FeedOS

The Market Data Operating System

API User Guide & Reference Guide

FeedOS v3.9.1.2 2011-07-28

FeedOS API User Guide

Java client API

USER AND REFERENCE GUIDE

FeedOS Java client API

Index

Overview

This document gives an overview of FeedOS Java client API.

Pre-requisite reading

Users are advised to read chapter 3 "FeedOS Overview" in document "FeedOS API – C++ User Guide". It explains most of FeedOS Market Data features and concepts.

Features available in Java API

The API is Object-Oriented. Many classes are provided to perform well-known actions on a market data server:

- connecting to a remote server
- sending simple requests (synchronously or asynchronous)
- starting subscriptions (asynchronous reception of real-time data)
- utility stuff: initialisation, conversions, constant values, etc

Initialize the API

How to initialise & terminate the API

Initialisation

Prior utilizing the API functions, you must call:

```
Session.init_api ("sample application")
```

The parameter is the friendly name of the user application.

Termination

When you are done using the API, you should call:

```
Session.shutdown_api ()
```

Connecting to a server

How to connect to a remote server

Session object

In order to communicate with a FeedOS server, you have to instantiate an object of class `com.feedos.api.core.Session`

You may instantiate as many objects as needed, if reaching several servers at the same time makes sense.

Initiating the connection

You have to call method **open**:

int open (SessionObserver observer, ProxyFeedos feedosProxy, long timeout)

- *Observer* : an object that receives signals corresponding to session events (open,heartbeats,close)
- *feedosProxy* : the connection proxy, it could be:
  - via a socket session:
    ProxyFeedosIPC (String serverDirectory, Credentials userCredential)
    - *userCredential* : this class implements the user login and the user password
    - *serverDirectory* : it refers to the feed handler installation folder
  - via a pipe communication:
    ProxyFeedosTCP (String feedosServerAddress, int feedosServerPort, Credentials userCredential)
    - *userCredential* : this class implements the user login and the user password
- *timeout* : connection timeout in milliseconds

This call is synchronous. The return code tells if the connection was successful.

Terminating the connection

You have to call **close()** on the target Session object.

Features available

A large set of requests (either simple ones or subscriptions) are available.

Classification and naming of Requests

Requests are grouped in « Services » (family of requests that provide functions for a given area) so the name of a request is really made of two pieces :

1. the « Service » (family of similar requests)
2. the « Request » itself

This will often result in names like « *SERVICE_REQUEST* ». Most often, no two requests with the same name coexist in different services, so we usually abbreviate the name of a given Request by forgetting the Service it belongs to.

For clarity sake, the name of services will often be abbreviated to a few letters.

The main Services available

Almost all the functions a user will ever need to process market data can be sorted in two « Services ».

REFERENTIAL (abbreviated as `ref`)

This consists of managing the referential data (markets and instruments). Also called « static data » or « dictionnary » or « fundamental characteristics ».

Users can retrieve the list of markets supported by the remote server, their contents (types and quantities of instruments) and the definition of instruments. A « lookup » function is provided to dynamically search instruments based on a pattern string, with optional filtering.

QUOTATION (abbreviated as `quot`)

This consists of retrieving price-related data: trades, best quotes, Open / Close / High / Low daily values, etc. Snapshot, Historical and real time Subscriptions are available.

Issueing Simple Requests

These are simple Request – Response dialogs.
They can be issued asynchronously (for multithreaded / parallel operations) or synchronously (simple call of a blocking function).

Asynchronous Operations

The user-defined Receiver class

User has to build a « receiver » class to handle incoming data. You can do so by deriving an interface that defines the necessary (unique) callback to implement:

```
com.feedos.api.requests.Receiver_Service_Request
```

The callback method to implement has the following signature:

```
void svcRequestResponse (Object user_context,
                         int rc,
                         output parameters...)
```

The (optional) *user_context* object is exactly what was passed by the user when the request was issued. It allows to easily correlate each request and its corresponding response.

The *returnc code* indicates if the request was successful. When it is the case, the remaining *parameters* contain valid data that can be processed.

The RequestSender class: Simple Requests

It is provided by the API in package `com.feedos.api.requests`

Users have to instantiate an object of this class, and pass it a *Session* object pointing to the target server, along with a timeout (in seconds):

```
RequestSender (target_server, timeout)
```

Each available function in the API appears as a method. Starting an asynchronous operation is done by calling such method, which has the following signature:

```
void asyncSvcRequest (Receiver_Service_Request receiver,
                      Object user_context,
                      intput parameters...)
```

The callback of the *receiver* object will be used to handle the results (either an error code in case of failure, or output data in case of success).

The (optional) *user_context* object will be passed as-is to the receiver callback.

The *input parameters* (if any) are specific to the given request.

Using a message queue for asynchronous call-back

The default behaviour of the Java API is to provide synchronous processing between the socket and the call-backs. It provides the best performance.

However, since the FeedOS Java API 3.8.3, it is possible to ask the request sender for a low level message queue:

```
RequestSender (target_server, timeout, msg_queue_size)
```

The first thread will read the socket and then push the low level FeedOS message to a Java *ArrayBlockingQueue*. Then a second thread will get the messages from the queue and then dispatch the message to the corresponding call-backs.

The number of message in the queue can be retrieve with:

```
RequestSender.getPduQueueCapacity()
```

The asynchronous processing provides lower performance than synchronous processing but it helps the client to keep connected when high throughput happens (during an exchange opening by example).

When the request sender is not used anymore, it has to be shutdown:

```
RequestSender.terminate()
```

Synchronous operations

The SyncRequestSender class

This class is provided by the API in package `com.feedos.api.requests`

Users have to instantiate an object of this class, and pass it a *Session* object pointing to the target server, along with a timeout (in seconds):

```
SyncRequestSender (target_server, timeout)
```

Each Simple Request available in the API appear as a method. Issueing a synchronous operation is done by calling such method, which has the following signature:

```
output_parameter syncSvcRequest (intput parameters...)
```

In case of error, an exception is thrown (class **`FeedOSException`** defined in package `com.feedos.api.core`).

In case of success, the *output_parameter* contains the resulting data (if any).

Using Asynchronous Subscriptions

These are subscribe-publish dialogs. Most of the time this is used to send realtime data but it may be used to send large volumes of information as a flow of small data chunks.

The user-defined Receiver class

User has to build a « receiver » class to handle incoming data. You can do so by deriving an interface that defines the necessary callbacks to implement:

```
com.feedos.api.requests.Receiver_Service_Request
```

There are 3 kinds of callback methods to implement.

The Response callback

It is the first callback triggered. In case of failure, it is also the last event that will occur regarding the requested subscription.

In case of succes, user should process the output parameters (if any), and expect to receive subsequent notifications that carry more (unsolicited) data.

```
void svcRequestResponse (int subscription_num,
                         Object user_context,
                         int rc,
                         output parameters...)
```

The arguments are pretty much the same as for the Simple Requests.

The *subscription_num* is the value that was returned by the API when the request was started. This can be used by the user to correlate asynchronous responses with the subscription they belong to. In this context, this parameter can be seen as a lightweight equivalent for *user_context*.

The UnsubNotif callback

This callback is triggered when the subscription is aborted by the server. When it happens, it is also the last event that will occur regarding the subscription.

It has the following signature:

```
void svcRequestUnsubNotif (int subscription_num,
                           Object user_context,
                           int rc)
```

The return code gives the reason why the subscription was aborted.

The notification callbacks

These callbacks (the number depends on the kind of subscription) are triggered whenever an event occur regarding the ongoing subscription.

They have the following kind of signature:

```
void svcRequestNotifEvent (int subscription_num,
                           Object user_context,
                           event parameters...)
```

Event is the name of the event that occurred.

The RequestSender class: Subscriptions

This class is provided by the API in package `com.feedos.api.requests`

Users have to instantiate an object of this class, and pass it a *Session* object pointing to the target server, along with a timeout (in seconds):

```
RequestSender (target_server, timeout)
```

Starting the subscription

Each subscription available in the API appear as a method. Starting a subscription is done by calling such method, which has the following signature:

```
int asyncSvcRequest_start (    receiver,
                               user_context,
                               intput parameters...)
```

The various callbacks of the *receiver* object will be used to handle results:

- initial response (either an error code in case of failure, or output data in case of success)
- eventually (and in case of success or course) the subsequent notifications, one per event that occurs.
- ultimately the « UnsubNotif » to signal that the subscription has been aborted by the server

The value returned is a unique identifier for this subscription. It will be passed to receiver callbacks as the *subscription_num*. This identifier is unique for a given instance of the API, even across different *RequestSender* and *Session* objects.

Stopping the subscription

Of course a user can decide to terminate an ongoing subscription. This is done by calling a « stop » method which is the counterpart of the « start » method described above. This method has the following signature:

```
void asyncSvcRequest_stop (subscription_num)
```

This method always succeed. No callback is triggered.

No more callback will be triggered for this subscription (although one could still be running in a different thread). You may want to implement some synchronization to enforce that a possibly running callback terminate before you can assume this subscription is effectively « off ».

Reference Guide

list of Simple Requests and Subscriptions available.

Service REFERENTIAL

REFERENTIAL Simple Requests

dumpStructure
Retrieve the « structure » of the referential data. This is made of a list of markets, a list of « branches », and number of known instruments.

A branch is a kind of containter for instruments of a given type (tags CFICode and SecurityType) in a given market (tag FOSMarketId).

Arguments: none.

getInstruments
Retrieve the characteristics of instruments.

Arguments: a list of instrument codes + an optional list of tags to retrieve.

lookup
Dynamically search for one or a small list of instruments.

Arguments: a pattern + optional filter and an optional list of tags to retrieve.

getVariableIncrementBandTable

Get the dynamic tick size tables for all the exchanges listed on a single server.

Arguments: non.

REFERENTIAL Subscriptions

download
Retrieval branches. More efficient and powerful than *lookup*.

Arguments: branch ids, optional timestamp for incremental download, optional filter.

downloadAndSubscribe
Retrieval branches. More efficient and powerful than *lookup*.

Arguments: branch ids, optional timestamp for incremental download, optional filter.

Service QUOTATION

QUOTATION Simple Requests

snapshotInstruments

Retrieve the « Level 1 » current values for one or several instruments.

Arguments: list of instrument codes, a list of tags to retrieved.

getHistoryDaily

Retrieve the Open/Close/High/Low daily values for a given instrument, over a specified period of time.

Arguments: instrument code, begin date + end date

getHistoryIntraday2

Retrieve the trade's daily values for a given instrument, over a specified period of time.

Arguments: instrument code, begin timestamp + end timestamp + number of « points » to retrieve (0 means « all », else a sampling is done).

getTradeConditionsDictionary

Retrieve the trade conditions dictionary.

Arguments: none.

QUOTATION Subscriptions

subscribeInstrumentsL1

Subscribe to « Level 1 » real-time data for one or several instruments.

Arguments: list of instrument codes, « content mask » to filter events.

subscribeInstrumentsMBL

Subscribe to « Extended Market By Level» real-time data with multi-layer order book composed of price, cumulated quantity and number of orders, for one or several instruments (continuous updates)

Arguments: list of instrument code, array of OrderBookLayerId.

subscribeOneInstrumentMarketSheet

Subscribe to the market sheet real-time data for one instrument.

Arguments: instrument code

download

Retrieval quotation data and referential data on branches

Arguments: branch ids, optional timestamp for incremental download, optional filter.

subscribeAllStatus
Subscribe to the market news.

Arguments: none.

Connection Subscriptions

subscribeFeedStatus

Subscribe to information and news about the feed state.

Arguments: none.