FXintegrate ESP Developers Guide

(For API Version 3)

Version 3.0 Revision 001
June 2006

Currenex, Inc.
12 East 41st Street
7th Floor
New York, NY 10017

Table of Contents

1 Introduction

Currenex is an independent, Internet-based, global foreign currency exchange. The Currenex FXtrades service is the first operational online global currency service to link institutional buyers and sellers worldwide. It is a complete trading solution that supports the entire trade cycle from initiation and execution to settlement and reporting.

Currenex ESP offers the ability of Corporate and Managed Funds to trade from streaming rates provided by maker banks or customer orders. It is fundamentally different from exchange models in that all bids and offer are fully attributed to the Maker and any Request for Execution is routed completely back to the Maker. It offers the buy-side effective integration of bank's proprietary systems and the sell side the necessary control of making prices and dealing.

Currenex has developed a successful model for multi-bank FX that has been proven in the foreign exchange spot market. By expanding this model to executable streaming prices, both the sell and buy side will realize cost savings and efficiencies.

To provide customers with the benefit of Straight Through Processing (STP) and automated trading, Currenex has developed a seamless connection and interaction framework that allows members to participate in Currenex FXtrades service platform, based on the FXintegrate integration framework. FXintegrate supports trading via an Application Programming Interface (API), it allows members to access the functionality of the FXtrades service with minimal human intervention.

FXintegrate allows members to leverage their existing treasury and automated trading systems in connecting to FXtrades. Currenex members can interact seamlessly with the FXtrades service from currently deployed applications through FXintegrate client interface. FXintegrate is a simple, secure and reliable interface that allows for tighter integration of automated trading applications with FXtrades and greatly improves the efficiency and accuracy of straight through processing efforts.

1.1 Document Scope

This document presents a detailed specification of the Currenex FXintegrate ESP Application Programming Interface (API). It is intended to provide an overview of the API framework and communications infrastructure and its application within a trading framework. Detailed API specifications and message information is not discussed here.

1.2 Overview

Currenex FXintegrate framework connects members to the Currenex FXtrades service. It allows members to deliver and receive a wide range of transactional and informational messages through an API based on Sun's Java platform. Its main function is to be the bridge that connects the FXtrades service with a members trading or treasury management system. With the aid of FXintegrate, FXtrades becomes a black box that receives and processes trade requests from external applications.

The FXintegrate framework consists of three logical components:

- Client integration adaptor (CLIENT)
- FXintegrate client API

- FXintegrate server process

The client integration adaptor (CLIENT) is an application that must be developed in order to connect a members trading system to FXtrades. This component communicates with the FXtrades service via the FXintegrate client API.

The focus of this programming guide will be on the FXintegrate ESP taker API, which is a set of classes and libraries that allows a member to communicate with FXtrades. The ESP Taker API allows members to place orders on Currenex and also receive currenex market data. A Taker member can accomplish the following:

- Create a communication session with the FXtrades service (connect, authenticate, re-authenticate, disconnect)
- Receive spot FX prices for a range of currency pairs as determined by Currenex and the banks supplying the rates.
- Submit orders to FXtrades for execution. The submitting order price is not dependent on the market prices received from currenex.

The FXintegrate server process translates and forwards trading messages from the ESP API to the FXtrades service. Because its operation is transparent to the user, the details of the FXintegrate server process are beyond the scope of this document.

1.3 Functionality

The ESP trading system presents a set of FX instruments (currency pairs and tenors) that may be traded in an electronically streaming market. Makers provide a set of streaming prices for those instruments that they are willing to execute on within a specified size range.

The dealing system presents the prices according to the user's criteria set up from the available prices provided by banks that have granted credit. The dealing system also supports Prime Brokerage to extend credit availability. The identity of the Maker associated with each bid and offer is provided with the price. An Advertised Amount (advertised maximum for which this price is applicable) is associated with the bid and offer prices.

A user (or system) may select any of the prices to execute (i.e. sell or buy). The user must provide an amount for the instrument to be sold or bought.

In addition to trading off of posted Maker prices, a user can enter a Private Order. Private Orders are not shown to the market, but are held by the system until a designated Maker streams a price that satisfies that order and are then sent to that Maker for Execution or matched against other customer orders.

All of these operations can be performed via the FXintegrate ESP API. The ESP API provides the following to client applications through the FXtrades service:

- Real-time trading and execution of FX spot.
- Support for receiving live executable and indicative prices.

- Support for specification and submission of Private Orders for execution by the system once the conditions of the order are met.
- Processing of multiple concurrent trades subject to the restrictions of the ESP trading model.
- Ability to query status of orders.
- Ability to cancel all active orders through a single message.

1.4 Architecture

1.4.1 Java API

FXintegrate provides a Java-based API for application development. Java's platform independence allows FXintegrate to function in a wide variety of environments and support integration across a range of different systems. In addition it is ideally suited to the development of Internet-enabled applications. Messages passed between FXtrades and the member banks are defined by Currenex and can be accessed through Java objects.

1.4.2 Communication

The FXintegrate service and its underlying communication framework utilize standard communication protocols and are designed to function either over the Internet or via a private circuit. Core communications are TCP/IP based, allowing for maximum flexibility and compatibility with member environments. To minimize the impact on existing network security configurations FXintegrate supports HTTPS tunneling and connections through web proxy servers and firewalls if required.

To enhance message reliability, FXintegrate's communication infrastructure has an automatic reconnect feature. If at any time the connection to FXtrades is lost, the communication layer can detect the connection drop and re-establish connection without user intervention.

The FXintegrate communication layer has been designed specifically to address the needs of high-volume, time-critical FX trading over the Internet. It has significant advantages over less resilient and higher latency protocols such as pure HTTPS.

1.4.3 Security

Currenex makes every effort to ensure that all aspects of trading are secure between the member bank and the FXtrades service. Whether it is securing the communication channel or preserving the integrity of trading messages, Currenex ensures that the data sent between the FXtrades service and members is not compromised in any way. Security falls into various categories, all of which are addressed by the FXintegrate service.

1.4.3.1 Data Encryption and Integrity

The FXintegrate communication layer provides both data encryption and message integrity. FXintegrate uses a 128-bit SSL (Secure Socket Layer) connection to communicate data between the member application and FXtrades. Whether the final communication takes place over the Internet or a dedicated channel, communicating over SSL ensures that the data is protected from eavesdroppers and that the data remains intact as it travels from one party to the other.

1.4.3.2 Client Authentication

Any client application gaining access to FXtrades through the FXintegrate service must login prior to being granted such access.

The FXintegrate service utilizes digital certificates for member authentication. An encrypted challenge is created using the member's public key, which is then decrypted by the member using its corresponding private key. The private key never leaves the member's network and is nearly impossible to reproduce.

2 ESP Fundamental Concepts

2.1 Executable Streaming Prices (ESP)

ESP enables market participants (MP's), such as corporations, managed funds, etc., to trade on real-time non-indicative streaming foreign exchange (FX) rates provided by market makers (MM's) or other client order prices. MP's can also submit orders to be matched against these rates by the Currenex matching engine.

2.1.1 Indicative prices

The market data update (MarketDataEntry) can be a tradable price or an indicative price. If the price is indicative, then the amount shown is only for indicative purposes. It cannot be used for trading or order matching. See javadocs for MarketDataEntry for more info.

2.2 Orders

MP's can submit orders for matching. Various different order types are supported. Following sections describe the order types, expiry types and other characteristics surrounding orders.

2.2.1 Order Action Types

OrderActionType	Description
New	To place a new order into Currenex FxTrades system.
Replace	Update or amend an existing active order.
Cancel	Cancel an active order.

2.2.2 Order Types

Order Type	Description
Market	Market orders are executed at the best available price. The order will continue to be filled until all the quantity is executed or cancelled.
Limit	Order executed when a specific price is met.
Stop Loss	Oorder that becomes a market order when a specific price level is reached or surpassed. E.g., a stop-loss order to buy becomes a market order in the system when the market rate is at or above the stop price, while a stop-loss order to sell becomes a market order in the system when the market rate is at or below the stop price.
Iceberg	Order type that's functionally the same as a limit order with the exception that the amount displayed to others viewing the order is only a fraction of the actual order amount.

2.2.3 Expiry Types

An expiry type can be associated with an order to indicate when the order should be removed for matching against other eligible orders. All times should be in Greenwich Mean Time (GMT).

Expiry Condition	Description
Good Till Cancel (GTC)	Orders with this expiry setting remain open and active until either executed or explicitly canceled by the client.
Immediate or Cancel (IOC)	The order is compared against the entire Currenex book. If no match is found, the order or remaining portions not immediately filled are canceled. Similar to the Fill or Kill (FOK) expiry type, except partial fills are possible.
Good Till Date/Time	The submitting client explicitly specifies the date and time at which an order is to be expired if not already executed. If only the date is specified, the system defaults the time to be 23:59:59 GMT.
Day	Orders with this expiry

Expiry Condition	Description
	type that have not been executed will be expired by the system at the end of the Currenex system day on which they were entered. All orders are assumed to be day orders unless specified using setExpiration.
Fill or Kill	The order is compared against the entire Currenex book. If no match is found or if the order cannot be executed in its entirety, it is canceled. Similar to the IOC expiry type except partials fills are not allowed.
Good For Seconds	Allows the placing of an order that is valid for a specific number of seconds after the time it is received by Currenex. Once the specified seconds have passed, if the order has not been executed, it is automatically expired by Currenex.

See Expiry Type creation section 5.6.4 to se how to create different Expiration types.

2.3 Partial Fills

Except for orders with the Fill or Kill (FOK) expiry type, all orders entered into the system have the partial fill flag enabled. The minimum fill amount on any fill can be specified using setMinimumExecutionAmount. If the client does not specify a minimum execution amount, then it defaults to Currenex system minimum trade size.

2.4 Allocations

A MP can specify the trade split information through an array of Allocation objects. Each Allocation object represents the information for one allocation of a particular trade. The allocationId is different from the actual trade the allocation is designated for.

Note: In the current version, allocations are supported only for the orders with expiration type "Fill or Kill (FOK)".

2.5 Pending Actions

A MP can specify the future action that will take affect on the trade after the order gets executed. Pending actions can be specified using the EnumPendingAction enumeration type. See javadoc for the supported actions.

2.6 Value Date (Check)

When submitting a new order, a MP can specify the settlement or value date of the transaction. By default, the value date is SPOT. In the current version only SPOT value date is supported. Currenex FxTrades service will reject orders with invalid value date. Value date can be specified using setValueDate OrderActionBuilder class. Value date can be created using FxDate class. See javadoc for more info.

Currenex will send value date or the settlement date on each fill execution result in the CxOrderExecution class.

2.7 Minimum Execution Amount

Minimum execution amount is a client specified value. Except for orders with the Fill or Kill (FOK) expiry type, a MP can specify the minimum fill amount on each execution. This can be set using setMinimumExecutionAmount in the OrderActionBuilder class.

Note: Outstanding amounts that fall below the specified minimum execution amount will not be filled. The last fill execution will contain openAmountBelowMin set to true follow by the OrderExecutionResult with type CANCELLED.

2.8 Currenex Market Minimum

The Currenex trading service maintains a market minimum. Orders entered for amounts less than this market minimum shall be rejected.

Note: Outstanding amounts resulting from partial fills that fall below the market minimum will not be filled, and must be directly canceled by the client to be removed from the system.

Currenex treats orders with residuals amounts for less than the equivalent of one (1) unit in the base currency, i.e., tag 151 LeavesQty < than 1 base currency unit, as completely filled. Attempts to cancel these orders will be rejected, since they are no longer active. Clients will be notified before any change in the market minimum is put into effect.

2.9 Trading Through a Third Party (Order Routing)

New Orders entered via an API client connection established by a third party and not the end client under which the orders will be booked must indicate the actual client user id and accountId in the New OrderAction message.

The New OrderAction message's AccountID and ClientUserID fields are used to identify the booking account and user.

- AccountID - represents a trading account or fund within an entity.
- ClientUserID - represents a booking user under the same entity as the trading account.

These fields can be set so that a client with multiple trading accounts can trade on behalf of these entities via a single API connection.

There are four possible AccountID and ClientUserID field value combinations, each described below. Booking user is the id used to enter a deal in the Currenex book.

1. Both ClientUserID and AccountID fields are set

   Booking user = ClientUserID
   Booking account = AccountID (A valid Account under the booking user entity)
   The API user session must have permission to book under ClientUserID.

2. ClientUserID is present; AccountId is not present

   Booking user = ClientUserID
   Booking account = default Account (sub fund) of the booking user
   The API user session must have permission to book under ClientUserID.

3. ClientUserID is not present; AccountId is present

   Booking user = The API user
   Booking account = AccountId
   Booking user entity must have a sub fund = AccountId.

4. Both the ClientUserID and AccountId fields are not present

   Booking user = The API user
   Booking account = the default Account (sub fund) of the booking user's entity

The AccountId and ClientUserID fields on cancel, replace, and order status request messages must match those used in the original new order message.

2.10 BrokerMatchId List

When submitting new orders, the customer can specify the BrokerMatchId list where by the order will be matched or executed against the specified BrokerMatchId list. The BrokerMatchId value can be either a direct execution bank id (in the case of direct bank relationship) or a hub. The BrokerMatchId list can be specified when sending new order request as an array of String. If null, then the order will be matched against all the relationships configured for the customer.

Note: The broker match id list should be specified only during new order action.

2.11 ExecBrokerId

When a trade is consummated from the order, each CxOrderExecution will contain the ExecBrokerId to convey the execution broker id value. This is applicable only for CxOrderExecution's sent as part of order fill reports. The ExecBrokerId will be null in the CxOrderExecution for a timed out trade, OrderStatusRequest and GetAllActiveOrders response messages.

The ExecBrokerId field represents the name of the counterparty of the trade.

2.12 PrimeBrokerId

If a trade is consummated from the order and the trade is given up to a prime broker (if configured to do so), then each CxOrderExecution will contain the PrimeBrokerId. This is applicable only in the CxOrderExecution's sent as part of order fill reports. The PrimeBrokerId will be null in the CxOrderExecution for a timed out trade, OrderStatusRequest and GetAllActiveOrders response messages.

3 Workflow

This section illustrates some of the workflows associated with particular usage scenarios.

3.1 Executable Streaming Prices (ESP)

A MP makes a request for a Currenex rate stream by sending a Market Data Request message. Requests for repeated groups of instruments are not currently supported. A new Market Data Request message has to be sent for each currency pair or instrument.

The market data updates are based on the subscription request data.

MDSubscriptionType:

SUBSCRIBE – Snapshot and updates: a subscription message to receive Market Data updates until it is either unsubscribed or the session is disconnected.

UNSUBSCRIBE – Disable request: a unsubscribe request type that stops all market data feeds for the specified instrument.

`MDBookType`

AGGREGATED – All like prices of a currency pair are grouped together so that total amounts are shown.

ATTRIBUTED – non-aggregated type. **Not supported in this release.**

`MDUpdateType`

TOP_OF_BOOK – to receive only the best price. Supports only aggregated book type. See MDBookType.

FULL_BOOK – to receive the full depth of the Currenex book. Supports attributed and aggregated types. See MDBookType.

`MDEntryType`

TWO_WAY – Request for two-way prices.

BID – Request only bid side

OFFER – Request only offer side

Note: - On each instrument only one **`MDEntryType`** type is supported. It is not possible to individual subscription for BID and Offer. If both BID and OFFER updates are needed, then TWO_WAY type should be used.

Currenex will continue to stream rates until either the stream is canceled by another market data request with MDSubscriptionType.UNSUBSCRIBE or the session is closed.

Only aggregated book (MDBookType.AGGREGATED) is supported in this release. Request for attributed (MDBookType.ATTRIBUTED) will be rejected.

3.1.1 Rate updates and cancels

If the MDSubscriptionType is SUBSCRIBE, Currenex continuously sends new rate updates to the client and reports when a rate is no longer available in the form of MarketDataUpdate messages. The MarketDataUpdate messages are notified in the form of a callback with the message id ExternalMsgId.MD_UPDATE_MSG. Each MarketDataUpdate contains the subscription request id and an array of MarketDataEntry objects. MDUpdateType in each MarketDataEntrty object is used to check the update type (New, Change or Cancel).

In the following sections, the rate workflow for each possible request type is described.

Aggregated FullBook

A complete aggregated book is sent to the client.

Rate cancels from the server can appear as cancels of a particular rate or as updates to an aggregate if other prices exist in the same price tier.

Rate updates from the server effectively cancel or update the old price by replacing it with the new price.

Attributed FullBook

A complete non-aggregated book is sent to the client.

Rate cancels from the server can appear as cancels of a particular rate.

Rate updates from the server effectively cancel or update the old price by replacing it with the new price.

Aggregated TopOfBook

Only the best prices for each currency pair, with all prices and amounts aggregated in the best price tier, are sent to the client.

Cancels are only delivered if the best price is affected in the particular currency pair. This could mean a new best price, if the previous best price was cancelled, or a new aggregate size, if the cancel removed one of many best prices.

Updates are only delivered to the client if the best price for a particular currency pair is affected. This could be a new best price or a new aggregate size.

3.1.2 Handling Market Data Entry id

With full book updates, more than one rate can be active for a particular currency pair. The MP is responsible for monitoring the MarketDataEntry ID field to keep track of these updates and manage the price book accordingly.

Each MarketDataEntry contains MDUpdateAction that can be used to distinguish between new, changes and delete. `MDUpdateAction.NEW,` new rate messages include the field `id`, which is unique for all new rate updates within an active session. When subsequent updates arrive with a `MDUpdateAction.CHANGE,` change, or MDUPdateAction.DELETE, delete, the `ID` of a previously streamed rate is referenced. In a full book scenario, the MP uses this id to determine the rates to remove from the book.

3.1.3 Re-subscriptions

Rate subscriptions are session based and are not permanent. A MP must re-subscribe to the currencies it is interested in receiving on each new connection. In cases of an abnormal shutdown or network caused disconnects, a MP will need to re-subscribe to the desired currency pairs.

3.2 Orders

3.2.1 Order Entry

A MP can submit an order to Currenex using OrderAction message with type OrderActionType.NEW. OrderActionBuilder helper class can be used to build the new OrderAction message. Assuming the client is successfully connected with a valid session, the client receives a message in response:

- ORDER_EXECUTION_RESULT – Confirmation
- ORDER_ACTION_ERROR – Reject if the data is incorrect

3.2.2 Order Replace

It is possible to update or replace an unfilled outstanding order without first having to cancel it.

Only unfilled open orders can be modified. Order replace requests on filled or partially filled orders will be rejected. The remaining amount on a partially filled order must be canceled and a new order entered to effect any change.

Order parameters such as price, quantity can be amended on an outstanding order without having to cancel and resubmit the order. For order submitted with expiry type GOOD_TILL_DATE, expiry setting also can be amended. All other parameters should be same as the original outstanding order.

A MP can replace an order by sending OrderAction message with type OrderActionType.REPLACE. OrderActionBuilder helper class can be used to build the replace OrderAction message.

Assuming the client is successfully connected with a valid session and the order is still active, the client receives a message in response:

- ORDER_EXECUTION_RESULT – Replace Confirmation
- ORDER_CANCEL_REJECT – If the order is already filled or not active
- ORDER_ACTION_ERROR – Reject if the data is incorrect

3.2.3 Order Cancellation

A MP can cancel an outstanding order. Only the outstanding amount can be canceled on an order that has been partially filled. Cancel requests can be rejected by the system if the order is currently being processed or not active.

A MP can cancel an order by sending OrderAction message with type OrderActionType.CANCEL. OrderActionBuilder helper class can be used to build the cancel OrderAction message.

Assuming the client is successfully connected with a valid session and the order is still active, the client receives a message in response:

- ORDER_EXECUTION_RESULT – Cancel Confirmation
- ORDER_CANCEL_REJECT – If the order is already filled or not active
- ORDER_ACTION_ERROR – Reject if the data is incorrect

3.2.4 Cancel all outstanding or active orders.

A MP can cancel all outstanding or active orders by sending CancelActiveUserOrdersReqeust message to currenex.

Assuming the client is successfully connected with a valid session and there are one or more active orders, the client receives OrderExecutionResult message for each active order that got cancelled. If there are no active orders to be cancelled, then no response is returned.

- ORDER_EXECUTION_RESULT – Cancel Confirmation

3.3 Order execution updates

Currenex sends OrderExecutionResult (MsgId= ORDER_EXECUTION_RESULT) messages to notify the change in the status of the order.

- Confirm the receipt of an accepted order
- Relay order fill information on active orders
- Relay order expiry or cancel information on active orders after the expiry time.

In a normal workflow, after sending an OrderExecutionResult message to indicate the receipt of the order acceptance, Currenex will continue to send one or more OrderExecutionResult messages to the MP to relay order fill information. If the order is completely filled, it will be indicated in the Execution Report with the type ExecutionResult.FILLLED. In cases of partial fills, Currenex will send OrderExecutionResult messages indicating partial fills (ExecutionResult.PARTIAL_FILL) until the order is completely filled, cancelled or expired.

Client can cancel the remaining portion of the active order at any time, or the remaining portion expires after the specified expiration time.

3.4 Order Status Check

A MP can check or query the status of an individual order or get all active orders.

3.4.1 Single order status

A MP can query for a state of an order by sending SingleOrderStatusRequest. The MP should send valid unique client order id and the Currenex confirmed draft id. MP can take necessary action based on the status.

Assuming the client is successfully connected with a valid session, the client receives a message in response for OrderStatusRequest message:

- SINGLE_ORDER_STATUS_RESPONSE – Status response
- STATUS_REQUEST_ERROR – Error if fail to process the status request.

3.4.2 Active orders list

A MP can get the list of all active or confirmed orders from Currenex by sending ActiveOrdersStatusRequest message.

Assuming the client is successfully connected with a valid session, the client receives a message in response for ActiveOrdersStatusRequest message:

- ACTIVE_ORDERS_STATUS_RESPONSE – Contains array of CxOrderExecutions for each confirmed or working order.
- STATUS_REQUEST_ERROR – Error if fail to process the status request.

3.5 Order Execution/Trade Timeout

If the outstanding order price match with a MM price based on the order type, Currenex sends the MM that streamed the rate an execution request. Currenex expects to receive an acceptance or rejection message within a specified number of seconds, usually 10. If no response is received within this time, the order is placed in an exception state. Currenex notifies the MP.

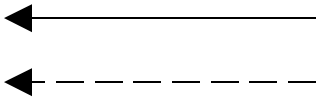
A CXOrderExecution in the OrderExecutionResult message has three amount related fields:

- Amount – the amount for which the order is placed.
- OpenAmount – the amount left over, if any, after a fill. Most meaningful with partial fills.
- CumDealtAmount - the total amount filled on the order including the latest fill.

If the MM does not respond within the configured time, e.g., 10 seconds, Currenex will NOT mark the fill as executed. However, it will continue to decrement the OpenAmount and increment the CumDealtAmount fields. The Execution result will have the ExecutionResultType as “UNKNOWN. If an exception occurs on the last partial fill needed to complete an order, the parent order is closed. Clients can contact Currenex customer support to verify the status of the trade and the order.

4 Message Workflow

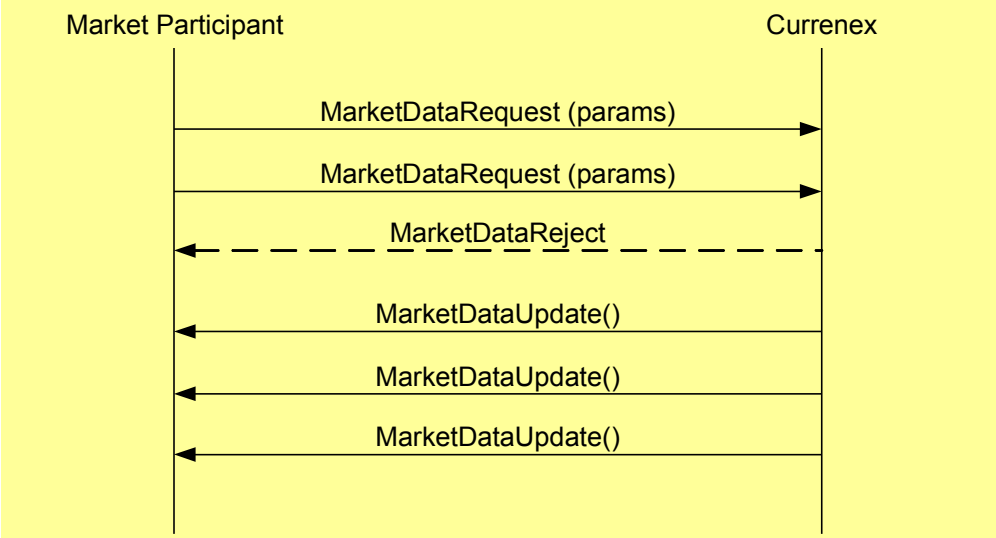
The following symbols are used in the below workflow:



Required Message flow and direction

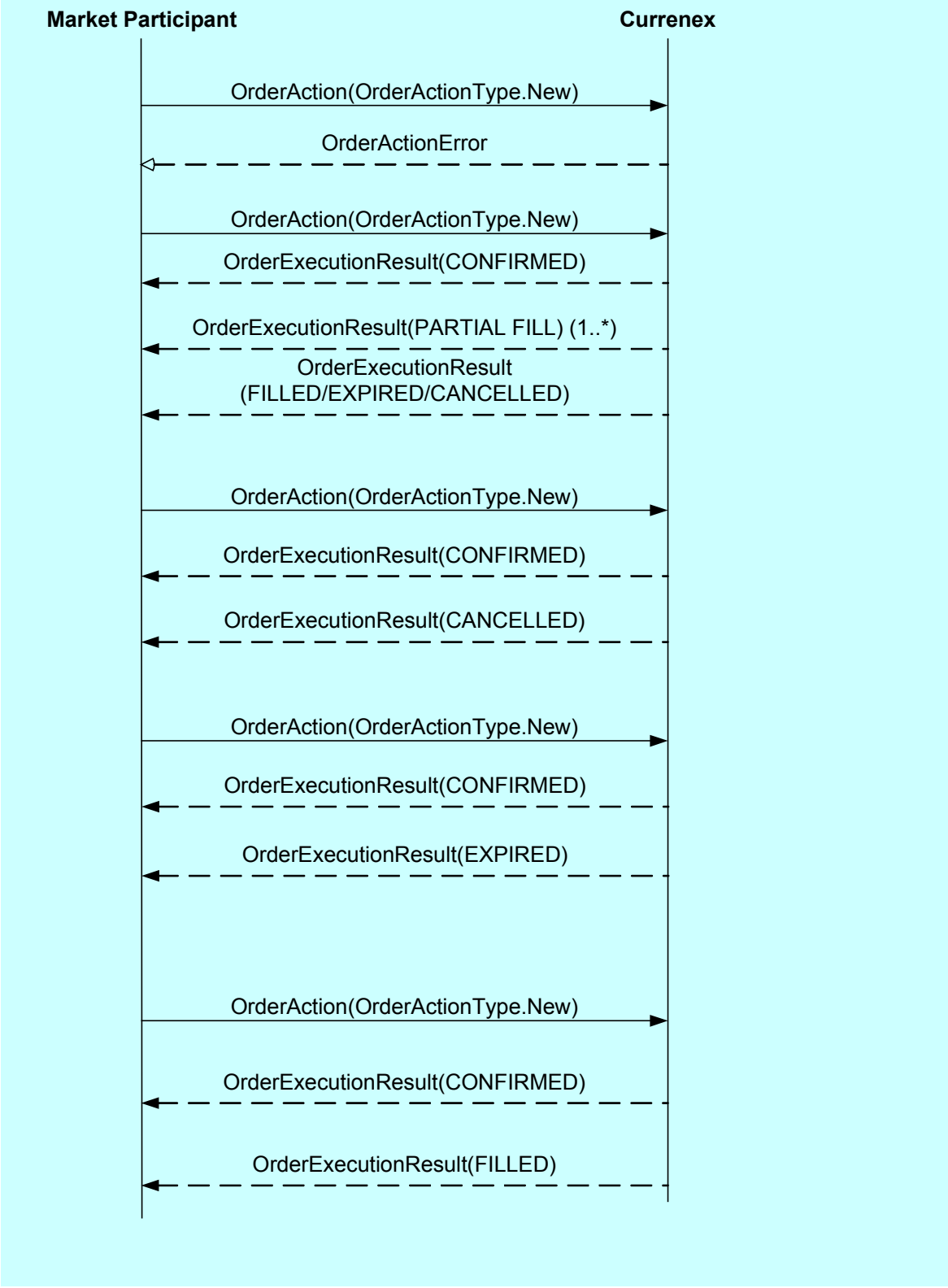
Optional or conditional message

4.1 ESP/Market Data Workflow

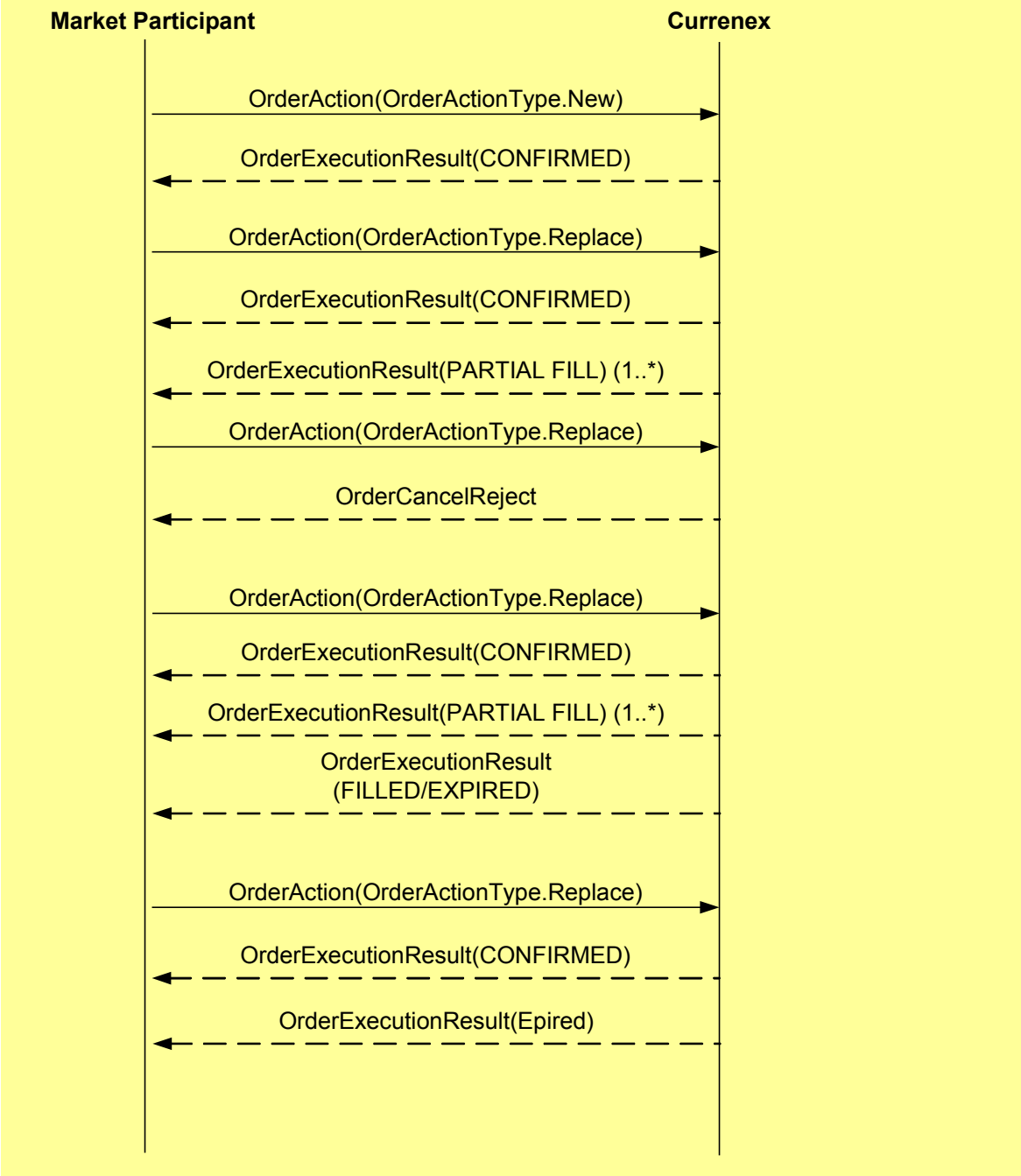


4.2 Order Workflow

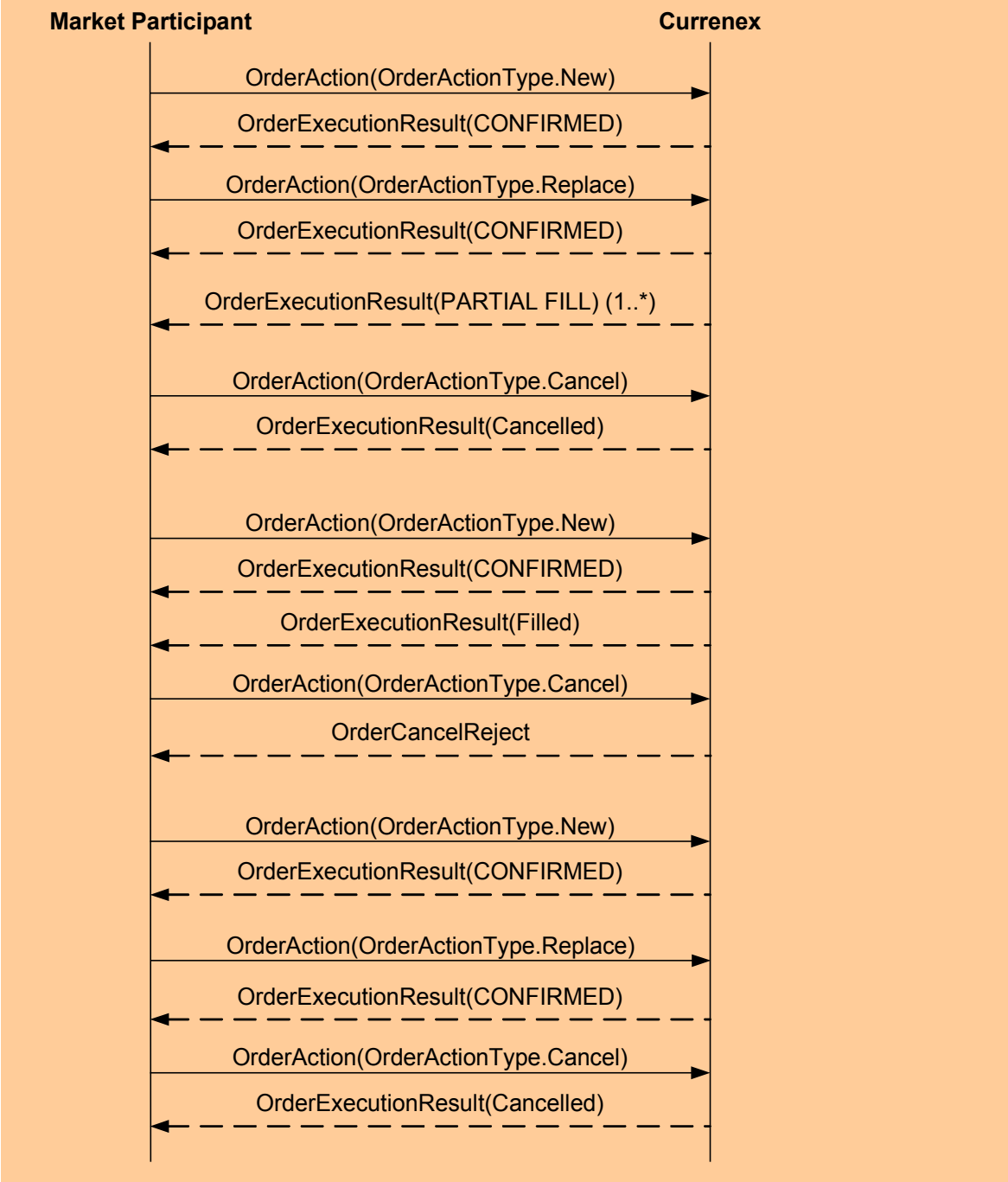
4.3 New order action workflow



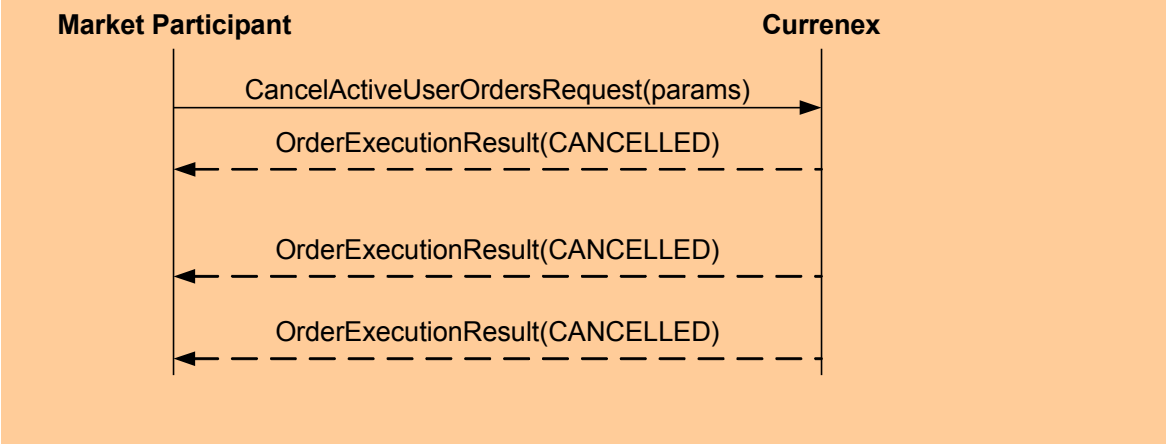
4.4 Replace order action workflow



4.5 Cancel order action workflow

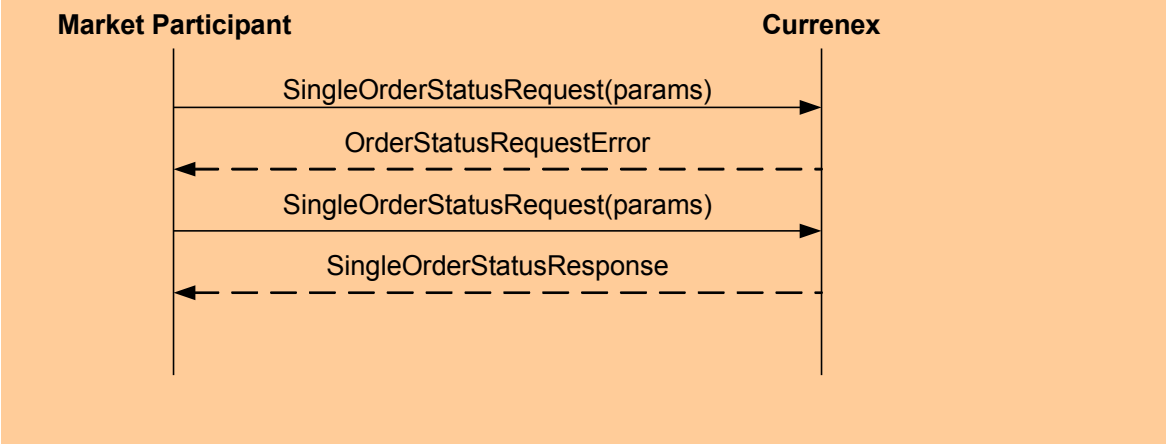


4.6 CancelActiveUserOrdersRequest workflow

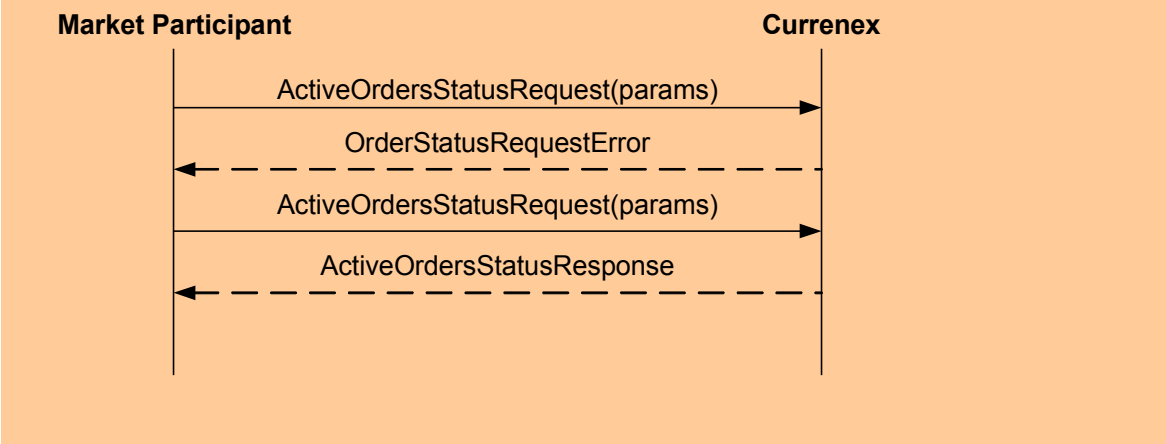


Note: - If there are zero active orders to be cancelled, no response message is sent.

4.7 SingleOrderStatusRequest workflow



4.8 ActiveOrdersStatuRequest workflow



5 Using the FXintegrate ESP API

This section gives an overview of how to write an ESP API client application. To learn more about the API in detail, see the javadoc documentation provided. See the example code provided in the distribution for more information on the API usage.

5.1 Creating the Client Class

The first step in creating a communication link between the client and the FXintegrate service is creating a valid session context and obtaining a message manager by which to send and receive messages. This section outlines the steps required to creating and authenticating to a session context. A single connection/session is sufficient for both market data and orders. But, it's strongly advised to use two different connections for better performance.

```
/**
 * Currenex ESP API Client implementation.
 */
Public class EspApiClient implements IMessageClientBlue {
// create a properties list for the SessionManager
// create custom properties
Properties p = new Properties();
p.setProperty(Constants.KEY_HOST, "integration.currenex.com");
p.setProperty(Constants.KEY_PORT, "443");
p.setProperty(Constants.KEY_TRANSPORT_TYPE,
Constants.VAL_TRANSPORT_TCP);

// get a session context
CxSessionContext ctx = CxSessionFactory.getCxSessionContext(p);

// begin authentication
// the connect call requires a message client and login id,
// see section 3 for CxSessionContext class details.
String encryptedChallenge = ctx.connect(this, loginId);

// create a RsaDecryptJce object and initialize it with
// decryption key information (see Appendix B)
RsaDecryptJce decrypt = new RsaDecryptJce();
Decrypt.init(PrivateKey);

// create authentication response message
String decryptedChallenge = new  decrypt.decrypteChallenge(encryptedChallenge);

// authenticate and catch exceptions
try {
        ctx.authenticate(authResp);
} catch (CxException e) { // handle CxException }
```

```
// start trading
try {
ctx.startTrading();
} catch (CxException e) { // handle CxException }

// we are authenticated and ready to send & receive trading messages.
Public void handleCxMessage(CxMsg cxMsg) {
        //handle Currenex message based on message type and id.
}

// Decrypt the challenge and send the value to Currenex to re-authenticate
Public void reauthenticate(String challenge) {
        //1. Descrypt the challenge
        //2. return the decrypted value
}
}
```

At this point, the CIA is authenticated to the FXtrades service and ready to send and receive messages.

The implementation object of IMessageClientBlue is passed to the CxSessionContext during the connect call will be called upon when a new message is sent to the CIA. IMessageClientBlue callback method, handleCxMessage, will be called directly by the underlying FXintegrate communication layer when new messages arrive. Since processing a message could take varying amounts of time, and new messages could arrive when a previous message is being processed, it is recommended that the handling of an incoming message by the CIA take place on a different thread. This way, the call to handleCxMessage can return immediately allowing the underlying communication layer to return to its duty of waiting for incoming messages.

5.2 Application message objects

Currenex provides Java objects for all messages to and from Currenex. All messages are delivered via a wrapper class of the type: **currenex.client.util.CxMsg**

From this object, a currenex.share.message.AMsg object can be retrieved by calling the CxMsg.getPayload() method. All ESP related messages extend the AMsg object. The current release of ESP supports the following **AMsg** subclasses:

- MarketDataRequest(to Currenex)
- MarketDataRequestReject(from Currenex)
- MarketDataUpdate(from Currenex)
- OrderAction(to Currenex)
- SingleOrderStatusRequest(to Currenex)
- ActiveOrdersStatusRequest(to Currenex)
- CancelActiveUserOrdersRequest(to Currenex)
- PublishPrices (from Currenex)

- OrderExecutionResult (from Currenex)
- OrderActionError (from Currenex)
- OrderCancelReject (from Currenex)
- SystemConnectionStatus (from Currenex)
- SystemError (from Currenex)

5.3 Message Sending and Replying

There are two method calls to send messages to Currenex via the CxSessionContext object. They are:

- CxSessionContext.send(AMsg aMsg)
- CxSessionContext.reply(AMsg replyMsg, CxMsg originalMsg)

5.4 Creating and Sending MarketData Actions

5.4.1 Market Data Subscribe request

```
AMsg mdSubscriptionReq = new MarketDataRequest(
        "EUR/USD-TWO_WAY",
            MDSubscriptionType.SUBSCRIBE,
            "EUR/USD",
            null, //For spot
            MDEntryType.TWO_WAY,
            MDDepthType.TOP_OF_BOOK,
        MDBookType.AGGREGATED);

cxMDSessionContext.send(mdSubscriptionReq);
```

5.4.2 Market Data UnSubscribe request

```
AMsg mdUnSubscriptionReq = new MarketDataRequest(
        "EUR/USD-TWO_WAY",
            MDSubscriptionType.UNSUBSCRIBE,
            "EUR/USD",
            null, //For spot
            MDEntryType.TWO_WAY,
        null,
        null);

cxMDSessionContext.send(mdUnSubscriptionReq);
```

5.5 Creating and Sending Order Actions

5.5.1 Creating OrderAction.New message

```
OrderActionBuilder orderActionBuilder = new
OrderActionBuilder(OrderActionType.NEW);

/* Set the params based on the order action and the order type */

orderActionBuilder.setId(<clientOrderId>);

//See OrderType class javadoc for other order types
orderActionBuilder.setOrderType(OrderType.LIMIT);
//set the currency pair BASECCY/TERMSCCY
orderActionBuilder.setCcyPair("EUR/USD");

//Buy/Sell side – See EnumBuySell javadoc for the constants
orderActionBuilder.setBuySell(EnumBuySell.SELL);

orderActionBuilder.setAmount(new Amount(<orderAmount>));

//Booking details - Optional
orderActionBuilder.setAccountId(<validAccountId>);
orderActionBuilder.setClientUserId(<validClientUserId>);

//Trade Ccy
orderActionBuilder.setTradeCcy("EUR");

//BrokerMatchIds
String[] brokerMatchIds = new String{} {"DemoBank1_Bank", "FxTrades_Hub"};
orderActionBuilder.setBrokerMatchIds(brokerMatchIds);

orderActionBuilder.setLimitRate(new Rate(<limitPrice>));

//Minimum execution fill amount. Null value means fill amount can be
anything.
orderActionBuilder.setMinExecutionAmount(<minimumExecutionAmount>);

//Order expiry type – See Expiration class javadoc for other types.
orderActionBuilder.setExpiration(Expiration.GOOD_TO_CANCEL);

// Get the message to send –throws OrderActionCreationException if the
required data is not set.
AMsg newOrderActionMsg = orderActionBuilder.getMessage();
```

```
//Send the message to currenex
cxSessionContext.send(newOrderActionMsg);
```

5.6 Cancel message

5.6.1 Creating OrderAction.Cancel(General)

```
OrderActionBuilder orderActionBuilder = new
OrderActionBuilder(OrderActionType.CANCEL);

//Unique id for cancel request
orderActionBuilder.setId(<clientOrderId>);

//Original client order id to be cancelled
orderActionBuilder.setOriginalId(originalClientOrderId);

//Confirmation Id from Currenex
orderActionBuilder.setDraftId(cxDraftId);

//CcyPair
orderActionBuilder.setCcyPair(ccyPair);

// Get the message to send - throws OrderActionCreationException if the
required data is not set.
AMsg cancelOrderActionMsg = orderActionBuilder.getMessage();

//Send the message to currenex
cxSessionContext.send(cancelOrderActionMsg);
```

5.6.2 Creating OrderAction.Cancel using CxOrderExecution

OrderActionBuilder class has an utility method “prepareCancel” that can be used to create cancel message from the CxOrderExecution object. This will build the message with the values set from the CxOrderExecution object. See OrderActionBuilder class javadoc for more info.

```
OrderActionBuilder orderActionBuilder = new
OrderActionBuilder.prepareCancel(cxOrderExecution)

/* Set the params based on the order action and the order type */
orderActionBuilder.setId(<clientOrderId>);

//Original client order id to be cancelled – Set by the utility method
//Confirmation Id from Currenex – Set by the utility method
```

```
// Get the message to send - throws OrderActionCreationException if the
required data is not set.
AMsg cancelOrderActionMsg = orderActionBuilder.getMessage();

//Send the message to currenex

cxSessionContext.send(cancelOrderActionMsg);
```

5.7 Replace message

5.7.1 Creating OrderAction.Replace (General)

```
OrderActionBuilder orderActionBuilder = new
OrderActionBuilder(OrderActionType.REPLACE);

/* Set the params based on the order action and the order type */
orderActionBuilder.setId(<clientOrderId>);

//Original client order id to be cancelled

orderActionBuilder.setOriginalId(originalClientOrderId);

//Confirmation Id from Currenex

orderActionBuilder.setDraftId(cxDraftId);

//See OrderType class javadoc for other order types

orderActionBuilder.setOrderType (OrderType.LIMIT);


//set the currency pair BASECCY/TERMSCCY

orderActionBuilder.setCcyPair("EUR/USD");

//Buy/Sell side – See EnumBuySell javadoc for the constants

orderActionBuilder.setBuySell(EnumBuySell.SELL);

orderActionBuilder.setAmount(new Amount(<orderAmount>));

orderActionBuilder.setLimitRate(new Rate(<limitPrice>));

//Minimum execution fill amount. Null value means fill amount can be
anything.

orderActionBuilder.setMinExecutionAmount(<minimumExecutionAmount>);
```

```
// Get the message to send – throws OrderActionCreationException if the
required data is not set.
AMsg replaceOrderActionMsg = orderActionBuilder.getMessage();

//Send the message to currenex

cxSessionContext.send(replaceOrderActionMsg);
```

5.7.2 Creating OrderAction.Replace using CxOrderExecution

OrderActionBuilder class has an utility method “prepareReplace” that can be used to create replace message from the CxOrderExecution object. This will build the message with the values set from the CxOrderExecution object. See OrderActionBuilder class javadoc for more info.

```
//Build the replace message using data in the CxOrderExecution object from
the OrderExecutionResult message received during confirmation.

OrderActionBuilder orderActionBuilder = new
OrderActionBuilder.prepareReplace(cxOrderExecution)

/* Set the params based on the order action and the order type */


orderActionBuilder.setId(<clientOrderId>);


//Original client order id to be cancelled – Set by the utility method

//Confirmation Id from Currenex – Set by the utility method

//See OrderType class javadoc for other order types – Set by the utility
method


//Buy/Sell side – See EnumBuySell javadoc for the constants  – Set by the
utility method


orderActionBuilder.setAmount(new Amount(<newOrderAmount>));


//Currency amount side – See EnumBaseTerms javadoc for the constants  – Set
by the utility method

orderActionBuilder.setLimitRate(new Rate(<newLimitPrice>));


// Get the message to send - throws OrderActionCreationException if the
required data is not set.
AMsg replaceOrderActionMsg = orderActionBuilder.getMessage();

//Send the message to currenex

cxSessionContext.send(replaceOrderActionMsg);
```

5.8 Status request message

5.8.1 SingleOrderStatusRequest message

```
AMsg orderStatusReqMsg = new SingleOrderStatusRequest(
    clientOrderId, /* Client unique order id sent in the order action */
    draftId, /* Currenex confirmation id */
    ccyPair, /* Currency pair */
    clientUserId /* Client user id. It can null if the order status request
is for the same user that placed the orders */
);
//Send the message to currenex
cxSessionContext.send(orderStatusReqMsg);
```

5.8.2 ActiveOrdersStatusRequest message

```
AMsg activeOrdersStatusReq = new ActiveOrdersStatusRequest(
    requested, /* Client unique request id */
    clientUserId /* client user id. Null defaults to the session user id */
    );
//Send the message to currenex
cxSessionContext.send(activeOrdersStatusReq);
```

5.8.3 CancelActiveUserOrdersRequest message

```
AMsg cancelAllActiveOrdersMsg = new CancelActiveUserOrdersRequest(
    requested, /* Client unique request id */
    clientUserId /* client user id. Null defaults to the session user id */
);
//Send the message to currenex
cxSessionContext.send(activeOrdersStatusReq);
```

5.9 Handling messages from Currenex

Message sent from Currenex should be handled by retrieving the message ID from the CxMsg object and comparing the ID to the currenex.share.external.message.ExternalMsgId object. The following is a section of sample code to perform message handling:

```
public void handleCxMessage(CxMsg msg) {
        switch (msg.getMsgId()) {
                case ExternalMsgId.MD_REQUEST_REJECT_MSG:
                        handleMarketDataReject(msg);  break;
                case ExternalMsgId.MD_UPDATE_MSG:
                        handleMarketDataUpdate(msg);          break;
                case ExternalMsgId.ORDER_EXECUTION_RESULT:
                        handleOrderExecutionResultMsg(msg);  break;
                case ExternalMsgId.ORDER_ACTION_ERROR:
                        handleOrderActionErrorMsg(msg); break;
                case ExternalMsgId.ORDER_CANCEL_REJECT:
                        handleOrderCancelRejectMsg(msg); break;
                case ExternalMsgId.SINGLE_ORDER_STATUS_RESPONSE:
                        handleSingleOrderStatusResponsetMsg(msg); break;
                case ExternalMsgId.ACTIVE_ORDERS_STATUS_RESPONSE:
                        handleActiveOrdersStatusResponseMsg(msg); break;
                case ExternalMsgId.SYSTEM_CONNECTION_STATUS:
                        handleSystemConnectionStatusMsg(msg); break;
                case ExternalMsgId.SYSTEM_ERROR:
                        handleSystemErrorMsg(msg); break;
                default: handleUnknownMsg(msg);
        }
}
```

5.10 Response messages

5.10.1 MarketDataRequestReject

Reject response message for a MarketDataRequst message.

```
public synchronized void handleMarketDataReject(CxMsg msg) {
    switch (msg.getPayload().getId()) {
    case ExternalMsgId.MD_REQUEST_REJECT_MSG:
          MarketDataRequestReject marketDataReject =
          (MarketDataRequestReject) msg.getPayload();
          LOG.warn("MD request rejected for "
            + marketDataReject.m_id
            + "' code=" + marketDataReject.m_errorCode
            + ", description=" + marketDataReject.m_errorDescription);
           break;
    }
}
```

5.10.2 MarketDataUpdate

ESP market data update response message.

```
public synchronized void handleMarketDataUpdate(CxMsg msg) {
    switch (msg.getPayload().getId()) {
    case ExternalMsgId.MD_REQUEST_REJECT_MSG:
          MarketDataUpdate marketDataUpdate =
             (MarketDataUpdate) msg.getPayload();

          MarketDataEntry[] mdEntries = marketDataUpdate.m_entries;
          for (int i = 0; i < mdEntries.length; i++) {
            MarketDataEntry mdEntry = mdEntries[i];
            if(mdEntry != null) {
                  LOG.info("mdEntryId = " + mdEntry.getId());
                  LOG.info("indicative = " + mdEntry.getIsIndicative());
                  LOG.info("action = " + mdEntry.getMDUpdateAction());
                  LOG.info("price = " + mdEntry.getInstrPrice());
                  If(mdEntry.getInstrPrice() != null) {
                      LOG.info("all-in-rate = "
```

```
                        + mdEntry.getInstrPrice().getRate());
                    LOG.info("spotRate = "
                        + mdEntry.getInstrPrice().getSpotRate());
                }

                LOG.info("money = " + mdEntry.getMoney());
                LOG.info("amountCcy = "
                        + mdEntry.getMoney().getCurrencyCode());
                LOG.info("amount = "
                        + mdEntry.getMoney().getAmount());
                //null value date defaults to spot.
                LOG.info("valueDate = " + mdEntry.getValueDate());
                LOG.info("sourceName = " + mdEntry.getSourceName());

                //Applicable for aggregated prices. Gives the count for
            the number of like prices.
                LOG.info("orderCount = " + mdEntry.getOrderCount());
            }
        }
        break;
    }
}
```

5.10.3 OrderActionError

Message sent in response to an error in processing the order action message from the client.

```
public synchronized void handleCxMessage(CxMsg msg) {
      switch (msg.getPayload().getId()) {
      case ExternalMsgId.ORDER_ACTION_ERROR:
            OrderActionError  orderActionError = (OrderActionError)
msg.getPayload();
            CxOrderReject orderError = orderActionError.getValue();
            String errorCode = orderError.getCode();
            String errorDescription = orderError.getDescription();
            String orderId = orderError.getClientOrderId();
                handleOrderActionError(orderActionError);
            break;
      }
}
```

5.10.4 OrderCancelReject

Message sent in response to an error in processing the cancel/replace order action message from the client. See javadoc for OrderCancelReject and CxOrderReject for more info.

```
public synchronized void handleCxMessage(CxMsg msg) {
      switch (msg.getPayload().getId()) {
      case ExternalMsgId.ORDER_CANCEL_REJECT:
            OrderCancelReject orderCancelReject = (OrderCancelReject)
msg.getPayload();


            CxOrderReject orderCancelReject = orderCancelReject.getValue();
            String errorCode = orderCancelReject.getCode();
            String errorDescription = orderCancelReject.getDescription();
            String orderId = orderCancelReject.getClientOrderId();
            String draftId = orderCancelReject.getDraftId();
                handleCancelReject(orderCancelReject);

            break;
      }
}
```

5.10.5 OrderExecutionResult

Message sent in response to the change in the order status. See javadoc for OrderExecutionResultMessage, CxOrderExecution and ExecutionResultType for detailed description.

```
public synchronized void handleCxMessage(CxMsg msg) {
      switch (msg.getPayload().getId()) {
            case ExternalMsgId.ORDER_EXECUTION_RESULT:
            OrderExecutionResult execResultMsg = (OrderExecutionResult) msg
                        .getPayload();
            CxOrderExecution orderExecution =
orderExecResultMsg.getExecution();
            handleOrderExecutionResultMsg(orderExecResultMsg);
      }
}
```

5.10.6 SingleOrderStatusResponse

Message sent in response to the SingleOrderStatusRequest.

```
public synchronized void handleCxMessage(CxMsg msg) {
      switch (msg.getPayload().getId()) {
            case ExternalMsgId.SINGLE_ORDER_STATUS_RESPONSE:
            SingleOrderStatusResponse statusResponseMsg =
(SingleOrderStatusResponse) msg.getPayload();
```

```
            CxOrderExecution orderExecution =
statusResponse.getExecution();
            handleSingleOrderStatusResponseMsg(statusResponseMsg);
      } }
```

5.10.7 ActiveOrdersStatusResponse

Message sent in response to the ActiveOrdersStatusRequestt.

```
public synchronized void handleCxMessage(CxMsg msg) {
      switch (msg.getPayload().getId()) {
            case ExternalMsgId.ACTIVE_ORDERS_STATUS_RESPONSE:
            ActiveOrdersStatusResponse statusResponseMsg =
(ActiveOrdersStatusResponse) msg
                        .getPayload();
            CxOrderExecution[] orderExecutions =
statusResponse.getExecutions();
            handleActiveOrdersStatusResponseMsg(statusResponseMsg);
      } }
```

5.11 Creating Expiry Types - Examples

```
/* Fill  or Kill */
Expiration fokExpiry = Expiration.FILL_OR_KILL;

/* Good for few seconds */
Expiration goodForSeconds = new Expiration(5);

/* Good for a date time */
// Expires after 21:30:05 GMT current day
SimpleDateFormat currentDateFmt = new SimpleDateFormat("dd/MM/yyyy");
SimpleDateFormat dateTimeFmt = new SimpleDateFormat("dd/MM/yyyy HH:mm:ss");
String currentDateOnlyStr = currentDateFmt.format(new Date());
Date expDateTime = dateTimeFmt.parse(currentDateOnlyStr+ " 21:30:05");
Expiration goodForDateTime = new Expiration(expDateTime);

/* Good until cancelled */
Expiration gtcExpiry = Expiration.GOOD_TO_CANCEL;

/* Day order */
Expiry dayExpiry = Expiration.DAY;
```

6 Error and Status Events

There are three levels of error messages in the FXintegrate framework: internal, system and trading. These messages are always sent from the server to the client and never in the opposite direction. Trading error messages pertain to error events that occur within the scope of a specific trade in the system, system error messages indicate something wrong at the API level, and internal error messages encompass all other error events. Each of the three types of error messages has their own structure that contains information specific to their purpose.

The server generates error messages to the adaptor that are either triggered by internal events or in response to specific messages. Adaptors should be prepared to handle both types of error message and respond appropriately.

6.1 Order Action Errors

All order action and execution errors are returned via the OrderActionError and OrderCancelReject messages based on the action. The **getValue**() method in the OrderActionError and OrderCancelReject returns CxOrderReject instance. CIA can get the client error code using **getCode**() method in the CxOrderReject class. CIA will get different descriptions for the same error code which provided more detailed information on the error. See the javadoc for more information on the class usage.

All order action & execution errors on the server are grouped into the client below error codes.

Order Execution Group Error Codes	
Code	**Definition**
12000	Missing mandatory fields in the request or the data in the request is invalid. Check the description for more detailed information.
12001	Invalid data in the request. Check the error description in the response message for more specific information.
12002	Invalid permissions
12003	System errors
12004	Order execution errors
12005	Order cancellation errors. This error code can be used to take necessary action as the order cancellation failed on Currenex side.
12006	Order update/replace errors. This error code can be used to take necessary action as the order replace failed on Currenex side. As simple example is by sending a cancellation.
12007	Unexpected error.
12008	Order cancellation failure warning message. No action required by the customer. This is generally sent when the customer trying to cancel an inactive or already filled order. Customers can also do an OrderStatusRequest to make sure the final state of the order is same as what is expected.

Order Execution Group Error Codes	
Code	**Definition**
12009	Order update failure warning message. No action required by the customer. This is generally sent when the customer trying to replace an inactive or already filled order. Customers can also do an OrderStatusRequest to make sure the final state of the order is same as what is expected.

6.2 System Errors

All system errors are returned via the *sysErrorEvent* message. Both internal errors and system errors fall into this category.

Below is a table of internal error codes and their definitions. All internal error codes start with a '1'. The description field of the internal message is very short and not necessarily tailored to the specific error it is reporting. Its purpose is more to provide a quick visual identification of what the error code means instead of providing a detailed account of the exact problem.

FXintegrate Internal Error Codes	
Code	**Definition**
1000	Unknown error.
1001	Internal server error; could have any number of underlying causes.
1002	Unexpected message; this error occurs if the server receives a server-to-client only message.
1003	Error parsing XML message most likely because message does not conform to version of DTD used by the server
1004	Version on message does not match version used by server.
1005	Timestamp on message invalid; it needs to be within an acceptable range.
1006	Message dropped because user has not logged on.
1007	The user specified in the logon request message is unknown to the server.
1008	No certificate found for the user specified in the logon request message.
1009	The TTL of the challenge string has been exhausted and the message is effectively dropped.
1010	The challenge string does not validate; this is a FATAL error and the connection will be dropped due to security concerns.
1011	The logon was rejected by the server.
1012	The session timed out; this occurs if all re-authentication attempts fail and the session timer expires.
1013	The request to start trading was rejected by the server.
1014	The request to stop trading was rejected by the server.

General system error messages identify error conditions that are not purely due to the internals of the server. All system error codes begin with '3'.

FXintegrate System Error Codes	
Code	**Description**
3000	A general internal error occurred.
3001	The physical connection to the FxTrades server has been lost. An application-level re-connect is required to re-establish a connection to the server.
3002	Connection status message. This message is delivered to indicate that the FXtrades service has been rebooted. The state of the connection between the API and FXtrades is disconnected, but this does signify that the connection will return shortly when FXtrades is restarted.

6.2.1 3001 – Connection Lost Status

This message is delivered when the client API's communication layer detected an unrecoverable exception in the connection. At this point, the connection between the adaptor and FXtrades is disconnected. The adaptor has a few options at this point. It could:

1. Report that the connection is disconnected and cease operation.
2. Attempt to connect again by calling the FXintegrate API's proper connection methods.

There is no guarantee that the reconnect process will succeed. The adaptor should have heuristics in determining when to abort the reconnect process.

6.2.2 3002 – Rebooted Status

This message is delivered to indicate that the remote FXtrades service has been rebooted. The state of the connection between the API and FXtrades is disconnected, but this does signify that the connection should return shortly when FXtrades is restarted.

Unlike 3001, which indicates an unrecoverable connection loss, the 3002 messages indicates the connection was lost not due to network related issues, but because the remote server is being rebooted. This is usually a scheduled activity on weekly bases. When this message is received, the adaptor should attempt to reconnect to FXtrades via the FXintegrate API's proper connection methods.

6.3 Trading Errors

Trading errors are fewer in number but can be caused by a greater number of scenarios. The description field for trading errors does provide greater detail and is tailored to the exact situation that caused it. All trading error codes start with a '2'.

FXintegrate Trading Error Codes	
Code	**Description**
2000	A general internal error occurred within the scope of the trade.
2001	Message contains an unknown identifier.
2002	XML signature failed to verify so the message was dropped.

6.4 Connection Status

Connection status messages provide the adaptor with an indication of the state of the underlying communication channel with the server. Because this channel is critical to the business functionality provided by the API, all adaptors should handle these messages and take appropriate action Details on how to handle these errors are provided in this section.

The following table lists the possible connection status codes.

FXintegrate System Connection Status Codes	
Code	**Definition**
3010	Connected indicator message. This message is delivered when the connection between the CIA and FXtrades is re-established. This message is usually delivered after a successful re-connection following a 3030 message.
3020	Disconnected indicator message. This message is delivered when the connection between the CIA and FXtrades is lost and the system could not re-establish the connection. It is usually delivered after an unsuccessful re-connection following a 3030 message.
3030	Connecting indicator message. This message is delivered when the client API's communication layer detected a connection drop between the CIA and FXtrades. This message indicates the API is attempting to re-establish the connection. It can be followed by either the 3010 message or the 3020 message.

6.4.1 3010 – Connected Indicator

This message is delivered when the connection between the CIA and FXtrades is re-established. This message is usually delivered after a successful re-connection following a 3030 message.

This message is also delivered when a connection is initially established with FXtrades or when subsequent connect method calls are made after a connection is dropped.

6.4.2 3020 – Disconnected Indictor

This message is delivered when the connection between the CIA and FXtrades is lost and the system could not re-establish the connection. It is usually delivered after an unsuccessful re-connection following a 3030 message.

This message means the CIA is disconnected from FXtrades. All subsequent trading messages sent via the FXintegrate API, prior to a reconnect, will result in errors. Since this message is only an indication of connection status, it should not be used for absolute

error handling. This message will either be preceded or followed by a 3001 error message (because of the asynchronous nature of the API, exact order cannot be determined).

6.4.3 3030 – Connecting Indicator

This message is delivered when the client API's communication layer detected a connection drop between the CIA and FXtrades. This message indicates the API is attempting to re-establish the connection. This message can be followed by either the 3010 message or the 3020 message.

In the connecting state, the FXintegrate API is queuing all outgoing messages. Once the connection is re-established, the messages are passed to FXtrades. If the connection cannot be re-established, queued message will be discarded.

7 Core Java API

7.1 API Configuration

The FXintegrate client API requires certain configuration information for normal operation. This information can be supplied to the API either through a properties file or by passing it a Java Properties object containing the required information at startup.

If a properties file is used, it must be called fxintegrate.properties. This file must be accessible through the CLASSPATH variable of the operating environment.

A list of all the configurable property names along with their type, description and an example are given in the table below.

fxintegrate.properties			
Property	**Type**	**Example**	**Description**
fxintegrate.host	String	integration.currenex.com	The name of the FXintegrate service host that resides within Currenex.
fxintegrate.port	Integer	443	The port on which the FXintegrate service is listening (typically 443).
fxintegrate.protocol	String	tcp	The protocol used to connect to the FXintegrate service. The tcp protocol is the only one currently supported
fxintegrate.proxyhost	String	bankproxy.bank.com	The name/address of the internal proxy that will allow external connections.
fxintegrate.proxyport	Integer	80	The port for the internal proxy.
fxintegrate.proxyuser	String	ProxyUser	The login name for the proxy, if required.
fxintegrate.proxypassword	String	XXXXXXXX	The password that corresponds to the proxy user, if required.
fxintegrate.userdomain	String	Currenex	The NT domain of the user. Required for MS Proxy only.
fxintegrate.usessl	Boolean	0 or 1	Indicates whether fxintegrate should use ssl to communicate with the host service.
fxintegrate.logging.priority	String	FATAL ERROR WARN INFO DEBUG	Tells FXintegrate Client API the detail of logging it should provide
fxintegrate.logging.location	String	logfile.name	The directory where the FXintegrate log file will be written. The FXintegrate client must have write access to this location.
fxintegrate.logging.file	String	fxintegrate.log	The name of the log file.
fxintegrate.logging.maxfilesize	String	200KB	Size of log file before being rolled. Number represented as bytes unless KB or MB is used to represent kilo-byte and mega-byte respectively.

fxintegrate.properties			
Property	**Type**	**Example**	**Description**
fxintegrate.logging.maxfilecount	Integer	10	Number of rolled files before oldest file gets deleted. When *maxfilesize* is reached, the file will be named file.1, file.2 and so on until maxfilecount is reached.
fxintegrate.logging.pattern	String	%r [%t] %-5p %c - %m%n	This formats the logging output for display on stdout and within files. For details on logging patterns, please refer to appendix C
fxintegrate.cert.root.ca.file	String	/certs/cx-ca.cert	This allows the client to specify which root CA certificate to trust when doing SSL authentication. Multiple trusted roots may be specified by appending ".1", ".2" etc. onto the property name.
fxintegrate.cert.host.verify	Boolean	true	This allows the client to specify that it wishes to verify that the hostname of the server to which it has connected matches the DN field of the certificate presented to it by the server during SSL authentication.

The properties file is a standard Java property resource file. The file contains a series of name-value pairs separated by an equal (=) sign. Comments can be inserted into the file with a pound sign (#) as the first character. The following is a sample properties file:

```
# fxintegrate.properties
fxintegrate.host=fxintegrate.currenex.com
fxintegrate.port=443
fxintegrate.protocol=tcp
fxintegrate.proxyhost=bankproxy.bank.com
fxintegrate.proxyport=88
fxintegrate.logging.priority=ERROR
fxintegrate.logging.location=/proc/logs/fxintegrate.log
fxintegrate.cert.root.ca.file=./cx_root_ca.der
fxintegrate.cert.int.ca.file=./cx_int_ca.der
```

7.2 API Classes - Connection

This section details the key classes that make up the public interface of the FXintegrate ESP API.

7.2.1 CxSessionContext

The CxSessionContext object represents a client connection to the FXintegrate service. It allows the client to connect, authenticate and disconnect from the FXintegrate service. Calls must be made in a particular order for proper connection to the trading service. The details of the initialization and authentication steps are explained in subsequent sections.

currenex.client.session.fxintegrate.CxSessionContext			
Return Type	**Method**	**Parameters**	**Description**
String	connect	CxMessageClient client String loginId	Establishes a connection to the FXtrades service. Returns an encrypted challenge string upon successful connection.
void	authenticate	String decryptedChallenge	This call should be made immediately after the connect call returns. It takes the decrypted challenge string.
void	startTrading	N/A	Indicates to the FXtrades service that the connected party is ready to participate in active trading.
void	stopTrading	N/A	Indicates to the FXtrades service that the connected party is no longer participating in active trading.
void	Disconnect	N/A	Instruct the session to terminate connection with the server. All session related resources will be removed on the server.
void	sendCxMessage	String msg	This method sends a client message to Currenex. The message should be an XML string conforming to the Currenex specified FX transaction DTD.
void	Quiesce	boolean	Not used.

7.2.2 CxSessionFactory

CxSessionFactory is a factory class that will return a concrete implementation of a CxSessionContext interface (explained in the previous section). This class is declared final and cannot be extended.

An FXintegrate client program should use this class to obtain a CxSessionContext as special initialization steps are performed by CxSessionFactory to create a CxSessionContext.

currenex.client.session.fxintegrate.CxSessionFactory			
Return Type	**Method**	**Parameters**	**Description**
CxSessionContext	getCxSessionContext	N/A	Construct and return a CxSessionContext object with the default values provided in the FXintegrate.properties file.
CxSessionContext	getCxSessionContext	Properties props	Construct and return a CxSessionContext object with the values provided in the Properties object passed in as a parameter

7.2.3 IMessageClientBlue

An FXintegrate client must implement this interface in order to participate in auto-trading with FXtrades. This interface specifies a single callback method that allows the client to receive messages from FXtrades.

An instance of a IMessageClientBlue is passed to the CxSessionContext when the client makes the connect call. The CxSessionContext will use this interface to communicate back to the client.

currenex.client.session.fxintegrate.CxMessageClient			
Return Type	**Method**	**Parameters**	**Description**
void	handleCxMessage	CxMsg cxMsg	Callback method used by the CxSessionContext receive messages from the FXtrades service.
String	Reauthenticate	String challengeString	Callback method used by the CxSessionContext object to signal the client that a re-authentication process is required. The client must supply a method that decrypts the supplied challenge string and returns it.

7.3 API Classes – MarketData

Y – Required and mandatory.

N – Optional field

C – Conditional

7.3.1 MarketDataRequest

Parameter	Type	Required	Description
Id	String	Y	Unique id for each subscription. Same id should be used during unsubscribe.
subscriptionType	MDSubscriptionType	Y	Market data subscription type. Possible types- 1. SUBSCRIBE 2. UNSUBSCRIBE
ccyPair	String	Y	Instrument or currency pair (BASE/TERMS)
valueDate	FxDate	N	Null – Defaults to spot. Only spot is supported in this version.
entryType	MDEntryType	Y	Price entry type. Supported types- 1. TWO_WAY 2. BID 3. OFFER
depthType	MDDepthType	C	Market depth type. 1. FULL_BOOK 2. TOP_OF_BOOK Required field for SUBSCRIBE type.
bookType	MDBookType	C	1. AGGREGATED 2. ATTRIBUTED Required field for SUBSCRIBE type.

7.3.2 MarketDataUpdate

Parameter	Type	Required	Description
Id	String	Y	Subscriiption request id in the MarketDataRequest message.
mdEntries	MarketDataEntry[]	Y	Set of market data entries.

7.3.3 MarketDataEntry

Parameter	Type	Required	Description
Id	String	Y	Market data entry id. Unique reference set by Currenex which should be used to manage the prices and the local book.
mdUpdateAction	MDUpdateAction	Y	Market data update actions. Possible types- 1. NEW 2. CHANGE 3. DELETE
mdEntryType	MDEntryType	Y	Possible types 1. BID 2. OFFER
ccyPair	String	Y	Instrument or currency pair.
Price	InstrumentPrice	C	Market data price. Null if the MDUpdateAction is DELETE.
Money	Money	C	The liquidity amount. Null if the MDUPdateAction is DELETE.
valueDate	FxDate	C	Value date.
deleteReason	String	C	Comment on the MDUpdateAction.DELETE
orderCount	Integer	C	Number of like orders or prices for the aggregated price amount
Position	Integer	C	The position in the depth
sourceName	String	C	The price source name.
Indicative	Boolean	Y	True if indicative price.
Comment	String	C	General comment
creationTime	Date	Y	Price creation time.

7.3.4 MarketDataRequestReject

Parameter	Type	Required	Description
Id	String	Y	Subscriiption request id in the MarketDataRequest message.
errorCode	Int	Y	Error code.
errorDescription	String	Y	Error description for the reject.

7.4 API Classes – Orders

7.4.1 Mandatory parameters based on order type and action type

The parameters that need to be set using the corresponding set methods based on the order types.

See javadoc for OrderActionBuilder class for the description of each parameter.

Parameter	Type	Market	Limit	Stop	Iceberg	New	Replace	Cancel
AccountId	String	N	N	N	N	N	N	N
Allocations	Allocation	N	N	N	N	N	N	N
Amount	Amount	**Y**	**Y**	**Y**	**Y**	**Y**	**Y**	N
BuySell	EnumBuySell	**Y**	**Y**	**Y**	**Y**	**Y**	**Y**	N
CcyPair	String	**Y**	**Y**	**Y**	**Y**	**Y**	**Y**	**Y**
ClientUserId	String	N	N	N	N	N	N	N
Comment	String	N	N	N	N	N	N	N
DraftId	String	**Y**	**Y**	**Y**	**Y**	N	**Y**	**Y**
Expiration	Expiration	N	N	N	N	N	N	N
Id	String	**Y**	**Y**	**Y**	**Y**	**Y**	**Y**	**Y**
LimitRate	Rate	N	**Y**	N	N	**Y**	**Y**	N
MaxDisplayAmount	Amount	**N**	**N**	N	**Y**	**Y**	**Y**	N
MinExecutionAmount	Amount	N	N	N	N	N	N	N
OrderType	OrderType	**Y**	**Y**	**Y**	**Y**	**Y**	**Y**	N
OriginalId	String	**Y**	**Y**	**Y**	**Y**	N	**Y**	**Y**
PendingAction	EnumPendingAction	N	N	N	N	N	N	N

StopRate	Rate	**N**	**N**	Y	N	**Y**	**Y**	N
TradeCcy	String	**Y**	**Y**	**Y**	**Y**	**Y**	N	N
ValueDate	FxDate	N	N	N	N	N	N	N

7.4.2

7.4.3 OrderActionBuilder

It is used to create OrderAction messages that need to be sent to Currenex.

- Create OrderActionBuilder for a specific action.
  - OrderActionBuilder builder = new OrderActionBuilder(OrderActionType)
- Set the mandatory and optinal fields using the corresponding set methods.
- Call builder.getMessae() which returns OrderAction message.
- Send the order action message to Currenex.

Parameter	Type	Description
AccountId	String	The account id (SubFund id) name as agreed to by Currenex and the client or else defaulted by the system. An API session can trade on one or more trading accounts.
Allocations	Allocation	Pre trade allocations
Amount	Amount	The order amount in the currency side specified in the tradeSide field.
BuySell	EnumBuySell	Order Buy/Sell enum type.
CcyPair	String	The currency pair of the order.
ClientUserId	String	Third-party transaction identifier; an explicit value agreed to with Currenex or else defaulted by the system.
Comment	String	Comment
DraftId	String	Unique order identifier assigned by Currenex. This is used to update or cancel an active order. This will be same on all fills, expired and cancel execution result message from Currenex. Note: this is not the same as the completed trade id, which appears in tradeId field of CxOrderExecution response.

Expiration	Expiration	Specifies how long an order remains in effect.
Id	String	Client assigned unique identifier for the order.
LimitRate	Rate	The rate at which the limit order is to be executed.
MaxDisplayAmount	Amount	Optional field used for Iceberg order type, to represent the amount to be shown to others viewing the order. If omitted, the actual order amount is the show amount.
MinExecutionAmount	Amount	Minimum execution amount on any fill. If the pending amount is less than the specified minimum execution amount, then the order will be cancelled by Currenex and a message is sent. If omitted, then it defaults to the actual order amount.
OrderType	OrderType	Order type enum.
OriginalId	String	The client assigned ID of the order to be canceled.
PendingAction	EnumPendingAction	Pending actions if any.
StopRate	Rate	The spot rate at which the stop order becomes effective.
TradeCcy	String	The dealt currency side of the order. This is the currency side for the amount specified in Amount field.
ValueDate	FxDate	Settlement date. If omitted it defaults to SPOT.s
BrokerMatchIds	String[]	List of broker id's that this order can be matched against.

7.5 Order action responses

7.5.1 CxOrderReject

Response message sent from Currenex for a cancel reject or order action error. See javadoc for more info.

- Use corresponding get methods of the below parameters to retrieve the values.

Parameter	Type	Description
ClientOrderId	String	Client unique order id sent as part of the OrderAction setId method.
ClientOrigOrderId	String	Original order id. Sent during cancel or replace action.

ClientUserId	String	Client user id sent as part of OrderAction setClientUserId method.
Code	Int	Error code
Description	String	Error/Reject description
DraftId	String	Currenex confirmation Id for cancels & replaces action. Null for New order action.
OrderType	OrderType	Order type
TransactionTime	Date	Server transaction time

7.5.2 CxOrderExecution

Response message sent from Currenex for an accepted order. See javadoc for more info.

- Receive the OrderExecutionResult message from Currenex in the handleCxMessage
- CxOrderExecution orderExec = orderExecutionResult.getExecution();
- Use corresponding get methods of the below parameters to retrieve the values.

Parameter	**Type**	**Description**	PARTIAL /FILLED	CONFIRMED/ EXPIRED/ CANCELLED
AccountId	String	The account id (SubFund id) name as agreed to by Currenex and the client or else defaulted by the system. An API session can trade on one or more trading accounts.	N	N
Allocations	Allocation	Pre trade allocations	N	N
Amount	Amount	The actual order amount in the currency side specified in the tradeSide field.	Y	Y
BuySell	EnumBuySell	Order Buy/Sell enum type.	Y	Y
CcyPair	String	The currency pair of the order.	Y	Y
ClientOrderId	String	The active client order id.	Y	Y
ClientUserId	String	Third-party transaction identifier; an explicit value agreed to with Currenex or else defaulted by the system.	N	Y

Comment	String	Comment	N	N
CumDealt Amount	Amount	The total filled amount on this order.		
DraftId	String	Unique order identifier assigned by Currenex in the Order confirmation for New or Replace actions.	Y	Y
ExecAgainst Amount	Amount	Contra amount for the executed amount on this fill.	Y	N
ExecAmount	Amount	Quantity bought/sold for this fill.	Y	N
ExecPrice	Rate	Execution rate on this fill.	Y	N
AvgExecPrice	Rate	Volume weighted average fill price. It's calculated based on the various partial fill execution price and the amounts.	Y	N
ExecutionClient UserId	String	Useid on the fill.	Y	N
IsActive	Boolean	Flag to indicate the current status of the order. If true, the order is still active and can perform cancel or replace actions. If false, the order is inactive.	Y	Y
IsOpenAmount BelowMin	boolean	True if the open amount after the fill is less than the client min execution amount or Currenex minimum.	Y	Y
LimitRate	Rate	The rate at which the limit order is to be executed.	Y(For Limit & Iceberg)	Y
MaxDisplay Amount	Amount	Optional field used for Iceberg order type, to represent the amount to be shown to others viewing the order. If omitted, the actual order amount is the show amount.	Y(For only Iceberg)	Y(For only Iceberg)
OpenAmount	Amount	Remaining amount after the last fill	Y	Y
OrderType	OrderType	Order type	Y	Y
StopPrice	Rate	Stop rate entered by the client.	Y(for Stop order type)	Y(For only Iceberg)
TradeDate	FxDate	Trade date & time	Y	N
TradeId	String	Unique Currenex identifiers for each fill. This is different from the draft id.	Y	N
TradeSide	EnumBaseT	The currency side.	Y	Y

	erms			
Transaction Time	Date	Server transaction time	Y	Y
Type	ExecutionResultType	Execution result type.	Y	Y
ValueDate	FxDate	Value date	Y	N
ExecBrokerId	String	The execution broker id on this execution.	Y	N
PrimeBrokerId	String	The trade give up bank id	Y	N

8 Session Management

8.1.1 Session Authentication

The FXintegrate framework utilizes a public-key crypto system for client authentication. As explained in the overview section, each client has a private key and certificate in its possession. The FXintegrate service relies on the client's possession of this private key for identification.

When a client performs a *connect* call to begin the authentication process, a message is sent to the FXintegrate service with the ID of the connecting client. Upon receiving the request, the FXintegrate service will perform the following steps in constructing an encrypted challenge:

1. Extract the client ID from the connection request message.
2. Extract the client's digital certificate from the certificate repository.
3. Generate a secure-random challenge message for the client.
4. Encrypt the challenge message with the client's public key extracted from the client's digital certificate.
5. Send the encrypted challenge message to the client.

Upon receiving the encrypted challenge, the client must decrypt the challenge string and return the result to the FXintegrate service to complete the authentication process. The steps necessary for the client to complete the authentication process are outlined below:

1. Base64 decode the challenge string.
2. Decrypt the base64 decoded byte array with the client's private key.
3. Base64 encode the decrypted byte array.
4. Send the base64 encoded decrypted challenge back to the server to complete the authentication process.

The call will complete successfully if the decrypted challenge is in fact accurate. If the decrypted challenge does not match the original challenge created by the FXintegrate service, an exception will be raised.

There is a response time associated with each authentication process. This time restriction is enforced to discourage crypto-analysis attacks against the encrypted challenge. The actual time requirement will depend on the security policy established by Currenex and could vary depending on requirements. If an authentication response is received after the response time expired, an exception is raised and the connecting client must reinitiate a connect call to the FXintegrate service, in which a new encrypted challenge will be included.

Attempts to send trading messages will fail if the client has not successfully completed the authentication process. A successful authentication process consists of completing the *connect* and *authenticate* method calls without exceptions being raised.

A CIA can connect to the FXtrades service via one of two connect calls. A standard connect call that returns a challenge string in the PKCS1 format and an alternate connect call that returns a challenge string in the PKCS7 format. Both formats are explained in

the sections below. Challenge encryption is performed during runtime, this means the client does not have to inform the server which encryption scheme is desired during set-up time, simply choose from one of the two available connect methods. Encryption scheme cannot be changed during the lifetime of a session or during the re-authentication of the same session. In order to change the authentication encryption format, the client must disconnect from Currenex and connect again via the CxSessionContext.

8.1.1.1 PKCS1 Authentication Format

One form of the connect call, when calling the connect method of CxSessionContext, returns a PKCS1 formatted authentication challenge. This challenge is a raw PKCS1 byte array, meaning it is a RSA encrypted challenge without other cryptographic wrappers. The challenge string in essence has three parameters that affect it's creating, thus affecting the decryption of the challenge string. They are:

Cipher – RSA

Mode of Operation – ECB (Electronic Code Book)

Padding – PKCS1

This means during the decryption process, the decryption tool must use these parameters to properly decrypt the challenge. These are standard parameters for a RSA operation. Any RSA capable tool should have methods or procedures in setting these parameters.

8.1.1.2 PKCS7 Authentication Format

An alternate form of connecting to Currenex is to invoke the connectPKCS7 method of CxSessionContext. PKCS7 is essentially a PKCS1 object wrapped in a cryptographic envelope. This provide more structure to PKCS1 and allow some more high level tools to decrypt the challenge without manually setting various parameters such as *Mode of Operation* and *Padding* parameters since PKCS7 is a self describing encoding standard.

8.1.2 Session Timeout and Re-authentication

Due to the sensitive nature of communications between Currenex and members, the FXintegrate service will, from time to time, require the connected client to re-authenticate itself when the session is near timeout.

For example, if a session is valid for 8 hours, then after 7 hours and 50 minutes have elapsed, the FXintegrate service could reissue an authentication request message (the actual session and elapsed time period is controlled by FXintegrate and determined by Currenex policy, its actual values should not affect normal client operations).

Upon completing the initial authentication procedures, subsequent re-authentication events will be directed to the client through a callback method. The client does not have to process a separate FXintegrate message as the message will be intercepted by the API and redirected via the callback. After receiving a new encrypted challenge through the callback, the client should handle the request in the same was as the initial connection to the service. The steps are:

1. Base64 decode the challenge string.
2. Decrypt the base64 decoded byte array with the client's private key.

3. Base64 encode the decrypted byte array.
4. Send the base64 encoded decrypted challenge back to the server to complete the authentication process.

Upon successful re-authentication, the client will not be separately notified. The client can assume the re-authentication procedure was accepted unless an error message is received.

8.1.3 Start Trading

The first step in gaining access to the FXtrades service is to authenticate through FXintegrate. Authentication through FXintegrate, however, does not mean the member is ready to trade. To actually begin trading, the client must call the startTrading method on the session context. This will signal to the FXtrades service that the member is connected and ready to accept price request messages.

8.1.4 Stop Trading

A member can stop active trading without logging out of the service by calling the stopTrading method on the session context The next section will talk about ending the trading session. To resume trading, simply call the startTrading method described in the previous section.

8.1.5 Session Disconnect

When the client has finished its communication session with the FXintegrate service, or when an error condition that forces a shutdown is detected, the client should explicitly disconnect to ensure that proper memory cleanup and release of resources can be performed by the FXintegrate service.

8.1.6 Session Establishment Sequence Diagram

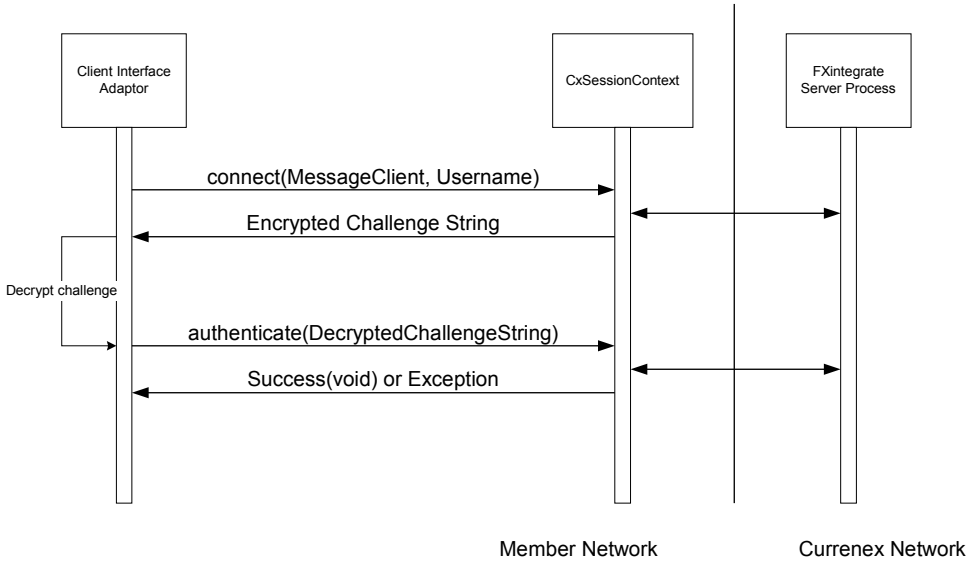


Figure 1: Session Establishment

9 Message Delivery & Threading Considerations

The FXintegrate client API is a multithreaded API. It delegates threads for message reception, message processing and message delivery. The exact threading architecture of the API is beyond the scope of this document, but it important to point out certain expectations of the API so thread synchronization and starvation do not occur.

Of the various areas mentioned in the previous paragraph, special attention must be paid to message delivery. Message delivery falls into two categories:

- Message delivery from API to CIA
- Message delivery from CIA to Currenex

9.1.1 Message delivery from API to CIA

In this category, messages from the API are delivered to the CIA via single threads. The API could be utilizing different threads for each delivery, but from the CIA's perspective, the messages appear as though they are being delivered via a single thread. The clearest characteristic of this from a CIA's point of view is that no message will be delivered unless the previous call to CxMessageClient's handleCxMessage method has returned.

The reason for this behavior is to make sure message arrive at the CIA in order, as the client API received them. If multiple threads were simultaneously deployed to deliver the messages, some messages could be received out of order, causing undesirable actions to be performed on trade requests.

The recommended method to handle messages is for the CIA's CxMessageClient implementation to queue incoming messages from the client API and return immediately. The decision to process the messages via a single thread or multi-threads is then left to the CIA to decide, the client API can continue to function and deliver messages in a timely fashion.

9.1.2 Message delivery from CIA to Currenex

When calling the sendCxMessage method on the CxSessionContext, the caller is guaranteed to return immediately. The client API queues each message and delivers them in order to Currenex. Every message is marked as guaranteed send from client API to Currenex. If the message is sent when there are connection issues, the client API will queue the messages and send them when there is a good connection. If the client API fails to reconnect and the CIA open a new connection, all the messages in the queue are dropped.

9.1.3 Override guarantee send from CIA to Currenex

Clients can now override the message sending guaranteed or not in two ways.

1. Specify the global config property “fxintegrate.guarantee.sending=true”. If the config property is not specified in the fxintegrate.properties, then it defaults to true.
2. Use sendCxMessage(AMsg, guaranteedFlag) in the CxSessionContext to override the default and the global guaranteed flag setting.

9.1.4 Receiving notifications on the dropped “non-guaranteed” messages

If the message sending is marked as **non-guaranteed** and if the Client API fail to send the messages (assuming the connection/session is still open) to Currenex, then the CIA will receive notifications on the dropped non-guaranteed messages in the form of a call-back method **handleDroppedMessage** method. Client API will not drop any **guaranteed** messages as long as the connection and session is not closed.

To receive callbacks on the dropped **non-guaranteed** messages,

- Implement currenex.client.session.fxintegrate.**DroppedClientMessageHandler** interface.
- Call **setDroppedClientMessageHandler**(currenex.client.session.fxintegrate.DroppedClientMessageHandler) method on the CxSessionContext instance.
- Implement **handleDroppedMessage(Object someMsg, String reason)** method.
- Check the message type based on message object.

  Example Code:

```
Public void handleDroppedMessage(Object someMsg, String reason) {
        If(msg instanceof OrderAction) {
                OrderAction droppedOrderMsg = (OrderAction)someMsg;
        }
}
```

Appendix A: Currenex Date/Time formats

Other than the *sendTime* element, all of FXintegrate's date information is handled in the format of a GeneralizedTime. GeneralizedTime is a time standard similar to Coordinated Universal Time (UTC), but created for the ASN.1 specification. A GeneralizedTime object is used to represent time values with a higher precision than done by the normal UTC time, which only allows a precision down to seconds. A GeneralizedTime value specifies the calendar date according to ISO 2014 (year, month, day: YYYYMMDD), the time value (any precision) and optionally the local time-shift against UTC time. A Z terminating the time value indicates that it corresponds to UTC time (Zulu time).

For instance: local time is GMT and represents the twelfth November 1997 at 15:30:10,5p.m:

- 19971112153010.5Z

Supposing a time-shift of two hours making the local time ahead from GMT, we may write:

- 19971112173010.5+0200

For situations where only the date is relevant, Currenex recommends setting the time information to zero.

Appendix B: RSA Decryption

During client authentication, the FXintegrate service uses a public-key crypto system to authenticate a connecting user. The FXintegrate service will encrypt a challenge string with the connecting client's public key embedded within a digital certificate. To prove that the connecting client is the actual user referenced by the digital certificate, the client must decrypt the encrypted challenge string with the associated private key.

As part of the FXintegrate client library, a class is provided to decrypt the challenge string. Currenex does not supply the underlying cryptographic library that this class requires. Members must purchase their own licenses for the cryptographic library of their choice.

The underlying encryption algorithm is RSA, so the use of this class requires the presence of an encryption library that provides support for the RSA algorithm.

This class uses the Java Cryptographic Extension (JCE) framework to provide its functionality. The Java™ Cryptography Extension (JCE) is a package that provides a framework and implementations for encryption, key generation and key agreement, and Message Authentication Code (MAC) algorithms. Support for encryption includes symmetric, asymmetric, block, and stream ciphers. The software also supports secure streams and sealed objects.

JCE is designed so that other qualified cryptography libraries can be plugged in as service providers, and new algorithms can be added seamlessly.

IAIK in Austria, has developed a JCE1.2 compliant provider that includes key generation, cipher and signature services for RSA.

For details on JCE 1.2, please see:

http://java.sun.com/products/jce

For details on IAIK's implementation of Sun's JCE 1.2, please see:

http://jcewww.iaik.at

In order to use this library, the client must install a JCE provider. The following is a list of JCE compliant providers identified by Sun.

http://java.sun.com/products/jce/jce12_providers.html

To date FXintegrate has only been tested with the IAIK JCE provider. However, the client is free to use this or any other JCE compliant provider.

currenex.share.util.RsaDecryptJce			
Return Type	**Method**	**Parameters**	**Description**
N/A	RsaDecryptJce	N/A	Null constructor, must call init() before decryption methods can be called.
N/A	RsaDecryptJce	PrivateKey	Construct a RsaDecryptJce

currenex.share.util.RsaDecryptJce			
Return Type	**Method**	**Parameters**	**Description**
			object with the provided java.security.PrivateKey.
byte[]	decrypt	byte[]	Decrypt the input byte[]. If decryption errors occur, exceptions will be thrown.
String	decryptChallenge	String	Convenience method provided for decrypting an encrypted challenge sent by FXtrades. The returned String object can be directly inserted into the authentication response message. This method call will, base64 decode the challenge string, then RSA decrypt the byte[], finally base64 encoding the resulting byte array. It is equivalent to calling decrypt(base64decode(challenge)) and then base64 encoding the returned byte array.
Void	init	PrivateKey	Initialize the RsaDecryptJce class with a java.security.PrivateKey object.

The following is an example of how to use the RsaDecryptJce class in a client authentication sequence. It uses the IAIK JCE library to obtain the private key that's stored on disk, in PKCS8 format, encrypted with a password.

```
// load the IAIK cryptographic provider
// this will allow IAIK to handle all crypto related
// functionality.
iaik.security.provider.IAIK.addAsProvider();

// Obtain the encrypted challenge string by calling the connect method on the
CxSessionContext
String challenge = ctx.connect(messageClient, loginId);

// Read the byte stream of the private key from disk.
// An assumption is made that the private key is stored in the
// encrypted PKCS8 format as a DER encoded stream.
// An alternate location is to store the private key in a java
// KeyStore object which can contain a set of keys and
// certificates.  The key storage format is left to the
// discretion of the implementor.
FileInputStream fis = new FileInputStream(KEYFILE_LOCATION);
```

```
int size = fis.available();
byte[] keyBuff = new byte[size];
fis.read(keyBuff);

// Construct the PKCS8 object, decrypt the key with its
// associated password.
iaik.pkcs.pkcs8.EncryptedPrivateKeyInfo pkcs8 = new
iaik.pkcs.pkcs8.EncryptedPrivateKeyInfo(keyBuff);
java.security.PrivateKey pKey = pkcs8.decrypt(PASSWORD);

// Now initialise the RsaDecryptJce object with the previously
// obtained PrivateKey object.
RsaDecryptJce decrypt = new RsaDecryptJce(pKey);

// Decrypt the challenge string.
String decChallenge = decrypt.decryptChallenge(challenge);

// Now authenticate to the service, assuming we already have
// a connected CxSessionContext object.
ctx.authenticate(decChallenge);
```

The decryption library included in this release of FXintegrate only provides support for JCE compliant software based decryption. This library is provided as a convenience and reference class. The Currenex member is free to implement the RSA decryption process in any way they choose. The only requirement is that the authentication response message contains the correctly decrypted challenge string.

Appendix C: FXintegrate Logging Patterns

The conversion pattern is closely related to the conversion pattern of the printf function in C. A conversion pattern is composed of literal text and format control expressions called *conversion specifiers*. You are free to insert any literal text within the conversion pattern.

Each conversion specifier starts with a percent sign (%) and is followed by optional *format modifiers* and a *conversion character*. The conversion character specifies the type of data, e.g. category, priority, date, thread name. The format modifiers control such things as field width, padding, left and right justification.

The following is a simple example.

If the conversion pattern is

"%-5p [%t]: %m%n"

then the logging statements would look something like the following:

DEBUG [main]: Message 1

WARN [main]: Message 2

Note that there is no explicit separator between text and conversion specifiers. The pattern parser knows when it has reached the end of a conversion specifier when it reads a conversion character. In the example above the conversion specifier **%-5p** means the priority of the logging event should be left justified to a with of five characters. The recognized conversion characters are as follows:

9.1.4.1.1 Logging Patterns	
Pattern Characters	**Pattern Description**
c	Used to output the category of the logging event. The category conversion specifier can be optionally followed by *precision specifier*, that is a decimal constant in brackets. If a precision specifier is given, then only the corresponding number of right most components of the category name will be printed. By default the category name is printed in full. For example, for the category name "a.b.c" the pattern **%c{2}** will output "b.c".

9.1.4.1.1 Logging Patterns	
Pattern Characters	**Pattern Description**
C	Used to output the fully qualified class name of the caller issuing the logging request. This conversion specifier can be optionally followed by *precision specifier*, that is a decimal constant in brackets. If a precision specifier is given, then only the corresponding number of right most components of the class name will be printed. By default the class name is output in fullly qualified form. For example, for the class name "org.apache.xyz.SomeClass", the pattern **%C{1}** will output "SomeClass". **WARNING** Generating the caller class information is slow. Thus, its use should be avoided unless execution speed is not an issue.
d	Used to output the date of the logging event. The date conversion specifier may be followed by a *date format specifier* enclosed between braces. For example, **%d{HH:mm:ss,SSS}** or **%d{dd MMM yyyy HH:mm:ss,SSS}**. If no date format specifier is given then ISO8601 format is assumed.
F	Used to output the file name where the logging request was issued. **WARNING** Generating caller location information is extremely slow. Its use should be avoided unless execution speed is not an issue.
l	Used to output location information of the caller which generated the logging event. The location information depends on the JVM implementation but usually consists of the fully qualified name of the calling method followed by the callers source the file name and line number between parentheses. The location information can be very useful. However, it's generation is *extremely* slow. It's use should be avoided unless execution speed is not an issue.
L	Used to output the line number from where the logging request was issued. **WARNING** Generating caller location information is extremely slow. Its use should be avoided unless execution speed is not an issue.
m	Used to output the application supplied message associated with the logging event.
M	Used to output the method name where the logging request was issued. **WARNING** Generating caller location information is extremely slow. Its use should be avoided unless execution speed is not an issue.

9.1.4.1.1 Logging Patterns	
Pattern Characters	**Pattern Description**
n	Outputs the platform dependent line separator character or characters. This conversion character offers practically the same performance as using non-portable line separator strings such as "\n", or "\r\n". Thus, it is the preferred way of specifying a line separator.
P	Used to output the priority of the logging event.
r	Used to output the number of milliseconds elapsed since the start of the application until the creation of the logging event.
t	Used to output the name of the thread that generated the logging event.
x	Used to output the NDC (nested diagnostic context) associated with the thread that generated the logging event.
%	The sequence %% outputs a single percent sign.

Below are various format modifier examples for the category conversion specifier.

Available Format Modifiers				
Format Modifier	**Left Justify**	**Minimum Width**	**Maxium Width**	**Description**
%20c	FALSE	20	none	Left pad with spaces if the category name is less than 20 characters long.
%-20c	TRUE	20	none	Right pad with spaces if the category name is less than 20 characters long.
%.30c	NA	none	30	Truncate from the beginning if the category name is longer than 30 characters.
%20.30c	FALSE	20	30	Left pad with spaces if the category name is shorter than 20 characters. However, if category name is longer than 30 characters, then truncate from the beginning.
%-20.30c	TRUE	20	30	Right pad with spaces if the category name is shorter than 20 characters. However, if category name is longer than 30 characters, then truncate from the beginning.