FactSet DataFeed API
Java Programmer’s Manual
Version 4.0

Table of 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

<table>
<thead>
<tr>
<th>North America - FactSet Research Systems Inc.</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>United States and Canada</td>
<td>+1.877.FACTSET</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Europe – FactSet Limited</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>United Kingdom</td>
<td>0800.169.5954</td>
</tr>
<tr>
<td>Belgium</td>
<td>800.94108</td>
</tr>
<tr>
<td>France</td>
<td>0800.484.414</td>
</tr>
<tr>
<td>Germany</td>
<td>0800.200.0320</td>
</tr>
<tr>
<td>Ireland, Republic of</td>
<td>1800.409.937</td>
</tr>
<tr>
<td>Italy</td>
<td>800.510.858</td>
</tr>
<tr>
<td>Netherlands</td>
<td>0800.228.8024</td>
</tr>
<tr>
<td>Norway</td>
<td>800.30365</td>
</tr>
<tr>
<td>Spain</td>
<td>900.811.921</td>
</tr>
<tr>
<td>Sweden</td>
<td>0200.110.263</td>
</tr>
<tr>
<td>Switzerland</td>
<td>0800.881.720</td>
</tr>
<tr>
<td>European and Middle Eastern countries not listed above</td>
<td>+44.(0)20.7374.4445</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pacific Rim- FactSet Pacific Inc.</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Japan Consulting Services (Japan and Korea)</td>
<td>0120.779.465 (Within Japan) +81.3.6268.5200 (Outside Japan)</td>
</tr>
<tr>
<td>Hong Kong Consulting (Hong Kong, China, India, Malaysia, Singapore, Sri Lanka, and Taiwan)</td>
<td>+852.2251.1833</td>
</tr>
<tr>
<td>Sydney Consulting Services</td>
<td>1800.33.28.33 (Within Australia) +61.2.8223.0400 (Outside Australia)</td>
</tr>
</tbody>
</table>

| E-mail Support                                | support@factset.com |
Document Organization and Audience

This document is intended for application programmers that are familiar with Java and object-oriented programming. Its purpose is to fully describe the functionality contained within the FactSet DataFeed Java API. This document is intended to be read cover-to-cover as a tutorial supplement to the included javadocs.

- Chapter 1 introduces the FactSet DataFeed Java API and defines key concepts and terminology
- Chapter 2 explains how to build applications using this API
- Chapter 3 describes the programming concepts at various stages of an application
- Chapter 4 describes the Permissioning Service

Document Convention

This document uses the following conventions:

- Code snippets use a courier 10 font - `consumer.connect()`
- 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.

Minimum System Requirements

The following is required in order to use the Java Toolkit and connect to FactSet systems:

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Minimum Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td>Version 8</td>
</tr>
<tr>
<td>TLS</td>
<td>1.2</td>
</tr>
</tbody>
</table>

Trademarks

FactSet® is a registered trademark of FactSet Research Systems, Inc.
UNIX® is a registered trademark of The Open Group.
Linux® is the registered trademark of Linus Torvalds in the U.S. and other countries.
Windows® is a registered trademark of Microsoft in the U.S. and other countries.
Oracle® and Java® are registered trademarks of Oracle and/or its affiliates.
All other brand or product names may be trademarks of their respective companies.
Chapter 1 Introduction

1.1 The FactSet DataFeed API

The FactSet DataFeed Java API is a multi-platform object-oriented Java framework used to communicate with the FactSet Data Server. 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 interface instances.

Applications have two choices when connecting to a data source: a FactSet Data Server or the local FactSet workstation. The chosen data source will authenticate as well as permission users for the various data sets available. Applications that attempt to connect without authorization will receive a connection error. Connected applications that request data they are not entitled to will instead receive an error message from the data source.

The first data source option is a FactSet Data Server, which is a system hosted by FactSet. Connections to a FactSet Data Server occur over the Internet or through a WAN via TCP/IP. Applications connect with a username, password, and the address information (i.e., IP and port number) for the FactSet Data Server. Alternatively, applications may also connect with a one-time password.

The second data source option is a local FactSet workstation, which uses the user’s existing FactSet terminal installation along with the permissions tied to that user’s serial number. Connections to a local FactSet workstation occur on the user’s local machine via inter-process communication and TCP/IP. Applications connect with a username and serial number. This configuration is designed for the consuming application to receive data just for local use on the workstation, not for sharing data to any other users.

Figure 1: Two API clients connected to the two different FactSet data sources
1.2 Terminology

The following terminology is used throughout this documentation:

<table>
<thead>
<tr>
<th>Terminology</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Programming Interface - a set of defined interfaces that applications use to extract information from the FactSet Data Server.</td>
</tr>
<tr>
<td>SDK</td>
<td>Software Development Kit - a collection of libraries, include files, documentation, and sample codes that make up this toolkit.</td>
</tr>
<tr>
<td>XML</td>
<td>eXtensible Markup Language - a defined standard for exchanging information. The information contains markup tags used to describe the data values.</td>
</tr>
<tr>
<td>TCP/IP</td>
<td>Transport Control Protocol over Internet Protocol - the protocol that this API uses to communicate to the FactSet Data Server.</td>
</tr>
<tr>
<td>FactSet Data Server</td>
<td>A server which provides permissioned access to FactSet data.</td>
</tr>
<tr>
<td>FDS</td>
<td>Multiple meanings. FDS is the ticker symbol for FactSet Research Systems Inc. Also, it may stand for the FactSet Data Server. The meaning is defined by its context.</td>
</tr>
<tr>
<td>Service</td>
<td>A data source or supplier identified by a string name.</td>
</tr>
<tr>
<td>FDS1</td>
<td>FactSet’s Streaming Production Data Service. For a complete description of the data fields, types, and possible values see the FactSet Data Service Specification.</td>
</tr>
<tr>
<td>FDS_FUND</td>
<td>FactSet’s Fundamental Data Service. Used for End of Day data.</td>
</tr>
<tr>
<td>FDS_C</td>
<td>FactSet’s Canned Data Service. Recorded data is replayed, used for testing.</td>
</tr>
<tr>
<td>FDS_PERM</td>
<td>FactSet’s Permission Service. Used by third party integrators to enforce end-users Exchange permissions using the Workstation Entitled API setup.</td>
</tr>
<tr>
<td>Consumer</td>
<td>Any application that uses this API.</td>
</tr>
<tr>
<td>Stream</td>
<td>A virtual tunnel of messages for a given request.</td>
</tr>
<tr>
<td>Subscription</td>
<td>A Java object encapsulating an active request to a data source</td>
</tr>
<tr>
<td>Callback</td>
<td>An application-defined function that is called by the API.</td>
</tr>
<tr>
<td>FID</td>
<td>Field Identifier - an integer identifier that describes the encoding and business meaning of a field value.</td>
</tr>
<tr>
<td>Opaque Data</td>
<td>Data without a defined interpretation, which is simply a pointer and size to the data.</td>
</tr>
<tr>
<td>Field/Value Pairs</td>
<td>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.</td>
</tr>
</tbody>
</table>
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:

- **Connect to the Data Server**: This will initiate the TCP connection and start an internal communication thread within the API.
- **Request Data**: Requests will be posted on a queue to be sent out via the communication thread.
- **Receive Messages via Interface Instances**: Incoming messages will be posted to a message queue by the communication thread. The application will call an API method to dispatch any available messages to the associated interface callback method. All interface callback methods will be invoked in the context of an application thread.
- **Disconnect from the Data Server**: The application may disconnect from the Data Server at any time. This will destroy the communication thread as well.

### Information on API Threads

The only application defined method that the API threads may invoke is the `onNotify` method of the `NotifyHandler` interface. This method should be used to notify application threads of incoming messages and no API methods or blocking methods should be invoked within `onNotify`. It is up to the application to give control back to the API (via the `RT_Consumer.dispatch` methods), in order to receive messages via callback methods.

The API will create one thread per `RT_Consumer` instance. This thread serves as a communication thread and is responsible for all of the TCP/IP communication with the data server. The thread will be created when the application connects to the FactSet Data Server, and destroyed when the application calls `RT_Consumer.disconnect`.

### 1.4 API Core Functionality and Benefits

The API provides the following services to applications:

- Abstract the underlying TCP/IP connection
- TCP connection failure handling
- Simplified data access
- Consistent interface for opening and closing streams
- Subscription management
- Logging
- Thread-safety

#### 1.4.1 Support for Multiple Development Platforms

Multiple development environments are supported by the API. This toolkit supports any development environment that has a fully functional Java run time environment (i.e., version 8 or higher) implementation.
1.4.2 TCP/IP Communication

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 all open streams of the condition (i.e., each stream will receive a stale message). Applications only need to monitor the individual streams, and not the connection as a whole.\(^1\)

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. Refresh data will be sent and each open stream will transition from a stale to a non-stale state.

Required Ports:

- tcp/6681 – Connection to Exchange DataFeed Server
- tcp/443 – Web-based authentication

api(-stage).df.factset.com and canned-stage.df.factset.com: tcp/6681 need to be opened outbound-initiated for subnets:

- 192.234.235.0 (255.255.255.0)
- 64.209.89.0 (255.255.255.0)

1.4.3 Security Protocols

Clients should not hardcode dependencies on any specific security protocol as FactSet is continuously reviewing security policies and reserves the right to disable support for older security protocols with short notice\(^2\). The current supported protocols are TLSv1.1 and TLSv1.2 but at a future date, these may be replaced with future versions. Clients should make sure that their software can handle ever changing Security Protocols.

1.4.4 Simplified Data Access

The API delivers data using field/value pairs. The RT_Message class allows applications to easily extract the data fields. This class supports both random and sequential access.

1.4.5 Request Consistency

The API provides a consistent interface for opening and closing streams. All requests will receive a Subscription instance associated with a virtual stream. This applies to both static (i.e., snapshot requests) and streaming requests. In addition, messages on any given stream are identified by the Subscription instance. To close a stream, the application should invoke the cancel() method on the desired Subscription instance.

---

\(^1\) The API does inform the application about the connection status as a whole, and the application can use this information in any way it sees fit.

\(^2\) As of 29-Jul-2017 support for security protocol TLSv1.0 is disabled and requests using this TLS version will fail.
1.4.6 Subscription Management

The API allows applications to request duplicate data items. Each item will receive its own Subscription instance. Although the virtual streams are independent, they will receive identical messages. However, there will be only a single stream to the data server. This optimization saves both CPU resources and network bandwidth.

1.4.7 Logging

To aid developers with troubleshooting and debugging, the API supports logging of error and informational messages to a java.util.logging.Logger object. Applications can add handlers to the logger to enable additional logging output (such as a text file). In addition, applications using the API are able to post events to the logger.

1.4.8 Threading Support

All classes in this API are thread-safe. Multiple threads are allowed to operate at the same instance.
Chapter 2 Building Applications

2.1 Toolkit Organization

The DataFeed Java toolkit is available for downloading here: [https://www.factset.com/download](https://www.factset.com/download). Depending on the desired platform, installation is available as part of the Windows Toolkit Installer, or as a standalone gzip file for Linux. The contents of the file follow the directory hierarchy outlined in the following table.

<table>
<thead>
<tr>
<th>Directory/Filename</th>
<th>Contents</th>
<th>Additional Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>RELNOTES.TXT</td>
<td>Contains the latest release notes for this version of the toolkit.</td>
<td></td>
</tr>
<tr>
<td>VERSION.TXT</td>
<td>Contains the toolkit’s version label.</td>
<td></td>
</tr>
<tr>
<td>com.factset.rt_X_Y_Z.jar</td>
<td>A JAR archive containing the DataFeed Java API classes.</td>
<td>X, Y and Z refer to version numbering, explained below.</td>
</tr>
<tr>
<td>SwingQuote.jar, etc</td>
<td>JAR archives of pre-built samples</td>
<td>Sample programs that are pre-built for testing</td>
</tr>
<tr>
<td>etc/</td>
<td>Definition and configuration files</td>
<td>Example: rt_fields.xml, fdsrt.cfg, Level2Markets.xml</td>
</tr>
<tr>
<td>doc/</td>
<td>Documentation</td>
<td>This document along with the Javadoc reference documentation.</td>
</tr>
<tr>
<td>sample/</td>
<td>Sample applications</td>
<td></td>
</tr>
</tbody>
</table>

2.2 Running Applications

The `com.factset.rt_X_Y_Z.jar` file must be included in the Java Virtual Machine (VM) classpath to execute applications built using the API. It is the responsibility of the application developer to ensure this library is available on all run time systems. Additionally, port 443 must be open for authentication purposes.

2.2.1 Windows Systems

In order to connect to the local FactSet workstation and use it as a data source, the current minimum supported or a more recent version of the FactSet workstation must be installed and running on the user’s machine.

2.3 Versioning

The FactSet DataFeed Java API has a standard 3-digit version (x.y.z) label. The first number is the major release number (x), followed by the minor release number (y). The last number, (z), is the revision number.

Changes in the minor release number (y) and the revision number (z) will guarantee binary compatibility with existing applications (i.e., recompilation is not necessary). A change to the major release number (x) may require source code changes for older applications. The severity of the change depends on the API release notes, and the manner in which the application makes use of the API.

For example, if the current API version is 1.0.0 and the new API is 1.0.1, applications may take advantage of the new features/fixes simply by installing the library on the run time systems. A version change to 2.0.0 may require source code changes depending on the type of changes and the application. A complete list of changes for a particular release will always be in the release notes located in the toolkit archive.
Chapter 3 Programming with the API

3.1 Program Setup and Initialization

A complete example of program setup:

```java
public class SimpleExample {
    private RT_Consumer consumer;

    private final MessageHandler msgHandler = new MessageHandler() {
        private String msgType;
        private String bid;
        private String ask;

        public void onMessage(Subscription sub, RT_Message msg) {
            if (msg.isError()) {
                System.err.println("Error: " + msg.getErrorDescription());
                sub.cancel();
                return;
            }

            if (msg.exists(FIDS.MSG_TYPE))
                msgType = msg.getField(FIDS.MSG_TYPE);
            if (msg.exists(FIDS.BID_1))
                bid = msg.getField(FIDS.BID_1);
            if (msg.exists(FIDS.ASK_1))
                ask = msg.getField(FIDS.ASK_1);

            System.out.println("Update: " + msgType + " Bid: " + bid + " Ask: " + ask);

            if (msg.isClosed())
                sub.cancel();
        }
    }

    private void run() {
        String connection = "user:pass@api.df.factset.com";
        consumer = new RT_Consumer(connection);

        try {
            consumer.connect();
        } catch (UnknownHostException e) {
            System.err.println("Error resolving host from: " + connection);
            return;
        } catch (LoginException e) {
            System.err.println("Incorrect username or password: " + connection);
            return;
        } catch (IOException e) {
            System.err.println("Error connecting to server: " + e.getMessage());
            return;
        }

        RT_Request req = new RT_Request("FDS1", "FDS-USA");
    }
}
```
Subscription sub = consumer.request(req, msgHandler);
System.out.println("Made a request for " + req + " sub: " + sub);

while (true) {
    consumer.dispatch(-1);
}

public static void main(String[] args) {
    SimpleExample example = new SimpleExample();
    example.run();
}

3.2 Connecting to a Data Source

An application should connect to a data source during initialization. There are two options when picking a data source to connect to:

- FactSet Data Server
- FactSet workstation.

A connection to a FactSet Data Server occurs over the Internet or through a WAN via TCP/IP. A connection to the local FactSet workstation occurs on the user’s local machine. When connecting to a Data Server, applications should construct a new RT_Consumer instance using one of its constructors. To connect to the local FactSet workstation, applications should construct a new RT_Consumer instance by using the newWorkstationInstance() method.\(^3\)

The connect() method is synchronous, and in rare cases this call may block for an extended period of time (currently set to 60 seconds). If applications wish to use a non-blocking connect, the asyncConnect() method should be invoked.

The host for production data is api.df.factset.com for production and api-stage.df.factset.com for beta. If canned data is required for development purposes the host canned-stage.df.factset.com with the FDS_C service should be used.

```
// connect to api.df.factset.com with user="client" and password="secret".
RT_Consumer consumer = new RT_Consumer("client:secret@api.df.factset.com");
consumer.connect(); // connects synchronously
consumer.asyncConnect(); // connects asynchronously

// connects to FactSet Workstation
RT_Consumer consumer = RT_Consumer.newWorkstationInstance("CLIENT-12345","");
consumer.connect(); // connects synchronously
consumer.asyncConnect(); // connects asynchronously
```

A synchronous connect will block until both the TCP connection is established and the application has successfully authenticated with the data source. If a synchronous connect fails, applications must do one of the following:

---
\(^3\) The FactSet workstation needs to be running in order to call the newWorkstationInstance() method. If the workstation is not running, the API will return an error.
1. Retry the connect method at some future time.
2. Connect asynchronously.
3. Exit the application.

An asynchronous connect will return immediately. If an asynchronous connect throws an exception, a connection will never be established. In this case, the application should log the error and exit. This is a rare condition that will only happen if an operating system resource, such as a thread, was not able to be created.

Upon returning from a successful asynchronous connect, the connection and authentication will be processed by an API thread. If registered, a ControlHandler instance will be invoked after a successful or unsuccessful connect. If the connection fails, the connection is retried periodically.

---

Applications must successfully invoke either `connect()` or `asyncConnect()` before dispatching any messages. However, applications are allowed to make requests before a connection is established. These requests are queued internally within the API until a successful connection is established.

---

### 3.2 Authentication

Currently both basic authentication and One Time Password are valid authentication methods, however basic authentication will eventually be deprecated, so new users are encouraged to support OTP.

#### 3.2.1 Basic Authentication

In order to connect to a FactSet Data Server using basic authentication, the application must set the host name (or IP address), the port number, the username, and the password. These items should be passed into the `RT_Consumer` constructor.

```java
// connect to the host "api-stage.factset.com"
// on the default port of "6681".
// username here is client and password is "aaa"
RT_Consumer consumer = new RT_Consumer("client:aaa@api-stage.factset.com");
```

Alternatively, you can pass in an array of strings with multiple connection strings into the constructor. When multiple connection strings are specified, the API will attempt to connect using each connection string, until a successful connection is made. If the connection is subsequently lost, the API will continue trying to connect using each connection string.

```java
String[] array = { "client:aaa@api-stage.factset.com", 
                   "client:aaa@api-stage2.factset.com", 
                   "client@10.2.4.5:4063" };

RT_Consumer consumer = new RT_Consumer(array);
```

---

4 If the connection is terminated (i.e., via a TERMINATE control message), connection attempts will no longer be retried. This is typical when the user credentials are invalid.

FactSet leverages the HMAC-Based One-Time Password Algorithm described in RFC 4226 (http://www.ietf.org/rfc/rfc4226.txt) and session tokens to ensure all requests to the API are made by authenticated users.

5
3.2.1.2 OTP : Retrieving a One Time Password

The authentication protocol for Exchange DataFeed uses HMAC-Based One Time Passwords. At the initial setup, the key administrator⁶ will need to follow the below steps to generate the key and counter required to authenticate 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 to select PROD, 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”
9. Alternatively take note of the values and use directly in set_connection_info.

3.2.1.3 OTP : Authenticating

Here are some examples of how to authenticate via using the OTP credentials supplied by the key administrator.

Use the counter file, no matter what:

```csharp
// connect to the host "api-stage.df.factset.com"
// on the default port of 6681
// the username here is "client"
// the identifier is "AAAA"
// don't specify the key or counter (we're using the file)
// the counter file is located under C:\MyPath
RT_Consumer consumer = new RT_Consumer( "api-stage.df.factset.com", "client", "AAAA", null, null, "C:\MyPath\AAAA.dat", false );
```

Use the counter file if it exists, otherwise use the provided key/counter and create the counter file:

```csharp
// connect to the host "api-stage.df.factset.com"
// on the default port of 6681
// the username here is "client"
// the identifier is "AAAA"
// the key is "5c7602...."
// the counter is "12345...."
// the counter file is located under C:\MyPath
RT_Consumer consumer = new RT_Consumer( "api-stage.df.factset.com", "client", "AAAA", "5c7602....", "12345....", false );
```

⁶ The key administrator needs to have access to be able to generate the key, contact your FactSet representative to get the required access enabled.
Use the provided key/counter regardless of whether the file exists. Overwrite the file with the new values:

```java
RT_Consumer consumer = new RT_Consumer("api-stage.df.factset.com", "client", "AAAA", "5c7602....", "12345...", "C:\MyPath\AAA.AAA.dat", true);
```
3.2.2 Synchronous Connect Sequence Diagram

Figure 3: Synchronous Connect Sequence Diagram
3.2.3 Synchronous Connect Example

```java
public class SyncConnectExample {
    private static RT_Consumer consumer;
    private static RT_FieldMap map;

    public static void main(String[] args) {
        try {
            map = new RT_FieldMap("./etc/rt_fields.xml");
        } catch (SAXException e) {
            System.err.println("Error parsing field map file");
            return;
        } catch (IOException e) {
            System.err.println("Error opening field map file");
            return;
        }

        String connection = "client:secret@ api-stage.df.factset.com:6681";
        RT_Consumer consumer = new RT_Consumer(connection);

        // if using OTP
        RT_Consumer consumer = new RT_Consumer(
            "api-stage.df.factset.com",
            "client",
            "AAAA",
            null,
            null,
            "C:\Path\To\Counterfile",
            false);

        try {
            consumer.connect();
        } catch (UnknownHostException e) {
            System.err.println("Error resolving host from: " + connection);
            return;
        } catch (LoginException e) {
            System.err.println("Incorrect username or password: " + connection);
            return;
        } catch (IOException e) {
            System.err.println("Error connecting to server: " + e.getMessage());
            return;
        }

        // ...
        // make requests
        // invoke RT_Consumer.dispatch to dispatch available messages
        // ...
    }
}
```

The example code above demonstrates how to connect to the Data Server synchronously.

The first step in many programs would be to load a field map file. This file is located in the etc directory of the toolkit. The RT_FieldMap class allows the application to translate field names to identifiers and vice versa. Although this step is not required, it helps with debugging and troubleshooting. For more information on the RT_FieldMap class, see the included javadocs.
The second step is to construct the RT_Consumer using the connection information (i.e., host = api-stage.df.factset.com, port = 6681, user = client, password = secret). After successfully constructing the RT_Consumer, the application invokes the connect method to attach to the Data Server. For additional details on setting the connection information, see section 3.2.1 Authentication or the included javadocs.

3.2.4 Asynchronous Connect Sequence Diagram

Figure 4: Asynchronous Connect Sequence Diagram
3.2.5 Asynchronous Connect Example

```java
public class AsyncConnectExample {
    private static RT_Consumer consumer;

    private static final ControlHandler cHandler = new ControlHandler() {
        public void onControl(boolean isConnected, RT_Message controlMsg) {
            if (controlMsg.isError()) {
                System.err.println("Error connecting: " +
                    controlMsg.getErrorDescription());
            }
            if (isConnected)
                System.out.println("Connection established");
        }
    };

    public static void main(String[] args) {
        String connection = "api-stage.df.factset.com:6681";
        consumer = new RT_Consumer(connection);
        // if using OTP
        RT_Consumer consumer = new RT_Consumer("api-stage.df.factset.com",
            "client",
            "AAAA",
            null,
            null,
            "C:\\Path\\To\\Counterfile",
            false);

        consumer.setControlHandler(cHandler);

        try {
            consumer.asyncConnect();
        } catch (IOException e) {
            System.err.println("Error initiating connection: " + e.getMessage());
            e.printStackTrace();
        }

        // ...
        // make requests
        // invoke RT_Consumer.dispatch to dispatch available messages
        // ...
    }
}
```

The example code above demonstrates how to connect to the FactSet Data Server asynchronously. The code is similar to the previous example. However, in this example, the `asyncConnect()` method is invoked. When a connection is finally established or if errors are encountered while trying to connect, the `onControl()` method of `cHandler` will be invoked.
3.3 Requests and Cancels

3.3.1 Opening the Stream

Requests are made using the `RT_Consumer.request` method. Although requests are typically made after connection establishment, the application can make requests at any time. If the API is disconnected from the server, or a particular service is not available, requests will be queued internally by the API and then returned Subscription instance will have the `isQueued` flag set to true. The request method is defined as follows:

```java
public Subscription request(RT_Request req, MessageHandler handler);
```

The `RT_Request` class is the first parameter required by the request method. This class can be constructed using a service and a key. A service is a string that identifies a data source and the symbol is the key for that particular data source. In addition, the `RT_Request` object allows applications to explicitly set the snapshot flag to true for a static request and false for a dynamic request. A dynamic request will open a virtual stream with the Data Server for that particular data element. A static request will also open a virtual stream, but the first message on that stream will indicate a closure of that stream. This type of request is typically called a snapshot request.

The second parameter of the request method is an instance of the `MessageHandler` interface. Classes wishing to implement this interface must implement using the following method:

```java
public void onMessage(Subscription sub, RT_Message msg);
```

This method acts as a callback and will receive the `Subscription` instance associated with the stream, along with the message.

On each call to `RT_Consumer.request` an `RT_Consumer.Subscription` instance will be returned. **Even if the request has been internally queued, a valid Subscription instance will be returned.** This Subscription instance is unique per stream and can act as a resource identifier for the stream that has just been opened. A Subscription instance is returned for both static and dynamic requests.

3.3.2 Closing the Stream

The messages for a stream are passed to the `MessageHandler.onMessage` method along with the `Subscription` instance. Subscriptions should be treated as system resources and eventually the stream should be closed. This resource can be freed in two ways: either by invoking the `RT_Consumer.disconnect` method or alternatively by invoking the `Subscription.cancel` method. The stream will continue to be open until one of these two methods is called. This is true even for snapshot requests. As mentioned before, snapshot data is treated as a request for a single message, and that message should have the close (i.e., end of stream) indicator set. This indicator tells the application that the stream is closed on the server-side. It is the responsibility of the application to make sure the Subscription is cancelled after receiving the snapshot message. A call to `Subscription.cancel` will close the stream on the client-side.

❖ **Failing to cancel a Subscription can cause applications to consume more memory and respond slower. Applications should treat Subscriptions as if they treat open files or open sockets.**
3.3.3 Dynamic Request

```java
public class DynamicRequestExample {
    private static RT_Consumer consumer;

    private static final MessageHandler handler = new MessageHandler() {
        public void onMessage(Subscription sub, RT_Message msg) {
            System.out.println("Message: " + msg);
            if (msg.isError())
                System.err.println("Error: " + msg.getErrorDescription());
            // if the server closed the stream, close our side as well
            if (msg.isClosed())
                sub.cancel();
        }
    };

    public static void main(String[] args) {
        // establish connection (see previous code)
        RT_Request req = new RT_Request("FDS1", "FDS-USA");
        Subscription sub = consumer.request(req, handler);

        // invoke RT_Consumer.dispatch to dispatch available messages
    }
}
```

The example code above shows a request for the symbol “FDS-USA” to the “FDS1” service. Since the snapshot flag parameter for constructing the RT_Request instance is not specified, the request is for a dynamic subscription by default. The RT_Request instance is passed to the request method, along with the MessageHandler instance. The Subscription instance associated with the stream is returned by the request method and assigned to the local variable sub.

The implemented onMessage method simply prints the message. However, it does check to see if the stream was closed, and if so, it closes the client-side stream by canceling the Subscription. The server may close the stream at any time. In addition, error messages (e.g., RT_E_NOT_FOUND, RT_E_TIMEDOUT) will cause the stream to set the close/end-of-stream indicator. The MessageHandler example handles both of these conditions.
### 3.3.4 Static Request

```java
public class StaticRequestExample {

    private static RT_Consumer consumer;
    private static final MessageHandler handler = new MessageHandler() {
        public void onMessage(Subscription sub, RT_Message msg) {
            System.out.println("Message: "+msg);
            // no reason to check if the stream is closed, since a static
            // request will only provide a single message
            sub.cancel();
        }
    };
    public static void main(String[] args) {
        // establish connection (see previous code)
        // construct a static RT_Request instance
        // the "true" parameter sets the snapshot flag
        RT_Request req = new RT_Request("FDS1", "FDS-USA", true);
        Subscription sub = consumer.request(req, handler);
        // invoke RT_Consumer.dispatch to dispatch available messages
    }
}
```

The request for snapshot data is similar to the one for dynamic data, except that the snapshot parameter is set to true. In fact, the MessageHandler from the previous example could have been used in this example. Since static requests will close the stream on the first message, the previous onMessage method would have cancelled the Subscription. However, the above example states the application’s intentions more clearly when the call to Subscription.cancel is explicit, rather than based on a predicate, like in the example in the previous section.

### 3.3.5 Bulk Subscriptions

Where possible it is recommended that applications request subscriptions in bulk.

> Note that the bulk interface is only supported for the FDS1 service

The significant changes between the bulk interface and the regular request interface are:

- The key for the RT_Request should be a comma-separated list of symbols, rather than a single symbol.
- The Subscription object that request() returns represents the entire bulk subscription. Cancelling that subscription will cancel every symbol within the bulk subscription.

Beyond these differences, the bulk interface acts identically to the standard interface. Whenever a message is received for any of the symbols in the bulk subscription, the callback specified to request() will be called. The tag that is used in the callback is the tag of the individual symbol, not of the bulk subscription as a whole. See the SwingBulk sample utility for an example of how the bulk interface is used.

---

7 Options are not supported within bulk requests at this point
3.3.6 Canceling Requests

Applications can cancel a Subscription at any time, even before receiving the first message. Once the application returns from the cancel method, the MessageHandler.onMessage method for the request identified by that Subscription will never be called\(^8\). This guarantee simplifies programming by allowing the application to clean up resources used by the MessageHandler instance immediately after the call to Subscription.cancel().

3.4 Processing Events

3.4.1 Normal Dispatching

```java
public class DispatchExample {
    private static RT_Consumer consumer;
    public static void main(String[] args) {
        // establish connection (see previous code)
        // make requests (see previous code)

        // invoke RT_Consumer.dispatch to dispatch available messages
        while (true) {
            consumer.dispatch(-1);
        }
    }
}
```

In order to dispatch messages to the MessageHandler instance(s), control must be handed back to the API. This is accomplished by calling the RT_Consumer.dispatch method. This method will flush all of the currently queued messages and returns.

The above code calls dispatch(-1) in an infinite loop. Passing -1 as a parameter will inform dispatch to wait indefinitely for messages to dispatch. However, the function will still return if at least a single MessageHandler was invoked. This is why the example code calls dispatch in a loop.

Messages from the FactSet Data Server are treated as events. These events are delivered via the MessageHandler instance that was registered at the time of request. If the application only wants to block for a defined time period, they can pass in a time value in milliseconds. A time value of zero will flush all messages and return immediately. For more information on the dispatch(), method, refer to the included javadocs.

3.4.2 Handling Exceptions

The dispatch method can throw an IllegalStateException for several reasons. The dispatch methods can be invoked from any thread at any time, but an exception will be thrown if dispatch is called concurrently (i.e., either from another thread or from within the MessageHandler.onMessage, NotifyHandler.onNotify, or ControlHandler.onControl methods). The only other reason dispatch can throw an IllegalStateException is if the application never successfully invoked the connect or asyncConnect method in the first place.

\(^8\) If the MessageHandler is registered by more than one stream, only the stream identified by the Subscription instance is affected by a call to cancel. Streams are allowed to share MessageHandler instances, and canceling a stream will only prevent the MessageHandler from being used in the context of the cancelled subscription.
The dispatch methods return a boolean indicating the TCP connection status of the RT_Consumer. This can indicate a loss of connection. As long as the connect method has been successfully invoked, the API will retry the connection every so often. Applications are encouraged to maintain the event loop during this time period. However, they can invoke disconnect if they choose to do so.

3.5 Processing the Messages

3.5.1 FID Value Pairs

The API makes heavy 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, commonly known as a FID or a field identifier.

The value is typically some opaque data delivered in string form. 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.

3.5.2 Field Identifiers

The current field identifiers are available from two sources. The first is the FIDS class. This class defines human-readable static final short data types for the current list of known field identifiers. This is the usual method of identifying a field by name in actual Java code.

The second file is rt_fields.xml also included in the toolkit. This file can be loaded by the RT_FieldMap class and allows applications to translate human readable names to field identifiers at run time (as opposed to compile time using the FIDS class).

3.5.3 Messages

All requests open a subscription, which is a virtual tunnel of messages.

Message data is contained in the RT_Message class. An RT_Message is a container of fields (i.e., fids and values). The fields can be extracted using the methods of the RT_Message class, for more information, see the RT_Message section in the included javadocs. A message also contains certain properties such as a key, permissions, and other status flags.
3.5.4 Processing a Message Example

```java
public class ProcessMessageExample {
    private static RT_Consumer consumer;

    private static final MessageHandler msgHandler = new MessageHandler() {
        private String msgType;
        private String bid;
        private String ask;

        public void onMessage(Subscription sub, RT_Message msg) {
            if (msg.isError()) {
                System.err.println("Error: " + msg.getErrorDescription());
                sub.cancel();
                return;
            }

            if (msg.exists(FIDS.MSG_TYPE))
                msgType = msg.getField(FIDS.MSG_TYPE);
            if (msg.exists(FIDS.BID_1))
                bid = msg.getField(FIDS.BID_1);
            if (msg.exists(FIDS.ASK_1))
                ask = msg.getField(FIDS.ASK_1);

            System.out.println("Update: " + msgType + " Bid: " + bid + " Ask: " + ask);

            if (msg.isClosed())
                sub.cancel();
        }
    }

    public static void main(String[] args) {
        // establish connection (see previous code)

        RT_Request req = new RT_Request("FDS1", "FDS-USA");
        Subscription sub = consumer.request(req, msgHandler);

        while (true)
            consumer.dispatch(-1);
    }
}
```

The example code above shows one way to process a message within the `MessageHandler.onMessage` callback method. The method simply prints the message type along with the cached bid and asks. In addition, it checks to see if the stream was closed, and if so, it closes the client-side stream by canceling the subscription.

The server may close the stream at any time. In addition, messages with the error flag set will cause the stream to set the close/end-of-stream indicator. The example `MessageHandler` handles both of these conditions.
3.6 Threading

3.6.1 Thread-safety

All classes in the DataFeed Java API are thread-safe. Applications are free to call the methods of these classes using multiple threads.

3.6.2 Threading Issues Using a Callback Interface Driven API

A potential for deadlock exists when all following conditions are true:

- The application uses more than one thread (not counting the API threads).
- More than one application thread uses the same RT_Consumer instance.
- The callback method needs to lock a shared object that is used by another application thread, which also shares the same RT_Consumer instance.
  
  OR

  The callback method needs to wait on a thread that uses the same RT_Consumer instance.

If all of the above statements hold true, applications need to be aware of two potential deadlock scenarios as described in the next section.

3.6.3 Avoiding Deadlock

The RT_Consumer class is thread-safe, but must also call back into the user application during the dispatch method. Furthermore, the API is fully reentrant, and a callback method is permitted to call additional API methods. To prevent a race condition in which one thread cancels a subscription right before a callback method is about to execute, the API must hold a lock during the method invocation to ensure that a cancel for that stream is held until the callback method is finished. Due to this scenario, it is possible for the application to deadlock. Care must be taken when in the handler method. **Callback methods should not wait on other threads that need access to the API (i.e., invoke the wait method to wait for a thread that invokes any methods of the same RT_Consumer instance).**

Furthermore, callback methods that need to lock an object, should be careful of the locking order. For example, the following sequence of two application threads may cause a deadlock:

<table>
<thead>
<tr>
<th>Callback Thread(1)</th>
<th>Application Thread(2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Locks Object XXX</td>
<td>Locks Object XXX</td>
</tr>
<tr>
<td>// some processing</td>
<td>// some processing</td>
</tr>
<tr>
<td>// ...</td>
<td>// calls an RT_Consumer:: method</td>
</tr>
<tr>
<td>Unlocks Object XXX</td>
<td>Unlocks Object XXX</td>
</tr>
<tr>
<td>return</td>
<td>// ...</td>
</tr>
</tbody>
</table>

---

9 The API guarantees that upon returning from cancel(), no additional callback methods on the closed stream are possible, see section 3.3.2 Closing the Stream for more information.
Since the application thread(#2) is locking object XXX and then calling an RT_Consumer method, its locking order is defined as:

1. Lock object XXX
2. Lock the RT_Consumer object (implicit from the method call)
3. Unlock RT_Consumer object (implicit from the return of the method call)
4. Unlock object XXX

However, the callback thread(#1)’s locking order is the following:

1. Lock the RT_Consumer object (implicit from dispatch)
2. Lock object XXX
3. Unlock object XXX
4. Unlock the RT_Consumer object (implicit from dispatch)

Since two threads lock the same objects in different orders, deadlock is possible. To eliminate the deadlock, the developer can synchronize the threaded application section which calls into RT_Consumer using the java synchronized keyword on the instance of RT_Consumer being used.

<table>
<thead>
<tr>
<th>Callback Thread(1)</th>
<th>Application Thread(2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Locks Object XXX</td>
<td>synchronized(consumer){</td>
</tr>
<tr>
<td>// some processing</td>
<td>\hspace{1cm} Locks Object XXX</td>
</tr>
<tr>
<td>// ….</td>
<td>// calls an RT_Consumer:: method</td>
</tr>
<tr>
<td>Unlocks Object XXX</td>
<td>\hspace{1cm} Unlocks Object XXX</td>
</tr>
<tr>
<td>return</td>
<td>\hspace{1cm} }</td>
</tr>
</tbody>
</table>

Chapter 4 Permission Service

FactSet has a permission system used to entitle its terminal users for real time or delayed exchange data for display-only-use, this system has been extended to enforce permissioning via third party integrators. By providing user permission maps, login status and an IP address check, the third party system can enforce FactSet’s terminal permissions in their own system. This is called the Workstation Entitled API permission setup.

❖ Clients who subscribe to the Enterprise DataFeed and manage their own permissions and exchange redistribution agreements do not need to use the Permission Service.

4.1 Requirements

To use the Workstation Entitled API permission scheme, every user needs to have a unique FactSet Serial Number, either linked to a FactSet Workstation or to a FactSet Launch account. FactSet maintains the individual user’s exchange permissions on their serial number. Exchange access through the third party terminal will be granted based on Serial Number access.

Every subscription to streaming data provided by FactSet contains a permission code. The third party system must match the permission code with the permission code contained in the user map of the user requesting the data. If there is a match, data can be passed on to that user. If there is no match, then the user is not entitled for the data and an error message should be displayed.

In order for FactSet to comply with its exchange commitments, the third party must follow the instructions of the FactSet permission system. FactSet will audit any third party implementation to ensure its permissions are being enforced correctly.

The Permission service encapsulates all of the FactSet permission logic in to a simple ALLOW/DENY notification. Third parties must subscribe, listen, and follow all the permission statuses relayed by the permission service. The permission service generates data for each individual Factset user. So, the third party system must request and continue to listen to the permission service using the FactSet USERNAME-SERIAL combination. For example, XYZCOMPANY-12345 is passed to the permission service. Any changes will be sent via the service. In addition, the third party must provide an IP address or list of IP addresses. These two sets of information will be all FactSet needs to make a judgment on whether a user has access to data or is denied.

As a response, the permission service will provide two sets of information. One is the login status, represented by a 1 or 0. When the status is 1, the third party is allowed to send FactSet exchange data to its third party terminals. When the status is 0, no FactSet exchange data may be sent. The second is the permission map which is only available if the user is logged on. The third party must match the streaming data to the user’s permission set to accurately permission the individual user.

❖ As mentioned above, the permission service is designed to provide streaming updates on the status of individual users. It is not necessary nor desirable to rapidly make new requests to this service in an attempt to discover changes because they will be streamed to the subscriber automatically.

Only one subscription is allowed for a particular user, if the user attempts so authenticate on a second machine a “Duplicate subscription Error” will be sent to the first request. This is by design to signal that a user is already logged on from a different terminal. The correct behavior will be to allow the new login request and invalidate the original connection.

❖ FactSet provide utilities for firms that may want to check on the status of an individual user using the permission service. Because of the duplication subscription behavior, this will shut down the individual in favor of the utility, which may not be the desired result.
4.1.1 Authenticating with a FactSet Workstation

The user can only receive data while being logged into the FactSet workstation on the same machine as where the third party terminal software is running, this will be confirmed by an IP address check and logon status check.

If the third party terminal tried to run with the user not being logged in to FactSet, or logged in on a different machine, the third party terminal would fail the login test and would not receive any data.

4.1.2 Authenticating with FactSet Launch

FactSet Launch is a web portal where multiple FactSet services can be accessed through a single sign-on, the user’s unique and permanent factset.net ID is used to login. The factset.net ID is linked to a FactSet Username and Serial Number with individual access to datasets and applications.

The Launch utility Activate my Terminal is available in the tools menu in FactSet Launch. The utility is collecting the local IP address from the machine where it is run to be used by the permission service. Activate my terminal is recommended to use through Chrome.

The user needs to authenticate through FactSet Launch on the same machine as the third party terminal is being used. This will be confirmed by an IP address check. Once authenticated access with be granted for 12 hours. After 12 hours the user needs to renew its access from launch.factset.com.

If the third party terminal is run without the user being authenticated through Launch in the last 12 hours, or authenticated on a different machine, the third party terminal fails the login test and will not receive any streaming data.

4.2 Workflow

An overview of the technology and workflow for this service is described below

- **Step 1**: FactSet has a centralized system that manages all its end users’ permissions, login status and Launch/Workstation IP address. A user logs into the FactSet terminal/Launch and the Permission Server is notified.
• **Step 2:** This system informs the DataFeed of the users’ current state of permissions, login status and Launch/Workstation IP address.

• **Step 3:** The DataFeed server will check the list of IP addresses sent to the API and if the Launch/Workstation IP is in the list, the DataFeed Server will also ensure the user is currently logged on, and pass information that the requirements were met. The Third Party system then has all the information it needs to permission their terminals.

• **Step 4:** If the user is not authenticated/logged into FactSet or the IP addresses do not match, then the third party system is not allowed to send exchange data to the third party terminal. If the user is authenticated/logged in and the IP addresses do match, then a second layer of permissioning takes place. The exchange data needs to be matched up with the permission map of the user by the third party server. If the end user has the proper permissions, then the exchange data can be displayed in the third party terminal, which runs on the same machine as the FactSet terminal.

If the end user’s permission map does not contain the needed entitlement, a message will be sent saying the user is not entitled and no exchange data will be sent to the terminal.

**Example:**

1. User requests FDS-USA
2. An FDS-USA trade message containing permission code 12345 is returned by the DataFeed server
3. Third party confirms that user has 12345 in their permission map
4. Third party allows FDS-USA to be seen by user

**Continuation of Example:**

5. The user’s permission to 12345 is removed
6. User’s list of permission codes is updated
7. Third party server denies user access to FDS-USA

The login information, permission maps and IP address checks are dynamic. If there is any change, the third party server should pick this up and the new logic should be applied.

**4.3 Audit Process**

FactSet has a number of tests designed to ensure the third party integrator is properly enforcing FactSet’s permissions. These are contained in a separate document available upon request from FactSet. FactSet will need to perform an audit at the third party’s office to ensure compliance.

**4.4 Service and Data Model**
The permission service name is FDS_PERM. The request keys to this service should be of the form USERNAME-serial number (e.g., FDS-12345).

The permission request will return a response with two fields. FID 9221 (USER_LOGIN_STATUS), will return a 1 or 0. 1 signifies the client is logged in/currently/authenticated and the IP addresses match. 0 signifies that the user is logged off/not authenticated or the IP addresses do not match.

For FID 9222 (USER_PERMISSIONS), there will be a comma delimited list of permission codes for the user. When a field exceeds 255 characters, the same 9222 fid is repeated with the new continuation of the permission code list. This continues until the list is complete.

The IP addresses need to be comma separated and sent through the constructor:

```
RT_Request(String service, String key, boolean snapshot, String authToken, String options)
```

```
RT_Request req = new RT_Request("FDS_PERM", "USER-SERIAL", false, ",", "1.2.3.4, 192.168.0.1");
```

### 4.4.1 Complete Permission Service Example

```java
default package src;
import java.io.IOException;
import java.net.*;
import java.util.*;
import javax.security.auth.login.LoginException;
import com.factset.rt.*; // import the API classes
import com.factset.rt.RT_Consumer.Subscription;

public class FDSPermExample {
    private RT_Consumer consumer;

    private final MessageHandler msgHandler = new MessageHandler() {
        private String loginStatus;
        private String permissions;

        // Execute on response from server
        public void onMessage(Subscription sub, RT_Message msg) {
            if (msg.isError()) {
                System.err.println("Error: " + msg.getErrorDescription());
                sub.cancel();
                return;
            }

            if (msg.isClosed())
                sub.cancel();
```
if(!msg.exists(FIDS.USER_LOGIN_STATUS)) {
    System.err.println("Error: No USER_LOGIN_STATUS found in permissions message");
    return;
}

//Fetch login status in response
loginStatus = msg.getField(FIDS.USER_LOGIN_STATUS);

Integer loginStatusValue = Integer.parseInt(loginStatus);
if(loginStatusValue == 1) {
    System.out.println("User is logged in, try to get permissions");
    permissions = msg.getPermissions();
    System.out.println("Permissions codes: "+permissions);
} else {
    System.out.println("User is logged off, or IP Address does not match");
    sub.cancel();
}

//Fetch all network IP Addresses for the local machine
private String getUserIPAddresses() {
    String result = "";
    Enumeration<NetworkInterface> interfaces;
    try {
        interfaces = NetworkInterface.getNetworkInterfaces();
        while(interfaces.hasMoreElements())
        {
            NetworkInterface n=interfaces.nextElement();
            Enumeration<InetAddress> iNetAddrs = n.getInetAddresses();
            while(iNetAddrs.hasMoreElements())
            {
                InetAddress i = iNetAddrs.nextElement();
                if(result.isEmpty())
                    result = i.getHostAddress();
                else
                    result += "," + i.getHostAddress();
            }
        }
    return result;
    } catch (SocketException e1) {
        e1.printStackTrace();
        System.err.println("Failed to fetch user IP addresses");
        return "";
    }
}

public void run() {

    String connection = "user:password@api.df.factset.com";
    String targetUser = "user-id";

    consumer = new RT_Consumer(connection);
try {
    consumer.connect();
} catch (UnknownHostException e) {
    System.err.println("Error resolving host from: " + connection);
    return;
} catch (LoginException e) {
    System.err.println("Incorrect username or password: " + connection);
    return;
} catch (IOException e) {
    System.err.println("Error connecting to server: " + e.getMessage());
    return;
}

String ipAddress = getUserIPAddresses();

if (ipAddress.isEmpty())
    return;

// Build a request for FDS_PERM service with local IP addresses
RT_Request request = new RT_Request("FDS_PERM", targetUser, false, "", ipAddress);
Subscription sub = consumer.request(request, msgHandler);
System.out.println("Made a request for " + request + " sub: " + sub);

while(true) {
    consumer.dispatch(-1);
}

public static void main(String[] args) {
    FDSPermExample example = new FDSPermExample();
    example.run();
}
Chapter 5 Level 2 Data

FactSet provides market depth in the Exchange DataFeed for Enterprise Streaming DataFeed users. In this document market depth is referred to as Level 2 data.

5.1 Requirements

Level 2 functionality in the Java toolkit requires version 1.2.0 or higher of the Exchange DataFeed Java Toolkit. Any application that is updated to use version 1.2.0 of the Java toolkit will need to recompile. Any applications that are going to use level 2 functionality will require a code change and to recompile.

5.2 Setting up Level 2 Data

There are two ways to receive Level 2 data: Raw Data and Sorted Data. For either type, the ticker requested must be appended with ":L2", this will subscribe to the Level 2 feed for the given ticker and is the only requirement to subscribe to raw Level 2 data. To subscribe to NASDAQ TotalView data the ticker should be appended with :TV.

Additional FactSet product permissions are needed to consume these data sets. The raw data request will provide all the bids and asks for an individual security. The updates will be sent in the order they are received by FactSet. To access prerecorded canned data for development efforts use the service FDS_C, the available ticker for canned level 2 data is:

**SIAC:**  
FDS  
IBM  
DIS  
JNJ  
WMT

**NASDAQ:**  
CSCO  
AAPL  
INTC  
MSFT  
AMZN

Sorted Data is identical to raw data, with the exception that every valid Level 2 message contains a field indicating the message’s sorted position. To enable Sorted Data, the Level 2 feature must be enabled after the connection is made before the Level 2 request is sent

```java
Consumer.enable(new RT_Level2Feature());
```

5.3 Level 2 Fields

<table>
<thead>
<tr>
<th>Field Id</th>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>150</td>
<td>BID_INDEX_1</td>
<td>Integer</td>
<td>Sorted Data only: The message’s position in a sorted list of bids.</td>
</tr>
<tr>
<td>250</td>
<td>ASK_INDEX_1</td>
<td>Integer</td>
<td>Sorted Data only: The message’s position in a sorted list of asks.</td>
</tr>
<tr>
<td>520</td>
<td>ORDER_CODE</td>
<td>String</td>
<td>Order Code</td>
</tr>
</tbody>
</table>
5.4 Processing Level 2 Data

There are a few specific rules for Level 2 messages that need to be followed to maintain an accurate record.

- If a message has the MSG_TYPE “D”, it represents a delete, and the corresponding entry, by ORDER_CODE, is no longer valid. For the Sorted Data functionality, it will no longer be considered when sorting the list and should be removed accordingly.
- For Sorted Data, each valid message will come with BID_INDEX_1 and/or ASK_INDEX_1 populated. These indicate the message’s position in the sorted list of bids and ask respectively. To handle these messages properly, the previous corresponding entry in the list, by ORDER_CODE, should be removed, and this message should be inserted at the position specified in the INDEX field.

5.4.1 Processing a Message Example

The example code below shows the basic structure for handling sorted Level 2 messages. The method checks for message type and bid or ask data, on which any processing can be done as needed.

In addition, it checks to see if the stream was closed and, if so, closes the client-side stream by canceling the subscription. The server may close the stream at any time; messages with the error flag set will cause the stream to set the close/end-of-stream indicator. The example Message Handler handles both of these conditions.

```java
public class ProcessL2MessageExample {
    private static RT_Consumer consumer;

    private static final MessageHandler msgHandler = new MessageHandler() {
        private String msgType;
        private String bidIndex;
        private String askIndex;

        public void onMessage(Subscription sub, RT_Message msg) {
            if (msg.isError()) {
                System.err.println("Error: " + msg.getErrorDescription());
                sub.cancel();
                return;
            }

            if (msg.exists(FIDS.MSG_TYPE))
                msgType = msg.getField(FIDS.MSG_TYPE);
            if (msg.exists(FIDS.BID_INDEX_1))
                bidIndex = msg.getField(FIDS.BID_INDEX_1);
            if (msg.exists(FIDS.ASK_INDEX_1))
                askIndex = msg.getField(FIDS.ASK_INDEX_1);
        }
    }
}
```

10 Only used in Nasdaq Level 2 feed, this is the only level 2 exchange that does not clear the book at the end of the day, quotes are just closed.
if (msgType.equals("D")) { // handle delete message }
else {
    if (!bidIndex.isEmpty()) { // handle bid data }
    if (!askIndex.isEmpty()) { // handle ask data }
}

if (msg.isClosed())
    sub.cancel();
}

public static void main(String[] args) {
    // establish connection (see previous code)
    consumer.enable(new RT_Level2Feature()); // before requesting
    RT_Request req = new RT_Request("FDS1", "FDS-USA:L2");
    Subscription sub = consumer.request(req, msgHandler);

    while (true)
        consumer.dispatch(-1);
}

See the SwingL2 sample utility for a more complete example, including logic for maintaining sorted bid and ask lists.

NOTE: The maximum number of simultaneous level 2 symbols per connection is limited to 100 symbols.
Chapter 6 Options Greeks Calculation

FactSet provides additional fields that return Greeks values and Implied Volatilities for Streaming DataFeed users.

6.1 Requirements

The Options Greeks Calculations require Version 4.0.1 of the Exchange DataFeed Java Toolkit. Any applications that want to use this new functionality will require a code change and to recompile.

6.2 New Implied Volatility and Greek Fields

<table>
<thead>
<tr>
<th>Field Id</th>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2613</td>
<td>ANALYTIC_PRICE_RULE</td>
<td>Integer</td>
<td>This is a flag to tell which price is being used in the analytic calculations. A value of 1 means that Mid price is used. A value of 2 means that during market hours a Mid price will be used and after market hours the settlement price will be used.</td>
</tr>
<tr>
<td>2614</td>
<td>EXPIRATION_DAYS_TO</td>
<td>Integer</td>
<td>The number of business days until the option expires</td>
</tr>
<tr>
<td>2620</td>
<td>DELTA</td>
<td>Decimal</td>
<td>The rate of change of option value with respect to changes in the underlying asset's price.</td>
</tr>
<tr>
<td>2621</td>
<td>GAMMA</td>
<td>Decimal</td>
<td>The rate of change in the delta with respect to the changes in the underlying asset's price.</td>
</tr>
<tr>
<td>2622</td>
<td>VEGA</td>
<td>Decimal</td>
<td>The sensitivity of the value of the option to the volatility of the underlying asset</td>
</tr>
<tr>
<td>2623</td>
<td>THETA</td>
<td>Decimal</td>
<td>The sensitivity of the value of the option to the passage of time</td>
</tr>
<tr>
<td>2624</td>
<td>RHO</td>
<td>Decimal</td>
<td>The sensitivity of the value of the option to the risk free interest rate</td>
</tr>
<tr>
<td>2630</td>
<td>IMP_VOL</td>
<td>Decimal</td>
<td>The volatility of the price of the underlying security that is implied by the market price of the option based on an option pricing model</td>
</tr>
<tr>
<td>2631</td>
<td>IMP_VOL_ASK</td>
<td>Decimal</td>
<td>The volatility of the price of the underlying security that is implied by the market ask price of the option based on an option pricing model</td>
</tr>
<tr>
<td>2632</td>
<td>IMP_VOL_BID</td>
<td>Decimal</td>
<td>The volatility of the price of the underlying security that is implied by the market bid price of the option based on an option pricing model</td>
</tr>
<tr>
<td>2633</td>
<td>IMP_VOL_CALC_RATE</td>
<td>Decimal</td>
<td>The calculated value of the interest rate using the option pricing model</td>
</tr>
<tr>
<td>2634</td>
<td>THEO_VALUE</td>
<td>Decimal</td>
<td>The calculated value of the option using the option pricing model</td>
</tr>
</tbody>
</table>

Please note that all fields except ANALYTIC_PRICE_RULE and EXPIRATION_DAYS_TO will be blank in the initial snapshot message. The values will begin streaming shortly after. The values will be recalculated based on any changes in the underlying asset or the option. The values will be sent at a maximum of once every 10 seconds.

---

For detailed information on how these fields are calculated please see FactSet Online Assistant Page 14933
6.2.1 Sample Data

response key: IBM#A1814C195000-USA, tag: 1, msg:
T:1 K:IBM#A1814C195000-USA E:0 Flags:AGB
NumFids = 11 Size = 151
  MSG_TYPE[1] Val = U Size=1
  DELTA[2620] Val = 0.553972 Size=8
  GAMMA[2621] Val = 0.007738 Size=8
  VEGA[2622] Val = 0.901767 Size=8
  THETA[2623] Val = -0.020611 Size=9
  RHO[2624] Val = 1.200264 Size=8
  IMPL_VOL[2630] Val = 22.392416 Size=9
  IMPL_VOL_ASK[2631] Val = 22.697414 Size=9
  IMPL_VOL_BID[2632] Val = 22.087504 Size=9
  IMPL_VOL_CALC_RATE[2633] Val = 22.392416 Size=9
  THEO_VALUE[2634] Val = 20.424996 Size=9

6.3 Risk Free Interest Rates

FactSet uses Sovereign Debt Benchmarks for Risk Free Interest Rates. The country will be determined based on the currency of the option and the period of time will be determined based on the expiration date of the option.

6.4 Setting up Greek Calculations

There are two steps required to turn on the Greek calculations

1. Import the library for Greeks.

   import com.factset.rt.RT_OptGreeksFeature;

2. Enable Greek Calculation after the connection is made before connecting the consumer.

   consumer.enable(new RT_OptGreeksFeature());

6.4.1 Processing a Message Example

```java
public class ProcessGreeksExample {
    private static RT_Consumer consumer;

    private static final MessageHandler msgHandler = new MessageHandler() {
        private String msgType;
        private String delta;
        private String gamma;
        private String vega;
        private String theta;
        private String rho;

        @Override
        public void onMessage(Subscription sub, RT_Message msg) {
            if (msg.isErrorResponse()) {
                System.err.println("Error: " + msg.getErrorDescription());
            }
        }
    }
```
sub.cancel();
return;
}

if (msg.exists(FIDS.MSG_TYPE)) msgType = msg.getField(FIDS.MSG_TYPE);
if (msg.exists(FIDS.DELTA)) delta = msg.getField(FIDS.DELTA);
if (msg.exists(FIDS.GAMMA)) gamma = msg.getField(FIDS.GAMMA);
if (msg.exists(FIDS.VEGA)) vega = msg.getField(FIDS.VEGA);
if (msg.exists(FIDS.THETA)) theta = msg.getField(FIDS.THETA);
if (msg.exists(FIDS.RHO)) rho = msg.getField(FIDS.RHO);

String output = "Received:" + " MsgType = " + msgType + ", Delta = " + delta + ", Gamma = " + gamma + ", Vega = " + vega + ", Theta = " + theta + ", Rho = " + rho;

System.out.println(output);
if (msg.isClosed()) {
    sub.cancel();
    System.out.println("Stream Closed by Server...");
}
}

public static void main(String[] args) {
    /**
     * establish connection (see previous code on how to do this)
     */
    consumer.enable(new RT_OptGreeksFeature()); // before requesting
    RT_Request req = new RT_Request("FDS1", "VOD.GB#CWX3D-IFEU");
    Subscription sub = consumer.request(req, msgHandler);
    while (true) {
        consumer.dispatch(-1);
    }
}
Appendix A: Document Revision History

The following are revisions made since Version 2.0.0

<table>
<thead>
<tr>
<th>Revisions</th>
<th>Sections</th>
</tr>
</thead>
<tbody>
<tr>
<td>Formatting corrections</td>
<td>All</td>
</tr>
<tr>
<td>Grammar</td>
<td>All</td>
</tr>
<tr>
<td>OTP Information Clarified</td>
<td>3.2.1</td>
</tr>
<tr>
<td>Contents of etc/ folder updated</td>
<td>2.1</td>
</tr>
<tr>
<td>Removed mention of Java version 6.0</td>
<td>1.4.1</td>
</tr>
</tbody>
</table>

The following are revisions made since the Version 1.1 revision C:

<table>
<thead>
<tr>
<th>Revisions</th>
<th>Sections</th>
</tr>
</thead>
<tbody>
<tr>
<td>Updated section for workstation connections</td>
<td>1.1</td>
</tr>
<tr>
<td>Added Port 443 requirement</td>
<td>2.2</td>
</tr>
<tr>
<td>Added Windows Systems Section</td>
<td>2.2.1</td>
</tr>
<tr>
<td>Added language to specify the FactSet workstation needs to be running when the newWorkstationInstance() method is called</td>
<td>3.2</td>
</tr>
<tr>
<td>Updated section for workstation connections</td>
<td>3.2</td>
</tr>
<tr>
<td>Added Chapter on Permissioning Service</td>
<td>4.1-4.4</td>
</tr>
<tr>
<td>Added Required Ports</td>
<td>1.4.2</td>
</tr>
<tr>
<td>Added Complete FDS_PERM sample</td>
<td>4.4.1</td>
</tr>
<tr>
<td>Added OTP information</td>
<td>3.2.1</td>
</tr>
<tr>
<td>Added Level 2 Chapter</td>
<td>5</td>
</tr>
<tr>
<td>Added Greeks Chapter</td>
<td>6</td>
</tr>
</tbody>
</table>