Market Data Specification

CryptoStruct GmbH

Version 3.56.0

Table of Contents

Chapter 1. Instruments and Market Data

Handling instruments and market data is a crucial part of any trading system. All instruments in the *CryptoStruct Trading System* are identified by an ID that is unique across all exchanges. The Backend UI can be used to browse through all available instruments and to find the ID of a desired instrument. When using the SDK, this is all that is needed to start trading. Given the ID of an instrument, the *Strategy Server* will automatically connect to the required market data services to make market data available for trading.

That being said, it can be useful to know what is happening behind the scenes. Typical uses cases for this are:

- Custom applications need access to realtime market data
- Processing recorded market data with custom tools

1.1. Accessing Master Data

Using the *CryptoStruct* Backend UI to search for instruments is only suitable for manually searching for single instruments. When searching for instruments at a larger scale or as part of custom tools, the Backend API comes into play. This is not a comprehensive guide for all what the Backend API has to offer, but demonstrates the basic usage that is needed to fetch and search master data of instruments.

The following API calls refer to a `<backend-ip>`, which is the IP address or DNS name of your Backend installation. If you don't have a Backend installation (e.g. because you are using only CryptoStruct market data), you should contact *CryptoStruct* to obtain the correct IP address to use in this case.

Query all exchanges:

```
http://<backend-ip>/api/exchanges
```

Query all instruments:

```
http://<backend-ip>/api/instruments
```

Querying all instruments returns a lot of data, so in most cases its wise to filter for certain things:

By Exchange

```
http://<backend-ip>/api/instruments?exchange_id=1
```

Open instruments only

```
http://<backend-ip>/api/instruments?state=open
```

Instruments that were active (open) after a certain date

```
http://<backend-ip>/api/instruments?active_after=2023-06-30
```

Of course all these filters can be combined.

1.2. Subscribing Realtime Market Data

To subscribe market data no authentication is required, but the IP address of your server needs to be whitelisted for the corresponding markets. Please contact *CryptoStruct* if your server is not whitelisted yet.

The very first step in subscribing market data is finding out, which *CryptoStruct* server provides market data for the instruments of interest. To find that out, you can query your backend via HTTP:

```
http://<backend-ip>/api/v2/marketdata?instruments=<instrument-id-1>,<instrument-id-2>
```

This will return a response that looks like this:


```
{
  "status": 200,
  "version": "4.9.0",
  "result": "success",
  "timestamp": "2023-06-08T12:11:12.507601Z",
  "cached": true,
  "count": 2,
  "data": [
    {
      "instrument_id": 22,
      "type": "perpetual",
      "code": "XBTUSD",
      "ticksize": 0.5,
      "decimals": 1,
      "lot_size": 100,
      "lot_decimals": 0,
      "exchange_id": 1,
      "exchange_code": "bitmex",
      "exchange_name": "BitMEX",
      "hosts": [
        {
          "host": "54.216.182.250",
          "ip": "54.216.182.250",
          "ipv6": null,
          "location": "eu-west-1",
          "port": 11003
        },
        {
          "host": "54.195.111.215",
          "ip": "54.195.111.215",
          "ipv6": null,
          "location": "eu-west-1",
          "port": 11003
        }
      ]
    },
    {
      "instrument_id": 67824,
      ...
    }
  ],
  "source": "http://masterdata.cryptostruct.com",
  "uri": "/api/v2/marketdata?instruments=22,67824",
  "cache_key": "PROXY::/api/v2/marketdata?instruments=22%2C67824"
}
```


Besides some basic master data (like lot size and tick size), the response contains a list of host addresses. For fail-over and redundancy reasons, there are usually multiple hosts available. You can pick any host from the list and connect to it on the corresponding port via websocket:

```
ws://54.216.182.250:11003/api/v6
```

Once you are connected, you can login and subscribe to instruments, see Section 1.5 for details.

For the time being, every host returned for an instrument also provides market data for all other instruments of the same exchange. The host addresses itself can change from time to time, but this happens not that often. Depending on how robust your solution needs to be, this can be used to simplify a few things and get something up and running a lot faster. For example, if you are interested in instruments of a certain exchange like *Bitmex*, you can manually query the host address just once, put it into the configuration of your software and subscribe all instruments this way. Of course, if the host address gets updated, your subscriptions will fail and you need to manually update your configuration accordingly.

1.2.1. Multiple Protocols and Encodings

The example described in Section 1.2 is simplified in one important aspect: It connects directly via websocket to the endpoint `/api/v6`, which means it uses version 6 of the *CryptoStruct* market data protocol, encoded with JSON. This is the easiest way to subscribe market data, but it is not the only one:

- Version 6 is not the first version of the protocol and it is not going to be the last. Whenever the protocol gets updated, it will get a new endpoint, while the previous version stays available for a while. This makes transitioning from one version to the next as smooth as possible and doesn't require everything to be updated at once.
- JSON is not the only supported encoding. Depending on the protocol version, other encodings (like SBE) might also be available at dedicated endpoints.
- If the *Market Data Adapter* is hosted on the same machine as the client, the client can not only connect via websocket, but also via *Domain Socket* (also known as *Unix Socket*), which will by-pass the networking stack of the operating system and has therefore a better latency.

To get a list of available options, the host returned by the backend can be queried for these:

```
http://54.216.182.250:11003/api/info?all=true
```

This will return a response like this:


```
{
    "version": "2.13.3-SNAPSHOT",
    "process_id": "bitmex-master",
    "capabilities": {
        "depthTopic": {
            "eventIdType": "ORDERED",
            "exchangeTimestampType": "UNKNOWN",
            "exchangeTimestampPrecision": "MILLIS"
        },
        "topOfBookTopic": {
            "eventIdType": "ORDERED",
            "exchangeTimestampType": "UNKNOWN",
            "exchangeTimestampPrecision": "MILLIS"
        },
        "tradesTopic": {
            "eventIdType": "UNORDERED",
            "exchangeTimestampType": "MATCHING_ENGINE",
            "exchangeTimestampPrecision": "MICROS"
        },
        "fundingRateTopic": null,
        "markPriceTopic": null,
        "indexPriceTopic": null,
        "crossTopicBookEventId": true,
        "predictedFundingRate": false
    },
    "endpoints": [
        {
            "endpoint": "/",
            "protocol_version": "1",
            "encoding": "JSON"
        },
        {
            "endpoint": "/api/v6",
            "protocol_version": "6",
            "encoding": "JSON",
            "domain_socket_path": "/tmp/bitmex-master_v6_json.socket"
        },
        {
            "endpoint": "/api/v6/sbe",
            "protocol_version": "6",
            "encoding": "SBE",
            "domain_socket_path": "/tmp/bitmex-master_v6_sbe.socket"
        }
    ]
}
```


Here you can see what endpoints or domain sockets to use for certain protocols and encodings. The `domain_socket_path` is only available, if the request was made from the same machine as the market data service. Furthermore, the market data capabilities display the available data feeds and properties, as explained in detail in Section 1.3.

1.2.2. Unix Domain Sockets

If the client process and the *Market Data Adapter* are running on the same machine, the `/api/info` endpoint also returns a file path `domain_socket_path`. This file can be used to open a domain socket connection to the *Market Data Adapter* which can be

beneficial in terms of latency, since it bypasses the networking stack of the operating system.

Conceptually the protocol used via domain sockets is the same as the one used for websockets. In contrast to websocket connections, domain sockets are raw byte streams (just like TCP), so they don't have a builtin concept of a message. To identify individual messages in this stream of bytes, additional information must be added to each message.

To mimic the behaviour of websocket connections, this additional information must not only contain the length of the message, but also a flag that indicates whether the message is a text message or a binary message.

In this case, each message must be prefixed with 32bit integer. The most significant bit of this integer indicates the message type (0=binary, 1=text), while the remaining 31 bits represent the size of the following message data (so it doesn't include the four bytes of the prefix integer itself). This 32bit integer must use big-endian byte order.

The *Market Data Adapter* uses the exact same prefix when sending messages. So the very first four bytes received on a domain socket contain the type and size of the very first message. After all these bytes have been received, the next four bytes contain the type and size of the second message, and so on for all following messages.

1.2.3. SBE

Using json as encoding has the benefit of being human-readable and libraries for reading and writing are easily available for all programming languages. That being said, it is also quite verbose and is not the most efficient encoding to work with (especially because all decimals are encoded as strings and need to be parsed). For these reasons CryptoStruct products usually use SBE encoding internally (https://www.fixtrading.org/standards/sbe/) instead of json.

All messages are semantically exactly the same as with json and provide the same set of features, just encoded with SBE. The stand-alone release of the market data protocol specification contains not only the essential SBE schema definition, but also sample data for all messages encoded with json and SBE.

To use SBE encoding instead of json:

- connect to the respective endpoint or use the respective file for domain socket (see Section 1.2.1)
- Flag your messages as `binary`
  - websockets already have support for this
  - for domain sockets, please see Section 1.2.2

SBE generates encoders and decoders for all messages automatically (based on the given schema) and most data defined by the schema is self-explanatory and very easy to map to this documentation. The only exception to this are decimals (mostly prices and quantities), which require a special encoding.

Decimal Encoding

The CryptoStruct trading system needs to deal with numbers that have up to 18 decimal places, and the usual primitive number types provided by SBE and most programming languages do not represent such numbers reliable.

For this reason such numbers are represented as a combination of an unscaled value (an arbitrary large integer) and a scale (also an integer). With these, a number can be calculated as follows:

```
number = unscaled * (10 ^ -scale)
```

In SBE the integer for the unscaled value is stored as an array of bytes, so it can store as many bytes as needed with minimal waste. The unscaled value uses `two's complement` (https://en.wikipedia.org/wiki/Two%27s_complement) to represent negatives values and stores its bytes in big-endian byte order (most significant byte first). The scale is an 8bit integer. Example:

```
unscaled value: [4,1]
scale = 2

The two bytes of the unscaled value form together a 16bit integer with value 1025, with this we get:

1025 * (10 ^ -2) = 10.25
```

1.3. Market Data Capabilities

The available capabilities for the market data feed vary based on the exchange and the *Market Data Adapter* version. The basic structure of these capabilities is illustrated in the following JSON:


```
{
    "depthTopic": {
        "eventIdType": "ORDERED",
        "exchangeTimestampType": "UNKNOWN",
        "exchangeTimestampPrecision": "MILLIS"
    },
    "topOfBookTopic": {
        "eventIdType": "ORDERED",
        "exchangeTimestampType": "UNKNOWN",
        "exchangeTimestampPrecision": "MILLIS"
    },
    "tradesTopic": {
        "eventIdType": "UNORDERED",
        "exchangeTimestampType": "MATCHING_ENGINE",
        "exchangeTimestampPrecision": "MICROS"
    },
    "fundingRateTopic": null,
    "markPriceTopic": null,
    "indexPriceTopic": null,
    "crossTopicBookEventId": true,
    "predictedFundingRate": false
}
```


The `*Topic` fields are the topics of the market data feed. If one of these is not available, it will be `null`.

`eventIdType` is always one of:

- `ORDERED`: event id is an ever-increasing sequence number
- `UNORDERED`: event id is a hash

`exchangeTimestampType` is always one of:

- `NONE`: no exchange timestamp available
- `UNKNOWN`: there is an exchange timestamp, but its origin is not documented
- `MATCHING_ENGINE`: exchange timestamp is set in the matching engine
- `EXCHANGE_OUT`: exchange timestamp is set when an event gets published to the client

`exchangeTimestampPrecision` is null if `exchangeTimestampType` is `NONE`, otherwise it is one of:

- `MILLIS`: millisecond precision
- `MICROS`: microsecond precision
- `NANOS`: nanosecond precision

`crossTopicBookEventId` is `true`, if `depthTopic` and `topOfBookTopic` use ordered event ids and share the same sequence at the exchange, so the event id can be used to detect if the depth data or the top-of-book data is more recent.

`predictedFundingRate` is true, if the predicted funding rate is supported by the exchange.

1.4. Processing Market Data Files

Market data gets recorded by *CryptoStruct* per instrument and day for all markets. These files are compressed with an algorithm called *zstd* (https://github.com/facebook/zstd).

There are ready-to-use tools available for all platforms to decompress these files as a whole, but *zstd* is designed to be very fast when decompressing and there are libraries available for most programming languages to do the decompression on-the-fly while reading the file.

After decompression, the file is a simple text file, so it is very easy to process. In this file, each line contains a single event (like an order book update or a trades event). The only exception to this is the very first line, which contains master data for the instrument and all related underlyings.

See Section 1.5 for details about market data events.

1.5. Market Data Protocol Specification

Messages (requests, responses, events) are simple JSON arrays. The very first element of such an array is a number representing the message type. The remaining elements of the message depend on this type.

```
[msgType, ...]
```

1.5.1. Requests

Login

The login request is structured like this:

```
[13, organization, application_name, application_version, process_id]
```

- organization: string, name of your organization or company
- application_name: string, name or type of your application
- application_version: string, version of your application
- process_id: string, id to identify the instance of your application

Full request example:

```
[13, "ACME", "MyApp", "1.2.3", "MyApp_Tokyo"]
```

In response you will receive information about the service you are connected to:

```
[14, process_id, application_version, protocol, capabilities]
```

- process_id: string, instance id of the market data service
- application_version: string, service version
- protocol: string, version of the protocol
- capabilities: json object, contains information about the available market data topics, see (_[market_data_capabilities])

Full response example:

```
[14, "bitmex-master-a", "1.23.4", "5",
{"depthTopic":{"eventIdType":"ORDERED","exchangeTimestampType":"UNKNOWN","exchangeTimestampPrecision":"MILLIS"},"topOfBookTopic":null,"tradesTopic":null,"crossTopicBookEventId":true}]
```

There is no explicit logout. Once you are logged in, you need to reconnect to change your login data.

Subscriptions

For subscriptions, you need to login first. Requests for subscribing an instrument are structured as follows:

```
[requestType, instrument, options]
```

- requestType: integer, always 11
- instrument: long, instrument id
- options: optional json object

- configure which market data topics should be subscribed
- enable server-side coalescing of top-of-book events (when during bursts or high load multiple top-of-events queue up, the MDA will directly send only the latest one, skipping the outdated updates in between)

Options example with default settings:

```
{
 "depthTopic": true,
 "tradesTopic": true,
 "topOfBookTopic": true,
 "indexPriceTopic": true,
 "markPriceTopic": true,
 "fundingRateTopic": true,
 "topOfBookCoalescing": false
}
```

Full example:

```
[11,22,{"depthTopic":false}]
```

On succesful subscription, you will receive an initial snapshot of the order book, followed by all other initial data and events. If subscription fails, you will receive an error state event for that instrument. An instrument can not be subscribed multiple times.

Unubscriptions

Requests for unsubscribing an instrument are structured as follows:

```
[requestType, instrument]
```

- requestType: integer, always 12
- instrument: long, instrument id

Full example:

```
[12,22]
```

1.5.2. Events

The basic message structure for all events (only exception: instrument state) looks like this:

```
[msgType, instrument, prevEventId, eventId, adapterTimestamp, exchangeTimestamp, data]
```

- msgType: integer, type of this message, one of:
  - 0: Snapshot
  - 1: Book update
  - 2: Trades
  - 5: Instrument state
  - 6: top-of-book
  - 7: Mark price
  - 8: Index price
  - 9: Funding rate
- instrument: long, instrument id
- prevEventId: string, id of the previous event (potentially empty/null)
- eventId: string, id of this event
- adapterTimestamp: long, receive timestamp (ns since epoch) at MDA (not guaranteed to be increasing)
- exchangeTimestamp: long, timestamp (ns since epoch) reported by exchange, 0 if not available
- data: json, different for each message type

Snapshot

Contains the full order book.

Data structure:

```
[[side, price, quantity, ordercount], [side, price, quantity, ordercount], ...]
```

- side: integer, book side: 0 (BID) or 1 (ASK)
- price: string, price
- quantity: string, quantity
- ordercount: integer, currently not used, always 1

Full example:

```
[0, 123456, "12345678-1", "12345678-2", 1580143531008103451, 1580143530031129024, [[0, "7098.25", "10", 1], ...[1, "7099.25","20", 1]]]
```

Book update

Contains level-based updates to order book.

Data structure:

```
[[side, price, quantity, ordercount], [side, price, quantity, ordercount], ...]
```

- side: integer, book side: 0 (BID) or 1 (ASK)
- price: string, price
- quantity: string, quantity
- ordercount: integer, currently not used, always 1

Full example:

```
[1, 123456, "12345678-1", "12345678-2", 1580143531008103451,1580143530031129024, [[0, "7098.25", "10", 1], ...[1, "7099.25","20", 1]]]
```

Trades

Contains a list of trades/fills that happended.

Data structure:

```
[[side, price, quantity,  tradeId, exchangeTradeTimestamp], [side, price, quantity, tradeId, exchangeTradeTimestamp], ...]
```

- side: integer, aggressive side: 0 (BID) or 1 (ASK)
- price: string, trade price
- quantity: string, trade quantity
- tradeId: string, exchange trade id (potentially empty/null)
- exchangeTradeTimestamp: integer, timestamp (ns since epoch) of trade, can be different from exchangeTimestamp of message itself

Full example:

```
[2, 123456, "12345678-2", "12345679-3", 1580143531008103451, 1580143530031129024, [[0, "7098.25", "10", "6as78d678asdf78as789fas",
11580143530031129024]]]
```

Top-of-Book

Contains an array with up to two levels (the top level per side). If one side of the book is empty, the array will have only one

element (for the non-empty side) and will have no elements at all for empty books. The order of bid and ask side is undefined.

Data structure:

```
[[side, price, quantity, ordercount], [side, price, quantity, ordercount]]
```

- side: integer, book side: 0 (BID) or 1 (ASK)
- price: string, price
- quantity: string, quantity
- ordercount: integer, currently not used, always 1

Full example:

```
[6, 123456, "12345678-1", "12345678-2", 1580143531008103451, 1580143530031129024, [[0, "7098.25", "10", 1], [1, "7099.25","20", 1]]]
```

Mark price

Mark price for instrument.

Data structure:

```
price
```

- price: string, mark price

Full example:

```
[7, 123456, "12345678-1", "12345678-2", 1580143531008103451, 1580143530031129024, "123.456"]
```

Index price

Index price for instrument.

Data structure:

```
price
```

- price: string, index price

Full example:

```
[8, 123456, "12345678-1", "12345678-2", 1580143531008103451, 1580143530031129024, "456.789"]
```

Funding rate

Current funding rate and time of next funding for instrument.

Data structure:

```
[fundingRate, nextFunding, predictedFundingRate]
```

- fundingRate: string, funding rate
- nextFunding: long, timestamp (ns since epoch) of next funding
- predictedFundingRate: string, predicted next funding rate

Full example:

```
[9, 123456, "12345678-1", "12345678-2", 1580143531008103451, 1580143530031129024, ["0.123", 1580163530031129024, "0.123"]]
```

Instrument state

The instrument state changes to ERROR when there is an issue (e.g. lost connection to exchange). An ERROR state is also emitted when subscribing an instrument fails (e.g. instrument does not exist). The instrument state message is different from other messages because it has no event ids and no exchange timestamp. Full structure:

```
[msgType, instrument, adapterTimestamp, state, message]
```

- msgType: integer, always 5
- instrument: long, instrument id
- adapterTimestamp: long, system timestamp (ns since epoch) at state event occurrence
- state: string, market data state of the instrument, "ERROR" or "READY"
- message: string, error details (potentially empty)

Full example:

```
[5, 123456, 1580143531008103451, "ERROR", "connection lost"]
```