Neurospeed.IO API manual

For NeuroSpeed API version 2.1.7

Table of contents:

1. Installation:

pip install neurospeed

2. general information

public API for working with **NeuroSpeed.IO** cloud-based biosignal processing engine. NeuroSpeed.IO API main features:

- user-level access:
- operate raw sensor data
- operate processed data
- operate cognitive insights
- attach callbacks to streaming data packets for easy operation
- handle sensor connect/disconnect events
- operate data recorder - export multi-sensor recordings to file
- customer-level access:
- handle user connectivity

working with the API requires **neurospeed.io** user account. Get one for free at http://neurobrave.com

API code is public and open and is available at https://bitbucket.org/neurobrave/neurospeed_python_api

All API functions are nothing more than convenient wrappers of REST and Socket.IO calls; therefore, using NeuroSpeed API library is not mandatory but highly recommended for convenience.
Public NeuroSpeed API is currently available for Python; however, since it's just a bunch of wrappers for HTTP requests and socket subscribers, it could be rapidly translated into any language. Contact us at sales@neurobrave.com to inquire about implementation for your language of choice.

3. Authentication

3.1. General information

Authentication and working with neurospeed.io is possible on 2 levels of access:

1. customer
2. user

customer is “root” account that houses all of it’s “user” accounts. There is typically one “customer” account for an organization and multiple (unlimited number) of user accounts. Customer-level access grants access to data of all users. User-level access is isolated to data of that user.
An organization will create multiple customer accounts in cases where data isolation between groups/domains is required.

3.2. Setting up cloud access credentials

There are 2 levels of access to the cloud.

The highest level is Customer Access, required for:

- operating the cloud recorder features
- querying for active users

The lowest level is User Access, required for:

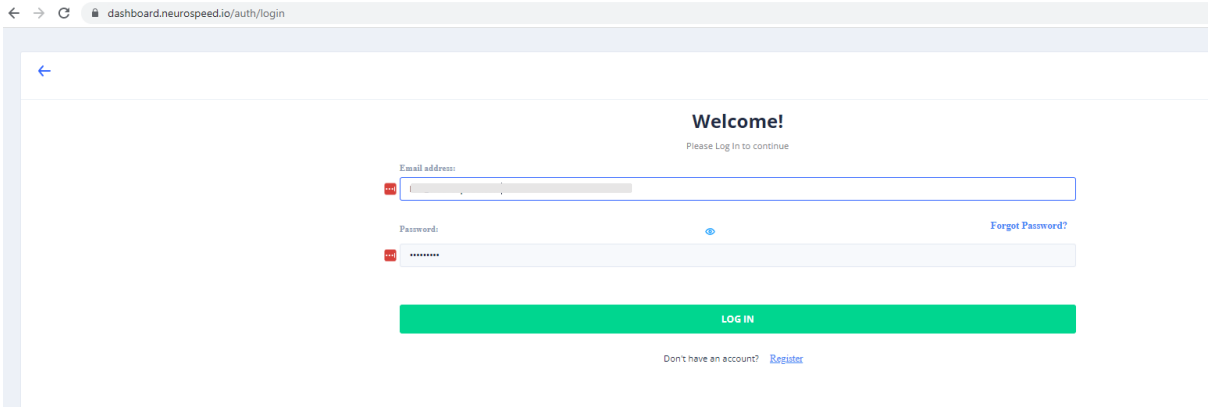
- up-streaming the raw data
- down-streaming processed data (markers, insights, statistics).

Credentials must be written in a python dictionary. It is possible to load a dictionary from a json structure stored in a text file. It is required to have the following credentials for both access levels.

for customer-level auth (to operate the recorder-exporter function)

```
customer_config = { "email": "my_dashboard_access_email_address",
                    "password": "my_dashboard_access_password"}
```

These are the same credentials used to log into *dashboard.neurpspeed.io*



If you are unable to log into the dashboard, please contact *support@neurobrave.com*

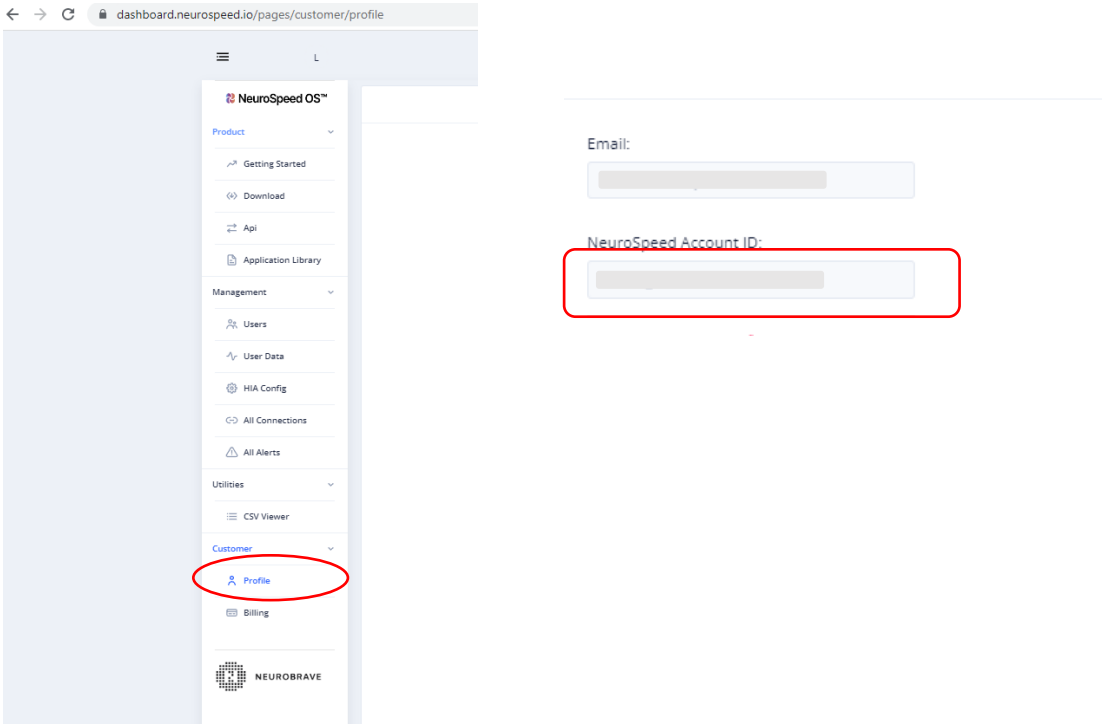
for user-level auth (to send the data)

user_config: {"account_id": "account_id_for_my_customer_profile",
"username": "username",
"user_password": "users_password"}

3.2.1. Locating account_id:

Visit *dashboard.neurospeed.io*. Using the left-hand navigation pane, navigate the *Profile* page.

On the *Profile* page there will be a field called “NeuroSpeed Account ID”. Copy the string inside the text box; use it for the *account_id* field of the user credentials dictionary:



3.2.2. Locating username

Visit *dashboard.neurospeed.io*. Using the left-hand navigation pane, navigate the *Users* page.

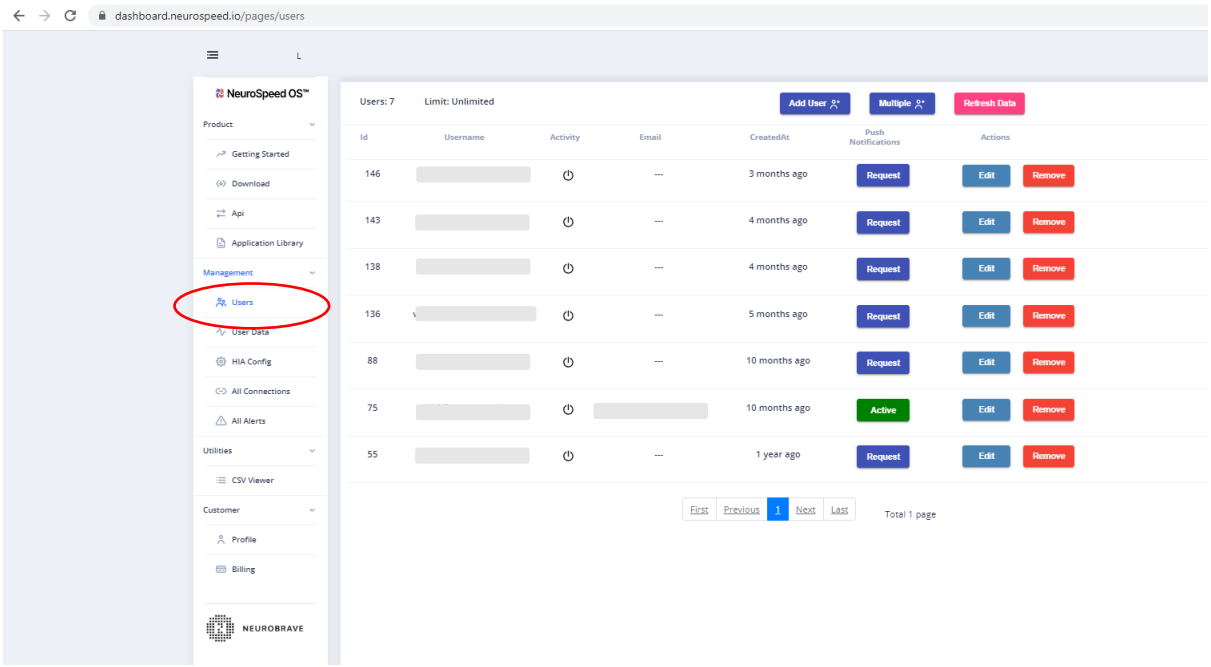
On the *Users* page there will be list of all the users available for the current customer. Note a user name for use with the API. use it for the *username* field of the user credentials dictionary:

In case there are no users shown, create a user instance first using 

button.

3.2.3. Locating the user-password

In case you do not remember the password for the user (it is NOT the same password you've used to access the dashboard), it's possible to reset the password using the Edit button.



3.3. authenticate as username-level access on neurospeed.io:

3.3.1. neurospeed.auth.auth_as_user_handler.Auth_AS_User_Handler(dict: user_config)

input: user_config – type dictionary with following fields:

```
{
   "email": "username@customersdomain.com",
   "password": "userpassord",
}
```

Returns
user_auth object

available methods for user_auth:

user_auth.login()
user_auth.get_username()

example:

from neurospeed.auth.auth_as_user_handler import Auth_AS_User_Handleruser_auth =
config_user1 = {“username”: “myusername”, “password”: “mypassword”}
Auth_AS_User_Handler(config_user1)
user_auth.login()

3.4. authenticate as customer-level access on neurospeed.io:

3.4.1. neurospeed.auth.auth_as_customer_handler.Auth_AS_Customer_handler(dict: customer_config)

input: customer_config – type dictionary with following fields:

```
{
   "email": "customer@customersdomain.com",
   "password": "customerspassword",
}
```

Returns
customer_auth object

3.4.2. available methods for custorer_auth:

3.4.2.1. customer_auth.login()

example:

```
from neurospeed.auth.auth_as_customer_handler import Auth_AS_Customer_handler
customer_config = {
   "email": "customer@customersdomain.com",
   "password": "customerpassword",
   }

customer_auth = Auth_AS_Customer_handler(customer_config)
customer_auth.login()
```

3.5. Helper functions:

3.5.1. neurospeed.utils.helper_service .UtilService.load_config_file(string: config_path)

helper function to load customer/user-level config from json file.
config_path is absolute path to .json file containing customer/user-level authentication credentials in the following format:

```
{
   "email": "customer@customersdomain.com",
   "password": "customerspassword",
}
```

Returns dictionary with credentials

example:

```
from neurospeed.utils.helper_service import UtilService
customer_config_path = os.path.join(str(Path().absolute()) ,"neurospeed\\config","customer_config.json")
customer_config = UtilService.load_config_file(customer_config_path)
```

4. downstream raw and processed data from neurospeed.io

4.1. general information

neurospeed.io outgoing data streams are organized like a chat room:
each user has “room” that their data is available in. to listen the data, a client must enter that room.
Multiple (unlimited) clients may listen to the room.
Each API instance is a client.

4.2. Streaming functions

4.2.1. neurospeed.api_socket_handlers.user_room_as_user_handler.UserRoom_AS_User_Handler(user_auth)

input: user_auth object
returns userRoom object that allows interfacing to data

4.2.2. available methods for userRoom

4.2.2.1. userRoom.set_data_external_handler(function: user_data_external_handler)

attach callback function that is invoked each time a data packet arrives. The function must receive the packet payload as an argument.
Inside that function the entire payload is available, including metadata (sensor information), raw sample data, processed data.

4.2.2.2. userRoom.set_device_events_external_handler(function: user_device_event_external_handler)

attach callback function to handle sensor connect/disconnect events.

4.2.2.3. userRoom.connect()

actually connect to the room and start listening to the data. Callback functions must be attached before connecting to the room

4.2.3. Example of subscribing to data stream

```
def user_data_external_handler(payload):
    username = user_auth.get_username()
    stream_id = payload["stream_id"]
    device_type = payload["device_type"]
    hia_id = payload["hia_id"]
    sensor_info = payload["sensor_info"]
    print("received data for %s form device %s"%(username, device_type)


userRoom.set_data_external_handler(user_data_external_handler)
```

4.2.4. packet payload structure

packet payload is type dictionary that has following fields:

4.2.4.1. payload metadata

payload["stream_id"]: string stream ID. Multiple streams may originate from single HIA.
payload["device_type"]: string describes sensor type “eeg”/”gsr”/”gamepad”/”custom”/et cetera
payload["hia_id"]: string 6-character HIA ID.
payload["sensor_info"]: dictionary sensor information

payload["sensor_info"]["sampling_frequency"] : integer or float sampling frequency
payload["sensor_info"]["channel_map"]: list of channel names

4.2.4.2. payload raw data

payload["data"]["sample"] : list of lists of multichannel data

4.2.4.3. payload processed data

payload["output"]['brainwave_power']['brainwave'] – list of values of frequency band power, for each individual channel
available bands:
'alpha', 'beta', 'theta', 'gamma', 'smr', 'delta', 'mu'

payload["output"]['cognitive_analysis']

list of values for each function cognitive analysis engine. To use this function, the functions must be activated at https://dashboard.neurospeed.io beforehand.

5. Upstream raw sensor data to neurospeed.io

5.1. General information

By far the best way to interface sensors to the cloud engine is to use the provided HIA (Hardware Interfacing Agent) utility. This simple program (Available on all platforms) provides multi-sensor streaming with direct support for a wide range of devices. It takes care of connectivity, configuration, timing, authentication. See HIA user manual for full feature list.

If you decide to send data manually after all, this section describes the way to perform this.

Possible reasons to implement data sending manually are: unsupported sensor type, custom data feed (maybe you want to log temperature together with sensor data, and HIA does not support temperature sensors).

5.2. Sensor info data structure

Each HIA client instance may subscribe to the cloud with multiple sensors.

Sensor_info is a simple data structure that contains the description of the sensor's basic properties. It does not contain the sensor data.

The properties for each sensor are:

Field	Description	Possible values
"sensor_id"	Unique data stream id for the sensor	String. May be any string, it's recommened to use the utility helper generator to create arbitrary strings that are guaranteed not to clash with any existing stream id's on the system. (see example)
"device_type	String, describes the type of sensor. Must be a value supported by NeuroSpeed.	Supported values: "EEG" "EDA" "GSR" "PPG" "keyboard" "gamepad" "user_data"
"channel_count"	Number of information (sample) channels. For example, for a 4-channel EEG device it'll be set to 4	Positive integer number. MUST match the channel map field. MUST match the actual payload to be sent.
"channel_map"	The data payloads are sent in order corresponding to this map. For example, for a 3-channel accelerometer data, this would be set to "channel_map" = ["X", "Y", "Z"]	List of strings. MUST match the "channel_count" field. MUST match the actual payload to be sent.

"sampling_frequency"	Used for cloud signal processing for the supported sensor types that require signal processing such as EEG and PPG.	Positive integer number
"buffer_length"	Optional parameter Length of each data packet in seconds. For example with sampling frequency of 100, if a data packets is to be transmitted every 0.1 seconds this should be set to 0.1	
"manufacturer_id"	Manufacturer name of the sensor	Arbitrary string
"sensor_id"		
"stream_state"	Optional parameter. State of the sensor.	“enabled” “disabled”

Changing sensor amount, types, properties is not possible on the fly. It is required to perform a disconnect() method, apply changes and perform connect() again.

Example of configuring a custom sensor type:

```
sensor_info = dict()
 user_data_stream_id =  "my_sensor_name_"' + UtilService.generateId(6)

 sensor_info[user_data_stream_id] =  {
 "device_type": "user_data",
 "channel_count": 3,
 "sampling_frequency": 600,
 "buffer_length" : 1.0,
 "manufacturer_id": "AnalogDevices",
 "sensor_id" : user_data_stream_id,
 "stream_id": user_data_stream_id,
 "stream_state" : "enabled",
  "channel_map" : ["X", "Y", “Z”]
 }
```

5.3. Upstream data interface functions

5.3.1. hia_sensor_info_user1 = get_sensors_mockup()

returns empty sensor information structure (dictionary) template.

5.3.2. HIA_Client(user1_auth, dict: hia_sensor_info_user1)

example:

hia_user1 = HIA_Client(user1_auth, dict: hia_sensor_info_user1)

inputs:

- user_auth data structure produced by the authentication module
- sensor info dictionary. The dictionary must have valid data fields, otherwise the NeuroSpeed cloud will reject the connection.

returns:

- HIA instance

5.3.3. available methods for HIA_Client instance:

5.3.3.1. hia_user1.update_sensor_info(string: first_stream_id, dict: my_custom_sensor_info)

update/append sensor information fields

Updating sensor info after connection is no allowed.

```
example:
my_custom_sensor_info = {
  'sensor_type': 'vehicle speed',
  'channel_count' : 2,
  'manufacturer_id': 'toyota'

}
```

5.3.3.2. hia_user1.set_socket_connection_external_handler(connection_external_handler)

5.3.3.3. hia_client.set_label(str label)

this function sets the label for all the transmitted data from this point onwards; so the data will appear labeled in the recorded/downloaded datafiles. Useful when conducting controlled experiments.

Inputs: label – string. Can be any valid string. Good example : “experiment_phase one”

Must not contain special characters or commas (“,”).

Returns: nothing

5.3.3.4. hia_user1.connect()

performs the connection to the cloud for the HIA client instance. Must be called before sending data packets.

inputs: nothing

Returns: nothing

5.3.3.5. hia_user1.is_connected()

returns Boolean value that reflects if the client is currently connected to NeuroSpeed cloud.

A client being connected is not necessarily equal to client sending packets.

5.3.3.6. hia_user1.send_data(list: data_packet_list_of_lists, string: stream_id)

puts data packet on a queue. The packet will be sent as soon as it reaches the end of the queue.

The queue is set up internally when initializing the HIA client instance. Packets are transmitted from the queue with constant rate limiting of 200mSec between packets. This does not limit the sampling rate since large amount of data may be in a packet (limited by socket limitation of 65535 bytes per packet). However, this affects latency as a new data will not reach the cloud faster than 200mSec. This is a safe method.

5.3.3.7. send_data_direct(list: data_packet, string: stream_id, timestamp=None, Boolean send_without_timestamp = False, string device_type = None)

this function sends data packet to the cloud immediately, without any rate limiting.

Inputs:

- data_packet - is type list of multichannel samples, each miltichannel sample is also type list
- stream id – must be existing steam id in the sensor_list of this instance of the HIA client.

  Hint: find steam id's by using this code:

  ```
  datatype = "user_data"
  stream_id = list(filter(lambda x: x.startswith(datatype), list(hia_sensor_info.keys())))[0]
  ```

- timestamp - must be list with same length as data_packet
- send_without_timestamp – if set to False; and there's no timestamp provided, the function will automatically generate timestamps for each sample. using current system time. If set to false, data will be sent out without a timestamp, and will receive a timestamp on the cloud.
- Device_type – string, must be a valid, supported device type. Must be of device type that matches the stream id field in the sensor_info dictionary that was used to initiate the HIA client instance.

Returns: nothing

5.3.3.8. hia_client.disconnect()

disconnects to the cloud for the HIA client instance. Closes the socket and lets the cloud know this sensor will not be sending any more data.

inputs: nothing

Returns: nothing

5.3.3.9. hia_client.get_sensor_info()

returns: sensor_info dictionary

5.3.3.10. hia_client.get_username()

returns: username string, that was used when constructing this instance of HIA client

5.3.3.11. hia_client.set_socket_connection_external_handler(handler)

inputs: hander – name of function that will be executed single time immediately when HIA client instance is connected

5.3.3.1. hia_client.set_socket_disconnect_external_handler(handler)

inputs: hander – name of function that will be executed single time immediately when HIA client instance is disconnected

6. Data recorder

6.1. General information

there can only be one recorder with status RECORDING at a time per USER. It is permissible to have multiple STOPPED recorders while also operating a RECORDING recorder.
previous active recorder must be stopped in order to create another one.
data still can be exported from stopped recorder, but recorder itself cannot be activated again.
actions available via recorder API:

- create_recorder(username):
- update_recorder(recorder_id, status):
- list_recorders(username)
- get_recorder(recorder_id):
- delete_recorder(recorder_id):

6.2. macros

the API library contains pre-built macros for high-level operation of the recorder/exporter interface. It is strongly recommended to use the macros. The macros take care of creating the recorder, polling for the correct status, handling edge cases such as a recorder being previously left “open” by mistake, HTTP timeouts, retries, et cetera.

To access the macros use the following import in your Python script:

from neurospeed import macros

6.2.1. recorder macros:

6.2.1.1. neurospeed.macros.start_recording(customer_auth, hia_config, recorder_name)

this function starts the recording session on NeuroSpeed. It blocks until the recorder is ready to receive data.

Inputs:

- Customer_auth object – generated by the neurospeed auithenticator module
- Hia_config – dictionary that contains the user-level credentials. Used to operate the recorder for the correct user.
- Recorder_name – a string that is used to identify the recorder. Does not have to be a unique string (it is permissible to have multiple recorders with the same recorder_name).

Returns:

- Recorder_id – integer number, uniquely identifies the recorder inside the userspace. Used to operate other macros.

6.2.1.2. stop_recording(recorder_id)

this function stop the recording session on NeuroSpeed. It blocks until the recorder is with status “stopped”.

Inputs:

- Recorder_id – integer number, uniquely identifies the recorder inside the userspace. Generated by start_recording() function. Must correspond to existing recorder ID on the cloud.

Outputs:

- Nothing

6.3. recorder control

6.3.1. neurospeed.api_http_handlers.recorder_http_handler.UserRoom_Recorder_Handler(customer_auth)

Input: customer authentication object (created by neurospeed.auth.auth_as_customer_handler.Auth_AS_Customer_handler())
Output : **recorder handle**r object

6.3.2. Recorder handler methods:

6.3.2.1. recorder_handler.create_recorder(username)

returns dictionary with following recorder information:

'request_status': “stopped”/”paused”/”pending”/”recording”
'recorder_name': string name,
'recorder_size': integer size in cloud memory/storage,
'columns': list of data columns,
'experiment_labels': list of experiment labels/markers, used when sending Marker stream during experiments
'stream_ids': list of sensor stream IDs as provided by HIA
'minimum_timestamp': timestamp of earliest packet
'maximum_timestamp': timestamp of latest packet
'id': integer that uniquely identifies the recorder within the user’s environment.
'status': “recording”/“stopped”/”paused”/”pending”/”recording”
'is_active': True/False
'packets_count': integer how many data packets recorded (not samples, packets. One packet typically has 10-1000 samples)
'created_at': datetime (string)
'updated_at': datetime (string)

6.3.2.2. recorder_handler.list_recorders(string: username)

returns list of currently existing recorders for given username

6.3.2.3. recorder_handler.delete_recorder(string: recorder_id)

6.3.2.4. recorder_handler.update_recorder(string: recorder_id, string: status)

status may be:

"paused" – to pause recording
“stopped” – to stop recording (can not be restarted)
“pending” – to unpause paused recorder

can not force “recording” as it requires cloud resource allocation. Recorder will automatically transition from “pending” to “recording”. Poll status to verify “recording” status.

6.3.2.5. recorder = recorder_handler.get_recorder(string: recorder_id)

returns recorder information object; useful to check recorder status:
status = recorder["status"] – returns string “stopped”/”paused”/”pending”/”recording” (can not force “recording” as it requires cloud resource allocation. Recorder will automatically transition from “pending” to “recording”. Poll status to verify “recording” status.

7. Data exporter

7.1. General information

Exporter object is used to convert data stored in a Recorder into downloadable, easily-readable CSV file.
Exporter object is typically created after Recorder is stopped, to download the data.

Typical flow: create exporter object for corresponding Recorder ID; wait for exporting to complete (typically by polling exporter status), download the data from URL (either programmatically or manually via dashboard).

7.1. macros

the API library contains pre-built macros for high-level operation of the recorder/exporter interface. It is strongly recommended to use the macros. The macros take care of creating the recorder, polling for the correct status, handling edge cases such as a recorder being previously left “open” by mistake, HTTP timeouts, retries, et cetera.

To access the macros use the following import in your Python script:

from neurospeed import macros

7.1.1. exporter macros:

7.1.1.1. download_data(customer_auth, recorder_id, exporter_config, hia_config, save_folder)

This function initiates data conversion to human-readable format; and downloads the data to the local machine.

This function blocks until the download is complete. Downloaded files are routinely several hundred megabytes in size.

Inputs:

- Customer_auth object – generated by the neurospeed auithenticator module
- Recorder_id – integer number, uniquely identifies the recorder inside the userspace. Generated by start_recording() function. Must correspond to existing recorder ID on the cloud.
- Exporter config: python dictionary with required exported parameters. Useful parameters include:

```
exporter_config = {
        'stream_ids': [], type list. Leave blank if unsure. List of sensor streams by HIA clients that feed raw data
        to the cloud.
        'is_all_columns': True/False,
        ‘is_all_streams': True/False,
        'selected_columns': [], type list. hint: look at recorder’s “columns” fields to see what data is available)
        "custom_name": string type, defines filename of created file,
        "start_timestamp_mode": "start",
```

```
        "end_timestamp_mode": "end",
    }
```

"custom_name", "start_timestamp_mode", "end_timestamp_mode" are mandatory fields.

- Hia_config – dictionary that contains the user-level credentials. Used to operate the recorder/exporter for the correct user.
- Save_folder – absolute (full) path to folder where the downloaded data is to be stored.
  Example to store the data into current working directory: save_folder = os.getcwd()
  Example to store the data into an arbitrary directory: save_folder = "C:\\neurobrave\\data"

Outputs:

- exported_filename - Absolute path to the downloaded data file in .csv format

7.2. Downloading exported data files

Note: download URL is only valid shortly after exporting is done, and deprecated fast for security reasons. Downloading manually via Dashboard is available indefinitely, until the Exporter is deleted.

Downloading file from URL is possible in python with:

```
import urllib.request
urllib.request.urlretrieve(url, "filename.csv")
```

7.3. exporter control

7.3.1. UserRoom_Recorder_Exporter_Handler(customer_auth : customer_auth)

Input: customer authentication object (created by neurospeed.auth.auth_as_customer_handler.Auth_AS_Customer_handler())
Returns exporter handler object

7.3.2. Available methods for exporter handler object:

7.3.2.1. create_exporter(string: recorder_id, dict: config):

input: recorder ID,
Exporter config: dictionary with required exported parameters. Useful parameters include:

```
exporter_config = {
    'stream_ids': [], type list. Leave blank if unsure. List of sensor streams by HIA clients that feed raw data to the cloud.
    'is_all_columns': True/False,
```

```
        'is_all_streams': True/False,
        'selected_columns': [], type list. hint: look at recorder's "columns" fields to see what data is available)
        "custom_name": string type, defines filename of created file,
        "start_timestamp_mode": "start",
        "end_timestamp_mode": "end",
    }
```

returns dictionary containing information about the exporter:

'id': integer ID
'exporter_name': string exporter name specified when created
'file_size': size in cloud memory/storage
'export_config': dictionary that was used when creating the exporter
'status': 'exporting'/'exported',
'created_at' datetime created (string)
'updated_at':datetime updated (string
'recorder': dictionary with corresponding recorder's metadata

exporter["status"] == "exported" – data conversion is complete, file available for download.
exporter["status"] != "exported"- cloud engine is still converting the recorder data to CSV file.

7.3.2.2. get_exported_url(string: exporter_id)

returns URL via which the exported data file can be downloaded

7.3.2.3. list_exporters(string: username)

returns list of currently existing exporters for given username

7.3.2.4. get_exporter(string: exporter_id)

returns information dictionary for input "exporter_id" (type string)

7.3.2.5. delete_exporter(string: exporter_id)

deletes the exporter. Data file contained in the exported is deleted as well. Data can not be restored after performing this operation.

7.3.3. exporter operation example

```
import urllib.request
exporter_config =  {
    "custom_name": "my_datafile.csv",
```

```
    "start_timestamp_mode": "start",
    "end_timestamp_mode": "end",
    }
my_exporter_handler = UserRoom_Recorder_Exporter_Handler(customer_auth)
exporter = My_exporter_handler.create_exporter(my_recorder_id, exporter_config)
my_exporter_id = str(exporter["id"])

while not my_exporter_handler .get_exporter(my_exporter_id)[“status”] == “exported”
        print(“data not ready”)
        time.sleep(5)
urllib.request.urlretrieve(my_exporter_handler.get_exporter_url(my_exporter_id), "filename.csv")
my_exporter_handler.delete_exporter(my_exporter_id)
```