FeedOS

The Market Data Operating System

API User Guide & Reference Guide

FeedOS API v3.7.2 2011-07-28

FeedOS API – User Guide

C++ client API

USER AND REFERENCE GUIDE

FeedOS C++ Client API

Index

Chapter 1

Document History

Date	Author	Action
2006-02-30	D.Fenouil	Initial version of FeedOS v3
2006-10-02	S.Guignot	Support for MarketSheet subscription
2006-12-19	D.Fenouil	New request REFERENTIAL_Download
2007-10-29	S.Guignot	Update interface of MarketSheet subscription
2008-02-12	D.Fenouil	Feed replay: support for "mixed L1/L2" and "Intraday Bars"
2008-04-25	D.Fenouil	Minor updates
2008-09-05	D.Fenouil	Tags PreviousDailySettlementPrice, ReutersInstrumentCode, BloombergTicker and various tags for Turquoise, OMX, BPIPE
2009-06-12	D.Fenouil	New notification ValuesUpdateOneInstrument in request SubscribeOneInstrumentMarketSheet. Tag PriceIncrement_dynamic_TableId and New request REFERENTIAL_GetVariablePriceIncrementTables
2009-07-15	D.Fenouil	Various new tags: LastAuctionPrice, LastAuctionVolume, StdMaturity and several market-specific ones
2011-06-16	O.Prodhomme	New requests QUOTATION_GetHistoryIntraday2, REFERENTIAL_GetTradeConditionsDictionary, QUOTATION_SubscribeInstrumentsMBL, REFERENTIAL_DownloadAndSubscribe and threading schemes documentation reworked

Overview

This document describes how to install and how to use the C++ Client API for FeedOS.

About this Manual

This document provides a short description of the concepts behind the FeedOS system, and explains how to use the OO C++ Application Programming Interface (classes, functions, types and values) to remotely access a FeedOS server.

This document describes:

- How to install the C++ API
- The main concepts involved in the FeedOS system.
- General guidelines and usage information
- Brief description of all API functions: behavior, input and output parameters, typical usage.

Audience

This document is applicable to any person who plans to develop software that need to interact with a FeedOS system. Good C++ skill is required (including utilization of the Standard Template Library).

Typographic Convention

- Sample `source code` or `file names` are in fixed font
- **Concepts** appears in bold
- *Placeholders* are in italic

Technical Support

In case of question you can contact technical support by e-mail at support@quanthouse.com

Chapter 2

Installing the API

This section describes how to install C++ API, and which configuration is needed.

Prerequisite

C++ API is available for the following platforms:

- Windows: various combinations of Windows 32/64bit and Visual Studio 2003, 2005 or 2008.
- Unix: various combinations of Linux/Solaris10/FreeBSD/MacOS X and GCC 3 & 4.

Other versions could be made available. Please send your request to support@quanthouse.com .

How to obtain the API

If you are a registered client, go to http://www.quanthouse.com/download .

Else contact us through sales@quanthouse.com or http://www.quanthouse.com .

How to install the API

The Win32 C++ Client API is provided as archives named something like:

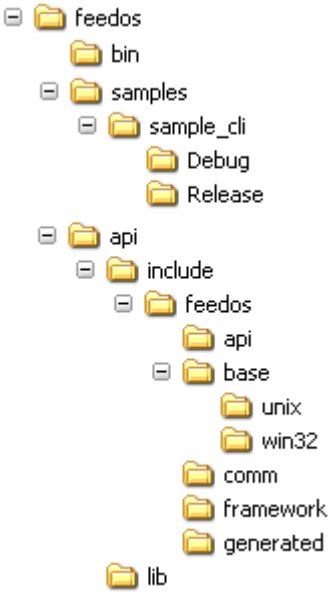
"`feedos_api_`***`version_platform_date.suffix`***"

Examples:

- feedos_api_v3.7.2.1_LINUX_x86_64_gcc4_2009-07-16.tgz
- feedos_api_v3.7.2.1_Windows_x86_vs9_2009-07-16.exe

Unzip the archive somewhere on your hard drive (e.g. `C:/FEEDOS`).

The layout of files is as follows:



How to upgrade the API

Simply unzip the new package over the old one.
Take a look at **`feedos/release_notes.txt`**

Uninstalling the API

As installation didn't alter special directories or the win32 Registry, simply delete the directory where you extracted the archive.

Chapter 3

FeedOS Overview

This section explains the main concepts used within FeedOS. It describes how to interact with a FeedOS server. A brief description of the available features is given.

Mains concepts

Adoption of international standards

Whenever possible, existing (and widely used) standards have been adopted within FeedOS.

The FIX protocol is the reference for providing functions or formatting data.
See http://fixprotocol.org

Universal Time Coordinates are used to represent dates and times.

The TZ database is used to represent time zones
See http://www.twinsun.com/tz/tz-link.htm

ISO 10383 (Market Identifier Code) is the preferred way for identifying financial **Markets**.
See http://www.iso15022.org/MIC/homepageMIC.htm

ISO 10962 is the preferred way for identifying financial **Instruments** (along with the old **Security Type** information from FIX).
See http://www.anna-web.com/ISO 10962/iso10962.htm

The same apply for countries (ISO 3166) or currencies (ISO 4217).

Identification of Financial Markets

As previously told, markets are identified using the ISO 10383 "MIC" standard. However, this is only the way that such identifiers are *entered by the user*, or *displayed* to him/her. They are actually converted into numeric (proprietary) identifiers in order to be processed internally. Two functions are provided to convert back and forth between textual values (like `XEUR`, `XLIF`, `XCBT`) and their internal counterparts.
See file `feedos/comm/FOSMarketId.h`

Identification of Financial Instruments

FeedOS provides two ways of identifying instruments. They are both concurrent and complementary.

Internal Code

This is a numeric value, used internally to uniquely identify an instrument. This is the recommended way of referencing an instrument because:

- Consistency: it is independent of whatever changes might occur on instrument's characteristics
- Performance: it is less consuming than other string-based schemes.
- Precision: it allows for easy unambiguous reference (unlike symbol-based schemes)

This numeric value is actually made of two parts: the most significant bits contain the parent market id, and the other bits are the unique identifier within this market.

Local Code

This is a proprietary market-dependent string, computed from a subset of instrument's characteristics (symbol, expiry date, etc). The syntax for the textual representation of this kind of instrument codes is as follows:

1. *market id* the parent market id, in textual form. Stored in tag `FOSMarketId`.
2. @ used as a delimiter
3. *LocalCodeStr* identifier for the instrument within the given market. Stored in tag `LocalCodeStr`.

In FeedOS, each market has its own rules for building *LocalCodeStr*.
This may be as simple as using exchange Symbol or ISIN, but could may be more or less complex depending on the nature of instruments present on the market.

For example, Local Codes for PUT options on EUREX have the form:

XEUR @ *symbol mm yy* P *strike*

Where *symbol* is the EUREX "product id", *mm* and *yy* are the expiry date, and *strike* is the strike price without decimal point. An optional "generation number" could even appear after a split event on the underlying.

The syntax may change over time, so **parsing or guessing Local Codes is not a good idea**.

See below "finding instrument code" for the correct procedure.

Instrument Data

Fields containing data are referenced within the API using a numeric identifier (as in FIX protocol).

These numeric identifiers are called **tag numbers**. Actually, most of them have the very same value as in FIX (`167` is SecurityType, `55` is Symbol, etc). However in FeedOS API these tags are clearly separated into two sets:

- Referential data.
  These values can be considered "static" (i.e. constant). They should not change much during lifetime of instruments.
  Values carried by these tags are called Referential **Attributes** in the API**.**

- Quotation data.
  These ones hold status and price-related values. They are often changing throughout a trading session.
  Values carried by these tags are called Quotation **Variables** in the API.

  There is an exception to this: **order book** values are stored and accessed in a specific way: list of bid/ask entries, each entry being a price and a quantity.

See the Reference Guide for a detailed list of supported tags.

See the following files for tag numbers:

feedos/api/tags_quotation.h
feedos/api/tags_quotation_context.h
feedos/api/tags_referential.h

Two predefined classes are provided for easy handling of tags. You can use them to store **tag values**.(either for referential data or prices). Those classes are designed to receive "as is" the data returned in API callbacks, and process them in a consistent manner.

See **feedos/framework/InstrumentReferentialData.h**
feedos/framework/InstrumentQuotationData.h
feedos/framework/InstrumentOrderBookData.h

Finding Instrument Codes

Principle

One FeedOS server can aggregate many markets, even those from other vendors.

What we call the "fundamental characteristics" of an instrument is the set of values that describe the instrument. Only standardised/official values should be retained. Proprietary codes should be avoided as much as possible (even FeedOS ones and even those codes from well known vendors).

On equity markets, a single field is usually sufficient to uniquely identify instruments. Several identifier fields can coexist (Symbol, ISIN, SEDOL) although one of them is usually the "official" way of identifying an instrument (usually the official Symbol).

On derivative markets, identity of an instrument is usually the combination of several values:

- product **Symbol**. Two examples:

    1. BMW on Eurex is the product symbol of all option contracts the underlying of which is the BMW stock.

    2. ES on CME is the product symbol of "e-mini S&P 500 futures".

- **CFICode** and/or **SecurityType**
  Typically: OPT for options, FUT for futures, WAR for warrants.
  Corresponding CFICodes start with O, F, RW.

- **StdMaturity, MaturityYear, MaturityMonth, MaturityDay**
  Not all of them may be provided. Check market rules for precise settlement/expiration/delivery.

Depending on the nature of instruments, more values may be needed to unambiguously identify a given contract: SecuritySubType, StrikePrice, OptionVersion, etc.

Suggested approach

The correct approach to locate instruments is:

1. Retrieve list of instruments from the server.

2. Search for instruments according to some of their fundamental characteristics.

3. When an instrument has been located, store the internal numeric code for further reference.

Interactive Search

Another (more human-friendly) approach is to search interactively. This can be achieved through API request `REFERENTIAL_Lookup`. Search is based on a pattern, and optional filtering/formatting can be applied on results. Here is an example using tool `feedos_cli`:

```
REF.lookup BMW 10 narrow ( {FOSMarketId XEUR} {CFICode OC} {MaturityYear 2008} )
```

This looks for instruments matching "BMW" among CALL options expiring in 2008 on market Eurex.

Requests and Responses

From a user point of view, the FeedOS system behaves in a Client / Server way. The client (user application) sends **Request** messages to the server. The server sends back **Response** messages (either with valid data or error codes). Both synchronous and asynchronous dialogs are implemented in API.

The API relieves the user code of low level tasks like building, encoding and sending messages. **Strong typing** alleviates most of formatting and encoding problems. Sending messages is performed by calling predefined **methods**. Receiving messages is accomplished by invoking **virtual methods acting as callbacks**. Even simpler, most requests can be issued in a synchronous fashion by calling a global function that takes input parameters, and returns both a return code and a set of output parameters.

The API supports two different types of Request / Response schemes, which we detail now.

Simple Request

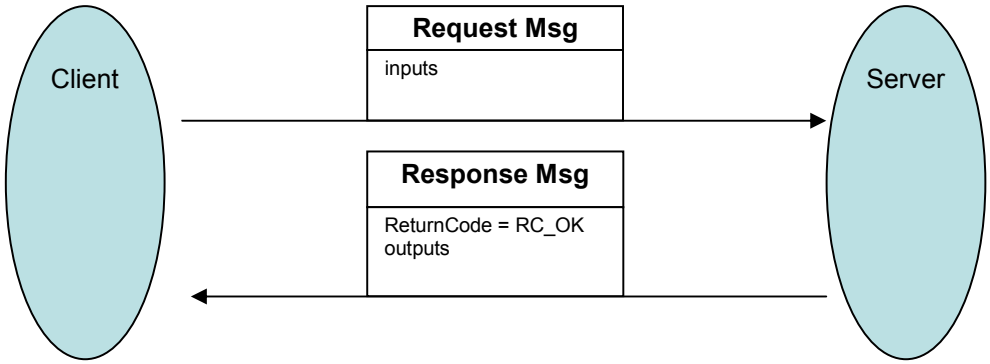
This is the easiest way of requesting data from the server, and most operations are achieved using this scheme.

A **Request** message is sent to the server. It usually contains one or more **input parameters**. The server answers with a corresponding **Response** message, containing either the correct **output parameters** or an **error code**.

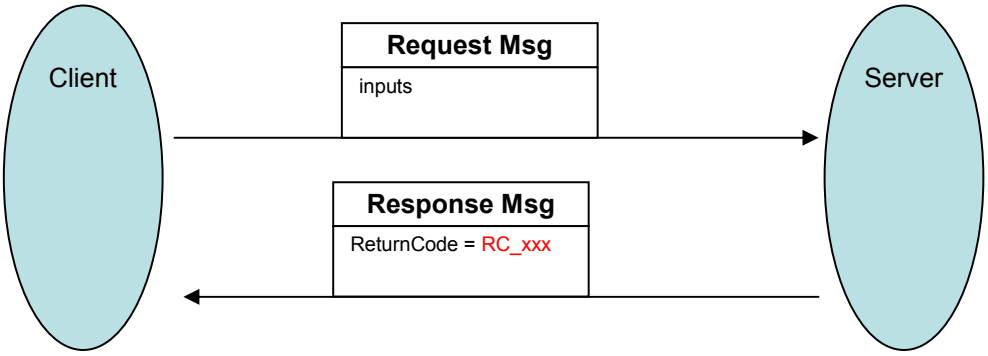
Both synchronous and asynchronous invocations are available in the API.

The following diagrams show the message cinematic (2 cases).

1 – Simple Request : Normal Case *:*



2 – Simple Request : Error Case :



Most of the time a refusal is due to invalid parameters. The reason is given by the return code.

Subscription Request

This is the way for the client to receive continuous (usually real-time) data. The goal is to establish a durable connection (a **subscription**) between the server and the client. A client object is used as a **receiver** for the requested data.
This is sometimes referred to as **streaming** of data.

A **Subscribe Request** message is sent to the server. It usually contains one or more **input parameters**. The server answers with a corresponding **Subscribe Response** message, containing either the correct **output parameters** (if the request has been accepted) or an **Error Code** (explaining why the request has been refused).

In case the request has been accepted, the server will send a **Notification** message to the client whenever new data arrives. Receipt of these messages by the API will trigger the **callback methods** of the client object. The user code is responsible for defining these methods to handle incoming data (response and notifications).

Nota Bene: most of the time, output parameters (if any) contained in the subscribe response can be considered as "initial data", and subsequent notifications contain "incremental data" that inform the client of updates of the initial data.

Nota Bene: depending on the subscription, several kinds of notifications may be available. There will be as many callbacks to implement as there are different notifications expected.

The subscription ends in two possible ways:

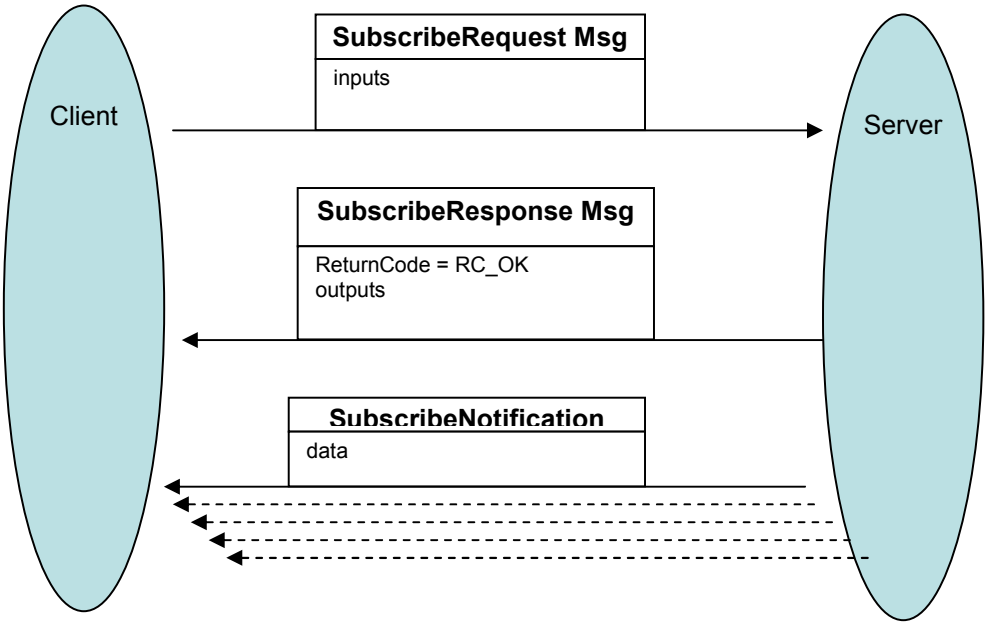
- The client can terminate the subscription by sending an **Unsubscribe Request**. This is the "normal" way.

- The server may abort the subscription by sending an **Unsubscribe Notification**. This is usually an error case.

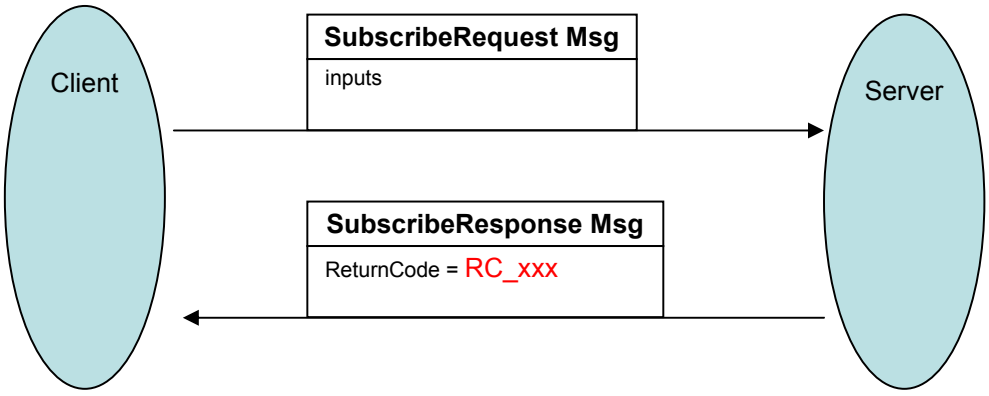
As data is continuously received, only asynchronous processing is possible.

The following diagrams explain message cinematic (4 cases):

1 – Subscription Accepted by server

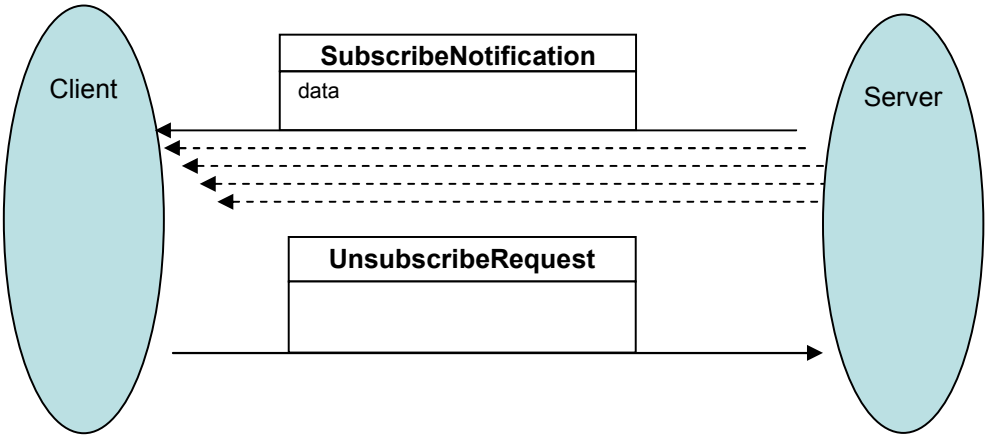


2 – Subscription: Refused by server



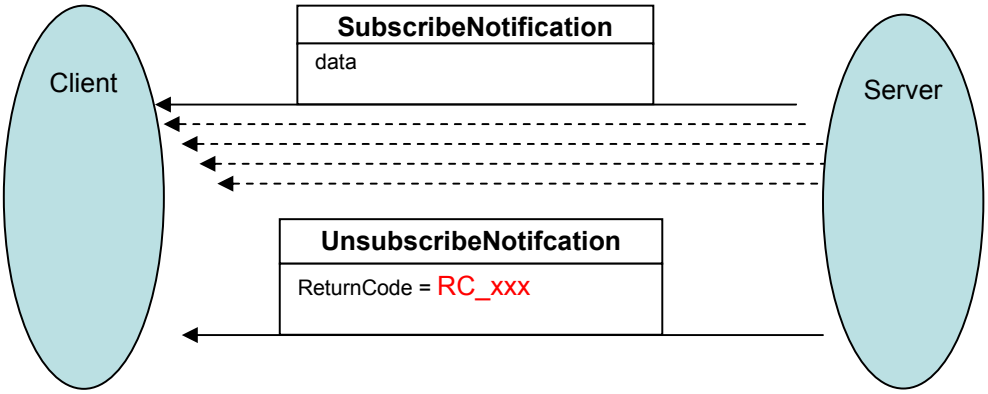
In some case server can refuse a subscription request, typically due to invalid parameters. The reason is given by the return code.

3 – Ongoing subscription: terminated by client



The unsubscribe request is always successful thus the server will never send an acknowledgement.

4 – Ongoing subscription: Aborted by server



Most of the time, an ongoing subscription is aborted due to a network disconnection. The reason is given by the return code.

Although the "subscribe request" mechanism is usually utilized to send realtime updates, it is also used for downloading data (referential and quotation). In this case, the "abort" message is used to signal the end of the download, and the return code is RC_OK.

Services and Operations

For clarity reasons, FeedOS features (or **Operations**) are grouped into **Services**. A Service is a set of operations that are related to the same type of data.

There are two main services:

1. Service REFERENTIAL is related to characteristics of financial instruments and markets (sometimes called *static data* or *dictionary*).

2. Service QUOTATION is related to "values": price and status information.

Service REFERENTIAL

This service provides a set of operations dedicated to the referential data.

The supported operations include the following requests:

- Performing a dump (all or part) of the referential data (markets and instruments)
    - `REFERENTIAL.DumpStructure`
    - `REFERENTIAL.Download`
    - `REFERENTIAL.DownloadAndSubscribe`
- Searching for instruments based on a "string pattern" with optional filtering based on various criteria
    - `REFERENTIAL.Lookup`
- Querying characteristics of individual markets or instruments
    - `REFERENTIAL.GetInstruments`
    - `REFERENTIAL.GetMarkets`

See program `sample/sample_cli` for example usage.

Service QUOTATION: realtime data

This service provides a set of operations dedicated to price and status information.

The supported operations include the following **Simple Requests**:

- Snapshot of instruments : retrieve current values (trading status, last price, volumes, best ask / bid limits, etc)
    - `QUOTATION.SnapshotInstrumentsL1`
- History : retrieve list of values over a period of time (intraday trades or daily open/high/low/close)
    - `QUOTATION.GetHistoryIntraday`
    - `QUOTATION.GetHistoryDaily`

Moreover, the following **Subscribe Requests** are available:

- Subscription to **trade events** for one or more instruments : same as "snapshot of instruments" + continuous updates
    - `QUOTATION.SubscribeInstrumentsL1`
- Subscription to **order book depth** ("Market By Level") for one or more instrument (initial snapshot + continuous updates), this request is deprecated.
    - `QUOTATION.SubscribeInstrumentsL2`
- Subscription to **order book depth** ("Market By Level") with multi-layer order book composed of price , cumulated quantity and number of orders, for one or more instrument (continuous updates)
    - `QUOTATION.SubscribeInstrumentsMBL`
- `Subscription to M`arket Sheet ("Market By Order") for one instrument: (initial snapshot + continuous updates)
    - `QUOTATION.SubscribeOneInstrumentMarketSheet`
      NB: not all markets support for kind of order book data

See program `sample/sample_cli` for example usage.

The Service QUOTATION: Feed Replay

It is possible to replay one feed (usually one feed = one market) over a given period of time. L1 and L2 data are supported, along with intraday bars and "mixed L1/L2" when this has been enabled on the replay server.

Some filtering is generally possible on server side, based on various criteria (list of instrument codes, markets, kinds of messages).

This is available through high-level interfaces: files `framework/FeedReplay*`

See program `sample/sample_feed_replay` for example usage.

Chapter 4

Using the API

Header files

A unique top-level include file can be used to load the declaration of all API features:

```
#include <feedos/api/api.h>
```

Namespaces

Several C++ "namespaces" are used to encapsulate types and values defined within the API. Three of them will be needed by user application code:

namespace	usage
`FeedOS::`	All basic types, classes and functions
`FeedOS::Types::`	All structures, enumerated values and complex types used in as input or output parameters in API Requests and Responses.
`FeedOS::API::`	Top level functions

You can avoid too much typing by using some or all of the following statements (use explicit naming in case of any ambiguity):

```
using namespace FeedOS;
using namespace FeedOS::Types;
```

Initialisation and termination of the API

The FeedOS C++ API must be initialized before any API request is invoked. Likewise, avoid instantiating API objects before initialization occur, as this may not work in future releases. Avoid global variables.

The initialization function is defined as follows. Please check that return code is `RC_OK`, and provide a comprehensive "user application name" (which will be sent to the server and could be used to troubleshoot connection problems).

The termination function is defined as follows, and should be called when application exits:

API Initialization & Termination

```
ReturnCode FeedOS::API::init (char const * user_application_name);

void FeedOS::API::shutdown ();
```

Using the optional built-in Trace Facility

A built-in trace facility is provided by the API. When it is enabled, the API itself uses that to trace internal behaviour. It is strongly advised to enable it, at least while the user application is being developed and tested.

(optional) Initialization of Built-in Trace Facility

```
// init trace facility (simple configuration: dump "errors" and "info" messages in a file)
ReturnCode init_trace_all_areas (     char const * logfile,
                                      bool append_mode              = true,
                                      bool flush_every_line         = true,
                                      bool enable_error_levels      = true,
                                      bool enable_warning_level     = true,
                                      bool enable_info_level        = true,
                                      bool enable_debug_levels      = false,
                                      bool enable_scope_level       = false
                                 );

// trace facility: setup specific config for USER area
ReturnCode init_trace_user_areas (    bool enable_error_levels      = true,
                                      bool enable_warning_level     = true,
                                      bool enable_info_level        = true,
                                      bool enable_debug_levels      = true,
                                      bool enable_scope_level       = true
                                 );
```

User application can use this trace facility, by using the macros `FEEDOS_LOG` and `FEEDOS_SCOPER`.
Take a look at sample programs provided with the API.

The STL

The Standard Template Library (which is part of C++ ISO standard) is extensively used by the API itself.

However, from the user application point of view this is not much visible. Only **`std::string`** and **`std::vector`** classes have to be used to access some parameters of API functions.

For those not familiar with the STL here is an abstract of common operations on a vector (which can be considered the C++ version of the good'ol C array).
Let's say that the variable `list` has been defined as: `std::vector<int> list;`

method	usage	example
`unsigned int size()`	Returns the number of elements in the vector	`nb_items = list.size();`
`void clear()`	Erases all the elements	`list.clear();`
`int & operator[](unsigned int n)`	Returns the n'th element	`int a = list[3];`
`void push_back (const int & v)`	Inserts a new element at the end	`list.push_back(100);`

Implementing the Client / Server behaviour

Nota Bene: this is optional material, as understanding the internal architecture of classes is not required to use the API.

The Client / Server pattern

Two entities are used to manage Requests and Responses.

1- `Requester`: Client entity used to send requests and receive responses.

2- `RequestHandler`: Server entity which processes requests and generates responses.

They are implemented using abstract base classes of the same name, from which are derived a set of classes to obtain the required specific behaviour.

Server behaviour

Base class `RequestHandler` is derived into `ProxyServer` which is in turn derived into `TCPProxyServer`. An instance of this class allows connecting to a remote FeedOS server and acts as a local "proxy" for this server (i.e. it encodes and sends Request messages over the network, and conversely receives / decodes/dispatches Response messages).
Nota Bene: as this is the only kind of RequestHandler available, an alias (*typedef*) for `TCPProxyServer` is called **`FeedOS::API::Connection`**. This is the type that will be used throughout the examples.

Client behaviour

Base class `Requester` is derived into:

1. `SyncRequester` (to implement synchronous processing)
2. `AsyncRequester_base` (base class for implementing asynchronous processing)
3. `Subscriber_base` (base class for implementing processing of Subscribe Requests)

`AsyncRequester_base` and `Subscriber_base` are in turn derived into several classes, one for each supported Request.

Client behaviour: how to perform Simple Requests

In order to implement the Request/Response dialog, a single object is responsible for both:

1. Sending the request, along with any required **input parameters**
2. Receiving response data (either an **error code** or **output parameters**, if any).

Synchronous processing

Every Simple Request available through the API has a corresponding global function defined.

This is a synchronous, blocking call that allows sending a Request and receiving the Response as if the remote server processing was taking place in a local function.

User application as little control over the process, but this may be the desired behaviour in two cases:

1. simple single-threaded user application that can afford to be blocked during requests
2. muli-threaded application where "background" thread(s) are responsible for performing requests. Meanwhile, the main thread may be executing some kind of "event processing loop" associated to a Graphical User Interface.

Example:

Simple Request, synchronous version : what is provided by the API
Global Function `API_QUOTATION_SnapshotInstruments`

```
// synchronous version (blocking function call)
ReturnCode API_QUOTATION_SnapshotInstruments
             (
                RequestHandler & req_handler,    // this is the target "server" object
                // outputs (passed by reference)
                FeedOS::Types::ListOfInstrumentStatus & Result,
                // inputs
                FeedOS::Types::ListOfPolymorphicInstrumentCode const & Codes,
                FeedOS::Types::ListOfQuotationTagNumber const & Values,
                int8 OrderBookDepth = 1,
                bool IgnoreInvalidCodes = false
             );
```

Simple Request, synchronous version : what should be done by user application

```
void main ()
{
  FeedOS::API::init("my_app");
  FeedOS::API::Connection server_connection;
  server_connection.sync_connect ("localhost", 6040, "joe", "password");

  // should define output parameters here
  API_QUOTATION_SnapshotInstruments
        (    server_connection,
             /* should pass the output parameters here */
             /* should pass the input parameters here */
        );
}
```

Asynchronous processing

Every Simple Request available through the API has a corresponding abstract base class that provides:

1. A predefined method for sending the request:
   `void send (target_req_handler, input_parameters…)`

2. A virtual "callback" method for handling the "response success" case:
   `void response_SERVICE_OPERATION_Success (output_parameters…)`

3. A virtual "callback" method for handling the "error response" case:
   `void response_SERVICE_OPERATION_Failed (ReturnCode)`

User application should derive this class and implement the two callback methods.

Example:

Simple Request, asynchronous version : what is provided by the API
Class `API_QUOTATION_SnapshotInstruments_base`

```
class API_QUOTATION_SnapshotInstruments_base: public AsyncRequester_base
{
public:
  API_QUOTATION_SnapshotInstruments_base ();

  void send (
              RequestHandler & handler,  // this is the target "server" object
              // inputs
              FeedOS::Types::ListOfPolymorphicInstrumentCode const & Codes,
              FeedOS::Types::ListOfQuotationTagNumber const & Values,
              int8 OrderBookDepth = 1,
              bool IgnoreInvalidCodes = false
            );

  // response callback : "success" + outputs
  virtual void response_QUOTATION_SnapshotInstruments_Success
                        (
                           // outputs
                           FeedOS::Types::ListOfInstrumentStatus const & Result
                        ) = 0;

  // response callback : some error occurred
  virtual void response_QUOTATION_SnapshotInstruments_Failed (ReturnCode rc) = 0;

};
```

Simple Request, asynchronous version : what should be implemented by user application

Class `MyAsyncSnapshotRequester`

```
class MyAsyncSnapshotRequester: public API_QUOTATION_SnapshotInstruments_base
{
public:

  // response callback : "success" + outputs
  virtual void response_QUOTATION_SnapshotInstruments_Success
                        (
                           // outputs
                           FeedOS::Types::ListOfInstrumentStatus const & Result
                        )
  {
      cout << "success! Result contains "<<Result.size()<<" entries"<< endl;
  }

  // response callback : some error occurred
  virtual void response_QUOTATION_SnapshotInstruments_Failed (ReturnCode rc)
  {
      cout << "ERROR, rc="<< rc << endl;
  }
};


void main ()
{
  FeedOS::API::init("my_app");
  FeedOS::API::Connection server_connection;
  server_connection.sync_connect ("localhost", 6040, "joe", "password");

  MyAsyncSnapshotRequester r;
  r.send (server_connection /* should pass the input parameters here */);

  // should not exit immediately here, to let the response message arrive
}
```

Client behaviour: how to perform Subscribe Requests

In this case a Requester object is responsible for sending the request whereas a Receiver object is responsible for handling all the (asynchronous) incoming data:

- To handle "subscription refused": initial Response and the error code it carries
- To handle "subscription accepted": initial Response and its output parameters, if any
- Notifications and the data they carry, if any
- the possible Abort message, and associated error code

The "streaming interface"

For clarity, the set of methods required to implement a "subscription receiver" is declared in an abstract base class called a **Streaming Interface**.

Example:

an example of a "Streaming Interface" class used to receive Subscription-related data
INTERFACE_QUOTATION_SubscribeInstruments_streaming

```
class INTERFACE_QUOTATION_SubscribeInstruments_streaming
{
public:
  // Response callback (called when subscription is ACCEPTED)
  virtual void response_QUOTATION_SubscribeInstruments_Started
                (
                   // output parameters ("initial" data)
                   uint32 inTicket,
                   FeedOS::Types::ListOfInstrumentStatus const & inSnapshot
                ) = 0;

  // Response callback (called when subscription is REFUSED)
  virtual void response_QUOTATION_SubscribeInstruments_Failed (ReturnCode rc) = 0;

  // Abort callback (called when subscription has been ABORTED by the server)
  virtual void aborted_QUOTATION_SubscribeInstruments (ReturnCode rc) = 0;

  // 1st Notification callback (a trade event occurred on the given instrument)
  virtual void notif_TradeEvent
                (
                   // incoming ("updated") data
                   FOSInstrumentCode inCode,
                   Timestamp const & inUTCTimestamp,
                   FeedOS::Types::QuotationTradeEvent const & inData
                ) = 0;

  // 2nd Notification callback (some "other values" have been updated)
  virtual void notif_ValuesUpdate
                (
                   // incoming ("updated") data
                   FOSInstrumentCode inCode,
                   Timestamp const & inUTCTimestamp,
                   FeedOS::Types::QuotationValuesUpdate const & inData
                ) = 0;

};
```

The Requester

For each Subscribe Request available through the API, there is class that provides the Requester behaviour using two predefined methods:

1. **`void start (target_req_handler, input_parameters…)`**
   This sends the Subscribe Request message, and returns immediately.

2. **`ReturnCode sync_start (target_req_handler, input_parameters…)`**
   This sends the Subscribe Request message, wait for the Response to arrive, calls the appropriate "response callback", and returns a copy of the ReturnCode received (`RC_OK` in case of success).

3. **`void stop ()`**
   This sends the Unsubscribe Request message, and returns immediately.
   Subscription is terminated, and no more data will be received until it is restarted using one of the methods above.

Two ways of implementing the Receiver

There is a choice to do here depending on what you think is best suited to your code:

1. The Requester and the Receiver roles are implemented by the same object:
   => Derive the class **`API_SERVICE_OPERATION_base`** and implement the virtual callbacks inherited from the corresponding **streaming interface**.
   Instances of this new class will be standalone, full-fledged Requester *and* Receiver objects.

2. The Receiver is an external object, acting as a "delegate":

   a. => Derive the **streaming interface** and implement the virtual callbacks. Instances of this new class are pure ("passive") Receivers, with no direct control over the ongoing subscription.

   b. => Use class **`API_SERVICE_OPERATION_delegate`**. Instances of this class are pure Requesters, and need to point to a "delegate" object that will process the incoming data.

The first solution is the more commonly-desired behaviour, and is consistent with how the Simple Requests work.
The second solution is useful to have a single "receiver" pointed to by many Requester objects.

Example using the first solution:

Subscribe Request: what should be implemented by user application
Class `MyMarketStatusSubscriber`

```
class MyMarketStatusSubscriber: public API_QUOTATION_SubscribeMarketsStatus_base
{
public:

  // callbacks inherited from INTERFACE_QUOTATION_SubscribeMarketsStatus_streaming
  virtual void response_QUOTATION_SubscribeMarketsStatus_Started
                  (
                     FeedOS::Types::ListOfMarketStatus const & inMarketsStatus
                  )
  {
     cout << "subscription accepted, initial status="
          << inMarketsStatus[0].getCurrentSessionStatus().getTradSesStatusEnum()
          << endl;
  }
  virtual void response_QUOTATION_SubscribeMarketsStatus_Failed (ReturnCode rc)
  {
     cout << "subscription REFUSED, rc="<< rc << endl;
  }
  virtual void aborted_QUOTATION_SubscribeMarketsStatus (ReturnCode rc)
  {
     cout << "subscription ABORTED, rc="<< rc << endl;
  }
  virtual void notif_MarketStatus     (FeedOS::Types::MarketStatus const & inStatus)
  {
     cout << "new status="
          << inStatus.getCurrentSessionStatus().getTradSesStatusEnum()
          << endl;
  }
  virtual void notif_MarketNews       (FeedOS::Types::MarketNews const & inNews)
  {
     cout << "NEWS headline="
          << inNews.getHeadline()
          << endl;
  }

  // the start(), sync_start() and stop() methods are inherited from
  //    API_QUOTATION_SubscribeMarketsStatus_base
};


void main ()
{
  FeedOS::API::init("my_app");
  FeedOS::API::Connection server_connection;
  server_connection.sync_connect ("localhost", 6040, "joe", "password");

  MyMarketStatusSubscriber r;
  r.start (server_connection /* should pass the input parameters here */);

  // should wait here, to let the response and notifications arrive continuously
}
```

Handling of instrument codes

As explained before, Internal Codes are not human-friendly. Users should always be presented (or requested for) Local Codes. These Local Codes can be passed as input parameters to most API Requests.

However in Responses sent by the server, instruments are always referenced using Internal Codes. User application is responsible for matching input Local Codes with output Internal Codes. This is usually achieved using a single line of code.

In order to make both flavours of instrument code coexist gracefully, a C++ class is provided in the API: **`PolymorphicInstrumentCode`**. It allows storing an instrument code in Local Code form, in Internal Code form, **or both**. It is a good idea to use this class to store instrument codes, and always have both forms loaded (when possible). Two methods of this class allow to "merge" one form into the polymorphic instance.

There are 2 typical lifecycle of an instrument code:

1. Starting with the Local Code
    a. An instance of `PolymorphicInstrumentCode` is built using the Local Code form
    b. Some Request is issued towards the server, where the `PolymorphicInstrumentCode` is sent as an input parameter
    c. The Response from the server contains (among other things) the corresponding Internal Code
    d. When parsing output parameter, don't forget to `merge()` the given Internal Code into the `PolymorphicInstrumentCode`
    e. When issuing other Requests for this instrument, simply pass the `PolymorphicInstrumentCode`. The Internal Code form will be used automatically, allowing optimal performance.
2. Starting with the Internal Code (returned by the server within a `REFERENTIAL.Lookup` response, for example)

This is always as simple as skimming through both input

Most API Requests take **Polymorphic Instrument Codes** as input parameters.

However, as soon as the correspondn

The two identification system can be used in same time. Requests with Instruments parameters use the class comm./PolymorphicInstrumentCode. The user can choose to use either LocalCode either InternalCode. In response Instrument Codes are always InternalCode. So it is clever to update objects PolymorphicInstrumentCode with their InternalCode form, in order to accelerate following request. To do it use the function PolymorphicInstrumentCode:merge_internal_code ().

Establishing and monitoring a network connection

As said above, the C++ class `API::Connection` (which is an alias for `FeedOS::TCPProxyServer`) is the only type of RequestHandler available in the API. It allows initiating a connection toward a FeedOS Server. An instance of that class *represents* a remote server. User application can send **requests** to it, which will be encoded and sent through the network. Likewise, **responses** received from the network will be decoded and dispatched to the appropriate **requester** object.

Initiating the connection

The `sync_connect()` method allows to establish the TCP connection, send logon information and wait for the response. The return code will be `RC_OK` if the process was successful.

Monitoring the connection

The provided class `FeedOS::ProxyServerObserver` may be derived in order to monitor the state of an `API::Connection` object. An instance of this new class should be passed to `API::Connection::add_observer()` to be notified of "events" (typically, a network disconnection).

Monitoring the network connection: what <u>may be implemented by user application</u>
Class `MyProxyObserver`

```
#include <feedos/api/ProxyServerObserver.h>

class MyProxyObserver: public FeedOS::ProxyServerObserver
{
public:
  // callbacks inherited from ProxyServerObserver
  virtual void proxy_state_changed_hook             (ProxyServer & p)
  {
     cout <<"connection state="<<p.get_login_state_str()<<endl;//don't rely on this (not accurate)
  }
  virtual void proxy_disconnected_hook              (ProxyServer & p)
  {
     cout <<"unexpected disconnection"<<endl;
  }
  virtual void proxy_connected_hook                 (ProxyServer & p)
  {
     cout <<"connection+authentication OK"<<endl;
  }
  virtual void proxy_heartbeat_hook                 (ProxyServer & p) {}
  {
     cout <<"received heartbeat from server: "<<p.get_heartbeat_server_timestamp()<<endl;
  }
  virtual void proxy_normal_disconnect_hook  (ProxyServer & p)
  {
     cout <<"user-requested disconnection"<<endl;
  }
};
void main ()
{
  FeedOS::API::init("my_app");
  FeedOS::API::Connection server_connection;

  MyProxyObserver obs;
  server_connection.add_observer (obs);

  server_connection.sync_connect ("localhost", 6040, "joe", "password");
  // put your code here
  server_connection.disconnect();

  FeedOS::API::shutdown ();
}
```

Monitoring the dispatching of events

Some users might want to know when incoming data are being dispatched (and associated user callbacks are called consequently). This may allow more efficient acquisition of locks/mutexes, or may allow processing data as batches, rather than individual messages.

The provided class `FeedOS::ProxyServer::MsgDispatchingHooks` may be derived in order to detect when a bunch of events are begin dispatched.

Monitoring the dispatching of events: what could be implemented by user application
Class `MyDispatchingHooks`

```
#include <feedos/api/ProxyServer.h>

class MyDispatchingHooks: public FeedOS::ProxyServer::MsgDispatchingHooks
{
public:
  // callbacks inherited from ProxyServer::MsgDispatchingHooks
  virtual void msg_dispatching_hook_begin ()
  {
     cout <<"message(s) are being dispatched"<<endl; // could acquire a mutex here
  }
  virtual void msg_dispatching_hook_end ()
  {
     cout <<"message(s) have been dispatched"<<endl; // could release a mutex here
  }
};

void main ()
{
  FeedOS::API::init("my_app");
  FeedOS::API::Connection server_connection;

  MyDispatchingHooks spy;
  server_connection.set_dispatching_hooks_obj (spy);

  server_connection.sync_connect ("localhost", 6040, "joe", "password");
  // put your code here (user callbacks are relieved from using the mutex)
  server_connection.disconnect();

  FeedOS::API::shutdown ();
}
```

Locking and synchronization

When using asynchronous processing (which is typically the case when dealing with subscriptions) appropriate locking must be performed to prevent memory corruptions due to concurrent accesses.
We suggest using classes `ExclusiveLock` and `ReadWriteLock` to encapsulate critical sections of code or to protect shared data structures.
See `feedos/base/lock.h`

Examples of creating a critical section to protect a shared variable:

Data	Locking Code

`std:string my_data;`
`ExclusiveLock my_data_lock;`
	`void some_callback (std:string v)`
	`{`
	`  // enter the critical section`
	`  my_data_lock.acquire();`
	
	`  // access the lock-protected data`
	`  my_data = v;`
	
	`  // leave the critical section`
	`  my_data_lock.release();`
	`}`

An alternate way for calling `acquire()` and `release()` automatically is to use a "scoper" object:

Alternate locking code

```
void some_callback (std:string v)
{
   {
      FeedOS::exlock_auto L(my_data_lock); // acquire() done in the constructor
      my_data = v;

   } // release() done in the destructor of object L

   // here is outside the critical section
}
```

Nota Bene: Macros **`SCOPE_AUTO_xxxx_LOCK`** `may be used to` encapsulate the definition of a scoper object.
See `feedos/base/lock.h`

Threading schemes

The FeedOS C++ API is multi-threaded which means that reading data from the network is accomplished by the API itself without the need of an event processing loop.
The user callbacks responsible for processing incoming data are triggered from these internal threads, in an asynchronous and unpredictable way.
Using synchronous functions may affect the short processing delays which may cause the disconnection of the API.

It is recommended to use `FeedOS::API::Connection` object, which presents the following threading model:

- one thread polling tcp socket
- one thread dedicated to events dispatching towards user callbacks
- a background thread monitoring heartbeat

Thread Dedicated to Event Dispatching

In some cases, users may need to control the incoming data dispatching towards the user callbacks, for that the API provides:

- `FeedOS::API::Connection::disable_internal_event_dispatching()` that disables the internal message dispatching
- `FeedOS::API::Connection::dispatch_events_forever()` dispatches the incoming messages enqueued in memory, this is a blocking call it returns RC_DISCONNECTED in case of unexpected network disconnection and RC_NORMAL_DISCONNECT in case the user triggered the disconnection.

Several samples that perform synchronous connection towards a server, and then starts dispatching messages until a normal disconnection (i.e. triggered by the user) happens are available under `feedos/api/sample/sample_app/*_dipatching_thread.cpp`

Managing the Underlying Socket Directly

In some cases, the user application may need full control over the network connection (single-threaded environment, global event loop, etc).
This can be achieved by using class `FeedOS::API::SimpleConnection` instead of `FeedOS::API::Connection`, which behaves in the same way, except that it exhibits the underlying TCP socket. Thus the user application can:
1. perform `select()` on the socket to detect incoming data
2. call method `read_socket()` to read data, and enqueue corresponding messages
3. call method `dispatch_pending_events()` to process enqueued messages or perform in another thread to maximize parallelization, for example by calling the method `dispatch_events_forever()`.

Several samples which process messages until a normal disconnection (i.e. triggered by the user) is available under
`feedos/api/sample/sample_app/*_socket_polling_thread.cpp`

Chapter 5

Building a user application

The sample application(s) provided with the API are a practical demonstration of how to use the API.
This covers both the source code of user application, and the makefile / project settings required to compile and link with the API.

A small Command Line Interface interpreter is provided to demonstrate how to use the most common features of the API.

Compiling & running the sample_cli

On Windows

Extract the ZIP archive (or run the self-extracting EXE) in some directory.

Go to sub-directory `feedos/samples` and open the file `samples.sln` (this is a Visual Studio "solution"). Build the project `sample_cli`.
Some packages may have script files instead (like `samples/sample_cli/BUILD.BAT`). Run it to invoke compiler and linker.

You can create a Windows shortcut or BAT file to invoke it from Explorer.
Or you can invoke it from a command line interface: Open a "command prompt window" (CMD.EXE), go to directory `feedos/api/samples/sample_cli/Debug/` then execute the file `sample_cli.exe`

On Unix

1) download the tarball in some temporary directory. For example: **/tmp**

2) decide where the API could be extracted. For example: **/home**

3) **`tar -C /home -zxf /tmp/feedos_api_3120_LINUX_i686_gcc.tgz`**
(if using API v3.1.2.0 for Linux x86)

4) `cd /home/feedos/api/samples/sample_cli`

5) `make`

This should build the executable file: `./sample_cli`

Testing the sample_cli

A small online help is available. Here below is an example of using the `sample_cli` to connect to a FeedOS server and issue a few requests (assuming your login is "test" and password is "secret"):

Using the `sample_cli` program to send requests

```
$ ./sample_cli 10.66.8.76 6040 test secret refstructure
refstructure ok

(  the list of Markets is displayed,
   along with the list of "branches" - instruments sorted by categories )

$ ./sample_cli 10.66.8.76 6040 test secret reflookup FDAX12
lookup ok, nb_hits=1
code = 12/1530
        FOSMarketId     = XEUR
        LocalCodeStr    = string{FDAX1205}
        Symbol  = string{FDAX}
        Description     = string{FUT ON DAX INDEX}
        CFICode = string{FXXXXX}
        SecurityType    = string{FUT}
$ ./sample_cli 10.66.8.76 6040 test secret refget XEUR@FDAX1205
refget ok
code = 12/1530
        PriceCurrency   = string{EUR}
        Symbol  = string{FDAX}
        Description     = string{FUT ON DAX INDEX}
        SecurityType    = string{FUT}
        FOSMarketId     = XEUR
        CFICode = string{FXXXXX}
        InternalCreationDate    = Timestamp{2005-10-17 11:06:45:640}
        InternalModificationDate        = Timestamp{2005-10-17 11:08:32:515}
        LocalCodeStr    = string{FDAX1205}
        UnderlyingLocalCodeStr  = string{DE0008469008}
        UnderlyingFOSInstrumentCode     = 12/1474
        MaturityYear    = uint16{2005}
        MaturityMonth   = uint8{12}
$ ./sample_cli 10.66.8.76 6040 test secret subscribe 12/1530
snapshot ok
code = 12/1530
        TradingStatus   = Enum{17}
        LastPrice       = float64{4886.5}
        LastTradeQty    = float64{6}
        LastTradeTimestamp      = Timestamp{2005-10-26 13:36:31:500}
        SessionTotalVolumeTraded        = float64{890390}
        SessionOpeningPrice     = float64{4900}
        SessionClosingPrice     = float64{0}
        SessionHighPrice        = float64{4932}
        SessionLowPrice = float64{4883.5}
        DailyTotalVolumeTraded  = float64{96186}
        DailyOpeningPrice       = float64{4900}
        DailyClosingPrice       = float64{0}
        DailyHighPrice  = float64{4932}
        DailyLowPrice   = float64{4881.5}
        PreviousClosingPrice    = float64{4854}
        PreviousBusinessDay     = Timestamp{null}
        bid # 0 price=4886     qty=9
        ask # 0 price=4886.5   qty=6
$ ./sample_cli 10.66.8.76 6040 test secret refdownload -v > instr.txt

       <the whole set of instruments is downloaded>
```

The project settings (win32)

Take a look on the Project Settings (for both Debug and Release configurations). This will tell you what is needed to compile and link with the API on win32.

The important settings are as follows:

<table>
<tr><th rowspan="2">Setting</th><th colspan="2">Value Required</th></tr>
<tr><th>Debug</th><th>Release</th></tr>
<tr><td>C++ => Code Generation => Runtime Library</td><td>Multi-threaded Debug DLL</td><td>Multi-threaded DLL</td></tr>
<tr><td>C++ => Preprocessor Definitions</td><td colspan="2">_REENTRANT</td></tr>
<tr><td>C++ => Precompiled Headers => Create/Use Precompiled Headers</td><td colspan="2">Not Using Precompiled Headers</td></tr>
<tr><td>C++ => General => Additionnal Include Directories</td><td colspan="2"><Where you unzipped the API> /feedos/api/include</td></tr>
<tr><td>Linker => General => Additionnal Library Directories</td><td colspan="2"><Where you unzipped the API> /feedos/api/lib</td></tr>
<tr><td rowspan="2">Linker => Input => Additionnal Dependencies</td><td colspan="2">zlib.lib
ws2_32.lib
mswsock.lib</td></tr>
<tr><td>feedos_api_debug.lib</td><td>feedos_api.lib</td></tr>
</table>

When invoking the compiler and linker directly, here is the list of options used:

Compiler:
```
-MD
-EHsc
-I../../api/include
```

Linker:
```
-MACHINE:x86                          "x64" for 64bit Windows
-NODEFAULTLIB:libcpmt.lib
MSVCPRT.LIB ws2_32.lib
LIBPATH:../../api/lib/ libfeedosapi.lib zlib.lib
```

The Makefile (unix)

Take a look on the Makefile for the sample_cli. Use similar options in your own build process.

Chapter 6

Reference Guide

Here is the list of data types and API classes / functions available.

Data types

Many different types are used throughout the API. Here is a short classification of types that user applications will have to deal with.

Typedefs for basic types

They are located in namespace `FeedOS::`

`uint32`	An unsigned int of size "32 bits"
`uint16`	An unsigned int of size "16 bits"
`uint8`	An unsigned int of size "8 bits"
`int32`	A signed int of size "32 bits"
`int16`	A signed int of size "16 bits"
`int8`	A signed int of size "8 bits"
`float32`	A floating point number in ISO 32 bits format
`float64`	A floating point number in ISO 64 bits format

Important user-level classes

`FeedOS::API::Connection`	Implements a connection towards a remote FeedOS server
`FeedOS::ProxyServerObserver`	A base class for implementing observer objects to monitor `Connection` instances.
`FeedOS::InstrumentReferentialData`	Stores referential-related values (**tags**) These are the instrument's characteristics ("static data")
`FeedOS::InstrumentQuotationData`	Stores quotation-related values (**tags**) These consist of status, price and volume information.

Utility classes

They are located in namespace `FeedOS::`

`FOSMarketID`	container for a FeedOS numeric MarketID
`FOSInstrumentCode`	container for a FeedOS instrument's numeric "internal code"
`LocalInstrumentCode`	container for a FeedOS instrument's "local code"
`PolymorphicInstrumentCode`	Sum of `FOSInstrumentCode` and `LocalInstrumentCode`
`Enum`	Container for an enumerated value
`String`	An alias for `std::string`
`Any`	A "union" container for a value. Supported syntaxes are all the "basic types" + `string` + `Enum`
`ReturnCode`	A container for an error code. See `feedos/comm/error.h`
`Timestamp`	Models a millisecond-precision time. Nota Bene: most of the time values are expressed in UTC.
`ExclusiveLock`	Models an exclusive lock mechanism.
`ReadWriteLock`	Models a many-readers / one writer lock mechanism.
`smart_ptr`	A generic reference-counting, auto-deallocate pointer type

Data types used as parameters to API functions

Regarding the "complex" types (structures, enums or typedefs) that are passed as parameters of API functions, they can be found in `feedos/api/generated/gen_types.h`
They are located in namespace `FeedOS::API::`

Here is the list of the most commonly used:

`TagNumber`	A numeric id for a tag.
`QuotationTagNumber`	A numeric id for a tag related to quotation data. This is an alias for `TagNumber`
`ReferentialTagNumber`	A numeric id for a tag related to referential data. This is an alias for `TagNumber`
`TagNumAndValue`	TagNum + Any This is a container for a **tag (id and value)**.
`CFICode`	An alias for `std::string`
`FOSMarketId`	An alias for `std::string`
`TransactionOrPrice`	Timestamp + price + quantity
`OrderBookEntry`	Price + Quantity
`OrderBook`	Timestamp + two lists of `OrderBookEntry` (one for Ask side, one for Bid side)
`QuotationTradeEvent`	A structure containing the data related to one "trade event": price, quantity, best limits price and quantity, etc. A special field "content" of type `QuotationContentMask` is used to tell what the relevant fields are.
`QuotationContentMask`	An alias for uint32. This is a bit field, and can be parsed using the macro `FEEDOS_QUOT_UPD_CONTENT_HAS(what)`
`QuotationValuesUpdate`	"content" mask + a list of Quotation Variables
`TradingSessionCharacteristics`	The characteristics of a trading session: internal id, FIX id / mode / method
`TradingSessionStatus`	The current status of a trading session: internal id, FIX status and various open/close times.
`MarketCharacteristics`	Characteristics of a financial market : internal numeric id, MIC string, time zone, country, list trading sessions, etc
`MarketStatus`	Status of a financial market : internal id + status of current trading session
`HMS_Time`	Hour + minute + second
`YMD_Date`	Year + month + day
`YMD_Date_packed`	Number of days since epoch (1970-01-01)

MarketBranchId	Market id + Security Type + CFICode This is used to reference a "branch" of instruments inside the whole referential data
MarketBranchContent	MarketBranchId + number of instruments in the branch
MarketContent	Market id + list of MarketBranchContent
InstrumentCharacteristics	internal instrument code + list of Referential Attributes
InstrumentStatus	internal instrument code + list of Quotation Variables + order book

List of supported Tags

You can use function `FeedOS::dump_supported_tags (std::ostream & );` to dump them all.

Referential-related tags ("Attributes")

FIX-compliant attributes

Tag	Syntax	Description
CFICode	String	See FIX standard, tag 461
SecurityType	String	See FIX standard, tag 167
SecuritySubType	String	See FIX standard, tag 762
Symbol	String	See FIX standard, tag 55
Description	String	See FIX standard, tag 107
Product	uint32	See FIX standard, tag 460
CountryOfIssue	String	See FIX standard, tag 470
PriceCurrency	String	See FIX standard, tag 15
ContractMultiplier	float64	See FIX standard, tag 231
StrikePrice	float64	See FIX standard, tag 202
StrikeCurrency	String	See FIX standard, tag 947
OptAttributeVersion	uint8	See FIX standard, tag 206 (*contract version* only)
NbLegs	uint8	See FIX standard, tag 555

FIX-interpolated attributes

Tag	Syntax	Description
FOSMarketId	uint16	Parent market's ID (internal numeric form) See FIX tag SecurityExchange
LocalCodeStr	String	Unique identifier inside parent market. See FIX tags SecurityID, SecurityIDSource
ForeignFOSMarketId	uint16	Real market the instrument is listed on.
ForeignMarketId	String	Same as above, used when the parent market doesn't have an ISO MIC
ISIN	String	See FIX tags Security[Alt]ID, Security[Alt]IDSource
UnderlyingLocalCodeStr	String	(for derivatives) identifier of underlying
UnderlyingFOSInstrumentCode	uint32	
StdMaturity	String	(for derivatives) official maturity: year+month/quarter/week
MaturityYear	uint16	(for derivatives) See FIX tags MaturityMonthYear, MaturityDate
MaturityMonth	uint8	
MaturityDay	uint8	
LegFOSInstrumentCode_***n***	uint32	(for multilegs) See FIX tags LegSecurityID, LegSecurityIDSource
LegRatioQty_***n***	float64	(for multilegs) See FIX tag LegRatioQty
LegFIXSide_***n***	char	(for multilegs) See FIX tag LegSide

Other attributes

Tag	Syntax	Description
ICB_IndustryCode	uint32	See Industry Classification Benchmark
ICB_SupersectorCode	uint32	
ICB_SectorCode	uint32	
ICB_SubsectorCode	uint32	
BloombergTicker	String	
ReutersInstrumentCode	String	

Market-specific attributes

Several markets have specific tags to carry non-standard attributes. See files `api/tags_*.h`

Attributes for internal use

NB: users should not have to use them.

Tag	Syntax	Description
`InternalCreationDate`	`Timestamp`	Date of creation (server time)
`InternalModificationDate`	`Timestamp`	Date of last modification (server time)
`InternalHideFromLookup`	`bool`	If TRUE the instrument should be considered “dead”
`InternalSourceId`	`uint16`	Id of the parent FeedHandler
`InternalMagic_`***`n`***		Internal details about instrument’s characteristics

Quotation-related tags ("Variables")

"Instant" values

Tag	Syntax	Description
TradingStatus	Enum	See FIX tag 326 « SecurityTradingStatus »
TradingSessionId	int8	The numeric ID of the session the last price belongs to. Starts with 0.
LastPrice	float64	price of the last trade (or spot of an index) or other indicative price: opening, closing, etc.
LastTradeQty	float64	Quantity of the last trade
LastTradeTimestamp	Timestamp	UTC Timestamp of the last trade (official market time)
LastAuctionPrice	float64	(during auction period) auction price
LastAuctionVolume	float64	(during auction period) auction volume

"Daily" values

Tag	Syntax	Description
DailyTotalVolumeTraded	float64	Cumulated volume of all trade quantities during the current trading day
DailyTotalAssetTraded	float64	Sum of all price*qty of all trades during the current trading day
DailyOpeningPrice	float64	Opening price of the current trading day
DailyClosingPrice	float64	Closing price of the current trading day
DailySettlementPrice	float64	Settlement price of the current trading day
DailyHighPrice	float64	Highest price of the current trading day
DailyLowPrice	float64	Lowest price of the current trading day
PreviousDailyClosingPrice	float64	DailyClosingPrice of previous trading day
PreviousDailySettlementPrice	float64	Settlement price of the previous trading day
PreviousBusinessDay	Timestamp	previous trading day
CurrentBusinessDay	Timestamp	current trading day
OpenInterest	float64	(for derivatives) Open Interest

"Context" values

See document "FeedOS API TradeConditions"

Tag	Syntax	Description
TradeCondition	String	See equivalent tag in FIX Protocol
Buyer	String	See equivalent tag in FIX Protocol
Seller	String	See equivalent tag in FIX Protocol
MARKET_LSE_*		
MARKET_EURONEXT_*		
MARKET_XETRA_*		
MARKET_CME_*		
MARKET_LIFFE_*		
MARKET_TURQUOISE_*		
MARKET_SWX_*		
MARKET_ICE_*		
MARKET_BME_*		
…		
COMSTOCK_*		See Comstock documentation
BPIPE_*		See Bloomberg documentation

For internal use

NB: users should not have to use that

Tag	Syntax	Description
InternalPriceActivityTimestamp	Timestamp	Date of last trade or best quote (server time)
InternalLastAuctionTimestamp	Timestamp	Time of last auction Price or Volume
InternalDailyOpenTimestamp	Timestamp	
InternalDailyCloseTimestamp	Timestamp	
InternalDailyHighTimestamp	Timestamp	
InternalDailyLowTimestamp	Timestamp	

List of supported requests

Service REFERENTIAL / simple requests

NB: see `samples/sample_cli/perform_*.cpp`

API_REFERENTIAL_DumpStructure

```
// outputs
FeedOS::Types::ListOfMarketCharacteristics & Markets,
FeedOS::Types::ListOfMarketContent & Content,
Timestamp & LastUpdateTimestamp
```

Dump the overall « referential structure » (list of markets, and list of all branches)

API_REFERENTIAL_GetInstruments

```
// outputs
FeedOS::Types::ListOfInstrumentCharacteristics & Instruments,
// inputs
FeedOS::Types::ListOfPolymorphicInstrumentCode const & Codes,
FeedOS::Types::ListOfReferentialTagNumber const & Format,
bool IgnoreInvalidCodes = false
```

Retrieve characteristics of given instruments

API_REFERENTIAL_GetMarkets

```
// outputs
FeedOS::Types::ListOfMarketCharacteristics & Markets,
// inputs
ListOfFOSMarketId MarketIds
```

Retrieve characteristics of given markets

API_REFERENTIAL_Lookup

```
// outputs
FeedOS::Types::ListOfInstrumentCharacteristics & Instruments,
// inputs
String const & Pattern,
uint16 MaxHits, FeedOS::Types::LookupMode Mode,
FeedOS::Types::ListOfReferentialAttribute const & Filter,
FeedOS::Types::ListOfReferentialTagNumber const & Format
```

Search for instruments, using a string pattern and optional filtering/formatting

API_REFERENTIAL_ResolveCodes

```
// outputs
FeedOS::Types::ListOfFOSInstrumentCode & InternalCodes,
// inputs
FeedOS::Types::ListOfPolymorphicInstrumentCode const & Codes,
bool IgnoreInvalidCodes = false
```

Translate a list of instrument “local codes” into corresponding “internal codes”

API_REFERENTIAL_GetVariablePriceIncrementTables

```
// outputs
FeedOS::Types::ListOfVariableIncrementPriceBandTable & Tables
```

Download list of “variable tick size” tables.
See referential attribute “PriceIncrement_dynamic_TableId”.

API_REFERENTIAL_GetTradeConditionsDictionary

```
            // outputs
            FeedOS::Types::ListOfTradeConditionsDictionaryEntry & Entries
```

Retrieve the trade conditions dictionary.

Service REFERENTIAL / subscribe requests

NB: see file `samples/sample_cli/perform_refdownload.cpp`

Class API_REFERENTIAL_Download_base

Start a pseudo subscription to receive referential data: whole or part, incremental or full refresh.

```
void start (
      RequestHandler & handler,
      // inputs
      ListOfMarketBranchId const & TargetBranches,
      ListOfReferentialAttribute const & FilteringAttributes,
      ListOfReferentialTagNumber const & Format,
      Timestamp const & LastUpdateTimestamp,
      bool SendCreated = true,
      bool SendModified = true,
      bool SendDeleted = true
      );
```

Class INTERFACE_REFERENTIAL_Download_streaming

Receive a list of instruments, based on various criteria set in `start()` method above

```
void response_REFERENTIAL_Download_Started (
      // outputs
      Timestamp const & inLastUpdateTimestamp // timestamp of remote data
      );

void response_REFERENTIAL_Download_Failed (ReturnCode rc);

void aborted_REFERENTIAL_Download (ReturnCode rc); // RC_OK indicates end of download

void notif_BranchBegin (
      // event data
      MarketBranchId const & inBranch,
      uint32 inCurrentQuantity, // total number of instruments in this branch
      uint32 inDeletedQuantity, // how many deleted instruments we are going to receive
      uint32 inCreatedQuantity, // how many created instruments we are going to receive
      uint32 inModifiedQuantity // how many modified instruments we are going to receive
      );

void notif_InstrumentsDeleted (
      // event data
      ListOfFOSInstrumentCode const & inInternalCodes // codes of deleted instruments
      );

void notif_InstrumentsCreated (
      // event data
      ListOfInstrumentCharacteristics const & inInstruments // new instruments
      );

void notif_InstrumentsModified (
      // event data
      ListOfInstrumentCharacteristics const & inInstruments // modified instruments
      );
```

Service REFERENTIAL / subscribe requests

NB: see file `samples/sample_cli/perform_refdownload_and_subscribe.cpp`

Class API_REFERENTIAL_DownloadAndSubscribe_base

Start a subscription to receive first referential data: whole or part, incremental or full refresh, and then referential updates.

```
void start (
      RequestHandler & handler,
      // inputs
      ListOfMarketBranchId const & TargetBranches,
      ListOfReferentialAttribute const & FilteringAttributes,
      ListOfReferentialTagNumber const & Format,
      Timestamp const & LastUpdateTimestamp,
      bool SendCreated = true,
      bool SendModified = true,
      bool SendDeleted = true,
      bool SendOtherData = true
      );
```

Class INTERFACE_REFERENTIAL_DownloadAndSubscribe_streaming

Receive a list of instruments, based on various criteria set in `start()` method above

```
void response_REFERENTIAL_DownloadAndSubscribe_Started ();

void response_REFERENTIAL_DownloadAndSubscribe_Failed (ReturnCode rc);

void aborted_REFERENTIAL_DownloadAndSubscribe (ReturnCode rc);

void notif_BranchBegin (
      // event data
      MarketBranchId const & inBranch,
      uint32 inCurrentQuantity, // total number of instruments in this branch
      uint32 inDeletedQuantity, // how many deleted instruments we are going to receive
      uint32 inCreatedQuantity, // how many created instruments we are going to receive
      uint32 inModifiedQuantity // how many modified instruments we are going to receive
      );

void notif_InstrumentsDeleted (
      // event data
      ListOfFOSInstrumentCode const & inInternalCodes // codes of deleted instruments
      );

void notif_InstrumentsCreated (
      // event data
      ListOfInstrumentCharacteristics const & inInstruments // new instruments
      );

void notif_InstrumentsModified (
      // event data
      ListOfInstrumentCharacteristics const & inInstruments // modified instruments
      );

void notif_MarkerTimestamp ( Timestamp const & inMarkerTimestamp ); // marker
timestamp
```

```
void notif_RealtimeBegin (); // realtime events starts


void notif_VariablePriceIncrementTables (
      // event data
      ListOfVariableIncrementPriceBandTable const & inTables // variable tick sizes
      );


void notif_TradeConditionsDictionary (
      // event data
      ListOfTradeConditionsDictionaryEntry const & inEntries // trade conditions
      );


void notif_MetaData (
      // event data
      ListOfTagDeclaration const & inTags,
      ListOfEnumTypeDeclaration const & inEnum,
      ListOfString const & inProviders
      ); // meta data
```

Service QUOTATION / simple requests

NB: see `samples/sample_cli/perform_*.cpp`

API_QUOTATION_GetHistoryDaily

```
            // outputs
            FOSInstrumentCode InternalCode,
            FeedOS::Types::ListOfDailyHistoryPoint & Values,
            // inputs
            PolymorphicInstrumentCode const & Code,
            YMD_Date const & LocalBeginDate,
            YMD_Date const & LocalEndDate
```

Retrieve a list of daily Open/High/Low/Close values

API_QUOTATION_GetHistoryIntraday2

```
            // outputs
            FOSInstrumentCode & InternalCode,
            FeedOS::Types::ListOfIntradayHistoryPoint2 & Values,
            // inputs
            PolymorphicInstrumentCode const & Code,
            Timestamp const & ServerUTCTimestamp_begin,
            Timestamp const & ServerUTCTimestamp_end,
            int32 NbPoints = 0
```

Retrieve a list of intraday trades (timestamp / price / quantity)

API_QUOTATION_SnapshotInstrumentsL1

```
            // outputs
            FeedOS::Types::ListOfInstrumentStatusL1 & Result,
            // inputs
            FeedOS::Types::ListOfPolymorphicInstrumentCode const & Codes,
            FeedOS::Types::ListOfQuotationTagNumber const & OtherValues,
            bool IgnoreInvalidCodes = false
```

Retrieve quotation values for one or more instruments

Service QUOTATION / subscribe requests

Class API_QUOTATION_SubscribeInstrumentsL1_base

Start a subscription to receive real time quotation data for one or more instruments.

```
void start (
      RequestHandler & handler,
      // inputs
      ListOfPolymorphicInstrumentCode const & Codes,
      ListOfQuotationTagNumber const & OtherValuesToLookFor,
      QuotationContentMask ContentMask = QuotationContentMask_EVERYTHING, bool
      IgnoreInvalidCodes = false
      );
```

Class INTERFACE_QUOTATION_SubscribeInstrumentsL1_streaming

Receive streamed data related to subscription above

```
void response_QUOTATION_SubscribeInstrumentsL1_Started (
      // outputs
      uint32 inTicket,
      FeedOS::Types::ListOfInstrumentStatusL1 const & inSnapshot
      );

void response_QUOTATION_SubscribeInstrumentsL1_Failed (ReturnCode rc);

void aborted_QUOTATION_SubscribeInstrumentsL1 (ReturnCode rc);

void notif_TradeEventExt (
      // event data
      FOSInstrumentCode inCode,
      Timestamp const & inServerUTCTimestamp,
      Timestamp const & inMarketUTCTimestamp,
      FeedOS::Types::QuotationTradeEventExt const & inData
      );
```

NB: see file `samples/sample_cli/perform_subscribe.cpp` to see how to process the TradeEventExt messages.

Class API_QUOTATION_SubscribeInstrumentsMBL_base

Start a subscription to receive real time updates of order book for one or several instruments.

```
void  start (
      // inputs
      ListOfPolymorphicInstrumentCode const & Codes,
      ListOfOrderBookLayerId const & LayerIds = ListOfOrderBookLayerId(),
      IgnoreInvalidCodes = false
      );
```

Class INTERFACE_QUOTATION_SubscribeInstrumentsMBL_Streaming

Receive streamed data related to subscription above

```
void response_QUOTATION_SubscribeInstrumentsMBL_Started (
      // outputs
        uint32 inTicket,
        FeedOS::Types::ListOfFOSInstrumentCode const & inInternalCodes
      );

void response_QUOTATION_SubscribeInstrumentsMBL_Failed (
        ReturnCode rc
      );

void aborted_QUOTATION_SubscribeInstrumentsMBL(
        ReturnCode rc
      );

void notif_MBLFullRefresh  (
      // event data
        FeedOS::Types::ListOfMBLSnapshot const & inData
      );

void notif_MBLOverlapRefresh  (
      // event data
        FeedOS::Types::MBLOverlapRefresh const & inData
      );

void notif_MBLDeltaRefresh  (
      // event data
        FeedOS::Types::MBLDeltaRefresh const & inData
      );

void notif_MBLMaxVisibleDepth  (
      // event data
      FeedOS::Types::MBLMaxVisibleDepth const & inData
      );
```

NB: see file `samples/sample_cli/perform_subscribe_mbl.cpp` to see how to process the Refresh/DeltaRefresh messages.

class API_QUOTATION_SubscribeOneInstrumentMarketSheet_base

Start a subscription to receive full market sheet (order id-based) for one instrument

```
void  start (
      RequestHandler & handler,
      // inputs
      PolymorphicInstrumentCode const & Code
      );
```

class INTERFACE_QUOTATION_SubscribeOneInstrumentMarketSheet_streaming

Receive realtime data related to subscription above.
"market timestamps" may be null, depending on the market.

Market sheet delta updates, i.e. order creation/modification/removal, provide the following parameters:

- `MarketSheetLevel` : a vector index, zero based (zero means the top of the market sheet)
- `MarketSheetEntry` : it holds the main attributes of an order ; an order ID, a price and a quantity

```
void response_QUOTATION_SubscribeOneInstrumentMarketSheet_Started (
            // outputs
            FOSInstrumentCode inInternalCode,
            Timestamp const & inLastUpdateServerUTCTimestamp,
            Timestamp const & inLastUpdateMarketUTCTimestamp,
            FeedOS::Types::MarketSheet const & inSnapshot);

void response_QUOTATION_SubscribeOneInstrumentMarketSheet_Failed (
            ReturnCode rc);

void aborted_QUOTATION_SubscribeOneInstrumentMarketSheet (ReturnCode rc);

// insert a new order at the given level
void notif_NewOrder     (
            // event data
            FOSInstrumentCode inCode,
            Timestamp const & inServerUTCTimestamp,
            Timestamp const & inMarketUTCTimestamp,
            FeedOS::Types::ListOfQuotationContextFlag const & inContext,
            FeedOS::Types::FIXSide inSide,
            FeedOS::Types::MarketSheetEntry const & inEntry,
            FeedOS::Types::MarketSheetLevel inLevel);

// update the order at the given level, and move it if inNewLevel != inOldLevel
void notif_ModifyOrder  (
            // event data
            FOSInstrumentCode inCode,
            Timestamp const & inServerUTCTimestamp,
            Timestamp const & inMarketUTCTimestamp,
            FeedOS::Types::ListOfQuotationContextFlag const & inContext,
            FeedOS::Types::FIXSide inSide,
            FeedOS::Types::MarketSheetEntry const & inEntry,
            FeedOS::Types::MarketSheetLevel inOldLevel,
            FeedOS::Types::MarketSheetLevel inNewLevel);
```

```
// remove the order at the given level
void notif_RemoveOneOrder     (
            // event data
            FOSInstrumentCode inCode,
            Timestamp const & inServerUTCTimestamp,
            Timestamp const & inMarketUTCTimestamp,
            FeedOS::Types::ListOfQuotationContextFlag const & inContext,
            FeedOS::Types::FIXSide inSide,
            String const & inOrderID,
            FeedOS::Types::MarketSheetLevel inLevel);

// remove all orders from level 0 to inLevel (included)
void notif_RemoveAllPreviousOrders  (
            // event data
            FOSInstrumentCode inCode,
            Timestamp const & inServerUTCTimestamp,
            Timestamp const & inMarketUTCTimestamp,
            FeedOS::Types::ListOfQuotationContextFlag const & inContext,
            FeedOS::Types::FIXSide inSide,
            String const & inOrderID,
            FeedOS::Types::MarketSheetLevel inLevel);

// remove all orders of the given side (bid, ask or both)
void notif_RemoveAllOrders    (
            // event data
            FOSInstrumentCode inCode,
            Timestamp const & inServerUTCTimestamp,
            Timestamp const & inMarketUTCTimestamp,
            FeedOS::Types::ListOfQuotationContextFlag const & inContext,
            FeedOS::Types::FIXSide inSide);

// refresh the whole market sheet with the given snapshot
void notif_Retransmission     (
            // event data
            FOSInstrumentCode inCode,
            Timestamp const & inServerUTCTimestamp,
            Timestamp const & inMarketUTCTimestamp,
            FeedOS::Types::MarketSheet const & inSnapshot);

// some value(s) have been updated (typically the TradingStatus)
void notif_ValuesUpdateOneInstrument (
            // event data
            FOSInstrumentCode inCode,
            Timestamp const & inServerUTCTimestamp,
            Timestamp const & inMarketUTCTimestamp,
            FeedOS::Types::ListOfQuotationVariable const & inValues);
```

Service QUOTATION / on-the-fly modification of an ongoing SubscribeInstrumentsL1

These requests take the form of a "simple request", but they reference on ongoing SubscribeInstrumentsL1 subscription. They can be used to dynamically change parameters of the ongoing subscription.

The "ticket" number has to be stored when subscription has been started, in callback `response_QUOTATION_SubscribeInstrumentsL1_Started`

API_QUOTATION_ChgSubscribeInstrumentsL1Add

```
// outputs
FeedOS::Types::ListOfInstrumentStatusL1 & Snapshot,
// inputs
uint32 Ticket,
FeedOS::Types::ListOfPolymorphicInstrumentCode const & Codes,
bool IgnoreInvalidCodes = false
```

API_QUOTATION_ChgSubscribeInstrumentsL1Remove

```
// inputs
uint32 Ticket,
FeedOS::Types::ListOfPolymorphicInstrumentCode const & Codes
```

API_QUOTATION_ChgSubscribeInstrumentsL1NewContentMask

```
// outputs
FeedOS::Types::ListOfInstrumentStatusL1 & Snapshot,
// inputs
uint32 Ticket,
FeedOS::Types::QuotationContentMask ContentMask,
bool ResendSnapshot = false
```

Service QUOTATION / on-the-fly modification of an ongoing SubscribeInstrumentsMBL

The "ticket" number has to be stored when subscription has been started, in callback `response_QUOTATION_SubscribeInstrumentsMBL_Started`

API_QUOTATION_ChgSubscribeInstrumentsMBLAdd

```
// outputs
FeedOS::Types::ListOfFOSInstrumentCode & InternalCodes,
// inputs
uint32 Ticket,
FeedOS::Types::ListOfPolymorphicInstrumentCode const & Codes,
FeedOS::Types::ListOfOrderBookLayerId const & LayerIds =
ListOfOrderBookLayerId(),
bool IgnoreInvalidCodes = false
```

API_QUOTATION_ChgSubscribeInstrumentsMBLRemove

```
// inputs
uint32 Ticket,
FeedOS::Types::ListOfPolymorphicInstrumentCode const & Codes
```

Service QUOTATION / replaying market data feeds

"filtered" versions exist for L1 and L2. They allow filtering on the server side based on a list of instrument codes or markets ids. See files `framework/FeedReplay*`

Class FeedReplay_base (base class for replaying events)

```
// call this to start replaying
void  start (     RequestHandler & handler,
                  uint16 InternalSourceId,
                  Timestamp const & RangeBeginInternalDate,
                  Timestamp const & RangeEndInternalDate,
                  float64 AccelerationFactor = 0
            );

//override in case you are interested in low-level timestamps
virtual void hook_begin_frame_replay(
                        unsigned int seq_num,
                        unsigned int timestamp_seconds,
                        unsigned int timestamp_millisec,
                        unsigned int timestamp_microsec
                  );
```

Class FeedReplay_L1 (for replaying Trades/BestLimits events)

```
//constructor: give a consumer object
FeedReplay_L1 (INTERFACE_QUOTATION_SubscribeAllDataAndStatus_streaming & consumer);

// set filter on message types (default is to process all messages)
void set_filter_MessageTypes (
                  bool notif_TradeEvent,
                  bool notif_ValuesUpdate,
                  bool notif_MarketStatus,
                  bool notif_MarketNews
            );
```

Class FeedReplay_L2 (for replaying Order Book events)

```
//constructor: give a consumer object
FeedReplay_L2 (INTERFACE_QUOTATION_SubscribeAllOrderBook_streaming & consumer);
```

Class FeedReplay_observer (for monitoring feed replay progress)

```
//use FeedReplay_base::set_observer() to plug an observer
//into a replay object

      class FeedReplay_observer
      {
      public:
            virtual void hook_begin_frame_replay
                  (     FeedReplay_base const & replayer,
                        Timestamp const &       frame_timestamp
                  ) = 0;
      };
```