FACTSET 〉 SEE THE ADVANTAGE

FactSet Broadcast Feed API

C++ Programmer's Manual and Reference
Version 1.0G

Contents

Notice

This manual contains confidential information of FactSet Research Systems Inc. or its affiliates ("FactSet"). All proprietary rights, including intellectual property rights, in the Licensed Materials will remain property of FactSet or its Suppliers, as applicable. The information in this document is subject to change without notice and does not represent a commitment on the part of FactSet. FactSet assumes no responsibility for any errors that may appear in this document.

FactSet Consulting Services

North America - FactSet Research Systems Inc.	
United States and Canada	+1.877.FACTSET
Europe – FactSet Limited	
United Kingdom	0800.169.5954
Belgium	800.94108
France	0800.484.414
Germany	0800.200.0320
Ireland, Republic of	1800.409.937
Italy	800.510.858
Netherlands	0800.228.8024
Norway	800.30365
Spain	900.811.921
Sweden	0200.110.263
Switzerland	0800.881.720
European and Middle Eastern countries not listed above	+44.(0)20.7374.4445
Pacific Rim- FactSet Pacific Inc.	
Japan Consulting Services (Japan and Korea)	0120.779.465 (Within Japan) +81.3.6268.5200 (Outside Japan)
Hong Kong Consulting (Hong Kong, China, India, Malaysia, Singapore, Sri Lanka, and Taiwan)	+852.2251.1833
Sydney Consulting Services	1800.33.28.33 (Within Australia) +61.2.8223.0400 (Outside Australia)
E-mail Support	
	support@factset.com

Document Organization and Audience

This document is intended for application programmers that are familiar with C/C++ and Object-Oriented Systems. Its purpose is to fully describe the functionality contained within the FactSet DataFeed API. This document is intended to be read cover-to-cover, and then act as a reference guide to application developers using the FactSet DataFeed API.

- Chapter 1 - Introduces FactSet DataFeed API and defines key concepts and terminology.
- Chapter 2 - Explains how to build and link applications using this API.
- Chapter 3 - Describes the programming concepts at various stages of an application.
- Chapter 4 - Lists the complete Class Reference.

Document Convention

This document uses the following conventions:

- Code snippets use a courier 10 font - `FEConsumer::log_open()`
- The directory delimiter character follows the UNIX convention - forward slash ('/')
- Items of importance will be in boxes of following type:

> ❖ ***Important notations will be in this type of box.***

Trademarks

Acknowledgements

This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit (http://www.openssl.org).

Chapter 1 Introduction

1.1 The FactSet DataFeed API

The FactSet DataFeed API is a multi-platform C++ object-oriented framework which is used to communicate with a FactSet data source. The API assists developers with all aspects of communication, request/message processing, and subscription management. The classes simplify data access by providing asynchronous messages to application-defined callbacks.

The data source will authenticate as well as permission the various data sets available. Applications that attempt to connect without authorization will receive a connection error. Connected applications that request data to which they are not entitled - will receive an error message from the data source.

The data source is the FactSet Data Server, which is a back-end system that is hosted by FactSet. Connections to a FactSet Data Server occur over the Internet or a WAN via TCP/IP. Applications must give log in credentials (i.e., username, key, and counter), and the address information (i.e., IP and port number) for the FactSet Data Server.

1.2 **Terminology**

The following terminology is used throughout this documentation:

Terminology	Meaning
API	Application Programming Interface - a set of defined interfaces that applications use to extract information from the FactSet Data Server.
SDK	Software Development Kit - a collection of libraries, include files, documentation, and sample codes that make up this toolkit.
TCP/IP	Transport Control Protocol over Internet Protocol - the protocol that this API uses to communicate to the FactSet Data Server.
FactSet Data Server	A server which provides permissioned access to FactSet data.
FactSet Authentication Server	A server which returns configuration information to the API and permissions the application to access the FactSet Data Server.
FDS	Multiple meanings. FDS is the ticker symbol for FactSet Research Systems Inc. It may also stand for the FactSet Data Server. The meaning is defined by its context.
Consumer	Any application that uses this API.
Stream	A virtual tunnel of messages for a given request.
Callback	An application-defined function that is called by the API.
Closure	A user-defined void * pointer that is passed back to an application-defined callback.
FID	Field Identifier - an integer identifier that describes the encoding and business meaning of a field value.
Opaque Data	Data without a defined interpretation, which is simply a pointer and size to the data.

Field/Value Pairs	A self-describing message format used in API messages. Each pair contains a FID and some opaque data. The FID defines the type and meaning of the data.

1.3 **High Level Overview**

The following diagram shows the logical connections to the FactSet Data Server:

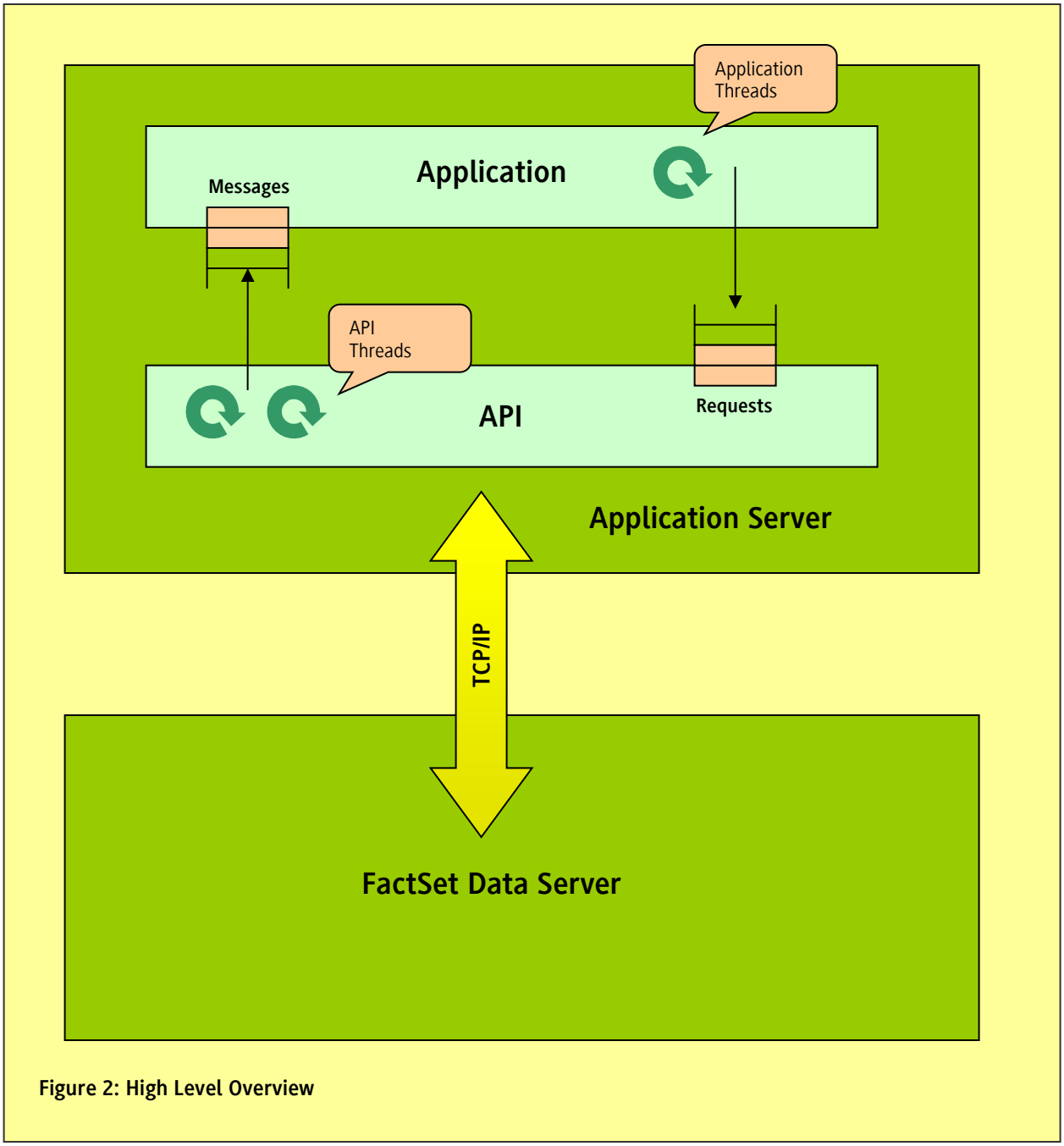

Figure 2: High Level Overview

Applications will use the interface defined by the API to do the following:

- **Authenticate with the Authentication Server**: Updates the API with some configuration information and enables connecting to the data server.
- **Connect to the Data Server**: This will initiate the TCP connection on which data will be sent.
- **Log in to the Data Server**: This will confirm a secure connection has been established for further communication.
- **Request Data**: Requests will be posted on a queue to be sent out via a communication thread.
- **Receive Messages via Event Handler**: Incoming messages will be returned via an application-defined callback called when dispatch is called.
- **Disconnect from the Data Server**: The application may disconnect from the Data Server at any time. Reconnecting will require beginning again from the authentication step.

1.4 API Core Functionality and Benefits

The API provides the following services to applications:

- Support for multiple development platforms
- Abstract the underlying TCP/IP connection
- TCP connection failure handling
- Simplified data access
- A consistent interface for opening and closing streams
- Subscription Management
- Logging
- Class-thread-safe, thread-aware

1.4.1 TCP/IP Communications

The API handles all aspects of the TCP/IP connection to the Data Server including problems related to asynchronous communication, byte-ordering, and the buffering needed when using stream-oriented protocols.

The API will detect TCP network failures and will notify the application of the condition.

The API will continuously retry the connection to the Data Server in the event of a TCP disconnect. Upon a successful reconnect, the current open streams will also be re-established.

Individual IPs will be communicated to clients upon trial;

Required Ports:

- 6670 - 6780

Access to the following addresses is also required:

- https://auth.factset.com/fetchotpv1
- https://lima-datafeed.factset.com/XMLTokenAuth

1.4.2 Simplified Data Access

The API delivers data using field/value pairs. The `MD::MD_Message` class allows applications to easily extract the data fields. This class supports both random and sequential access. Furthermore, the application can coerce the data values to required data types.

1.4.3 Request Consistency

The API provides a consistent interface for opening and closing subscriptions using a string identifier. The format of this identifier is PRODUCT_CODE[1]|SECURITY_TYPE|ISO_CODE|VENDOR_SYMBOL, and it is possible to subscribe to a prefix of this identifier to subscribe to everything matching that prefix[2]. To close a subscription, the application needs to pass the subscribed identifier or prefix back to the API.

1.4.4 Snapshotting

If requested, the initial message on the stream may contain all fields for all symbols requested to get current values. Subsequent messages will only contain the fields that have changed. This behavior will require an application to keep state of all the fields for a given a stream if necessary,

1.4.5 Logging and Configuration Management

To aid developers with troubleshooting and debugging, the API supports logging of error and informational messages to standard error (cerr). Applications can request that an actual log file be opened and messages be directed to that file.

Applications typically need to “soft-code” certain application settings. For example, the hostname of the FactSet Data Server should be stored in some configuration file or system registry.

1.4.6 Threading Support

This API is both thread-aware and in some cases thread-safe. Not all objects are thread-safe, but the entire API is thread-aware. The definitions of thread-aware and thread-safe are as follows:

Thread-aware: The code in question does not use static or global variables without the use of mutexes. All IN/OUT parameters are passed via the stack, and methods never return references to non-const static objects. These conventions allow objects of the same class to be independent of each other. ***All API classes are thread-aware, and multiple threads are allowed to operate on objects of the same class, provided that each thread is operating on its own object.*** However, thread-aware objects are not permitted to be operated on by multiple threads at-a-time without the use of a mutex. The notion of thread-aware is commonly called *class-thread-safe.*

---

[1] See the Data Service manual for a list of the FactSet product codes and security types

[2] While a prefix of the full identifier is valid, a prefix of a specific piece is not. For example, one may subscribe to “9001|” to receive messages for all security types, iso codes and symbols under the 9001 product code. However, one is unable to subscribe to “900” in order to receive messages for all product codes starting with those characters.

Thread-Safe: Multiple threads are allowed to operate on the same object. The only API class that is thread-safe is the `FEConsumer` class.

Chapter 2 Building Applications

2.1 Toolkit Organization

Directory/Filename	Contents	Additional Notes
RELNOTES.TXT	Contains the latest release notes for this version of the toolkit.	
VERSION.TXT	Contains the toolkit's version label and build number.	
include/	The API header files	
lib/	The API library files	
doc/	Documentation	This document along with additional supporting documentation.
sample/	Sample applications	

2.1.1 Supported Compilers, Operating Systems, and Architectures

Every effort has been made to test different compilers, versions, and architectures using this API. However, the C++ language does not dictate standards at the binary level. Compiler vendors are free to implement many of the standard C++ concepts in their own way. Some examples include exception handling, name mangling, and multiple inheritance implementations. Therefore, application developers may be forced to use a limited set of compilers supported by this toolkit.

The toolkit currently supports the following compilers:

- Linux x86_64 gcc 5.3 and higher, using the C++03 ABI

Additional platforms may be added in the future. Please contact FactSet Consulting Services if a platform does not appear in the list above.

2.2 Running Applications

Dynamic libraries need to be installed on any system that will execute applications built using the API. It is the responsibility of the application developer to ensure these libraries are available on all run-time systems.

2.2.1 UNIX Systems

Applications linked with UNIX dynamically linked libraries (so files), will use the path at link time to locate its shared libraries. If the production systems have these libraries installed in different locations, there are two options. The first is to set the linker flags to use a different runpath. The second is to set the LD_LIBRARY_PATH environment variable before starting the application. This variable should contain the path of the shared object file.

The dynamic loader utility, `ldd`, can be used to verify that all applications are able to locate all dependent shared objects.

2.2.2 OpenSSL and Security

Due to the changing nature of computer security, dependencies such as OpenSSL will not be bundled with the library so that users are free to update in case a vulnerability is discovered. Users are expected to maintain sufficiently new versions of these dependencies as required by the API for secure communication.

Chapter 3 Programming with the API

3.1 Program Setup and Initialization

3.1.1 Standard Conventions

This API is designed so that its interfaces adhere to a common set of standards. The following conventions are used by the FactSet real-time API:

- All methods are lower case with the '_' character to separate words.
- All methods that need to return an error do so via the `Error` class.
- All *IN* parameters are passed into API methods either by value, const reference, or const pointer. The only exception to this is the closure argument. It is passed as a non-const pointer although the API cannot modify the contents of this area.
- All *OUT* parameters are passed into the application via a non-const reference argument given by the application.
- All *IN/OUT* parameters are passed in by non-const reference.

3.1.2 Closure Arguments

It is common for API's that support callbacks to accept user arguments during callback setup. These arguments, also known as closure arguments, are passed back to the application as parameters to the callback function. All API functions that accept a callback, will also accept a `void *` closure argument. It is up to the application to define its meaning. The API treats this pointer as an opaque piece of data and will not modify its content.

3.1.3 A Complete Example

```
#include <string>
#include <iostream>
#include "FEConsumer.h"
#include "BroadcastFields.h"
using namespace FactSet::Datafeed;

void on_message(const std::string &topic, const FEConsumer::MsgType &msg, void *closure) {
    // Get sequence number
    int seq = -1;
    if(!msg.getInt(BroadcastFieldId::FDS_SEQ_NUM, seq)) {
        std::cout << "Message for " << topic << " missing sequence number"
                  << std::endl;
        return;
    }
    std::cout << "Message for " << topic << " with sequence number " << seq
```

```
                << std::endl;
}

int main(int argc, char **argv) {
    // Create consumer with 1 subscriber and 1 worker
    FEConsumer consumer(1, 1);
    // Authenticate
    Error err = consumer.authenticate("<host>", "<user>", "<serial>", "<key id>",
        "<key>","<counter>", "<path>", false);
    if(err != Error::NoError) {
        return -1;
    }
    // Connect
    err = consumer.connect();
    if(err != Error::NoError) {
        return -1;
    }
    // Log in
    err = consumer.log_in();
    if(err != Error::NoError) {
        return -1;
    }
    // Set a message callback for any topic
    FEConsumer::SocketType fd;
    err = consumer.register_callback("", &on_message, nullptr, fd);
    if(err != Error::NoError) {
        return -1;
    }
    // Subscribe to a topic
    std::string topic = "9001|1|USA|FDS";
    err = consumer.subscribe(topic, FEConsumer::REQUIRE_SNAPSHOT, FEConsumer::LIVE);
    if(err != Error::NoError) {
        return -1;
    }
    // Dispatch with 1000ms timeout
    while(true) {
        consumer.dispatch(1000);
    }
    return 0;
}
```

3.2 Connecting to a Data Source

An application connects to a data source during initialization. A connection to a FactSet Data Server occurs over the Internet or a WAN via TCP/IP. When connecting to a Data Server, applications should authenticate using the `authenticate()` or `authenticate_async()` function, then call the `connect()` or `connect_async()` function, and finally call the `log_in()` or `log_in_async()` function.

The `authenticate()`, `connect()`, and `log_in()` functions are synchronous, and in rare cases a call may block for an extended period of time (timeout duration is configurable via the `timeout_sec()` function on `FEConsumer`). If applications wish to use a non-blocking call the `authenticate_async()`, `connect_async()`, and `log_in_async()` functions are available.
Repeated calls to `connect()` or `log_in()` will return an error if the `FEConsumer` is already connected or logged in, respectively.

By default, the field_map.txt file will be requested and saved to etc/field_map.txt during log in. The failure to request or save this file will not prevent the log in attempt from succeeding. Requesting this file can be avoided by using log_in(false) or log_in_async(false). See *3.7 Requesting Files* for more information on file requests.

```
// Connect to DataFeed server synchronously
// Each of the following functions calls will block for up to
//   consumer.timeout_sec() seconds before timing out
// Each function will return an Error, which is ignored here for simplicity,
//   but should be handled in practice
Error err = consumer.authenticate("<host>", "<user>", "<serial>", "<key id>",
        "<key>","<counter>", "<path>", false);
err = consumer.connect();
err = consumer.log_in();

// Connect to DataFeed server asynchronously
consumer.event_cb(&on_event); // Set event callback
consumer.authenticate_async("<host>", "<user>", "<serial>", "<key id>",
        "<key>","<counter>", "<path>", false); // Begin asynchronous request chain

// Each Event will contain an error which can be obtained using the error() member
// function, which is ignored here for simplicity, but should be handled in
// practice
static void on_event(const Event &e, void *closure) {
    switch(e.type()) {
    case Event::AUTHENTICATE:
        consumer.connect_async();
        break;
    case Event::CONNECT:
        // This event only exists to indicate whether the connect attempt was
        // successful or not, but the consumer is not connected until the
        // CONNECTION_CHANGED event indicates as such
        break;
    case Event::CONNECTION_CHANGED:
        bool connected;
        e.connection_changed_results(connected);
        if(connected && !consumer.logged_in()) {
```

```
            consumer.log_in_async();
        }
        break;
    case Event::LOGIN:
        // Assuming no error, the login has completed successfully here
        break;
    }
}
```

A synchronous connect operation will block until the connection is established. If a synchronous connect operation fails, applications must do one of the following:

1. Retry the connect operation at some future time.
2. Connect asynchronously.
3. Exit the application.

> ❖ ***integrators are expected to limit the number of connection retires in case of failures to avoid unnecessary load on the DataFeed servers. Abusing the services may result in the account beeing locked down without any prior notice. If there are any questions on the design of the service please reach out to your FactSet representitive.***

An asynchronous connect operation will return immediately and will always return a CONNECT event. If the returned CONNECT event contains an error, a connection will never get established. In this case, the application should log the error and exit.

Upon returning from a successful asynchronous connect operation, the connection will be processed by an API thread. A CONNECTION_CHANGED event will be raised whenever the connection state changes, including immediately after a new connection is established.

> ❖ ***If connect() or connect_async() return an error, the connection will never get established. Applications must issue a successful connect before logging in, subscribing to topics, and receiving any messages. This behaviour is true for both asynchronous and synchronous connections.***

3.2.1 Authentication

Using a counter file

authenticate("address", "client", "AAAA", “”, “”, ".", false); – The API will connect to the specified IP address on port 667*. It will use a username of “client” and a One Time password generated by the key and counter as per 3.2.1.1 located in a file named AAAA.data located in the current directory.

Using parameters

authenticate("address", "client", "AAAA", “Key”, “Counter”, ".", true); – Same as above, however the given key and counter will be used instead of one located in ./AAAA.data. A new counter file will be created at ./AAAA.data (or overwritten, if one previously existed) containing the given key and current counter, for future use.

Session File

After a successful authentication, a session will be created and store in a session file. This file will be stored in the same directory as the counter file and will be named <Key Id>-Session.data (e.g. AAAA-Session.data). A session is valid for up to 12 hours before expiring. While valid, the previous session will be used to authenticate instead of the key and counter to prevent needlessly running up the counter value.

3.2.1.1 Retrieving the One Time Password

The authentication protocol for Exchange Datafeed is using One Time password. At the initial setup the key administartor[3] will need to follow the below steps to generate the key and counter required to autheticate with OTP.

1. Go to http://auth-setup.factset.com
2. Login using the FactSet .NET account received in the welcome email.
3. Enter the serial number tied to the server account used to connect to the feed.
4. Make sure the PROD is selected, rather than BETA.
5. Click Get New Key.
6. Create a new file - On the first line, copy and paste the “Key” from the web site (don’t include the word “Key:”, just the actual string).
7. On the second line, copy and paste the counter value.
8. Save this file as <KeyId>.data. Most likely that will be “AAAA.data” and use this file as input in the authenticate function as per above.
9. Alternatively take note of the values and use directly in authenticate.

3.3 Subscriptions

3.3.1 Message Callback

In order to receive messages, a message callback must be set. It is not necessary to set this callback before subscribing, but it is a good idea to do so, otherwise some messages may be missed before the callback can be updated.

[3] The key administrator needs to be given access to be able to generate the key, contact your FactSet representative to get the required access enabled.

To add or remove a message callback, the `FEConsumer::register_callback()` and `FEConsumer::unregister_callback()` functions can be used respectively. These functions are defined as follows:

```
Error register_callback(const std::string&          topic_prefix,
                        FEConsumer::MessageCallback callback,
                        void*                       closure
                        SocketType&                 fd_out);
Error unregister_callback(const std::string& topic_prefix);
Error unregister_callback(SocketType fd);
```

The first parameter is the topic prefix. Any incoming message that begins with this prefix will be delivered with this callback. If multiple prefixes match the message topic, the one which matches the most characters will be used. Callbacks can be unregistered by their topic prefix.

The second parameter is the message callback. The callback function must have the following signature:

```
void (*) (const std::string &topic, const MsgType &msg, void *closure);
```

The third parameter is a closure to return in the callback as described in section 3.1.2. Note that it is necessary for the callback function to accept a closure even if no closure will be provided.

The final parameter is an output parameter indicating a file descriptor. On Linux, this is of type int. This file descriptor will become readable when messages are available on the queue for this callback. Note that this file descriptor is strictly a signaling mechanism and should not be used to read from directly, instead use the dispatch() method as described in section 3.3.5. Callbacks can be unregistered by their file descriptor.

3.3.2 Subscribing

Requests are made using the `FEConsumer::subscribe()` or `FEConsumer::subscribe_async()` function. The subscribe functions are defined as follows:

```
Error subscribe(const std::string& topic,
                SnapshotMode       snapshot_mode,
                DataMode           data_mode);
void subscribe_async(const std::string& topic,
                     SnapshotMode       snapshot_mode,
                     DataMode           data_mode);
```

The first parameter required by the subscribe method is the topic string. The topic string contains 4 pipe-delimited fields, and may be formatted as follows :

PRODUCT_CODE
PRODUCT_CODE|SECURITY_TYPE
PRODUCT_CODE|SECURITY_TYPE|ISO_CODE
PRODUCT_CODE|SECURITY_TYPE|ISO_CODE|VENDOR_SYMBOL

Subscriptions operate based on a prefix search. For example, if the topic string was a particular PRODUCT_CODE|SECURITY_TYPE|ISO_CODE combination, all messages matching that combination will be received regardless of their individual symbols. Note that attempting to make overlapping subscriptions will return an error. For example, if there is an active subscription to “9001|1|USA|IBM” attempting to subscribe to “9001|1|USA” will return an error, because both subscriptions will include messages with topic “9001|1||USA|IBM.”

The second parameter indicates whether snapshot messages are required (REQUIRE_SNAPSHOT) or not (NO_SNAPSHOT). If this is set to REQURE_SNAPSHOT, then incoming market data messages on that subscription will be queued until the cached snapshot messages are received for each topic in the subscription. After all snapshots are received, the message callback will be called for each queued message that is newer than the snapshot message for its topic. If the queued message was older than the snapshot message, it will simply be dropped.

The final parameter indicates whether data received should be live data (LIVE, aka “real-time”), delayed data (DELAYED), or prerecorded data (CANNED).

3.3.3 Snapshot Queue

While waiting for snapshots, incoming market data messages will be queued as described in the above section. It is possible to set the size of this queue using the `FEConsumer::max_snapshot_queue_size()` function. The function definition is as follows:

```
void max_snapshot_queue_size(unsigned int size);
```

The maximum queue size must be changed before subscribing in order for that subscription to be affected by the changes. It is possible to change the maximum queue size after subscriptions have been made, however only future subscriptions will be affected. Once the maximum queue size has been reached, the oldest queued messages will be dropped in favor of new messages.

3.3.4 Unsubscribing

To cancel an existing subscription, the `FEConsumer::unsubscribe()` or `FEConsumer::unsubscribe_async()` functions can be used. These are defined as follows:

```
Error unsubscribe(const std::string &topic);
void unsubscribe_async(const std::string &topic);
```

The topic passed into these functions must be the same topic that was given during the original subscription.

3.3.5 Dispatching

In order to have the callbacks called to actually receive messages, the `FEConsumer::dispatch()` or `FEConsumer::dispatch_fd()` functions can be used. These are defined as follows:

```
Error dispatch(long timeout_ms);
```

```
Error dispatch_fd(SocketType fd);
```

The first of these can be used to poll and dispatch from every callback queue with a single call to the API. The poll will wait up to timeout_ms milliseconds for messages to become available before timing out. If a timeout occurs, this will not be considered an error and `Error::NoError` will be returned. Using this function is equivalent to using an external polling system to poll all file descriptors returned by `FEConsumer::register_callback()` and calling `FEConsumer::dispatch_fd()` on each file descriptor that is readable.

The second of these can be used to immediately dispatch from a specific callback queue. The argument provided for fd should be a file descriptor that was previously returned by `FEConsumer::register_callback()`. It should be ensured that the given file descriptor is readable before calling this function.

It is possible to retrieve the file descriptor for the callback with a given topic prefix from the API by using the `FEConsumer::get_notify_socket()` function. This is defined as follows:

```
Error get_notify_socket(const std::string& topic_prefix, SocketType& fd_out);
```

It should be noted that the file descriptor for a given topic will not become readable as soon as messages become available by default. Instead, the notify will occur after a certain number of messages become available or a certain amount of time has passed. These values can be configured by using the following functions:

```
void set_notify_queue_message_limit(size_t limit);
void set_notify_queue_time_limit_ms(int limit);
```

Furthermore, messages will move through two of these batching queues before finally being delivered to the application. Due to this, if there is some limitation on the maximum latency before messages are delivered, the time limit should be set to half of this maximum.

3.4 Processing Events

3.4.1 Event Callback

In order to receive events, the event callback must be set. To set the callback, the `FEConsumer::event_cb()` function can be used. The function definitions are as follows:

```
void event_cb(EventCallback callback);
void event_cb(EventCallback callback, void *closure);
```

The first parameter is the event callback. The callback function must have the following signature:

```
void (*)(const Event &e, void *closure);
```

The second parameter, which is optional, is a closure to return in the callback as described in section 3.1.2. Note that it is necessary for the callback function to accept a closure even if no closure will be provided.

3.4.2 Event Spawning

Each asynchronous function has an associated event type which will be spawned after that function has completed. Although synchronous functions will not spawn events, it is still possible to receive a CONNECTION_CHANGED event while using exclusively synchronous functions if the toolkit gets disconnected (or reconnects) to the data server. The following table lists the possible event types and their sources:

Event Type	Source
STOP	FEConsumer::stop_async()
AUTHENTICATE	FEConsumer::authenticate_async()
CONNECTION_CHANGED	Can occur at any time when a connection is established or lost
CONNECT	FEConsumer::connect_async(). Note that this event does not indicate that a connection has been established, only that the connection request has completed.
DISCONNECT	FEConsumer::disconnect_async()
LOGIN	FEConsumer::log_in_async()
SUBSCRIBE	FEConsumer::subscribe_async()
UNSUBSCRIBE	FEConsumer::unsubscribe_async()
SET_MSG_CB	FEConsumer::set_msg_cb_async()

3.4.3 Event Handling

Event objects each contain an `Error`, `EventType`, and unique string id which can be accessed using the `Event::error()`, `Event::type()`, and `Event::id()` member functions, respectively.

Additionally, certain event types may contain some additional results regarding the completed operation. The only event type which has additional results is the `CONNECTION_CHANGED` event and these results can be obtained using the following function:

```
Error connection_changed_results(bool &currently_connected) const;
```

An error will be returned if the event is not a CONNECTION_CHANGED event or if the event is somehow malformed. The provided currently_connected reference will be set to true if the consumer is connected after this event. In the case of an error occurring, the contents of currently_connected are undefined and should be ignored.

3.5 Processing Messages

3.5.1 FID Value Pairs

The API makes use of the widely accepted standard of representing data as field/value pairs. This self-describing data structure tags all data elements with an integer identifier (FID or field identifier).

The value is typically some opaque binary data and its associated size. Every field/value pair has an agreed-upon meaning by both the data sources and the consuming applications. This meaning can never be changed once published to the applications. Furthermore, the values are rarely null-terminated. This allows data values to contain binary data. Applications should never assume null-terminated field values, unless the publishing data-source makes this guarantee.

3.5.2 Field Identifiers

The current field identifiers are available in a standard C++ include file named BroadcastFields.h. This file defines human-readable static constant integers for the current list of known field ids. This is the usual method of identifying a field by name in actual C++ code.

3.5.3 Messages

Active subscriptions will return `MD::MD_Message` messages, which is simply a container of fields (i.e., fids and values). The fields can be extracted using the member functions defined in the `MD::MD_Message` class (see section *4.2 Fields and Messages* for more information).

3.6 Threading

3.6.1 Thread-safe Classes

The only class that is completely thread-safe is the `FEConsumer` class. This class manages all interactions with the FactSet Data Server. Applications are free to call the methods of the `FEConsumer` class using multiple threads.

3.6.2 Thread-unsafe Classes

The remaining classes are thread-unsafe. The reason is that these classes tend to be used by a single thread at a time. The `MD::MD_Message`, `Error`, and `Event` classes are all container classes that are usually used by a single thread. Applications should provide their own locking if these objects need to be shared by multiple threads.

3.6.3 Class-thread-safe

Multiple threads are allowed to access different objects of the same class without locking. Creation and destruction of objects are also thread-safe. This is known as being class-thread-safe. All API classes are class-thread-safe.

- ***Applications must link with a thread-safe runtime library.***

3.7 Requesting Files

Configuration files can be requested by using `FEConsumer::request_file()` or `FEConsumer::request_file_async()`. The function definitions are as follows:

```
Error request_file(const std::string& filename, std::string& contents_out);
void request_file_async(const std::string& filename);
```

Filename indicates the name of the file to request. Valid filenames are as follows:

- field_map.txt
- product_codes.txt
- exchange_data.txt

For synchronous requests, contents_out will be set to the contents of the requested file if no error occurs. For asynchronous requests, `Event::get_value("Contents")` can be used to retrieve these contents.

Chapter 4 API Class Reference

4.1 API Constants

4.1.1 Error Codes

All error codes within the API are conveyed to the application via the Enumeration `Error::ErrorCode`. The list of possible errors is noted below. API methods that need to return error information will do so using the `Error` class.

> ❖ ***The FactSet API does not throw standard or non-standard exceptions. However, the underlying system libraries may throw exceptions such as std::bad_alloc.***

```
// include file:  "Error.h"
namespace FactSet {
    namespace Datafeed {
        class Error {
            enum ErrorCode {
                FE_NO_ERROR,     // All is good
                FE_E_UNKNOWN,    // Unknown error
                FE_E_NO_SERV,    // No service available
                FE_E_NOT_FOUND, // Item was not found
                FE_E_RENAME,     // Item has been renamed
                FE_E_EXISTS,     // Item already exists
                FE_E_LIMIT,      // Maximum application limit has been reached
                FE_E_PROTOCOL,   // Any protocol error (e.g. message, file format,
                                 // network)
                FE_E_INVAL,      // Invalid parameter to method call
                FE_E_RESOURCE,   // Operating system resource exhausted
                FE_E_NO_CONN,    // No connection to the server
                FE_E_VERSION,    // Incorrect version
                FE_E_SHUTDOWN,   // User has shut down the system
                FE_E_ACCESS,     // Permission denied
                FE_E_TIMEOUT     // Operation timed out
            };
        }
    } // namespace Datafeed
} // namespace FactSet
```

4.1.2 Field Identifiers

Field identifiers are integers that can be used to index into `MD::MD_Message` objects. The list of field identifiers is available programmatically in the include file: "**BroadcastFields.h**" Applications that need to reference field identifiers by a symbolic name can do so by including this field identifier file.

4.2 FID Fields and Messages

The `MD::MD_Message` class is the class which represents all messages in the system. The API delivers `MD::MD_Message` references to client callback routines. This class contains information applicable to all messages. Messages also contain a collection of FID fields.

4.2.1 FID Fields

A FID field is data that is identified by an integer. The data is opaque (i.e., binary data with a size).

4.2.2 Messages

The `MD::MD_Message` class can be created by applications. In this case, the lifetime of these objects are strictly managed by the creator. Typically, applications will be given references to `MD::MD_Message` objects as callback parameters. These objects are owned by the API and their lifetime is valid during the callback routine only. If applications wish to extend the lifetime, they should make copies of the object using the copy constructor provided.

MD::MD_Message Interface

The following methods allow the application to query various pieces of information:

- `bool empty() const;` - Returns true if the given message contains no fields
- `bool exists (FieldID id) const;` - Returns true if the given field is present in the message
- `uint32_t getSize() const;` - Returns the size of the message in bytes
- `iterator begin() const;` - Returns beginning iterator for stl style iteration
- `iterator end() const;` - Returns end iterator for stl style iteration

The following methods allow for reading the FID field data:

```
bool getInt      (FieldID id, int         &result) const;
bool getInt64    (FieldID id, int64_t     &result) const;
bool getDouble   (FieldID id, double      &result) const;
bool getString   (FieldID id, String      &result) const;
bool getString   (FieldID id, std::string &result) const;
bool getEpochNS  (FieldID id, MD_EpochNS  &result) const;
```

The get(…) methods listed above get data values identified by the integer (field identifier or FID) and store it in the given reference. If this operation is successful, they will return true. A similar set of methods also exist for adding and updating fields. The appropriate function should be used to read data based on that field's real type as listed in the comments located in BroadcastFields.h. *Note that getString(FieldID, std::string &) is a convenience wrapper that calls getString(FieldID, String&) and assigns the value into an std::string.*

4.3 FEConsumer

The `FEConsumer` class manages the connection to the FactSet Data Server. It is the class used for all interaction with the data server. This class is the heart of the FactSet Real-Time API.

4.3.1 Creating and Destroying the FEConsumer

A number of constructors are available for creating the `FEConsumer`, listed below:

```
FEConsumer(int sub_count, int worker_count, int high_watermark);
FEConsumer(int sub_count, int worker_count);
```

For both of these functions, sub_count indicates the number of subscriber threads to be created while worker_count indicates the number of worker threads (see the following section 4.3.2 for details). The additional parameter high_watermark indicates the maximum number of messages to store on subscriber and worker messages queues before dropping messages. If no high_watermark is given, it will be set to unlimited.

To shut down the consumer gracefully, the `FEConsumer::stop()` functions can be used. These functions have the following definitions:

```
Error stop();
void stop_async();
```

These functions will completely stop the operation of the consumer, close all connections, and end all background threads.

4.3.2 Subscribers and Workers

Subscribers are threads which directly communicate with the data server. Subscriptions will be evenly distributed among all subscribers in a round robin manner. When a subscriber receives a message, it will immediately pass this message to a worker for processing. The chosen worker is determined by the topic of the message, such that messages of the same topic will always be passed to the same worker. After the worker has finished its processing, the message will be stored in a callback specific queue based on its topic until it is finally dispatched by the application.

4.3.3 Logging

The `FEConsumer` class will log errors and a number of informational messages as it performs operations. By default, these messages are output to the standard error stream. It is possible to redirect these messages to a file using the `FEConsumer::log_open()` function. The function definition of this function is as follows:

```
static Error log_open(const std::string &filename, bool append);
```

The first parameter indicates the file path and filename of the file where output will be redirected. This can either be an absolute path or a relative path from the application execution directory. By default, if this file already exists it will be renamed to <filename>.old and the previous .old file, if it exists, will be deleted. If it is more desirable to append to the previous file instead, the append argument can be set to true.

Additionally, a previously opened log file may be closed using the `FEConsumer::log_close()` function. This function simply causes the `FEConsumer` to revert back to logging to standard error.

4.3.4 Synchronous vs. Asynchronous Interface

The `FEConsumer` class contains both a synchronous and asynchronous interface for operations that will not return immediately. Both interfaces may be used by the same application and may be used interchangeably depending on requirements and preference. Although both interfaces can be used to accomplish the same tasks, there are a few differences between them, which are listed in the following table.

Synchronous	Asynchronous
Block until the operation times out or completes successfully or unsuccessfully, returning an Error to indicate status	Returns immediately without blocking, returning void
Does not spawn events (however, CONNECTION_CHANGED events may still spawn)	Spawns an event after the operation times out or completes successfully or unsuccessfully
Can not be called from callback threads	**Can** be called from callback threads

4.3.5 Operation Timeouts

As mentioned in the previous section, operations are able to time out if they take too long. The timeout duration can be set using the `FEConsumer::timeout_sec()` function. The function definition is as follows:

```
void timeout_sec(unsigned int seconds);
```

The only parameter to this function is the number of seconds to wait before considering an operation to be timed out. After timing out, synchronous functions will return an `Error` with the code `Error::FE_E_TIMEOUT`. If an asynchronous call was made instead, an event with the appropriate `EventType` will be spawned with the same `Error::FE_E_TIMEOUT` error. If it is desired for operations to never time out, the constant `FEConsumer::NO_TIMEOUT` can be passed to this function. Changing the timeout duration will only affect calls made after the timeout was changed.

4.3.6 Querying Values

It is possible to retrieve the values for many of the settings available in the `FEConsumer` class. These functions will either return the default value for the given setting or the value as it was previously set by the application. The functions are as follows:

`bool connected() const;` - Returns true if the consumer is currently connected to the data server
`bool authenticated() const;` - Returns true if the consumer has complete authentication
`bool logged_in() const;` - Returns true if the consumer is logged in
`unsigned int timeout_sec() const;` - Returns the number of seconds to wait before operations are timed out
`unsigned int max_snapshot_queue_size() const;` - Returns the maximum number of streaming messages to queue before dropping old messages in favor of new ones while waiting for snapshot data to be received.