

# MarketFactory API handbook

---

MarketFactory engineering

*Tuesday 22<sup>nd</sup> November, 2016*  
MF-API-HANDBOOK-WIP



# Contents

This document describes the interfaces to the services provided by MarketFactory, via which MarketFactory customers may access multiple FX venues for market data and trading. Facilities for both market-making and regular trading (“taking”) are available.

Access is provided via the SBE protocol using an open-source implementation and alternatively an purely API-based approach (referred to as the “classic” API) using MarketFactory’s own messaging.

Chapter 1 describes how to use the SBE implementation to access market data and trading functionality via the MFClient class.

Chapters 2 and 3 describe the SBE MFSession and MFHandler interfaces which together enable the integration of the MarketFactory client into a customer’s trading system.

Chapters 4 and 5 describe how to access market data and order management functionality via the SBE MFClient interface.

Chapter 6 describes the considerations to be made when migrating from the classic library-based MFAPI to the protocol-based SBE interface.

Chapters 8 and 9 are devoted to regular “taking” trading and market-making via the classic API.

Chapter 10 contains a glossary and notes on venue-specific behaviors

Appendix A contains the mappings from XML to the message types. These mappings are machine-generated directly from the XML.

Appendix B details the mappings for each message field in the classic MFAPI messages to the fields as used in the SBE implementation.

# Contents

<b>1</b>	<b>Using the SBE MFClient interface</b>	<b>5</b>
1.1	session creation . . . . .	5
<b>2</b>	<b>The SBE session interface</b>	<b>9</b>
2.1	session creation and logon . . . . .	9
2.2	session destruction and logoff . . . . .	10
2.3	composition . . . . .	10
<b>3</b>	<b>The SBE Handler interface</b>	<b>13</b>
3.1	handler interfaces . . . . .	13
3.2	common callbacks . . . . .	13
<b>4</b>	<b>Market data via SBE</b>	<b>15</b>
4.1	market data session . . . . .	15
4.2	messages . . . . .	15
4.3	handler callbacks . . . . .	17
4.4	callbacks . . . . .	17
<b>5</b>	<b>Trading handlers in SBE</b>	<b>19</b>
5.1	subscriptions . . . . .	19
5.2	messages . . . . .	19
<b>6</b>	<b>STP booking interface</b>	<b>21</b>
6.1	introduction . . . . .	21
6.2	session features . . . . .	21
6.3	handler features . . . . .	22
<b>7</b>	<b>Migrating from the classic MFAPI</b>	<b>23</b>
<b>8</b>	<b>The Classic MFAPI</b>	<b>25</b>
8.1	API connection lifecycle . . . . .	25
8.1.1	Connecting an MFClient to the MF APIServer . . . . .	25
8.1.2	MFClient states . . . . .	27
8.2	Dispatching Events . . . . .	27
8.2.1	shutdown . . . . .	31
8.2.2	handling disconnections gracefully . . . . .	31

8.3	subscription model	31
8.3.1	usage	31
8.3.2	Subscriptions	32
8.3.3	SubscriptionEventMessage messages	32
8.3.4	sticky subscriptions	36
8.4	Timing	38
8.4.1	market-data vs. trading connections	38
8.4.2	Timestamps	38
8.4.3	Clock Drift	39
8.5	architecture	39
8.5.1	components	39
8.6	events and messages	40
8.6.1	market data events	41
8.7	market data	41
8.7.1	market data	41
8.7.2	Entry Types (MDElement)	42
8.7.3	Pure entries	42
8.7.4	Structured entries	42
8.7.5	Trade Entries	43
8.8	data types	43
8.8.1	Numerical representation	43
8.9	trading interface	43
8.9.1	submit order	44
8.9.2	order types	44
8.9.3	client order identifiers	45
8.9.4	Trading Events	46
8.9.5	Delayed Order Responses	46
8.9.6	order time-in-force	47
8.9.7	Trade Capture Sequences	47
8.10	Sequenced messages and replay	47
8.11	drop copy feeds	49
8.11.1	ECN Drop Copy Feeds	49
8.11.2	Internal Drop Copy Feeds	49
8.12	administrative interface	50
8.12.1	MFAPIClient MFAdmin user Users	50
8.12.2	Parameterized Orders	53
8.12.3	Term orders	54
8.13	Banks	54
8.13.1	Quote Market Banks	54
8.13.2	Limit Order Book Banks	56

## 9 Market-making via the classic API 57

9.1	introduction	57
9.2	market-making concepts	57
9.3	feeds and trading venues	57
9.4	subscriptions	58
9.5	Message Replay	58
9.6	The MFMarketMaker interface	58
9.6.1	market-making scenarios	59
9.7	venue-specific considerations	64
9.7.1	Currenex RFQ	64
9.7.2	FXAll	64
9.7.3	Order fields	65
9.7.4	TradeCapture - detail data element	65
9.7.5	Error conditions	65
9.8	terminology	66
9.8.1	Identifiers within the MFAPI	66
<b>10</b>	<b>Venue-specific notes</b>	<b>69</b>
10.1	Bloomberg TradeBook	69
10.1.1	notes	69
10.2	CME	69
10.2.1	notes	69
10.3	EBS	69
10.3.1	TradeCapture and OrderDone messages received out-of-order	69
10.3.2	EBS Ai	70
10.3.3	EBS Live	71
10.4	Hotspot	72
10.4.1	Hotspot venue-specifics	72
10.5	Reuters	72
10.5.1	venue-specific notes	72
<b>11</b>	<b>Platform-specific notes</b>	<b>75</b>
11.1	Java platform notes	75
11.1.1	Java Zero-GC MFClient	75
<b>A</b>	<b>SBE protocol type mappings</b>	<b>77</b>
A.1	framing	77
A.2	market data	77
A.2.1	message types	77
A.2.2	message fields	78
A.2.3	trading	79
<b>B</b>	<b>MFAPI / SBE message field mappings</b>	<b>85</b>
B.0.1	type mappings	85

B.0.2 message field mappings – trading messages . . . . .	85
B.0.3 message field mappings – market data messages . . . . .	87

## Introduction

This document provides narrative documentation on the MarketFactory Application Programming Interface (“MFAPI”). It should be read before beginning integration with an existing trading system.

The Whisperer software provides connectivity, market data, and trading functions via a uniform interface to a wide range of foreign exchange ECNs and futures exchanges (collectively referred to as “Trading Venues”)

Each Trading Venue has its own combination of transport, format for messages, and conventions for market data and order management. Whisperer normalizes these differences into a common interface while preserving details of the original market data.

The server-side components of the system are implemented in C++. MarketFactory provides client-side implementations of the MFAPI in C++ (as static libraries for Linux), Java JAR files, and .NET assemblies. The language-specific MFAPI documentation should be read in conjunction with this. In each case, the language-specific documentation are authoritative for that implementation.

The Whisperer API tarball contains automatically-generated API documentation for both Java (JavaDoc) and C++ (Doxygen), in the top-level java/doc/ and cpp/doc/ directories.

## The SBE interface

The SBE (*Simple Binary Encoding*) protocol was commissioned by the CME in 2013, and has become the de facto successor to FIX / FAST. MarketFactory- has integrated SBE codecs into the Whisperer server product and provides a client implementation based on the the SBE reference sources, as provided by Real Logic Ltd. The SBE-based implementation provides for backwards ~~compatibility~~compatibility and a codec more efficient than that used by the classic MarketFactory API.

Martin Thompson of Real Logic Ltd. describes the SBE reference implementation as “a compiler that takes a message schema as input and then generates language specific stubs. The stubs are used to directly encode and decode messages from buffers. The SBE tool can also generate a binary representation of the schema that can be used for the on-the-fly decoding of messages in a dynamic environment, such as for a log viewer or network sniffer.”

The MarketFactory SBE client is delivered as an open source package, containing the MarketFactory message specification, the Real Logic compiler, and a higher level MarketFactory interface to the generated SBE codecs. Running the build process in the package generates the SBE codecs for C++, Java, and .NET.



# Using the SBE MFClient interface

The MFClient is the central class for communicating with the MarketFactory -servers. It provides factory methods to create market data and trading session objects (both of which have MFSession objects as a base class) as well as serving as the reactor/event loop for the client sessions.

The returned MFSession objects (or subclasses thereof) handle the network interaction with the MarketFactory -servers.

## 1.1 session creation

After instantiating an MFClient object, call `newTcpTradingSession()`, passing it your trading handler, which implements the MFTradingHandler interface. The handler interface is described in the next chapter.

This method takes MarketFactory login credentials (username and password), and a MarketFactory server address (hostname and port). The boolean option `autoRelogin` is used to indicate if the MFSession instance should automatically re-login when a disconnection of the MFClient from the MarketFactory server occurs.

Once the session is created, call `run()` to start the event loop, which dispatches incoming events from MarketFactory to the functions defined in the user-provided trading and market data handlers. Only one thread should ever call this method.

Calling `run()` triggers the `onStart` method on any registered handlers.

To stop dispatching events abruptly, call `stop`. This stops the event loop and closes open sockets. Calling `stop` triggers the `onStop` method on all registered handlers. This may be called from an external thread or internally from a handler.

The preferred method of disconnection is sending `logoutRequest()` for each session and then calling `stop`.

```
/**  
 * Creates a session for connecting to a particular  
 * venue gateway over TCP.  
 */
```

```

    * @param port specify 0 to have it automatically choose port.
    * @param autoRelogon automatically relogons when a disconnection occurs.
    */
MFMarketDataSession newTcpMarketDataSession(MFMarketDataHandler handler,
                                             String venueName,
                                             long venueID,
                                             String username,
                                             String password,
                                             String hostname,
                                             int port,
                                             boolean autoRelogon) throws MFException;

/**
 * Creates a session for connecting to a particular
 * venue gateway over TCP.
 *
 * @param port specify 0 to have it automatically choose port.
 * @param reconnectOnDisconnect automatically reconnects when a disconnection
 * occurs.
 */
MFMarketDataSession newTcpMarketDataPriceDepthBookSession(MFPriceDepthBookHandler
    handler,
                                                           String venueName,
                                                           long venueID,
                                                           String username,
                                                           String password,
                                                           String hostname,
                                                           int port,
                                                           boolean reconnectOnDisconnect)
    throws MFException;

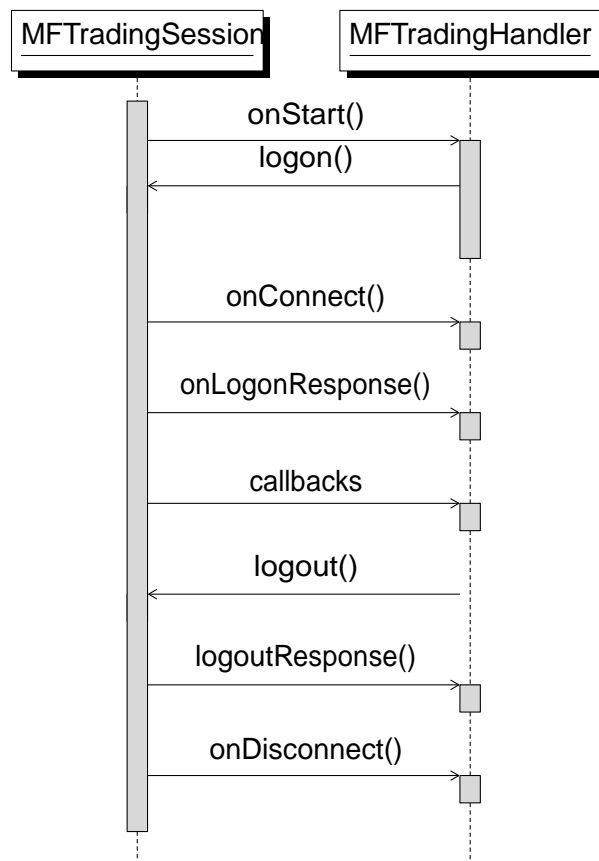
}

```

**Listing 1.1** – market data session factory methods

`newTcpMarketDataSession` expects a `MFMarketDataHandler` `newTcpMarketDataPriceDepthBookSession` expects a `MFPriceDepthBookHandler`

As for the `newTcpTradingSession()` method, both of these market data-related methods take `MarketFactory` login credentials (username and password), and `MarketFactory` server address details (hostname and port). The boolean option `autoReconnect` is used to indicate if the `MFCClient` instance should automatically re-login when a disconnection of the `MFCClient` from the `MarketFactory` server occurs.



**Figure 1.1.** – Showing the flow to demonstrate the session lifecycle of a session. Trading is shown, but MarketData is similar.

The MFClient layer lives in the directory \$MFAPI/MFClient/.

MFClient is the central class for managing sessions with MarketFactory. Session objects are generated by the factory methods of MFClient, and are automatically registered with this instance of MFClient.

com.marketfactory.api contains interfaces, while the packages under com.marketfactory.internal contains the implementations.

# The SBE session interface

The MFCClient provides factory methods to create market data and trading session objects, both of which are derived from the MFSession class.

MFSession objects handle the network interaction with the MarketFactory servers, such as market data subscriptions and trading actions from the client's system. They are also responsible for making calls into customer-implemented handlers conforming to the MFMarketDataHandler or MFTradingHandler interfaces.

Handlers implementing the MFMarketDataHandler or MFTradingHandler interfaces do the work of reacting to messages received from the MarketFactory server.

## 2.1 session creation and logon

First, use the MFFactory class to create an MFCClient instance. Second, create an instance of your application's trading handler (which implements the MFTradingHandler interface). Third, use the factory method `.newTcpTradingSession()` on the MFCClient instance to create the trading session object, passing in your application's trading handler. This returns an instance of MFTradingSession, which connects to MarketFactory via TCP. At this point, the dispatch of events to handlers may be started by calling `.run()` on the client.

```
final MFCClient client = MFFactory.createMFCClient();
final MyTradingHandler handler = new MyTradingHandler(client);
MFTradingSession session = client.newTcpTradingSession(handler, USERNAME, PASSWORD,
    HOSTNAME, PORT, true);
client.run();
```

**Listing 2.1** – creating a trading session

Calling `newTcpTradingSession()` above does not connect and logon the session. Once `client.run()` is called, and the session starts the first handler function to be called is `onStart()`. The `onStart()` handler function is responsible for calling the `.logon()` method on the session object.

Note that the MFTradingSession object is used for all venues – all but one of the methods on it include a `venueID` argument.

```

@Override
public void onStart(final MFTradingSession session) {
    try {
        session.logon();
    } catch (MFException e) {
        System.out.println("Failed to logon.");
        e.printStackTrace();
        System.exit(-1);
    }
}

```

**Listing 2.2** – trading session logon

The call to `session.logon()` will, in the of a successful logon, the `on(LogonResponse logonResponse, MFTradingSession session)` callback will be called.

## 2.2 session destruction and logoff

A session may be logged off in one of two ways: synchronously at the request of the client, or asynchronously by the server, either as a result of the client being a slow consumer of data, or being disconnected by MarketFactory support personnel.

```

@Override
public void on(com.marketfactory.mfclient.api.core.trading.Logout logout,
MFMarketDataSession session) {
    System.out.println("Processing logoff request on trading session " +
        session.getVenueName() + ", reason: " + logout.getText());
    try {
        session.logoutResponse(); // acknowledge logout request
    } catch (final MFException mfe) {
        mfe.printStackTrace();
        System.exit(-1);
    }
}

```

**Listing 2.3** – processing logoff request from server

## 2.3 composition

A `MFCClient` object contains a single `MFTradingSession` and multiple `MFMarketDataSession` instances, the latter referred to by *Venue ID*, which is the numeric value used for identifying MarketFactory feeds.

The `MFTTradingSession` is the interface for placing orders. The terminology used representing trading events is based on FIX 5.0 terminology. The `MFTTradingHandler` provides callbacks for each message type representing trading events, and `MFTMarketDataHandler` for callbacks for each message type relating to market data events.

Factory method `newTcpTradingSession`, creates a `MFTTradingSession` instance. `newTcpTradingSession` is passed the user-defined trading handler, which implements the `MFTTradingHandler` interface. The handler interface is described in the next section.

This method takes `MarketFactory` login credentials (username and password), and `MarketFactory` server address (hostname and port). The boolean option `autoRelogon` is used to indicate if the `MFTClient` instance should automatically re-login when a disconnection of the `MFTClient` from the `MarketFactory` server occurs.

Once created, call `run()` to start the event loop, which dispatches incoming events from `MarketFactory` to the functions defined in the user-provided trading and market data handlers. Only one thread should ever call this method.

Calling `run()` triggers the `onStart()` method on any registered handlers.

To stop dispatching events, send a logout, wait for a logout response, and then call `stop()`. This stops the event loop and closes open sockets. Calling `stop()` triggers the `onStop()` method on all registered handlers. This may be called from an external thread or internally from a handler.





## The SBE Handler interface

### 3.1 handler interfaces

The MarketFactory SBE implementation contains two interfaces to implement in order to connect to Whisperer.

MFTTradingHandler is the interface for trading operations, while MFMarketDataHandler is the interface for processing marketdata messages.

As an example of a simpler interface there is also MFPriceDepthBookHandler. This should be used when one wishes to consume market data snapshot-by-snapshot. Handlers extending this class receive complete PriceDepthBook updates rather than each individual MDEntryDecoder and MarketDataIncrementalRefreshDecoder.

### 3.2 common callbacks

These callbacks are common to all Handler types.

method	usage
onStart	Called immediately before the handler becomes active in the MFClient.run() event loop.
onStop	Called when the session is being removed from processing in the event loop.
onConnect	Called when the session establishes a socket connection.
onDisconnect	Called when the session socket disconnects.

**Table 3.1.** – methods common to both session types



# Market data via SBE

## 4.1 market data session

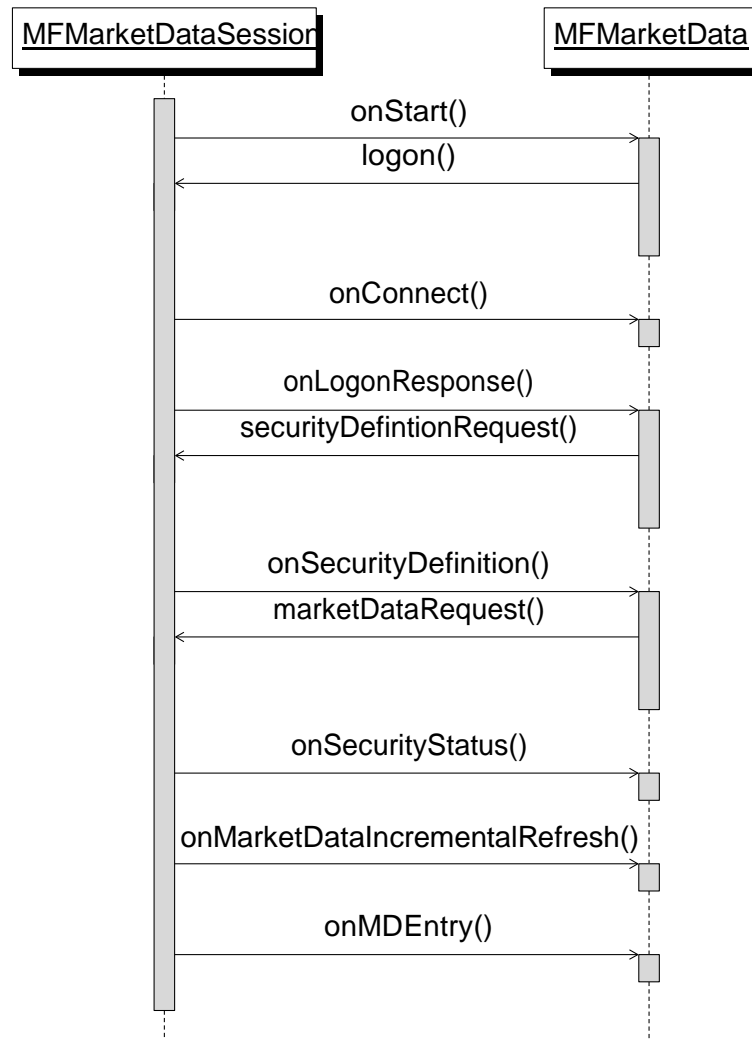
The MFMarketDataHandler interface specifies callbacks for each message type generated by an MFMarketDataSession object.

## 4.2 messages

These messages are received by the “on(...)” methods of a handler implementing the MFTradingHandler interface.

Message type	usage
LogonResponse	Called upon a successful logon.
Logout	Called when a logout is request to the client. The client should respond with a logout response.
LogoutResponse	A response to a successful logout request.
Heartbeat	Called when a heartbeat is received.
SecurityDefinition	Called on each security definition in response to a security definition request.
SecurityStatus	Called on a response to a marketdata request to notify the subscription status of a particular security.
MarketDataRequestReject	Called on a response to a marketdata request to notify the request could not be processed.
MarketDataIncrementalRefresh	Called on the beginning of a group of marketdata updates.
MDEntry	Called after a MarketDataIncrementalRefresh for each additional update in the group.
BatchesCompleted	Called to signal that a EBS batch is complete.

**Table 4.1.** – Market data message types



**Figure 4.1.** – Showing the flow to demonstrate subscribing to market data. After sending a MarketDataRequest, the client will receive a SecurityStatus message, which will indicate the status of the subscription, as well as the trade and value date of the instrument. Then the client will receive a “snapshot”, which will be the first MarketDataIncrementalRefresh group (which will have “numMDEntries” MDEntry messages following it, which are part of the same update), where the first entry will be an entry of type EmptyBook.

### 4.3 handler callbacks

Each of the following handlers implements the onStart(), onStop(), onConnect(), onDisconnect().

- MFMarketDataHandler
- MFTradingHandler
- MFPriceDepthBookHandler

### 4.4 callbacks

method	usage
onStart	Called immediately before the handler becomes active in the MFClient.run() event loop.
onStop	Called when the session is being removed from processing in the event loop.
onConnect	Called when the session establishes a socket connection.
onDisconnect	Called when the session socket disconnects.

**Table 4.2.** – methods common to both session types



# Trading handlers in SBE

## 5.1 subscriptions

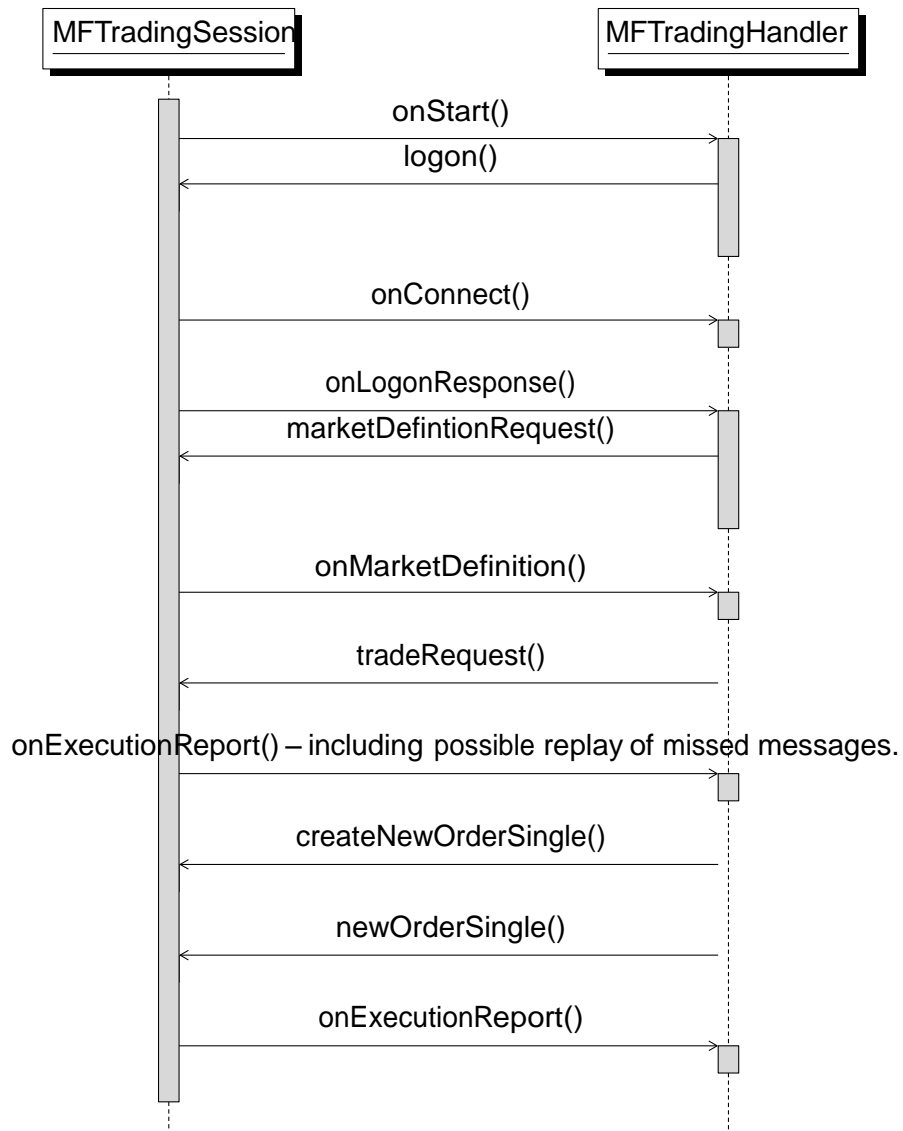
The MFTTradingHandler interface specifies callbacks for each message type generated by an MFTTradingSession object.

## 5.2 messages

These messages are received by the “on(...)” methods of a handler implementing the MFTTradingHandler interface.

Message type	usage
LogonResponse	Called upon a successful logon.
Logout	Called when a logout is request to the client. The client should respond with a logout response.
LogoutResponse	A response to a successful logout request.
Heartbeat	Called when a heartbeat is received.
ResendRequest	A request from the server to resend messages. Used to notify what messages may have been dropped. Response to this with a sequence gap fill.
MarketDefinition	Gives the definition of a particular venue.
TradingSessionStatus	A response to the trading session request. This notifies the client of the subscription status for a trading venue.
ExecutionReport	A response to an order action (order submission, modification, or cancellation) to give the user the status of the order.
OrderCancelReject	Sent when an attempt to cancel an order fails.

**Table 5.1.** – Order management message types



**Figure 5.1.** – Showing the flow to demonstrate subscribing to trading, replaying, and submitting an order. The client can subscribe to trading for each venue described in a `MarketDefinition`. Each venue has its own sequence number for messages. To submit orders, the application will use the creation methods, which do a zero-alloc creation of the `newOrderSingle`, and then send the message once the fields are filled in.



# STP booking interface

## 6.1 introduction

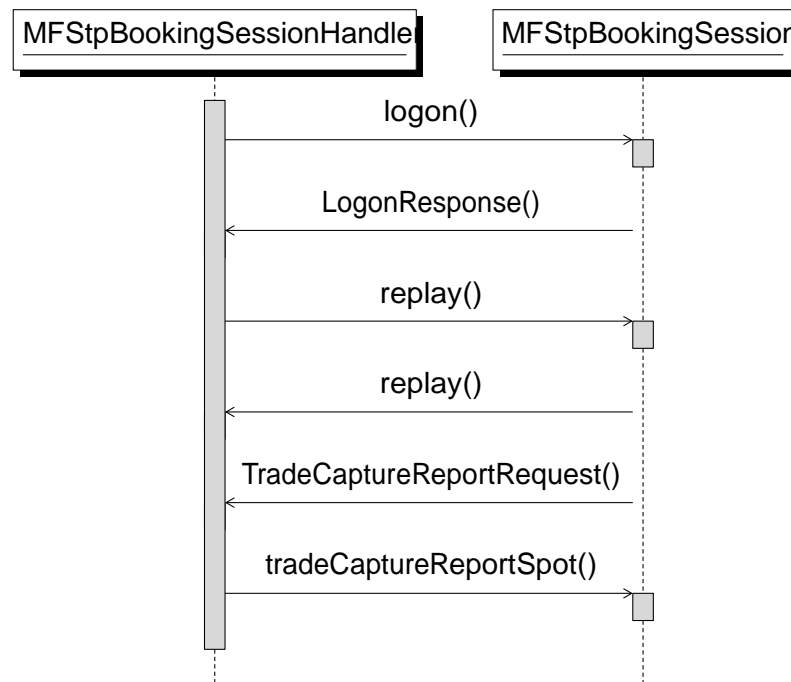
The STP booking interface is intended to be used for booking trades (fills) into a back-office system, such as Murex or Spectrum, via the MarketFactory Whisperer platform. Connection is via the SBE interface in the same manner as for market data and trading as defined in previous chapters.

## 6.2 session features

The MFStpBookingSession interface provides several methods:

- `tradeCaptureReportSpot()` used for submitting a `TradeCaptureReportSpot` via MarketFactory to the back-office platform.
- `createTradeCaptureReportSpot()` factory method to create a new `TradeCaptureReportSpot` object for submission.
- `tradeCaptureReportForward()` used for submitting a `TradeCaptureReportForward` via MarketFactory to the back-office platform.
- `createTradeCaptureReportForward()` factory method to create a new `TradeCaptureReportForward` object for submission.
- `tradeCaptureReportSwap()` used for submitting a `TradeCaptureReportSwap` via MarketFactory to the back-office platform.
- `createTradeCaptureReportSwap()` factory method to create a new `TradeCaptureReportSwap` object for submission.

Note that the `TradeCaptureReport` objects do not contain the same fields as the `TradeCaptureReport` defined in the `MFTTradingHandler`.



**Figure 6.1.** – MFStpBookingSession message sequence. This shows normal function. Both sides must have synced their sequence numbers before the server will send the TradeCaptureReportRequest. This session syncing is accomplished via the nextExpectedSeqNum field that is sent during logon.

## 6.3 handler features

The MFStpBookingHandler interface provides the basic set of callbacks in common with MFTradingHandler and MFCommonMarketDataHandler ( onStart(), onStop(), onConnect(), onDisconnect(), on( LogonResponse, ...), on( Logout, ...), on( LogoutResponse, ...), and on( Heartbeat, ... ) ).

However, the usage is different. Due to the nature of the session, both sides are expected to replay messages in case of dropped connection. The client is to replay TradeCaptureReports (with PossDupFlag set to true), while the server will replay TradeCaptureReportAcks. The replay is controlled via the nextExpectedSeqNum fields which are sent during logon, this communicates to the other side what sequence number it is expecting of the other side, and therefore if replay is required or not.

## Migrating from the classic MFAPI

MarketFactory has historically made its services available via a proprietary API, accessed through a binary provided to the customer in the form of a C++ library, .NET assembly, and Java JAR file. This interface uses an encoding more efficient than the protocols used by many venues. However, the approach taken in defining the protocol also meant that the client and server components needed to be upgraded together. Whilst compatibility was maintained within a given major release (for example 3.13.x), and updates for logic errors or additional venues were allowed, it was not possible to add new values to type enumerations on the server without simultaneously upgrading the client, and effectively changing versions.

There are several considerations to be taken into account when migrating from the existing MarketFactory API.

- In the classic API, all operations were made on the MFClient object. In the SBE implementation, trading and market data operations are split between the MFTradingSession and MFMarketDataSession.
- A *VenueID* is exactly the same as a numeric “feedID” in the classic MFAPI.

Please see Appendix B for the translations between fields in the the classic MFAPI messages and the corresponding messages in the SBE implementation.



# The Classic MFAPI

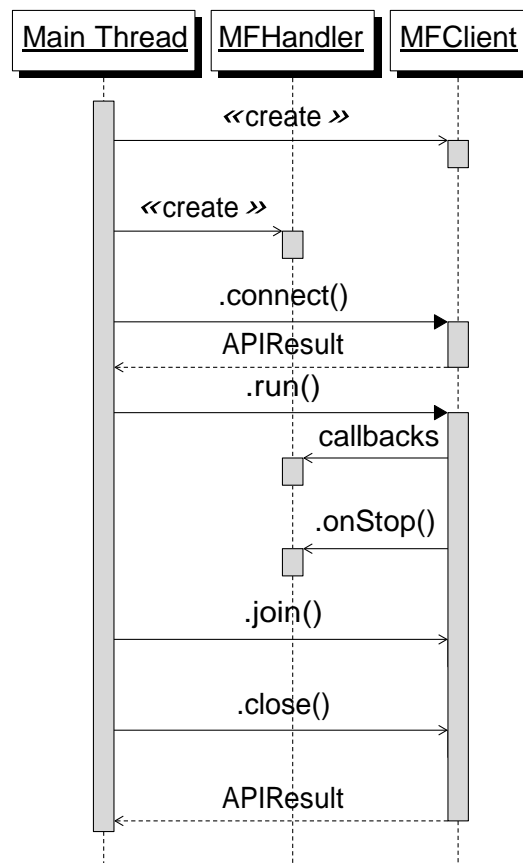
## 8.1 API connection lifecycle

### 8.1.1 Connecting an MFClient to the MF APIServer

The Customer Application creates an MFClient object via the static factory function provided by the MFAPI.

The Customer Application provides the hostname / IP, port and credentials (username, password assigned by MarketFactory and unique to each Customer Application). Two Customer Applications using the same credentials cannot connect at the same time. If this happens, the second MFClient object will receive USER\_ALREADY\_CONNECTED upon calling .connect(). The original user session remains unaffected. If, however, the already-connected user has been detected as idle for greater than the server-side “timeoutClient” idle time then the second connection takes over.

Creating the MFClient object establishes a TCP connection between the Customer Application and the Whisperer application.



**Figure 8.1.** – Connection lifecycle showing clean disconnection from Whisperer

### 8.1.2 MFClient states

An MFClient object may be in one of three possible states:

1. Not connected. Both `isConnected()` and `isRunning()` return `false`.
2. Connected, but not running `isConnected()` returns `true` if the client object is connected.
3. Connected and running `isRunning()` returns `true` if it is currently dispatching (e.g. `run()` has been called).

The “Connected” state means that the MFClient is connected to the Whisperer APIServer component. “Running” means that the MFClient is dispatching received messages via invoking callbacks on an MFHandler object. Dispatch of messages starts after `run()` is called on the MFClient instance.

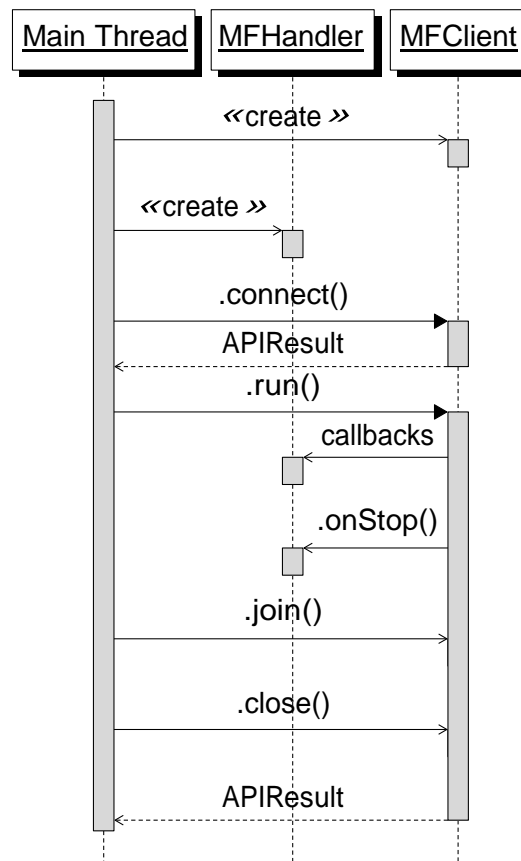
## 8.2 Dispatching Events

The MFAPI does not start a background thread to process events from the socket connection. This is done specifically so that the Customer Application has complete control over its resources. This means that the Customer Application must explicitly call the `run()` function / method to perform the required event loop processing. (The terms “function” and “method” are synonymous.)

The `run()` method processes incoming messages and dispatches them to the Customer Application’s MFHandler interface implementation; it is important to note that the `run()` method will block indefinitely as it waits for new incoming messages.

There are four scenarios in which the Client Application can return from the `run()` method (immediately before which we fire the `onStop()` callback for cases 2, 3, 4):

1. The MFAPI does not successfully start dispatching.
2. The Customer Application makes a request to stop event processing (possibly from a separate thread.) This prompts the callback `MFHandler.onStop()` and then returns `APIResult.OK`.
3. An internal error occurs during dispatch. Prior to returning, the connection to Whisperer will be terminated. This prompts the callback `MFHandler.onDisconnect()` followed by either `MFHandler.onConnect()` or `MFHandler.onStop()`.
4. The Customer Application’s MFHandler callback throws an exception, which is then rethrown by `run()`.



**Figure 8.2.** – Connection lifecycle showing clean disconnection from Whisperer

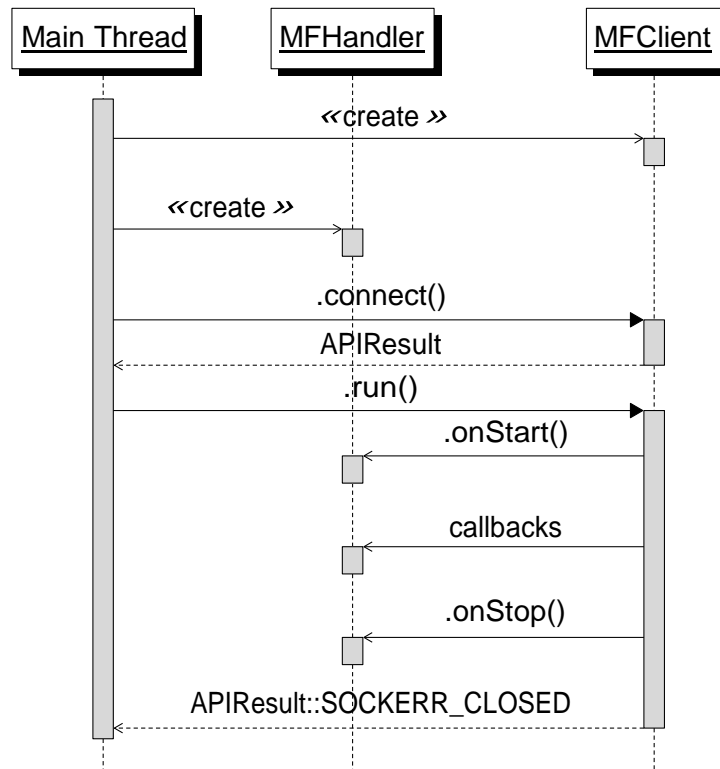
Dispatch processing can be done in the main thread or in a separate thread. This depends on the design of the Customer Application. If there are multiple MFClient instances, and thus multiple connections to the Whisperer, each MFClient should be dispatched from separate threads to avoid processing delays.

If the MFClient is not connected, calling run() returns error code APIResult::NOT\_CONNECTED. This can occur if MFClient.close() is called or if the connection has been broken. The run() method can be called if the MFClient is stopped but is still connected. (For example, if the stop() method is called.)

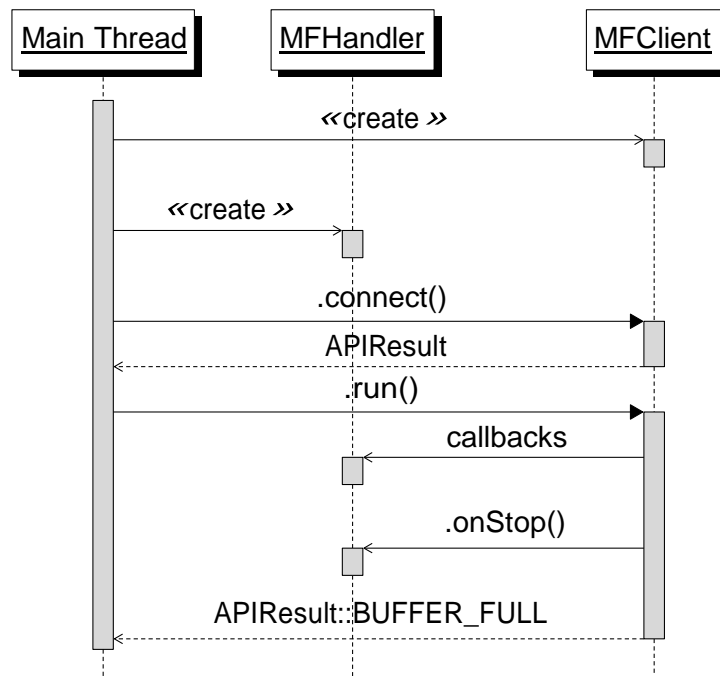
The run() method is passed an implementation of the MFHandler interface, provided by the Customer Application.

The Customer Application is responsible for dispatching events in a consistent and timely manner, otherwise data could accumulate on the socket receive buffer. If this condition persists, eventually the Whisperer server will detect the buffer filling up and disconnect the Customer Application's socket connection. The run() method will return with an error of SOCKERR\_CLOSED. As described above, dispatch processing can be handled in a separate thread.

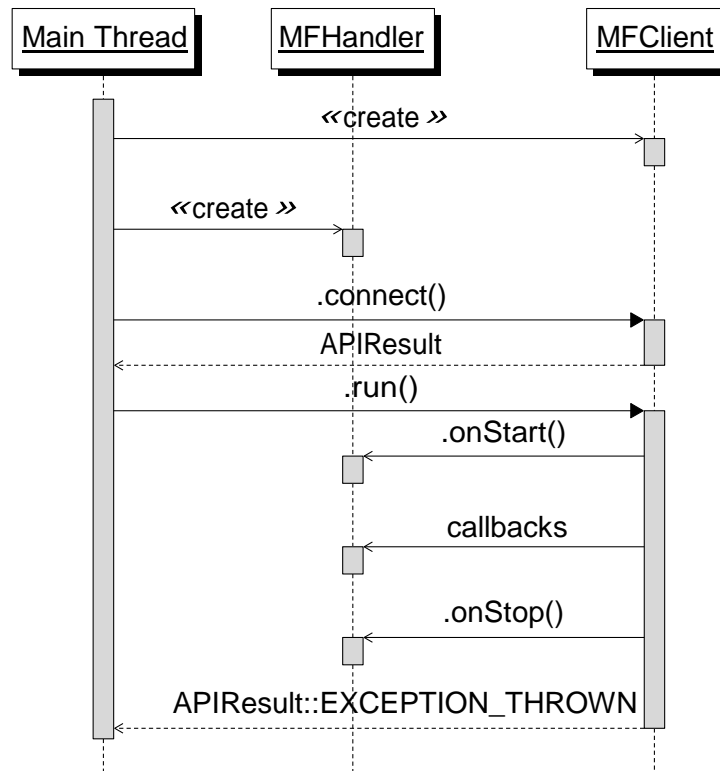




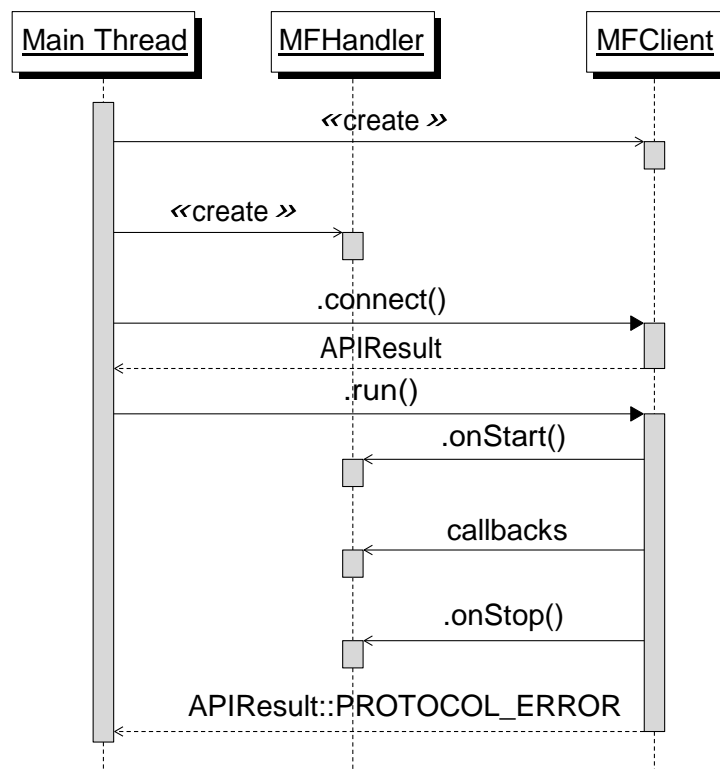
**Figure 8.3.** – Connection lifecycle showing disconnection from Whisperer due to network error.



**Figure 8.4.** – Connection lifecycle showing disconnection from Whisperer due to slow handling of callbacks by Customer Application



**Figure 8.5.** – Connection lifecycle showing disconnection from Whisperer due to unhandled exception in handler callback.



**Figure 8.6.** – Connection lifecycle showing disconnection from Whisperer due to client/server version mismatch.

### 8.2.1 shutdown

In order to terminate a running dispatcher loop, the Customer Application should call `MFCClient.stop()`. The corresponding `run()` invocation will exit with the appropriate status code. A stopped client is still connected, and as described in the preceding section, it is possible to call `run()` to resume dispatching events. To disconnect from the Whisperer server, the Customer Application should call `MFCClient.close()`.

`MFCClient.close()` may only be called when the `MFCClient` is not dispatching events.

### 8.2.2 handling disconnections gracefully

If the Customer Application gets disconnected from the Whisperer server, the `onDisconnect()` callback is invoked by the handler. Once the Whisperer server detects the disconnection of the Customer Application, it cancels all outstanding orders associated with the Customer Application.

It is important to note that during this very short window, (between the disconnection of the Customer Application and the order cancellations) a few additional trade messages (confirms) could trickle in. If this occurs, the Customer Application would miss these messages. To replay these missed messages, the Customer Application can make a special replay request, described in the *sequenced messages and replay* section.

Note that upon reconnection, if sticky subscriptions are disabled, the Customer Application will need to resubscribe for market data and trading.

## 8.3 subscription model

```
APIResult::Type subscribeMD(FeedID_t feedID, MarketID_t marketID);
APIResult::Type subscribeOF(FeedID_t feedID);
APIResult::Type unsubscribeMD(FeedID_t feedID, MarketID_t marketID);
APIResult::Type unsubscribeOF(FeedID_t feedID);
APIResult::Type unsubscribeAll();'
```

### 8.3.1 usage

- Every subscription request results in a single response in the form of a `SubscriptionEventMessage`.
- There is a subscription request identifier `clSubID` in the messages (not exposed to the customer yet; we will do that)

- Subscriptions are persistent across calls to `run()`
- Subscriptions are cancelled automatically if the client disconnects.
- Subscriptions may get cancelled automatically when a handler goes down (client must resubscribe).

```
void onSubscriptionEvent(const SubscriptionEventMessage& event, MFClient& client);
```

### 8.3.2 Subscriptions

Once a connection to the Whisperer server is established, the Customer Application needs to make subscriptions to the Feeds and Markets in which it is interested. There are two types of subscriptions: market data and trading. They differ in the following manner:

- Market data Subscriptions: The Customer Application must specify the Feed and Market combination in which it is interested: `subscribeMD(int feedID, int marketID)`.
- Trading: The Customer Application must only specify the Feed on which it wants trading enabled. Once the trading subscription is made: `subscribeOF(int feedID)` and an acknowledgement is received, the Customer Application is able to place trades on the specified Feed.

When making subscriptions the Customer Application should check the return code. For example, if the provided feed id is invalid the method will return `APIResult.INVALID_FEED` and the subscription request will not be sent to the Whisperer server.

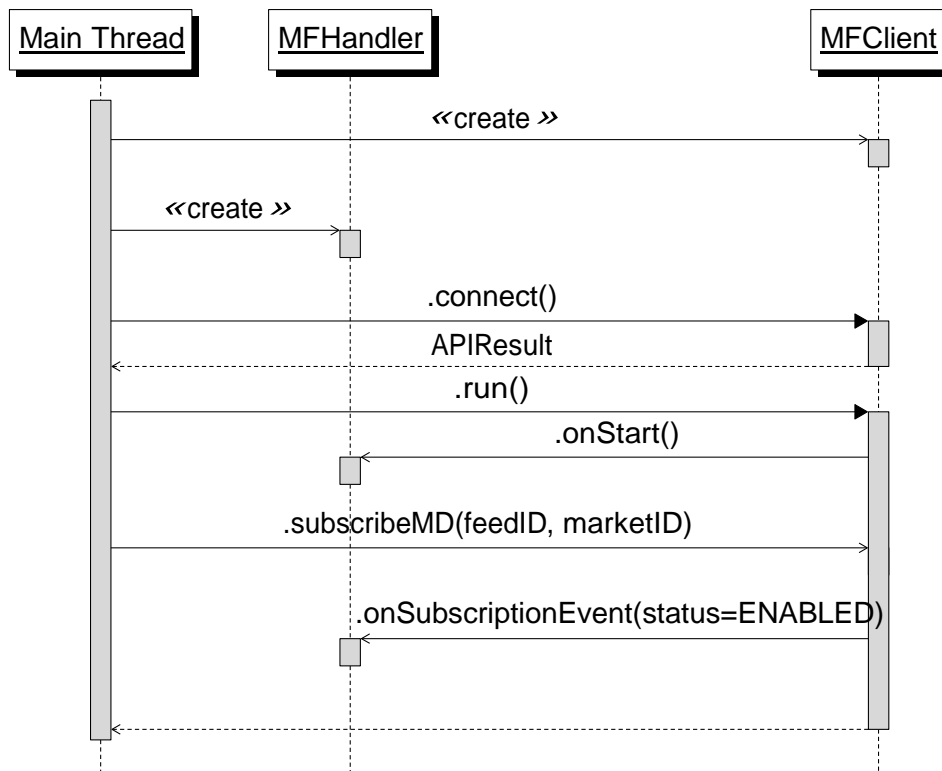
To disable subscriptions, the Customer Application must call the corresponding `unsubscribeMD()` or `unsubscribeOF()` method.

If the Customer Application becomes disconnected from the Whisperer server, all outstanding subscriptions to both market data and trading are automatically cancelled. Subscriptions do not persist across disconnections (unless the client is using sticky subscriptions).

### 8.3.3 SubscriptionEventMessage messages

Each subscription (for both market data and trading) is always acknowledged with a `SubscriptionEventMessage` message sent from the Whisperer.

`SubscriptionEventMessage` messages are received asynchronously via the `onSubscriptionEvent()` callback on the handler interface. The Customer Application should expect to receive an acknowledgement for each subscription request it makes.



**Figure 8.7.** – Successful market data subscription.

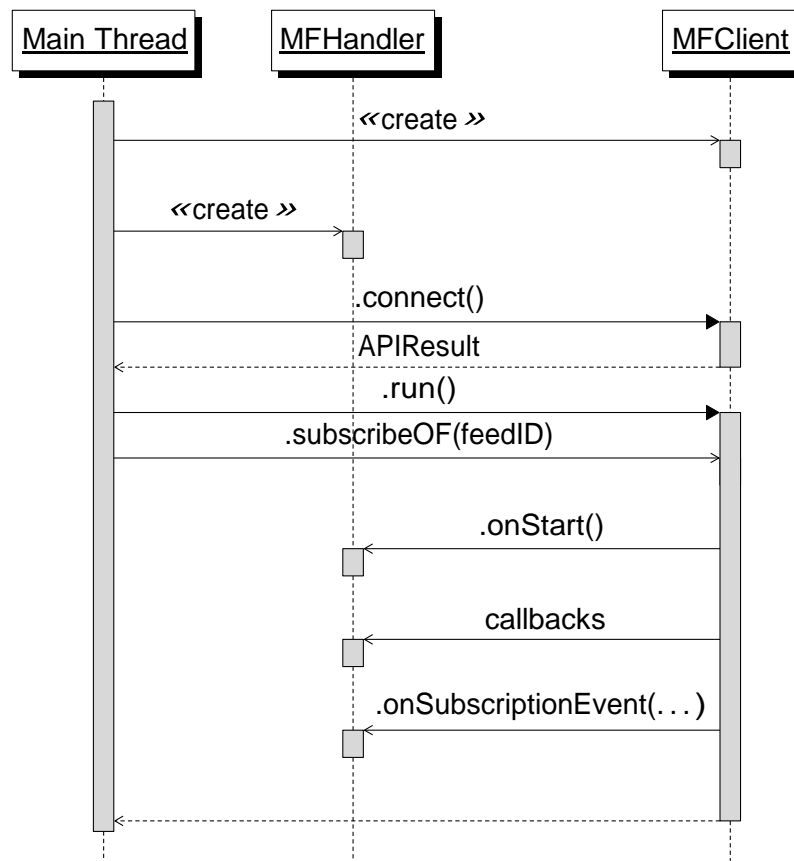
SubscriptionEventMessage messages can convey three possible subscription states, based on the value of the status field:

- where status is set to ENABLED, meaning that the Customer Application's subscription request was successful.
- where status is set to DISABLED, meaning that the Customer Application's subscription is no longer active. This can result only from the Customer Application un-subscribing from the Feed / Market.
- where status is set to ERROR, meaning that an error occurred and the subscription is no longer active. The error is indicated in the reason field of the SubscriptionEventMessage.

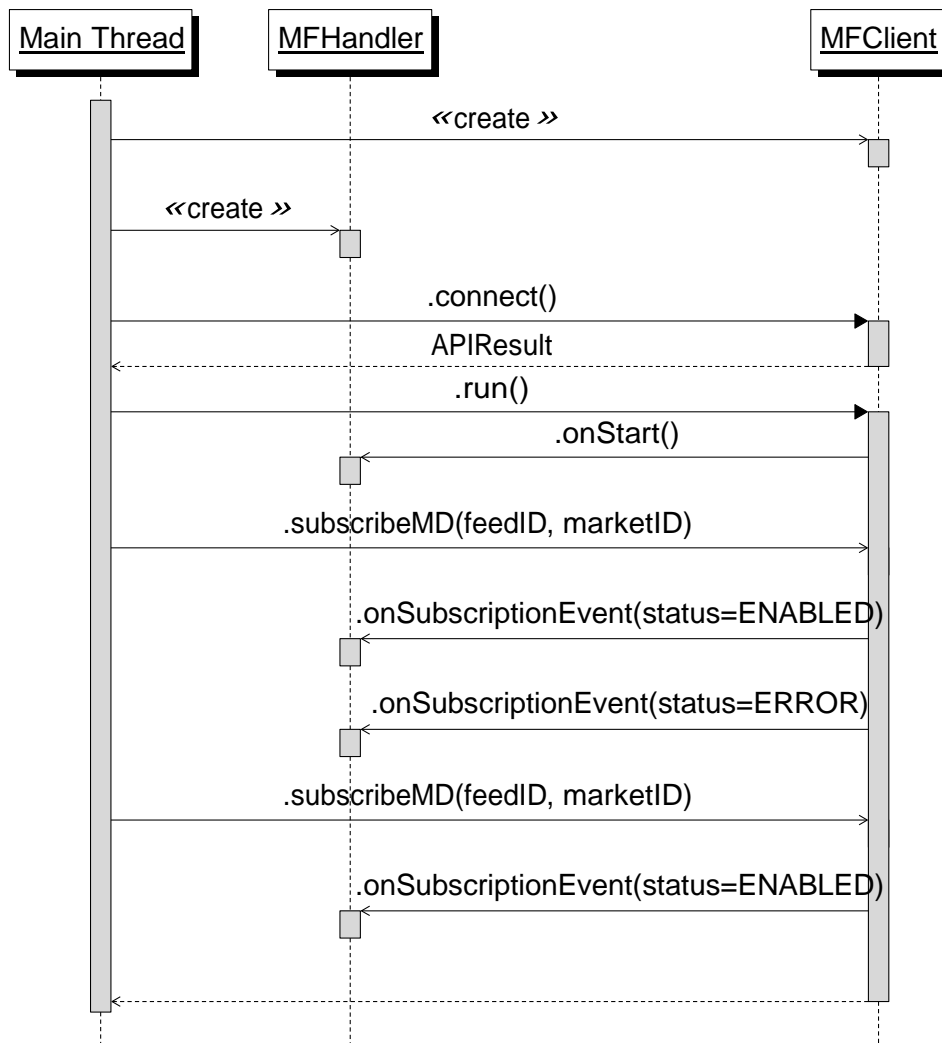
The following is an example of the sequence of messages where a Customer Application subscribes to market data:

The following is an example of the sequence of messages where a Customer Application unsuccessfully subscribes to trading:

SubscriptionEventMessage messages where status is set to ERROR are triggered when the Whisperer server loses a connection to a Trading Venue. All subscriptions to the affected Trading Venue are disabled. (For example, certain Trading Venues close temporarily at the end of their



**Figure 8.8.** – Unsuccessful trading subscription. The SubscriptionEventMessage passed to .onSubscriptionEvent() would have the status field set to ERROR, and the reason field the venue message, for example “Trading not allowed, please contact prime broker.”



**Figure 8.9.** – Temporarily disabled market data subscription.

trading day. This would trigger the generation of a SubscriptionEventMessage message with the status field set to ERROR

If a Customer Application receives a SubscriptionEventMessage message with the status field set to ERROR then it must re-subscribe to the affected Feed / Market combinations – this holds both for market data and trading subscriptions..

The following is an example of the sequence of messages where a Customer Application's market data subscription becomes temporarily disabled, and it must re-subscribe:

The Customer Application can configure its own re-subscription logic or it can use Sticky Subscriptions, described in the next section. (It is important to note that if custom re-subscription logic is implemented, the re-subscription attempts should be configured to at least 30 second intervals.)

### 8.3.4 sticky subscriptions

A Sticky Subscription is defined as a subscription (trading or market data) that remains active until the Customer Application explicitly unsubscribes. This means that if the connection to the exchange goes down, the subscription is NOT cancelled. The Whisperer application will automatically re-subscribe to the Trading Venue.

As before, the Customer Application will be notified that a Trading Venue is down with a `SubscriptionEventMessage` message where the status field is set to `ERROR`. However with Sticky Subscriptions set to `true`, the Customer Application does not need to re-subscribe. Once the Trading Venue is back online, the Customer Application will be notified with a `SubscriptionEventMessage` message where the status field set to `ENABLED`.

To enable sticky subscriptions, call `setStickySubscriptions(true)`. This forces all future subscriptions to be set to sticky.

It is important to note that subscriptions initiated before the `setStickySubscriptions(true)` is explicitly made remain non-sticky (re-subscriptions are necessary.)

There is an optional second argument to `setStickySubscriptions()`: `clearSubscriptionsOnStop`. This defaults to `true` in the description above, meaning that if the client disconnects from Whisperer with the `onDisconnected()` or `onStop()` callbacks, all sticky subscriptions will be cleared.

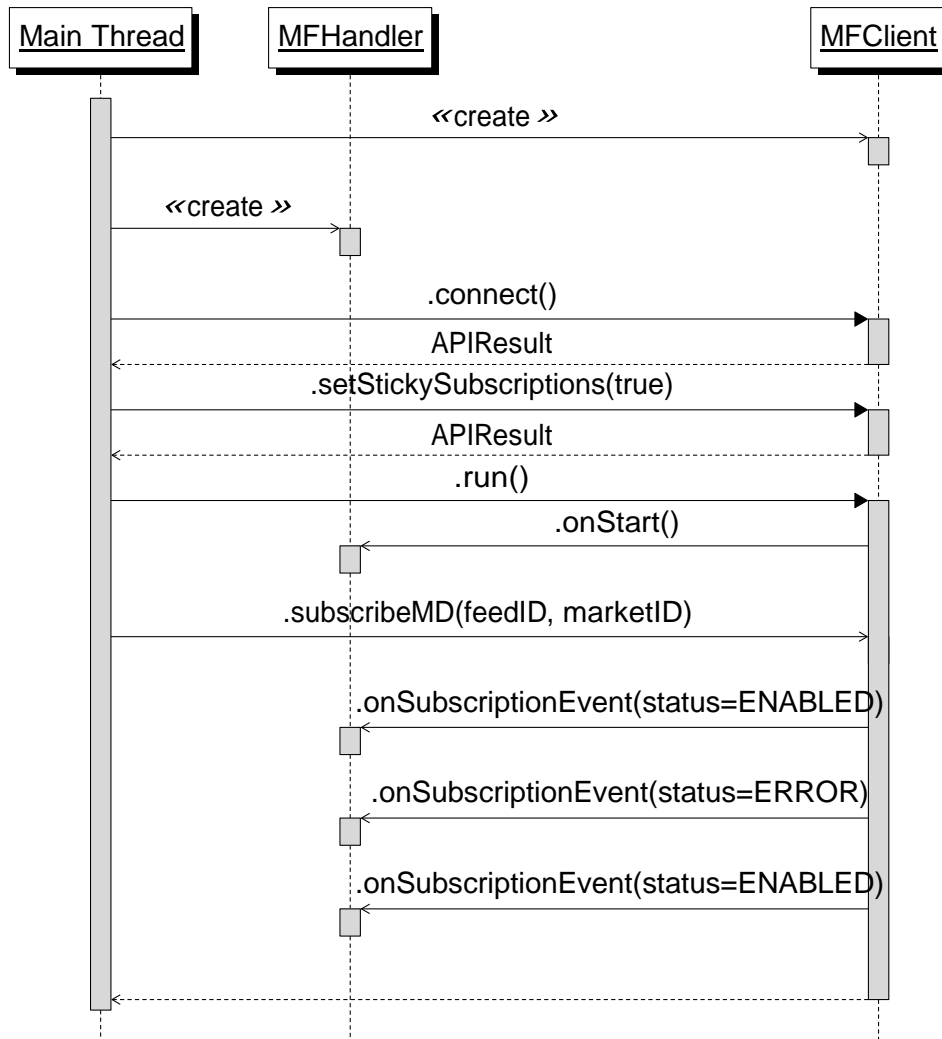
The customer must resubscribe after these callbacks to continue receiving data. When passed `false` for this argument, sticky will be preserved on disconnect and the customer must only subscribe once within the `MFClient` lifetime. Note that sticky subscriptions, and state in general, only apply to single instances of `MFClient`, which is to say that no state is shared between `MFClient` instances.

The following is an example of the sequence of messages where the sticky subscription is set and the market data subscription becomes temporarily disabled.

It is important to note that following the `SubscriptionEventMessage` message with status set to `ERROR`, the Whisperer will automatically re-subscribe and will only send the `SubscriptionEventMessage` message with the status field set to `ENABLED` message once it is able to successfully subscribe.

Even if the Trading Venue is down at the time of the initial subscription request and a `SubscriptionEventMessage` message with status set to `ERROR`, is initially received, the Whisperer will continue to re-try and will notify the Customer Application when the subscription is finally enabled due to a successful connection.





**Figure 8.10.** – Sticky Subscription Example. When `onSubscriptionEvent()` is invoked, the reason field is set to something like “reason=“Connection to ECN is down””

The `MFCClient` object keeps track of all sticky subscriptions and it will automatically resubscribe once it reconnects to the Whisperer server.

A sticky subscription persists over client disconnections from the server as well (either requested by the client through `.stop()` or unexpected). This means that if the connection to Whisperer goes down, the subscription is NOT considered cancelled.

## 8.4 Timing

### 8.4.1 market-data vs. trading connections

Market-data and trading connections are logically independent of each other and therefore are not synchronous. Market data updates and Trading Events may be out of order with respect to one another. Market data updates and Trading Events are however always delivered in chronological order.

### 8.4.2 Timestamps

Timestamps are expressed as nanoseconds since the UNIX Epoch (midnight of January 1st, 1970 UTC). Where the source resolution is limited to milliseconds, the nanoseconds portion defaults to 0.

All timestamps are expressed in universal time coordinates (UTC). To interpret the timestamps locally, timezone adjustments need to be made.

The following timestamps are used by the MFAPI:

- `timeExchange` as provided by the Venue to the same precision as their native message format.
- `timeArrival` set by the `FeedServer` component within the Whisperer server, when the message is received from the Trading Venue.
- `timeHandler` -set by the `FeedServer` in the Whisperer server, when the message has been processed by the `FeedServer`.
- `timeEnhanced` set at the output of the market-data enhancement component.
- `timeApiServer` set in the MFAPI Server client handler within the Whisperer, when the a message is received from an MFAPI Client.

- `timeApiClient` set in the client MFAPI (on the Customer Application).

`timeEnhanced` is captured at the same point as `timeHandler` and only corresponds to a message generated internally by the `FeedServer`. An example is when an aggregated feed loses connectivity to one of its constituent venues. A series of quotes will be removed from the book and these updates will have `timeEnhanced` rather than `timeHandler` set.

Not all messages have all timestamps set. For example, if the `Whisperer API Server` rejects an order, `timeExchange` will not apply and will be set to zero.

### 8.4.3 Clock Drift

Please note that the timestamps may not correlate with each other if they were measured on different computers. For example, the `timeExchange` (time at Trading Venue) and `timeHandler` cannot be subtracted to compute the delay between matching time and reception time, because of clock drift.

## 8.5 architecture

### 8.5.1 components

`Whisperer` is the name given to the collection of software is a multi-process application which software may reside on a single server or on a network of servers which may be separated geographically. The `Whisperer` server provides connectivity to spot and futures foreign exchange trading venues. “Connectivity” is defined as the ability to receive market data from and manage orders on a Trading Venue.

A Client Application is one which incorporates the `MarketFactory API` as part of its implementation. Common terminology and components of the system referred to in the remainder of this document include:

- **Whisperer** The generic name for the `MarketFactory` server software. A `Whisperer` installation consists of an `APIServer` (which faces the customer), and one or more `FeedServer` components (which face the exchanges), as well as auxiliary data-logging components which are used internally and do not interact directly with the outside world.
- **Customer** A `MarketFactory` client who has a business relationship with a number of Venues and wishes to connect to them all in a unified manner using the `MarketFactory API`.
- **Venue** An organisation which provides trading facilities, such as a bank or ECN. A Venue may provide the same services via multiple Feeds (for example via different protocols).

- **APIClient** The MarketFactory-provided software which MarketFactory customers link into their local application in order to connect to the MarketFactory Whisperer API Server.
- **Handler** The Customer-written software which performs different actions on reception of messages of various types from the API Server. This inherits from the MFHandler class.
- **API Server** The server-side software component to which the APIClient connects.
- **FeedServer** The server-side software component which connects from MarketFactory to the Venue.
- **Feed** The term for a connection to a Venue, which (normally) consists of two login sessions to the venue, one for market data, and one for order entry. A MarketFactory Customer may have more than one Feed to a given Venue, depending on the relationship the Customer has with the Venue. Feeds are referred to by numeric and string identifiers. These identifiers are globally consistent across all Whisperer installations.
- **Market** The MarketFactory name for a given instrument, for example EUR/USD for spot FX, or 6EH6 for CME Eurodollar futures expiring in March 2016.
- **MFAPI message** an application-level protocol data unit which is received by an MFClient object, and based on its type is passed to the appropriate function in a user-supplied class derived from MFHandler.

## 8.6 events and messages

“Events” are those things which happen in the real world, or in the context of MarketFactory, on a Venue. The term “message” as used in this document may be considered to be the reification of an Event.

In the first instance, the message is generated by the Venue and is sent (in the native format of the Venue) to the MarketFactory FeedServer component within the Whisperer software.

Whisperer normalizes these messages and forwards them to MFAPI clients. The API client software decodes this message and invokes the appropriate callback function on the Client Handler. This callback function sees the message as a MarketFactoryMFAPI object, not as the original message from the Venue.

```
void onMarketData(const MarketView& marketData, MFClient& client);
void onEventReport(const EventReport& eventReport, MFClient& client);
void onBatchComplete(const int batchDefID, MFClient& client);
```

## 8.6.1 market data events

**snapshots vs. incrementals** There are two types of market-data updates. A Snapshot sends all the market-data available (Pure, Structured, Trade) for the particular Feed/Market combination. An Incremental update sends only the information that has changed.

**requesting snapshots** A Snapshot is the first message received following a successful subscription receiving a SubscriptionEventMessage message handled by onSubscriptionEvent where the status field in the message is set to ENABLED). A snapshot can be empty which means that the order book is empty. Following the initial snapshot, incremental updates are sent. Under certain conditions additional snapshots are sent. This is a result of the FeedHandler re-syncing with the Trading Venue. The Customer Application must be prepared to process either. A snapshot can be explicitly requested via the requestSnapshot() method.

**Processing Snapshots** To process an empty snapshots, clear all entries and replace them with the values provided.

For snapshot updates, the updateAction field is is always set to NEW.

**Processing Incremental Updates** updateAction applies only to Pure Entries, which are indexed by price.

Possible values of updateAction are:

1. NEW Indicates the addition of a new price level and amount to order book.
2. DELETE Remove price and amount from order book. (Amount is set to 0)
3. CHANGE Update amount at the corresponding price level.

## 8.7 market data

### 8.7.1 market data

Market data updates are conveyed with the onMarketData() callback in the handler interface. Each update is for a specific feedID / marketID combination and contains a MarketView object. The MarketView object is either a full snapshot of the order book or an incremental update. In both cases, it contains a list of MDEntry objects, which are essentially market data entries.

### 8.7.2 Entry Types (MDElement)

There are three market data entry types, described in the following sections.

Figure 8. Conceptual model for pure and structured book entries

### 8.7.3 Pure entries

Pure Entries are designated by the MDElement.LEVEL enumeration and are indexed by price.

For a snapshot update there can be zero or more Pure Entries. A snapshot with no Pure Entries is an empty snapshot showing an empty order book.

For incremental updates, there must be at least one Pure Entry. The price field is always a valid numerical value, and the amount field is always a non-zero positive number. (Negative amounts are invalid. Zero amounts are valid for Pure Entries being deleted.)

Table 5. Pure Entry Enumeration

Pure Entry Enumeration (MDElement.*)	Definition
LEVEL	Each price level has an amount which represents the liquidity available at the specified price.

### 8.7.4 Structured entries

Structured Entries are used to encapsulate the different market conventions used by Trading Venues. (They are called “Structured” because they represent market data structures outside of a pure order book.) They are only populated where applicable and therefore may not be available in all updates. The price and amount fields may take on special values, which are described in the following section.

Below are the different types of Structured Entries:

Table 6. Structured Entry Enumeration

Structured Entry Enumeration (MDElement.*)	Definition
AMOUNT_CONS	Amount Constraint: is the worst price that needs to be paid or given to be able to trade a fixed minimum amount. This amount is specified by the Trading Venue and varies by currency pair. (This applies specifically to the “Reuters Standard Quantity”. )
SPREAD_CONS	Price/Spread Constraint: is the total amount that is available from the top of the book up to a fixed number of price levels away from top of book. In this case, the amount varies, while the price level remains fixed relative to the top of the book.

EXCHANGE\_BEST Trading Venue's Best Price: is the best price available on the Trading Venue. It is not credit screened. (This entry type is provided by Reuters.)

LOCAL\_BEST Best Local Price: is the best price bid or offer on the local trading floor; it is not necessarily dealable. (This only applies to EBS.)

### 8.7.5 Trade Entries

Trade Entry conveys information about the last transaction on the Trading Venue. No amount information is provided. (The special value of NA is therefore used in the amount.)

Trade Entry Enumeration (MDElement.\*) Definition

LAST\_TRADE Last Trade is provided by Reuters, Currenex, FXCM, and Hotspot. The side (bid / offer) of the resting quote is provided.

LAST\_TRADE\_NO\_SIDE No side information is provided. (This applies to GFX.)

WORST\_TRADE Worst Trade is provided by EBS; it is either the highest offer paid (worst offer) or the lowest bid given (worst bid) since the previous market update. The side is provided.

## 8.8 data types

### 8.8.1 Numerical representation

Prices and amounts are wrapped in the MFFloat class, which holds a fixed precision integer of 9 decimal places. The MFFloat class provides functions to check for special values, which are represented as large negative constants.

## 8.9 trading interface

As described in the Subscription Section earlier, the mechanics of the trading interface is similar to market data. The Customer Application must subscribe to the Feed on which it wants trading enabled.

Once the SubscriptionEventMessage containing the ENABLED status is received, the Customer Application is then able to place trades on the specified Feed.

### 8.9.1 submit order

To place an order, call `submitOrder()`.

`feedID` Feed to a valid Trading Venue. (Trading Subscription must be made first.)

`marketID` Identifier for the Market.

`side` BID means “I BUY”, while OFR means “I SELL” Note: BID / OFR refers to the side of the order, not the side of the quote being targeted.

`amount` needs to a valid `MFFloat` object

### 8.9.2 order types

**MARKET** The order is executed at the best price available.

**LIMIT** The order is executed at the specified limit price or better.

**MARKET\_LIMIT** The order is executed at the best price available; if the order can only be partially filled, the remaining amount remains in the CME’s order book as a limit order (at the specified limit price.) *(Only applicable to CME)*

**MARKET\_PROTECTION** The order is executed at the best price available; if the order can only be partially filled, the remaining amount remains in the CME’s order book as a limit order, where the limit price = *(bestbid when order was submitted + protectionPoints)* or *(bestoffer - protectionPoints)*. *(Only applicable to CME)*

(Protection points are preset by the CME by product. For currency futures, this value is typically 20 ticks. Please see <http://www.cmegroup.com/confluence/display/EPICSANDBOX/GCC+Price+> for further details.)

**STOP** The order is activated when the order’s stop price is traded in the market. Once activated, it becomes a market order. For a bid, the stop price must be greater than the last traded price. For an offer, the stop price must be less than the last traded price.

**STOP\_LIMIT** The order is activated when the order’s stop price is traded in the market. Once activated, the order becomes a limit order. For a bid, the stop price must be greater than the last traded price. For an offer, the stop price must be less than the last traded price.

**STOP\_PROTECTION** The order is activated when the order’s stop price is traded in the market. Once activated, the order becomes a limit order, where for bids the limit price = *(stop price + protectionPoints)* and for offers the limit price = *(stop price - protectionPoints)*.



IOC (Immediate or Cancel) If the order cannot be executed immediately, it is cancelled. Partial fills are permitted.

FOK (Fill-Or-Kill) If the order cannot be executed immediately for the FULL amount, it is canceled.

GTC (Good-till-Cancelled) The order remains in the order book until it is fully executed or until the user cancels it.

DAY (Good-for-day) The order remains in the order book until it is fully executed or until the end of the trading day: (when venue does daily shut down or at 1700 NYC time) limit price The maximum purchase price (for bids) or minimum sale price (for offers).

### 8.9.3 client order identifiers

The Customer Application can populate clOrdID and use this to track Trading Events. The clOrdID field is limited to a maximum size of 20 characters.

Valid characters include the printable ASCII character set (32 <= decimal value <=126) with the following exceptions:

character	name	decimal value
~	Tilde	126
`	Accent Grave	60
_	Underscore	95
!	Exclamation Mark	33
*	Asterisk	42
,	Comma	44
-	Hyphen	45
:	Colon	58
=	Equals Sign	61
[	Left Square Bracket	91
]	Right Square Bracket	93

**Table 8.1.** – disallowed characters in clOrdID

Where minQty is supported by the ECN, if the remaining amount drops below minQty, the remaining amount will be canceled by the ECN, resulting in OrderCanceled instead of OrderDone as the final state.

**modify order** modifyOrder() is supported on certain exchanges, namely CME, EBS, Currenex, Fxcm, FXall and Hotspot. Only GTC orders can be modified; both price and amount can be changed. To modify an order the customer must specify a new clNewID and the original clOrdID of the order.

Note that `submitOrder()` and `modifyOrder()` both result in the same `onOrderSubmitted()` callback on success. To differentiate, the Customer must correlate the requested `clNewID` with the response `clOrdID`.

The same approach applies for `OrderCanceled`, `OrderCancelRejected`, and `OrderRejected` responses.

In the case of `OrderCanceled`, the value in the `cxIID` field corresponds to `clNewID` value in the `ModifyOrder` message.

In the case of `OrderCancelRejected`, the value in the `clOrdID` field corresponds to `clOrdID` in `OrderSubmitted`.

**order cancellation** To cancel an order, call the `cancelOrder()` method. Either the `orderId` or the `clOrdID` can be used.

Additionally, a valid `cxIID` is required to identify the cancellation request. This identifier is restricted to the same format as `clOrdID`.

#### 8.9.4 Trading Events

In response to outgoing trading actions, a number of Trading Events follow. In this section we describe various scenarios and messages to be expected.

Note that these events are normalized across all exchanges and will be presented in the same manner through the MFAPI.

#### 8.9.5 Delayed Order Responses

If a Customer request, namely `SubmitOrder`, `ModifyOrder`, and `CancelOrder`, results in no acknowledgment from the venue within some configurable amount of time, the Whisperer APIServer will send an `EventReport` message to be processed by the `onEventReport` callback in the `MFHandler`.

This `EventReport` message will have the `eventCode` field set to `ERROR` and the `eventContents` field will contain a JSON-encoded object. This allows for arbitrary fields to be added on per-venue basis.

Upon receiving an error event, the Customer may search the contents for `"format": "json"` and proceed to parse as JSON if found. Example contents:

```
{
```

```

    "format": "json",
    "source": "marketfactory",
    "type": "OrderError",
    "orderID": LongLong,
    "clOrdID": "String",
    "reason": "No order response received within time constraint: 0.5s"
}

```

### 8.9.6 order time-in-force

**immediate-or-cancel orders** These orders are placed by setting the `timeInForce` `SubmitOrder.timeInForce` to `TimeInForce.IOC` (immediate or cancel).

To cancel an order, call `cancelOrder()` method with the `orderID` or `clOrdID` of the order to be canceled. If the order is already filled or the `Id` is invalid, the cancellation will be rejected. Whisperer will instead respond with a `OrderCancelRejected` message that will contain the reason for rejection.

### 8.9.7 Trade Capture Sequences

There may be more than one `TradeCapture` events per order, depending on the market activity at the Trading Venue. They represent individual fills. There is a single status for `TradeCapture` messages with the `status` field set to `DONE` for most Trading Venues.

For EBS there are two other states, `PENDING` and `ERROR`. The `PENDING` status is an early indication of a fill but not a full confirmation. A `PENDING` `TradeCapture` can in very rare cases go to an `ERROR` status, in which case EBS would need to be contacted by phone.

#### general notes

- An `OrderDone` message always signals the completion of an order, `IOC` or `GTC`.
- An `OrderCanceled` message always signals the completion of an order, either for the full amount or for the final portion of it.

## 8.10 Sequenced messages and replay

Every trading message sent from the Whisperer application to the Customer Application has a sequence number. The Whisperer application maintains a running sequence number for every feed and MFAPI user. The sequence number for each (user, feed) pair begins at 0 and

is incremented by one with every trading message sent by the feed to the user. The sequence number is reset on a weekly basis (over the weekend).

A Customer Application may choose to replay trading messages at any time. The sequence number for a Customer Application persists through disconnects. The number could be retrieved by calling `getSequenceID(feedID)` once the Customer Application has been successfully subscribed (after receiving `SubscriptionEventMessage` with status enabled). -2 means the last sequence number is unknown, it is returned if not subscribed yet. -1 means there are no trading messages.

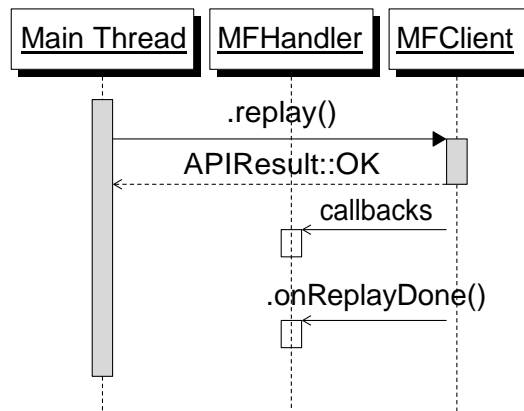
An example of a situation where a customer would request a replay is if the customer application becomes disconnected. Upon reconnection, it would request a replay for missed messages beginning with last sequence number stored. This can be done by calling `replay(X, n+1, -1)`, where `n` is the last sequence number that the Customer Application received from feed `X`. This sequence number would need to be stored by the Customer Application, and it would need to call `replay()` for each feed it may have missed messages from.

Each replay request will result in either a message replay or a replay rejection. The end of a replay is conveyed with a `ReplayDone` message with status `ReplayStatus.DONE`.

The replay rejection is also conveyed with the `ReplayDone` message but with status `ReplayStatus.REJECTED`.

Below are reasons why a replay request would be rejected:

- Replay request sent when the client is not subscribed.
- Replaying a fixed interval (from and to sequence number). This is no longer supported.
- Replay request sent while the client is in replay mode.
- Replay request containing more than 1000 messages. This value is fixed in the Whisperer configuration. (This limitation may go away in newer Whisperer versions)



**Figure 8.11.** – Replay sequence

## 8.11 drop copy feeds

### 8.11.1 ECN Drop Copy Feeds

An ECN Drop Copy Feed provides a sequenced copy of all trades done on the ECN. The set of trades sent is determined by the permissions assigned by the ECN.

The Drop Copy Feed does not provide for subscriptions to market data nor allow for trading; however subscriptions are still necessary. The MFAPI client needs to subscribe to trading. The Customer Application is assigned permission to receive Drop Copy messages via the MFAPI login assigned by MarketFactory.

Not all ECNs provide this functionality. You can check if a given feed is a Drop Copy Feed by looking at the `APIFeed.flags` bitmask and comparing it with `FeedFlag.DROPCOPY`.

For feeds having complex counter party information the `counterpartyId` field will contain a JSON object representing the information. (See Feed specific appendix for details)

### 8.11.2 Internal Drop Copy Feeds

An Internal Drop Copy Feed provides a sequenced copy of trades done within the Whisperer system. Each Drop Copy Feed feed may be configured to include all trades possibly filtering by user, feed, market, or a combination of all three. This configuration is done server-side, and the user is requested to contact MarketFactory support for this, depending on the former's requirements.

Subscription is necessary in a similar fashion to the ECN Drop Copy Feeds.

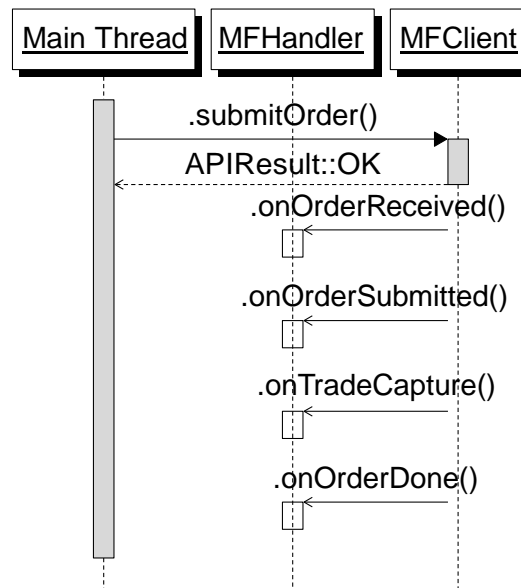


Figure 8.12. – Complete IOC fill

## 8.12 administrative interface

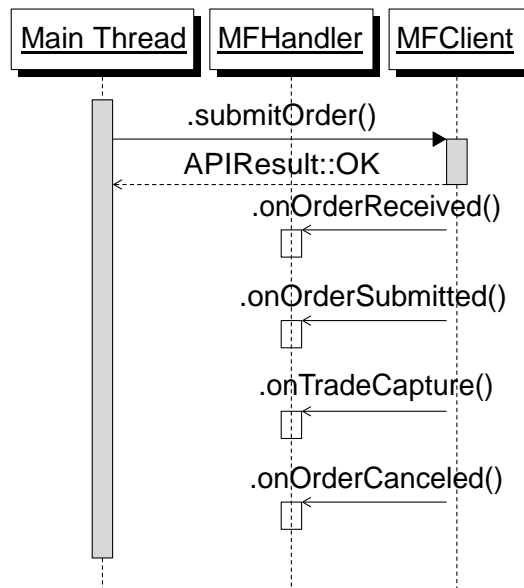
### 8.12.1 MFAPI Client MFAAdmin user Users

MarketFactory can configure MFAPI Clients to have administrative rights. This includes the ability to lock and unlock other MFAPI Clients' trading privileges and logins. (Locking an MFAPI Client will cancel all its outstanding orders and force the MFAPI Client to disconnect from the MarketFactory server.)

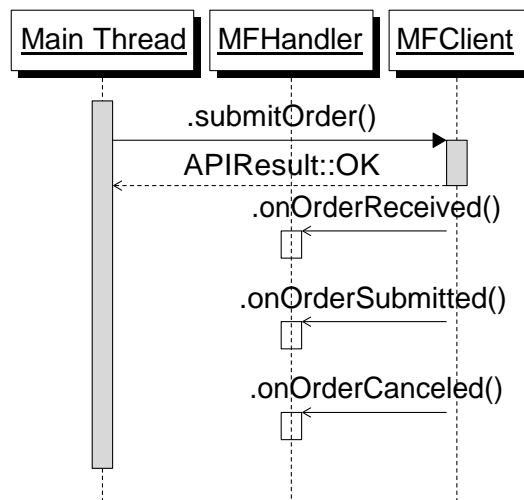
On the client side, the administrator MFAPI client must be of type MFAAdmin. Administrative rights are intended for MFAPI clients that monitor overall API Client trading activity. Typically, administrative clients also listen to dropcopy trading messages. If an administrative client is defined, it must connect and explicitly call `allowClientsToConnect()` in order for other API Clients to connect, subscribe to market data and trade. The administrator must call `allowClientsToConnect()` every time it re-connects. This is intended to allow the administrator to do any cleaning up or catching up once it re-connects before allowing other users to connect.

If the MFAAdmin user becomes disconnected and remains disconnected for over a configurable amount of time e.g. 3min), all other MFAPI clients will also be disconnected and be prevented from connecting. If more than one administrator is defined, then at least one administrator needs be connected for the MFAPI Clients to remain connected and trading.

**Complete IOC Fill** An OrderDone message signals the completion of the order and that it has been fully filled.



**Figure 8.13.** – Partial IOC fill

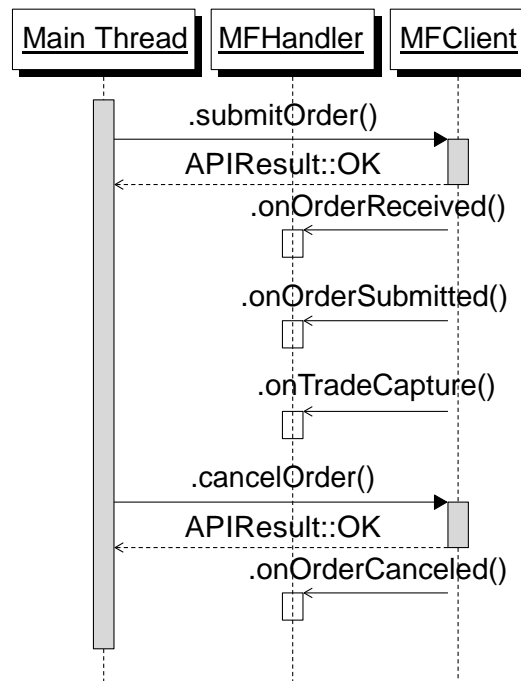


**Figure 8.14.** – IOC miss – nothing done

**Partial IOC Fill** This occurs when part of an order gets filled the sequence concludes with an OrderCanceled event for the remaining size of the order.

**IOC miss** If no part of the IOC order can be filled, it is cancelled without any TradeCapture events.

**IOC with Minimum Quantity** If the minimum quantity cannot be filled, it is cancelled. If the minimum quantity can be filled, the order results in trade captures. If there is a remaining amount, it is subsequently cancelled:



**Figure 8.15.** – Cancellation after partial fill

**Resting (GTC) Limit Orders** Resting orders are specified by using `timeInForce GTC`, and `orderType` as `orderType.LIMIT`. They remain in the order book of the ECN until fully filled or cancelled.

**Complete Fill** This sequence is similar to the Complete IOC Fill except that there may be arbitrary time passing while some of your order sits on the book to be filled:

**Resting (GTC) Limit Orders with `maxShow`** `maxShow` is used to specify the amount to display in the market. It can be used in conjunction with resting limit orders:

**Cancelling after partial fill** Resting orders can be cancelled after partial fills. On ECNs where trade captures can have either a PENDING or DONE state, trade captures with a state of DONE may be received after the `OrderCanceled` message is sent.

(one or more trade capture sequences which have a status DONE may be sent, in the case where there are PENDING trade captures sent)

**Resting (GTC) Market-Limit Orders** Orders with `orderType` set to `MARKET_LIMIT` are executed at the best market price. The remaining amount remains in the order book at the specified limit price:

**Resting (GTC) market orders with protection** Orders with `orderType` set to `MARKET_PROTECTION` are executed at the best market price. The remaining amount remains in the order book at the



limit price of best market +/- Protection Points (depending on whether the order is a bid or offer).

For an offer, the Protection Points would be subtracted. Protection Points are specified by the CME. See <http://www.cmegroup.com/confluence/display/EPICSANDBOX/GCC+Price+Banding>

Resting (GTC) Stop-Limit Orders with *orderType* set to STOP\_LIMIT) are activated when the stop price is traded in the market. The order then becomes a limit order, using specified limit price.

**Resting (GTC) Market-Stop Orders with Protection** Resting (GTC) Stop Orders with Protection (with *orderType* set to STOP\_PROTECTION) are activated when the stop price is traded in the market. The order then becomes a limit order of stop price +/- Protection Points.

For a bid, the Protection Points would be added. Protection Points are specified by the CME. See <http://www.cmegroup.com/confluence/display/EPICSANDBOX/GCC+Price+Banding>

## 8.12.2 Parameterized Orders

The MFAPI also has the ability to allow extendable parameters to order submission. This is done by filling in an ArrayList with of key value pairs (both being strings) and passing this to the appropriate submit method. This enables:

### EBS

- The ability to submit an order with *PriceDiscretion* (as the key) and a price increment (as a String) as the value.
- The ability to submit an iceberg order with different display behaviour using “DisplayMethod” (as the key) and a method, either “1” or “3” (as in a String) as the value.
- The ability to submit an iceberg order with different replenishment times using “IcebergHighRandomTime” (as the key) and a milliseconds (encoded in a String) as the value.

### FXAll

- The ability to submit an order with a minimum amount by using “MinAmount” as the key and a numeric value (encoded in a String) as a value.

### HotSpot

- The ability to submit an order with a minimum amount by using “MinAmount” as the key and a numeric value (encoded in a String) as a value.

## Currenex

- The ability to submit an order with a minimum amount by using “MinAmount” as the key and a numeric value (encoded in a String) as a value.
- The ability to submit an order with the order quantity in the term currency by using “OnTerm” as the key and the term currency as a value.

## RBC

- The ability to submit an order with the order quantity in the term currency by using “OnTerm” as the key and the term currency as a value.

## Barx

- The ability to submit an order with the order quantity in the term currency by using OnTerm as the key and the term currency as a value.

## Citi

- The ability to submit an order with the order quantity in the term currency by using OnTerm as the key and the term currency as a value.

### 8.12.3 Term orders

Term orders are orders where

hotspot switches the base and counter currencies. Other venues have the amount in the

## 8.13 Banks

### 8.13.1 Quote Market Banks

Most bank venues are quote-driven markets. This means that prices are streamed with an associated identifier that must be given when an order is submitted to execute on the price. The bank quote-driven markets themselves have different prescribed methods for hitting / filling prices; MarketFactory manages these differing rules. This is different than ECNs, which use a limit order book. The approach that MarketFactory uses to normalize banks is to stream aggregated prices and amounts, similar to the way prices are streamed for ECNs, but to also put the individual quote amounts in the components field. These MDEntry items are marked with MDElement.LEVEL\_QUOTE.

To execute on these quotes, the user traverses the list of MDEntry structures and examines the components of each price level. To perform a “sweep” the user must submit multiple limit orders, one for each component amount up to the desired amount that the user wants to trade. These orders usually have a time-in-force of FOK (Fill-Or-Kill), though there are some exceptions.  
xxx

For instance, given this snapshot for EUR/USD:

```
<MktDataMessage ... mdEntryList=[  
  
  (price=1300590000 amount=3000000000000000000 updateAction=NEW  
  element=LEVEL_QUOTE side=BID source=8 numberOfOrders=2  
  components="2000000+1000000")  
  
  (price=1300570000 amount=2000000000000000000 updateAction=NEW  
  element=LEVEL_QUOTE side=BID source=8 numberOfOrders=1 components="")  
  
  (price=1300550000 amount=1000000000000000000 updateAction=NEW  
  element=LEVEL_QUOTE side=BID source=8 numberOfOrders=1 components="")  
  
  (price=1300930000 amount=2000000000000000000 updateAction=NEW  
  element=LEVEL_QUOTE side=OFR source=8 numberOfOrders=2  
  components="1000000+1000000")  
  
  (price=1300950000 amount=3000000000000000000 updateAction=NEW  
  element=LEVEL_QUOTE side=OFR source=8 numberOfOrders=1 components="")  
  
  (price=1301010000 amount=1000000000000000000 updateAction=NEW  
  element=LEVEL_QUOTE side=OFR source=8 numberOfOrders=1 components=""))]  
...>
```

then to buy 3.5MM, then the user would submit three orders: 1MM at 1.30093, 1MM at 1.30093, and 1.5MM at 1.30095.

Even though the user is submitting orders on individual quotes, the user does not have to manage the quoteIDs as Whisperer does this for the user. Besides making book management easier for the user, this allows Whisperer to give price improvement in the case where the market changes before the order is sent to the venue.

Whisperer only publishes tradeable prices to the user. If a bank publishes indicative prices, they are ignored

### 8.13.2 Limit Order Book Banks

Not all banks are quote markets. For example, Morgan Stanley and UBS display prices equivalent to an ECN-like limit order book, while some, such as Citi, provide both quote-driven markets and and ,

Users may execute on these venues in a manner similar to an ECN. In this case, the MDEntry items would be marked with MDElement.LEVEL, like ECNs.

Internally, these venues allow “sweeping” (execution on multiple price levels) by submitting VWAP orders. This order is constructed by giving the total desired amount and the volume weighted average price for all of the levels that the user desires to "sweep". Users of Whisperer just need to submit a limit order, Whisperer internally converts this order to the equivalent VWAP order.

# Market-making via the classic API

## 9.1 introduction

## 9.2 market-making concepts

Market-making is the process of offering liquidity to the market. In contrast to the “taker” MFAPI, market-making involves sending two-sided quotes to the market and reacting when either side is hit by another exchange member.

Market-making may take two forms – RFQ and continuous price streaming.

## 9.3 feeds and trading venues

The Whisperer system or MarketFactory Aggregator provides connectivity to a number of Market Making Venues. As previously mentioned, a “Feed” is a connection to a Trading Venue via the MFAPI.

Depending on the configuration, there may be multiple Feeds to a single Trading Venue. For example, a customer may have multiple Currenex connections, each pointing to a different Currenex liquidity pool; this would appear to the Customer Application as separate Currenex Feeds, each with their unique numeric identifier and name.

A Feed is the fundamental unit to which the Customer Application subscribes. A market making Feed can support both pricing and trading. The pricing feed may sometimes be referred to as the “stream” or “quote” feeds.

- In the Pricing interface, messages, in the form of mass quotes, flow from the Whisperer application to the Customer Application.
- In the Trading interface, messages are bi-directional. Trading actions flow from the Whisperer Application to the Customer application, while Trading events flow from the Customer application to the Whisperer Application.

Each feed is identified by a unique integer in the Whisperer application, as in the Taker model. Subscriptions are made to specific feeds. Market data and trading messages contain the feedID identifying the Trading Venue with which they are associated.

## 9.4 subscriptions

Once a connection to the Whisperer server is established, the Customer Application needs to make subscriptions to the Feeds and Markets in which it is interested. There are two types of subscriptions: pricing and trading. They differ in the following manner:

- **Pricing** `subscribePricing(int feedID, String clSubID)`
- **Trading** `subscribeTrading(int feedID, String clSubID)`

Note that the subscription request identifier `clSubID` should be used to track the subscription request. Once subscribed to **both** pricing and trading, the client will receive quote requests for instruments to stream to the venue.

When making subscriptions the Customer Application should check the return code. For example if the provided feed id is invalid the method will return `APIResult.INVALID_FEED` and the subscription request will not be sent to the Whisperer server.

To stop receiving quote requests call the corresponding `unsubscribePricing()` or `unsubscribeTrading()` method.

If the Customer Application is disconnected from the Whisperer server, all outstanding subscriptions to both market-data and trading are automatically cancelled. Subscriptions do not persist across disconnections (unless the client is using sticky subscriptions).

## 9.5 Message Replay

The trading subscription of the maker API supports replay of messages in case of a disconnection. This is described in the *Sequenced Messages And Replay* section.

## 9.6 The MFMarketMaker interface

The MFMarketMaker is designed similarly to MFClient in terms of style but offers a different work flow. The sections above describing connecting, dispatching, subscriptions, replay, etc. all remain true but Taker market data and trading have been replaced with market making functionality.

After a successful connection to the server, there are two new subscription types required to begin market making. To receive QuoteRequest messages and start quoting, use `subscribePricing()`. To receive order submissions use `subscribeTrading()`.

In the case of pricing, the maker has the option to reject individual QuoteRequests or to initiate a stream of MFQuotes for each request. The QuoteRequest remains active and quoting may continue until either the maker sends CancelQuote or receives a MassQuoteAck indicating request expiration. If at any time a maker quote is rejected, the session will disconnect.

Trading requires an active QuoteRequest and streaming MFQuotes before the maker will receive NewOrder from a taker. The maker may choose to respond with RejectOrder or AcceptOrder, but if the response isn't timely, the venue will send OrderTimeout. In the rare case AcceptOrder is rejected, the venue will send TradeCaptureAck. MarketFactory may also send this message if an acceptance didn't reach the exchange due to disconnection. Additional details on coding and venue-specific behavior can be found in the javadoc and venue matrix spreadsheet respectively.

Note that market making is based on negative acknowledgements – responses are only on error, not on success. Additionally, there are a few restrictions specific to maker clients.

Only one preconfigured user will have access to a maker venue – multiple users are not allowed. MarketFactory will provide one MFAPI user per maker feed

There are no risk checks, offline reports, or dropcopies when market making.

## 9.6.1 [market-making scenarios](#)

**streaming quotes**

**reject quote request**

**reject quote request by venue**

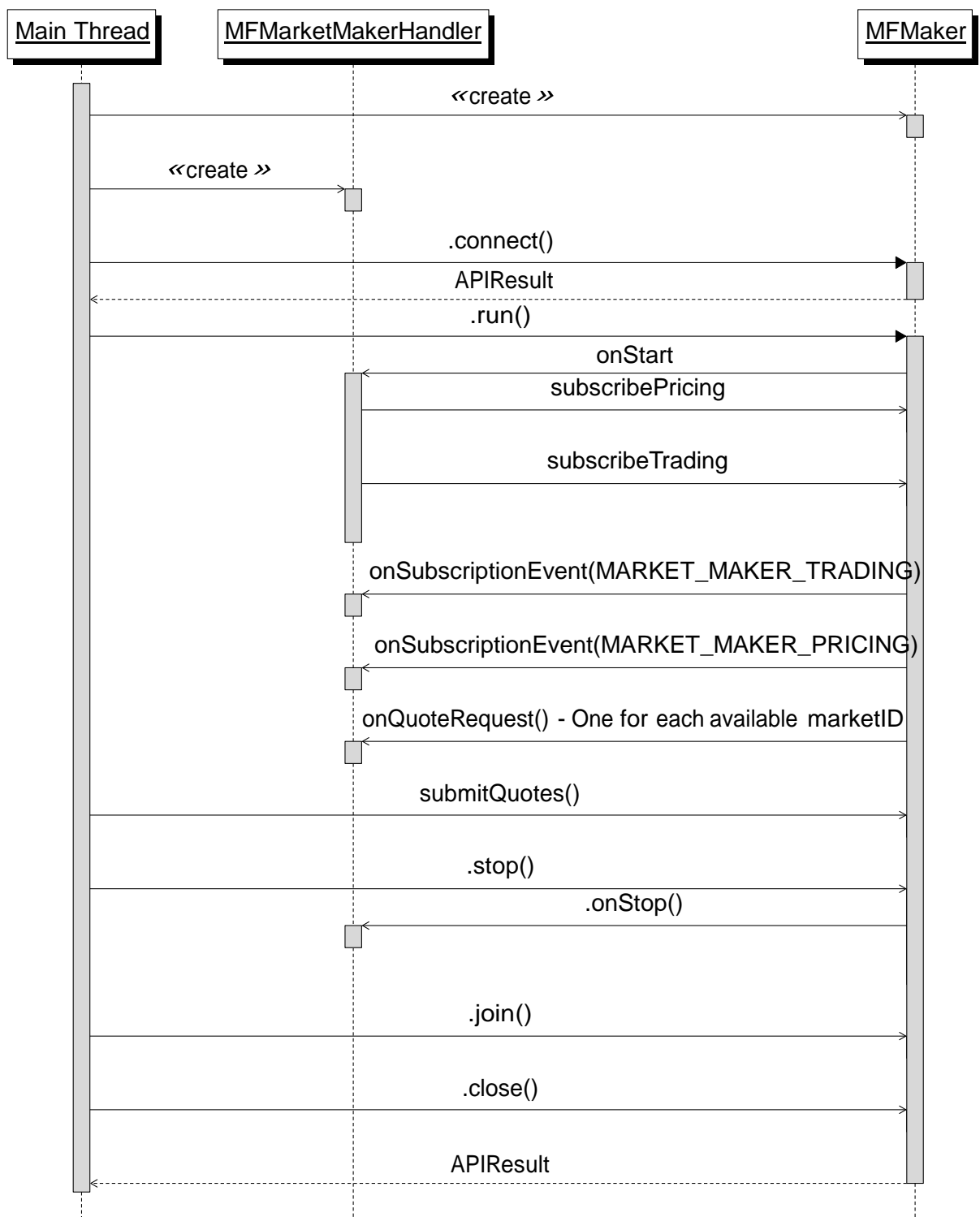
**cancelling quotes**

**application error**

**order rejection**

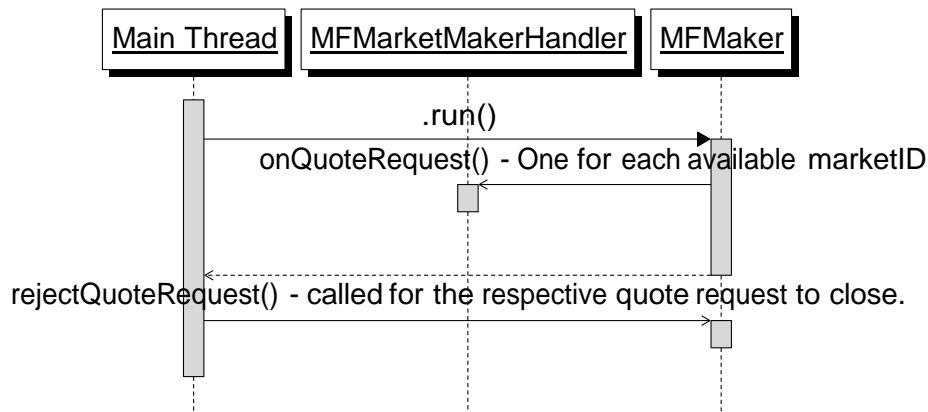
**rejected order acceptance**

**order timeout**

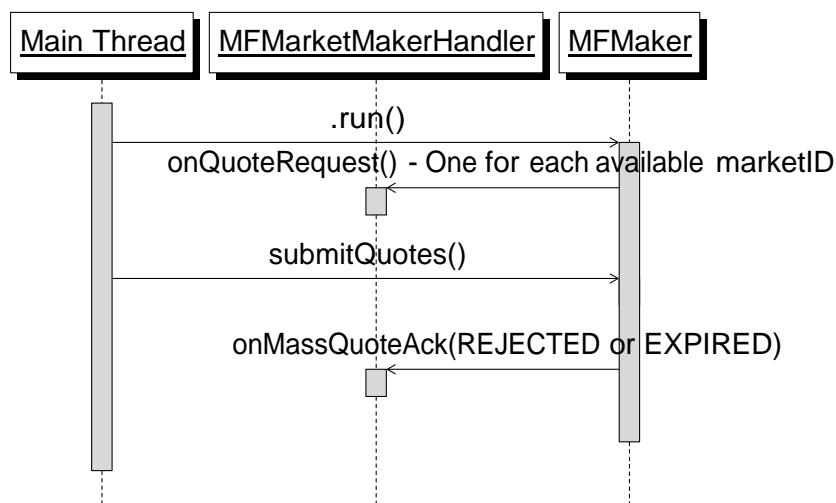


**Figure 9.1.** – Showing a normal flow of submit quotes through whisperer. Make sure both the trading and pricing connects are subscribed. Most venues require this as a precondition for them to send the quoteRequest messages to the maker.

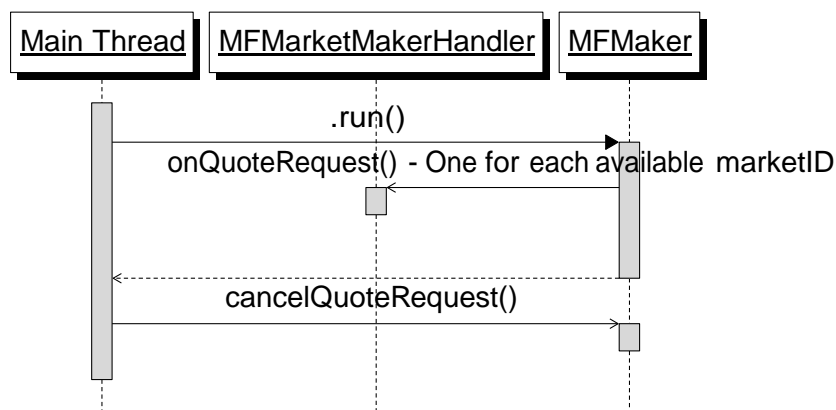




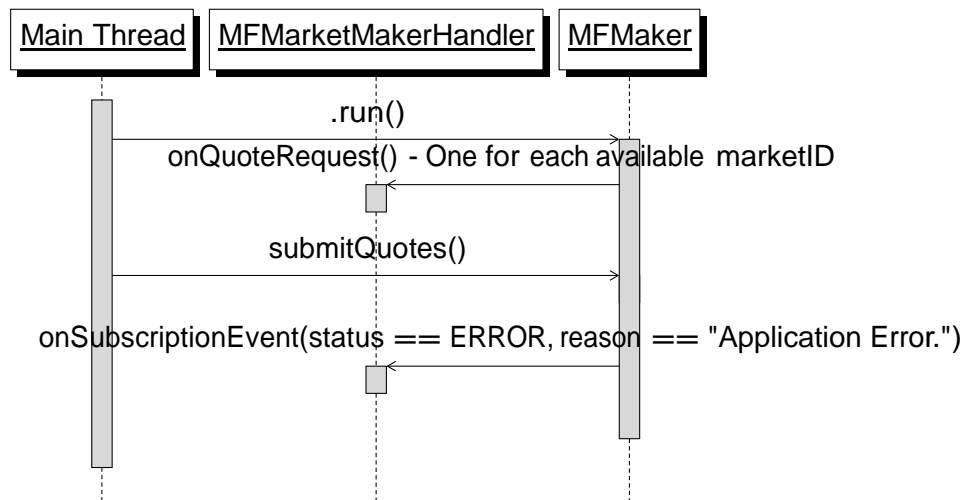
**Figure 9.2.** – A maker rejecting a quote request from the venue. This is for the case where a maker does not wish to price to an available stream. (Assuming a subscription to Pricing and Trading)



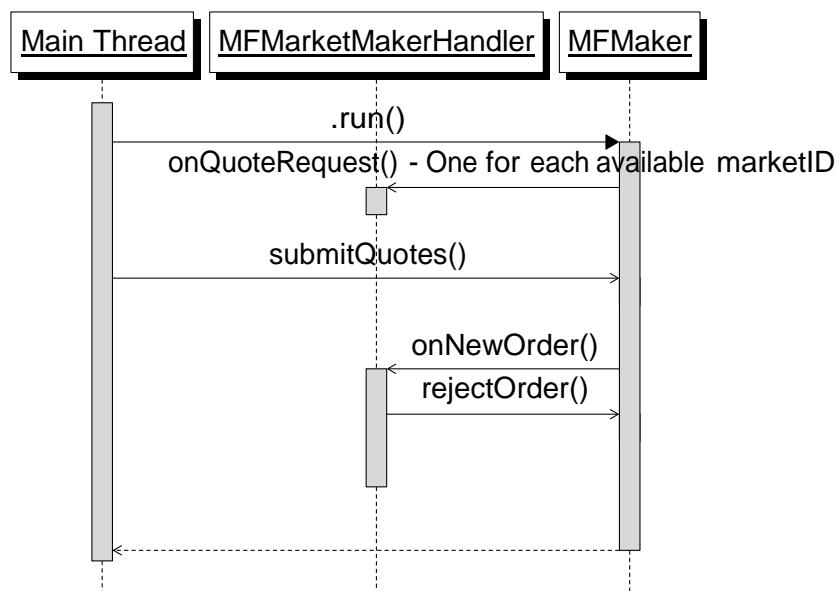
**Figure 9.3.** – The venue rejecting a quote request. This is for the case where the venue decides to close a stream. (Assuming a subscription to Pricing and Trading)



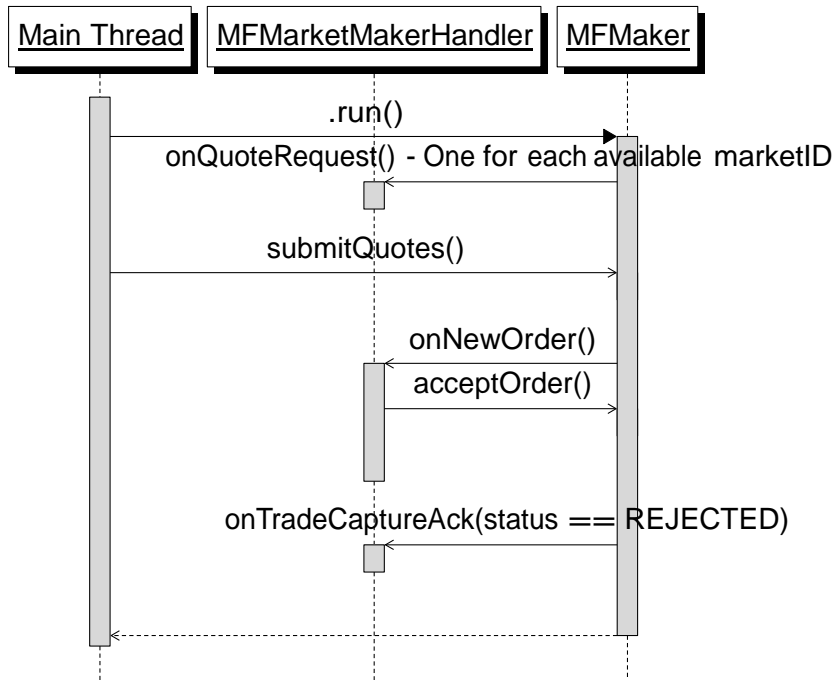
**Figure 9.4.** – A maker cancelling a quote request from the venue. This is for the case where a maker wants to cancel quotes on the venue. The options are either all live quotes, quotes for a particular symbol, or a particular quote. The supported options differ by venue. (Assuming a subscription to Pricing and Trading)



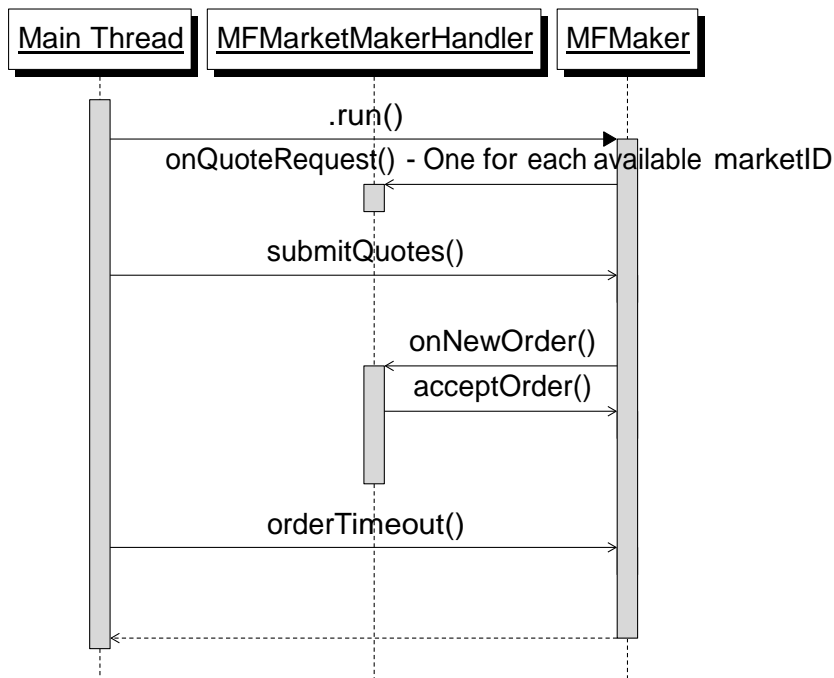
**Figure 9.5.** – This occurs when there is an unrecoverable error, such as a `BusinessMessageReject` being sent from the venue because of an invalid price (such as a negative price). This acts as a kill switch to get the maker out of the market by disconnecting the sessions since the error can not be programmatically recovered from. (Assuming a subscription to Pricing and Trading)



**Figure 9.6.** – A maker accepting a new order from the venue. When a client executes on an order, Whisperer will send a `NewOrder` to the maker for them to accept or reject. Rejecting an order will notify the exchange to not fill the order. (Assuming a subscription to Pricing and Trading)



**Figure 9.7.** – A maker accepting a new order from the venue but the venue rejecting the acceptance. This might happen, for instance, when a `acceptOrder` fails validation, due to a wrong `quoteID`. (Assuming a subscription to Pricing and Trading)



**Figure 9.8.** – A maker accepting a new order from the venue, but doing so too slow. This will generate an `OrderTimeout` message from the venue notifying the response time was too slow. This causes the fill to be rejected. (Assuming a subscription to Pricing and Trading)

## 9.7 venue-specific considerations

As described in the previous section, the general flow is the same among venues, but each venue has specific behaviors which is described here:

### 9.7.1 Currenex RFQ

- QuoteRequest: On most venues the liquidity provider (LP) will get quote requests immediately after subscribing. On Currenex RFQ however it would be received only after the taker requests pricing.
- Only one level could be streamed.
- Once order is accepted the stream is assumed canceled so the LP shouldn't stream prices on this stream after that.
- OrderTimeout is not supported.
- Order ID generation: When responding to a NewOrder, either with an accept or reject, the LP is expected to generate the orderID and the execID.
- Behavior on bad messages: On Currenex, if the LP enters in a bad quote, such as one with incorrect side, the LP will be unsubscribed with the text "Application Error" and will be disconnected.

### 9.7.2 FXAll

- QuoteRequest: When the LP receives an initial quote request from the venue, it may not have an amount filled. FXAll does not fill in this value, it is up to the LP to specify their own amount.
- Up to 3 levels in a stream.
- Order ID generation: On FXAll, the LP is expected to use the values that were filled in the NewOrder message.
- Behavior on bad messages: For FXAll, the failure is silent, there will be no price in the market.
- isBaseSpecified: For FXAll, this field is only applicable on the NewOrder. It is not applicable in the response messages (AcceptOrder, RejectOrder).

### 9.7.3 Order fields

The following table describes when specific order field should be present and who is responsible for filling in the value.

field	FXAll	Hotspot	Currenex RFQ	Currenex OUCH	Fastmatch	Integral	BB FXGo
execID	Venue	Venue	Maker	Maker	Maker	Maker	Maker
orderID	Venue	Venue	Maker	N/A	Maker	Maker	Maker
clOrdID	Venue (1)	Venue	Maker	Maker	Maker	Venue	Venue
tradeDate	Venue	Venue	Maker (2)	N/A	Maker	Maker	Maker
settlDate	Venue	Venue	Venue	Venue	Maker	Venue	Maker
quoteID	Maker	Maker	Maker	Maker(see footnote)	Maker	Maker	Maker

**Table 9.1.** – order fields

- (note this maps to quoteID)
- (Currenex would override this if incorrect)
- For Currenex OUCH, if the maker supplies numeric quoteID values these will be passed through to the exchange. Non-numeric quoteID values will be mapped to numeric ID values for presentation to Currenex and then mapped back to the original quoteID when returned from the venue.

### 9.7.4 TradeCapture - detail data element

The detail element of the trade capture message will contain a JSON object with the following optional elements

- Account - the value from tag 1 in the execution report.
- ContraTrader - the value from tag 375 occurrence 2 if it exists.
- ExecBroker - the value from tag 76.

### 9.7.5 Error conditions

#### FXAll

- If a MassQuote
- ContraTrader - the value from tag 375 occurrence 2 if it exists.

- ExecBroker - the value from tag 76.

## 9.8 terminology

**configuration** MarketFactory Support personnel configure and monitor Whisperer installations to individual customer requirements. This configuration includes:

- Trading Venues for which an instance of the Whisperer software application has credentials.
- Markets supported on each Trading Venue, and instrument-specific parameters.
- Market enhancement parameters and performance settings

The list of Trading Venues and the list of instruments available on each Trading Venue is available to the Customer Application via the MFAPI.

A “Feed” is a connection to a Trading Venue via the MFAPI, for example “HOTSPOT” or “CME”. A

“Market” is an instrument available on a Trading Venue, for example “EUR/USD” or “6EH5”. The

Whisperer application logs all events to binary log files. these logs are used by MarketFactory Support staff for troubleshooting, order history forensics, and reporting on market statistics.

### 9.8.1 Identifiers within the MFAPI

During the lifetime of an order, the order identifier as seen by the Client and Venue “clOrdID” may change as a result of cancel-and-replace operations. By contrast, the identifier used internally by Whisperer does not change.

`orderId` *integer* - MarketFactory internal order identifier

name	<code>orderId</code>
type	<i>integer</i>
description	MarketFactory internal order identifier A globally unique identifier for each Whisperer session, across all Customer Applications.
allocation	MFAPI
scope	globally unique across all clients during a session

`clOrdID` *string* - Client Order ID

Allocated by: Customer Application at order submission when he submits a new order

Scope: Unique per (deal code, Trading Venue) during a session

A string provided by the Customer Application. It should be unique per Feed, Customer Application, and Session. An error may be returned if two Customer Applications use the same clOrdID values to track different orders (however there is no guarantee that would happen because some venues just ignore duplicate orders). All Trading Actions and Events carry this field.

The Customer Application may leave that field empty and use only the MarketFactory Order Id described above everywhere except in the order submission.

To avoid problems, clOrdID values must be unique. To strike a balance between uniqueness and readability, a unique identifier may be constructed by the concatenation of company name, current timestamp, user id, and process id.

*exOrdID string* - Exchange Order ID

Type: Variable-length string

Allocated by: Trading Venue when the order is accepted by the former

Scope: Unique per Trading Venue

Typically it is a long string. It is propagated to all relevant Trading Events.

*exTradeID string* - Exchange Trade ID

Allocated by: Trading Venue TradeCapture is generated

Scope: Unique per Trading Venue

A string that uniquely identifies a TradeCapture event. The TradeCapture message also carries the clOrdID and exOrdID.

*cxIID string* - Cancel ID

Allocated by: Customer Application upon cancellation request

Scope: Unique per Trading Venue, Customer Application, and session

A string provided by the Customer Application to identify a cancellation request. clOrdID can be also used to track the status of all cancellation requests.





## Venue-specific notes

### 10.1 Bloomberg TradeBook

#### 10.1.1 notes

- a connection may have either partial or final fills set.

### 10.2 CME

#### 10.2.1 notes

- The CME allows the Customer to modify the order submitted. Price, Amount, Order Type, Time in Force, and StopPrice can be modified.

### 10.3 EBS

#### 10.3.1 TradeCapture and OrderDone messages received out-of-order

On rare occasions, the sequence of a TradeCapture event may straddle the end of an order. An OrderDone message may be received before the final TradeCapture. For example, this is a valid sequence of events:

MFClient	MFAPIServer
SubmitOrder (GTC)	→
	← OrderReceived (acknowledgement from MFAPIServer)
	← OrderSubmitted (acknowledgement from Venue)
	← TradeCapture <sub>1</sub> , matchStatus set to PENDING
	← TradeCapture <sub>1</sub> , matchStatus set to DONE
	← TradeCapture <sub>2</sub> , matchStatus set to PENDING
	← OrderDone
	← TradeCapture <sub>2</sub> , matchStatus set to DONE

**Table 10.1.** – GTC order with multiple fills

### 10.3.2 EBS Ai

- The market data updates are time-sliced; they are snapshots of the ECNs order book. Updates are sent every 250ms.
- An EXCHANGE\_BEST value is provided. It is translated from the BEST entry.
- DEALABLE and DEALABLEPLUS entries are translated to pure LEVEL entries. Where the two-level book contains no prices or amounts, we provide pure entries with zero-level amounts.
- REGULAR entry is translated to an amount constraint (AMOUNT\_CONS). It represents the price at which there is a cumulative amount of liquidity available; the specific cumulative amount depends on the currency pair.
- OUTSIDE entry is translated to a spread constraint (SPREAD\_CONS). It represents the cumulative amount of liquidity at a specific spread/offset from DEALABLE. Again, the specific offset depends on the currency pair.
- LOCAL entry is translated to the best local price (LOCAL\_BEST). It represents the best price placed on the same trading floor (under the same dealing code.)
- PAID entry is the worst (highest) offer hit in the last time interval (time-slice); it is mapped to the WORST\_TRADE element on the bid side.
- GIVEN entry is the worst (lowest) bid lifted in the last time interval (time-slice); it is mapped to the WORST\_TRADE element on the offer side.
- EBS is the only ECN that sends a TradeID value along with each trade capture. ExecID values are usually used to uniquely identify each fill. However in this case, TradeID values can also be used to uniquely identify fills, with the added benefit of also being the same identifier sent to the back office. Therefore, we have chosen to populate exTradeID field with TradeID instead of the execID.
- on EBS Ai, FIX tag 1003 (TradeID) maps to exTradeID, and FIX tag 17 (ExecID) maps to exPendingID. Some customers use exTradeID to reconcile with STP and some use exPendingID as well.
- PAID and GIVEN is mapped to the WORST\_TRADE and may be seen in the MktDataMessage example below, it is associated with the a specific timeExchange.

```
2015-12-10 17:00:04 172,131 <MktDataMessage (mvd=(fmID=(marketID=1 feedID=32)
timeExchange=1449766804171000000 timeArrival=1449766804171986828
```

```
timeHandler=1449766804172112422 timeEnhanced=0 timeApiServer=0 timeApiClient=0
isSnapshot=F mdEntryList=[(price=121450000000 amount=-9223372036854775805
updateAction=NEW element=WORST_TRADE side=BID source=1 numberOfOrders=-1
components="") (price=121445000000 amount=-9223372036854775805
updateAction=CHANGE element=EXCHANGE_BEST side=BID source=1 numberOfOrders=-1
components="") (price=121455000000 amount=0 updateAction=DELETE element=LEVEL
side=BID source=1 numberOfOrders=0 components="") (price=121450000000 amount=0
updateAction=DELETE
```

- On EBS Ai, timeExchange corresponds to sending time on the AI message. The value is always populated except when isSnapshot is true. In this case, MarketFactory generates the snapshot – it is based on the content of order book as we see it at that moment. This occurs when a new user connects or if a snapshot is requested.
- EBS Ai data is provided in “views”

Price Update Spread (for *EBS Market* pairs) Amount Full amount (for *EBS Markets* pairs)

### 10.3.3 EBS Live

- The market data updates are time-sliced; they are snapshots of the ECNs order book. Updates are sent every 100ms.
- ORDERBOOK entries are translated to pure LEVEL entries.
- EBS Live market data is not credit-screened. If you have an EBS Live feed, it is assumed that you have enough credit with your prime broker that the entries are actionable, and thus we treat those entries as credit-screened.
- REGULAR entry is translated to an amount constraint (AMOUNT\_CONS). It represents the price at which there is a cumulative amount of liquidity available; the specific cumulative amount depends on the currency pair (same as EBS Ai.)
- PAID and GIVEN fields are mapped in the same manner as EBS Ai.
- EBS Live market data contains two timestamps – msg\_time time holds the UTC time at which the EBS MRF Server generated the message. This is stored in timeExchange in the MarketFactory MktDataMessage.

upd\_time is the UTC time of the market event as recorded by the EBS system. This is stored in the components field.

```
2015-12-10 18:00:02 205,300 <MktDataMessage (mvd=(fmID=(marketID=1
```

```

feedID=44) timeExchange=1449770402204000000 timeArrival=1449770402205245050
timeHandler=1449770402205292693 timeEnhanced=0 timeApiServer=0
timeApiClient=0 isSnapshot=F mdEntryList=[(price=121455000000
amount=10000000000000000 updateAction=NEW element=LEVEL side=BID source=1
numberOfOrders=1 components="1449770402200000000") (price=121405000000 amount=0
updateAction=DELETE element=LEVEL side=BID source=0 numberOfOrders=0
components="") (price=121460000000 amount=-9223372036854775805 updateAction=NEW
element=WORST_TRADE side=OFR source=1 numberOfOrders=-1 components="")
(price=121460000000 amount=0 updateAction=DELETE element=LEVEL side=OFR
source=1 numberOfOrders=0 components="") (price=121465000000
amount=10000000000000000 updateAction=CHANGE element=LEVEL side=OFR source=1
numberOfOrders=1 components="1449770402200000000") (price=121510000000
amount=30000000000000000 updateAction=NEW element=LEVEL side=OFR source=1
numberOfOrders=3 components="1449770402200000000"))] fmIDsInBatch=[(marketID=1
feedID=44)] batchID=6500485 signalID=6500485)>

```

## 10.4 Hotspot

### 10.4.1 Hotspot venue-specifics

- Only pure entries are provided. We aggregate updates by price level.
- This feed provides streamed data and is not time-sliced.
- The Hotspot FIX market-data feed provides only full snapshots of the order book, while the Hotspot ITCH market-data feed provides streaming incremental updates. In both cases, we translate this into incremental updates to streamline processing.
- Hotspot also provides an unaggregated feed, which will be supported in a future version of the MFAPI.
- exTradeID contains the exchange-side TradeID (FIX tag 1003) as set by Hotspot.

## 10.5 Reuters

### 10.5.1 venue-specific notes

- An EXCHANGE\_BEST value is provided, and corresponds to an onMarketUpdate() callback in MAPI.

- The onScreenedMarketUpdate() callback is translated to a pure LEVEL entry.
- Reuters has a “Standard Quantity” which is sent via the onAggregateUpdate() callback. This is a price at which there is a cumulative amount of liquidity available; the specific amount depends on currency pair and is specific to Reuters. This is translated to an amount constraint.
- Reuters sends market data updates every 250ms. However, it is important to note that the amount constraint updates and the pure level updates do not occur in sync. The MarketFactory software keeps track of this internally and only forwards information that is new and consistent.
- The counterPartyId data element in the TradeCaptureMessage will contain a JSON object with the Party information.

```
{
  "PartyInfo": [ {
    "PartyID":"LONDON BANK E",
    "PartySubIDs": [ {
      "PartySubID":"BBAG",
      "PartySubIDType":"25"
    } ]
  } ]
}
```

PartyID = 1117 PartySubIDs.PartySubID = 1121 PartySubIDs.PartySubIDType = 1122



## Platform-specific notes

### 11.1 Java platform notes

#### 11.1.1 Java Zero-GC MFClient

- MarketFactory provides a Zero-GC Java MFClient. Using this client enables the program to not have garbage collection pauses (given that the program using the library is written in a way that also does not generate garbage). To use this, link to MFAPI-zerogc.jar. Implementing to this library is similar to implementing to the regular library, except for the following differences:
- Once the MFClient has established a connection, it will no longer allocate memory from the heap. If there is a disconnection, then heap memory will be allocated. Therefore, if too many disconnections occur, the process should be restarted in order to avoid a GC pause. (This is dependent on how the JVM is tuned).
- The message is reused after each callback to the class implementing MFHandler. Therefore, the program must make a copy of the data in the message if it wants to use the data after the callback returns to MFClient. Keeping a reference to any data will result in errors.
- A pool of recycled messages is allocated for each MFClient. Therefore there can be multiple MFClients per process. Only one thread can run the event loop of the MFClient by calling the `run(...)` method. During this time, multiple threads can trade independently (by calling `submit`, `modify`, and `cancel`) and still be thread-safe.
- All uses of `java.lang.String` are replaced with `MFString`. This is a mutable version of `java.lang.String`. Therefore, all mutating methods manipulate the actual class and do not generate copies.
- `MFFloat` is also a mutable class. Therefore, like `MFString`, all mutating methods manipulate the actual class and do not generate copies.
- Many of the classes have a `deepCopy()` method. In order to use this properly, you must allocate memory for each of the destination's member objects and arrays as the method does not do this for you. This is because it is assumed that the user needs to control these

memory allocations (usually through object pooling) in order to prevent garbage from being generated.



# SBE protocol type mappings

These type mapping tables are generated from the contents of the XML files found in  
\$/MFAPI/MFProto/src/main/sbe/.

## A.1 framing

Non-official sbe message. This comes before each message to indicate the length of the message. This should be used to iterate to each additional message in order to support the use case of the sender sending a newer version of the message (which has additional fields) but the decoder still being able to process without having to change any code.

name	id	type
MessageLength	20002	uint16

**Table A.1.** – MessageLength

## A.2 market data

### A.2.1 message types

name	id	description
Logon	2	Used to logon to a market data session.
LogonResponse	3	An acknowledgement from the server that the logon was successful as well as the expected heartbeat interval.
Logout	4	A request to stop the session.
LogoutResponse	5	An acknowledgement that the session has been terminated.
Heartbeat	6	This is sent from each side of the connection after x seconds of inactivity.
SecurityDefinitionRequest	7	A request for a list of a venue's securities and their attributes.
SecurityDefinition	8	Describes the attributes of a given security.
MarketDataRequest	9	A request to create, cancel, or otherwise manipulate a subscription for a single security from a specific venue.
MarketDataRequestReject	10	A rejection of a request for market data.
SecurityStatus	11	Contains details of a venue status, trade/settle dates, and more. Comes as a response to a MarketDataRequest and should be processed before the first market data message. Can also arrive asynchronously if the state of a security changes.
MarketDataIncrementalRefresh	0	First message, and possibly only, in a Market data update group. Contains common information about the updates to follow. Also contains, for the sake of compression, a single MDEntry update.
MDEntry	1	Part of a Market data update group and, if necessary, will always come after a MarketDataIncrementalRefresh message. Contains the details for an atomic change to the quote book.
RegisterForBatch	12	Register for an EBS-Ai batch.
UnregisterForBatch	13	Unregister for an EBS-Ai batch.
BatchesCompleted	14	Message sent when an EBS-Ai batch of market data updates is done.
HistoricMarketDataRequest	15	A request to stream historic market data

## A.2.2 message fields

#### HistoricMarketDataRequest

name	id	type
VenueID	20022	VenueID
SubscriptionRequestType	263	SubscriptionRequestType
SecurityID	48	SecurityID
EffectiveTime	168	Timestamp
ValidUntilTime	62	Timestamp

**Table A.2.** – A request to stream historic market data

### A.2.3 trading

**Table A.3.** – Logon

name	id	type	description
Username	533	Username	Id given to user for logging on.
Password	554	Password	Password used to logon.

Used to logon to a trading data session.

**Table A.4.** – LogonResponse

name	id	type	description
------	----	------	-------------

An acknowledgement from the server that the logon was successful as well as the expected heartbeat interval.

**Table A.5.** – ResendRequest

name	id	type	description
VenueID	20022	VenueID	Numeric ID of a given venue.
BeginSeqNo	7	SeqNum	unknown
Count	20018	uint32	Number of messages, starting at the BeginSeqNo, to resend.

A request to replay messages for a given venue

**Table A.6.** – SequenceResetGapFill

name	id	type	description
VenueID	20022	VenueID	Numeric ID of a given venue.
NewSeqNo	36	SeqNum	unknown

A request to replay messages for a given venue

**Table A.7.** – TradingSessionStatusRequest

name	id	type	description
SubscriptionRequestType	263	SubscriptionRequestType	unknown
VenueID	20022	VenueID	Numeric ID of a given venue.
NextExpectedMsgSeqNum	789	SeqNum	unknown

A request to subscribe to a venue in order to begin trading.

**Table A.8.** – TradingSessionStatus

name	id	type	description
VenueID	20022	VenueID	Numeric ID of a given venue.
TradingSubscriptionStatus	20020	TradingSubscriptionStatus	unknown
TradSesStatus	263	TradSesStatus	unknown
Text	58	Text	Explanation of status state, if any.

A response to a TradingSessionStatusRequest indicating the state of the venue. This event will be sent if the state of the venue changes.

**Table A.9.** – Heartbeat

name	id	type	description
VenueID	20022	VenueID	Numeric ID of a given venue.
CurrentSeqNum	20019	SeqNum	This is the last sequence number sent. Used to detect message drop

This is sent from each side of the connection periodically.

**Table A.10.** – Logout

name	id	type	description
Text	58	Text	Free format text string. May include reason for logout.

A request to stop the session.

**Table A.11.** – LogoutResponse

name	id	type	description
------	----	------	-------------

An acknowledgement that the session has been terminated.

**Table A.12.** – MarketDefinitionRequest

name	id	type	description
SubscriptionRequestType	263	SubscriptionRequestType	unknown

Request for a definition of the venue.

**Table A.13.** – MarketDefinition

name	id	type	description
VenueID	20022	VenueID	Numeric ID of a given venue.
MarketID	1301	Text	General name of the Market/Value. EBS-Ai for example.
VenueName	20024	Text	MF defined market/venue name. Used to identify a special, specific MF config

Contains attributes of the given venue.

**Table A.14.** – NewOrderSingle

name	id	type	description
MsgSeqNum	34	SeqNum	Currently not used.
PossDupFlag	43	Boolean	unknown
VenueID	20022	VenueID	Numeric ID of a given venue.
ClOrdID	11	ClOrdIDString	Unique identifier for Order as assigned by the buy-side.
SecurityID	48	SecurityID	Numeric ID of a given security.
Side	54	Side	unknown
TransactTime	60	Timestamp	Time when the order is generated.
OrderQty	38	DecimalQtyNULL	Required. Order quantity in mantissa format.
MinQty	110	DecimalQtyNULL	Minimum quantity to execute. Used to prevent partial fills under th
OrdType	40	OrdType	unknown
Price	44	PriceNULL	Order limit price. Not required on all order types (for example, Ma
StopPx	99	PriceNULL	Stop price. Only required if this order is a Stop-type order. Must be
TimeInForce	59	TimeInForce	unknown
MaxShow	210	DecimalQtyNULL	Where available, sets the maximum amount to show. Must be in m

Message for submitting a single order.

**Table A.15.** – OrderCancelRequest

name	id	type	description
MsgSeqNum	34	SeqNum	Currently not used.
PossDupFlag	43	Boolean	unknown
VenueID	20022	VenueID	Numeric ID of a given venue.
SecurityID	48	SecurityID	Numeric ID of a given security.
ClOrdID	11	ClOrdIDString	Identifier of this cancel request.
OrigClOrdID	41	ClOrdIDString	ClOrdID of the order to cancel.
OrderID	37	OrderID	The MF assigned orderID of the order to cancel. Can set to null.
TransactTime	60	Timestamp	Time when the cancel is generated.

Cancel an outstanding order.

**Table A.16.** – OrderCancelReject

name	id	type	description
MsgSeqNum	34	SeqNum	unknown
PossDupFlag	43	Boolean	True if this message could be a duplicate. If true, the hand
VenueID	20022	VenueID	Numeric ID of a given venue.
SecurityID	48	SecurityID	Numeric ID of a given security.
ClOrdID	11	ClOrdIDString	The ClOrdID of the Cancel request.
OrigClOrdID	41	ClOrdIDString	The ClOrdID of the order to be canceled
OrderID	37	OrderID	The MF assigned orderID of the order to cancel.
SecondaryOrderID	198	OrderIDString	The OrderID assigned by the venue.
OrdStatus	39	OrdStatus	unknown
CxlRejReason	102	CxlRejReason	unknown
TransactTime	60	Timestamp	Time when the cancel is generated.
TimeArrival	20008	Timestamp	Time when the execution report is received by MF.
Text	58	Text	Free format text string. May include extra information abo

Response order was not able to be canceled.

**Table A.17.** – ExecutionReport

name	id	type	description
MsgSeqNum	34	SeqNum	unknown
PossDupFlag	43	Boolean	unknown
VenueID	20022	VenueID	Numeric ID of a given venue.
SecurityID	48	SecurityID	Numeric ID of a given security.
ClOrdID	11	OrderIDString	The current ClOrdID of the order. Can differ from the original value submitted on the NewOrder.
OrigClOrdID	41	OrderIDString	The ClOrdID of the original value submitted on the NewOrder.
OrderID	37	OrderID	The MF assigned orderID.
SecondaryOrderID	198	OrderIDString	The OrderID assigned by the venue.
LastQty	32	DecimalQtyNULL	Quantity of order executed in this execution, if any.
LastPx	31	PriceNULL	Price of this last fill, if any.
OrdType	40	OrdType	unknown
TimeInForce	59	TimeInForce	unknown
Side	54	Side	unknown
LeavesQty	151	DecimalQtyNULL	Quantity remaining to execute on this order. After a system cancel, the quantity is zero.
ExecID	17	OrderIDString	Unique identifier for this execution report.
ExecType	150	ExecType	unknown
TradeID	1003	OrderIDString	TradeID as assigned by EBSAi.
OrdStatus	39	OrdStatus	unknown
TradeDate	75	LocalMktDate	Trade date.
SettlDate	64	LocalMktDate	Settlement date.
CounterPartyID	20023	OrderIDString	Counterparty ID
TransactTime	60	Timestamp	Time when the execution report was generated.
TimeArrival	20008	Timestamp	Time when the execution report was received by MF.
Text	58	Text	Free format text string. May include extra information about the order.
OrigVenueID	20025	VenueID	VenueID of the source of this execution report. Used when placing orders.

A message for responses from the venue for order actions and events.

**Table A.18.** – OrderCancelReplaceRequest

name	id	type	description
MsgSeqNum	34	SeqNum	Currently not used.
PossDupFlag	43	Boolean	unknown
VenueID	20022	VenueID	Numeric ID of a given venue.
SecurityID	48	SecurityID	Numeric ID of a given security.
ClOrdID	11	ClOrdIDString	Identifier of this request.
OrigClOrdID	41	ClOrdIDString	ClOrdID of the order to modify.
OrderID	37	OrderID	The MF assigned orderID of the order to cancel. Can set to null. It is the original orderID.
Side	54	Side	The original order side.
TransactTime	60	Timestamp	Time when the cancel is generated.
OrderQty	38	DecimalQtyNULL	The new order quantity, if changing, or the original order quantity if not.
OrdType	40	OrdType	unknown
Price	44	PriceNULL	The new order price, if changing, or the original price if not.
TimeInForce	59	TimeInForce	The original order time in force.





# MFAPI / SBE message field mappings

As might be expected, the range of types used differs between the two systems.

This chapter contains field translations per message. This is the mechanism by which a Whisperer server allows connections from both classic MFAPI and SBE-enabled clients.

## B.0.1 type mappings

## B.0.2 message field mappings – trading messages

SubmitOrderMessage	NewOrderSingle
feedID	venueID
marketID	securityID
orderID	?
clOrdID	clOrdID stopPrice
	MFloat(stopPx)
maxShow	MFloat(maxShow)
timeInForce	convert(im.timeInForce)
amount	convertQtyToMFloat(im.orderQty.mantissa())
price	convertPriceToMFloat(im.price.mantissa())
side	convert(im.side)
ordType	convert(im.ordType)
timeApiClient	transactTime

**Table B.1.** – SubmitOrderMessage

CancelOrderMessage	OrderCancelRequest
userID	0
feedID	venueID
marketID	securityID
clOrdID	clOrdID
cxlID	clOrdID
origClOrdID	origClOrdID
clOrdID	origClOrdID
timeApiClient	transactTime
orderID	orderID

**Table B.2.** – SubmitOrderMessage

ModifyOrderMessage	OrderCancelReplaceRequest
userID	0
feedID	venueID
marketID	securityID
clOrdID	origClOrdID
clNewID	clOrdID
timeInForce	timeInForce
amount	orderQty
price	price
side	side ordType
	ordType
timeApiClient	transactTime
orderID	orderID

**Table B.3.** – ModifyOrderMessage

OrdType (SBE)	OrderType (classic MFAPI)
OrdType::Market	OrderType::MARKET
OrdType::Limit	OrderType::LIMIT
OrdType::StopLoss	OrderType::STOP
OrdType::StopLimit	OrderType::STOPLIMIT
OrdType::PreviouslyQuoted	OrderType::PREVIOUSLY_QUOTED
OrdType::PreviouslyQuotedAmountTier	OrderType::PREVIOUSLY_QUOTED_AMT_TIER
OrdType::VwapSweep	OrderType::VWAP_SWEEP
OrdType::MarketWithLeftOverAsLimit	OrderType::MARKETLIMIT

**Table B.4.** – Order type identifier mapping

mftrading::Side (SBE)	MarketFactory::Side (classic MFAPI)
mftrading::Side::Buy	MarketFactory::Side::BID
mftrading::Side::Sell	MarketFactory::Side::OFR

**Table B.5.** – Side type identifier mapping

mftrading::TimeInForce (SBE)	MarketFactory::TimeInForce (classic MFAPI)
mftrading::TimeInForce::Day	MarketFactory::TimeInForce::DAY
mftrading::TimeInForce::GTC	MarketFactory::TimeInForce::GTC
mftrading::TimeInForce::IOC	MarketFactory::TimeInForce::IOC
mftrading::TimeInForce::FOK	MarketFactory::TimeInForce::FOK

**Table B.6.** – Time-in-force identifier mapping

mftrading::TimeInForce (SBE)	MarketFactory::TimeInForce (classic MFAPI)
mftrading::TimeInForce::Day	MarketFactory::TimeInForce::DAY
mftrading::TimeInForce::GTC	MarketFactory::TimeInForce::GTC
mftrading::TimeInForce::IOC	MarketFactory::TimeInForce::IOC
mftrading::TimeInForce::FOK	MarketFactory::TimeInForce::FOK

**Table B.7.** – Time-in-force identifier mapping

B.0.3 message field mappings – market data messages

MarketDataIncrementalRefresh (SBE)	MktDataMessage (classic MFAPI)
	isSnapshot=TRUE
MDEntry <sub>0</sub>	MDEntry
...	
MDEntry <sub>N</sub> (lastMDEntryInGroup set to true.)	

Table B.8. – market data – snapshot

In this case, the mapping is not one-to-one between messages. In SBE, a snapshot is represented as a MarketDataIncrementalRefresh followed by multiple MDEntry messages, the final one having the flag lastMDEntryInGroup set to true.

MarketDataIncrementalRefresh (SBE)	MktDataMessage (classic MFAPI)
	isSnapshot=FALSE
MDEntry <sub>0</sub>	
...	
MDEntry <sub>N</sub>	

Table B.9. – market data – incremental

