Coinbase Pro Asyncronous Websocket Client Documentation

Release 1.0.7

Tony Podlaski

Aug 19, 2018

Contents

An Asyncronous Python WebSocket Client for Coinbase Pro

CHAPTER 1

CoPrA Features

CoPrA (**Co**inbase **Pr**o **A**sync) is an asyncronous websocket client written in Python for use with the Coinbase Pro digital currency trading platform. To learn about Coinbase Pro's WebSocket service including the available channels and the data they provide, please see Coinbase Pro's WebSocket API documentation.

- Coinbase Pro WebSocket client class with callback hooks for managing every phase of a WebSocket session
- supports user authentication
- compatible with Python 3.5 or greater
- built on **Autobahn|Python**, the open-source (MIT) real-time framework for web, mobile & the Internet of Things.
- utilizes Python's asyncio concurrency framework
- open source (MIT license)

CHAPTER 2

Examples

While `copra.websocket.Client` is meant to be overridden, it can still be used 'as is' to test the module through the command line.

```
# example.py

import asyncio

from copra.websocket import Channel, Client

loop = asyncio.get_event_loop()

ws = Client(loop, Channel('heartbeat', 'BTC-USD'))

try:
    loop.run_forever()
except KeyboardInterrupt:
    loop.run_until_complete(ws.close())
    loop.close()
```

Running the above:

```
$ python3 example.py
{'type': 'subscriptions', 'channels': [{'name': 'heartbeat', 'product_ids': ['BTC-USD
↪']}]}
{'type': 'heartbeat', 'last_trade_id': 45950713, 'product_id': 'BTC-USD', 'sequence':␣
↪6254273323, 'time': '2018-07-05T22:36:30.823000Z'}
{'type': 'heartbeat', 'last_trade_id': 45950714, 'product_id': 'BTC-USD', 'sequence':␣
↪6254273420, 'time': '2018-07-05T22:36:31.823000Z'}
{'type': 'heartbeat', 'last_trade_id': 45950715, 'product_id': 'BTC-USD', 'sequence':␣
↪6254273528, 'time': '2018-07-05T22:36:32.823000Z'}
{'type': 'heartbeat', 'last_trade_id': 45950715, 'product_id': 'BTC-USD', 'sequence':␣
↪6254273641, 'time': '2018-07-05T22:36:33.823000Z'}
{'type': 'heartbeat', 'last_trade_id': 45950715, 'product_id': 'BTC-USD', 'sequence':␣
↪6254273758, 'time': '2018-07-05T22:36:34.823000Z'}
```

(continues on next page)

(continued from previous page)

```
{'type': 'heartbeat', 'last_trade_id': 45950720, 'product_id': 'BTC-USD', 'sequence':␣
↪6254273910, 'time': '2018-07-05T22:36:35.824000Z'}
.
.
.
```

CoPrA supports authentication allowing you to receive only messages specific to your user account. **NOTE:** This requires registering an API key at Coinbase Pro.

```
# example2.py

import asyncio

from copra.websocket import Channel, Client

KEY = YOUR_KEY
SECRET = YOUR_SECRET
PASSPHRASE = YOUR_PASSPHRASE

loop = asyncio.get_event_loop()

channel = Channel('user', 'LTC-USD')

ws = Client(loop, channel, auth=True, key=KEY, secret=SECRET, passphrase=PASSPHRASE)

try:
    loop.run_forever()
except KeyboardInterrupt:
    loop.run_until_complete(ws.close())
    loop.close()
```

Running the above:

```
$ python3 example2.py
{'type': 'subscriptions', 'channels': [{'name': 'user', 'product_ids': ['LTC-USD']}]}
{'type': 'received', 'order_id': '42d2677d-0d37-435f-a776-e9e7f81ff22b', 'order_type
↪': 'limit', 'size': '50.00000000', 'price': '1.00000000', 'side': 'buy', 'client_oid
↪': '00098b59-4ac9-4ff8-ba16-bd2ef673f7b7', 'product_id': 'LTC-USD', 'sequence':␣
↪2311323871, 'user_id': '642394321fdf8343c4006432', 'profile_id': '039ff148-d490-
↪45f9-9aed-0d1f6412884', 'time': '2018-07-07T17:33:29.755000Z'}
{'type': 'open', 'side': 'buy', 'price': '1.00000000', 'order_id': '42d2677d-0d37-
↪435f-a776-e9e7f81ff22b', 'remaining_size': '50.00000000', 'product_id': 'LTC-USD',
↪'sequence': 2311323872, 'user_id': '642394321fdf8343c4006432', 'profile_id':
↪'039ff148-d490-45f9-9aed-0d1f6412884', 'time': '2018-07-07T17:33:29.755000Z'}
.
.
.
```

More detailed examples can be found on the Examples page.

CHAPTER 3

Installation

3.1 Stable release

To install Coinbase Pro Asyncronous Websocket Client, run this command in your terminal:

```
$ pip install copra
```

This is the preferred method to install Coinbase Pro Asyncronous Websocket Client, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

3.2 From sources

The sources for Coinbase Pro Asyncronous Websocket Client can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/tpodlaski/copra
```

Or download the tarball:

```
$ curl  -OL https://github.com/tpodlaski/copra/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

CHAPTER 4

Usage

Warning: Any references made below to specific aspects of the Coinbase Pro API such as the channels and the data they provide may be out of date. Please visit Coinbase Pro's WebSocket API documentation for the authorative and up to date information.

4.1 Introduction

The CoPrA API provides two classes for creating a WebSocket client for the Coinbase Pro platform. The first, `copra.websocket.Channel`, is intended to be used "as is." The second, `copra.websocket.Client`, is the actual client class. It provides multiple callback methods to manage every stage of the client's life cycle.

4.2 Channel

At the heart of every WebSocket connection is the concept of a channel. A channel provides a specific type of data about one or more currency pairs. `copra.websocket.Channel` has two attributes: it's name `name` and the product pairs the channel is observing, `product_ids`.

The current channels provided by the Coinbase Pro API are:

- **heartbeart** - heartbeat messages are generated once a second. They include sequence numbers and last trade IDs that can be used to verify no messages were missed.
- **ticker** - ticker messages are sent every time a match happens providing real-time price updates.
- **level2** - level2 messages provide a high level view of the order book. After the initial snapshot of the order book is delivered, messages are sent every time the volume at specific price tier on the buy or sell side changes.
- **full** - the full channel provides real-time updates on orders and trades. There are messages for every stage of an orders life cycle including: received, open, match, done, change, and activate.
- **user** - the user channel provides the same information as the full channel but only for the authenticated user. As such you will need to be authenticated to susbsribe. This requires a Coinbase Pro API key.

- **matches** - this channel consists only of the match messages from the full channel.

The Coinbase Pro exchange currently hosts four digital currencies:

- **BTC** - Bitcoin
- **BCH** - Bitcoin Cash
- **ETH** - Etherium
- **LTC** - Litecoin Cash

And allows 3 fiat currencies for trading:

- **USD** - US Dollar
- **EUR** - Euro
- **GBP** - Great British Pounds (Sterling)

Not every combination of currencies is available for trading, however. The current currency pairs (or products) avaialable for trade are:

- **BTC-USD**
- **BTC-EUR**
- **BTC-GBP**
- **ETH-USD**
- **ETH-EUR**
- **ETH-BTC**
- **LTC-USD**
- **LTC-EUR**
- **LTC-BTC**
- **BCH-USD**
- **BCH-EUR**
- **BCH-BTC**

These are the product IDs referenced below.

Before connecting to the Coinbase Pro Websocket server, you will need to create one or more channels to subscribe to.

First, import the `Channel` class:

```
from copra.websocket import Channel
```

The channel is then initialized with its name and one or more product IDs. The heartbeat channel for the Bitcoin/US dollar pair would be initialized:

```
channel = Channel('heartbeat', 'BTC-USD')
```

A channel that recieves ticker information about the pairs Etherium/US dollar and Litecoin/Euro would be initialized:

```
channel = Channel('ticker', ['ETH-USD', 'LTC-EUR'])
```

As illustrated above, the product ID argument to the `Channel` constructor can be a single string or a list of strings.

To listen for messages about Bitcoin/US Dollar and Litecoin/Bitcoin orders for an authenticated user:

```
channel = Channel('user', ['BTC-USD', 'LTC-BTC'])
```

As noted above, this will require that the `Client` be authenticated. This is covered below.

4.3 Client

The `Client` class represents the Coinbase Pro WebSocket client. While it can be used "as is", most developers will want to subclass it in order to customize the behavior of its callback methods.

First it needs to be imported:

```
from copra.websocket import Client
```

For reference, the signature of the `Client` `__init__` method is:

```
def __init__(self, loop, channels, feed_url=FEED_URL,
             auth=False, key='', secret='', passphrase='',
             auto_connect=True, auto_reconnect=True,
             name='WebSocket Client')
```

Only two parameters are required to create a client: `loop` and `channels`.

`loop` is the Python asyncio loop that the client will run in. Somewhere in your code you will likely have something like:

```
import asyncio

loop = asyncio.get_event_loop()
```

`channels` is either a single `Channel` or a list of `Channels` the client should immediately subscribe to.

`feed_url` is the url of the Coinbase Pro Websocket server. The default is `copra.websocket.FEED_URL` which is wss://ws-feed.pro.coinbase.com:443.

If you want to test your code in Coinbase's "sandbox" development environment, you can set `feed_url` to `copra.websocket.SANDBOX_FEED_URL` which is wss://ws-feed-public.sandbox.pro.coinbase.com:443.

`auth` indicates whether or not the client will be authenticated. If True, you will need to also provide `key`, `secret`, and `passphrase`. These values are provided by Coinbase Pro when you register for an API key.

`auto_connect` determines whether or not to automatically add the client to the asyncio loop. If true, the client will be added to the loop when it (the client) is initialized. If the loop is already running, the WebSocket connection will open. If the loop is not yet running, the connection will be made as soon as the loop is started.

If `auto_connect` is False, you will need to explicitly call `client.add_as_task_to_loop()` when you are ready to add the client to the asyncio loop and open the WebSocket connection.

`auto_reconnect` determines the client's behavior is the connection is closed in any way other than by explicitly calling its `close` method. If True, the client will automatically try to reconnect and re-subscribe to the channels it subscribed to when the connection unexpectedly closed.

`name` is a simple string representing the name of the client. Setting this to something unique may be useful for logging purposes.

4.3.1 Callback Methods

The `Client` class provides four methods that are automatically called at different stages of the client's life cycle. The method that will be most useful for developers is `on_message()`.

on_open()

`on_open` is called as soon as the initial WebSocket opening handshake is complete. The connection is open, but the client is **not yet subscribed**.

If you override this method it is important that **you still call it** from your subclass' `on_open` method, since the parent method sends the initial subscription request to the WebSocket server. Somewhere in your `on_open` method you should have `super().on_open()`.

In addition to sending the subsciption request, this method also logs that the connection was opened.

on_message(message)

`on_message` is called everytime a message is received. `message` is a dict representing the message. Its content will depend on the type of message, the channels subscribed to, etc. Please read Coinbase Pro's WebSocket API documentation to learn about these message formats.

Note that with the exception of errors, every other message triggers this method including things like subscription confirmations. Your code should be prepared to handle unexpected messages.

This default method just prints the message received. If you override this method, there is no need to call the parent method from your subclass' method.

on_error(message, reason)

`on_error` is called when an error message is received from the WebSocket server. `message` a is string representing the error, and `reason` is a string that provides additional information about the cause of the error. Note that in many cases `reason` is blank.

The default implementation just logs the message and reason. If you override this method, your subclass only needs to call the parent's method if want to preserve this logging behavior.

on_close(was_clean, code, reason)

`on_close` is called whenever the connection between the client and server is closed. `was_clean` is a boolean indicating whether or not the connection was cleanly closed. `code`, an integer, and `reason`, a string, are sent by the end that initiated closing the connection.

If the client did not initiate this closure and `client.auto_reconnect` is set to True, the client will attempt to reconnect to the server and resubscribe to the channels it was subscribed to when the connection was closed. This method also logs the closure.

If your subclass overrides this method, it is important that the subclass method calls the parent method if you want to preserve the auto reconnect functionality. This can be done by including `super().on_close(was_clean, code, reason)` in your subclass method.

4.3.2 Other Methods

close()

`close` is called to close the connection to the WebSocket server. Note that if you call this method, the client will not attempt to auto reconnect regardless of what the value of `client.auto_reconnect` is.

subscribe(channels)

subscribe is called to susbcribe to additional channels. `channels` is either a single Channel or a list of Channels.

The original channels to be subscribed to are defined during the client's initialization. `subscribe` can be used to add channels whether the client has been added to asyncio loop yet or not. If the loop isn't yet running, the client will subscribe to all of its channels when it is. If the loop is already running, the subcription will be appended with new channels, and incoming data will be immediately received.

unsubscribe(channels)

`unsubscribe` is called to unsubscribe from channels. `channels` is either a single Channel or a list of Channels.

Like `subscribe`, `unsubscribe` can be called regardless of whether or not the client has already been added to the asyncio loop. If the client has not yet been added, `unsubscribe` will remove those channels from the set of channels to be initially subscribed to. If the client has already been added to the loop, `unsubscribe` will remove those channels from the subscription, and data flow from them will stop immediately.

CHAPTER 5

Examples

5.1 Ticker

The following code, saved as `ticker.py`, when run from the command line prints a running ticker for the product ID supplied as an argument to the script.

```
#!/usr/bin/env python3

import asyncio
from datetime import datetime
import sys

from copra.websocket import Channel, Client

class Tick:

    def __init__(self, tick_dict):
        self.product_id = tick_dict['product_id']
        self.best_bid = float(tick_dict['best_bid'])
        self.best_ask = float(tick_dict['best_ask'])
        self.price = float(tick_dict['price'])
        self.side = tick_dict['side']
        self.size = float(tick_dict['last_size'])
        self.time = datetime.strptime(tick_dict['time'], '%Y-%m-%dT%H:%M:%S.%fZ')

    @property
    def spread(self):
        return self.best_ask - self.best_bid

    def __repr__(self):

        rep  = "{}\t\t\t\t   {}\n".format(self.product_id, self.time)
        rep += "================================================================\n"
        rep += "   Price: ${:.2f}\t    Size: {:.8f}\t  Side: {: >5}\n".format(self.
→price, self.size, self.side)
```

(continues on next page)

(continued from previous page)

```
        rep += "Best ask: ${:.2f}\tBest bid: ${:.2f}\tSpread: ${:.2f}\n".format(self.
↪best_ask, self.best_bid, self.spread)
        rep += "=============================================================\n"
        return rep


class Ticker(Client):

    def on_message(self, message):
        if message['type'] == 'ticker' and 'time' in message:
            tick = Tick(message)
            print(tick, "\n\n")

product_id = sys.argv[1]

loop = asyncio.get_event_loop()

channel = Channel('ticker', product_id)

ticker = Ticker(loop, channel)

try:
    loop.run_forever()
except KeyboardInterrupt:
    loop.run_until_complete(ticker.close())
    loop.close()
```

Streaming a ticker for LTC-USD:

```
$ ./ticker.py LTC-USD

LTC-USD                              2018-07-12 21:40:38.501000
=============================================================
   Price: $75.73            Size: 0.22134981      Side:   buy
Best ask: $75.73        Best bid: $75.67        Spread: $0.06
=============================================================



LTC-USD                              2018-07-12 21:40:38.501000
=============================================================
   Price: $75.74            Size: 0.29362708      Side:   buy
Best ask: $75.74        Best bid: $75.67        Spread: $0.07
=============================================================



LTC-USD                              2018-07-12 21:40:41.202000
=============================================================
   Price: $75.68            Size: 0.19211000      Side:  sell
Best ask: $75.74        Best bid: $75.68        Spread: $0.06
=============================================================



LTC-USD                              2018-07-12 21:41:09.452000
=============================================================
```

(continues on next page)

(continued from previous page)

```
   Price: $75.71            Size: 0.63097536      Side:   buy
Best ask: $75.71        Best bid: $75.68        Spread: $0.03
============================================================

^C
$
```

CHAPTER 6

Public API Reference

The following is an API reference of CoPrA generated from Python source code and docstrings.

Warning: This is a *complete* reference of the *public* API of CoPrA. User code and applications should only rely on the public API, since internal APIs can (and will) change without any guarantees. Anything *not* listed here is considered a private API.

6.1 Module `copra.websocket`

Asynchronous WebSocket client for the Coinbase Pro platform.

class copra.websocket.**Channel**(*name*, *product_ids*)

A WebSocket channel.

A Channel object encapsulates the Coinbase Pro WebSocket channel name *and* one or more Coinbase Pro product ids.

To read about Coinbase Pro channels and the data they return, visit: https://docs.gdax.com/#channels

Variables

- **name** (*str*) – The name of the WebSocket channel.
- **product_ids** (*set of str*) – Product ids for the channel.

__init__(*name*, *product_ids*)

Parameters

- **name** (*str*) – The name of the WebSocket channel. Possible values are heatbeat, ticker, level2, full, matches, or user
- **product_ids** (*str or list of str*) – A single product id (eg., 'BTC-USD') or list of product ids (eg., ['BTC-USD', 'ETH-EUR', 'LTC-BTC'])

Raises **ValueError** – If name not valid or product ids is empty.

class copra.websocket.**Client**(*loop*, *channels*, *feed_url='wss://ws-feed.pro.coinbase.com:443'*, *auth=False*, *key=''*, *secret=''*, *passphrase=''*, *auto_connect=True*, *auto_reconnect=True*, *name='WebSocket Client'*)

Asyncronous WebSocket client for Coinbase Pro.

__init__(*loop*, *channels*, *feed_url='wss://ws-feed.pro.coinbase.com:443'*, *auth=False*, *key=''*, *secret=''*, *passphrase=''*, *auto_connect=True*, *auto_reconnect=True*, *name='WebSocket Client'*)

Parameters

- **loop** (*asyncio loop*) – The asyncio loop that the client runs in.
- **channels** (*Channel or list of Channels*) – The channels to initially subscribe to.
- **feed_url** (*str*) – The url of the WebSocket server. The defualt is copra.WebSocket.FEED_URL (wss://ws-feed.gdax.com)
- **auth** (*bool*) – Whether or not the (entire) WebSocket session is authenticated. If True, you will need an API key from the Coinbase Pro website. The default is False.
- **key** (*str*) – The API key to use for authentication. Required if auth is True. The default is ''.
- **secret** (*str*) – The secret string for the API key used for authenticaiton. Required if auth is True. The default is ''.
- **passphrase** (*str*) – The passphrase for the API key used for authentication. Required if auth is True. The default is ''.
- **auto_connect** (*bool*) – If True, the Client will automatically add itself to its event loop (ie., open a connection if the loop is running or as soon as it starts). If False, add_as_task_to_loop() needs to be explicitly called to add the client to the loop. The default is True.
- **auto_reconnect** (*bool*) – If True, the Client will attemp to autom- matically reconnect and resubscribe if the connection is closed any way but by the Client explicitly itself. The default is True.
- **name** (*str*) – A name to identify this client in logging, etc.

Raises **ValueError** – If auth is True and key, secret, and passphrase are not provided.

add_as_task_to_loop()

Add the client to the asyncio loop.

Creates a coroutine for making a connection to the WebSocket server and adds it as a task to the asyncio loop.

close()

Close the WebSocket connection.

on_close(*was_clean*, *code*, *reason*)

Callback fired when the WebSocket connection has been closed.

(WebSocket closing handshake has been finished or the connection was closed uncleanly).

Parameters

- **was_clean** (*bool*) – True iff the WebSocket connection closed cleanly.
- **code** (*int or None*) – Close status code as sent by the WebSocket peer.
- **reason** (*str or None*) – Close reason as sent by the WebSocket peer.

on_error(*message*, *reason=''*)
Callback fired when an error message is received.

Parameters

- **message** (*str*) – A general description of the error.
- **reason** (*str*) – A more detailed description of the error.

on_message(*message*)
Callback fired when a complete WebSocket message was received.

You will likely want to override this method.

Parameters **message** (*dict*) – Dictionary representing the message.

on_open()
Callback fired on initial WebSocket opening handshake completion.

The WebSocket is open. This method sends the subscription message to the server.

subscribe(*channels*)
Subscribe to the given channels.

Parameters **channels** (*Channel or list of Channels*) – The channels to subscribe to.

unsubscribe(*channels*)
Unsubscribe from the given channels.

Parameters **channels** (*Channel or list of Channels*) – The channels to subscribe to.

CHAPTER 7

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

7.1 Types of Contributions

7.1.1 Report Bugs

Report bugs at https://github.com/tpodlaski/copra/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

7.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

7.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

7.1.4 Write Documentation

Coinbase Pro Asyncronous Websocket Client could always use more documentation, whether as part of the official Coinbase Pro Asyncronous Websocket Client docs, in docstrings, or even on the web in blog posts, articles, and such.

7.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/tpodlaski/copra/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

7.2 Get Started!

Ready to contribute? Here's how to set up *copra* for local development.

1. Fork the *copra* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/copra.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv copra
$ cd copra/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 copra tests
$ python setup.py test or py.test
$ tox
```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

7.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.7, 3.4, 3.5 and 3.6, and for PyPy. Check https://travis-ci.org/tpodlaski/copra/pull_requests and make sure that the tests pass for all supported Python versions.

7.4 Tips

To run a subset of tests:

```
$ python -m unittest tests.test_copra
```

7.5 Deploying

A reminder for the maintainers on how to deploy. Make sure all your changes are committed (including an entry in HISTORY.rst). Then run:

```
$ bumpversion patch # possible: major / minor / patch
$ git push
$ git push --tags
```

Travis will then deploy to PyPI if tests pass.

CHAPTER 8

Credits

8.1 Development Lead

- Tony Podlaski <tony@podlaski.com>

8.2 Contributors

None yet. Why not be the first?

CHAPTER 9

License

MIT License

Copyright (c) 2018, Tony Podlaski

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

CHAPTER 10

History

10.1 0.1.0 (2018-07-06)

- First release on PyPI.

10.2 0.2.0 (2018-07-07)

- Added Client authentication.

10.3 0.3.0 (2018-07-09)

- Added reconnect option to Client.

10.4 0.4.0 (2018-07-10)

- Added subscribe and unsubscribe methods to Client.

10.5 1.0.0 (2018-07-12)

- Added full documentation of the CoPrA API.

10.6 1.0.1 (2018-07-12)

- Fixed typos in the documentation.

10.7 1.0.2 (2018-07-12)

- Added Examples page to the documentation.

10.8 1.0.3 (2018-07-16)

- More documentation typos fixed.

10.9 1.0.4 - 1.0.5 (2018-07-17)

- Non-API changes.

10.10 1.0.6 (2018-08-19)

Updated Autobahn requirement to 18.8.1

Python Module Index

C

Index

Symbols

A

C

O

S

U