FACTSET 〉 SEE THE ADVANTAGE

FactSet Exchange DataFeed API

.NET Programmer's Manual

Version 5.2F

Last updated: 4th February 2022

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	Description	Changes made
02/04/2022	5.2F	Section 3.7	Added details on Option and Future Chains
05/28/2021	5.2E	Section 4.3.3	Exposed Serialize and Deserialize functionality for the RTMessage class.
03/29/2021	5.2E	Section 4.6.1	Removed requirement to set user/serial when making a call to ConnectToWorkstation() or ConnectToWorkstationAsync()
03/29/2021	5.2E	Section 1.4.13	Bulk Subscription is not supported by the toolkit
03/29/2021	5.2E	Section 1.4.14	Adding Subscription Limits
08/14/2020	5.2D	Section 1.4.2	Added details on Sydney VIP
08/14/2020	5.2D	Section 1.4.12	Added details on Custom Conflation
08/14/2020	5.2D		Updated Global support procedures
	5.2	Section 4.6.2	Updated documentation for logging from the unmanaged portion of the toolkit.
		Section 3.3	Removed section on Bulk Subscriptions
		Section 5	Added complete FDS_PERM example
	4.0	Section 3.5.5	Added documentation for unchanged fields

Effective Date	Version Number	Description	Changes made
		Section Appendix B	Added documentation for aggregation
	3.0	Section 4.1.1, Appendix A	Added new error code
	1.0	Section 7	Added Chapter on Level 2 functionality
		Section 3.3	Added section on Bulk Subscriptions
		Section 3	Added details on OTP

Table of Contents

Document Organization and Audience

This document is intended for application programmers that are familiar with the .NET Framework and the C# programming language. 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 5 - Describes the permissioning service.
- Chapter 6 - Describes Options Greek calculations
- Chapter 7 - Describes Level 2 data
- Chapter 8 - Lists utilities available
- Appendix - Extends the class reference by providing additional details.

Chapter 1 Introduction

1.1 The FactSet DataFeed.NET API

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

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 that is hosted by FactSet. Connections to a FactSet Data Server occur over the Internet or through a WAN via TCP/IP. Applications must be given a username, password, and the address information (i.e., IP and port number) for the FactSet Data Server.

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 must be given 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.

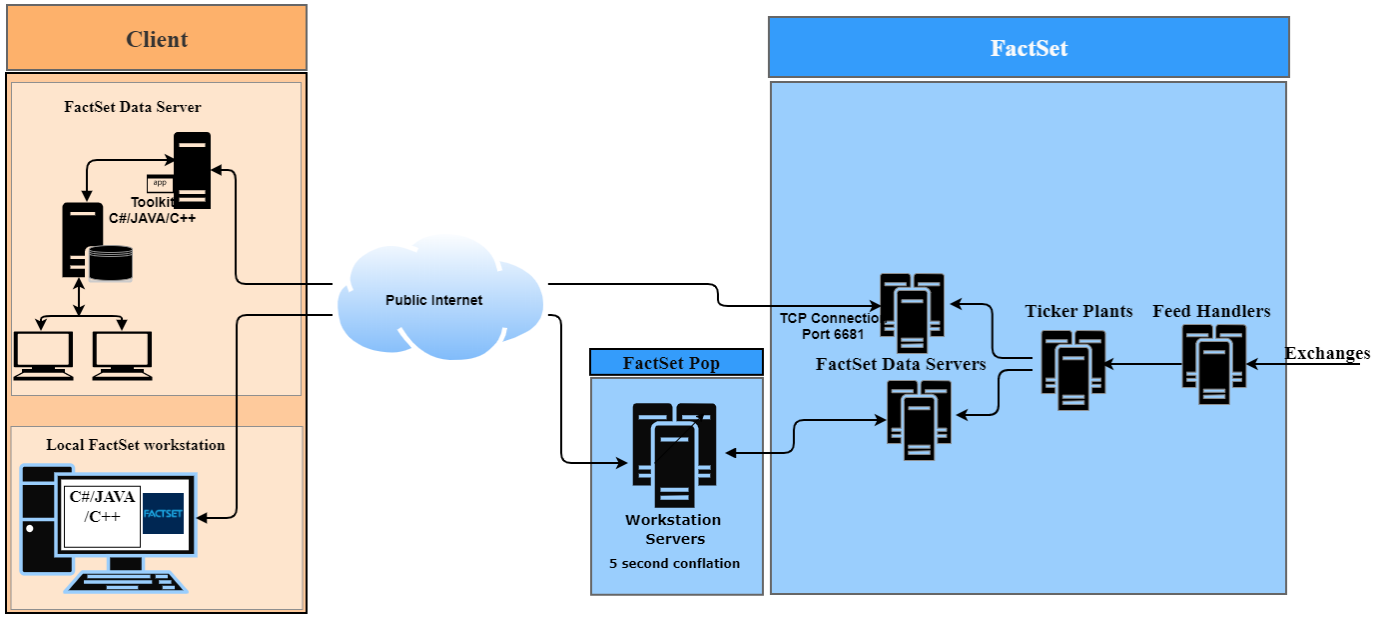


Figure 1 Two API clients connected to the two different FactSet data sources

1.2 Terminology

The following terminology is used throughout this documentation:

Terminology	Meaning
API	Application Programming Interface - a set of defined interfaces that applications use to extract information from the FactSet Data Server.
SDK	Software Development Kit - a collection of libraries, include files, documentation, and sample codes that make up this toolkit.
XML	Extensible Markup Language - a defined standard for exchanging information. The information contains markup tags used to describe the data values.
TCP/IP	Transport Control Protocol over Internet Protocol - the protocol that this API uses to communicate to the FactSet Data Server.
FactSet Data Server	A server which provides permissioned access to FactSet data.
FDS	Multiple meanings. FDS is the ticker symbol for FactSet Research Systems Inc. It is also the namespace in which this API resides. Finally, it may stand for the FactSet Data Server. The meaning is defined by its context.
Service	A data source or supplier identified by a string name.
FDS1	FactSet Data Service Version 1 - a well-known data service. For a complete description of the data fields, types, and possible values see the *FactSet Data Service Manual*.
FDS_FUND	FactSet's Fundamental Data Service. Used for End of Day data.
FDS_C	FactSet's Canned Data Service. Recorded data is replayed, used for testing.
FDS_PERM	FactSet's Permission Service. Used by third party integrators to enforce end-users Exchange permissions using the Workstation Entitled API setup.
Consumer	Any application that uses this API.
Stream	A virtual tunnel of messages for a given request.
FID	Field Identifier - an integer identifier that describes the encoding and business meaning of a field value.
Opaque Data	Data without a defined interpretation, which is simply a pointer and size to the data.
Field/Value Pairs	A self-describing message format used in API messages. Each pair contains a FID and some opaque data. The FID defines the type and meaning of the data.

1.3 High Level Overview

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

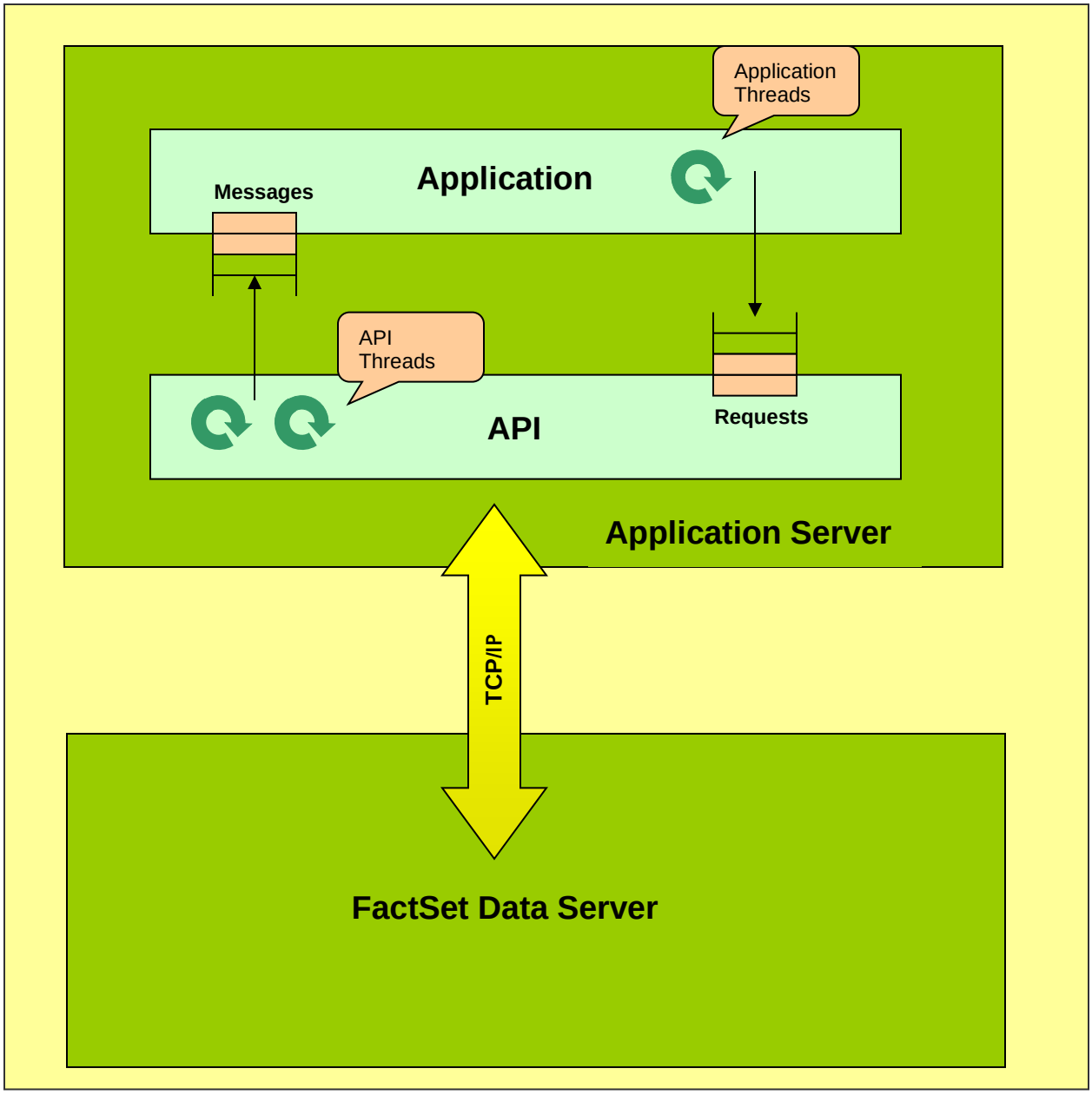


Figure 2: High Level Overview

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

- **Connect to the Data Server**: This will initiate the TCP connection and start an internal communication thread within the API, and in the case of an asynchronous connection, it will create a background event dispatch thread.
- **Request Data**: Requests will be posted on a queue to be sent out via the communication thread.
- **Receive Messages via Event Handler**: Incoming messages will be posted to a message queue by the communication thread. The Console application will call an API method to synchronously dispatch any available events. All event handlers will be executed in the context of an application thread running the dispatch loop. The UI application will call an API asynchronous method to start an additional background thread to dispatch events to the UI thread context.
- **Disconnect from the Data Server**: The application may disconnect from the Data Server at any time. This will destroy the communication and dispatch, if asynchronous, threads as well.

Information on API Threads

The API will create at least one thread per RTConsumer object. This thread serves as a communication thread and is responsible for all of the TCP/IP communication with the data server. The thread is created when the application connects to the FactSet Data Server, and destroyed when the application calls Disconnect().

If you are calling Dispatch() synchronously in a loop the message event handlers will be called in context of that thread. This is more typical for server-side applications. See for instance the **ConsoleSync** sample.

However, if you are developing a UI WPF, WinForms, or ASP.NET applications or if you are just using a corresponding Application.Run() function in a Console application an additional background thread will be created to dispatch events to UI thread. See for instance the WinForms and WPF samples.

In addition, the API starts a single global maintenance thread. This thread will be created only if the application uses the Logging interfaces within the API. Once this thread is created, it can only be destroyed on program termination.

1.4 API Core Functionality and Benefits

The API provides the following services to applications:

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

1.4.1 Support for Multiple Development Platforms

Supported .NET Frameworks are 3.5, 4.0, 4.5.

To run an application in .NET Framework 4.0 and upper please update your *app.config* file:

```
<startup useLegacyV2RuntimeActivationPolicy="true">
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5" />
</startup>
```

1.4.2 TCP/IP Communications

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

Hosts:

- Production:
  - api.df.factset.com
  - api-syd.df.factset.com [2]
- Staging:
  - api-stage.df.factset.com
- Canned:
  - canned-stage.df.factset.com

Note: Host api-syd.df.factset.com, connects directly to FactSet's Sydney ticker plant for Australian and New Zealand data. This host should be used by clients in Australia or New Zealand requesting data for the local exchanges (global data is also available on this host allowing AUS/NZ clients to only use this host for all their data needs).

Firewall access for outbound-initiated TCP 6681 to the following destinations will have to be opened.

- 192.234.235.0 (255.255.255.0)
- 64.209.89.0 (255.255.255.0)
- 164.55.240.0 (255.255.240.0) [3]

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 notice4. 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

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 Clients should contact their FactSet account team for access to the Sydney host
3 This IP address is only required for the Sydney host (api-syd.df.factset.com).
4 As of 29-Jul-2017 support for security protocol TLSv1.0 is disabled and requests using this TLS version will fail

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

1.4.5 Request Consistency

The API provides a consistent interface for opening and closing streams. All requests will receive an RTConsumer.Subscription object. This applies to both static (i.e., snapshot) and dynamic requests. In addition, messages on any given stream will be associated by the stream identifier. To close a stream, the application needs to pass the stream subscription back to the API.

1.4.6 Subscription Management

The API allows applications to request duplicate data items. Each item will create its own stream. 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 Snapshot message

The initial message on the stream will contain all the fields for a message. Subsequent messages may only contain the fields that have changed. This behavior may require an application to keep state of all the fields for a given a stream.

1.4.8 Logging and Configuration Management

To aid developers with troubleshooting and debugging, the API supports logging of error and informational messages to standard error (cerr) in Debug and Release modes and debug/assert messages to Diagnostics.Debug output in Debug mode only. Applications can provide an actual log file opened and messages be directed to that file.
There is a sample how to redirect Debug messages to standard output:

```
Diagnostics.Debug.Listeners.Add(new TextWriterTraceListener(Console.Out));
```

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. The API includes functionality to assist applications in querying configuration files and system registries.

It is recommended that when logging, the FileStream.Flush() mechanism is not used. In .NET 3.5, this function does not guarantee flushing to disk and when used directly with the FactSet DataFeed .NET API’s logging may cause unwanted performance or other hindrances. In .NET 4.0, this method was improved but still has an issue. In .NET 4.5 this may be acceptable for use.

In addition to possible performance degradation, the following exception may be thrown when using this function call:
System.IO.IOException: The OS handle's position is not what FileStream expected. Do not use a handle simultaneously in one FileStream and in Win32 code or another FileStream. This may cause data loss.

```
at System.IO.FileStream.VerifyOSHandlePosition()
at System.IO.FileStream.WriteCore(Byte[] buffer, Int32 offset, Int32 count)
at System.IO.FileStream.FlushInternalBuffer()
at System.IO.FileStream.Flush(Boolean flushToDisk)
at System.IO.FileStream.Flush()
at System.IO.StreamWriter.Flush(Boolean flushStream, Boolean flushEncoder)
at System.IO.StreamWriter.Flush()
at System.Diagnostics.TextWriterTraceListener.Flush()
at System.Diagnostics.TraceInternal.WriteLine(String message)
at System.Diagnostics.Trace.WriteLineIf(Boolean condition, String message)
at TestClientSync.TestClientSyncMain.ProcessMessage(RTConsumer cons, RTMessage& msg,
RTSubscription sub) in c:\Users\USERNAME \Datafeed.NET\APPLICATION_NAME\FILE_NAME.cs:line
800
```

1.4.9 Threading Support

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

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

1.4.10 Caching

The initial message (i.e. the snapshot message) on the stream will contain all available fields for a symbol. Subsequent messages may only contain the fields that have changed. This behavior may require an application to keep state of all the fields for a given a stream. For the benefit of the application, the API will perform this caching. A cached data record is associated with every stream and is available during callback processing. An application may use this record in any matter it sees fit. Caching is not enabled by default. It needs to be manually specified that the application should perform caching.

1.4.11 Filtering and Bucketing

FactSet applies quote filtering to some exchanges to control the bandwidth, the filtering level is exchange dependent, typically between 0.25s and 1s, all exchanges are not filtered. Note that only quotes are filtered, all trades are sent.
For a local connection to the FactSet workstation the data is bucketed to 5 second updates.

1.4.12 Custom Conflation [5]

Subscribers of market data stream may not need to get every single update from the feed. In those cases, they can request timed updates (aka bucketed) by adding an "interval" argument to the options string that can be passed into a subscription request.

The format required is "interval=<ms>" and the valid values for the interval are as follows:

- 500
- 1000
- 2000
- 5000

Note: Block trades are not bucketed

1.4.13 Subscription Requests

The Toolkit does not support bulk subscription requests, only individual subscription requests are allowed. Users can make individual subscription requests by creating a loop calling RT_Request for each symbol.

1.4.14 Subscription Limits

Please contact your FactSet Account team if you would like to increase your subscription limit

5 Clients should contact their FactSet Account Team for the feature to be enabled.

Chapter 2 Building Applications

2.1 Toolkit Organization

The toolkit is extracted from a simple Windows MSI file. The installation folder is specified during installation using a standard dialog (defaults to C:\Users*xxx*\AppData\Local\FactSet\DataFeed). The directory hierarchy is outlined in the following table.

Directory/Filename	Contents	Additional Notes
RELNOTES.TXT	Contains the latest release notes for this version of the toolkit.	
VERSION.TXT	Contains the toolkit's version label and build number.	
LICENSES.TXT	Contains license agreements.	
bin	The actual debug and release .NET assemblies, IntelliSense xml-file, binary utilities and samples, if any	
etc	Definition files	Example: rt_fields.xml
sample	Sample applications	

2.1.1 Supported Compilers, Operating Systems, and Architectures

The toolkit currently supports the following platforms:

- .NET Framework 3.5

There are Release and Debug mode assemblies for 32 and 64-bit architectures.
Additional platforms may be added in the future. Please contact FactSet Consulting Services if a platform does not appear in the list above.

2.1.2 Assembly Locations

Assemblies installation directory tree:

```
bin\
        x86\
                Release\
                Debug\
        x64\
                Release\
                Debug\
```

FactSet.Datafeed.dll is a set of mixed-mode assemblies. There is an assembly for each combination of supported Configuration-Platform pairs. For more information regarding mixed-mode assembly see: Mixed (Native and Managed) Assemblies.

2.2 Compiling Applications

The first step in compiling an application would be to extract, if necessary, and install the toolkit to a directory.
To simplify the following steps, assume that the root of this archive is located in *{FDS_ROOT}*. In the default case this would be set to “C:\Program Files\FactSet\DataFeed.NET”.

Add to your project settings reference to the corresponding FactSet.Datafeed.dll assembly according to the **Platform Settings** you are building you application for (x86 or x64).

Add to you source code the line:

```
using FactSet.Datafeed;
```

2.3 Running Applications

The .NET assembly needs to be installed on any system that will execute applications built using the API. It is the responsibility of the application developer to ensure these libraries are available on all run-time systems. In order to connect to the local FactSet workstation and use it as a data source, version 2011.1 of the workstation must be installed on the user’s machine.

2.4 Versioning

The FactSet DataFeed.NET’s dynamically linked library file has a standard 4-digit version (x.y.z.b) label. The first number is the major release number (x), followed by the minor release number (y), then the revision number (z), and then the build number (b). The .NET assembly version attribute follows the standard .NET version schema: MajorVersion.MinorVersion.BuildNumber.Revision.

Changes in only the revision number (z) will guarantee binary compatibility with existing applications (i.e., recompilation is NOT necessary). Changes in the minor release number (y) ensure source code compatibility, but applications built previously MUST be recompiled to use the newer library. 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 2.0.1 and the new API is 2.0.2, applications may take advantage of the new features/fixes simply by installing the library on the run-time systems. If the new version label is 2.1.1, the application must recompile, but source code changes are not necessary. A version changes to 3.0.1 may require source code changes (depends 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

3.1.1 Standard Conventions

The API follows standard .NET Framework conventions.

3.1.2 Namespace

An application should use the following statement:

```
using FactSet.Datafeed;
```

3.1.3 A Complete Example

```
using System;
using FactSet.Datafeed;

namespace DFsample
{
  class Program
  {
    static FieldMap fldMap;

    static void Main(string[] args)
    {
      fldMap = FieldMap.Create(@"rt_fields.xml");

      using (RTConsumer consumer = new RTConsumer())
      {
        consumer.ConnInfo = @"username:password@api.df.factset.com";

        consumer.Connect();
        Console.WriteLine("Connected to: {0}", consumer.ConnInfo);

        RTRequest req = new RTRequest("FDS1", "F-USA", true);
        RTConsumer.RTSubscription subs = null;
        subs = consumer.MakeRequest(req,
          (RTConsumer.RTSubscription sub, RTMessage msg) =>
          {
            Console.WriteLine("Snapshot message:\n{0}", msg);
            consumer.Cancel(subs);
          }
          );

        while (subs.IsSubscribed)
        {
          consumer.Dispatch(-1);
        }

        consumer.Disconnect();
        Console.WriteLine("Disconnected from: {0}", consumer.ConnInfo);
      }
    }
  }
}
```

3.2 Connecting to a Data Source

An application connects to a data source during initialization. There are two options when picking a data source to connect to: a FactSet Data Server and the local FactSet workstation. A connection to a FactSet Data Server occurs over the Internet or 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 set the connection information, and then call the Connect() or ConnectAsync() function. There are two ways to authenticate the user. One is basic authentication and the other is OTP authentication. OTP authentication is done by passing the credentials to SetConnInfo function. To connect to the local FactSet workstation, only a call to the ConnectToWorkstation() or ConnectToWorkstationAsync() function is required[6].

The Connect() and ConnectToWorkstation () are synchronous, and in rare cases a call may block for an extended period of time (currently set to 60 seconds). If applications wish to use a non-blocking connect call ConnectAsync() and ConnectToWorkstationAsync() connect functions.

If the RTConsumer object has already established a connection, the repeating calls to connect will **disconnect, drop** all active requests, and try to connect again using current connection info settings.

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 DataFeed server synchronously
consumer.ConnInfo = RTConsumer.MakeConnInfo("username", "password",
  "api.df.factset.com");
consumer.Connect();        //            blocks here

//  connect to DataFeed server asynchronously
consumer.ConnInfo = RTConsumer.MakeConnInfo("username", "password",
  "api.df.factset.com");
consumer.ConnectCompleted += new EventHandler<ConnectCompletedEventArgs>(
  object sender, ConnectCompletedEventArgs e) =>
  {
    var consumer = sender as RTConsumer;
    if (consumer.IsConnected)
    {
      Console.WriteLine("Connected to: {0}, status: {2}, msg:{1}\nServices:",
        consumer.ConnectedToHost, e.CntrlMsg, e.CtrlType);
      foreach (var svc in consumer.Services)
        Console.WriteLine("\t{0}", svc);
    }
    else
    {
      Console.WriteLine("Disconnected from: {0}, status: {2}, msg: {1}\n",
        consumer, e.CntrlMsg, e.CtrlType);
      return;
    }
  }
  );

// connect to Datafeed server with OTP Authentication
consumer.SetConnInfo("hostPort", "userSerial", "deviceId", "key", "counter", "path", "forceInput");

consumer.ConnectAsync();  //  doesn't block
```

6 The ConnectToWorkstation/ConnectToWorkstationAsync () and the FactSet workstation need to be run under the same user or FactSet will give an error that there are more than one instance of Marquee running.

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

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

An asynchronous connect operation will return immediately. If an asynchronous connect operation returns an error, a connection will never get established. In this case, the application should log the error and exit. This is a rare condition which will only happen if an operating system resource could not be created (like a thread).

Upon returning from a successful asynchronous connect operation, the connection and authentication will be processed by an API thread. A consumer.*ConnectCompleted* event will be raised after a successful or unsuccessful connect operation. If the initial connect failed, then the *ConnectCompletedEventArgs.Error* will reference a *DatafeedException* object. The exception is inherited from *System.ApplicationException* and has set the *Data* property with pairs of error_code->function_name as its data.

If the connection fails during dispatch, the connection is retried periodically.

❖ If Connect() or ConnectToWorkstation () returns an error, the connection will never get established. Applications must issue a successful connect before dispatching any messages. This behaviour is true for both asynchronous and synchronous connections. 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.

3.2.1 Connection Strings

Currently both basic authentication and One-Time password[7] is valid authentication methods but users are being migrated to OTP.

One Time Password - SetConnInfo

Consumer.SetConnInfo("apt-stage.factset.com", "client", "AAAB", NULL, NULL, " C:\\Path\\To\\Counterfile", true); - The API will connect to the host “api-stage.factset.com” on the default port of “6681”. It will use a username of “client” and a One Time password generated by the key and counter as per 3.2.1.1 using the Key ID AAAA and the key and counter file C:\\Path\\To\\Counterfile.

Basic Authentication ConnInfo
In order to connect to a FactSet Data Server, the application must set the host name (or IP address), the port number(optional), the username, and the password. These items should be assigned to the RTConsumer.ConnInfo property.

consumer.ConnInfo = “username: password@api-stage.factset.com”; – The API will connect to the host “api-stage.factset.com” on the default port of “6681”. It will use a username of “username” and a password of “password”.
Alternalevly you can use `RTConsumer.MakeConnInfo` helper functions:
Consumer.ConnInfo = RTConsumer.MakeConnInfo (“username”, “password”, “host”, port);

7 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

consumer.ConnInfo = "username@10.2.4.5:4063"; – The API will connect to the host 10.2.4.5 on port 4063 using the username "username". The password is empty in this case.

It is also possible to set multiple connection strings at once by concatenating each string, separated by a pipe ('|'). The following is an example.

consumer.ConnInfo = "username:p@ssw0rd@api-stage.factset.com|username:p@ssw0rd@api-stage2.factset.com"; – 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 connecting using each connection string.

❖ Multiple conneciton strings are typically used when a firm has a WAN connection to FactSet. In a normal setup, there are 4 separate IP addresses given for valid connections. The program can connect using all 4 connections using the pipe delimited list**.**

The ConnInfo assignment will only return an error if the host could not be extracted given the specified URI. It does not check if the host is valid, or if the host can be translated to an actual IP address. It will simply store the connection information for later use. The Connect() method will later use this information to resolve the hostname and port before attempting the connection to the FactSet Data Server.

3.2.1.1 Retrieving the One Time Password

The authentication protocol for Exchange Datafeed is using One Time password. At the initial setup the key administrator[8] 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 the PROD is selected, rather than BETA.
5. Click Get New Key.
6. Create a new file - On the first line, copy and paste the "Key" from the web site (don't include the word "Key:", just the actual string).
7. On the second line, copy and paste the counter value.
8. Save this file as .data. Most likely that will be "AAAA.data" and use this file as input in the setConnInfo function as per below.
9. Alternatively take note of the values and use directly in setConnInfo.

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

3.2.1.2 Connect with OTP

In the below samples three different examples of how to use the key and counter extracted above is used in

```
// Connect to api-stage.df.factset.com with user="client" with key/counter file
// C:\Path\To\Counterfile\AAAA.data contains the key (hex string) on the first
// line and the counter (decimal format) on the second, for user "client" and // device ID "AAAA"
Consumer.setConnInfo("api-stage.df.factset.com", "client", "AAAA", NULL, NULL, "C:\\Path\\To\\Counterfile", false);

// Connect to api-stage.df.factset.com with user="client" with key/counter file, // or given values if no file exists
// If C:\Path\To\Counterfile\AAAA.data contains a key and counter, those will be // used instead of the given key
"5c706e..." and counter "730332..."
// Otherwise the given key/counter will be used and // C:\Path\To\Counterfile\AAAA.data will be created from the
given values to be
// used for subsequent attempts
Consumer.setConnInfo("api-stage.df.factset.com", "client", "AAAA", "5c706e...", "730332...",
"C:\\Path\\To\\Counterfile", false);

// Connect to api-stage.df.factset.com with user="client" with given values
// regardless of existing key/counter file.
// The given key/counter will be used and C:\Path\To\Counterfile\AAAA.data will be // overwritten or created from the
given values to be used for subsequent attempts
Consumer.setConnInfo("api-stage.df.factset.com", "client", "AAAA", "5c706e...", "730332...",
"C:\\Path\\To\\Counterfile", true);
```

3.2.2 Synchronous Connect Sequence Diagram

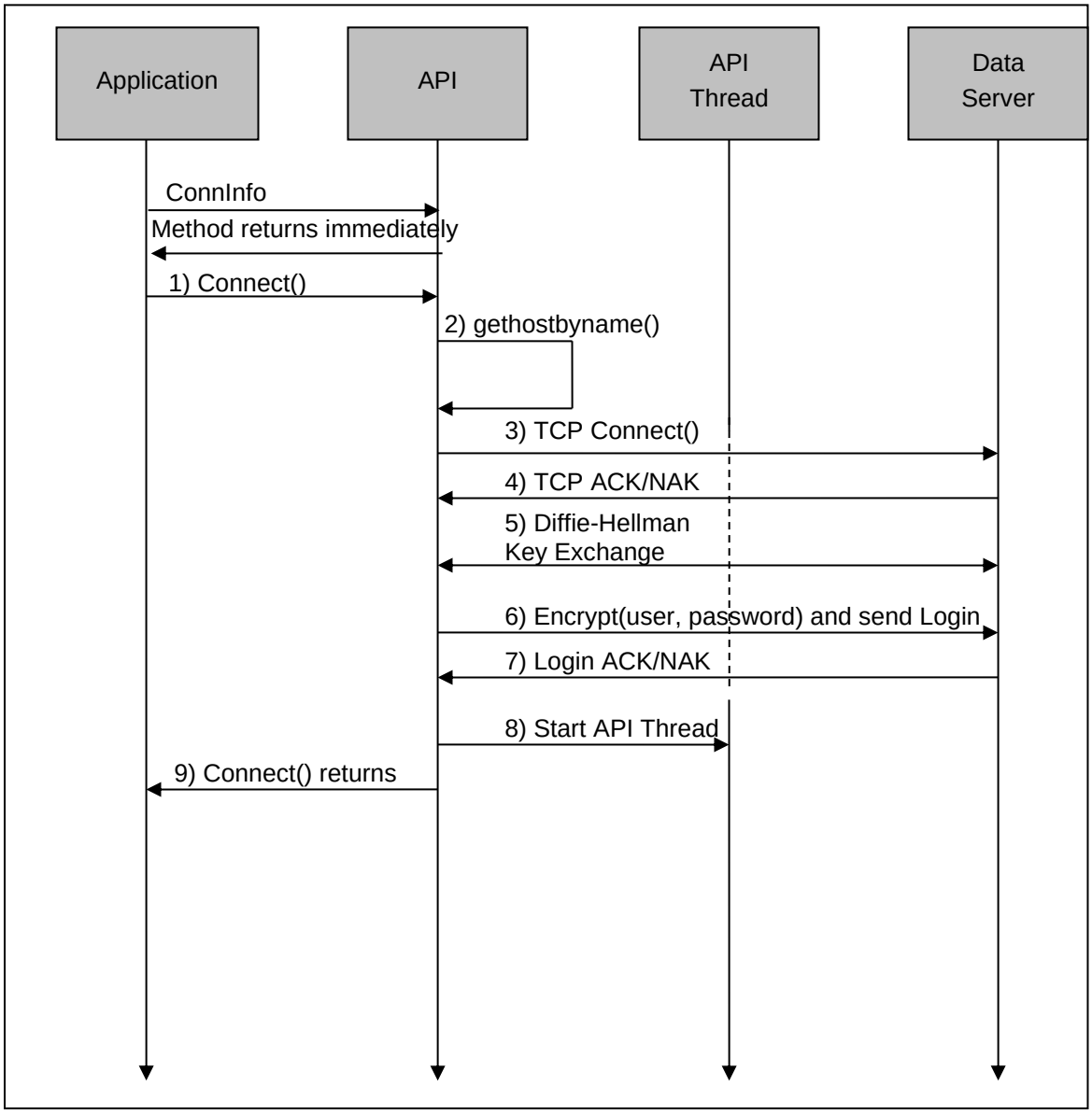


Figure 3: Synchronous Connect Sequence Diagram

3.2.3 Synchronous Connect Example

```
using System;
using FactSet.Datafeed;

namespace DFsample
{
  class Program
  {
    static FieldMap fldMap;

    static void Main(string[] args)
    {
      fldMap = FieldMap.Create(@"rt_fields.xml");

      using (RTConsumer consumer = new RTConsumer())
      {
        try
        {
          consumer.ConnInfo = @"username:password@api-stage.factset.com";

          consumer.Connect();

          // …
          // make requests
          // process the event loop and handle the callbacks
          // …
        }
        catch (Exception ex)
        {
          Console.Error.WriteLine("Exception: {0}", ex);
        }
      }
      …
    }
  }
}
```

The example code above demonstrates how to connect to the Data Server synchronously.[9]

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 code above uses a relative path location (“rt_fields.xml”) based on the application’s working directory. Applications should ensure that this path is correct. The FieldMap class allows the application to translate field names to field ids and vice versa. Although this step is not absolutely necessary, it helps with debugging and troubleshooting. For more information on the FieldMap class, see section 4.5.1FieldMap Class.

The second step is to pass in the connection information (i.e., host = api-stage.df.factset.com, port = 6681, user = ”username”, password = ”password”). After setting the connection information, the application calls Connect() to attach to the Data Server. For additional details on setting the connection information, see section 3.2.1 Connection Strings or section 4.6.1 RTConsumer Class.

9 All of the supporting code outlined in this section is provided in the sample toolkit directory named Example3.

3.2.4 Asynchronous Connect Sequence Diagram

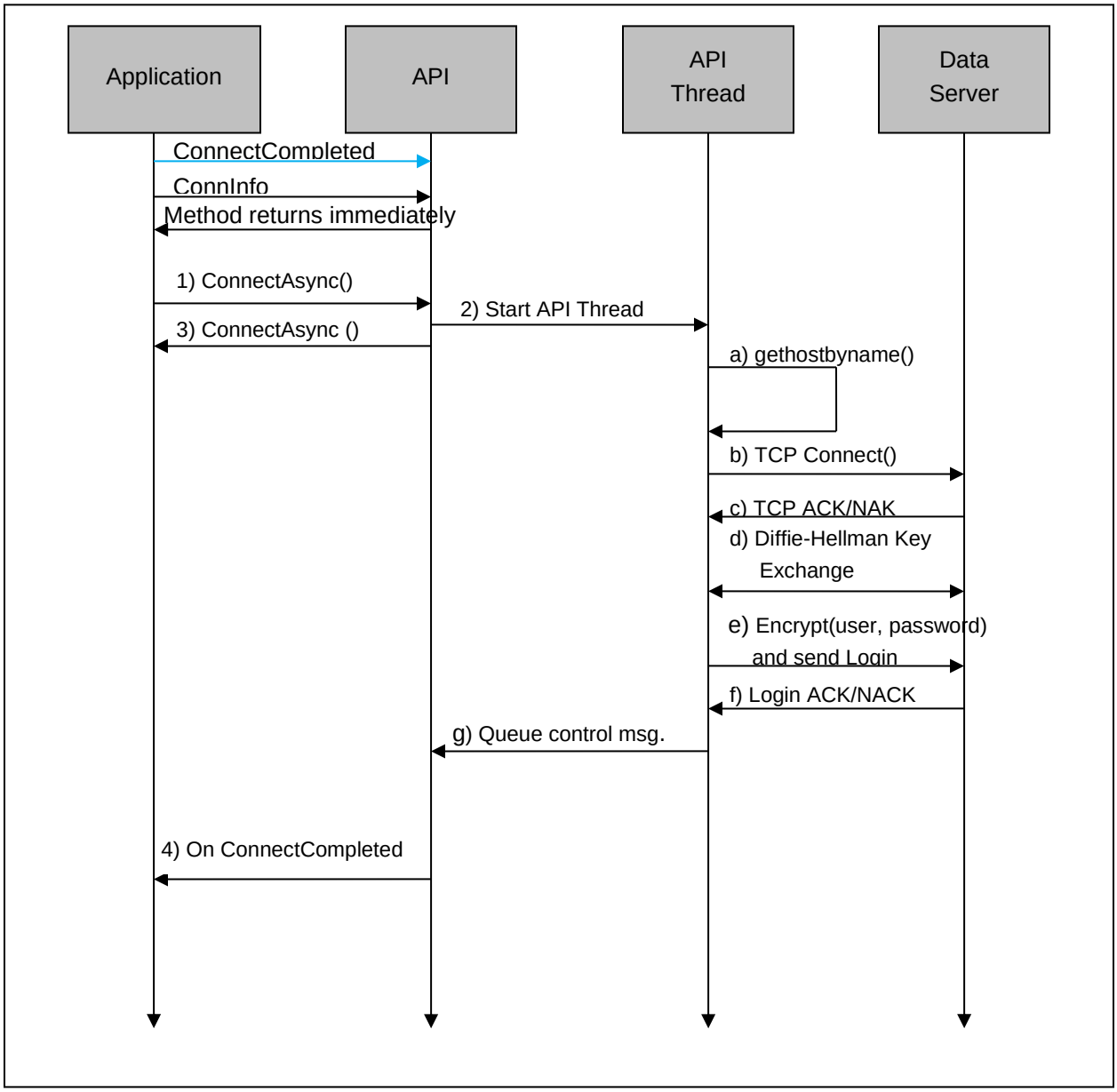


Figure 4: Asynchronous Connect Sequence Diagram

3.2.5 Asynchronous Connect Example

```
using System;

using FactSet.Datafeed;

namespace SampleAppRun
{
  class AppRunMain
  {
    static FieldMap fldMap;
    static RTConsumer consumer = new RTConsumer();

    static Exception connectError = null;

    static void Main(string[] args)
    {
      fldMap = FieldMap.Create("rt_fields.xml");

      consumer.ConnectCompleted +=
        new EventHandler<ConnectCompletedEventArgs>(
          (object sender, ConnectCompletedEventArgs e) =>
          {
            if (consumer.IsConnected)
            {
              Console.WriteLine("Connected to: {0}, status: {2}, msg: {1}",
                consumer.ConnectedToHost, e.CntrlMsg, e.CtrlType);
            }
          }
          else
          {
            Console.WriteLine("Disconnected from: {0}, status: {2}, msg: {1}\n",
              consumer, e.CntrlMsg, e.CtrlType);
            return;
          }
          );

              consumer.ConnInfo = @"username:password@api-stage2.factset.com";
      consumer.ConnectAsync();
      // …
      // make requests
      // process the event loop and handle the callbacks
      // …
    }

}
```

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

3.2.6 Using lambda expressions

The easiest way to receive a data or control messages into your application is to use a lambda expression.
For asynchronous connection you need to add a new event handler to the RTConsumer.ConnectCompleted event and call ConnectAsync function. For instance:

```
consumer.ConnectCompleted +=
        new EventHandler<ConnectCompletedEventArgs>(
          (object sender, ConnectCompletedEventArgs e) =>
          {
            OnConnect (sender, e);
          }
         );
consumer.ConnectAsync();
```

For data messages you need to pass a lambda expression as an argument to the RTConsumer.MakeRequest function. For instance:

```
var req = new RTRequest(svcName, symbol, isSnapshot);
      RTConsumer.RTSubscription sub = null;
      sub = consumer.MakeRequest(req, (RTConsumer.RTSubscription sub, RTMessage msg) =>
      {
        MsgHandler(sub, msg);
      }
        );
```

To check results of asynchronous message pumping you need to register a new handler for the RTConsumer.DispatchCompleted event. For instance:

```
consumer.DispatchCompleted +=
          (object sender, RunWorkerCompletedEventArgs e) =>
          {
            if (e.Error != null)
            {
              Console.Error.WriteLine("Dispatch error:\n\t{0}", e.Error.Message);
              isCanceled = true;
            }
            else
            {
              Console.WriteLine("Dispatch completed");
            }
          };
```

3.2.7 Proxy Support

FactSet now provides support for proxy servers in the Datafeed API. API users can set the proxy information by calling one of the following methods :

```
RT_Consumer::SetHttpProxy(System::String^ proxy_host);
RT_Consumer::SetHttpProxy(System::String^ proxy_host,
                          System::String^ proxy_user,
                          System::String^ proxy_password);
```

3.3 Requests and Cancels

3.3.1 Opening the Stream

Requests are made using the `RTConsumer.MakeRequest()` 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. The request method is defined as follows:

```
RTConsumer.RTSubscription MakeRequest(RTRequest req, RTConsumer.OnMessageDelegate
msgHandler);
```

The RTRequest class is the first parameter required by the MakeRequest() method. This class can be constructed using a service and 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 RTRequest 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 called a snapshot request.

The second parameter of the request method is the application-defined event handler procedure. This procedure is defined as an RTConsumer.OnMessageDelegate and must have the following signature:

```
void OnMessageDelegate(RTConsumer.RTSubscription sub, RTMessage msg);
```

This callback function will receive the message.
User can use lambda function to pass additional parameters to the function.

```
RTRequest req = new RTRequest("FDS1", "FDS-USA", true);
RTConsumer.RTSubscription subs = null;
subs = consumer.MakeRequest(req,
  (RTConsumer.RTSubscription sub, RTMessage msg) =>
  {
    Console.WriteLine("Snapshot message:\n{0}", msg);
    consumer.Cancel(subs);
  }
  );
```

MakeRequest returns a unique RTConsumer.Subscription object.

3.3.2 Closing the Stream

The messages for a stream will be passed to an event handler function. Eventually the stream should be closed and the RTConsumer.Subscription freed. This resource can be freed in either two ways: 1) calling RTConsumer.Disconnect() or 2) calling RTConsumer.Cancel(subscription). The stream will continue to be open until one of these two functions 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 (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 RTConsumer.Cancel() will close the stream on the client-side.

3.3.3 Subscription Ownership and Lifetime

Subscriptions are assigned by the API and given to clients. The lifetime of every subscription is controlled by the application. Applications create subscription via the call to the MakeRequest() method and free the subscription via the Cancel() method.

3.3.4 Dynamic Request

```
RTRequest req = new RTRequest("FDS1", "FDS-USA");
RTConsumer.RTSubscription subs = null;
subs = consumer.MakeRequest(req,
  (RTConsumer.RTSubscription sub, RTMessage msg) =>
  {
    Console.WriteLine("Streaming message:\n{0}", msg);

    if (msg.IsError)
    {
      Console.WriteLine("Error: {0}, {1}", msg.Error, msg.ErrorDescription);
    }

    if (msg.IsClosed || (msg.IsComplete && sub.IsSnapshot))
    {
      consumer.Cancel(subs);
    }
  }
);
```

The example code above shows a request being made for the symbol “FDS-USA” under the “FDS1” service. Since we are creating the RTRequest using (service, symbol) constructor, the request is for a dynamic subscription. The RTRequest object gets passed to the request method, along with the lambda function to handle the message update events. The Subscription for the stream is returned in the local variable subs.

The event handler 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 subs. The server may close the stream at any time. In addition, error messages (e.g., ErrorCode.NotFound) will cause the stream to set the close/end-of-stream indicator. The example handler covers both of these conditions.

3.3.5 Static Request

```
// create a static RTRequest for service=FDS1, symbol=IBM-USA
// The parameter "true" sets the IsSnapshot_property
RTRequest req = new RTRequest("FDS1", "FDS-USA", true);
RTConsumer.RTSubscription subs = null;
subs = consumer.MakeRequest(req,
  (RTConsumer.RTSubscription sub, RTMessage msg) =>
  {
    Console.WriteLine("Snapshot message:\n{0}", msg);

    // No reason to check IsClosed, since we only want a single
    // message. So just call Cancel() after processing
    consumer.Cancel(subs);
  }
);
```

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 event handler from the previous example could have been used in this example. Since static requests will close the stream on the first message, the previous handler would have cancelled the subs. However, the current example states the application's intentions more clearly when the call to Cancel() is explicit (like above), rather than based on a predicate (like the previous example).

3.3.6 Canceling Requests

Applications cancel the request using the Subscription object given at the time of request. Applications can cancel the subscription at any time (even before receiving the first message). Once the application returns from the Cancel() method, the event handler for the request identified by that subscription will NEVER be called [10]. This guarantee simplifies programming by allowing the application to clean up resources used for callback processing immediately after the call to Cancel(). This cleanup is typically done in destructors/finalizers, but that does not have to be the case.

3.4 Processing Events

3.4.1 Synchronous Dispatching

A server-side development use case:

```
static void Main(string[] args)
{
  // set up connection (see previous code)
  // make a request    (see previous code)

  // dispatch messages
  while (true) {
    consumer.Dispatch(-1);
  }
}
```

10 If the handler is registered by more than one stream, only the stream identified by the subscription is affected by a call to Cancel(). Streams are allowed to share event handlers, and canceling a stream will only prevent the callback from being used in the context of the cancelled subscription.

In order to dispatch messages to the appropriate callbacks, control must be handed back to the API. This is accomplished by calling `RTConsumer.Dispatch(…)`. This method will flush all of the currently queued messages and return.

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

Reponses from the FactSet Data Server are treated as events. These events are delivered via the handlers that were setup at the time of request. If the application wants to prevent blocking indefinitely, 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 `Dispatch()`, see section 4.6.1 RTConsumer Class.

3.4.2 Asynchronous Dispatching

The UI based applications like WPF, WinForms or ASP.NET should use asynchronous dispatching. If you call *RTConsumer.ConnectAsync*() or *RTConsumer.ConnectToWorkstationAsync*() the API will start a *BackgroundWorker* thread to dispatch messages and will call event handlers in the UI thread context in the background.

If you call the synchronous *RTConsumer.Dispatch*() function after invoking any of *ConnectAsync* functions the API will throw a *DatafeedException*.

It is possible to have asynchronous dispatching in a console applicationas well. For instance, you can make a class inherited from the Windows.Application class and override OnStartup to setup the *RTConsumer.ConnInfo* property, call the *RTConsumer.ConnectAsync* function, and start making requests. After you call Application.Run() the message event handlers will be called in the context of the main console thread. Please see the ConsoleAppRun sample for more details.

If you don’t use the Application class and connect asynchronously from a console application the messages still will be dispatched by the background thread but the event handlers will be called by the ThreadPool members and that could **break the sequence** of message delivery to your application logic.

For more details, look at the WinForms, WPF and ConsoleAppRun samples.

3.4.3 Handling Errors, Exceptions

Dispatch can throw DatafeedException. The exception is inherited from System.ApplicationException and provides ErrorCode and a failed function name as a Key/Value pair in the Data property. Having an exception with ErrorCode.Shutdown is a serious error and may be due to the application deleting the RTConsumer object, invoking Disconnect(), or not handling a TERMINATE[11] control message. In case that handling the exception depends on its particular ErrorCode, this can be obtained by a call to get_err_c(). The following example illustrates the synchronous case:

11 A TERMINATE control message is only given when an application issues an asynchronous connect, and the server failed to authenticate based on the user credentials given in RTConsumer.ConnInfo.

```
try
{
  consumer.Dispatch(333);
}
catch (DatafeedException ex)
{
  // Print exception message
  Console.Error.Write(ex.ToString());

  // Non-fatal error, consumer will try to reconnect on it's own.
  if (ex.get_err_c() == ErrorCode.NoConn) continue

  // Terminate in case of ErrorCode.Shutdown
  if (ex.get_err_c() == ErrorCode.NoConn) {
    break
  }
}
```

The samples contain examples of handling other DatafeedExceptions as well examples of handling those exceptions thrown during asynchronous dispatching.

3.4.4 The .NET Garbage Collector

If your application uses multiple threads and object instances, then use the server garbage collection instead of the workstation garbage collection. Server garbage collection operates on multiple threads, whereas the workstation garbage collection requires multiple instances of an application to run their own garbage collection threads and compete for CPU time.

An application that has a low load and performs tasks infrequently in the background, such as a service, could use the workstation garbage collection with concurrent garbage collection disabled.

To enable the server garbage collector, update the application configuration file as follows:

```
<configuration>
  <runtime>
    <gcServer enabled="true"/>
  </runtime>
</configuration>
```

3.5 Processing the Messages

3.5.1 RTFieldId 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 a field identifier RTFieldId.
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. User can use FidField.ValueType property to define the appropriate data type of the field value.

3.5.2 Field Identifiers

The current field identifiers are as RTFieldId enum.
There is a rt_fields.xml file (also included in the toolkit). This file can be loaded by the RTFieldMap class and allows applications to translate human readable names to field ids at runtime.

3.5.3 Messages

All requests open a stream, which is a virtual tunnel of messages. A message has certain properties such as a type, permissions, a key, and some other flags. Message data is contained in the RTMessage class.

An RTMessage is simply a container of fields (i.e., fids and values). The fields can be extracted using the member functions defined in the RTMessage class (see section 4.3.3 RTMessage Class for more information).

3.5.4 Processing a Message Example

```
using FactSet.Datafeed;

namespace SampleConsAppSynch
{
  class MainSync
  {
…
    static void MsgHandler(RTConsumer.RTSubscription sub, RTConsumer cons,
      RTMessage msg)
    {
      Debug.Assert(sub != null && sub.IsSubscribed);

      if (msg.IsClosed || (msg.IsComplete && sub.IsSnapshot))
      {
        cons.Cancel(sub);
      }

      DateTime msgTime = msg.MsgTime;
      Console.WriteLine("\nMessage received at {0}:{1}:{2}.{3}",
        msgTime.Hour, msgTime.Minute, msgTime.Second, msgTime.Millisecond);

      decimal? lastPrice = msg.Get<decimal>(RTFieldId.LAST_1);
      Console.WriteLine("Last price: {0}", lastPrice);

      int? lastVolume = msg.Get<int>("LAST_VOL_1");
      Console.WriteLine("Last volume: {0}", lastVolume);

      double? shares = msg.Get<double>(RTFieldId.SHARES_OUTSTANDING);
      Console.WriteLine("Shares outstanding: {0}", shares);
```

```
    DateTime? lastDate = msg[RTFieldId.LAST_DATE_1].Get<DateTime>();
    if (lastDate != null)
    {
      Console.WriteLine("Last date: {0}",
        ((DateTime)lastDate).ToShortDateString());
    }
    DateTime? lastTime = msg.Get<DateTime>(RTFieldId.LAST_TIME_1);
    if (lastTime != null)
    {
      Console.WriteLine("Last time: {0}",
        ((DateTime)lastTime).ToString("HH:mm:ss.fff"));
    }

    var fld = msg[RTFieldId.ASK_1];
    if (fld != null && !fld.IsEmpty) Console.WriteLine(fld.ToString());
    fld = msg["ASK_VOL_1"];
    if (fld != null && !fld.IsEmpty) Console.WriteLine(fld.ToString());

    fld = msg[RTFieldId.BID_1];
    if (fld != null && !fld.IsEmpty) Console.WriteLine(fld.ToString());
    fld = msg["BID_VOL_1"];
    if (fld != null && !fld.IsEmpty) Console.WriteLine(fld.ToString());
  }
…
}
```

The example code above shows one way to process a message in an event handler. The handler simply prints the message time along with the last, bid and ask fields. 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. There is no error handling in the sample.

3.5.5 Field Deltas

In order to help reduce the bandwidth of the feed on the wire, FactSet does not send fields that have not changed from one message to the next. For many applications this may be fine, however there are some cases where the presence of a field in a message may have a meaning to certain applications. FactSet provides two ways for applications to handle this.

Processing Per Message
Every message contains a special field, called UNCHANGED_FIELDS. This is a comma delimited list of fields that were present on the feed from the exchange, but were not sent out in the message because the value has not changed since the last time this field was sent. Looking at this field is useful for applications that may care about the presence of a small subset of unchanged fields, but don't need to handle the entirety of them.

Processing Per Connection
The API provides a connection-level interface to hide the UNCHANGED_FIELDS from the user, and to extract the values from the API's cache and inject them back into the message prior to handing off the message to the application level code. This is achieved by calling the method "SetSendUnchangedFields(bool enable）" on an instance of the RT_Consumer class. Setting the **enable** argument to **true** will turn this feature on, and users will see all the fields in the messages, regardless of whether or not they have changes. Note that there is some processing overhead to using this feature. By default, this feature is disabled.

3.5.6 Caching the record

Caching is not enabled by default. It is enabled by using the function EnableCaching in RTConsumer instance. The example code below shows the RTConsumer instance using EnableCaching function to cache the record.

```
using System;
using FactSet.Datafeed;

namespace DFsample
{
  class Program
  {
    static FieldMap fldMap;

    static void Main(string[] args)
    {
      fldMap = FieldMap.Create(@"rt_fields.xml");

      using (RTConsumer consumer = new RTConsumer())
      {
        consumer.ConnInfo = @"username:password@api.df.factset.com";
              // Enable caching
              consumer.EnableCaching(true);
       // set up connection
              // make requests
              RTRequest req = new RTRequest("FDS1", "FDS-USA");
              RTConsumer.RTSubscription subs = null;
              subs = consumer.MakeRequest(req,
       (RTConsumer.RTSubscription sub, RTMessage msg) =>
        {
          // Get the cached Record
          RTRecord rec;
          consumer.GetRecord(sub, out rec);

          Console.WriteLine("Streaming message:\n{0}", msg);

          if (msg.IsError)
          {
           Console.WriteLine("Error: {0}, {1}", msg.Error, msg.ErrorDescription);
          }

          if (msg.IsClosed || (msg.IsComplete && sub.IsSnapshot))
          {
            consumer.Cancel(subs);
          }
        }
       );

      }
    }
  }
}
```

3.6 Threading

3.6.1 Thread-safe Classes

The only two classes that are completely thread-safe are the RTConsumer and FieldMap classes. These classes manage the requests, cancels, and the connection to the FactSet Data Server. Applications are free to call the methods of both the RTConsumer and FieldMap class using multiple threads.

3.6.2 Thread-unsafe Classes

The remaining classes are thread unsafe. The reason is that these classes tend to be used by a single thread at a time. The RTRequest, RTMessage, are all container classes that are usually used by a single thread. Applications should provide their own locking if these objects need to be shared by multiple threads.

3.6.3 Class-thread-safe

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

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

3.6.4 Threading Issues Using an Event-driven API

A potential for deadlock exists when:

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

OR
The callback routine needs to wait on a thread that uses the same RTConsumer object.

3.7 Chain Requests

3.7.1 Option Chains

Use the service FDS_OPT_CHAIN to request the underlying options available for a root symbol. For additional filtering to the list of option contracts returned for the root symbol filters can be applied.

The available filters are:

- **Strikes**: atTheMoney, inTheMoney, outOfMoney for at the money, in the money and out of money respectively. Case is not sensitive. It is also possible to filter by a strike price range, strikes=400-600 to get the options in that range of strikes.
- **Put or Call:** put or call, for one type only
- **Months**: Months=1,2 up to 12 for January to December. nearThreeMonths: for the next 3 months
- Chaining filters: filters can be chained by using &, examples:
    - months=5,6,7&inTheMoney
    - strikes=210-450&months=1,2,3
    - inTheMoney&outOfMoney&put
    - put&nearThreeMonths

And any combination of chaining is allowed.

The filter is defined within the options of the FDS_OPT_CHAIN request, if the options are left blank all available contracts will be returned.

In the below example Apple put options expiring in July with a strike of 300 will be returned.

```
RTRequest chainreq = new RTRequest("FDS_OPT_CHAIN", "AAPL-USA");
chainreq.Options = "months=7&strikes=300-300&put"
```

3.7.2 Future Chains

Use the service FDS_FUT_CHAIN to request the underlying future contracts available for a root symbol, using the continuous symbol. For example, use the continuous symbols for Corn, C00-USA, to return the available, active, Corn contracts.

Chapter 4 API Class Reference

4.1 API Constants

4.1.1 Error Codes

All errors within the API are conveyed to the application via the DatafeedException.Data collection property. The key part of each element has the type of Enumeration `ErrorCode`. The list of possible errors is noted below.

```
enum ErrorCode {
  NoError    =  0,   // All is good...
  Unknown    = -51,  // Unknown error
  NoServ     = -52,  // No Service Available
  NotFound   = -53,  // Field, or Record not found
  Rename     = -54,  // Record has been renamed
  TimedOut   = -55,  // Operation Timed Out
  Exists     = -56,  // Already exists
  Limit      = -57,  // Maximum application limit has been reached
  Protocol   = -58,  // Any Protocol error (message, file format, network)
  Inval      = -59,  // Invalid parameter to method call
  Resource   = -60,  // Operating system resource exhausted
  NoConn     = -61,  // No connection to the server
  Version    = -62,  // Incorrect version
  Shutdown   = -63,  // User has shutdown the system
  Access     = -64   // Permission denied
```

4.1.2 Field Identifiers

Field identifiers are integers that can be used to index into the `RTMessage` object. The list of field identifiers is available programmatically as RTFieldId enumeration.

4.1.3 DatafeedException

The API reports an error by throwing the DatafeedException object. The class is inherited from ApplicationException.

```
namespace FactSet.Datafeed
{
  public class DatafeedException : System.ApplicationException
  {
    public DatafeedException (ErrorCode rc, string funcName, string message);

    public ErrorCode get_err_c();

    public virtual string ToString() override;
  }
}
```

4.2 Requests

API requests are made using the RTRequest class. An RTRequest is a simple container class for the type of request, the service for which the request is destined, and the key identifying a record within a particular service.

```
namespace FactSet.Datafeed
{

 public class RTRequest
 {
  public RTRequest();
  // Constructor of streaming or static RTRequest object.
  public RTRequest(string svcName, string key, bool snapshotOnly);
  // Constructor of streaming RTRequest object.
  public RTRequest(string svcName, string key);

  // The ServiceName property represents the Datafeed service name.
  public string ServiceName { get; }
  // The Key property identifies the subject being requested.
  public string Key { get; }
  // The IsSnapshot property represents the snapshot only request.
  public bool IsSnapshot { get; }
  // The Options property represents additional request data.
  public string Options { get; }
  // The AuthenticationToken property contains authentication information.
  public string AuthenticationToken { get; }

  // Returns a string that represents the RTRequest object.
  virtual override string ToString();
 }
}
```

4.2.1 RTRequest Class

```
public RTRequest();
// Constructor of streaming or static RTRequest object.
public RTRequest(string svcName, string key, bool snapshotOnly);
// Constructor of streaming RTRequest object.
public RTRequest(string svcName, string key);
```

The RTRequest object lifetime is controlled by its creator (usually the application itself). The API simply accepts constant references to these objects.

Example

RTRequest req = new consumer.MakeRequest (“FDS1”, “FDS-USA”);
The above example creates an RTRequest object that can be used to request the symbol “FDS-USA” from the service “FDS1.” Since the snapshot only flag is set to false (the default value), this request will initiate a stream of updates.

RTRequest Interface

```
// The ServiceName property represents the Datafeed service name.
public string ServiceName { get; }
// The Key property identifies the subject being requested.
public string Key { get; }
// The IsSnapshot property represents the snapshot only request.
public bool IsSnapshot { get; }
// The Options property represents additional request data.
public string Options { get; }
// The AuthenticationToken property contains authentication information.
public string AuthenticationToken { get; }
```

4.3 FID Fields and Messages

The `RTMessage` class is the class which represents all messages in the system. The API delivers `RTMessage` references to client event handler. This class contains information applicable to all messages. For example, messages will have a key, message flags, message state, and may also have associated permission information and/or an error condition. Messages also contain a collection of RTFidFields.

4.3.1 FID Fields

A FID field is data that is identified by an RTFieldId enum value.

```
namespace FactSet.Datafeed
{
  public class FidField : IComparable<FidField>, IEquatable<FidField>
  {
    public RTFieldId Id { get; }
    public RTFieldType ValueType { get; }

    public string Name { get; }
    public string Value { get; set; }

    public override string ToString();
    public override bool Equals(object obj);
    public bool Equals(FidField other);

    public int CompareTo(FidField other);

    public bool IsEmpty { get; }

    public Nullable<T> Get<T>() where T : struct, ValueType;
  }
} //  FactSet.Datafeed
```

FidField Interface

- `FidField.Empty()` - returns true if there is no data or the size of the data is zero.
- `FidField.Id` - is the field identifier.
- `FidField.Value` - represents the data for this field identifier.
- `FidField.Get<T>` - returns a nullable value converter to type T

4.3.2 Messages

```
namespace FactSet.Datafeed {
  public class RTMessage : IEnumerable<FidField>, IEnumerable
  {
    public string Key { get; set; }

    public bool IsClosed { get; set; }

    public string Permissions { get; }

    public bool IsPermissioned { get; }

    public bool IsResponse { get; set; }

    public bool IsUpdate { get; }

    public bool IsComplete { get; set; }

    public bool IsError { get; }
    public ErrorCode Error { get; }
    public  string ErrorDescription { get; }

    public bool IsStale { get; }
    public string StaleReason { get; }

    public bool IsAggregated { get; }

    public Nullable<T> Get<T>( RTFieldId id ) where T : struct, ValueType;

    public Nullable<T> Get<T>( string fldName ) where T : struct, ValueType;

    public FidField GetField( RTFieldId id );

    public bool ContainsField( RTFieldId id );
    public bool ContainsField( string idName );

    public int Count { get; }

    public override virtual string ToString();

    public FidField this[RTFieldId id] {}

    public FidField this[string fldName] {}

    public DateTime MsgTime { get; }
  }
} //  FactSet.Datafeed
```

4.3.3 RTMessage Class

Object Creation, Destruction, and Lifetime
The RTMessage class is can be created by applications. In this case, the lifetime of these objects are strictly managed by the creator. Typically, applications will be given reference to RTMessage objects as event handler parameters. These objects are owned by the API and their lifetime is valid during the callback routine only.
RTMessage Interface

The following properties allow the application to query various pieces of information.

- **string Key** – returns key identifying the subscription within a particular service.
- **string StaleReason** - Should be used if the message is stale (i.e. IsStale returns true).
- **ErrorCode Error** - returns the error code of an error message. For a complete list of possible errors see Appendix B.
- **string ErrorDescription** - Should be used if the message is error (i.e. IsError returns true).
- **bool IsStale()** - returns true for stale messages.
- **bool IsError()** - returns true for error messages.
- **bool IsClosed()** - returns true when the stream is closed.
- **bool IsResponse()** - returns true for the initial message.
- **bool IsUpdate()** - returns true if the message is an update to an initial message (opposite of IsResponse).
- **bool IsComplete** - returns true if this message is the last message in an initial response, which may include multiple messages.
- **bool IsAggregated** - returns true if this message has been combined with similar updates because of bandwidth constraints.
-

The get functions allow the user to extract fields identified by an RTFieldId enum or field name.

- **Nullable<T> Get<T>(RTFieldId id) where T : struct, ValueType**
- **Nullable<T> Get<T>(string fldName) where T : struct, ValueType**
  - Returns the nullable field data converterd to type T. If the FID is not found or not convertible, null is returned.

FidField GetField(RTFieldId id) - returns a FidField object.
Methods to query field information

- **bool IsEmpty()** - returns true if there are no fields in the message.
- **bool ContainsField(RTFieldId id)** - returns true if the field exists in the message.
- **bool ContainsField(string idName)** - returns true if the field exists in the message.
- **int Count()** - returns the number of fids in the message.

In addition, the RTMessage class offers the possibility to serialize objects to byte array and deserialize them back to RTMessage objects.
.

- **public ErrorCode Serialize(IntPtr byteArray, IntPtr size);**
- **public ErrorCode Deserialize(IntPtr byteArray, int size);**
- **public int Size();**

To serialize an RTMessage object, start by creating an empty byte array with the size of the message. Call Size() to get the message size. To access the managed object from unmanaged memory we need to pin the object in memory using a GCHandle and get a pointer to it's memory address. See below for details:

```
  //Create a byte array and get a pointer to it
byte[] byteArray = new byte[message.Size()];
GCHandle pinnedByteArray = GCHandle.Alloc(byteArray, GCHandleType.Pinned);
IntPtr pinnedArrayPtr = pinnedByteArray.AddrOfPinnedObject();

//Get a pointer to the message size
int msgSize = message.Size();
GCHandle pinnedMsgSize = GCHandle.Alloc(msgSize, GCHandleType.Pinned);
IntPtr msgSizePtr = pinnedMsgSize.AddrOfPinnedObject();

//Get serialized message to pinnedArrayPtr
ErrorCode result = message.Serialize(pinnedArrayPtr, msgSizePtr);
```

Deserialize from byte array to object as below:

```
 //Deserialization
//Create an empty RTMessage and and pass the byteArray to be Deserialized
RTMessage msgDeserialized = new RTMessage();
ErrorCode errorCode = msgDeserialized.Deserialize(pinnedArrayPtr, byteArray.Length);
```

Make sure to free the handles as soon as possible to improve the efficiency of the garbage collector.

```
  //Free the handles
pinnedByteArray.Free();
pinnedMsgSize.Free();
```

4.4 Records

The API supports internal caching of certain types of records. This allows applications to reduce the amount of state needed to keep up-to-date. The RTRecord is the class which represents all records in the system.

```
// #include "RTRecord.h"
namespace FactSet {
  namespace Datafeed {
    #pragma region RTRecord class
    /// <summary>
```

```
    /// RTRecord class manages as a wrapper for RT_Record class.
    /// It is the class used to fetch details of RT_Record class in .Net.
    /// </summary>
    public ref class RTRecord : public System::ComponentModel::Component
    {
    public:
      RTRecord(const FIRE_NAMESPACE::RT_Record* rt_rec);
      ~RTRecord();
      virtual System::String^ ToString() override;
      String^ GetKey();
      String^ Get(int Fid);
      String^ GetByIdx(int idx);
      int Count();
      bool Exists(int Fid);
      bool Empty();
      bool IsStale();
      bool IsPermissioned();
      String^ GetPermissions();
      String^ GetStaleReason();

    private:
       const FIRE_NAMESPACE::RT_Record* m_rec;
    }
   #pragma endregion
  }
}
```

4.4.1 RTRecord Class

#include "RTRecord.h"
The RTRecord class is given to the application as a callback parameter. Applications can use this interface to gain access to fields that have not changed and thus may not be in the RTMessage object. This object represents the current state of the record after the current update message has been applied.
Object Creation, Destruction, and Lifetime

Applications should not create RTRecord classes. This object is passed as a callback parameter. The lifetime of this parameter is valid during the callback routine only.

RTRecord Interface

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

- **String^ GetKey() –** returns the key of the record in String structure.
- **bool IsStale() –** returns true for stale records
- **bool IsPermissioned() –** returns true if the record has permissions set.
- **String^ GetPermissions() –** returns the permissions in the form of string.
- **String^ GetStaleReason() –** returns the stale reason in String structure.

The get functions allow the user to extract fields identified by an integer fid.

- **String^ Get(int Fid) –** returns field data in the form of string.
- **String^ GetByIdx(int idx) –** gets the field at a particular index. The index is zero-based to a maximum value of the count() – 1

Methods to query field information:

- **int Count() -** returns the number of fids in the record.
- **bool Empty() –** returns true if there are no fids in the record.
- **bool Exists(int Fid) -** returns true if the FID exists in the record.

4.5 Field Translation

Field IDs are simple values from the enum RTFieldId. However, these integers are typically managed by an associated human-readable name. The `FieldMap` class assists in translating names to ids and vice versa.

```
namespace FactSet.Datafeed
{
  public enum RTFieldType
  {
    Reserved = 0,
    Character = 1,
    String = 2,
    Integer = 3,
    Price = 4,
    Decimal = 5,
    Enumeration = 7,
    Date = 8,
    Time = 9,
  }

  // FieldDefinition class defines a message data field ID, name and value type.
  public class FieldDefinition : IComparable<FieldDefinition>,
    IEquatable<FieldDefinition>
  {
    public RTFieldId Id { get; }  //  Field identifier.
    public string Name { get; }  // Field name.
    public RTFieldType ValueType; //  Field value type.
  }

  // The FieldMap class is available to applications for translating names
  // to field ids and vice-versa. Translating FIDs to names assists with
  // debugging messages and also allows for human-readable configuration files.
  public sealed class FieldMap : IEnumerable<FieldDefinition>
  {
    // A factory function to create a FieldMap object by loading field
    // definitions from an XML configuration file.
    public static FieldMap Create( string fileName );
    // <summary>Destroys internal state and frees the memory.</summary>
    public static void Destroy();

    // XML configuration file name.
    public static string FileName { get; }
    // The number of fields in the map.
    public static int Count { get; }
    // The number of fields in the map.
    public static int Size { get; }

    public static string GetFieldName( RTFieldId fid );

    // Indexers
    // Returns field definition by index.
    public static FieldDefinition get_Fld(int idx);
    public FieldDefinition this[int idx] {}
    // Returns field definition by field ID.
    public FieldDefinition this[RTFieldId fldId] {}
    // Returns field definition by field Name.
    public FieldDefinition this[string fldName] {}
  }
}
```

4.5.1 FieldMap Class

The `FieldMap` class is available to applications for translating field names to field ids and vice-versa. Translating FIDs to names assists with debugging messages and records and also allows for human-readable configuration files. In essence, this class allows names to be translated to numbers at run-time.

Object Creation, Destruction, and Lifetime

Applications can create an FieldMap class using the static function FieldMap.Create(…). This will create a FieldMap object based on the filename specified as the parameter to Create(). The file should adhere to the correct xml specification for a field's file (see the *FactSet Data Service Specification Document*). A FieldMap file is supplied in the toolkit. The location is "etc/rt_fields.xml". Applications that wish to utilize name translation should load this file at program startup.

Once the object is created, it is owned by the application. Applications may destroy the object anytime using the static member function FieldMap.Destroy(). It is recommended that applications create a FieldMap object during startup, and let the operating system clean up the system memory resources during program shutdown.

FieldMap Interface

In addition to each field having a name and *RTFieldId* identifier, every field is associated with an application-defined *RTFieldType*. It is important to note that once a type is published for a specific field identifier, it will NEVER be changed. Applications may make assumptions about the published type of the field at compile time. For a more detailed description about each type, see the *FactSet Data Service Specification Document.*

- static FieldMap Create(string fileName);
- static void Destroy();

The above functions can create and destroy FieldMap objects. To create an FieldMap object an xml file must be passed as an argument. The Create() functions will throw an exception if the file could not be opened. To destroy a FieldMap object, call the FieldMap.Destroy() method. This function will also make sure the the internal default map file is cleared as well.

- FieldDefinition FieldMap[int idx]
- FieldDefinition FieldMap[RTFieldId fldId]
- FieldDefinition FieldMap[string fldName]

The indexers allows the application to get the field definition object via its integer index, FieldId or name. If the field id is not found, the method will return null.

- **`int Count()`** - returns the number of fields in the map.

The above functions are used to support iteration of all the fields within the FieldMap.

4.5.2 Control message type enum and event handler argument class

These types are used to process control messages from FactSet Data Server,

```
enum ControlType {
  UNKNOWN = 0,
  DISCONNECTED, // Not connected to data server.
  CONNECTED,  // Connected to data server.
  SERVICE_ENABLE, // New service(s) is available for requests.
  SERVICE_DISCONNECT, // Service(s) is no longer available.
  SERVICE_DISABLE,  // Service(s) is no longer available.
  TERMINATE // The FactSet data server is requesting the application
  // to terminate its connection (i.e. the application MUST call disconnect()).
}

// On control message event handler argument.
public class ConnectCompletedEventArgs :
  System.ComponentModel.AsyncCompletedEventArgs
{
  internal ConnectCompletedEventArgs(RTConsumer cons, bool isConnected,
    RTMessage cntrl_msg, System.Exception e, bool canceled, object userState);

  public RTConsumer Consumer { get; }

  // The control message.
  public RTMessage CntrlMsg { get; }

  // Indicates the current status of the connection to the
  // data server.
  public bool IsConnected { get; }
  // Type of the delivered control message. See ControlType enum.
  public ControlType CtrlType;
}
```

4.6 RTConsumer

The RTConsumer class manages the connection to the FactSet Data Server. It is the class used to connect, request data, cancel subscriptions, manage the event loop, and finally disconnect. This class is the heart of the FactSet Real-Time API.

```
public class RTConsumer : System.ComponentModel.Component
{
  public RTConsumer();

  // Delegate to handle incoming data messages.
  public delegate void OnMessageDelegate( RTConsumer.RTSubscription sub,        RTMessage msg );

  // Connection info property in format: UserName:Password@HostName:PortNumber
  // PortNumer is optional.
  // Use pipe '|' delimiter to indicate multiple connection strings.
  public string ConnInfo { get; set; }

  // Helper function to create connection info string.
  // Returns a connection string or String.Empty if userName or hostPort
  // parameters are empty.
  public static string MakeConnInfo( string userName, string pwd,
    string hostPort );
  public static string MakeConnInfo( string userName, string pwd,
    string host, int port );

  // RTSubscription class is a unique resource identifier created per request.
  public class RTSubscription : IEquatable<RTSubscription>,
    IComparable<RTSubscription>
  {
    // The symbol subscribed to.
    public string Key { get; }

    // Indicates if the object is still active.
    public bool IsSubscribed { get; }
    // Indicates if a snapshot request was issued.
    public bool IsSnapshot { get; }

    // Total number of data messages received.
    public long Count { get; }
  };

#region Synchronous connect
  // The function synchronously connects the API to a FactSet
  // server over an Internet or WAN connection. The connection info must be
  // set by ConnInfo property. User has to run the Dispatch()
  // event loop to receive the data messages.
  public void Connect();

  // The function synchronously connects the API to a FactSet
  // workstation application running on the current machine.
  // User has to run the Dispatch()
  // event loop to receive the data messages.
  /// Remarks: This function will fail if version 2011.1 or higher of the
  // FactSet workstation is not installed. If the FactSet workstation application
  // is not running, the function will start it.
  public void ConnectToWorkstation();
#endregion

#region Asynchronous connect
```

```
    // Control messages event handler.
    // The dispatch event loop has to be running to receive the data messages.
    // Remarks: See also 'Event-based Asynchronous Pattern Overview' in MSDN for more details.
    public event EventHandler<ConnectCompletedEventArgs> ConnectCompleted;

    // The function asynchronously connects the API to a FactSet
    // server over an Internet or WAN connection. The connection info must
    // be set by ConnInfo property.
    // See 'Event-based Asynchronous Pattern Overview' in MSDN for more details.
    public void ConnectAsync();

    // The function asynchronously connects the API to a FactSet
    // workstation application running on the current machine.
    // User has to run the Dispatch()
    // event loop to receive the data messages.
    // Remarks: This function will fail if version 2011.1 or higher of the
    // FactSet workstation is not installed. If the FactSet workstation application
    // is not running, the function will start it.
    public void ConnectToWorkstationAsync();

    // Cancels asynchronous connection request if it has not completed.
    public void ConnectAsyncCancel();

    /// Indicates if asynchronous connection is in progress.
    public bool IsBusy { get; }
#endregion

    // Synchronously dispatches messages to message handlers.
    // The use has to connect to FactSet server using Connect() function.
    void Dispatch( int millisecondsTimeout );

    // Asynchronous dispatching has completed.
    // Remarks: Check the RunWorkerCompletedEventArgs.Error property for asynchronous dispatch errors.
    public event System.ComponentModel.RunWorkerCompletedEventHandler DispatchCompleted;

    // Disconnects from FactSet server and cancel all active Subscriptions.
    public int Disconnect();
    // Status of current connection.
    public bool IsConnected { get; }

    // Returns the hostname of the current connection.
    public string ConnectedToHost { get; }

    // Issues a request to FactSet Data server.
    // Returns an active RTSubscription object.
    public RTSubscription MakeRequest( RTRequest req, OnMessageDelegate msgHandler );

    // Cancels the subscription.
    public ErrorCode Cancel( RTSubscription sub );

    // Retrieves the available services (i.e., data sources).
    public List<string> Services { get; }
  }
```

4.6.1 RTConsumer Class

The RTConsumer class manages the connection to the FactSet Data Server. Applications will use this object to open a connection to the FactSet Data Server and request information. This class will manage all the subscriptions on behalf of the application.

Object Creation, Destruction, and Lifetime

Applications create and control the lifetime of RTConsumer objects.

RTConsumer Interface

Defining the EventHandler signatures

The application can receive three types of notification event handlers. This first type is a control notification:

- public event EventHandler<ConnectCompletedEventArgs> ConnectCompleted;

Applications that want to receive control messages should define an event handler with the above signature to receive these events. Control notifications will typically indicate events such as Connected, Disconnected, Service Attachment, and so on. For a complete list of control messages, see the Appendix located at the end of this document. For instance:

```
cons.ConnectCompleted +=
        new EventHandler<ConnectCompletedEventArgs>(
          (object sender, ConnectCompletedEventArgs e) =>
          {
            OnConnect(sender, e);
          }
          );
```

The event argument property (`ConnectCompletedEventArgs.IsConnected`) indicates if the current connection is valid. The ConnectCompletedEventArgs.CtrlType propety has information such as the type of control notification.
The second type of notification is for application messages to items requested:

- delegate void OnMessageDelegate(RTMessage msg);

Applications should define the event handler to receive messages for requests. For instance:

```
var req = new RTRequest("FDS1", "F-USA");
RTConsumer.RTSubscription sub = null;
sub = consumer.MakeRequest(req, (RTConsumer.RTSubscription sub, RTMessage
msg) => { MsgHandler(sub, consumer, msg); } );
```

Registering for Control Messages

The application can install a single callback to receive control notifications. Control messages are used to dynamically inform applications of various events such as the following:

Removal of a service – When a service is removed and no longer available for requests, a control message will be sent to the application.

Addition of a new service – When a new service has attached and is ready to accept new connections a control message will be sent to the application.

TCP connection termination – When a connection socket is terminated (for any reason), a control message will be sent. The reason for termination will be included in the control message.

TCP connection notification (for asynchronous connections) – When the application is using asynchronous connections, the API will deliver a CONNECTED control message upon a successful TCP connect.
For complete details on the type of control messages that can be received, as well how each one should be handled see Appendix C.

In case if user using a synchronous dispatching, the third type of notification is RTConsumer.DispatchCompleted event should be set to check the RunWorkerCompleteEventArgs.Error property result for a dispatch error. For instance:

```
consumer.DispatchCompleted +=
  object sender, RunWorkerCompletedEventArgs e) =>
 {
   if (e.Error != null)
   {
     Console.Error.WriteLine("Dispatch error:\n\t{0}", e.Error.Message);
     isCanceled = true;
   }
   else
   {
     Console.WriteLine("Dispatch completed");
   }
 };
```

Setting and Getting the Connection Information

Use the RTConsumer.ConnInfo property and the RTConsumer.MakeConnInfo helper functions to set the connection information:

```
string ConnInfo;
static string MakeConnInfo( string userName, string pwd,
     string hostPort );
static string MakeConnInfo( string userName, string pwd,
     string host, int port );
```

In order to connect to the FactSet Data Server, 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 `MakeConnInfo()` methods as parameters. The functions take null-terminated strings as input. The following outlines some examples.

consumer.ConnInfo = "client:aaa@api-stage.factset.com" – The API will connect to the host "api-stage.factset.com" on the default port of "6681". It will use a username of "client" and a password of "aaa".

consumer.ConnInfo = "client@10.2.4.5:4063" – The API will connect to the host 10.2.4.5 on port 4063 using the username "client". The password is empty in this case.**consumer.ConnInfo = "reg:/HKEY_LOCAL_MACHINE/Software/FactSet/FDF"** – The API will look for a property named RT_CONNECTION in the Windows registry. The hive location is HKEY_LOCAL_MACHINE, and /Software/FactSet/FDF is the path within the hive.

consumer.ConnInfo = "file:/etc/connection_info.cfg" – The API will look for a property named RT_CONNECTION in the file /etc/connection_info.cfg.

It is also possible to set multiple connection strings at once by concatenating each string, separated by a pipe ('|'). The following is an example.

consumer.ConnInfo = "client:aaa@api-stage.factset.com|bryan:aaa@api-stage2.factset.com" – 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 connecting using each connection string.

Setting the ConnInfo property will only throw DatafeedException if the host could not be extracted from the specified URI. It does not check if the host is valid, or if the host can be translated to an actual IP

address. It will simply store the connection information for later use. The Connect() methods will later use this information to resolve the hostname and port before attempting the connection to the FactSet Data Server.

The following code retrieves the connection information:string connInfo = consumer.ConnInfo;

> ❖ The RTConsumer.ConnInfo property does NOT return the same string that was passed to it. The ConnInfo can take a URI resource. This resource can identify a file, registry location, or the connection string itself. Instead, get ConnInfo returns the resolved user, password, host, and port information from the set URI.

Connecting to a FactSet server

- void Connect();
- void ConnectAsync();
- void ConnectAsyncCancel();

The Connect functions connect the API to a FactSet server over an Internet or WAN connection. The connection info must be set with the ConnInfo property.

Connecting to a local FactSet workstation

- void ConnectToWorkstation();
- ConnectToWorkstationAsync();

The ConnectToWorkstation functions connect the API to the FactSet workstation program running on the current machine. It is not necessary to set the connection info when using the ConnectToWorkstation functions.

These functions are only supported in Windows and will fail if version 2011.1 or higher of the FactSet workstation is not installed. If the FactSet workstation program is not running, the ConnectToWorkstation functions will attempt to start it.

Asynchronous vs. synchronous connect calls

Connect() and ConnectToWorkstation () are synchronous operations, and thus, will block until a valid connection is established. If the functions return without throwing an exception, applications can assume that the connection is valid. If an exception is thrown, the connection attempt has failed. Applications should retry the connect operation at some time in the future or exit. Although applications can issue requests before a successful connection, applications will NOT be able to call Dispatch(), since Dispatch() will throw DatafeedException with the Shutdown ErrorCode.

Applications that wish to connect asynchronously (i.e., non-blocking), should use Async functions. The connect functions will return immediately. If an error is returned, the asynchronous connect has failed [12]. If an asynchronous connection operation returns without error, the connection is in progress. A control event handler will be invoked upon a successful or unsuccessful connect.

- If connect function throws an exception, the connection will never get established. Applications must issue a successful connect before dispatching any messages. This behaviour is true for both asynchronous and synchronous connections.

12 Applications should exit if an async connect operation fails. This is an unrecoverable, serious error.

Errors

Connect functions can throw DatafeedException with the following errors:

Error	Description
Version	Returned if the incorrect library is linked at compile time.
Inval	Returned if the application did not set the host and port using ConnInfo property.
Protocol	Returned if the connection is not returning the valid protocol. This may occur if the application attempts a TCP connect to some unknown server.
Access	Returned if the username and/or password are invalid.
{System Errno}	If the API could not resolve the host name, open the TCP connection, or create the communication thread a system errno is returned. It is a positive error number from the native platform.

ConnectToWorkstation functions can throw DatafeedException with the following errors:

Error	Description
Version	Returned if the incorrect library is linked at compile time.
Protocol	Returned if the connection is not returning the valid protocol.
Access	Returned if the username and/or password are invalid.
Resource	Returned if the API is unable to communicate with the FactSet workstation. This can occur if the proper version of the FactSet workstation isn't installed on the machine or the API is unable to start the workstation program.
{System Errno}	If the API could not resolve the host name, open the TCP connection, or create the communication thread a system errno is returned. It is a positive error number from the native platform.

Disconnecting and querying the connection status and available services

- ErrorCode Disconnect();

Disconnect will tear down the TCP connection to the server. Applications can specify whether the internal subscriptions should also be cleaned up. The default is to cancel all the subscriptions when the application issues a Disconnect.

- bool IsConnected { get; }

The IsConnected property simply returns the status of the connection. It is possible that a network or server condition can cause a TCP disconnect during normal operation. In this case, the API will attempt to retry the connection every so often. Immediately after a disconnect occurs all open streams will receive stale messages. When the connection is re-established the open streams will be notified (via the EventHandler) with the refreshed non-stale data.

- public string ConnectedToHost { get; }

ConnectedToHost property returns the hostname of the current connection. If there is no valid connection, it will return null. This is most useful when multiple connection strings are specified in ConnInfo.

- public List<string> Services { get; }

`Services` property retrieves all the available services (i.e., data sources).

Requesting and canceling data streams

- RTConsumer.RTSubscription MakeRequest(RTRequest, RTConsumer.OnMessageDelegate);
- ErrorCode Cancel(RTConsumer.RTSubscription);

MakeRequest() and Cancel() are the main entry points to open and close data streams. Applications call the request method to initiate a stream. The request should be passed in via the RTRequest object (see section 4.2 Requests for more information). OnMessageDelegate callback parameter has to be passed into the API. The function returns an RTSubscription object. This is the resource that identifies the open stream. Applications should cancel the subscription using the Cancel() method.

If the MakeRequest() function doesn't throw an exception it will ALWAYS return a subscription. For example, if the application requests a service which is not known, the request is still put in a queue, and when the service is attached, the request will be sent. If a null pointer is passed as a parameter to MakeRequest(), the function will throw an ArgumentNullException.

Also, it is possible to issue requests on a disconnected system. In this case, the request will be sent as soon as the connection is established.

❖ Every successful call to MakeRequest() will generate a subscription. The application is responsible for managing that resource. There are only two ways to free the resource: Cancel() method or calling Disconnect().

Managing the event loop

Eventually applications will have to return control to the API so the API can dispatch the message and control callbacks. This is accomplished by calling `Dispatch()`.

- **`void Dispatch( int millisecondsTimeout );`**

This method will flush the events from an internal notification queue. If there are no events to dispatch, the Dispatch() call can wait for events by specifying the milliseconds Timeout parameter. This parameter is the maximum time-to-wait in milliseconds. A negative wait time means wait indefinitely.

The Dispatch() method will always return after flushing the notification queue. The application should call this function multiple times either in a loop or in a system notification procedure. If a wait time is specified, and events are currently in queue, Dispatch() will flush the events (by calling the appropriate event handlers), and then return immediately.
Dispatch() waits only if there are no events to be dispatched.

Dispatch can throw DatafeedException with two errors: ErrorCode.NoConn and ErrorCode.Shutdown. The Shutdown is a serious error and may be due to the application deleting the RTConsumer object, invoking Disconnect(), or not handling a TERMINATE control message.

❖ As long as the application issues a successful Connect() and NEVER deletes the underlying RTConsumer object, Dispatch() will never throw ErrorCode.Shutdown.

The exception with NoConn error is thrown when the server or network disconnects the TCP connection to the API. This error is not as serious as Shutdown. As long as the API was once connected (via a successful call to Connect()), the API will retry the connection every so often (see section 3.4.3 Handling Errors, Exceptions). Applications are encouraged to maintain the event loop during this time period, however, they can disconnect if they choose to do so.

Although many server-side or console applications can call `Dispatch()` in a loop, many GUI or server-type applications will need to manage their own event loop. RTConsumer is inherited from the System.ComponentModel.Component class. If a user is developing GUI application using WinForms, WPF, ASP.NET, they should connect by ConnectAsync functions, and the RTConsumer will pump messages in a background thread. The correspond Application.Run will dispatch event callbacks in GUI thread.

4.6.2 Logging Within and Outside the API

The API consists of two code layers: .NET wraps the managed (C++/CLI) code which wraps the underlying unmanaged (native C++) core.

The managed layer uses the Diagnostics.Debug class to report an error, so the user can easy redirect the output to any TraceListener.

The unmanaged layer by default logs errors to standard error stream (stderr/cerr). Console based applications can redirect stderr to a file by using the `RTLogger.SetSink()` method. Applications that run as a Windows Service do not have this option, so a different interface is required here. Service based applications should use the method `RTLogger.OpenFile()` method to set the log file for the toolkit's unmanaged layer. This file will contain all messages that the toolkit logs, based on the log level set.

- static LogLevel RTLogger.Level

By default, all levels are logged. Applications can change the minimum level that will be logged. For example, a value of LogLevel.All means that all levels will be logged, and a value of RTLogLevel.None means nothing will be logged. A value of LogLevel.Warning will log all levels greater than or equal to Warning (i.e. Warning and Error). The following logging levels are supported:

```
enum LogLevel {
  All,
  Debug,
  Info,
  Warning,
  Error,
  None
};
```

- **static void RTLogger.SetSink(System.IO.FileStream stream)**
  - Redirects stderr and cerr to passed stream.
- **static void RTLogger.RestoreOutput()**
  - Restores output to stderr and cerr to previous state.
- **static void RTLogger.OpenFile(System::String^ filepath)**
  - Opens a file at the given path for unmanaged logging. The file is truncated.
- **static void RTLogger.OpenFile(System::String^ filepath, bool append)**
  - Opens a file at the give path for unmanaged logging. The file is appended to or truncated.
- **static void RTLogger.CloseFile()**
  - Closes a file previously opened for unmanaged logging.

Chapter 5 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 re-distribution agreements do not need to use the Permission Service.

5.1 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 wil shut down the induvidual in favor of the utility, which may not be the desired result.

5.1.1 Authenticating with a FactSet Workstation

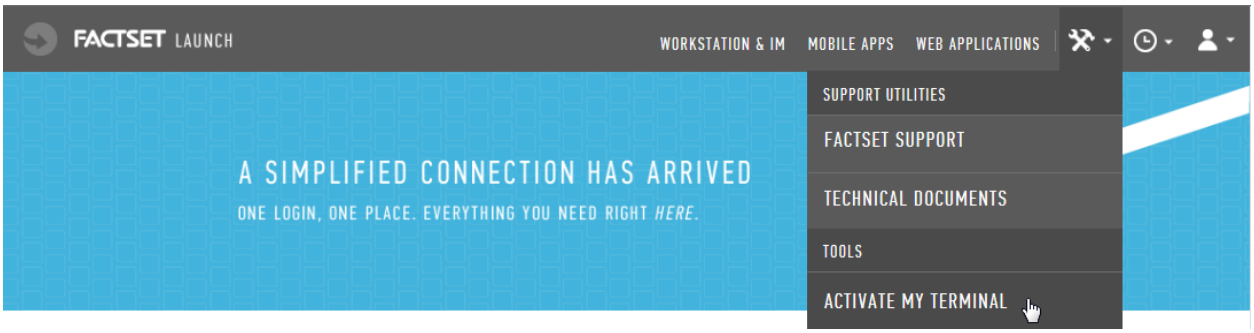
The user can only receive data while being logged into the FactSet workstation on the same machine as where the third- party terminal software is running, this will be confirmed by an IP address check and logon status check.

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.

5.1.2 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.

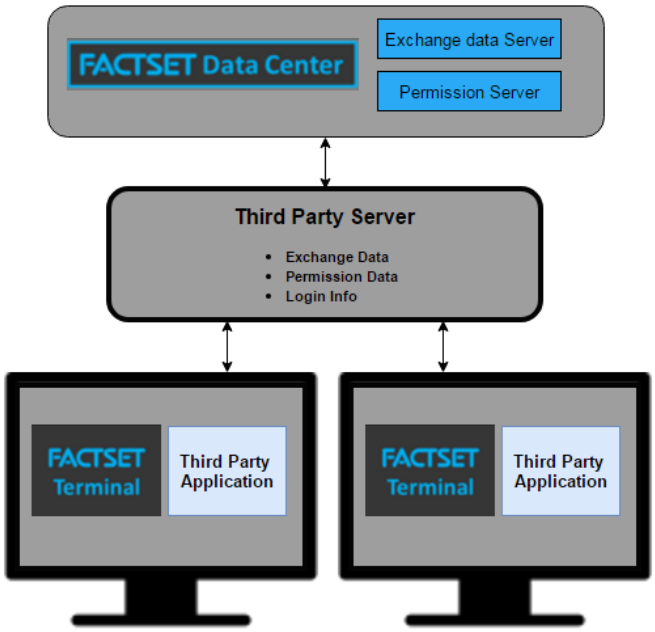
This will be confirmed by an IP address check. Once authenticated access with be granted for 12 hours. After 12 hours the user needs to renew its access from launch.factset.com.

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.

5.2 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 :

1. The users's permission to 12345 is removed
2. FactSet provides notification of user's change in permission code to third party server
3. 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 will be notified and the new logic should be applied.

5.3 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.

5.4 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 2 fields. FID 9221 (USER_LOGIN_STATUS) will return a 1 or 0. A value of 1 signifies that the client is logged in currently, and the IP addresses match. A value of 0 signifies that the user is logged off 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 property:
string RTRequest.Options;

```
        static void Main(string[] args)
    {
      RTConsumer consumer = new RTConsumer();
      consumer.ConnInfo = RTConsumer.MakeConnInfo("<Server Account User-Serial>", "<password>", "api-
stage.df.factset.com");
      consumer.Connect();
      Console.WriteLine($"Connected to server");
      RTRequest permreq = new RTRequest("FDS_PERM", "<Individual User-Serial>");
      permreq.Options = "<Individual User's IP address list>";
      RTConsumer.RTSubscription subs = consumer.MakeRequest(permreq, (RTConsumer.RTSubscription sub,
RTMessage msg) =>
      {
        if(msg.ContainsField(RTFieldId.USER_PERMISSIONS))
        {
          // USER_PERMISSIONS is a repeated field, it will repeat in the message and needs to be iterated to get the
complete list
          var i = msg.GetEnumerator();
          string permission = "";
          foreach( var p in msg)
          {
            permission = permission + "," + p.Value;
          }
          Console.WriteLine($"Perm field : {permission}");
        }
        else
        {
          Console.WriteLine($"Message : {msg}");
        }
      });

      Console.WriteLine("Starting to dispatch");
      while(true)
      {
        consumer.Dispatch(1000);
      }
    }
```

Chapter 6 Options Greeks Calculation

The inclusion of the Greek field calculations in an option record is discretionary. By default, the fields are not included. Please see section 6.4 for the details on how to request these fields.
FactSet provides additional fields that return Greeks values and Implied Volatilities for Streaming DataFeed users.

6.1 Requirements

The Options Greeks Calculations require version 1.3.0.1 or higher of the Exchange DataFeed C# .NET toolkit. Any applications that plan to use version 1.3.0.1 (or higher) of the latest toolkit will need to recompile. 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

Field Id	Name	Type	Description
2613	ANALYTIC_PRICE_RULE	Integer	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 the Mid price will be used and after-market hours the settlement price will be used.
2614	EXPIRATION_DAYS_TO	Integer	The number of business days until the option expires
2620	DELTA	Decimal	The rate of change of option value with respect to changes in the underlying asset's price
2621	GAMMA	Decimal	The rate of change in the delta with respect to the changes in the underlying asset’s price
2622	VEGA	Decimal	The sensitivity of the value of the option to the volatility of the underlying asset
2623	THETA	Decimal	The sensitivity of the value of the option to the passage of time
2624	RHO	Decimal	The sensitivity of the value of the option to the risk-free interest rate
2630	IMP_VOL	Decimal	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
2631	IMP_VOL_ASK	Decimal	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

Field Id	Name	Type	Description
2632	IMP_VOL_BID	Decimal	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
2633	IMP_VOL_CALC_RATE	Decimal	The calculated value of the interest rate using the option pricing model
2634	THEO_VALUE	Decimal	The calculated value of the option using the option pricing model

Please note that all fields except ANALYTIC_PRICE_RULE and EXPIRATION_DAYS_TO will be blank in the 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.

6.2.1 Sample Data

```
  response key: IBM#A1814C195000-USA, 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. Include the OptionsGreeks class for the Greeks Calculations.

using OptionsGreeks;

2. Enable Greek Calculation after the connection is made before the option request is sent

(your RTConsumer object).OptionsGreeksEnabled = true;

6.5 Accessing Greeks

The RTMessage will be updated with the additional Greeks Fields

```
        using OptionsGreeks;
        consumer.OptionsGreeksEnabled = true;
        RTRequest req = new RTRequest(svcName_, option.Key, false);

    decimal? val = msg.Get<decimal>(RTFieldId.DELTA);
    if (val.HasValue)
    {
      Delta = val.Value;
        }
    val = msg.Get<decimal>(RTFieldId.GAMMA);
    if (val.HasValue)
    {
      Gamma = val.Value;
        }

    val = msg.Get<decimal>(RTFieldId.VEGA);
    if (val.HasValue)
    {
      Vega = val.Value;
        }

    val = msg.Get<decimal>(RTFieldId.THETA);
    if (val.HasValue)
    {
      Theta = val.Value;
    }

    val = msg.Get<decimal>(RTFieldId.RHO);
    if (val.HasValue)
    {
      Rho = val.Value;
        }

    val = msg.Get<decimal>(RTFieldId.IMPL_VOL);
    if (val.HasValue)
    {
      ImplVol = val.Value;
        }

    val = msg.Get<decimal>(RTFieldId.IMPL_VOL_ASK);
    if (val.HasValue)
    {
      ImplVolAsk = val.Value;
        }

    val = msg.Get<decimal>(RTFieldId.IMPL_VOL_BID);
    if (val.HasValue)
    {
      ImplVolBid = val.Value;
    }

    val = msg.Get<decimal>(RTFieldId.IMPL_VOL_CALC_RATE);
    if (val.HasValue)
    {
      ImplVolCalcRate = val.Value;
    }
```

```
val = msg.Get<decimal>(RTFieldId.THEO_VALUE);
if (val.HasValue)
{
  TheoValue = val.Value;
}
```

Chapter 7 Level 2 Data

FactSet provides market depth in the Exchange DataFeed for Enterprise Streaming DataFeed users. The additional bid and ask information may be called Level 2, market depth, or order book data, depending on the exchange. In this document market depth is referred to as Level 2 data.

7.1 Requirements

Level 2 functionality in the .NET toolkit requires version 2.0 or higher of the Exchange DataFeed .NET Toolkit. Any applications that are updated to use version 2.0 of the latest toolkit will need to be recompiled. Any applications that are going to use level 2 functionality will require a code change and to recompile.

7.2 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 an additional field indicating the message's sorted position (BID_INDEX_1 and ASK_INDEX_1). To enable Sorted Data, the Level 2 feature must be enabled after the connection is made before the Level 2 request is sent.

```
(your RTConsumer object).Level2Enabled = true;
```

This feature will allow consumers to create a display that has the bids/ask in price order.

7.3 Level 2 Fields

In addition to the currently supported data fields, the following table described the new fields added for the Level 2 content set.

Field Id	Name	Type	Description
150	BID_INDEX_1	Integer	Sorted Data only: The message's position in a sorted list of bids.
250	ASK_INDEX_1	Integer	Sorted Data only: The message's position in a sorted list of asks.
520	ORDER_CODE	String	Order Code
521	MM_STAT_BITMASK	String	Shows Open/Closed quotes[13]
522	MARKET_MECHANISM_TYPE	String	Used to show the order type as in market order or limit order
523	MARKET_MAKER_ID	String	Market Maker ID

Not every exchange will populate every new field that has been added. The new fields will be used with the level 1 fields BID_1/ASK_1, BID_VOL_1/ASK_VOL_1, and BID_TIME_1/ASK_TIME_1.

7.4 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. These must be processed properly to avoid stale date in the Level 2 record. 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.

13 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.

7.4.1 Processing a Message Example

The example code below shows one way to process a sorted Level 2 message from a callback. The callback function checks message type and bid/ask data, at which point any processing of that data can be done. In addition, it checks to see if the stream was closed, and if so, it closes the client-side stream by canceling the tag.

The server may close the stream at any time. In addition, error messages (like RT_E_NOT_FOUND) will cause the stream to set the close/end-of-stream indicator.

```
consumer.Level2Enabled = true;
RTRequest req = new RTRequest(svcName_, symbol);
RTConsumer.RTSubscription subscription = consumer.MakeRequest(req, (RTConsumer.RTSubscription sub,
RTMessage msg) =>
        {
                if (msg.GetField(RTFieldId.MSG_TYPE).Value == "D") {
                        // handle delete message
                }
                else {
                        if (msg.ContainsField(RTFieldId.BID_INDEX_1)) {
                                // handle bid data
                        }
                        if (msg.ContainsField(RTFieldId.ASK_INDEX_1)) {
                                // handle ask data
                        }
                }
        }
);
```

See the Level2Quote sample utility included in the C++ toolkit 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 8 Utilities

8.1 Performance Monitoring Utility

This utility is a standalone executable provided with the .NET toolkit that uses the FactSet DataFeed .NET API. The utility can help provide statistical information about how a given application might perform on your given system.

TestClientSync utility command line sample:

TestClientSync -f *A_Ticker_List.txt* -u *user-name* -p *password* --host *datafeed-server* --trace-level 3 --log_msg_age_thresold=20 --**user_time_threshold**=200 --report_queue=100 --**busy_wait**=50

Usage:

`--svc=VALUE`	the FDS service name
`-u, --user=VALUE`	user name
`-p, --pwd=VALUE`	user password
`--host=VALUE`	host name
`--port=VALUE`	port to connect
`-s, --symbol=VALUE`	comma-delimited list of symbols to subscribe
`-f, --file=VALUE`	file name to load symbols
`-v, --verbose`	to print incoming messages
`-l, --log=VALUE`	log file name
`-w, --workstation`	connect to FactSet Workstation
`-t, --trace-level=VALUE`	diagnostics trace messages level
`--busy_wait=VALUE`	microseconds to simulate a busy work in the message handler function
`--log_msg_age_threshold=VALUE`	average message age threshold to log the message queue statistics, microseconds
`--user_time_threshold=VALUE`	warning threshold for time spend in inside user handler, microseconds
`--report_queue=VALUE`	number of received messages to check/print current message queue size,average message age and total # of messages
`--fields=VALUE`	optional full path to rt_fields.xml file
`-h, -?, --help`	show this message and exit

Appendix A: Error values

The ErrorCodeEnumeration:

Error Number	Code	Description
{Any positive number}	A system error	This is the platform-specific error (via GetLastError() or errno).
-51	Unknown	Unknown/Serious error.
-52	NoServ	The service is not available for requests.
-53	NotFound	A resource or key was not found.
-54	Rename	The stream has been renamed. You should close the current stream.
-55	TimedOut	The request for a resource or a key has timed out. You can retry the operation.
-56	Exists	The resource already exists.
-57	Limit	An application-level threshold has been reached.
-58	Protocol	There is an error on the byte stream during deserialization.
-59	Inval	Either the operation is not supported or an argument is invalid.
-60	Resource	A system resource is unavailable.
-61	NoConn	The connection to the data server is disconnected.
-62	Version	There is an incompatibility with the library being used and the compiled application.
-63	Shutdown	The application has disconnected the API and is attempting to dispatch messages.
-64	Access	Permission denied. The user does not have access.

Appendix B: Control Messages

The following table shows the possible control messages that can be delivered to the application-defined control callback procedure as ControlType enumeration:

Control Type	Meaning	Additional Information
DISCONNECTED	The TCP Connection to the data server is disconnected.	The error information can be obtained via the get_error() method on the message object. The error will be one of the error values that connect() can return.
CONNECTED	The TCP Connection to the data server is connected.	The current active service names are in the Message. The FID, FIDS::SERVICE_NAME, is used repetitively.
AGGREGATION_ENABLE	The sending rate from the server has been slowed down to allow the inbound message queue to drain	Messages received will be aggregated into buckets until the server detects that the queue size if back to within normal limits. This state is typically entered due to a slow consuming application.
AGGREGATION_DISABLE	The sending rate from the server has returned to normal	The server detected that the slow consuming application has caught up.
SERVICE_ENABLE	New services are now available for requests.	The service names are in the Message. The FID, FIDS::SERVICE_NAME, is used repetitively.
SERVICE_DISCONNECT	Services have become stale. Existing streams will now transition to stale.	The service names are in the Message. The FID, FIDS::SERVICE_NAME, is used repetitively.
SERVICE_DISABLE	The services are no longer accepting any new streams.	The service names are in the Message. The FID, FIDS::SERVICE_NAME, is used repetitively.
TERMINATE	The FactSet data server is requesting the application terminate its connection (i.e. the application MUST call disconnect())	This is typical when the authentication for an asynchronous connection has failed. Applications MUST call disconnect when receiving this message. If they do not, the API will call disconnect on its behalf.

The control event handler has the following prototype:

- void OnConnect(object sender, ConnectCompletedEventArgs e);

The ConnectCompletedEventArgs.IsConnectedboolean property will always indicate the current status of the TCP connection to the data server.

The CtrlType is a ControlType property. The enumeration of the possible control types are listed in the table above.
Additional data fields may be present based on the type of control message. These fields are located in the ConnectCompletedEventArgs.CntrlMsg property.

Appendix C: Connection Strings and URI's

Connection Strings:

Connection strings allow applications to specify host and authentication information as a single string. The syntax is as follows:

[USER][:PASSWORD][@]**HOST**[:PORT][14]

The **HOST** value can be either a host name or dotted decimal (e.g., fdshost, api-stage.df.factset.com or 10.14.1.6).

HOST is a mandatory parameter.
PORT can either be an integer or a service name (e.g., fdsserv or 6681). This is an optional parameter and defaults to 6681 if not specified.

The USER:PASSWORD@ part is optional[15]. USER is the FactSet-supplied username, PASSWORDD is the FactSet-supplied password.

Examples:
client:aaa@fdshost means authenticate using a username of "bryan" and a password of "aaa" to the server "fdshost" on the default port 6681.

client@10.2.4.5:4063 means connect to the host at 10.2.4.5 on port 4063 with a username set to bryan.

Connection URI's

Connection URI's are universal resource identitifiers that allow the API to resolve a connection string and its individual components. If a specific protocol is not given, the URI itself is a connection string.

14 [ ] indicate optional
15 Although the user and password information 2020is optional in the connection string, the API must have a user and password in order to authenticate with the FactSet Data Server.

Global Client Support

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

Acknowledgements

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