Market Stream over WebSocket

DRAFT

Version: 0.7

Hello World in HTML5/JavaScript

Let's start with a simple client based on HTML/JavaScript, running inside the browser.

This is a very basic client, but you should be able to execute all the following examples:

Hello World

```
<html lang="en">
<head>
    <title>SIX Market Data Stream - WebSocket</title>
</head>
<body style="margin: 35px">

<h1>SIX Market Data Streaming - WebSocket</h1>

<form>
    <label for="inputElement"></label>
    <textarea id="inputElement" placeholder="type your query..." style="width:1400px;font-size: medium" rows="
16">
query HelloWorld {
  snapshot(scheme: TICKER_BC, ids: ["ABBN_4"]) {
    type
    requestedId
    requestedScheme
    orderBook { position side size value }
  }
}
    </textarea>
    <br/>
    <input onclick="sendMsg();" value="send" type="button">
</form>

<div id="outputElement" style="width:1400px; height: 300px; background: #eee; overflow:scroll; margin-top: 10px"
></div>

<script>
    const uri = "wss://web.api.six-group.com/api/findata/v1/websocket";
    const webSocket = new WebSocket(uri);
    const inputElement = document.getElementById("inputElement");
    const outputElement = document.getElementById("outputElement");
```

```
    webSocket.onmessage = function (event) {
        const json = event.data;
        outputElement.innerHTML += "<br> Server>: " + "<b>" + json + "</b>";
        outputElement.scrollTop = outputElement.scrollHeight;
    };

    webSocket.onopen = function () {
        console.log("connection opened");
    };

    webSocket.onclose = function (event) {
        console.log("connection closed: " + JSON.stringify(event));
    };

    webSocket.onerror = function (event) {
        console.log("error: " + JSON.stringify(event));
    };

    window.onclose = function () {
        console.log("connection closed");
        webSocket.close();
    }

    inputElement.addEventListener("keyup", function (event) {
        event.preventDefault();
        if (event.key === "Enter") {
            sendMsg();
        }
    });

    function sendMsg() {
        const json = JSON.stringify({query: inputElement.value});
        webSocket.send(json);
        outputElement.innerHTML += "<div style='color:green'>Client> " + json + "</div>";
        inputElement.value = "";
    }
</script>
</body>
</html>
```

For the complete reference about WebSocket please check the official documentation (WebSocket - Web APIs | MDN (mozilla.org)).

Other languages

If you are using another programming language, please check the official WebSocket library:

- Python websockets · PyPI
- Java WebSocket (Java SE 11 & JDK 11) (oracle.com)

Streaming

It is possible to stream data by using the **startStream** operation in GraphQL:

Hello World Response

```
subscription {
   startStream(scheme: TICKER_BC, ids: ["ABBN_4"]) {
    type
    requestedId
    requestedScheme
    high {
      value
      unixTimestamp
    }
    low {
      value
      unixTimestamp
    }
  }
}
```

if you run the request when the Swiss Market is open (Mon-Fri from 8-17:30 as per 22.11.2022), you will receive a single **START** messages followed by one or more **UPDATE** messages.

Hello World Response


```
{"data":{"startStream":{"type":"START","requestedId":"ABBN_4","requestedScheme":"TICKER_BC","high":{"value":
29.05,"unixTimestamp":1667981959.201569},"low":{"value":28.65,"unixTimestamp":1667980882.547938}}}}
{"data":{"startStream":{"type":"UPDATE","requestedId":"ABBN_4","requestedScheme":"TICKER_BC","high":{"value":
28.77,"unixTimestamp":1667993376.222199}}}}
{"data":{"startStream":{"type":"UPDATE","requestedId":"ABBN_4","requestedScheme":"TICKER_BC","high":{"value":
28.78,"unixTimestamp":1667993376.222199}}}}
...
```


Snapshot

Let's see in detail some basic **snapshot** queries. With snapshot is possible to request the status of the market as now, without any updates.

High/Low example

This query is just requesting a snap of the ABB instrument on the Swiss Market (1222171,4) and selecting the message type and echoing the request parameters:

Hello World Request

```
{
  snapshot(scheme: TICKER_BC, ids: ["ABBN_4"]) {
    type
    requestedId
    requestedScheme
    high {
      value
      unixTimestamp
    }
    low {
      value
      unixTimestamp
    }
  }
}
```

A single message with be sent back with all the requested data:

Hello World Response


```
{
  "data": {
    "snapshot": [
      {
        "type": "SNAPSHOT",
        "requestedId": "ABBN_4",
        "requestedScheme": "TICKER_BC",
        "bestBid": null,
        "bestAsk": null,
        "high": {
          "value": 29.05,
          "unixTimestamp": 1667981959.201569
        },
        "low": {
          "value": 28.65,
          "unixTimestamp": 1667980882.547938
        }
      }
    ]
  }
}
```


Order Book

Order book with depth = 10:

Order Book snapshot request

```
query {
  snapshot(scheme: VALOR_BC, ids: ["1222171_4"]) {
    type
    requestedId
    requestedScheme
    orderBook {
      position
      side
      size
      value
      unixTimestamp
    }
  }
}
```

Order Book snapshot response


```
{
  "data": {
    "snapshot": [
      {
        "type": "SNAPSHOT",
        "requestedId": "1222171_4",
        "requestedScheme": "VALOR_BC",
        "orderBook": [
          {
            "position": 1,
            "side": "ASK",
            "size": 7215,
            "value": 28.78,
            "unixTimestamp": 1667923233.552173
          },
          {
            "position": 2,
```

```
    "side": "ASK",
    "size": 10756,
    "value: 28.79,
    "unixTimestamp": 1667923206.864032
  },
  {
    "position": 3,
    "side": "ASK",
    "size": 7781,
    "value": 28.8,
    "unixTimestamp": 1667923233.552173
  },
  {
    "position": 4,
    "side": "ASK",
    "size": 16935,
    "value": 28.81,
    "unixTimestamp": 1667923206.847641
  },
  {
    "position": 5,
    "side": "ASK",
    "size": 9256,
    "value": 28.82,
    "unixTimestamp": 1667923239.939986
  },
  {
    "position": 6,
    "side": "ASK",
    "size": 6370,
    "value": 28.83,
    "unixTimestamp": 1667923234.428848
  },
  {
    "position": 7,
    "side": "ASK",
    "size": 2327,
    "value": 28.84,
    "unixTimestamp": 1667923126.710191
  },
  {
    "position": 8,
    "side": "ASK",
    "size": 5484,
    "value": 28.85,
    "unixTimestamp": 1667923241.962111
  },
  {
    "position": 9,
    "side": "ASK",
    "size": 3949,
    "value": 28.86,
    "unixTimestamp": 1667923241.96682
  },
  {
    "position": 10,
    "side": "ASK",
    "size": 3163,
    "value": 28.87,
    "unixTimestamp": 1667923200.473025
  },
  {
    "position": 1,
    "side": "BID",
    "size": 2188,
    "value": 28.77,
    "unixTimestamp": 1667923239.913224
  },
  {
    "position": 2,
    "side": "BID",
```

```
            "size": 4550,
            "value": 28.76,
            "unixTimestamp": 1667923207.679513
          },
          {
            "position": 3,
            "side": "BID",
            "size": 11556,
            "value": 28.75,
            "unixTimestamp": 1667923217.446753
          },
          {
            "position": 4,
            "side": "BID",
            "size": 9671,
            "value": 28.74,
            "unixTimestamp": 1667923204.5842
          },
          {
            "position": 5,
            "side": "BID",
            "size": 17968,
            "value": 28.73,
            "unixTimestamp": 1667923200.021515
          },
          {
            "position": 6,
            "side": "BID",
            "size": 7879,
            "value": 28.72,
            "unixTimestamp": 1667923200.021505
          },
          {
            "position": 7,
            "side": "BID",
            "size": 5658,
            "value": 28.71,
            "unixTimestamp": 1667923207.018346
          },
          {
            "position": 8,
            "side": "BID",
            "size": 2196,
            "value": 28.7,
            "unixTimestamp": 1667923200.02148
          },
          {
            "position": 9,
            "side": "BID",
            "size": 4296,
            "value": 28.69,
            "unixTimestamp": 1667923200.02147
          },
          {
            "position": 10,
            "side": "BID",
            "size": 4247,
            "value": 28.68,
            "unixTimestamp": 1667923217.605718
          }
        ]
      }
    ]
  }
}
```


Symbology

How to query with different symbols? By now the snapshot and startStream have two parameters.

scheme is a **ListingScheme** enum and it has the following values:

- VALOR_BC This is the primary key of listings in SIX.
- ISIN_BC
- SEDOL_BC
- CUSIP_BC
- TICKER_BC A ticker symbol is an abbreviation used to uniquely identify publicly traded shares of a particular stock on a particular stock market.
- FIGI_BC The Financial Instrument Global Identifier (FIGI), formerly Bloomberg Global Identifier (BBGID), is an open standard, unique identifier of financial instruments that can be assigned to instruments including common stock, options, derivatives, futures, corporate and government bonds, municipals, currencies, and mortgage products. FIGI here is the Global Share Class requires also the BC to uniquely identify a listing.

ids is a list of strings. The strings are validated to contains only meangiful data.

Examples

It is possible to reference the same listing in several way. For example:

scheme	ids	resulting listing
VALOR_BC	12222171_4	ABBN @ SwissMarket
ISIN_BC	CH0012221716_4	ABBN @ SwissMarket
SEDOL_BC	7108899_4	ABBN @ SwissMarket
TICKER_BC	ABBN_4	ABBN @ SwissMarket
FIGI_BC	BBG001SCX147_4	ABBN @ SwissMarket
...		
more		

List of BC codes for other markets:

BC	Market
104	Hong Kong Exchanges and Clearing Ltd
1143	Stock Exchange of Hong Kong-Odd Lot
106	Tokyo Stock Exchange
114	Bombay Stock Exchange Ltd. (Mumbai)
123	KRX Korea Exchange, Stock Market
214	Shenzhen Stock Exchange
215	Shanghai Stock Exchange
358	Korea Exchange (Kosdaq)
729	KRX Korea Exchange, After-Hour Single-Price Trading
731	KOSDAQ, After-Hour Single-Price Trading
1346	Shanghai Stock Exchange Level 1
67	Nasdaq

Error handling

In Symbology

Error during the resolution of symbol to the actual listing are delivered as message type = **ERROR.** For example, resolving the id "foobar" as TICKER_BC will produce an error:

Symbology request

```
{
  snapshot(scheme: TICKER_BC, ids: ["foobar"]) {
    type
    requestedId
    requestedScheme
  }
}
```

The following error is delivered:

Symbology response

```
{
  "data": {
    "snapshot": [
      {
        "type": "ERROR",
        "requestedId": "foobar",
        "requestedScheme": "TICKER_BC"
      }
    ]
  },
  "errors": [
    {
      "message": "id foobar not found",
      "extensions": {
        "classification": "DataFetchingException"
      }
    }
  ]
}
```

it is important to note that the error is delivered with same requestedId/requestedScheme pair in the input. Plus, there is a graphql error (the errors array) that provides more details.

Other errors

The API could deliver other errors without even creating a message. This happens if the request is wrong as GraphQL syntax or because it is invalid JSON.

Request with no selected fields

```
{
  snapshot(scheme: TICKER_BC, ids: ["%%%"]) {
               # no selected fields
  }
}
```

Response for no selected fields

```
{
  "errors": [
    {
      "message": "Invalid Syntax : offending token '}' at line 4 column 3",
      "locations": [
        {
          "line": 4,
          "column": 3
        }
      ],
      "extensions": {
        "classification": "InvalidSyntax"
      }
    }
  ]
}
```

Lookup

It is possible to select a small portion of reference data associated with the selected listing.

lookup is delivered only in **snapshot** and as START message in **startStream**.

Lookup request

```
query {
  snapshot(scheme: VALOR_BC, ids: ["1222171_4"]) {
    type
    requestedId
    requestedScheme
    lookup {
      listingName
      marketName
      listingCurrency
      listingStatus
    }
  }
}
```

Data:

Lookup response

```
{
  "data": {
    "snapshot": [
      {
        "type": "SNAPSHOT",
        "requestedId": "1222171_4",
        "requestedScheme": "VALOR_BC",
        "lookup": {
          "listingName": "ABB N",
          "marketName": "SIX SX",
          "listingCurrency": "CHF",
          "listingStatus": "LISTED"
        }
      }
    ]
  }
}
```

Fields

The following fields are available by now:

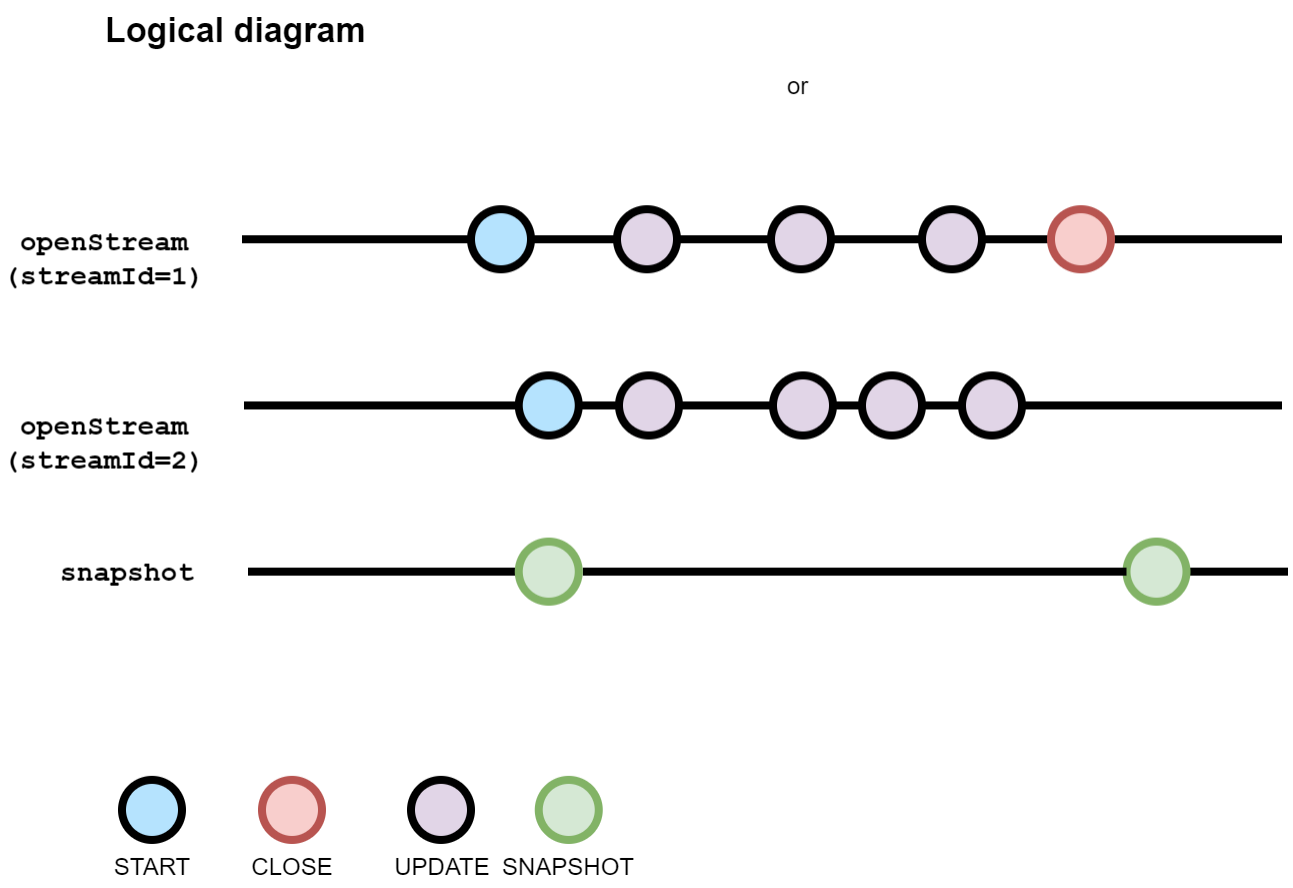
- open
- close
- high
- low
- bestBid
- bestAsk
- orderBook
- last (lastTrade)
- cumulatedVolume
- settlementPrice
- openInterest
- vwap

Message Types

Every message has a type attached to it. This is an enum with a well-defined behavior.

Possible values:

- **START**: this is delivered only by **startStream** subscription and it represents the "initial image" of the market stream. All data, including reference data, is delivered in this message
- **UPDATE**: this is delivered only by **startStream** subscription and it represent an update. Reference data is not delivered in this message.
- **CLOSE**: this is delivered only by **closeStream** mutation and represents the end message of the stream
- **SNAPSHOT**: delivered only be **snapshot** query
- **ERROR**: delivered in case is not possible to resolve the requested listing (e.g. wrong isin)



Multiple streams over the same connection

In order to stream different fields for different listings, it is possible to "tag" every message with a **streamId**. This is delivery as part of every message.

It enables advanced workflow, like the possibility to start / stop multiple stream within the same WebSocket connection identified by a unique **streamId**.

If the streamId is not unique within the current WebSocket connection, an error is raised.

Start Stream

```
subscription {
  startStream(streamId: "first stream" scheme: VALOR_BC, ids: ["1222171_4"]) {
    type
    requestedId
    requestedScheme
    orderBook {
      position
      side
      size
    }
  }
}

subscription {
  startStream(streamId: "second stream" scheme: TICKER_BC, ids: ["AAPL_67"]) {
    type
    requestedId
    requestedScheme
    orderBook {
      position
      side
      size
    }
  }
}
```

These 2 queries are running concurrenctly. Since we provided a unique **streamId**, it is possible to stop an individual stream:

Close a stream

```
mutation {
  closeStream(streamId: "first stream") {
    type
    requestedId
    requestedScheme
  }
}
```

NB: please note that, while **closeStream** supports the selection of data fields, some of them are not yet filled (e.g. open, close, orderBook).

Conflation

By default, all messages are sent without any conflation. However, it could be hard to keep up with the flow of messages during peak hours (especially on big markets).

For this reason, **startStream** provides two parameters:

- **conflationType** is an enum with several values
- **conflationInterval** is a string like "10s", "1500ms", "1000ns"

For example to get only the most up-do-date message in a window of 5 seconds, use the following query:

Conflation

```
subscription {
  startStream(
    conflationType: INTERVAL
    conflationInterval:"5s"
    streamId: "**** CONFLATED  ****"
    scheme: VALOR_BC
    ids: ["1222171_4"]
  ) {
    streamId
    type
    requestedId
    requestedScheme
    lookup {
      listingName
      marketName
      listingCurrency
    }
    orderBook {
      position
      side
      size
      value
    }
  }
}
```

Data Types

The GraphQL schema uses well specified and public available formats to represent data like timestamps, numbers, currencies and so on.

Data Type	Format	Examples
ZonedDateTime	Date time with timezone information represented as ISO 8601 with format **YYYY-MM-DD'T'HH:mm:ss+hh:mm**. See https://www.iso.org/iso-8601-date-and-time-format.html	"2021-01-01T07:45:1502:00" "2021-01-01T07:45:15+01:00"
Unix Timestamp	IEEE 754 Float with format **seconds.microseconds** since Epoch	1667969700.662919
Currency	ISO 4217 with 3 letters code See https://www.iso.org/iso-4217-currency-codes.html	"CHF" "USD" "EUR"
Country	ISO 3166 with 2 letters code See https://www.iso.org/iso-3166-country-codes.html	"CH" "US" "IT" "FR"

Advanced query capabilities

With GraphQL it is possible to select exactly what fields are needed and nothing else will be delivered.

It is possible to use all the features of GraphQL. Please refer to the specification or to your GraphQL client library to see what is possible.

- aliases
- fragments
- @skip/@include

- any combination of the one above

This is true for **snapshot** as well as for **openStream**.

Aliases

Since the API uses GrapHQL, it is possible to define aliases for objects and fields.

For example, let's imagine that in your application, the model is using **intradayHighPrice** and **intradayLowPrice** but, in the current schema, those fields are named **high** and **low** respectively.

In the query is possible to introduce such aliases to accommodate the application model:

QueryWithSkip

```
query QueryWithAliases {
  snapshot(scheme: TICKER_BC ids: ["ABBN_4"]) {
    type
    intradayHighPrice: high {
      value
      unixTimestamp
    }
    intradayLowPrice: low {
      value
      unixTimestamp
    }
  }
}
```

And let's say that the client application model wants to use **datetime** instead of **unixTimestamp :**

QueryWithSkip

```
query QueryWithAliases {
  snapshot(scheme: TICKER_BC ids: ["ABBN_4"]) {
    type
    intradayHighPrice: high {
      value
      datetime: unixTimestamp
    }
    intradayLowPrice: low {
      value
      datetime: unixTimestamp
    }
  }
}
```

Skip/Include

It is possible to define a query than skip/include fields based on a parameter.

In the following example, to be able to debug timestamp issues the more expensive ISO 8601 format is included conditionally:

QueryWithSkip

```
query QueryWithDebugging($debugging: Boolean! = false) {
  snapshot(scheme: TICKER_BC ids: ["ABBN_4"]) {
    type
    requestedId @skip(if: $debugging)
    requestedScheme @skip(if: $debugging)
    lookup {
      listingName
      marketName
      listingCurrency
      listingStatus
    }
    high {
      value
      unixTimestamp
      timestamp @include(if: $debugging)
    }
    low {
      value
      unixTimestamp
      timestamp @include(if: $debugging)
    }
  }
}
```

Implementation notes for GraphQL over WebSocket channel

The interface to interact with the service is **WebSocket**, using **Text messages** encoded in **JSON**.

The JSON messages follow the structure of the GraphQL specification for request and response.

To interact with the API encode in JSON an object with a string property named "query". Inside you can put the graphQL query. This is the easiest way to start using the API:

```
{
  "query": "{ snapshot(scheme: TICKER_BC, ids: \"ABBN_4\") { type requestedScheme requestedId }"
}
```

This is a good way to start, but in production the preferred way is to leverage bind **variables**.

The client application can define a query once and to reuse it multiple times with different parameters:

```
{
   "operationName": "MyQuery"
   "query":  "query MyQuery($scheme: ListingScheme!, $ids: [UserInput!]!) {\n  snapshot(scheme: $scheme, ids:
$ids) { type requestedScheme  requestedId }  } }"
   "variables": {
      scheme: "TICKER_BC",
      ids: "ABBN_4"
   }
}
```

Regardless of how the previous query has been sent to the WebSocket, the response will be always:

```
{
  "data": {
    "snapshot": [
      {
        "type": "SNAPSHOT",
        "requestedScheme": "TICKER_BC",
        "requestedId": "ABBN_4"
      }
    ]
  }
}
```