

Create a basic Bitmex connector!

Try to answer the questions of the assignment, then create a BitmexClient class and try to code the following methods, using the API documentation.

You will also have to instantiate the class in the main.py module: *bitmex = BitmexClient(..., ..., ...)*

<https://www.bitmex.com/app/apiOverview>

<https://www.bitmex.com/api/explorer/>

- `get_contracts()`
- `place_order()`
- `get_balances()`
- `get_order_status()`
- `cancel_order()`
- `get_historical_candles()`
- `_make_request()`
- `_generate_signature()`
- `_start_ws()`
- `_on_open(), _on_close(), _on_error, _on_message()`
- `_subscribe_channel()`

The `get_bid_ask()` method isn't required.

The answers to the 5 questions are available at the end of this document.

Do your best to code as many things as you can before checking the next video dedicated to the Bitmex connector, try to keep in mind the good coding practices, good luck!

Question 1

When accessing authenticated endpoints, how will the headers part of the request be different from the one we have seen on Binance? Use the Bitmex API documentation to answer this question.

Question 2

How can you modify the classes from the *models.py* module so that they can parse the info, *contract_info*, *candle_info*, *order_info* arguments both for Binance Futures and Bitmex?

Question 3

Can BTC quantities returned by the API (available balance, order value, margin etc...) be used without performing any calculation on them? If not, how would you convert the BTC quantities returned by the *get_balances()* method (for example) to a number usable in your program?

Question 4

What is the difference between subscribing to a Websocket channel on Binance and on Bitmex?

Question 5

When generating a signature for authenticated REST endpoints, the important thing is to know what the second argument of the *hmac.new()* method must be.

Find in the documentation the part that deals with the signature and try to adapt the *generate_signature()* method to the differences you notice compared with the Binance API.

Question 1

When accessing authenticated endpoints, how will the headers part of the request be different from the one we have seen on Binance? Use the Bitmex API documentation to answer this question.

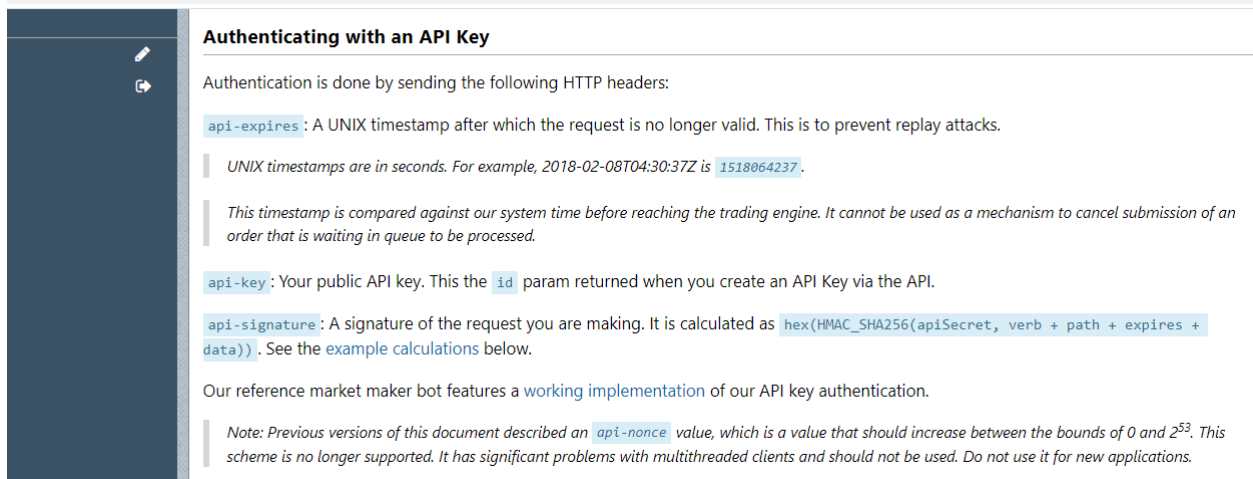
Answer

Binance requires only one key in the headers dictionary: `X-MBX-APIKEY`, while Bitmex requires 3 keys:

- `api-expires`
- `api-key`
- `api-signature`

The Bitmex headers won't be an instance variable (`self.headers = ...`) because the `api-expires` and `api-signature` fields will be different for every request, so we will create a `headers` variable inside the `make_request()` method.

bitmex.com/app/apiKeysUsage



The screenshot shows the Bitmex API documentation page for API key authentication. The page has a dark sidebar on the left with a search icon and a home icon. The main content area is titled 'Authenticating with an API Key' and contains the following text:

Authentication is done by sending the following HTTP headers:

api-expires: A UNIX timestamp after which the request is no longer valid. This is to prevent replay attacks.

UNIX timestamps are in seconds. For example, 2018-02-08T04:30:37Z is 1518064237.

This timestamp is compared against our system time before reaching the trading engine. It cannot be used as a mechanism to cancel submission of an order that is waiting in queue to be processed.

api-key: Your public API key. This is the `id` param returned when you create an API Key via the API.

api-signature: A signature of the request you are making. It is calculated as `hex(HMAC_SHA256(apiSecret, verb + path + expires + data))`. See the [example calculations](#) below.

Our reference market maker bot features a [working implementation](#) of our API key authentication.

Note: Previous versions of this document described an `api-nonce` value, which is a value that should increase between the bounds of 0 and 2^{53} . This scheme is no longer supported. It has significant problems with multithreaded clients and should not be used. Do not use it for new applications.

Question 2

How can you modify the classes from the `models.py` module so that they can parse the `info`, `contract_info`, `candle_info`, `order_info` arguments both for Binance Futures and Bitmex?

Answer

- Add another argument called *exchange*
- Add an *if ... else* statement in the `__init__` method.

Example with the **Balance** model:

```
1. class Balance:
2.     def __init__(self, info: typing.Dict, exchange: str):
3.         if exchange == "binance":
4.             self.initial_margin = float(info['initialMargin'])
5.             self.maintenance_margin = float(info['maintMargin'])
6.             self.margin_balance = float(info['marginBalance'])
7.             self.wallet_balance = float(info['walletBalance'])
8.             self.unrealized_pnl = float(info['unrealizedProfit'])
9.
10.        elif exchange == "bitmex":
11.            self.initial_margin = info['initMargin']
12.            self.maintenance_margin = info['maintMargin']
13.            self.margin_balance = info['marginBalance']
14.            self.wallet_balance = info['walletBalance']
15.            self.unrealized_pnl = info['unrealisedPnl']
```

Question 3

Can BTC quantities returned by the API (available balance, order value, margin etc...) be used without performing any calculation on them? If not, how would you convert the BTC quantities returned by the `get_balances()` method (for example) to a number usable in your program?

Answer

All BTC quantities retrieved with the API are returned in satoshis. Since 1 satoshi = 0.00000001 BTC, you must multiply any BTC number that you get by 0.00000001.

Ideally, you will even create a global variable (on top of the modules) that contains the BTC value of 1 satoshi. Name it `BITMEX_MULTIPLIER`, for example in the `models.py` module:

```
1. import typing
2.
3.
4. BITMEX_MULTIPLIER = 0.00000001
5.
6.
7. class Balance:
8.     def __init__(self, info, exchange):
9.         if exchange == "binance":
10.             self.initial_margin = float(info['initialMargin'])
11.             self.maintenance_margin = float(info['maintMargin'])
12.             self.margin_balance = float(info['marginBalance'])
13.             self.wallet_balance = float(info['walletBalance'])
14.             self.unrealized_pnl = float(info['unrealizedProfit'])
15.
16.         elif exchange == "bitmex":
17.             self.initial_margin = info['initMargin'] * BITMEX_MULTIPLIER
18.             self.maintenance_margin = info['maintMargin'] * BITMEX_MULTIPLIER
19.             self.margin_balance = info['marginBalance'] * BITMEX_MULTIPLIER
20.             self.wallet_balance = info['walletBalance'] * BITMEX_MULTIPLIER
21.             self.unrealized_pnl = info['unrealisedPnl'] * BITMEX_MULTIPLIER
```

Question 4

What is the difference between subscribing to a Websocket channel on Binance and on Bitmex?

Answer

On Bitmex, you need to subscribe to topics, that is to say you will be updated about every contract. While on Binance you can select a specific contract (for example ETHUSDT) and get updates only about this one.

So the argument of the `_subscribe_channel()` method will rather be the name of the topic rather than a Contract object.

```
def _subscribe_channel(self, topic: str)
```

```
self._subscribe_channel("instruments")
```

Question 5

When generating a signature for authenticated REST endpoints, the important thing is to know what the second argument of the `hmac.new()` method must be.

Find in the documentation the part that deals with the signature, and try to adapt the `generate_signature()` method to the differences you notice compared with the Binance API.

Answer

The second argument of the HMAC algorithm must be a string concatenation of the **method** (this is what they refer to as *verb* in the following screenshot), the endpoint (**without** the base URL and **with** the parameters, if there are parameters of course), and the current Unix timestamp + a few seconds.

For example:

```
verb = GET
path = /api/v1/order?symbol=XBTUSD&reverse=true
```

```
expires = 1611152956
```

The expires part must be converted to a string to be able to concatenate it with the other parts. We will add 5 seconds to the current timestamp, meaning that 5 seconds after now, the request won't be valid anymore.

```
1. expires = str(int(time.time()) + 5)
2. headers['api-expires'] = expires
```

The *data* part is only in case your request has a what's called a *body*, a *body* is a different way to pass parameters to your request, you put the parameters to the request *body* instead of adding them at the end of the URL.

You can do the string concatenation directly in the `generate_signature()` method or do it in the `make_request()` method and then pass the string as an argument to `generate_signature()`:

Example:

```
1. def _generate_signature(self, method: str, endpoint: str, expires: str, data: typing.Dict):
2.     if len(data) > 0:
3.         message = method + endpoint + "?" + urlencode(data) + expires
4.     else:
5.         message = method + endpoint + expires
6.     return hmac.new(self._secret_key.encode(), message.encode(), hashlib.sha256).hexdigest()
7.
8. headers['api-signature'] = self._generate_signature(method, endpoint, expires, data)
```

Or

```
1. def _generate_signature(self, message: str):
2.     return hmac.new(self._secret_key.encode(), message.encode(), hashlib.sha256).hexdigest()
3.
4. if len(data) > 0:
5.     message = method + endpoint + "?" + urlencode(data) + expires
6. else:
7.     message = method + endpoint + expires
8.
9. headers['api-signature'] = self._generate_signature(message)
```