FACTSET 〉 SEE THE ADVANTAGE

FactSet Broadcast DataFeed API

Java Programmer's Manual

Version 1.0A

Last updated: 7th December 2021

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.

Revision History

Effective Date	Version Number	Sections	Changes made
12/07/2021	1.0A		Initial release

Table of Contents

Notice 2
Revision History 2
Document Organization and Audience 5
Chapter 1 Introduction 6
1.1 The FactSet DataFeed API 6
1.2 Terminology 6
1.3 High Level Overview 8
1.4 API Core Functionality and Benefits 9
1.4.1 TCP/IP Communications 9
1.4.2 Simplified Data Access 10
1.4.3 Request Consistency 10
1.4.4 Snapshotting 10
1.4.5 Logging and Configuration Management 10
1.4.6 Threading Support 11
Chapter 2 Building Applications 12
2.1 Toolkit Organization 12
Maven distribution: All dependency JARs are available in a lib directory 12
Dependencies included distribution: 12
Uber JAR distribution: 13
2.1.1 Supported Build Environments 13
2.2 Running Applications 13
Chapter 3 Programming with the API 14
3.1 Program Setup and Initialization 14
3.1.1 Standard Conventions 14
3.1.2 Closure Arguments 14
3.1.3 A Complete Example 14
3.2 Connecting to a Data Source 16
3.2.1 Authentication 18
3.3 Subscriptions 20
3.3.1 Message Callback 20
3.3.2 Subscribing 21

3.3.3 Snapshot Queue 21
3.3.4 Unsubscribing 22
3.3.5 Dispatching 22
3.4 Processing Events 22
3.4.1 Event Callback 22
3.4.2 Event Spawning 22
3.4.3 Event Handling 23
3.5 Processing Messages 24
3.5.1 FID Value Pairs 24
3.5.2 Field Identifiers 24
3.5.3 Messages 24
3.6 Threading 25
3.6.1 Thread-safe Classes 25
3.6.2 Thread-unsafe Classes 25
3.6.3 Class-thread-safe 25
3.7 Requesting Files 25
Chapter 4 API Class Reference 26
4.1 API Constants 26
4.1.1 Error Codes 26
4.1.2 Field Identifiers 26
4.2 FID Fields and Messages 27
4.2.1 FID Fields 27
4.2.2 Messages 27
4.3 FEConsumer 28
4.3.1 Constructing the FEConsumer 28
4.3.2 Subscribers and Workers 29
4.3.3 Logging 29
4.3.4 Synchronous vs. Asynchronous Interface 29
4.3.5 Operation Timeouts 30
4.3.6 Querying Values 30
Global Client Support 31
Trademarks 31

Document Organization and Audience

This document is intended for application programmers that are familiar with Java 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.

Chapter 1 Introduction

1.1 The FactSet DataFeed API

The FactSet DataFeed API is a multi-platform Java object-oriented framework that 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 authenticates and provides permissions to the various data sets available. Applications that attempt to connect without authorization receive a connection error. When connected and applications request data to which they are not entitled, they receive an error message from the data source.
The data source is the FactSet Data Server, which is a back-end system hosted by FactSet. Connections to a FactSet Data Server occur over the Internet or a WAN through TCP/IP. Applications must provide the following to access the FactSet Data Server:

- Log in credentials: username, key, and counter.
- Address information: IP and port number.

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, includes 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	Server that provides permissioned access to FactSet data.
FactSet Authentication Server	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	Application that uses this API.

Stream	Virtual tunnel of messages for a given request.
Callback	Application-defined function that is called by the API.
Closure	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 to the size of the data.
Field/Value Pairs	Self-describing message format used in API messages. Each pair contains an 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:

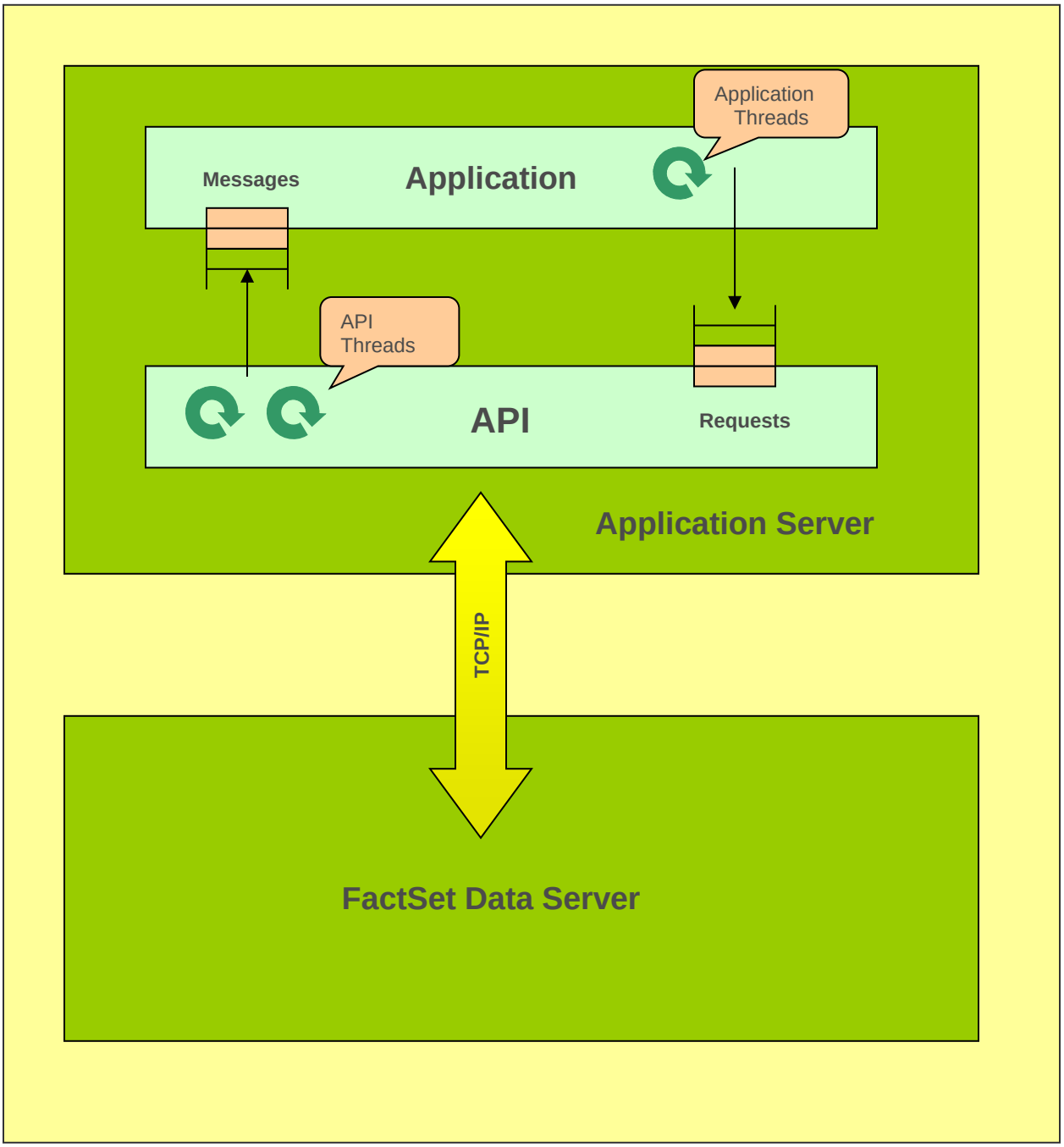

Figure 2: High Level Overview

Applications 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 connection to the data server.
- **Connect to the Data server**: Initiates the TCP connection on which data will be sent.
- **Log in to the Data server**: Ensures that a secure connection has been established for further communication.
- **Request Data**: Requests will be posted on a queue to be sent out through a communication thread.
- **Receive Messages Through Event Handler**: Incoming messages are returned through an application-defined callback when dispatch is called.
- **Disconnect from the Data server**: When an application disconnects from the Data server abruptly due to any reason, you must start from the authentication step to reconnect.

1.4 API Core Functionality and Benefits

The API provides the following services to applications:

- Support for multiple development platforms
- Abstracts the underlying TCP/IP connection
- TCP connection failure handling
- Simplified data access
- 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 the stream-oriented protocols.

The API detects TCP network failures and notifies the network condition to the application.

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 are re-established.

Individual IPs will be communicated to clients upon product 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 the 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 into the 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. You can subscribe to a prefix of this identifier and also subscribe to everything that matches a particular prefix. To close a subscription, the application must pass the subscribed identifier or prefix 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 contain only the fields that have changed. This behavior will require an application to keep the state of all the fields for a given 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 using the Log4j2 logging facade. Any logging implementation can be used with this façade by including its dependencies. By default, the Log4j2 API uses a SimpleLogger that simply prints logs with ERROR level or higher to the console. However, applications must use a custom implementation specific to their needs. It is recommended to use the Log4j-core implementation, but any other implementation can be used with a bridge dependency. Here is an example of the dependencies needed for the SLF4J implementation using Maven as a build tool:

```
<!-- This first API dependency is optional, since the FactSet API already includes it, but it is recommended for clarity -->
<dependency>
   <groupId>org.apache.logging.log4j</groupId>
   <artifactId>log4j-api</artifactId>
   <version>2.13.3</version>
</dependency>
<dependency>
   <groupId>org.apache.logging.log4j</groupId>
   <artifactId>log4j-to-slf4j</artifactId>
```

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

```
   <version>2.13.3</version>
</dependency>
<dependency>
   <groupId>org.slf4j</groupId>
   <artifactId>slf4j-api</artifactId>
   <version>1.7.9</version>
</dependency>
<dependency>
   <groupId>org.slf4j</groupId>
   <artifactId>slf4j-jdk14</artifactId>
   <version>1.7.28</version>
</dependency>
```

Logging configuration is done through the chosen implementation. For other application configurations, 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

The Threading Support 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 through the stack, and the 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 by multiple threads at-a-time without the use of a mutex. The notion of thread-aware is commonly called *class-thread-safe.*
- **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

There are three different distributions available depending on the application's needs and build environment.

Maven distribution: All dependency JARs are available in a lib directory.

Directory/Filename	Contents	Additional Notes
README.md	General API information and installation/build instructions	
docs/	JavaDoc documentation for the API	
samples/	Sample applications for the API	Each sample project contains a target/bin directory with the pre-built JARs for all samples
toolkit/	API core	Has all JARs and pom.xml files that need to be installed locally. Also includes install scripts

Dependencies included distribution:

Directory/Filename	Contents	Additional Notes
README.md	General API information and build instructions	
docs/	JavaDoc documentation for the API	
samples/	Sample applications for the API	Each sample project contains a bin directory with the pre-built JARs for all samples
lib/	API dependency libraries	Includes the API JARs themselves in addition to all dependency JARs

Uber JAR distribution:

Directory/Filename	Contents	Additional Notes
README.md	General API information and build instructions	
docs/	JavaDoc documentation for the API	
samples/	Sample applications for the API	Each sample project contains a bin directory with the pre-built JARs for all samples
bdf-toolkit.jar	Uber JAR for the API containing all dependencies	

2.1.1 Supported Build Environments

This API is compiled using OpenJDK 14. It is highly recommended that this JDK version is used for the development of any application using this API. The JDK versions may work, but there is a chance of failure when used with other versions. It is known to fail when you build or run an application with a Java Runtime Environment from Java 8 or earlier.

The recommended build automation tool for applications using this API is Apache Maven. While not mandatory, using Apache Maven as a build tool allows an application to leverage the Maven distribution of this API for ideal dependency management. For any other build tool, a different distribution should be chosen and dependencies in the distribution must be added to a project's build path manually. If Maven is chosen as a build tool, including the API in an application consists of running the install script included in the distribution to install the dependencies to the local repository, and then including the following dependency in the pom.xml file:

```
<dependency>
   <groupId>com.factset.bdf</groupId>
   <artifactId>bdf-toolkit</artifactId>
   <version>${version}</version>
</dependency>
```

2.2 Running Applications

All applications with a dependency on this API must use the Java Runtime Environment included with OpenJDK 14 to run. When using Apache Maven as a build automation tool, the API dependency must be included as shown in the section above for the API to be available within the application. When using any other build tool, the API JARs and all dependency JARs must be included as External Archives in the application's Build Path. Additionally, there must be a field map file for static initialization of the application's *FieldMap* located at *etc/field_map.txt* relative to the execution directory.

Chapter 3 Programming with the API

3.1 Program Setup and Initialization

3.1.1 Standard Conventions

The 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 camelCase.
- All methods, which need to return an error, return through the Error class.

3.1.2 Closure Arguments

It is common for APIs, which support callbacks, to accept user arguments during callback setup. These arguments, also known as closure arguments, are passed to the application as parameters to the callback function. All API functions that accept a callback have an additional function signature that will also accept an Object closure argument. It is up to the application to define its meaning. The API treats this Object as an opaque piece of data and will not modify its content.

3.1.3 A Complete Example

```
import java.net.InetSocketAddress;
import java.nio.file.Path;

import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;

import com.factset.bdf.toolkit.FDSOTP;
import com.factset.bdf.toolkit.FDSUser;
import com.factset.bdf.toolkit.FEConsumer;
import com.factset.bdf.toolkit.Topic;
import com.factset.bdf.toolkit.FEConsumer.DataMode;
import com.factset.bdf.toolkit.FEConsumer.DataSource;
import com.factset.bdf.toolkit.DispatchOptions;
import com.factset.bdf.toolkit.Error;
import com.factset.message_lib.md.MDMessage;

public class Example {
   private static final Logger logger = LogManager.getLogger(Example.class);

   public static void onMessage(Topic topic, MDMessage msg) {
      logger.info("Topic: {}\nMessage: {}", topic, msg);
   }
```

```
public static void main(String[] args) {
    String server = "<server>";
    int port = 0;
    FDSUser user = new FDSUser("<user>", "<serial>");
    FDSOTP otp = new FDSOTP(user, "AAAA", Path.of("etc"));

    // Construct an FEConsumer with 1 worker
    FEConsumer consumer = FEConsumer.newBuilder().workers(1).build();

    // Authenticate
    Error err = consumer.authenticate(new InetSocketAddress(server, port), otp);
    if (err != null) {
        logger.error(err.getDescription());
        consumer.close();
        return;
    }

    // Connect
    err = consumer.connect();
    if (err != null) {
        logger.error(err.getDescription());
        consumer.close();
        return;
    }

    // Log In
    err = consumer.login();
    if (err != null) {
        logger.error(err.getDescription());
        consumer.close();
        return;
    }

    // Set a default message callback for all data sources
    consumer.registerDefaultCallback(DataSource.All, (topic, msg) -> onMessage(topic, msg));

    // Subscribe to a topic
    err = consumer.subscribe(new Topic("9001|1|USA|FDS"), DataMode.RealTime, true);
    if (err != null) {
        logger.error(err.getDescription());
        consumer.close();
```

```
            return;
        }

        // Dispatch continuously, blocking up to 1 second for each call if no
        // messages are available
        DispatchOptions opts = new DispatchOptions(1000);
        while (true) {
            consumer.dispatch(opts);
        }
    }
}
```

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 through TCP/IP. When connecting to a Data Server, applications must authenticate using the authenticate() or authenticateAsync() function, and call the connect() or connectAsync() function, and then call the login() or loginAsync() function.

The authenticate(), connect(), and login() functions are synchronous. In few cases, a call may block for an extended period of time (timeout duration is configurable through the setEventTimeout() function on FEConsumer). If applications want to use a non-blocking call, the authenticateAsync(), connectAsync(), and loginAsync() functions are available.

Repeated calls to connect() or login() 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 make the request or save this file does not prevent the log in attempt from succeeding. You can avoid requesting this file by using login("") or loginAsync(""). See 3.7 Requesting Files for more information on file requests.

```
// Set up connection parameters
String server = "<server>";
int port = 0;
FDSUser user = new FDSUser("<user>", "<serial>");
FDSOTP otp = new FDSOTP(user, "<keyId>", Path.of("<path>"));

// Connect to DataFeed server synchronously
// Each of the following functions calls will block for up to
//   consumer.getEventTimeout() milliseconds 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(new InetSocketAddress(server, port), otp);
```

```
err = consumer.connect();
err = consumer.login();

// Connect to DataFeed server asynchronously
// Set event callback
consumer.setEventCallback((e) -> onEvent(e));
// Begin asynchronous request chain
consumer.authenticateAsync(new InetSocketAddress(server, port), otp);

// Each Event will contain an Error which can be obtained using the getError()
// function, which is ignored here for simplicity, but should be handled in
// practice
public static void onEvent(Event e) {
   switch(e.getCause()) {
   case Authenticate:
      consumer.connectAsync();
      break;
   case Connect:
      consumer.loginAsync();
      break;
   case Login:
      // Assuming no error, the login has completed successfully here
      break;
   }
}
```

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

- Retry the connect operation at some future time.
- Connect asynchronously.
- Exit the application.

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

An asynchronous connect operation returns immediately and will always create a Connect event. If the returned Connect event contains an error, a connection never gets established. In this case, the application must log the error and exit.

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

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

3.2.1 Authentication

One-Time Password (OTP) authentication is used when connecting with the API. An FDSOTP authentication instance can be constructed as follows:

FDSUser user = new FDSUser(“<username>”, “serial”);
FDSOTP otp = new FDSOTP(user, “<keyId>, e.g. AAAA”, Path.of(“etc”));

The path is where the key/counter file is located, and it is also where a session file will be created (see below).

Authenticating using a counter file

consumer.authenticate(new InetSocketAddress("<hostAddress>", port), otp); – The API connects to the specified IP address on the specified 667* port. It uses a FactSet user name and serial number provided for the FDSUser 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 etc/ directory.

Using parameters

otp.overwriteWith(“<key>”, “<counter>”);
consumer.authenticate(new InetSocketAddress("<hostAddress>", port), otp); – Same as above. However, the given key and counter is used instead of one located in etc/AAAA.data. A new counter file is created at etc/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 stored in a session file. This file will be stored in the same directory as the counter file and will be named <keyId>-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 uses the One-Time Password. During the initial setup, the key administartor must follow the steps below 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** to generate a KeyID, Key, and Counter.

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

To receive messages, a message callback must be set. It is recommended to set the message callback before you subscribe to avoid missing some messages if the callback is not updated.

You can use FEConsumer.registerDefaultCallback() to add a fallback message callback and FEConsumer.unregisterDefaultCallback() to remove a fallback message callback.

You can use the FEConsumer.registerCallback() function to add a message callback and FEConsumer.unregisterCallback() function to remove a message callback. These functions are defined as follows:

```
Error registerDefaultCallback(DataSource source, BiConsumer<Topic, MDMessage> callback);
Error unregisterDefaultCallback(DataSource source);
Error registerCallback(Topic t, DataSource source, BiConsumer<Topic, MDMessage> callback);
Error unregisterCallback(Topic t, DataSource source);
```

There are additional functions for registering callbacks with a closure object, which are similar to the above methods. The DataSource parameter in these functions allows the application to differentiate between different message sources. For example, market data source, on-demand snapshot source and recovery source.

The Topic parameter specifies the topic prefix that the callback must be applied to. Any incoming message that begins with this prefix is delivered with this callback. If multiple prefixes match the message topic, the one which matches the most characters is used. Callbacks can also be unregistered by their topic prefix. The BiConsumer<Topic, MDMessage> parameter is the message callback. It is recommended to use lambdas to pass in the callbacks. For example:

```
void onMessageReceived(Topic topic, MDMessage msg);
consumer.registerDefaultCallback(DataSource.All, (t, m) -> onMessageReceived(t, m));
```

If you are using a closure callback, the approach looks like the below:

```
void onMessageReceived(Topic topic, MDMessage msg, Object closure);
consumer.registerDefaultCallback(DataSource.All, (t, m, c) -> onMessageReceived(t, m, c), new
SomeCustomObject());
```

The closure parameter is described in section 3.1.2.

The registered callbacks are invoked when executing the dispatch() method as described in section 3.3.5.

3.3.2 Subscribing

You can subscribe through the FEConsumer.subscribe() or FEConsumer.subscribAsync() function. The subscribe functions are defined as follows:

```
Error subscribe (Topic t, DataMode mode, boolean requestSnapshot);
void subscribeAsync (Topic t, DataMode mode, boolean requestSnapshot);
```

The first parameter required by the subscribe method is the topic prefix. The topic prefix contains four pipe-delimited fields, and can 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 prefix was a particular PRODUCT_CODE|SECURITY_TYPE|ISO_CODE combination, all messages matching that combination are received regardless of their individual symbols.

Note: If you attempt to make overlapping subscriptions, it will return an error. For example, if there is an active subscription to “9001|1|USA|IBM” and if you are attempting to subscribe to “9001|1|USA”, it returns an error. The reason for this is that both subscriptions include messages with the topic “9001|1||USA|IBM.”

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

The third parameter indicates whether snapshot messages are required or not. If this is set to true, then the incoming market data messages on that subscription are queued until the cached snapshot messages are received for each topic in the subscription. After all the snapshots are received, the message callback is called for each queued message that is received after the snapshot message for its topic. If the queued message is received before the snapshot message, it will simply be dropped.

3.3.3 Snapshot Queue

While you wait for the snapshots, the incoming market data messages are queued as described in the above section. It is possible to set the size of this queue using the FEConsumer.setMaximumSnapshotQueueSize() function. The function definition is as follows:

```
void setMaximumSnapshotQueueSize(int queueSize);
```

Each worker with the FEConsumer class has its own queue. The maximum queue size is per-worker. Once the maximum queue size is reached, the oldest queued messages are dropped in favor of new messages. All the messages for a specific topic always go to the same worker. Hence, there is no risk of dropping messages in the wrong order when queues fill up. Changing the maximum queue size has an immediate effect on all active and future requests.

3.3.4 Unsubscribing

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

```
Error unsubscribe(Topic topic);
void unsubscribeAsync(Topic topic);
```

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

3.3.5 Dispatching

To have the callbacks called to receive and process messages, the FEConsumer.dispatch() functions can be used. These are defined as follows:

```
Error dispatch();
Error dispatch(DispatchOptions options);
```

The plain dispatch call, shown first, dispatches with the default options. It can be used to pool and dispatch from every callback queue with a single call to the API. If there are no messages available in any queues, then the call is not blocked and returns immediately.

Note: If you do not have any available messages, it does not constitute an error.

The second call enables you to customize the dispatching options. One such customization is a dispatch timeout. The timeout is the maximum amount of time to wait, in milliseconds, when there are no messages available. The dispatch call continues to poll all queues until the timeout is reached. A timeout is not an Error in this case, and it simply means that there are no messages to process. Another customization option is to specify specific data sources and topics on which to dispatch. If either or both are specified, the dispatch call only looks at the queues with a matching data source and/or topic.

3.4 Processing Events

3.4.1 Event Callback

To receive and process events, the event callback must be set. To set the callback, the FEConsumer.setEventCallback() functions can be used. The function definitions are as follows:

```
void setEventCallback(Consumer<Event> cb);
void setEventCallback(BiConsumer<Event, Object> cb, Object closure);
```

In these methods, the Consumer and BiConsumer parameters are the Functional Interfaces packaged with Java. It is recommended to use lambdas when setting these callbacks. The closure parameter in the second function is a closure to return in the callback as described in section 3.1.2.

3.4.2 Event Spawning

Each asynchronous function has an associated event that is spawned after that function is completed. Although synchronous functions will not spawn events, it is possible to receive a ConnectionChanged, SnapshotComplete, and RecoveryComplete event while using exclusively synchronous functions.

ConnectionChanged is spawned if the toolkit gets disconnected (or reconnects) from the data server. SnapshotComplete and RecoveryComplete are spawned when the corresponding request is completed. RequestSnapshot and Recovery are spawned by async calls when the request is completed, being processed, or sent.

The following table lists the possible event causes and their sources:

Event Cause	Source
Unknown	Unknown
Authenticate	FEConsumer.authenticateAsync()
ConnectionChanged	Can occur at any time when a connection is established or lost
Connect	FEConsumer.connectAsync()
Disconnect	FEConsumer.disconnectAsync()
Login	FEConsumer.loginAsync()
Subscribe	FEConsumer.subscribeAsync()
Unsubscribe	FEConsumer.unsubscribeAsync()
RequestFile	FEConsumer.requestFileAsync()
ConnectionRecovery	The API completed a connection recovery attempt after losing a connection
RequestSnapshot	FEConsumer.requestSnapshotAsync()
SnapshotComplete	A Snapshot request has completed, either as part of a normal subscription or for an on-demand snapshot request
Recovery	FEConsumer.recoverAsync()
RecoveryComplete	Recovery request is completed

3.4.3 Event Handling

Event objects contain an Error, Cause, and unique string ID. These are accessed using the Event.getError(), Event.getCause(), and Event.getId() member functions respectively. If the Event corresponds to an operation that is successfully completed, Event.getError() returns null.

Additionally, certain events may contain some additional information regarding the completed operation, such as the input parameters and result information. This additional information is stored as key-value String pairs. An individual value is accessed using Event.getData(), and the entire key-value map of Event data can be accessed using Event.getDataMap().

3.5 Processing Messages

3.5.1 FID Value Pairs

The API uses 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 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

All fields are loaded into the static FieldMap when the application starts. A field identifier is obtained using the FieldMap.getId() function and by providing any String field name as a parameter.

3.5.3 Messages

Active subscriptions return MDMessage messages that are simply a container of fields. For example. FIDs and values). The fields can be iterated and extracted using the member functions defined in the MDMessage 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

All the classes except the FEConsumer class are thread-unsafe where all of these classes tend to be used by a single thread at a time. The MDMessage, Error and Event classes are all container classes that are usually used by a single thread. Applications must 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. Construction of objects is also thread safe. All API classes are class-thread-safe.

3.7 Requesting Files

You can request configuration files through FEConsumer.requestFile()or FEConsumer.requestFileAsync(). The function definitions are as follows:

```
String requestFile(String filename);
String requestFile(String filename, boolean download);
void requestFileAsync(String filename);
void requestFileAsync(String filename, boolean download);
```

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

field_map.txt

product_codes.txt

exchange_data.txt

For synchronous requests, the returned string is the contents of the requested file if there is no occurrence of an error. If an error occurs, the function returns Null. For example, you can use the Event.getData(“Contents”) function for asynchronous requests to retrieve these contents. If the download parameter is set to true, the file is also downloaded to the configuration directory that has been set for the consumer.

Chapter 4 API Class Reference

4.1 API Constants

4.1.1 Error Codes

All error codes within the API are communicated to the application through the Enumeration Error.Code. The list of possible errors is noted below. The API methods return the error information using the Error class.

❖ The FactSet API will only return exceptions in the case of invalid user inputs following Java conventions. The primary examples for invalid inputs are providing a nonexistent file path when setting a consumer configuration or providing null as the value for a required parameter in an API method.

Event Cause	Source
Unknown	Unexpected failure due to unknown reasons
Connection	Connection to the server failed
Protocol	Error occurred in the communication protocol
Access	The user lacks necessary permissions for the requested operation
DuplicateEntry	The output of this operation already exists
NotFound	Required resource not found
InvalidTopic	Requested topic was invalid
InvalidOperation	Requested operation was not valid for the state of the toolkit
Timeout	Requested operation timed out
UnexpectedSubscribe	Unexpected subscribe response was received from the server
UnexpectedUnsubscribe	Unexpected unsubscribe response was received from the server

4.1.2 Field Identifiers

Field identifiers are integers that can be used to index into MDMessage objects. Applications can use the FieldMap.getId() function to reference field identifiers by a symbolic name and by providing the symbolic name as a String argument.

4.2 FID Fields and Messages

The MDMessage class represents all messages in the system. The API delivers MDMessage 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

An FID field is data that is identified by an integer. The data is opaque. For example, binary data with a size.

4.2.2 Messages

When the API delivers MDMessage references to client callback routines, it does not alter the message. In other words, it is safe for an application to store and use an MDMessage later.

MDMessage Interface

MDMessage extends Iterable<Field>, which means that the fields in a message can be iterated over using the standard Java loops, either a for loop, a combination of iterators or a while loop.

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

- boolean empty(); - Returns True if the given message contains no fields
- boolean exists (short fieldId); - Returns True if the given field is present in the message
- int getSize(); - Returns the size of the message in bytes
- int fieldCount(); - Returns the number of fields in the message
- Iterator<Field> iterator(); - Returns beginning Field iterator
- Field getField(short fieldId); - Returns the field in the message with the given field ID
- Field getField(String fieldName); - Returns the field in the message with the given symbolic name

The following methods allow access to the data in the Field class:

- short id(); - Returns the field ID
- String name(); - Returns the field name
- FieldMap.FieldType type(); - Returns the type of the value held in the field
- int size() ; - Returns the size, in bytes, of the data for the field value
- Object get(); - Returns the data value held in the field
- Character getChar(); - Returns the Character value held in the field
- Short getShort(); - Returns the Short value held in the field
- Integer getInt(); - Returns the Integer value held in the field
- Long getLong(); - Returns the Long value held in the field
- Float getFloat(); - Returns the Float value held in the field
- Double getDouble(); - Returns the Double value held in the field
- String getString(); - Returns the String value held in the field

- byte[] getBinaryBuffer(); - Returns the byte[] value held in the field
- DateTime getDateTime(); - Returns the DateTime value held in the field
- EpochNS getEpochNS(); - Returns the EpochNS value held in the field

The getter methods listed above allow an application to parse the data for any Field into its correct type. If for some reason the application does not care about types, it can use the get() method to return the value of any field as an Object. If using an incorrect type getter on a Field (e.g. using getInt() on a String field), the getter will return null. If the application is unsure of the type of a field, it can use the type() method to obtain it.
Here is an example of extracting the Integer value for the PRODUCT_CODE field in an MDMessage:
message.getField("PRODUCT_CODE").getInt();

4.3 FEConsumer

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

4.3.1 Constructing the FEConsumer

All instances of FEConsumer should be constructed using the FEConsumer.Builder class. This builder class allows for flexible configurations while constructing a consumer. Configurations can be applied manually by chaining the methods of a Builder or they can be automatically applied from a JSON config file. The FEConsumer will be constructed when calling FEConsumer.Builder.build(). The examples below show examples of manual configuration with a config file.

Manual Configuration

```
FEConsumer.newBuilder().workers(3).snapshotTimeout(10000).eventTimeout(5000).zmqHeartbeatInterval(5000).zmqHeartbeatTimeout(15000).build();
```

Using a Configuration File

```
FEConsumer.newBuilder().apply("etc/consumer.json");
```

Where the contents of etc/consumer.json are:

```
{
   "feconsumer": {
      "workers": 3,
      "snapshotTimeout": 10000,
      "eventTimeout": 5000
   },
   "zmq": {
      "heartbeatInterval": 5000,
      "heartbeatTimeout": 15000
   }
}
```

FEConsumer implements the AutoCloseable interface, so it can be used in a try-with-resources block without worrying about resource cleanup. To shut down the consumer manually, the FEConsumer.close() functions can be used. These functions have the following definitions:

```
void close();
void closeAsync();
```

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 that 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 immediately passes this message to a worker for processing. The chosen worker is determined by the topic of the message so that the messages of the same topic are passed to the same worker. When the worker finishes its processing, the message is stored in a callback specific queue based on its topic until it is finally dispatched by the application.

4.3.3 Logging

There are many reasons why the FEConsumer class may log messages, ranging from trace messages to error messages. Logging setup and configuration is discussed in section 1.4.5.

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 (Error is null for successful operations)	Returns immediately without blocking, returning void
Does not spawn events (however, there are still events that are not specific to any operation that may occur without any application interaction)	Spawns an event after the operation times out or completes successfully or unsuccessfully, which will be processed using the Event callback
Cannot be called from within the Event callback	**Can** be called from within the Event callback

4.3.5 Operation Timeouts

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

```
void setEventTimeout(int timeout);
```

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 return an Error with the code Error.Code.Timeout. If an asynchronous call was made, an event with the appropriate Event.Cause is spawned with the same Error.Code.Timeout error. If you want the operations to never time out, a value of 0 can be passed to this function. Changing the timeout duration only affects the 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 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:

- boolean isAuthenticated(); - Returns True if the consumer has successfully authenticated.
- boolean isConnected(); - Returns True if the consumer has successfully connected to the server.
- boolean isLoggedIn(); - Returns True if the consumer is logged in.
- boolean isSubscribed(Topic t); - Returns True if the consumer is subscribed to the given topic.
- int getEventTimeout(); - Returns the number of seconds to wait before operations are timed out.
- int getHeartbeatTimeout(); - Returns the heartbeat timeout in seconds for the consumer's connection with the server.
- int getHeartbeatInterval(); - Returns the heartbeat interval in seconds for the consumer's connection with the server.
- int getSnapshotTimeout(); - Returns the number of seconds to wait before a snapshot operation is timed out.
- int getRecoveryTimeout(); - Returns the number of seconds to wait before a recovery operation is timed out.
- int getMaximumSnapshotQueueSize(); - Returns the maximum number of streaming messages for the queue for each worker before dropping old messages in favor of new ones while waiting for snapshot data to be received.
- Path getConfigFilesDirectory(); - Returns the directory Path where config files should be downloaded.

Global Client Support

If you have any questions, submit a request through https://issuetracker.factset.com under the “Broadcast Streaming” category.

If you do not have login credentials for Issue Tracker, Email to datafeed_support@factset.com.

For general assistance, contact your local FactSet Consultant or Salesperson or Email support@factset.com.

Trademarks

FactSet is a registered trademark of FactSet Research Systems, Inc.

Microsoft is a registered trademark, and Windows is a trademark of Microsoft Corporation.

All other brand or product names may be trademarks of their respective companies.