# connectordb Documentation

***Release 0.3.0***

**ConnectorDB contributors**

**Apr 10, 2018**

# Contents

CHAPTER 1

# Tutorial

This page is the tutorial for ConnectorDB's python interface.

## 1.1 Installing

To start using ConnectorDB, you need to install it!:

```
pip install connectordb
```

If you don't have pip installed, you can follow these instructions.

You will also want to install apsw:

```
sudo apt-get install python-apsw
```

If on Windows, you can get the binaries from here. APSW is used for logging support.

## 1.2 Connecting

Once installed, you can log in as a user:

```
import connectordb
cdb = connectordb.ConnectorDB("username","password",url="https://cdb.mysite.com")
```

...and as a device:

```
import connectordb
cdb = connectordb.ConnectorDB("apikey",url="https://cdb.mysite.com")
```

After logging in, you can check your device by looking at the path:

```
>>> cdb.path
'myuser/user'
```

## 1.3 Basics

Let's check what streams we have:

```
>>> cdb.streams()
[[Stream:myuser/user/productivity], [Stream:myuser/user/mood]]
```

Since you're logged in as a device, you see your streams. You can access this device's streams directly by name:

```
>>> productivity = cdb["productivity"]
>>> productivity
[Stream:myuser/user/productivity]
```

Let's see some of the stream's properties:

```
>>> productivity.schema
{'type': 'integer', 'maximum': 10, 'minimum': 0}
>>> productivity.datatype
'rating.stars'
>>> productivity.description
''
```

This is a star-rating stream. This stream should have a description, though - let's set it!

```
>>> productivity.description = "A star rating of my productivity"
>>> productivity.description
'A star rating of my productivity'
```

You can also see all of the stream's properties and set them:

```
>>> productivity.data
{'name': 'productivity', 'ephemeral': False, 'datatype': 'rating.stars', 'description
↪': 'A star rating of my productivity', 'downlink': False, 'schema': '{"type":
↪"integer","minimum":0,"maximum":10}', 'icon': '', 'nickname': ''}
>>> productivity.set({"nickname": "My Productivity"})
```

The same methods of access work for users and devices as well.

## 1.4 Stream Data

Let's do some basic analysis of the productivity stream:

```
>>> len(productivity)
9
```

Looks like I only have 9 productivity ratings in my stream. All of the data in streams is accessible like a python array. The first element of the array is the oldest datapoint. ConnectorDB's streams are append-only, so you don't need to worry about data disappearing/appearing in the middle of the array.

Let's get the most recent productivity rating

```
>>> productivity[-1]
{'t': 1464334185.668, 'd': 8}
```

Looks like I was really productive recently (8/10 stars)! When exactly was this timestamp?

```
>>> from datetime import datetime
>>> datetime.fromtimestamp(productivity[-1]["t"])
'Fri May 27 03:29:45 2016'
```

Let's perform analysis of the whole stream. We can get the full stream's data by getting productivity[:]

```
>>> productivity[:]
[{'t': 1464250199.934, 'd': 8}, {'t': 1464334179.605, 'd': 7}, {'t': 1464334180.216,
↪'d': 5}, {'t': 1464334180.88, 'd': 9}, {'t': 1464334181.782, 'd': 3}, {'t':␣
↪1464334183.308, 'd': 1}, {'t': 1464334183.752, 'd': 5}, {'t': 1464334184.46, 'd': 4}
↪, {'t': 1464334185.668, 'd': 8}]
```

For our current analysis, we don't need the timestamps:

```
>>> productivity[:].d()
[8, 7, 5, 9, 3, 1, 5, 4, 8]
```

The returned data is of a special list class that has some useful extensions, such as directly computing the data array. Let's find the mean:

```
>>> productivity[:].mean()
5.555555555555555
```

If we only care about the mean, it is inefficient to query the entire dataset from ConnectorDB, only to perform an aggregation that returns a single value. We can use PipeScript to perform the aggregation on the server:

```
>>> productivity(transform="mean | if last")
[{'t': 1464334185.668, 'd': 5.555555555555555}]
```

You can go here for a PipeScript tutorial (PipeScript is ConnectorDB's transform engine)

Using the call syntax, you can also query ConnectorDB by time. To get the datapoints from the last minute:

```
>>> productivity(t1=time.time() -60, t2=time.time())
```

Finally, let's plot the rating vs time:

```
>>> from pylab import *
>>> data = productivity[:]
>>> plot(data.t(),data.d())
>>> show()
```

## 1.5 Subscribing

Suppose now that you want to do something whenever your mood is greater than 8 stars. To do this, you need to somehow be notified when this happens. ConnectorDB allows devices to subscribe to streams, so that you get data the moment it is inserted:

```
>>> def subscriber(stream,data):
...     print(stream,data)
>>> productivity.subscribe(subscriber)
```

Now go to the ConnectorDB web app, and change your productivity rating. You should see your new data be printed the moment you click on the rating.

But we only want to get datapoints where productivity is greater than 8! Let's unsubscribe.

```
>>> productivity.unsubscribe()
```

ConnectorDB's subscriptions accept transforms, so we filter the datapoints with rating 8 or lower.

```
>>> productivity.subscribe(subscriber,transform="if $>8")
```

Now you should only get messages when the rating is greater than 8 stars!

Subscribing allows your devices to react to your data. Before continuing, let's unsubscribe:

```
>>> productivity.unsubscribe(transform="if $>8")
```

The transform string used during unsubscribing must be exactly the same as the one used when subscribing, because you can have multiple subscriptions each with different transforms.

## 1.6 Devices

We know how to view data in ConnectorDB - let's figure out how to create it in the first place.

We will go back to the cdb device we logged in with. Let's make a new stream:

```
>>> newstream = cdb["newstream"]
>>> newstream.exists()
False
```

This stream doesn't exist yet, so make it:

```
>>> newstream.create({"type":"string"})
```

Let's add data!

```
>>> len(newstream)
0
>>> newstream.insert("Hello World!")
>>> len(newstream)
1
```

Note that we are currently logged in as the user device. This is not recommended. ConnectorDB is built with the assumption that every physical program/object using it has its own associated device, using which it accesses the database. Therefore, let's create a new device for ourselves.

We must first go to the user to list devices

```
>>> cdb.user.devices()
[[Device:test/user], [Device:test/meta]]
```

ConnectorDB comes with two devices by default, the user and meta device. The meta device is hidden in the web interface, as it holds log streams. The user device represents the user.

```
>>> newdevice = cdb.user["newdevice"]
>>> newdevice.exists()
False
>>> newdevice.create()
```

Now let's log in as that device:

```
>>> newdevice.apikey
'4d79a2c0-3a02-45da-7131-9f5f3d6e4696'
>>> mydevice = connectordb.ConnectorDB(newdevice.apikey,url="https://cdb.mysite.com")
```

You'll notice that this device is completely isolated - it does not have access to anything but itself and its own streams. This is because the default role given to devices assumes that they are not to be trusted with data.

**Warning:** ConnectorDB's permissions structure is there to disallow snooping - and not active malice. Each device can create an arbitrary amount of streams and is not rate limited by default.

## 1.7 Downlinks

One of the powerful features of ConnectorDB are downlinks. First let's see an unusual property of devices:

```
>>> mys = mydevice["mystream"]
>>> mys.create({"type": "number"})
```

```
>>> s = newdevice["mystream"]
```

Notice that both s and mys refer to the same stream. The difference between the two is that s is logged in as a user, and has access to everything, and mys is logged in as the device which owns mystream.

```
>>> mys.insert(54)
>>> s.insert(12)
connectordb._connection.AuthenticationError: '403: Write access to stream data denied.
↪ (529afdba-9cdc-48ae-4fbb-0e8adf6d3ed9)'
```

What happened here? Shouldn't s be able to write the stream?

ConnectorDB is set up such that only the owning device can write its streams. This is to enforce isolation. Each device should only write to its own streams.

**Note:** All permissions can be modified to suit your liking in connectordb's configuration files. This behavior is in the default configuration.

There is one major case where this behavior would be suboptimal. Suppose you want to control your lights through ConnectorDB. Your lights create a stream which gives the current on/off state, and want other devices to be able to turn the lights on and off.

This is what the downlink property of a stream is for

```
>>> mys.downlink = True
```

Now you can insert the data!

```
>>> len(s)
>>> 2
>>> s.insert(3)
>>> len(s)
>>> 2
```

...It looks like the insert succeeded, but the data wasn't inserted!?

ConnectorDB's downlinks do not actually permit you to insert data directly to the stream - the stream reflects reality, and your lights are currently off. The intervention (turn lights on/set thermostat to 75F) is placed into a special downlink stream

```
>>> s.length(downlink=True)
1
>>> s(downlink=True)
[{'t': 1464350691.0983202, 'd': 3, 'o': 'test/user'}]
```

The downlink stream says that the device 'test/user' wants the value to be 3. Now it is the owning device's (lights) job to set the actual stream value correctly.

This would usually be done by subscribing to the downlink stream

```
>>> def lightcontrol(streamname,data):
...     print("The lights are now",data[-1]["d"])
...     return data
>>> mys.subscribe(lightcontrol,downlink=True)
```

By returning True from the light control callback, or returning the data, we're acknowledging that we set the value - and the stream value is accepted. We can also return an arbitrary datapoint to set a different value, or return False, or nothing at all, which will not acknowledge the datapoint. This is useful when there is a time delay between setting goal value and actual value (such as when controlling a thermostat).

Now, when we set values, they are inserted to the stream after being acknowledged by the device:

```
>>> len(s)
2
>>> s.insert(9)
>>> len(s)
3
```

And that's it!

You now know enough to begin using ConnectorDB. There are two major components which were not touched upon in this tutorial: logging and datasets.

The python interface includes special logging code which allows you to easily write logging devices which periodically synchronize data with ConnectorDB. If gathering data from sensors, you probably want to use the logger.

Datasets are in the queries section - they enable you to perform computation by combining multiple streams into one tabular structure which is easy to plug into machine learning and statistical packages. If doing advanced analysis, you'll want to look at datasets.

CHAPTER 2

# Core API Reference

This page is the direct API reference.

The User, Device, and Stream objects all inherit from ConnectorObject, meaning that all methods and properties in ConnectorObject can be accessed from any other object in the core API.

The ConnectorDB object is the API main entrance point, and it inherits from Device. When logging in to ConnectorDB, you are logging in as a device, and all operations are done in reference to that device:

```
import connectordb
cdb = connectordb.ConnectorDB("apikey")
#Prints the full username/devicename path of the logged in device
print cdb.path
```

This is something you must be aware of when logging in as a user. Using a password actually logs you in as the user device, and all operations are done in reference to this device. Therefore, when logging in as a user, you will need to do the following:

```
import connectordb
cdb = connectordb.ConnectorDB("username","password")
# cdb is now the username/user device
myuser = cdb.user
# myuser is the "username" user, which can list devices
print myuser.devices()
```

## 2.1 ConnectorObject

**class** connectordb._connectorobject.**ConnectorObject**(*database_connection*, *object_path*)

Bases: object

Users, devices and streams are all built upon the base *ConnectorObject*. The methods from ConnectorObject can be accessed from any user, device or stream.

Do not use this object directly. The API is accessed using the ConnectorDB class (below).

**data**
Returns the raw dict representing metadata

**delete**()
Deletes the user/device/stream

**description**
Allows to directly set the object's description. Use as a property

**exists**()
returns true if the object exists, and false otherwise. This is useful for creating streams if they exist:

```
cdb = connectordb.ConnectorDB("myapikey")

mystream = cdb["mystream"]

if not mystream.exists():
    mystream.create({"type":"string"})
```

**icon**
Allows to directly get and set the icon. An icon can be URLencoded (data:image/) or use an icon from the material design set (https://material.io/icons/), prepended with "material:", and with spaces replaced by underscores.

**name**
Returns the object's name. Object names are immutable (unless logged in is a database admin)

**nickname**
Allows to directly set the object's user-friendly nickname. Usage is as a property:

```
cdb.nickname = "My Nickname!"

print cdb.nickname
```

**refresh**()
Refresh reloads data from the server. It raises an error if it fails to get the object's metadata

**set**(*property_dict*)
Attempts to set the given properties of the object. An example of this is setting the nickname of the object:

```
cdb.set({"nickname": "My new nickname"})
```

note that there is a convenience property *cdb.nickname* that allows you to get/set the nickname directly.

## 2.2 ConnectorDB

**class** connectordb._connectordb.**ConnectorDB**(*user_or_apikey=None*, *user_password=None*, *url='http://localhost:3124'*)

Bases: *connectordb._device.Device*

ConnectorDB is the main entry point for any application that uses the python API. The class accepts both a username and password in order to log in as a user, and accepts an apikey when logging in directly from a device:

```
import connectordb
cdb = connectordb.ConnectorDB("myusername","mypassword")

#prints "myusername/user" - logging in by username/password combo
```

```
#logs in as the user device.
print cdb.path
```

**close**()
shuts down all active connections to ConnectorDB

**count_devices**()
Gets the total number of devices registered with the database. Only available to administrator.

**count_streams**()
Gets the total number of streams registered with the database. Only available to administrator.

**count_users**()
Gets the total number of users registered with the database. Only available to administrator.

**import_users**(*directory*)
Imports version 1 of ConnectorDB export. These exports can be generated by running user.export(dir), possibly on multiple users.

**info**()
returns a dictionary of information about the database, including the database version, the transforms and the interpolators supported:

```
>>>cdb = connectordb.ConnectorDB(apikey)
>>>cdb.info()
{
    "version": "0.3.0",
    "transforms": {
        "sum": {"description": "Returns the sum of all the datapoints that go␣
↪through the transform"}
        ...
    },
    "interpolators": {
        "closest": {"description": "Uses the datapoint closest to the␣
↪interpolation timestamp"}
        ...
    }
}
```

**ping**()
Pings the ConnectorDB server. Useful for checking if the connection is valid

**reset_apikey**()
invalidates the device's current api key, and generates a new one. Resets current auth to use the new apikey, since the change would have future queries fail if they use the old api key.

**users**()
Returns the list of users in the database

## 2.3 User

**class** connectordb._user.**User**(*database_connection*, *object_path*)
Bases: *connectordb._connectorobject.ConnectorObject*

**create**(*email*, *password*, *role='user'*, *public=True*, ***kwargs*)
Creates the given user - using the passed in email and password.

You can also set other default properties by passing in the relevant information:

```
usr.create("my@email","mypass",description="I like trains.")
```

Furthermore, ConnectorDB permits immediate initialization of an entire user tree, so that you can create all relevant devices and streams in one go:

```
usr.create("my@email","mypass",devices={
    "device1": {
        "nickname": "My train",
        "streams": {
            "stream1": {
                "schema": "{"type":"string"}",
                "datatype": "train.choochoo"
            }
        },
    }
})
```

The user and meta devices are created by default. If you want to add streams to the user device, use the "streams" option in place of devices in create.

**devices**()
Returns the list of devices that belong to the user

**email**
gets the user's email address

**export**(*directory*)
Exports the ConnectorDB user into the given directory. The resulting export can be imported by using the import command(cdb.import(directory)),

Note that Python cannot export passwords, since the REST API does not expose password hashes. Therefore, the imported user will have password same as username.

The user export function is different than device and stream exports because it outputs a format compatible directly with connectorDB's import functionality:

connectordb import < mydatabase > <directory >

This also means that you can export multiple users into the same directory without issue

**import_device**(*directory*)
Imports a device from the given directory. You export the device by using device.export()

There are two special cases: user and meta devices. If the device name is meta, import_device will not do anything. If the device name is "user", import_device will overwrite the user device even if it exists already.

**public**
gets whether the user is public (this means different things based on connectordb permissions setup - connectordb.com has this be whether the user is publically visible. Devices are individually public / private.)

**role**
Gets the role of the user. This is the permissions level that the user has. It might not be accessible depending on the permissions setup of ConnectorDB. Returns None if not accessible

**set_password**(*new_password*)
Sets a new password for the user

**streams**(*public=False*, *downlink=False*, *visible=True*)
Returns the list of streams that belong to the user. The list can optionally be filtered in 3 ways:

- public: when True, returns only streams belonging to public devices

- downlink: If True, returns only downlink streams
- visible: If True (default), returns only streams of visible devices

## 2.4 Device

**class** connectordb._device.**Device**(*database_connection*, *object_path*)
Bases: *connectordb._connectorobject.ConnectorObject*

**apikey**
gets the device's api key. Returns None if apikey not accessible.

**create**(*public=False*, ***kwargs*)
Creates the device. Attempts to create private devices by default, but if public is set to true, creates public devices.

You can also set other default properties by passing in the relevant information. For example, setting a device with the given nickname and description:

```
dev.create(nickname="mydevice", description="This is an example")
```

Furthermore, ConnectorDB supports creation of a device's streams immediately, which can considerably speed up device setup:

```
dev.create(streams={
    "stream1": {"schema": '{"type":"number"}'}
})
```

Note that the schema must be encoded as a string when creating in this format.

**enabled**
gets whether the device is enabled. This allows a device to notify ConnectorDB when it is active and when it is not running

**export**(*directory*)
Exports the device to the given directory. The directory can't exist. You can later import this device by running import_device on a user.

**import_stream**(*directory*)
Imports a stream from the given directory. You export the Stream by using stream.export()

**public**
gets whether the device is public (this means different things based on connectordb permissions setup - connectordb.com has this be whether the device is publically visible. Devices are individually public/private.)

**reset_apikey**()
invalidates the device's current api key, and generates a new one

**role**
Gets the role of the device. This is the permissions level that the device has. It might not be accessible depending on the permissions setup of ConnectorDB. Returns None if not accessible

**streams**()
Returns the list of streams that belong to the device

**user**
user returns the user which owns the given device

## 2.5 Stream

**class** connectordb._stream.**Stream**(*database_connection*, *object_path*)

Bases: *connectordb._connectorobject.ConnectorObject*

**__call__**(*t1=None*, *t2=None*, *limit=None*, *i1=None*, *i2=None*, *downlink=False*, *transform=None*)

By calling the stream as a function, you can query it by either time range or index, and further you can perform a custom transform on the stream:

```
#Returns all datapoints with their data < 50 from the past minute
stream(t1=time.time()-60, transform="if $ < 50")

#Performs an aggregation on the stream, returning a single datapoint
#which contains the sum of the datapoints
stream(transform="sum | if last")
```

**__getitem__**(*getrange*)

Allows accessing the stream just as if it were just one big python array. An example:

```
#Returns the most recent 5 datapoints from the stream
stream[-5:]

#Returns all the data the stream holds.
stream[:]
```

In order to perform transforms on the stream and to aggreagate data, look at __call__, which allows getting index ranges along with a transform.

**__len__**()

taking len(stream) returns the number of datapoints saved within the database for the stream

**__module__ = 'connectordb._stream'**

**__repr__**()

Returns a string representation of the stream

**append**(*data*)

Same as insert, using the pythonic array name

**create**(*schema='{}'*, ***kwargs*)

Creates a stream given an optional JSON schema encoded as a python dict. You can also add other properties of the stream, such as the icon, datatype or description. Create accepts both a string schema and a dict-encoded schema.

**datatype**

returns the stream's registered datatype. The datatype suggests how the stream can be processed.

**device**

returns the device which owns the given stream

**downlink**

returns whether the stream is a downlink, meaning that it accepts input (like turning lights on/off)

**ephemeral**

returns whether the stream is ephemeral, meaning that data is not saved, but just passes through the messaging system.

**export**(*directory*)

Exports the stream to the given directory. The directory can't exist. You can later import this device by running import_stream on a device.

**insert**(*data*)
insert inserts one datapoint with the given data, and appends it to the end of the stream:

```
s = cdb["mystream"]

s.create({"type": "string"})

s.insert("Hello World!")
```

**insert_array**(*datapoint_array*, *restamp=False*)
given an array of datapoints, inserts them to the stream. This is different from insert(), because it requires an array of valid datapoints, whereas insert only requires the data portion of the datapoint, and fills out the rest:

```
s = cdb["mystream"]
s.create({"type": "number"})

s.insert_array([{"d": 4, "t": time.time()},{"d": 5, "t": time.time()}],␣
↪restamp=False)
```

The optional *restamp* parameter specifies whether or not the database should rewrite the timestamps of datapoints which have a timestamp that is less than one that already exists in the database.

That is, if restamp is False, and a datapoint has a timestamp less than a datapoint that already exists in the database, then the insert will fail. If restamp is True, then all datapoints with timestamps below the datapoints already in the database will have their timestamps overwritten to the same timestamp as the most recent datapoint hat already exists in the database, and the insert will succeed.

**length**(*downlink=False*)

**schema**
Returns the JSON schema of the stream as a python dict.

**sschema**
Returns the JSON schema of the stream as a string

**subscribe**(*callback*, *transform=''*, *downlink=False*)
Subscribes to the stream, running the callback function each time datapoints are inserted into the given stream. There is an optional transform to the datapoints, and a downlink parameter.:

```
s = cdb["mystream"]

def subscription_callback(stream,data):
    print stream, data

s.subscribe(subscription_callback)
```

The downlink parameter is for downlink streams - it allows to subscribe to the downlink substream, before it is acknowledged. This is especially useful for something like lights - have lights be a boolean downlink stream, and the light itself be subscribed to the downlink, so that other devices can write to the light, turning it on and off:

```
def light_control(stream,data):
    light_boolean = data[0]["d"]
    print "Setting light to", light_boolean
    set_light(light_boolean)

    #Acknowledge the write
    return True
```

```
# We don't care about intermediate values, we only want the most recent␣
↪setting
# of the light, meaning we want the "if last" transform
s.subscribe(light_control, downlink=True, transform="if last")
```

**unsubscribe**(*transform=''*, *downlink=False*)

Unsubscribes from a previously subscribed stream. Note that the same values of transform and downlink must be passed in order to do the correct unsubscribe:

```
s.subscribe(callback,transform="if last")
s.unsubscribe(transform="if last")
```

**user**

user returns the user which owns the given stream

connectordb._stream.**query_maker**(*t1=None*, *t2=None*, *limit=None*, *i1=None*, *i2=None*, *transform=None*, *downlink=False*)

query_maker takes the optional arguments and constructs a json query for a stream's datapoints using it:

```
#{"t1": 5, "transform": "if $ > 5"}
print query_maker(t1=5,transform="if $ > 5")
```

# CHAPTER 3

## Logging Data

Sometimes you just want to gather data at all times, and sync to ConnectorDB periodically. The logger allows you to do exactly such a thing.

The logger caches datapoints to a local sqlite database, and synchronizes with ConnectorDB every 10 minutes by default.

Suppose you have a temperature sensor on a computer with an intermittent internet connection.

You can use the Logger to cache data until a sync can happen:

```
def getTemperature():
    #Your code here
    pass

from connectordb.logger import Logger

def initlogger(l):
    # This function is called when first creating the Logger, to initialize the values

    # api key is needed to get access to ConnectorDB
    l.apikey = raw_input("apikey:")

    # If given a schema (as we have done here), addStream will create the stream if␣
↪it doesn't exist
    l.addStream("temperature",{"type":"number"})

    # Sync with ConnectorDB once an hour (in seconds)
    l.syncperiod = 60*60

# Open the logger using a cache file name (where datapoints are cached before syncing)
l = Logger("cache.db", on_create=initlogger)

# Start running syncer in background (can manually run l.sync() instead)
l.start()

# While the syncer is running in the background, we are free to add data
```

```
# to the cache however we want - it will be saved first to the cache file
# so that you don't lose any data, and will be synced to the database once an hour
while True:
    time.sleep(60)
    l.insert("temperature",getTemperature())
```

The logger requires the python-apsw package to work. It is a thread-safe sqlite wrapper, which is used to safely store your data between synchronization attempts.

On ubuntu, you can run `apt-get install python-apsw`. On windows, you will need to download the extension package from http://www.lfd.uci.edu/~gohlke/pythonlibs/#apsw , and install using pip.

## 3.1 Logger

**class** connectordb.logger.**Logger**(*database_file_path*, *on_create=None*, *apikey=None*, *onsync=None*, *onsyncfail=None*, *syncraise=False*)

Bases: `object`

Logger enables logging datapoints with periodic synchronization to a ConnectorDB database. the logged datapoints are cached in a sqlite database, as well as the necessary connection data, so that no data is lost, and settings don't need to be reloaded from the database after initial connection.

**addStream**(*streamname*, *schema=None*, ***kwargs*)

Adds the given stream to the logger. Requires an active connection to the ConnectorDB database.

If a schema is not specified, loads the stream from the database. If a schema is specified, and the stream does not exist, creates the stream. You can also add stream properties such as description or nickname to be added during creation.

**addStream_force**(*streamname*, *schema=None*)

This function adds the given stream to the logger, but does not check with a ConnectorDB database to make sure that the stream exists. Use at your own risk.

**apikey**

The API key used to connect to ConnectorDB. This needs to be set before the logger can do anything! The apikey only needs to be set once, since it is stored in the Logger database.

Note that changing the api key is not supported during logger runtime (after start is called). Logger must be recreated for a changed apikey to come into effect.

**cleardata**()

Deletes all cached data without syncing it to the server

**close**()

Closes the database connections and stops all synchronization.

**connectordb**

Returns the ConnectorDB object that the logger uses. Raises an error if Logger isn't able to connect

**data**

The data property allows the user to save settings/data in the database, so that there does not need to be extra code messing around with settings.

Use this property to save things that can be converted to JSON inside the logger database, so that you don't have to mess with configuration files or saving setting otherwise:

```
from connectordb.logger import Logger
```

```
l = Logger("log.db")

l.data = {"hi": 56}

# prints the data dictionary
print l.data
```

**insert**(*streamname*, *value*)
Insert the datapoint into the logger for the given stream name. The logger caches the datapoint and eventually synchronizes it with ConnectorDB

**insert_many**(*data_dict*)
Inserts data into the cache, if the data is a dict of the form {streamname: [{"t": timestamp,"d":data,...]}

**lastsynctime**
The timestamp of the most recent successful synchronization with the server

**name**
Gets the name of the currently logged in device

**ping**()
Attempts to ping the currently connected ConnectorDB database. Returns an error if it fails to connect

**serverurl**
The URL of the ConnectorDB server that Logger is using. By default this is connectordb.com, but can be set with this property. Note that the property will only take into effect before runtime

**start**()
Start the logger background synchronization service. This allows you to not need to worry about syncing with ConnectorDB - you just insert into the Logger, and the Logger will by synced every syncperiod.

**stop**()
Stops the background synchronization thread

**sync**()
Attempt to sync with the ConnectorDB server

**syncperiod**
Syncperiod is the time in seconds between attempting to synchronize with ConnectorDB. The Logger will gather all data in its sqlite database between sync periods, and every syncperiod seconds, it will attempt to connect to write the data to ConnectorDB.

CHAPTER 4

# Queries

ConnectorDB includes special queries optimized for generating datasets from the time series data associated with streams. There are 2 main query types:

**merge** Puts together multiple streams into one combined stream

**dataset** Takes multiple streams and uses their time series to generate tabular data ready for use in plotting and ML applications.

## 4.1 Merge

**class** connectordb.query.merge.**Merge**(*cdb*)

Bases: object

Merge represents a query which allows to merge multiple streams into one when reading, with all the streams merged together by increasing timestamp. The merge query is used as a constructor-type object:

```
m = Merge(cdb)
m.addStream("mystream1",t1=time.time()-10)
m.addStream("mystream2",t1=time.time()-10)
result = m.run()
```

**addStream**(*stream*, *t1=None*, *t2=None*, *limit=None*, *i1=None*, *i2=None*, *transform=None*)

Adds the given stream to the query construction. The function supports both stream names and Stream objects.

**run**()

Runs the merge query, and returns the result

connectordb.query.merge.**get_stream**(*cdb*, *stream*)

## 4.2 Dataset

**class** connectordb.query.dataset.**Dataset**(*cdb*, *x=None*, *t1=None*, *t2=None*, *dt=None*, *limit=None*, *i1=None*, *i2=None*, *transform=None*, *posttransform=None*)

Bases: object

ConnectorDB is capable of taking several separate unrelated streams, and based upon the chosen interpolation method, putting them all together to generate tabular data centered about either another stream's datapoints, or based upon time intervals.

The underlying issue that Datasets solve is that in ConnectorDB, streams are inherently unrelated. In most data stores, such as standard relational (SQL) databases, and even excel spreadsheets, data is in tabular form. That is, if we have measurements of temperature in our house and our mood, we have a table:

| Mood Rating | Room Temperature (F) |
|---|---|
| 7 | 73 |
| 3 | 84 |
| 5 | 79 |

The benefit of having such a table is that it is easy to perform data analysis. You know which temperature value corresponds to which mood rating. The downside of having such tables is that Mood Rating and Room Temperature must be directly related - a temperature measurement must be made each time a mood rating is given. ConnectorDB has no such restrictions. Mood Rating and Room Temperature can be entirely separate sensors, which update data at their own rate. In ConnectorDB, each stream can be inserted with any timestamp, and without regard for any other streams.

This separation of Streams makes data require some preprocessing and interpolation before it can be used for analysis. This is the purpose of the Dataset query. ConnectorDB can put several streams together based upon chosen transforms and interpolators, returning a tabular structure which can readily be used for ML and statistical applications.

There are two types of dataset queries

**T-Dataset** T-Dataset: A dataset query which is generated based upon a time range. That is, you choose a time range and a time difference between elements of the dataset, and that is used to generate your dataset.

| Timestamp | Room Temperature (F) |
|---|---|
| 1pm | 73 |
| 4pm | 84 |
| 8pm | 79 |

If I were to generate a T-dataset from 12pm to 8pm with dt=2 hours, using the interpolator "closest", I would get the following result:

| Timestamp | Room Temperature (F) |
|---|---|
| 12pm | 73 |
| 2pm | 73 |
| 4pm | 84 |
| 6pm | 84 |
| 8pm | 79 |

The "closest" interpolator happens to return the datapoint closest to the given timestamp. There are many interpolators to choose from (described later).

Hint: T-Datasets can be useful for plotting data (such as daily or weekly averages).

**X-Dataset** X-datasets allow to generate datasets based not on evenly spaced timestamps, but based upon a stream's values

Suppose you have the following data:

| Timestamp | Mood Rating | | Timestamp | Room Temperature (F) |
|---|---|---|---|---|
| 1pm | 7 | | 2pm | 73 |
| 4pm | 3 | | 5pm | 84 |
| 11pm | 5 | | 8pm | 81 |
| | | | 11pm | 79 |

An X-dataset with X=Mood Rating, and the interpolator "closest" on Room Temperature would generate:

| Mood Rating | Room Temperature (F) |
|---|---|
| 7 | 73 |
| 3 | 84 |
| 5 | 79 |

**Interpolators** Interpolators are special functions which specify how exactly the data is supposed to be combined into a dataset. There are several interpolators, such as "before", "after", "closest" which work on any type of datapoint, and there are more advanced interpolators which require a certain datatype such as the "sum" or "average" interpolator (which require numerical type).

In order to get detailed documentation on the exact interpolators that the version of ConnectorDB you are are connected to supports, you can do the following:

```
cdb = connectordb.ConnectorDB(apikey)
info = cdb.info()
# Prints out all the supported interpolators and their associated␣
↪documentation
print info["interpolators"]
```

**__init__**(*cdb*, *x=None*, *t1=None*, *t2=None*, *dt=None*, *limit=None*, *i1=None*, *i2=None*, *transform=None*, *posttransform=None*)

In order to begin dataset generation, you need to specify the reference time range or stream.

**To generate a T-dataset::** d = Dataset(cdb, t1=start, t2=end, dt=tchange)

**To generate an X-dataset::** d = Dataset(cdb,"mystream", i1=start, i2=end)

Note that everywhere you insert a stream name, you are also free to insert Stream objects or even Merge queries. The Dataset query in ConnectorDB supports merges natively for each field.

The only "special" field in this query is the "posttransform". This is a special transform to run on the entire row of data after the all of the interpolations complete.

**__module__ = 'connectordb.query.dataset'**

**addStream**(*stream*, *interpolator='closest'*, *t1=None*, *t2=None*, *dt=None*, *limit=None*, *i1=None*, *i2=None*, *transform=None*, *colname=None*)

Adds the given stream to the query construction. Additionally, you can choose the interpolator to use for this stream, as well as a special name for the column in the returned dataset. If no column name is given, the full stream path will be used.

addStream also supports Merge queries. You can insert a merge query instead of a stream, but be sure to name the column:

```
d = Dataset(cdb, t1=time.time()-1000,t2=time.time(),dt=10.)
d.addStream("temperature","average")
d.addStream("steps","sum")

m = Merge(cdb)
m.addStream("mystream")
m.addStream("mystream2")
d.addStream(m,colname="mycolumn")

result = d.run()
```

**run**()
Runs the dataset query, and returns the result

connectordb.query.dataset.**param_stream**(*cdb*, *params*, *stream*)

The official python client for ConnectorDB.

To install the client:

```
pip install connectordb
```

Another optional requirement is python-apsw.

The client enables quick usage of the database for IoT stuff and data analysis:

```
import time
import connectordb

# Log in as a device
cdb = connectordb.ConnectorDB("apikey")

# Get the temperature stream
temp = cdb["temperature"]

if not temp.exists():
    temp.create({"type": "number"}) # connectordb streams use JSON schemas

while True:
    time.sleep(1)
    t = get_temperature()
    temp.insert(t)
```

The client also allows anonymous access of database values if the database is configured to allow public access:

```
import connectordb
cdb = connectordb.ConnectorDB()

usr = cdb("myuser")
```

CHAPTER 5

# Indices and Tables

- genindex
- modindex
- search

# Python Module Index

## C

# Index

## Symbols



## A



## C



## D



## E

## G



## I



## L



## M



## N



## P



## Q



## R



## S



## U