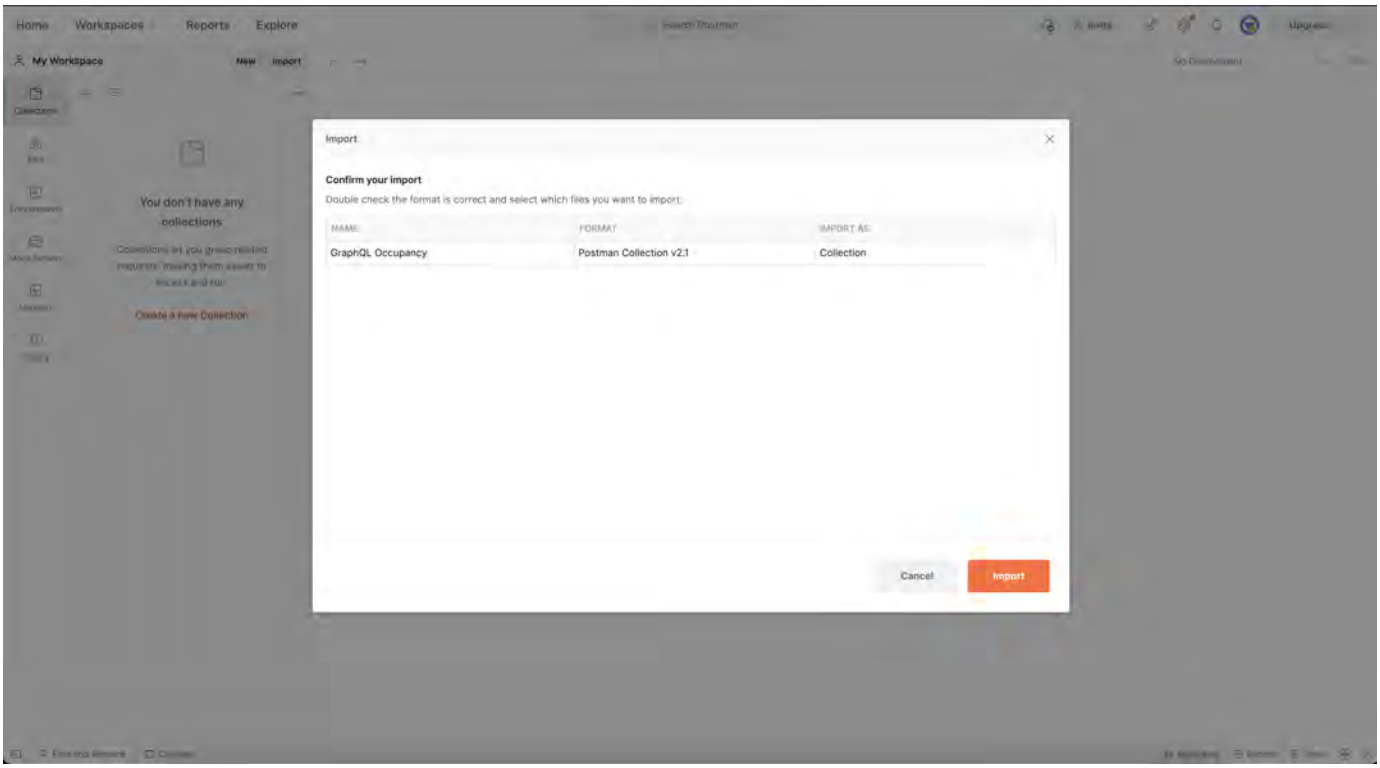
Importing Collection

1. Click **import** in My Workspace section.



1. Upload the collection file and click

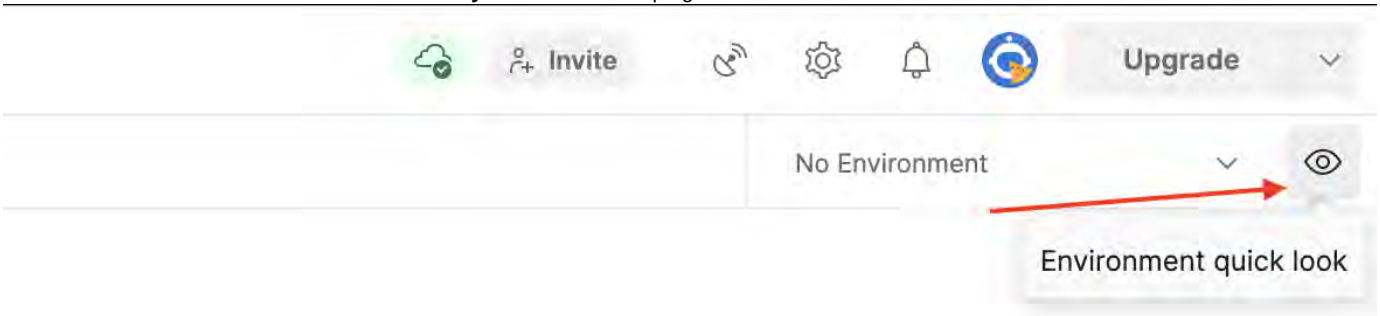
Import.



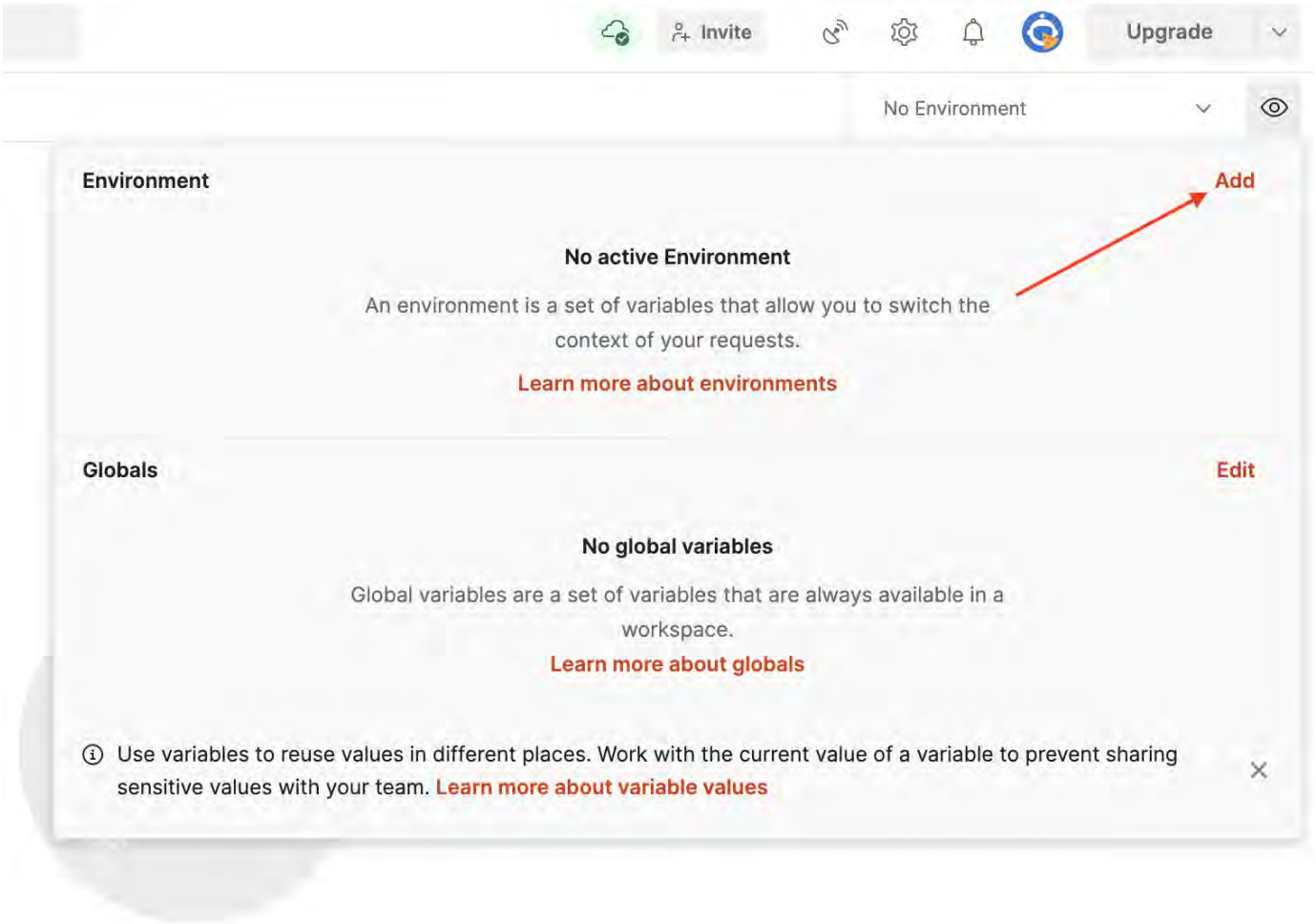
Adding API key to Postman Environment

This collection uses the **api_key** variable to add an authorization token to the x-api-key header.

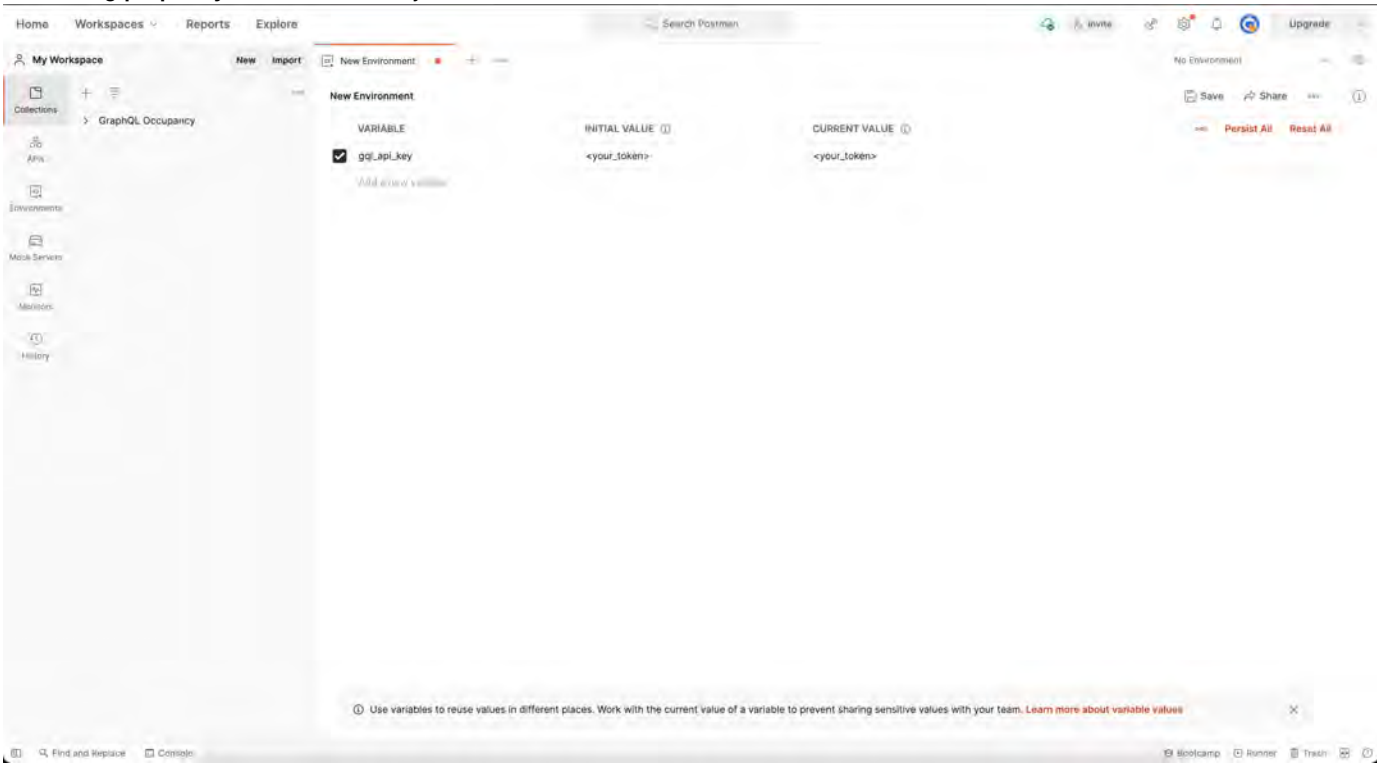
1. To create a new environment click on the **eye icon** near the top right corner.



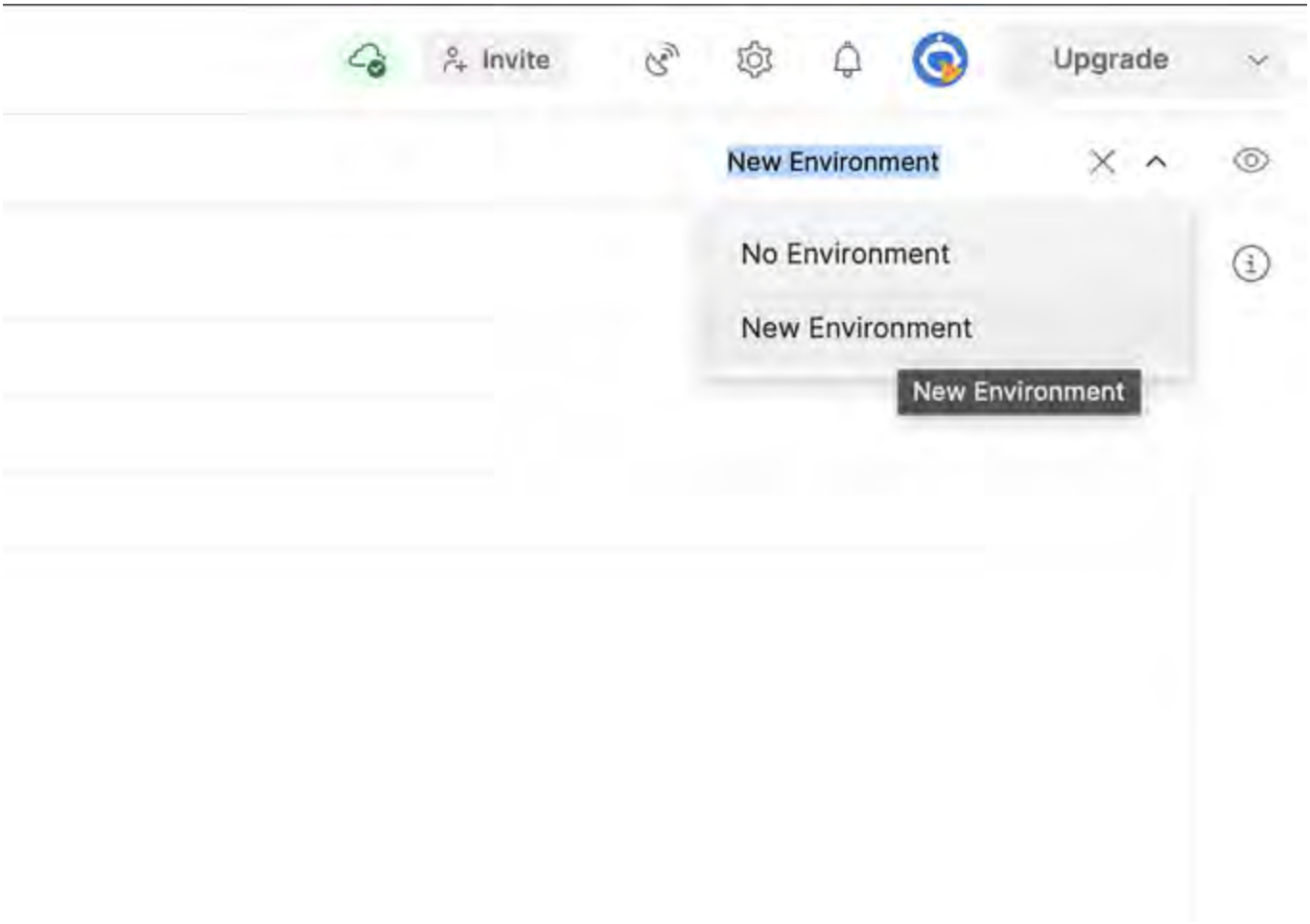
2. Click **Add** to add a new environment.



3. Add the **gql_api_key** variable name and your token in the initial value.



4. Select the New Environment from the list.



5. You can now test the requests in the GraphQL Occupancy Collection.