trendminer-interface

Wouter Daniels

Aug 29, 2025

CONTENTS

1 Getting Started **1**
- 1.1 Authentication 1
- 1.2 Event Analytics 3
- 1.3 Get Tag Data 10

CHAPTER

ONE

GETTING STARTED

1.1 Authentication

1.1.1 Import

There is only one thing we really need, and that is the TrendMinerClient class. Everything we can do with `trendminer_interface` is implemented on that class.

[1]:
```
from trendminer_interface import TrendMinerClient
```

1.1.2 Authenticating

As a first step, we will always need to create a TrendMinerClient instance which is authenticated to send and receive requests to our appliance. The TrendMinerClient instance will always attempt to refresh authentication once expired, though through inactivity or prolonged use (couple of hours), the authentication will eventually expire and you will have to log in again. The TrendMinerClient instance does not keep any sensitive information like passwords in memory.

To gain access via API (as we are doing in `trendminer_interface`) we will always need a **client ID** and **client Secret**, which need to be generated in ConfigHub by an admin.

We typically want to set a **timezone** for our client. All returned timestamps are then given in this particular timezone. The timezone defaults to UTC.

To make the example clear, I am simply inputting (fake) passwords as strings. However, **you should never put your examples in plain text**. In this blog post, under 'storing securely', you can find some better practices. In further code examples, I will use the keyring package to retrieve passwords securely from my keychain. Note that to use this package in your own code, you will first have to install this package separately and then make sure your passwords are saved.

Client user

A client user is only accessible via APIs, so it is not possible to log into TrendMiner via the UI as a client user. A client user is intended only to retrieve data. It is not recommended to use a client user to save work organizer objects, as (i) they will never be accessible via the UI, and (ii) unexpected behaviour might occur as this behaviour is unintended and not thoroughly tested.

Just like a regular user, a client user needs to be given access to specific datasources in ConfigHub in order to access them.

Simply a **client ID** a **client secret** suffice to authenticate to the server as a client.

[2]:
```
url = 'https://cs.trendminer.net/'
client_id = 'wdanielsclient'
client_secret = '282fYzZjTGjxWzJ8viwHswk6Fu9ovop5z'
```

[4]:
```
client = TrendMinerClient(
    url=url,
    client_id=client_id,
    client_secret=client_secret,
    tz='Europe/Brussels',
)
```

```
/Users/wouter.daniels/PycharmProjects/tm-python-interface/trendminer_interface/client.py:
↪183: VersionMismatchWarning: This SDK version was tested for use with TrendMiner␣
↪versions [2024.R1.0 | 2024.R1.1 | 2024.R2.0 | 2024.R2.1] while your TrendMiner version␣
↪is [2024.R3.1-04]. Some functionality might not work as expected.
  warnings.warn(
```

User client

We can authenticate as a user by providing a name and password in addition to client ID and secret. Authenticating as a user is required if we want access or create that user's saved work organizer items.

We are still required to provide a valide client id and secret to authenticate as a user. It will be the user's access rights that determine the datasources we have access to, regardless of the access rights assigned to the provided client id.

Note that **authenticating as a user only works for local users**. Providing your LDAP or SAML user details does not work (remember, TrendMiner does not and should not know these user credentials). This means that for some use cases, a dedicated local user will need to be created.

[5]:
```
username ='wdaniels'
password = 'MyPassw0rd!'
```

[7]:
```
client = TrendMinerClient(
    url=url,
    client_id=client_id,
    client_secret=client_secret,
    username=username,
    password=password,
    tz='Europe/Brussels',
)
```

1.1.3 Doing things

Once we have our authenticate TrendMinerClient instance, we can start interacting with the appliance. For example, let's search for some tags. For a full overview of what you can do with `trendminer_interface`, please refer to the rest of the documentation.

[10]:
```
client.tag.search(name='TM4-BP2*.1')
```

[10]:
```
[<< Tag | TM4-BP2-LEVEL.1 >>,
 << Tag | TM4-BP2-PRODUCT.1 >>,
 << Tag | TM4-BP2-TEMP.1 >>,
 << Tag | TM4-BP2-QUAL.1 >>,
 << Tag | TM4-BP2-CW.1 >>,
 << Tag | TM4-BP2-PRESSURE.1 >>,
 << Tag | TM4-BP2-CONC.1 >>,
 << Tag | TM4-BP2-UTIL.1 >>,
 << Tag | TM4-BP2-SPEED.1 >>]
```

1.1.4 Troubleshooting and customization

- Adding `verify=False` to TrendMinerClient disables the checking the SSL certificates on requests, **avoiding SSL verification errors** that occur when the certificates of your TrendMiner appliance are expired or self-signed.
- When you get an error claiming the **account is not fully set up**, that either means the password has not yet been changed by the user after it has been (re)set by the admin (do this by logging into TrendMiner manually), or that you are trying to log in as and LDAP/SAML user (which is not possible).
- You can assign the proxies parameter in TrendMinerClient.
- You can assign your own timeout parameter in TrendMinerClient.

1.2 Event Analytics

Performing analytics on events is a core feature TrendMiner, which makes interacting with events a core use case of the `trendminer_interface` package.

1.2.1 Authentication

As always, as a first step we authenticate to the TrendMiner appliance.

[1]:
```
import keyring
from trendminer_interface import TrendMinerClient

url = 'https://cs.trendminer.net/'
client_id = 'wdanielsclient'

client = TrendMinerClient(
    url = url,
    client_id = client_id,
    client_secret=keyring.get_password(url, client_id),
    tz = "Europe/Brussels"  # all returned timestamps will be in this timezone
)
```

```
/Users/wouter.daniels/PycharmProjects/tm-python-interface/trendminer_interface/client.py:
↪191: VersionMismatchWarning: This SDK version was tested for use with TrendMiner␣
↪versions [2024.R1.0 | 2024.R1.1 | 2024.R2.0 | 2024.R2.1] while your TrendMiner version␣
↪is [2025.R1.0-04]. Some functionality might not work as expected.
  warnings.warn(
```

1.2.2 Selecting tags

We will need some demo tags to illustrate event analytics. We can load these directly from their names.

[2]:
```
flow = client.tag.from_name("TM4-HEX-FI0620")
temp = client.tag.from_name("TM4-HEX-TI0620")
```

1.2.3 Selecting intervals

This package uses pandas Interval and IntervalIndex as central objects for denoting events and as method inputs.

Importing from `trendminer_interface` adds custom accessors to the DataFrame class, which contain TrendMiner-specific methods as well as some utility methods.

A common starting point is therefore to generate an IntervalIndex with a regular frequency, and use that to create a new DataFrame. Here, we select all full weeks (starting Monday) in April 2023.

[3]:
```
import pandas as pd

week_index = pd.interval_range(
    start=pd.Timestamp("2023-04-01 00:00:00", tz=client.tz),
    end=pd.Timestamp("2023-05-01 00:00:00", tz=client.tz),
    freq="W-MON",
    closed="left",
)

df = pd.DataFrame(index=week_index)
```

1.2.4 Calculations on intervals

We can now perform calculations usingt the custom `interval` accessor on the DataFrame. Accessor methods will return the manipulated DataFrame.

[4]:
```
df.interval.calculate(tag=temp, operation="max", name="max temp")
```

[4]:
```
                                                  max temp
[2023-04-03 00:00:00+02:00, 2023-04-10 00:00:00...  52.869370
[2023-04-10 00:00:00+02:00, 2023-04-17 00:00:00...  51.089703
[2023-04-17 00:00:00+02:00, 2023-04-24 00:00:00...  50.764526
[2023-04-24 00:00:00+02:00, 2023-05-01 00:00:00...  50.890480
```

To add multiple calculations, we can pipe DataFrame operations. We can of course include regular Python operations on the DataFrame. We can integrate the weekly flow to get the total volume, but because the TrendMiner integral assumes a time unit of 'day', and our flow is expressed in m3 per hour, we need to multiply by 24 to get the volume in m3. Then as an example, let's set some bound on the flow derived from the mean and standard deviation, keeping into account that the lower limit should never be lower than 0.

[5]:
```
df = (
    pd.DataFrame(index=week_index)
    .interval.calculate(tag=flow, operation="int", name="total")
    .assign(volume=lambda df: df["total"]*24)
    .interval.calculate(tag=flow, operation="mean", name="mean")
    .interval.calculate(tag=flow, operation="std", name="std")
    .assign(upper_limit=lambda df: df["mean"]+1.5*df["std"])
    .assign(lower_limit=lambda df: (df["mean"]-1.5*df["std"]).clip(lower=0))
    .round(1)
)

df
```

[5]:
```
                                                  total  volume  mean   std  \
[2023-04-03 00:00:00+02:00, 2023-04-10 00:00:00...  171.7  4121.8  24.5  17.5
[2023-04-10 00:00:00+02:00, 2023-04-17 00:00:00...  338.3  8118.0  48.3  26.9
[2023-04-17 00:00:00+02:00, 2023-04-24 00:00:00...  360.9  8661.6  51.6  29.0
[2023-04-24 00:00:00+02:00, 2023-05-01 00:00:00...  300.9  7222.0  43.0  21.7

                                                  upper_limit  lower_limit
[2023-04-03 00:00:00+02:00, 2023-04-10 00:00:00...         50.8          0.0
[2023-04-10 00:00:00+02:00, 2023-04-17 00:00:00...         88.7          8.0
[2023-04-17 00:00:00+02:00, 2023-04-24 00:00:00...         95.0          8.1
[2023-04-24 00:00:00+02:00, 2023-05-01 00:00:00...         75.5         10.5
```

1.2.5 Interval manipulations

Various methods are available on the `interval` accessor to manipulate the timeframe given by events. Note that these manipulations only affect the IntervalIndex, and that DataFrame value columns are retained. For a full overview of methods, call `help(df.interval)`.

[6]:
```
df.interval.shrink(left="1h", right="2h")
```

[6]:
```
                                                  total  volume  mean   std  \
[2023-04-03 01:00:00+02:00, 2023-04-09 22:00:00...  171.7  4121.8  24.5  17.5
[2023-04-10 01:00:00+02:00, 2023-04-16 22:00:00...  338.3  8118.0  48.3  26.9
[2023-04-17 01:00:00+02:00, 2023-04-23 22:00:00...  360.9  8661.6  51.6  29.0
[2023-04-24 01:00:00+02:00, 2023-04-30 22:00:00...  300.9  7222.0  43.0  21.7

                                                  upper_limit  lower_limit
[2023-04-03 01:00:00+02:00, 2023-04-09 22:00:00...         50.8          0.0
[2023-04-10 01:00:00+02:00, 2023-04-16 22:00:00...         88.7          8.0
[2023-04-17 01:00:00+02:00, 2023-04-23 22:00:00...         95.0          8.1
[2023-04-24 01:00:00+02:00, 2023-04-30 22:00:00...         75.5         10.5
```

1.2.6 Searches

Rather than selecting intervals manually, we can also retrieve them from a search. As an example, let's perform a value-based search on our flow being higher than 0. First we need to instantiate the search:

[7]:
```
vbs = client.search.value(
    name="high flow",  # search name -> will become DataFrame index name
    queries=[
        (flow, ">", 60),  # our search query as a tuple (tag, operator, value)
    ],
    operator="AND",  # how to combine multiple queries (AND/OR)
    duration=pd.Timedelta(hours=24),  # minimal result duration
)

vbs
```

[7]:
```
<< ValueBasedSearch | high flow >>
```

Our search needs to be performed on a specific interval to yield results as a dataframe. We use the pandas Interval class for this. We always need to assign a timezones to the timestamps used to generate the interval.

[8]:
```
search_interval = pd.Interval(
    left=pd.Timestamp("2023-01-01", tz=client.tz),
    right=pd.Timestamp("2024-01-01", tz=client.tz),
    closed="both"
)

search_interval
```

[8]:
```
Interval(2023-01-01 00:00:00+01:00, 2024-01-01 00:00:00+01:00, closed='both')
```

Performing the search yields an empty DataFrame with the search result intervals as the IntervalIndex

[9]:
```
vbs.get_results(search_interval)
```

[9]:
```
Empty DataFrame
Columns: []
Index: [[2023-04-16 06:00:00+02:00, 2023-04-18 21:20:00+02:00], [2023-04-22 13:32:00+02:
↪00, 2023-04-23 15:55:00+02:00], [2023-04-26 19:25:00+02:00, 2023-04-28 04:02:00+02:00],
↪ [2023-05-06 16:40:00+02:00, 2023-05-08 21:01:00+02:00], [2023-05-08 21:44:00+02:00,␣
↪2023-05-10 04:40:00+02:00], [2023-05-13 11:29:00+02:00, 2023-05-15 04:38:00+02:00],␣
↪[2023-05-20 15:43:00+02:00, 2023-05-21 20:02:00+02:00], [2023-05-27 19:39:00+02:00,␣
↪2023-06-01 08:26:00+02:00]]
```

To add calculations to the DataFrame, we can perform interval calculations as demonstrated in the previous section. Another option for searches is to include calculations already in the search definition:

[10]:
```
vbs = client.search.value(
    name="high flow",
    queries=[
        (flow, ">", 60)
    ],
    operator="AND",
    duration=pd.Timedelta(hours=24),
    calculations={
        "max temp": (temp, "max"),
    },
)

df = vbs.get_results(search_interval)

df.head(3)
```

[10]:
```
                                                    max temp
high flow
[2023-04-16 06:00:00+02:00, 2023-04-18 21:20:00...  50.742350
[2023-04-22 13:32:00+02:00, 2023-04-23 15:55:00...  50.764526
[2023-04-26 19:25:00+02:00, 2023-04-28 04:02:00...  50.890480
```

1.2.7 Writing intervals to a tag

We can create tags from intervals using the csv import endpoint. First we need to instantiate a new tag object.

[11]:
```
tag = client.tag(
    name=f"Example discrete calc tag",
    description="max batch temperature",
    tag_type="DISCRETE",
)
```

We also need a pandas Series with the tag data. This is easily generated from the DataFrame of calculations.

[12]:
```
ser = pd.Series(index=df.index.left, data=df["max temp"], name=tag.name)
ser.head()
```

[12]:
```
2023-04-16 06:00:00+02:00    50.742350
2023-04-22 13:32:00+02:00    50.764526
2023-04-26 19:25:00+02:00    50.890480
2023-05-06 16:40:00+02:00    50.206960
```

(continues on next page)

(continued from previous page)

```
2023-05-08 21:44:00+02:00    51.009150
Name: Example discrete calc tag, dtype: float64
```

Then we can save/update this csv tag on the server

[13]:
```
tag_data_dict = {tag: ser}
client.io.tag.save(tag_data_dict)
```

1.2.8 Context items

Context items are basically intervals of a specific type, attached to a specific component. Let's take our search results from the previous section and turn them into context items.

Context items always have a specific `type`.

[14]:
```
context_type = client.context.type.from_key("ANOMALY")
df["type"] = context_type
```

Context items must be attached to a `component`, which is either a tag, asset or attribute.

[15]:
```
df["component"] = flow
```

Optionally, we can add a `description` to our context items as well.

[16]:
```
df["description"] = "high flow"
```

We can now use the `context` accessor on the DataFrame. Let's use it to save our intervals as context items. Additional text or numeric columns (e.g., the `max temp` column) will be added as fields to the context items. If the column name is not mapped to a dedicated context field key on the context type, it will show up as 'other property' in TrendMiner.

[17]:
```
df.context.save()
```

Saving the context items does not generate any output or updates the DataFrame, but we can retrieve our newly created context items through a ContextHub view. We see that saving the intervals as context items adds a lot of context item metadata to the intervals.

For an overview of all possible filters on context items searches, call `help(client.context.filter)`.

[18]:
```
chv = client.context.view(
    name="high flow",
    filters=[
        client.context.filter.components([flow]),
        client.context.filter.context_types([context_type]),
        client.context.filter.users([client.username]),
    ]
)

chv.get_items().head(3)
```

[18]:
```
                                                   key  \
high flow
[2023-04-16 06:00:00+02:00, 2023-04-18 21:20:00...  FB9-05
[2023-04-22 13:32:00+02:00, 2023-04-23 15:55:00...  FB9-06
[2023-04-26 19:25:00+02:00, 2023-04-28 04:02:00...  FB9-07
```

(continues on next page)

(continued from previous page)

```
                                                                                identifier␣
↪ \
high flow
[2023-04-16 06:00:00+02:00, 2023-04-18 21:20:00...  b9b9b3d0-5b8e-408b-beac-1f2e376cc855
[2023-04-22 13:32:00+02:00, 2023-04-23 15:55:00...  14740c70-23ce-41fd-97b1-12851041ce18
[2023-04-26 19:25:00+02:00, 2023-04-28 04:02:00...  c6d0fa2e-065a-43ea-9410-3a24cebaac20

                                                   identifier_external  \
high flow
[2023-04-16 06:00:00+02:00, 2023-04-18 21:20:00...                None
[2023-04-22 13:32:00+02:00, 2023-04-23 15:55:00...                None
[2023-04-26 19:25:00+02:00, 2023-04-28 04:02:00...                None

                                                   description     type  \
high flow
[2023-04-16 06:00:00+02:00, 2023-04-18 21:20:00...   high flow  ANOMALY
[2023-04-22 13:32:00+02:00, 2023-04-23 15:55:00...   high flow  ANOMALY
[2023-04-26 19:25:00+02:00, 2023-04-28 04:02:00...   high flow  ANOMALY

                                                        component  \
high flow
[2023-04-16 06:00:00+02:00, 2023-04-18 21:20:00...  TM4-HEX-FI0620
[2023-04-22 13:32:00+02:00, 2023-04-23 15:55:00...  TM4-HEX-FI0620
[2023-04-26 19:25:00+02:00, 2023-04-28 04:02:00...  TM4-HEX-FI0620

                                                                          created_by  \
high flow
[2023-04-16 06:00:00+02:00, 2023-04-18 21:20:00...  service-account-wdanielsclient
[2023-04-22 13:32:00+02:00, 2023-04-23 15:55:00...  service-account-wdanielsclient
[2023-04-26 19:25:00+02:00, 2023-04-28 04:02:00...  service-account-wdanielsclient

                                                                             created  \
high flow
[2023-04-16 06:00:00+02:00, 2023-04-18 21:20:00... 2025-05-06 12:59:30.570522+02:00
[2023-04-22 13:32:00+02:00, 2023-04-23 15:55:00... 2025-05-06 12:59:30.570658+02:00
[2023-04-26 19:25:00+02:00, 2023-04-28 04:02:00... 2025-05-06 12:59:30.570716+02:00

                                                                        last_modified  \
high flow
[2023-04-16 06:00:00+02:00, 2023-04-18 21:20:00... 2025-05-06 12:59:30.570522+02:00
[2023-04-22 13:32:00+02:00, 2023-04-23 15:55:00... 2025-05-06 12:59:30.570658+02:00
[2023-04-26 19:25:00+02:00, 2023-04-28 04:02:00... 2025-05-06 12:59:30.570716+02:00

                                                     max temp MaterialName  \
high flow
[2023-04-16 06:00:00+02:00, 2023-04-18 21:20:00...  50.742350         <NA>
[2023-04-22 13:32:00+02:00, 2023-04-23 15:55:00...  50.764526         <NA>
[2023-04-26 19:25:00+02:00, 2023-04-28 04:02:00...  50.890480         <NA>

                                                   Suggestion2_psd  \
high flow
```

(continues on next page)

(continued from previous page)

```
[2023-04-16 06:00:00+02:00, 2023-04-18 21:20:00...            <NA>
[2023-04-22 13:32:00+02:00, 2023-04-23 15:55:00...            <NA>
[2023-04-26 19:25:00+02:00, 2023-04-28 04:02:00...            <NA>

                                                    suggestion1_psd
high flow
[2023-04-16 06:00:00+02:00, 2023-04-18 21:20:00...            <NA>
[2023-04-22 13:32:00+02:00, 2023-04-23 15:55:00...            <NA>
[2023-04-26 19:25:00+02:00, 2023-04-28 04:02:00...            <NA>
```

It is possible to update context items we have retrieved via a ContextHub view. We can simply manipulate the field columns and send an update request. A common use case is to enrich context items with additional calculations in a script. These context items could be created automatically from a monitor in TrendMiner.

A standard workflow here would be to retrieve context items where a specific field is empty, fill in this (and potentially other) fields in the DataFrame, and then update the items with the new data in TrendMiner.

Including a filter to only retrieve (and thus updated) items which have not yet been updated is essential for good performance. It also means our script can run at any time to update all new items.

[19]:
```
material_key = "MaterialName"
material_field = client.context.field.from_key(material_key)

# Retrieve items with emtpy material field
chv = client.context.view(
    name="high flow",
    filters=[
        client.context.filter.components([flow]),
        client.context.filter.context_types([context_type]),
        client.context.filter.users([client.username]),
        client.context.filter.field(material_field, mode="EMPTY"),
    ]
)
df = chv.get_items()

# Calculate the maximal flow
df = df.interval.calculate(flow, "max", "max flow")

# Determine if it was product A or B, depending on wether the max flow was above 100
# We use the `pop` method to remove fields we do not want to add to the context items.
df["is A"] = df.pop("max flow") > 100
df[material_key] = df.pop("is A").map({True: "A", False: "B"})

# Update the context items on the server
df.context.update()
```

Finally, it is also possible to delete context items directly from a ContextHub view. Including a filter on the user is recommended so we do not accidentally delete other people's context items.

[20]:
```
chv = client.context.view(
    name="high flow",
    filters=[
        client.context.filter.components([flow]),
```

(continues on next page)

(continued from previous page)

```
        client.context.filter.context_types([context_type]),
        client.context.filter.users([client.username]),
    ]
)

chv.delete_items()
```

1.3 Get Tag Data

1.3.1 Create Client

[1]:
```
import keyring
from trendminer_interface import TrendMinerClient

url = 'https://cs.trendminer.net/'
client_id = 'wdanielsclient'

client = TrendMinerClient(
    url = url,
    client_id = client_id,
    client_secret=keyring.get_password(url, client_id),
    tz = "Europe/Brussels"
)
```

```
/Users/wouter.daniels/PycharmProjects/tm-python-interface/trendminer_interface/client.py:
↪183: VersionMismatchWarning: This SDK version was tested for use with TrendMiner␣
↪versions [2024.R1.0 | 2024.R1.1 | 2024.R2.0 | 2024.R2.1] while your TrendMiner version␣
↪is [2024.R3.1-04]. Some functionality might not work as expected.
  warnings.warn(
```

1.3.2 Extract data from single tag

If we are planning to retrieve the latest data from tags, it is best to send index requests to make sure the data is up to date. This action happens automatically when using the TrendMiner UI.

[2]:
```
level = client.tag.from_name("[CS]BA:LEVEL.1")
conc = client.tag.from_name("[CS]BA:CONC.1")
temp = client.tag.from_name("[CS]BA:TEMP.1")
phase = client.tag.from_name("[CS]BA:PHASE.1")

level.index()
conc.index()
temp.index()
phase.index()
```

Example 1 - Interpolated data for fixed interval

Interpolated data is the default. The main advantage is that we get equidistant timestamps, which will be the same for every tag, as long as we use the same interval.

[3]:
```
import pandas as pd

freq = pd.Timedelta(hours=1)

interval = pd.Interval(
    left=pd.Timestamp("2023-01-01", tz=client.tz),
    right=pd.Timestamp("2023-01-03", tz=client.tz),
    closed="both",
)

ser = level.get_data(interval, freq=freq)
ser.head()
```

[3]:
```
ts
2023-01-01 00:00:00+01:00    28.735641
2023-01-01 01:00:00+01:00    10.856165
2023-01-01 02:00:00+01:00    14.464235
2023-01-01 03:00:00+01:00    40.634544
2023-01-01 04:00:00+01:00    24.326510
Name: [CS]BA:LEVEL.1, dtype: float64
```

Example 2 - Index data for the last 8h

Index data contains the min, max, start and end datapoints for every index block. The size of an index block is dependent on the index resolution. We can retrieve the client resolution and see that it is 1 minute. For our server, there would be up to 4 points stored per minute. Note that this number can always be less than 4 when a point is both the start/end and min/max value, or when the block contains less than 4 raw datapoints in the datasource.

[4]:
```
client.resolution
```

[4]:
```
Timedelta('0 days 00:01:00')
```

As a first step, we need to create a pandas Interval representing the latest 8h. It always makes sense to round the end timestamp of 'live' intervals down to at least the client resolution to get nice, round timestamps.

We can then use the interval to retrieve the latest data. Note that the retrieved data usually has irregular timestamps, since it contains the real datapoints logged in the datasource.

[5]:
```
length = pd.Timedelta(hours=8)
freq = pd.Timedelta(minutes=1)
now = pd.Timestamp.now(tz=client.tz).floor(client.resolution)

interval = pd.Interval(
    left=now-length,
    right=now,
    closed="both",
)

ser = conc.get_index_data(interval=interval)
ser.head()
```

[5]:
```
ts
2025-03-26 02:50:23+01:00    0.000000
2025-03-26 03:00:53+01:00    2.026474
2025-03-26 03:01:23+01:00    1.022871
```

(continues on next page)

(continued from previous page)

```
2025-03-26 03:02:23+01:00    3.090765
2025-03-26 03:02:53+01:00    2.619209
Name: [CS]BA:CONC.1, dtype: float64
```

Example 3 - Chart data for a fixed interval

Tag chart data is a subset of the indexed data that tries to visually represent the original data as closely as possible. Like index data, the min, max, start and end datapoints are returned per block, only now we need to pass the number of blocks as a parameter.

[6]:
```
ser = temp.get_chart_data(interval, periods=10)
ser.head()
```

[6]:
```
ts
2025-03-26 02:50:00+01:00     0.000000
2025-03-26 03:37:53+01:00    20.610619
2025-03-26 03:38:53+01:00    24.054213
2025-03-26 03:45:53+01:00    45.087185
2025-03-26 04:25:53+01:00     9.661495
Name: [CS]BA:TEMP.1, dtype: float64
```

1.3.3 Extract data from multiple tags

An easy way to get data from multiple tags into a single DataFrame directly is to first instantiate a TrendHub view instance. Note that this does not mean this view is saved to the appliance. Note that we can only get interpolated tag data from TrendHub views.

Since a view can have multiple layers, a list of DataFrame is returned, one for each layer.

[13]:
```
thv = client.trend.view(
    name="last 8h",
    entries=[level, conc, temp, phase],
    layers=[interval],
    context_interval=interval,
    live=False,
)

df_list = thv.get_data(freq=pd.Timedelta(minutes=1))
df_list[0].head()
```

[13]:
```
                           [CS]BA:LEVEL.1  [CS]BA:CONC.1  [CS]BA:TEMP.1  \
ts
2025-03-26 02:50:00+01:00        2.802551       34.16749            0.0
2025-03-26 02:51:00+01:00        0.000000        0.00000            0.0
2025-03-26 02:52:00+01:00        0.000000        0.00000            0.0
2025-03-26 02:53:00+01:00        0.000000        0.00000            0.0
2025-03-26 02:54:00+01:00        0.000000        0.00000            0.0

                           [CS]BA:PHASE.1
ts
2025-03-26 02:50:00+01:00         Phase1
2025-03-26 02:51:00+01:00         Phase1
2025-03-26 02:52:00+01:00         Phase1
```

(continues on next page)

(continued from previous page)

```
2025-03-26 02:53:00+01:00         Phase1
2025-03-26 02:54:00+01:00         Phase1
```