cognite-extractor-utils Documentation

Release 1.2.1

Cognite

Oct 20, 2020

Contents

The extractor-utils package is an extension of the Cognite Python SDK intended to simplify the development of data extractors for Cognite Data Fusion.

CHAPTER 1

Installation

To install this package:

```
pip install cognite-extractor-utils
```

If the Cognite SDK is not already installed, the installation will automatically fetch and install it as well.

To get going, consult the *Quickstart*

CHAPTER 2

Contents

2.1 Quickstart

2.1.1 Installation

To install this package:

```
pip install cognite-extractor-utils
```

If the Cognite SDK is not already installed, the installation will automatically fetch and install it as well.

2.1.2 `cognite.extractorutils.uploader`

API requests can be expensive to perform, even with a well-performing backend. The solution is to batch toghether more data into a single API request.

The uploader module contains upload queues. These will hold data until a configured condition is met, triggering an upload. For example, instead of

```
client = CogniteClient()

while not stop:
    timestamp, value = source.query()

    client.datapoints.insert((timestamp, value), external_id="my-timeseries")
```

which would do very many API requests to CDF, you could do

```
client = CogniteClient()
upload_queue = TimeSeriesUploadQueue(cdf_client=client, max_upload_interval=1)

upload_queue.start()
```

(continues on next page)

(continued from previous page)

```
while not stop:
    timestamp, value = source.query()

    upload_queue.add_to_upload_queue((timestamp, value), external_id="my-timeseries")

upload_queue.stop()
```

The `max_upload_interval` specifies the maximum time (in seconds) between each API call. The upload method will be called on `stop()` as well so no datapoints are lost. You can also use the queue as a context:

```
client = CogniteClient()

with TimeSeriesUploadQueue(cdf_client=client, max_upload_interval=1) as upload_queue:
    while not stop:
        timestamp, value = source.query()

        upload_queue.add_to_upload_queue((timestamp, value), external_id="my-
↪timeseries")
```

This will call the `start()` and `stop()` methods automatically.

You can also trigger uploads after a given amount of data is added, by using the `max_queue_size` keyword argument instead. If both are used, the condition being met first will trigger the upload.

Similar queues exists for RAW rows and Events too.

2.1.3 `cognite.extractorutils.statestore`

A state store object is a dictionary from an external ID to a low and high watermark, used to keep track of the progress of a front- or backfill between runs.

A state is a tuple, typically containing two timestamps (range of extracted data).

You can choose the back-end for your state store with which class you're instatiating:

```
# A state store using a JSON file as remote storage:
states = LocalStateStore("state.json")
states.initialize()

# A state store using a RAW table as remote storage:
states = RawStateStore(
    cdf_client = CogniteClient(),
    database = "extractor_states",
    table = "my_extractor_deployment"
)
states.initialize()
```

The `initialize()` method loads all the states from the configured remote store.

You can now use this state store to get states:

```
low, high = states.get_state(external_id = "my-id")
```

You can set states:

```
states.set_state(external_id = "another-id", high=100)
```

and similar for `low`. The `set_state(...)` method will always overwrite the current state. Some times you might want to only set state *if larger* than the previous state, in that case consider `expand_state(...)`:

```
# High watermark of another-id is already 100, nothing happens in this call:
states.expand_state(external_id = "another-id", high=50)

# This will set high to 150 as it is larger than the previous state
states.expand_state(external_id = "another-id", high=150)
```

To store the state to the remote store, use the `synchronize()` method:

```
states.synchronize()
```

2.1.3.1 Integrating with upload queues

You can set a state store to automatically update on upload triggers from an upload queue by using the `post_upload_function` in the upload queue:

```
states = LocalStateStore("state.json")
states.initialize()

uploader = TimeSeriesUploadQueue(
    cdf_client = CogniteClient(),
    max_upload_interval = 10
    post_upload_function = states.post_upload_handler()
)

# The state store is now updated automatically!

states.synchronize()
```

2.1.4 `cognite.extractorutils.metrics`

The metrics module contains a general, pre-built metrics collection, as well as tools to routinely push metrics to a remote server.

The `BaseMetrics` class forms the basis for a metrics collection for an extractor, containing some general metrics that all extractors should report (such as e.g. CPU and memory usage of the extractor). To create your own set of metrics, subclass this class and populate it with extractor-specific metrics, as such:

```
class MyMetrics(BaseMetrics):
    def __init__(self):
        super().__init__(extractor_name="my_extractor", extractor_version=__version__)

        self.a_counter = Counter("my_extractor_example_counter", "An example counter")
        ...
```

The metrics module also contains some Pusher classes that are used to routinely send metrics to a remote server, these can be automatically created with the `start_pushers` method described in `configtools`.

2.1.5 cognite.extractorutils.configtools

The configtools contains base classes for configuration, and a YAML loader to automatically serialize these dataclasses from a config file.

Configs are described as `dataclass``es, and use the ``BaseConfig` class as a superclass to get a few things built-in: config version, Cognite project and logging. Use type hints to specify types, use the `Optional` type to specify that a config parameter is optional, and give the attribute a value to give it a default.

For example, a config class for an extractor may look like the following:

```
class ExtractorConfig:
    parallelism: int = 10

    state_store: Optional[StateStoreConfig]
    ...

@dataclass
class SourceConfig:
    host: str
    username: str
    password: str
    ...


@dataclass
class MyConfig(BaseConfig):
    extractor: ExtractorConfig
    source: SourceConfig
```

You can then load a YAML file into this dataclass with the *load_yaml* function:

```
with open("config.yaml") as infile:
    config = load_yaml(infile, MyConfig)
```

2.1.5.1 The config object

The config object can additionally do several things, such as:

Creating a `CogniteClient` based on the config:

```
client = config.cognite.get_cognite_client("my-client")
```

Setup the logging according to the config:

```
config.logger.setup_logging()
```

Start and stop threads to automatically push all the prometheus metrics in the default prometheus registry to the configured push-gateways:

```
config.metrics.start_pushers(client)

# Extractor code

config.metrics.stop_pushers()
```

Get a state store object as configured:

```
states = config.extractor.state_store.create_state_store()
```

2.2 Package reference

2.2.1 `configtools` - Utilities for reading, parsing and validating config files

The `configtools` module exists of tools for loading and verifying config files for extractors.

Extractor configurasions are conventionally written in *hyphen-cased YAML*. These are typically loaded and serialized as *dataclasses* in Python.

2.2.1.1 Config loader

cognite.extractorutils.configtools.**load_yaml**(*source: Union[TextIO, str]*, *config_type: Type[T]*, *case_style: str = 'hyphen'*, *expand_envvars=True*) → T

Read a YAML file, and create a config object based on its contents.

Parameters

- **source** – Input stream (as returned by open(...)) or string containing YAML.
- **config_type** – Class of config type (i.e. your custom subclass of BaseConfig).
- **case_style** – Casing convention of config file. Valid options are 'snake', 'hyphen' or 'camel'. Should be 'hyphen'.
- **expand_envvars** – Substitute values with the pattern ${VAR} with the content of the environment variable VAR

Returns An initialized config object.

Raises *InvalidConfigError* – If any config field is given as an invalid type, is missing or is unknown

2.2.1.2 Base classes

The `configtools` module contains several prebuilt config classes for many common parameters. The class `BaseConfig` is intended as a starting point for a custom configuration schema, containing parameters for config version, CDF connection and logging.

Example:

```
@dataclass
class ExtractorConfig:
    state_store: Optional[StateStoreConfig]
    ...

@dataclass
class SourceConfig:
    ...


@dataclass
class MyConfig(BaseConfig):
```

(continues on next page)

(continued from previous page)

```
    extractor: ExtractorConfig
    source: SourceConfig
```

class cognite.extractorutils.configtools.**BaseConfig**(*version: Union[str, int, None], cognite: cognite.extractorutils.configtools.CogniteConfig, logger: cognite.extractorutils.configtools.LoggingConfig*)

Basis for an extractor config, containing config version, `CogniteConfig` and `LoggingConfig`

class cognite.extractorutils.configtools.**CogniteConfig**(*project: str, api_key: Optional[str], idp_authentication: Optional[cognite.extractorutils.authentication.Authentic data_set_id: Optional[int], external_id_prefix: str = '', host: str = 'https://api.cognitedata.com'*)

Configuration parameters for CDF connection, such as project name, host address and API key

class cognite.extractorutils.configtools.**LoggingConfig**(*console: Optional[cognite.extractorutils.configtools._ConsoleLog file: Optional[cognite.extractorutils.configtools._FileLogging*)

Logging settings, such as log levels and path to log file

class cognite.extractorutils.configtools.**MetricsConfig**(*push_gateways: Optional[List[cognite.extractorutils.configtools._PushG cognite: Optional[cognite.extractorutils.configtools._CogniteMe*)

Destination(s) for metrics, including options for one or several Prometheus push gateways, and pushing as CDF Time Series.

class cognite.extractorutils.configtools.**RawDestinationConfig**(*database: str, table: str*)

class cognite.extractorutils.configtools.**StateStoreConfig**(*raw: Union[cognite.extractorutils.configtools.RawSta NoneType] = None, local: Union[cognite.extractorutils.configtools.LocalSt NoneType] = None*)

class cognite.extractorutils.configtools.**RawStateStoreConfig**(*database: str, table: str, upload_interval: int = 30*)

class cognite.extractorutils.configtools.**LocalStateStoreConfig**(*path: str, save_interval: int = 30*)

2.2.1.3 Exceptions

exception cognite.extractorutils.configtools.**InvalidConfigError**(*message: str*)

Exception thrown from `load_yaml` if config file is invalid. This can be due to

- Missing fields

- Incompatible types
- Unkown fields

2.2.2 `metrics` - Automatic pushers of performance metrics

Module containing tools for pushers for metric reporting.

The classes in this module scrape the default Prometheus registry and uploads it periodically to either a Prometheus push gateway, or to CDF as time series.

The `BaseMetrics` class forms the basis for a metrics collection for an extractor, containing some general metrics that all extractors should report. To create your own set of metrics, subclass this class and populate it with extractor-specific metrics, as such:

```
class MyMetrics(BaseMetrics):
    def __init__(self):
        super().__init__(extractor_name="my_extractor", extractor_version=__version__)

        self.a_counter = Counter("my_extractor_example_counter", "An example counter")
        ...
```

class cognite.extractorutils.metrics.**AbstractMetricsPusher**(*push_interval: Optional[int] = None*, *thread_name: Optional[str] = None*)

Bases: `abc.ABC`

Base class for metric pushers. Metric pushers spawns a thread that routinely pushes metrics to a configured destination.

Contains all the logic for starting and running threads.

Parameters

- **push_interval** – Seconds between each upload call
- **thread_name** – Name of thread to start. If omitted, a standard name such as Thread-4 will be generated.

start() → None
Starts a thread that pushes the default registry to the configured gateway at certain intervals.

stop() → None
Stop the push loop.

class cognite.extractorutils.metrics.**BaseMetrics**(*extractor_name: str*, *extractor_version: str*, *process_scrape_interval: float = 15*)

Bases: `object`

Base collection of extractor metrics. The class also spawns a collector thread on init that regularly fetches process information and update the `process_*` gauges.

To create a set of metrics for an extractor, create a subclass of this class.

Note that only one instance of this class (or any subclass) can exist simultaneously

The collection includes the following metrics:

- startup: Startup time (unix epoch)

- finish: Finish time (unix epoch)
- process_num_threads Number of active threads. Set automatically.
- process_memory_bytes Memory usage of extractor. Set automatically.
- process_cpu_percent CPU usage of extractor. Set automatically.

Parameters

- **extractor_name** – Name of extractor, used to prefix metric names
- **process_scrape_interval** – Interval (in seconds) between each fetch of data for the `process_*` gauges

class cognite.extractorutils.metrics.**CognitePusher**(*cdf_client: cognite.client._cognite_client.CogniteClient*, *external_id_prefix: str*, *push_interval: int*, *asset: Optional[cognite.client.data_classes.assets.Asset] = None*, *thread_name: Optional[str] = None*)

Bases: *cognite.extractorutils.metrics.AbstractMetricsPusher*

Pusher to CDF. Creates time series in CDF for all Gauges and Counters in the default Prometheus registry.

Optional contextualization with an Asset to make the time series observable in Asset Data Insight. The given asset will be created at root level in the tenant if it doesn't already exist.

Parameters

- **cdf_client** – The CDF tenant to upload time series to
- **external_id_prefix** – Unique external ID prefix for this pusher.
- **push_interval** – Seconds between each upload call
- **asset** – Optional contextualization.
- **thread_name** – Name of thread to start. If omitted, a standard name such as Thread-4 will be generated.

start() → None

Starts a thread that pushes the default registry to the configured gateway at certain intervals.

stop() → None

Stop the push loop.

class cognite.extractorutils.metrics.**PrometheusPusher**(*job_name: str*, *url: str*, *push_interval: int*, *username: Optional[str] = None*, *password: Optional[str] = None*, *thread_name: Optional[str] = None*)

Bases: *cognite.extractorutils.metrics.AbstractMetricsPusher*

Pusher to a Prometheus push gateway.

Parameters

- **job_name** – Prometheus job name
- **username** – Push gateway credentials
- **password** – Push gateway credentials

- **url** – URL (with portnum) of push gateway
- **push_interval** – Seconds between each upload call
- **thread_name** – Name of thread to start. If omitted, a standard name such as Thread-4 will be generated.

clear_gateway() → None
Delete metrics stored at the gateway (reset gateway).

start() → None
Starts a thread that pushes the default registry to the configured gateway at certain intervals.

stop() → None
Stop the push loop.

cognite.extractorutils.metrics.**safe_get**(*cls: Type[T]*) → T
A factory for instances of metrics collections.

Since Prometheus doesn't allow multiple metrics with the same name, any subclass of BaseMetrics must never be created more than once. This function creates an instance of the given class on the first call and stores it, any subsequent calls with the same class as argument will return the same instance.

```
>>> a = safe_get(MyMetrics)  # This will create a new instance of MyMetrics
>>> b = safe_get(MyMetrics)  # This will return the same instance
>>> a is b
True
```

Parameters **cls** – Metrics class to either create or get a cached version of

Returns An instance of given class

2.2.3 statestore - Storing extractor state between runs locally or remotely

The statestore module contains classes for keeping track of the extraction state of individual items, facilitating incremental load and speeding up startup times.

At the beginning of a run the extractor typically calls the initialize method, which loads the states from the remote store (which can either be a local JSON file or a table in CDF RAW), and during and/or at the end of a run, the synchronize method is called, which saves the current states to the remote store.

class cognite.extractorutils.statestore.**AbstractStateStore**
Bases: abc.ABC

Base class for a state store.

delete_state(*external_id: str*) → None
Delete an external ID from the state store.

Parameters **external_id** – External ID to remove

expand_state(*external_id: str*, *low: Optional[Any] = None*, *high: Optional[Any] = None*) → None
Like set_state, but only sets state if the proposed state is outside the stored state. That is if e.g. low is lower than the stored low.

Parameters

- **external_id** – External ID of e.g. time series to store state of
- **low** – Low watermark
- **high** – High watermark

get_state(*external_id: Union[str, List[str]]*) → Union[Tuple[Any, Any], List[Tuple[Any, Any]]]
Get state(s) for external ID(s)

Parameters **external_id** – An external ID or list of external IDs to get states for

Returns A tuple with (low, high) watermarks, or a list of tuples

initialize(*force: bool = False*) → None
Get states from remote store

outside_state(*external_id: str*, *new_state: Any*) → bool
Check if a new proposed state is outside state interval (ie, if a new datapoint should be processed).

Returns true if new_state is outside of stored state or if external_id is previously unseen.

Parameters

- **external_id** – External ID to test
- **new_state** – Proposed new state to test

Returns True if new_state is higher than the stored high watermark or lower than the low watermark.

post_upload_handler() → Callable[[List[Dict[str, Union[str, List[Dict[Union[int, float, datetime.datetime], Union[int, float, str]]], List[Tuple[Union[int, float, datetime.datetime], Union[int, float, str]]]]]], None]
Get a callable suitable for passing to a time series upload queue as post_upload_function, that will automatically update the states in this state store when that upload queue is uploading.

Returns A function that expands the current states with the values given

set_state(*external_id: str*, *low: Optional[Any] = None*, *high: Optional[Any] = None*) → None
Set/update state of a singe external ID.

Parameters

- **external_id** – External ID of e.g. time series to store state of
- **low** – Low watermark
- **high** – High watermark

synchronize() → None
Upload states to remote store

class cognite.extractorutils.statestore.**LocalStateStore**(*file_path: str*)
Bases: *cognite.extractorutils.statestore.AbstractStateStore*

An extractor state store using a local JSON file as backend.

Parameters **file_path** – File path to JSON file to use

delete_state(*external_id: str*) → None
Delete an external ID from the state store.

Parameters **external_id** – External ID to remove

expand_state(*external_id: str*, *low: Optional[Any] = None*, *high: Optional[Any] = None*) → None
Like set_state, but only sets state if the proposed state is outside the stored state. That is if e.g. low is lower than the stored low.

Parameters

- **external_id** – External ID of e.g. time series to store state of
- **low** – Low watermark

- **high** – High watermark

get_state(*external_id: Union[str, List[str]]*) → Union[Tuple[Any, Any], List[Tuple[Any, Any]]]
Get state(s) for external ID(s)

Parameters **external_id** – An external ID or list of external IDs to get states for

Returns A tuple with (low, high) watermarks, or a list of tuples

initialize(*force: bool = False*) → None
Load states from specified JSON file

Parameters **force** – Enable re-initialization, ie overwrite when called multiple times

outside_state(*external_id: str*, *new_state: Any*) → bool
Check if a new proposed state is outside state interval (ie, if a new datapoint should be processed).

Returns true if new_state is outside of stored state or if external_id is previously unseen.

Parameters

- **external_id** – External ID to test
- **new_state** – Proposed new state to test

Returns True if new_state is higher than the stored high watermark or lower than the low watermark.

post_upload_handler() → Callable[[List[Dict[str, Union[str, List[Dict[Union[int, float, datetime.datetime], Union[int, float, str]]], List[Tuple[Union[int, float, datetime.datetime], Union[int, float, str]]]]]]], None]
Get a callable suitable for passing to a time series upload queue as post_upload_function, that will automatically update the states in this state store when that upload queue is uploading.

Returns A function that expands the current states with the values given

set_state(*external_id: str*, *low: Optional[Any] = None*, *high: Optional[Any] = None*) → None
Set/update state of a singe external ID.

Parameters

- **external_id** – External ID of e.g. time series to store state of
- **low** – Low watermark
- **high** – High watermark

synchronize() → None
Save states to specified JSON file

class cognite.extractorutils.statestore.**NoStateStore**
Bases: *cognite.extractorutils.statestore.AbstractStateStore*

A state store that only keeps states in memory and never stores or initializes from external sources.

delete_state(*external_id: str*) → None
Delete an external ID from the state store.

Parameters **external_id** – External ID to remove

expand_state(*external_id: str*, *low: Optional[Any] = None*, *high: Optional[Any] = None*) → None
Like set_state, but only sets state if the proposed state is outside the stored state. That is if e.g. low is lower than the stored low.

Parameters

- **external_id** – External ID of e.g. time series to store state of

- **low** – Low watermark
- **high** – High watermark

get_state(*external_id: Union[str, List[str]]*) → Union[Tuple[Any, Any], List[Tuple[Any, Any]]]
Get state(s) for external ID(s)

Parameters **external_id** – An external ID or list of external IDs to get states for

Returns A tuple with (low, high) watermarks, or a list of tuples

initialize(*force: bool = False*) → None
Get states from remote store

outside_state(*external_id: str*, *new_state: Any*) → bool
Check if a new proposed state is outside state interval (ie, if a new datapoint should be processed).

Returns true if new_state is outside of stored state or if external_id is previously unseen.

Parameters

- **external_id** – External ID to test
- **new_state** – Proposed new state to test

Returns True if new_state is higher than the stored high watermark or lower than the low watermark.

post_upload_handler() → Callable[[List[Dict[str, Union[str, List[Dict[Union[int, float, datetime.datetime], Union[int, float, str]]], List[Tuple[Union[int, float, datetime.datetime], Union[int, float, str]]]]]]], None]
Get a callable suitable for passing to a time series upload queue as post_upload_function, that will automatically update the states in this state store when that upload queue is uploading.

Returns A function that expands the current states with the values given

set_state(*external_id: str*, *low: Optional[Any] = None*, *high: Optional[Any] = None*) → None
Set/update state of a singe external ID.

Parameters

- **external_id** – External ID of e.g. time series to store state of
- **low** – Low watermark
- **high** – High watermark

synchronize() → None
Upload states to remote store

class cognite.extractorutils.statestore.**RawStateStore**(*cdf_client: cognite.client._cognite_client.CogniteClient*, *database: str*, *table: str*)
Bases: *cognite.extractorutils.statestore.AbstractStateStore*

An extractor state store based on CDF RAW.

Parameters

- **cdf_client** – Cognite client to use
- **database** – Name of CDF database
- **table** – Name of CDF table

delete_state(*external_id: str*) → None
Delete an external ID from the state store.

Parameters **external_id** – External ID to remove

expand_state(*external_id: str*, *low: Optional[Any] = None*, *high: Optional[Any] = None*) → None
Like set_state, but only sets state if the proposed state is outside the stored state. That is if e.g. low is lower than the stored low.

Parameters

- **external_id** – External ID of e.g. time series to store state of
- **low** – Low watermark
- **high** – High watermark

get_state(*external_id: Union[str, List[str]]*) → Union[Tuple[Any, Any], List[Tuple[Any, Any]]]
Get state(s) for external ID(s)

Parameters **external_id** – An external ID or list of external IDs to get states for

Returns A tuple with (low, high) watermarks, or a list of tuples

initialize(*force: bool = False*) → None
Get states from remote store

outside_state(*external_id: str*, *new_state: Any*) → bool
Check if a new proposed state is outside state interval (ie, if a new datapoint should be processed).

Returns true if new_state is outside of stored state or if external_id is previously unseen.

Parameters

- **external_id** – External ID to test
- **new_state** – Proposed new state to test

Returns True if new_state is higher than the stored high watermark or lower than the low watermark.

post_upload_handler() → Callable[[List[Dict[str, Union[str, List[Dict[Union[int, float, datetime.datetime], Union[int, float, str]]], List[Tuple[Union[int, float, datetime.datetime], Union[int, float, str]]]]]]], None]
Get a callable suitable for passing to a time series upload queue as post_upload_function, that will automatically update the states in this state store when that upload queue is uploading.

Returns A function that expands the current states with the values given

set_state(*external_id: str*, *low: Optional[Any] = None*, *high: Optional[Any] = None*) → None
Set/update state of a singe external ID.

Parameters

- **external_id** – External ID of e.g. time series to store state of
- **low** – Low watermark
- **high** – High watermark

synchronize() → None
Upload states to remote store

2.2.4 uploader - Batching upload queues with automatic upload triggers

Module containing upload queue classes. The UploadQueue classes chunks together items and uploads them together to CDF, both to minimize the load on the API, and also to speed up uploading as requests can be slow.

Each upload queue comes with some configurable conditions that, when met, automatically triggers an upload.

Note: You cannot assume that an element is uploaded when it is added to the queue, since the upload may be delayed. To ensure that everything is uploaded you should set the *post_upload_function* callback to verify. For example, for a time series queue you might want to check the latest time stamp, as such (assuming incremental time stamps and using timestamp-value tuples as data point format):

```
state_store = LocalStateStore("states.json")

queue = TimeSeriesUploadQueue(
    cdf_client=my_cognite_client,
    post_upload_function=state_store.post_upload_handler(),
    max_upload_interval=1
)
```

class cognite.extractorutils.uploader.**AbstractUploadQueue**(*cdf_client: cognite.client._cognite_client.CogniteClient, post_upload_function: Optional[Callable[[List[Any]], None]] = None, max_queue_size: Optional[int] = None, max_upload_interval: Optional[int] = None, trigger_log_level: str = 'DEBUG', thread_name: Optional[str] = None*)

Bases: abc.ABC

Abstract uploader class.

Parameters

- **cdf_client** – Cognite Data Fusion client to use
- **post_upload_function** – A function that will be called after each upload. The function will be given one argument: A list of the elements that were uploaded.
- **max_queue_size** – Maximum size of upload queue. Defaults to no max size.
- **max_upload_interval** – Automatically trigger an upload each m seconds when run as a thread (use start/stop methods).
- **trigger_log_level** – Log level to log upload triggers to.
- **thread_name** – Thread name of uploader thread.

add_to_upload_queue(**args*) → None

Adds an element to the upload queue. The queue will be uploaded if the queue byte size is larger than the threshold specified in the config.

start() → None

Start upload thread if max_upload_interval is set, this called the upload method every max_upload_interval seconds.

stop(*ensure_upload: bool = True*) → None

Stop upload thread if running, and ensures that the upload queue is empty if ensure_upload is True.

Parameters **ensure_upload** (*bool*) – (Optional). Call upload one last time after shutting down thread to ensure empty upload queue.

upload() → None
Uploads the queue.

class cognite.extractorutils.uploader.**EventUploadQueue**(*cdf_client: cognite.client._cognite_client.CogniteClient, post_upload_function: Optional[Callable[[List[cognite.client.data_classes.eve None]] = None, max_queue_size: Optional[int] = None, max_upload_interval: Optional[int] = None, trigger_log_level: str = 'DEBUG', thread_name: Optional[str] = None*)
Bases: *cognite.extractorutils.uploader.AbstractUploadQueue*

Upload queue for events

Parameters

- **cdf_client** – Cognite Data Fusion client to use
- **post_upload_function** – A function that will be called after each upload. The function will be given one argument: A list of the events that were uploaded.
- **max_queue_size** – Maximum size of upload queue. Defaults to no max size.
- **max_upload_interval** – Automatically trigger an upload each m seconds when run as a thread (use start/stop methods).
- **trigger_log_level** – Log level to log upload triggers to.
- **thread_name** – Thread name of uploader thread.

add_to_upload_queue(*event: cognite.client.data_classes.events.Event*) → None
Add event to upload queue. The queue will be uploaded if the queue size is larger than the threshold specified in the __init__.

Parameters **event** – Event to add

start() → None
Start upload thread if max_upload_interval is set, this called the upload method every max_upload_interval seconds.

stop(*ensure_upload: bool = True*) → None
Stop upload thread if running, and ensures that the upload queue is empty if ensure_upload is True.

Parameters **ensure_upload** (*bool*) – (Optional). Call upload one last time after shutting down thread to ensure empty upload queue.

upload() → None
Trigger an upload of the queue, clears queue afterwards

class cognite.extractorutils.uploader.**FileUploadQueue**(*cdf_client: cognite.client._cognite_client.CogniteClient, post_upload_function: Optional[Callable[[List[cognite.client.data_classes.event*
None]] = None, max_queue_size: Optional[int] = None, max_upload_interval: Optional[int] = None, trigger_log_level: str = 'DEBUG', thread_name: Optional[str] = None, overwrite_existing: bool = False)

Bases: *cognite.extractorutils.uploader.AbstractUploadQueue*

Upload queue for files

Parameters

- **cdf_client** – Cognite Data Fusion client to use
- **post_upload_function** – A function that will be called after each upload. The function will be given one argument: A list of the events that were uploaded.
- **max_queue_size** – Maximum size of upload queue. Defaults to no max size.
- **max_upload_interval** – Automatically trigger an upload each m seconds when run as a thread (use start/stop methods).
- **trigger_log_level** – Log level to log upload triggers to.
- **thread_name** – Thread name of uploader thread.

add_to_upload_queue(*file_meta: cognite.client.data_classes.files.FileMetadata, file_name: Union[str, os.PathLike] = None*) → None

Add file to upload queue. The queue will be uploaded if the queue size is larger than the threshold specified in the __init__.

Parameters

- **file_meta** – File metadata-object
- **file_name** – Path to file to be uploaded. If none, the file object will still be created, but no data is uploaded

start() → None

Start upload thread if max_upload_interval is set, this called the upload method every max_upload_interval seconds.

stop(*ensure_upload: bool = True*) → None

Stop upload thread if running, and ensures that the upload queue is empty if ensure_upload is True.

Parameters **ensure_upload** (*bool*) – (Optional). Call upload one last time after shutting down thread to ensure empty upload queue.

upload() → None

Trigger an upload of the queue, clears queue afterwards

class cognite.extractorutils.uploader.**RawUploadQueue**(*cdf_client: cognite.client._cognite_client.CogniteClient, post_upload_function: Optional[Callable[[List[Any]], None]] = None, max_queue_size: Optional[int] = None, max_upload_interval: Optional[int] = None, trigger_log_level: str = 'DEBUG', thread_name: Optional[str] = None*)

Bases: *cognite.extractorutils.uploader.AbstractUploadQueue*

Upload queue for RAW

Parameters

- **cdf_client** – Cognite Data Fusion client to use
- **post_upload_function** – A function that will be called after each upload. The function will be given one argument: A list of the rows that were uploaded.
- **max_queue_size** – Maximum size of upload queue. Defaults to no max size.
- **max_upload_interval** – Automatically trigger an upload each m seconds when run as a thread (use start/stop methods).
- **trigger_log_level** – Log level to log upload triggers to.
- **thread_name** – Thread name of uploader thread.

add_to_upload_queue(*database: str*, *table: str*, *raw_row: cognite.client.data_classes.raw.Row*) → None

Adds a row to the upload queue. The queue will be uploaded if the queue size is larger than the threshold specified in the __init__.

Parameters

- **database** – The database to upload the Raw object to
- **table** – The table to upload the Raw object to
- **raw_row** – The row object

start() → None

Start upload thread if max_upload_interval is set, this called the upload method every max_upload_interval seconds.

stop(*ensure_upload: bool = True*) → None

Stop upload thread if running, and ensures that the upload queue is empty if ensure_upload is True.

Parameters **ensure_upload** (*bool*) – (Optional). Call upload one last time after shutting down thread to ensure empty upload queue.

upload() → None

Trigger an upload of the queue, clears queue afterwards

class cognite.extractorutils.uploader.**SequenceUploadQueue**(*cdf_client: cognite.client._cognite_client.CogniteClient, post_upload_function: Optional[Callable[[List[Any]], None]] = None, max_queue_size: Optional[int] = None, max_upload_interval: Optional[int] = None, trigger_log_level: str = 'DEBUG', thread_name: Optional[str] = None, create_missing=False*)

Bases: *cognite.extractorutils.uploader.AbstractUploadQueue*

add_to_upload_queue(*rows: Union[Dict[int, List[Union[int, str, float]]], List[Tuple[int, Union[int, float, str]]], List[Dict[str, Any]], cognite.client.data_classes.sequences.SequenceData], column_external_ids: Optional[List[str]] = None, id: int = None, external_id: str = None*) → None

Add sequence rows to upload queue. Mirrors implementation of SequenceApi.insert. Inserted rows will be cached until uploaded

Parameters

- **rows** – The rows to be inserted. Can either be a list of tuples, a list of ["rownumber": ..., "values": ...] objects, a dictionary of rowNumber: data, or a SequenceData object.
- **column_external_ids** – List of external id for the columns of the sequence
- **id** – Sequence internal ID Use if external_id is None
- **external_id** – Sequence external ID Us if id is None

set_sequence_column_definition(*col_def: List[Dict[str, str]], id: int = None, external_id: str = None*) → None

Set sequence column definition

Parameters

- **col_def** – Sequence column definition
- **id** – Sequence internal ID Use if external_id is None
- **external_id** – Sequence external ID Us if id is None

set_sequence_metadata(*metadata: Dict[str, Union[str, int, float]], id: int = None, external_id: str = None, asset_external_id: str = None, dataset_external_id: str = None*) → None

Set sequence metadata. Metadata will be cached until the sequence is created. The metadata will be updated if the sequence already exists

Parameters

- **metadata** – Sequence metadata
- **id** – Sequence internal ID Use if external_id is None
- **external_id** – Sequence external ID Us if id is None
- **asset_external_id** – Sequence asset ID

- **dataset_external_id** – Sequence dataset id

start() → None
Start upload thread if max_upload_interval is set, this called the upload method every max_upload_interval seconds.

stop(*ensure_upload: bool = True*) → None
Stop upload thread if running, and ensures that the upload queue is empty if ensure_upload is True.

Parameters **ensure_upload** (*bool*) – (Optional). Call upload one last time after shutting down thread to ensure empty upload queue.

upload() → None
Trigger an upload of the queue, clears queue afterwards

class cognite.extractorutils.uploader.**TimeSeriesUploadQueue**(*cdf_client: cognite.client._cognite_client.CogniteClient, post_upload_function: Optional[Callable[[List[Dict[str, Union[str, List[Dict[Union[int, float, datetime.datetime], Union[int, float, str]]], List[Tuple[Union[int, float, datetime.datetime], Union[int, float, str]]]]]], None]] = None, max_queue_size: Optional[int] = None, max_upload_interval: Optional[int] = None, trigger_log_level: str = 'DEBUG', thread_name: Optional[str] = None, create_missing: bool = False*)

Bases: *cognite.extractorutils.uploader.AbstractUploadQueue*

Upload queue for time series

Parameters

- **cdf_client** – Cognite Data Fusion client to use
- **post_upload_function** – A function that will be called after each upload. The function will be given one argument: A list of dicts containing the datapoints that were uploaded (on the same format as the kwargs in datapoints upload in the Cognite SDK).
- **max_queue_size** – Maximum size of upload queue. Defaults to no max size.
- **max_upload_interval** – Automatically trigger an upload each m seconds when run as a thread (use start/stop methods).

- **trigger_log_level** – Log level to log upload triggers to.
- **thread_name** – Thread name of uploader thread.
- **create_missing** – Create missing time series if possible (ie, if external id is used)

add_to_upload_queue(**, id: int = None, external_id: str = None, datapoints: Union[List[Dict[Union[int, float, datetime.datetime], Union[int, float, str]]], List[Tuple[Union[int, float, datetime.datetime], Union[int, float, str]]]] = []*) → None

Add data points to upload queue. The queue will be uploaded if the queue size is larger than the threshold specified in the __init__.

Parameters

- **id** – Internal ID of time series. Either this or external_id must be set.
- **external_id** – External ID of time series. Either this or external_id must be set.
- **datapoints** – List of data points to add

start() → None

Start upload thread if max_upload_interval is set, this called the upload method every max_upload_interval seconds.

stop(*ensure_upload: bool = True*) → None

Stop upload thread if running, and ensures that the upload queue is empty if ensure_upload is True.

Parameters **ensure_upload** (*bool*) – (Optional). Call upload one last time after shutting down thread to ensure empty upload queue.

upload() → None

Trigger an upload of the queue, clears queue afterwards

2.2.5 util - Miscellaneous utilities

The util package contains miscellaneous functions and classes that can some times be useful while developing extractors.

class cognite.extractorutils.util.**EitherId**(***kwargs*)

Bases: object

Class representing an ID in CDF, which can either be an external or internal ID. An EitherId can only hold one ID type, not both.

Parameters

- **id** – Internal ID
- **or external_id** (*externalId*) – external ID

Raises TypeError – If none of both of id types are set.

content() → Union[int, str]

Get the value of the ID

Returns The ID

type() → str

Get the type of the ID

Returns 'id' if the EitherId represents an internal ID, 'externalId' if the EitherId represents an external ID

cognite.extractorutils.util.**ensure_time_series**(*cdf_client: cognite.client._cognite_client.CogniteClient, time_series: Iterable[cognite.client.data_classes.time_series.TimeSeries]*) → None

Ensure that all the given time series exists in CDF.

Searches through the tenant after the external IDs of the time series given, and creates those that are missing.

Parameters

- **cdf_client** – Tenant to create time series in
- **time_series** – Time series to create

cognite.extractorutils.util.**set_event_on_interrupt**(*stop_event: threading.Event*) → None

Set given event on SIGINT (Ctrl-C) instead of throwing a KeyboardInterrupt exception.

Parameters **stop_event** – Event to set

Python Module Index

C

Index

T

U