DAVANTIS API

Introduction

This is the developer manual for the DAVANTIS HTTP-API. This document will provide you with the guidelines to integrate DAVANTIS videoanalytics devices into a third party software. This guide is meant to develop a client from scratch. The client will be able to receive DAVANTIS alarms and access to additional functionality of a DAVANTIS device such as: live camera, inputs/outputs management.

Copyright

Confidential notice

Required knowledge

- Basic understanding of HTTP is required.
- Knowledge about DAVANTIS videoanalytics servers is useful but not required.

Additional documentation

As this manual does not provide full details on methods and models, it should be used alongside the full documentation of the API.

If you don't have the documentation contact DAVANTIS to obtain it.

Validation

Once development is completed at the third-party side, it is required that DAVANTIS technical team validates the integration in order to assure its stability and usability. Only validated integrations will be officially allowed.

Please contact DAVANTIS technical team (+34) 935868993 / support@DAVANTIS.com to schedule a validation of the new integration.

Code samples

The code samples are all in Python 3. It is used for its simplicity and easy to read syntax, but all samples can be easily translated to other languages with http libraries.

It is assumed in all code samples that we have imported the `http.client` module (included in the default Python 3 installation) and we have created an HTTP connection like this:

```
import http.client

connection = http.client.HTTPConnection('192.168.1.100:21000')
```

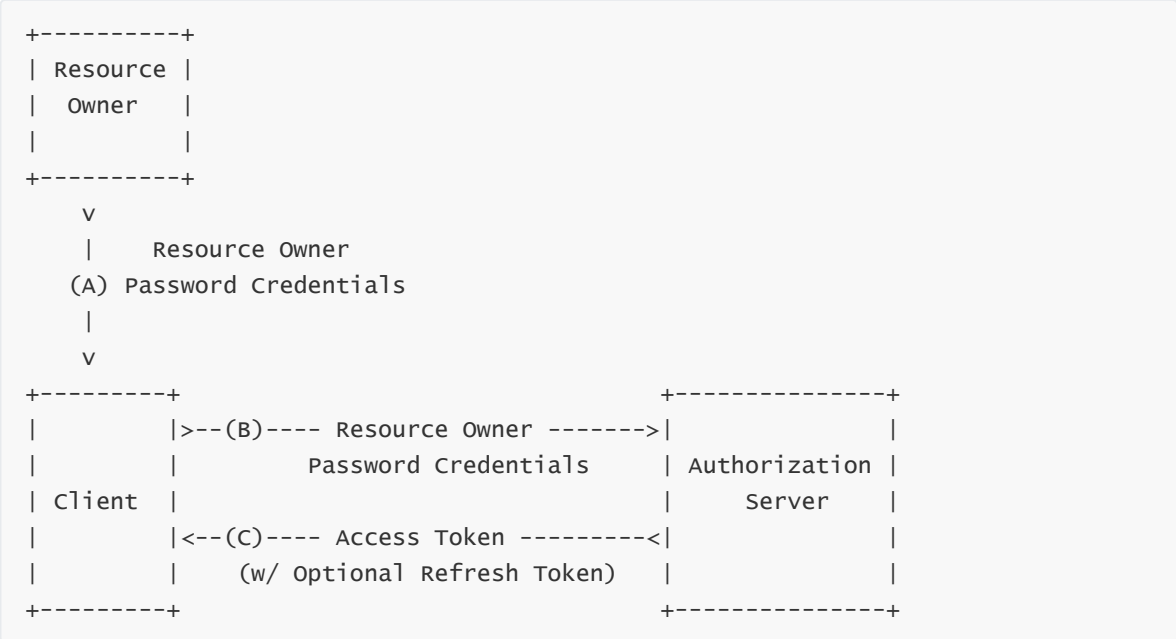
Where `192.168.1.100` is the IP of the DAVANTIS site you are trying to connect to. The default port for the API is 21000.

Authentication

The authentication method of the DAVANTIS API follows the Resource Owner Password Flow. All resources require authentication in the form of a Bearer Access Token that should be included in the HTTP header.

Obtaining an access token

Username and password credentials are sent to the DAVANTIS server using `POST /oauth/token` and it responds with an access token.



Example code to obtain a token (using username `admin` and an empty password):

```
body = '?grant_type=password&username=admin&password='
connection.request('POST', '/oauth/token', body)

print(connection.getresponse().read().decode())
```

Output:

```
{"access_token":"bt2HtS51DK5MPLZD","token_type":"bearer","expires_in":3599,"refr
esh_token":"V3Ie7G6xznWkgzQY"}
```

The grant_type for this operation must always be `password`. The credentials are the ones of the users configured in the DAVANTIS software through the desktop interface. This API does not allow the creation of new users.

The `expires_in` field is the number of seconds until the access token will no longer be valid.

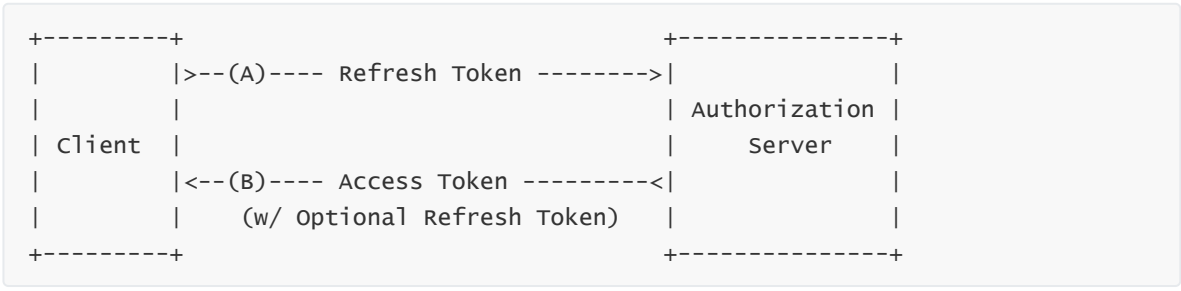
To use the token we have to add it in the `Authorization` header like this:

```
auth_header = {'Authorization': 'Bearer bt2HtS51DK5MPLZD'}

connection.request('GET', '/installation', body=None, headers=auth_header)
```

Refreshing a token

When the access token expires you will get an errorcode of `keymanagement.service.access_token_expired` when trying to access a secured resource. To get a new access token you should use the refresh token with the `refresh_token` grant type.



Example code to refresh a token:

```
body = 'grant_type=refresh_token&refresh_token=V3Ie7G6xznWkgzQY'
connection.request('POST', '/oauth/token', body)

print(connection.getresponse().read().decode())
```

Output:

```
{"access_token":"dpXenHObdVfc7M9x","expires_in":3599,"refresh_token":"yChQNGt2yHVv7Zuh"}
```

With the new access token and refresh token, you can repeat the process indefinitely.

Refreshing the token allows you to not store the user credentials in your application.

Cameras

A DAVANTIS site consists of one or more cameras that generate alarms based on the analyzed images. One of the most basic information is getting a list of all the cameras in the site.

Obtaining a list of cameras

The method `GET /cameras` returns an array of camera objects containing all cameras of the site.

For example, we can get all the cameras and print the names of the ones that are thermal cameras:

```
import json

auth_header = {'Authorization': 'Bearer dpXenHObdVfc7M9x'}

connection.request('GET', '/cameras', body=None, headers=auth_header)

response = connection.getresponse()
cameras = json.loads(response.read().decode())

for cam in cameras:
    if cam['thermal']:
        print(cam['alias'])
```

Live stream

It might be useful in some situations to see the live stream of a camera. For that you can use `GET /cameras/{id}/stream/mjpeg`. As you might have guessed the format is Motion JPEG. The live stream includes the same marking of intrusions you see on alarm videos and snapshots.

This sample opens a web browser with the stream of the first camera:

```
import json
import webbrowser

auth_header = {'Authorization': 'Bearer dpXenHObdVfc7M9x'}

#Get all cameras
connection.request('GET', '/cameras', body=None, headers=auth_header)
response = connection.getresponse()
cameras = json.loads(response.read().decode())

#Open /cameras/{id}/stream/mjpeg with the id of the first camera
webbrowser.open('http://' + connection.host + ':' + str(connection.port) +
'/cameras/' + str(cameras[0]['id']) + '/stream/mjpeg')
```

Alarms

If an intruder is detected by a camera, an alarm is generated. Alongside the alarm, an snapshot and a video are generated to aid the operator in the verification of the alarm. Obtaining DAVANTIS alarms is the most important feature of this API. It can be used to retrieve past alarms or to check periodically if there are new alarms to show them in real time.

Obtaining a list of all alarms

Obtaining a list of alarms can be done with `GET /alarms`.

This example obtains a list of alarms and prints the starting time and description of each of them:

```
import json

auth_header = {'Authorization': 'Bearer dpXenHObdVfc7M9x'}

connection.request('GET', '/alarms', body=None, headers=auth_header)

response = connection.getresponse()
alarms = json.loads(response.read().decode())

for alarm in alarms:
    print(alarm['started'] + ' - ' + alarm['description'])
```

> Note: The number of alarms returned has a maximum, by default 300. They are ordered by time in descending order.

Filtering alarms

Usually a client will not want to obtain all alarms generated by a DAVANTIS site, in this section we will explain the different types of alarms so you can decide which are the ones that you need and show you how to filter the rest.

Filters are encoded using a query string in the url.

Disarmed alarms

One of the most important concepts to understand is that at any given moment, some or all cameras could be unarmed. Arming and disarming cameras is an action that is usually performed by people on site and, when a camera is unarmed, people or vehicles that may appear in the images should not be considered real intrusions. This should not be confused with the `active` state of a camera, as a deactivated camera never generates alarms even if the system is armed.

In the default configuration, the site will generate alarms even if a camera is disarmed. This can be used for maintenance and statistical purposes. However, most users will just want to see alarms generated when the system was armed, specially at monitoring stations, since they only want to detect real intrusions in a costumer's site. This is the purpose of the `cra` filter.

To get only the alarms generated when the camera was armed you need to set the `cra` filter to `yes`:

```
query = '?cra=yes'
connection.request('GET', '/alarms' + query, body=None, headers=auth_header)
```

To get only the alarms generated when the camera was disarmed you need to set the `cra` filter to `no`:

```
query = '?cra=no'
connection.request('GET', '/alarms' + query, body=None, headers=auth_header)
```

To get both, just not apply the filter.

Filter by date

Another useful filter when obtaining alarms, specially to review old alarms, is to filter by date with `min_date` and `max_date`. You can use one or both at the same time.

The filters use the same format as any date returned by the API: ISO 8601. It is recommended that you specify the time zone when using date filters, but if no time zone information is provided the API parses it as its local time zone.

> Note: In the `GET /installation` method you can obtain the time zone of the installation in the `current_utc_offset` parameter.

This is an example of how to filter alarms by date:

```
#Filter from the start of 21st of april to 2:30 of the same day (both in UTC)
query = '?' + 'min_date=2016-04-21Z&max_date=2016-04-21T02:30:00Z'
connection.request('GET', '/alarms' + query, body=None, headers=auth_header)
```

Filtering by camera

Filters can be used to obtain certain info associated to specific cameras, for example, a typical requet would be get alarms of an specific amount of cameras.

Here is an example of how to filter alarms by camera(s):

```
query = '?' + 'camera_id=0,3,4'
connection.request('GET', '/alarms' + query, body=None, headers=auth_header)
```

> Note: In DAVANTIS system camera 0 means camera system and all system alarms (excepting specific system alarmas of each cam, such as camera lost/recover) are associated to camera 0

Filtering by event

Filters can be used to obtain certain info associated to specific event, for example, a typical requet would be get alarms of an specific amount of events.

Here is an example of how to filter alarms by event(s):

```
query = '?' + 'event_id=102,11,10,0'
connection.request('GET', '/alarms' + query, body=None, headers=auth_header)
```

> ## Note:
>
> ## There are two types of system events:
>
> - 0 => generic system events
> - 102 => armed/disarmed event
>
> ## And seven types of detection events:
>
> - 1 => Movement

> - 2 => Person
> - 3 => Vehicle
> - 7 => Other
> - 10 => Sabotage
> - 11 => Intrusion
> - 18 => External trigger

Real time alarm monitoring

Most clients would like to alert the user in real time if an intrusion is detected. For this, the client must implement a pooling method to request `/alarms` periodically. How the client does this is up to you but we recommend doing it in the following way:

1. Obtain alarms and store the id of the most recent alarm.
2. Wait some time
3. Obtain alarms again, using the `min_id` filter with the value of the id we stored earlier. If you get new alarms, store the id of the most recent one.
4. Repeat

As the id's of the alarms increase over time, if you filter by `min_id` with the newest alarm you have, you will only receive new alarms. This request is very fast so don't hesitate to do it several times per second if your application needs it.

> Note: You probably want to filter the alarms that are generated when the site is not armed. See the Disarmed Alarms section for more information.

Here is an example of the pooling mechanism in python:

```
import json
import time

auth_header = {'Authorization': 'Bearer dpXenHObdVfc7M9x'}

#Get all alarms
connection.request('GET', '/alarms', body=None, headers=auth_header)
response = connection.getresponse()
alarms = json.loads(response.read().decode())
min_id = alarms[0]['id'] #As alarms are sorted by id in descending order, the
first alarms is always the newest one
print('Starting alarm: ' + str(min_id))

while(True):
    time.sleep( 0.5 )

    #Get alarms newer than the newest alarm we have
    query = '?' + 'min_id=' + str(min_id)
    connection.request('GET', '/alarms' + query, body=None, headers=auth_header)

    response = connection.getresponse()
    response_string = response.read().decode()

    #If there are no new alarms the status code will be 204 NO CONTENT.
    if response.status == 200:
        #If we have new alarms decode them and update the min_id
        alarms = json.loads(response_string)
```

```
        min_id = alarms[0]['id']
        for alarm in alarms:
            print('New alarm: ' + alarm['description'])
```

Obtaining alarm video and snapshot

To download a snapshot you use `GET /alarms/{idalarm}/snapshot` and to download a video you use `GET /alarms/{idalarm}/video`. The format of the files returned are in the 'Content-Type' header. All snapshots are in jpeg format, but videos can be in mp4, wmv or avi.

Some alarms that are consider "system" alarms do not have snapshot neither video associated. For example, alarms that are generated when a camera is activated/deactivated or when state of the inputs change. In addition, videos take time to record so it could be possible to obtain an alarm that has not yet finished recording. To check if the media associated is available, we have the `snapshot_status` and `video_status` fields in the alarm model. They can have 3 values: `ready`, `recording` (it will be ready when it finishes) or `unavailable` (there is no file associated with this alarm). If you try to download a file that is not `ready` you will get a 404 error. It is recommended to show the user this status, specially when a alarm is still recording, because it can be confusing to not have a video of an intruder alarm. By default, a video takes 10 seconds to record.

Here is sample code that finds the latest alarm with `video_status` as `ready` and downloads its video file:

```
import json

auth_header = {'Authorization': 'Bearer dpXenHObdVfc7M9x'}

alarm_id = 0

#Get latest alarms
connection.request('GET', '/alarms', body=None, headers=auth_header)
response = connection.getresponse()
alarms = json.loads(response.read().decode())

#Find an alarm with video_status ready
for alarm in alarms:
    if alarm['video_status'] == 'ready':
        alarm_id = alarm['id']
        break

#Get alarm video
connection.request('GET', '/alarms/'+ str(alarm_id) + '/video', body=None,
headers=auth_header)
response = connection.getresponse()
response_bytes = response.read()

ctype = response.getheader('Content-type') #If the file is, for example, in mp4
this will be video/mp4
extension = ctype[len('video/'):] #Remove the first part of content-type so we
are left with the extension
with open('alarm' + str(alarm_id) + '.' + extension, 'wb') as f:
    f.write(response_bytes)
```

Deterrent measures

At a remote alarm monitoring station, after the CMS operators has confirmed that the alarm was triggered by a real intrusion, it may be needed to activate deterrent measures such as sirens, lights, etc, to dissuade the intruders before the authorities arrive at the site. In a DAVANTIS site this measures are controlled by relays (device outputs), and although what those relays may activate afterwards is up to the site's owner, the API provides information for the user to know what they do so they can activate them appropriately.

To get all the available relays you can use `GET /relays`. Each relay has an id, an alias and a status (a `true` status means the relay is activated at this moment). The alias is a string that describes what the relay activates. It can take the values `light`, `sound`, `water`, `panic` or `other`.

To activate one of these relays, you have to use `PATCH /relays/{id}`. In a PATCH method you include in the body a partial model of the resource you want to modify, in this case a relay, and the parameters supplied will be modified while leaving the rest untouched. If you want to activate a relay you need to set the `status` to `true`:

```
import json

auth_header = {'Authorization': 'Bearer dpXenHObdVfc7M9x'}

#Get all relays
connection.request('GET', '/relays', body=None, headers=auth_header)
response = connection.getresponse()
relays = json.loads(response.read().decode())

relay_id = relays[0]['id']

print('Activating relay %d: %s' % (relay_id,relays[0]['alias']))

relay_patch = json.dumps({'status': True})

connection.request('PATCH', '/relays/' + str(relay_id), body=relay_patch,
headers=auth_header)
```

Relays stay activated for an amount of time that can be configured at the site (by default is 30s).

Arming and disarming

Arming or disarming the cameras in a site is done by arming or disarming its associated inputs. If an input is armed, all the cameras associated with that input will be armed too.

To get a list of the inputs you can use `GET /inputs`.

```
import json

auth_header = {'Authorization': 'Bearer dpXenHObdVfc7M9x'}

#Get all inputs
connection.request('GET', '/inputs', body=None, headers=auth_header)
response = connection.getresponse()
inputs = json.loads(response.read().decode())

for inp in inputs:
    print('%d: %r' % (inp['id'],inp['status']))
```

The inputs can either be controlled by software or by hardware. If they are controlled by software you can use `PATCH /inputs/{id}` to arm or disarm the inputs. If they are controlled by hardware this API cannot change its status and you will get an error if you try.

An example of how to set the status of the first input to `true` (armed):

```
import json


auth_header = {'Authorization': 'Bearer dpXenHObdVfc7M9x'}

#Get all inputs
connection.request('GET', '/inputs', body=None, headers=auth_header)
response = connection.getresponse()
inputs = json.loads(response.read().decode())

input_id = inputs[0]['id']

print('Activating input %d' % (input_id))

input_patch = json.dumps({'status': True})

connection.request('PATCH', '/inputs/' + str(input_id), body=input_patch,
headers=auth_header)

response = connection.getresponse()
response_s = response.read().decode()

if response.status == 200:
    print('Input %d activated' % (input_id))
else:
    print(response_s)
```