InfoReach

Engineered for what's next

InfoReach FIX Engine

FIX Java Programmer's Guide

Version 9.0

Contents

Introduction

InfoReach FIX Engine's library provides user applications with all the necessary APIs for establishing FIX-session connections with counter parties, for creating and parsing messages in both standard FIX and FIXML formats, and for listening to various types of event flow that results from business-level message exchange, and from FIX connection session-level monitoring. The **InfoReach** engine library contains two independent parts, one of which is responsible for initiating, terminating, and supporting the FIX session-level protocol during application run-time. The other part supports messaging API for creation and parsing of messages that can be sent or received over the FIX connections. The messaging part of the **InfoReach** engine can be used by applications independently from the FIX session-level support part.

Together with the engine library, the **InfoReach** distribution contains tool applications for monitoring deployed FIX connections at run-time, configuring deployment of the engine network, and preparing a custom version of the FIX field and message sets. The *InfoReach User Guide* contains a detailed description of these tools.

1 InfoReach Messaging Overview

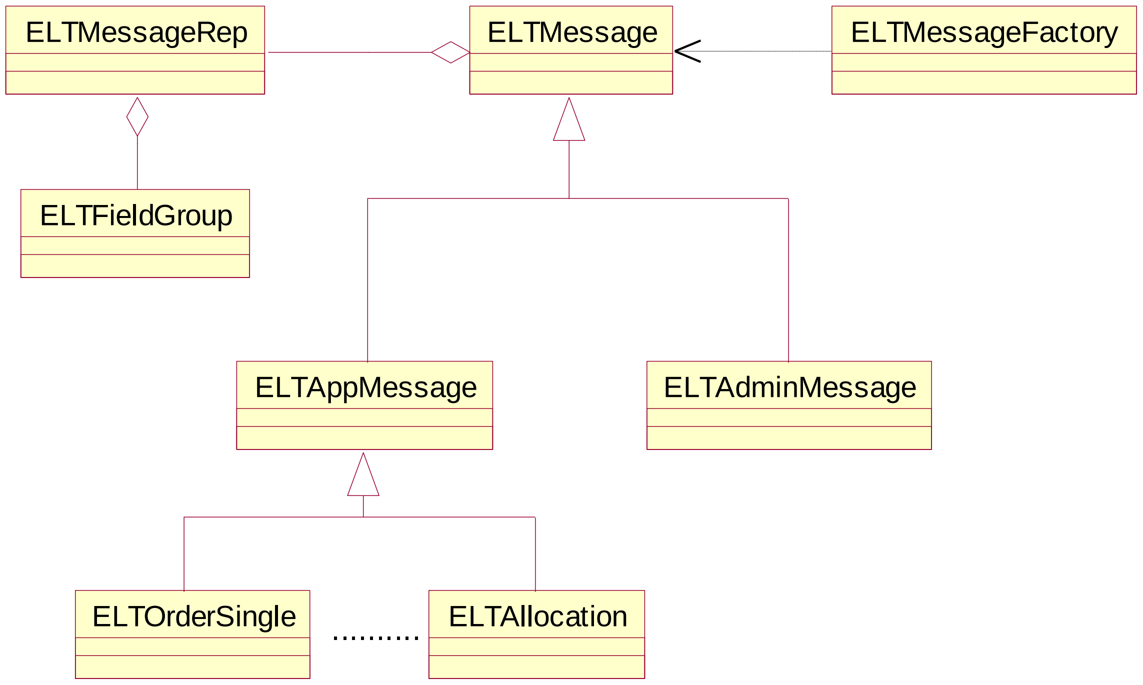
1.1 Field and Message Sets

InfoReach's messaging packages can be used to process FIX messages in FIX4.0, FIX4.1, FIX4.2, FIX4.3, FIX4.4 and FIXML formats. All messages described in the FIX protocol documentation are supported and can be sent or received by **InfoReach** engines. The advantage of InfoReach is that all message and field definitions for different protocol versions are stored outside the library in special XML files (distributed to users along with the library). This means that new fields, messages, or even versions can be supported. No upgrade is required, and only XML configuration files need to be augmented with new information. The **Metadata Editor** tool distributed with **InfoReach** provides capability to manage protocol-standard and user-defined FIX fields and messages.

1.2 InfoReach Message Hierarchy

InfoReach's message hierarchy is shown in Figure 1. It illustrates that every message class (application or admin) is derived from the **ELTMessage** and through it owns the object of type **ELTFieldGroup**. The object of type **ELTFieldGroup** is a generic collection of values. Each value is associated with a specific message field. A variable number of values can be stored in the **ELTFieldGroup** object as required for each particular message.

Figure 1: Message Hierarchy



The **ELTFieldGroup** class may be the only class the client application uses for creating and sending messages through **InfoReach's** engines. An application would instantiate the **ELTFieldGroup** object, and populate the required number of fields using **ELTFieldGroup's** interface. It would also instruct its engine proxy (see sections 2.1 and 2.2) to create a message object of the right type and format that conforms to the protocol version, and send it to the required destination. In this scenario, the engine will instantiate the message object of the correct type and format by calling **ELTMessageFactory**. In case the application needs to work with the message object and not just the **ELTFieldGroup** object, it can make a call by itself to **ELTMessageFactory**. Section 2.2.3 presents sample code for creating and sending a FIX **NewOrderSingle** message through the engine.

1.3 FIXML Support

InfoReach engines can send and receive messages in FIXML format. With FIXML, the whole message body is represented as one string in XML format attached to the **XmlData** field (tag 213). The application can take the responsibility of creating this string when it tries to send a FIXML message(s), and it can decode this string upon reception of FIXML message. In a different scenario, the message can be created in standard way as described in section 2.2. The engine will convert it to FIXML format before it goes out. The engine can also automatically convert received FIXML messages. The conversion of sent and received messages will take place if the connection's format properties ("wireFormat" and "publishFormat") are set showing that the conversion is required (see description of "wireFormat" and "publishFormat" properties in 3.2). All FIXML conversions are currently done according to version "FIXML 4.2 - 1.1.0".

InfoReach's engine view provides users with the ability to visually create and send messages in either FIX or FIXML format. For more information, see the ***InfoReach User Guide***.

1.4 XML conversion

The **ELTFieldGroup** object can be converted to plain XML representation for easy data exchange between different application components. Use **ELTFieldGroup's** method *toXMLData()* to create an **XMLData** object from an **ELTFieldGroup** object. For example, to get XML string representation from a field group use the following call:

```
String xmlOrderSingle = fields.toXMLData().toString();
```

Simple order-single field group will be converted to the following XML structure:

```
<?xml version='1.0'?>

<FieldGroup>
    <Field name="HandlInst" value="1" />
    <Field name="Side" value="1" />
    <Field name="OrdType" value="1" />
    <Field name="TransactTime" value="20010505-11:25:00" />
    <Field name="OrderQty" value="400" />
    <Field name="Currency" value="USD" />
    <Field name="MsgType" value="D" />
    <Field name="Symbol" value="MSFT" />
    <Field name="ClOrdID" value="Order.1" />
</FieldGroup>
```

If the **ELTFieldGroup** contains subgroups, then the subgroups will appear inside the XML structure as sub-elements:

```
<?xml version='1.0'?>

<FieldGroup>
    <Field name="HandlInst" value="1" />
    <Field name="Side" value="1" />
    <Field name="OrdType" value="1" />
    <Field name="TransactTime" value="20010505-11:25:00" />
    <Field name="OrderQty" value="400" />
    <Field name="Currency" value="USD" />
    <Field name="MsgType" value="D" />
    <Field name="NoAllocs" value="2" />

    <FieldGroup name="NoAllocs">
        <Field name="AllocAccount" value="Account1" />
        <Field name="AllocShares" value="2000" />
    </FieldGroup>

    <FieldGroup name="NoAllocs">
```

```
            <Field name="AllocAccount" value="Account2" />
            <Field name="AllocShares" value="3000" />
        </FieldGroup>

        <Field name="Symbol" value="MSFT" />
        <Field name="ClOrdID" value="Order.1" />
    </FieldGroup>
```

2 Using Engine Proxy API for FIX Communication

NOTE: Engine Proxy API methods can throw multiple exceptions. See the on-line reference for a detailed description of all exceptions.

2.1 Connecting to Counter Parties

The application can instruct the engine managing client-mode FIX connections to initiate a specific connection. It does this by sending a logon message to the counterparty. The server engine, implementing server mode connections, waits on the assigned port for incoming connection requests (FIX logons). It then authenticates the requests and initializes the FIX session. Authentication of the logon requests received by all server engines on the network is handled by the engine manager's *pluggable* **Authenticator** object. The **Authenticator** object can be registered with the engine manager at runtime by the process instantiating the engine manager (see section 2.1.1).

Both client and server engines maintain multiple FIX connections. Engines themselves, without any involvement from the application, are responsible for all session-level communication, session sequence number management, heartbeats, message logging, session recovery, and any other session administration-related tasks.

As mentioned, the application only interfaces with engines through the engine proxy that it instantiates. The engine proxy API contains three methods that the application can use to instruct engines to initiate, terminate, or test FIX connections:

-
  ```
  public void connect(UserTicket usrTicket, ELTConnectionName connName)
      throws ELTConnectionUnknownException,
             ELTConnectionUnavailableException,
             ELTInvalidLogonException,
             SecurityException
  ```

  The client application calls this method whenever it needs to instruct the engine network to initiate a client-mode FIX connection with the name 'connName'. A client-mode connection with this name has to be described in the engine network configuration resource, and it has to be associated with the client engine visible to the application through the application's proxy. This method will throw an exception if the connection properties are not accessible. This is also true if the engine implementing this connection is invisible to the application.

  The **UserTicket** type parameter must be passed to the **connect()** method so the proxy has information about the user and the session initiating the connection. This information is used for security checks or logging. The global user ticket of the process can always be obtained through the **GlobalSystem’s** static **getUserTicket()** method:

```
ELTEngineProxy proxy = ELTSystem().getEngineProxy();
proxy.connect(GlobalSystem.getUserTicket(),
              new ELTConnectionName(“ClientConnAtoB”));
```

If there is a need for special user and session information to be attached to the connection, then a custom user ticket may be created at run time. For example, if the application wants to initiate the client connection ‘**ClientConnAtoB**’ for user ‘**user1**’ and session ‘**s-03/22/2000**,’ then it would make the following calls:

```
ELTEngineProxy proxy = ELTSystem().getEngineProxy();
proxy.connect(new UserTicket(“user1”, “s-03/22/2000”),
              new ELTConnectionName(“ClientConnAtoB”));
```

-
  ```
  public void connect(UserTicket usrTicket,
                      ELTConnectionName connName
                      Object m)
      throws ELTConnectionUnknownException,
             ELTConnectionUnavailableException,
             ELTInvalidLogonException,
             SecurityException
  ```

  This method is the same as above except that it accepts **ELTFieldGroup** as a third parameter that contains all of the extra information required by the counter-party in the logon message. Almost any field value can be specified in this **ELTFieldGroup** except the following:

  ```
              ELT.fieldBeginString
              FIX.fieldBodyLength
              FIX.fieldCheckSum
              ELT.fieldResetSeqNumFlag
              ELT.fieldPossDupFlag
              ELT.fieldPossResend
              ELT.fieldIsFromCounterParty
              ELT.fieldMessageFormat
              ELT.fieldMessageEncoding
              ELT.fieldMessageId
              ELT.fieldGlobalMessageId
              ELT.fieldConnectionName
  ```

-
  ```
  public void disconnect(UserTicket usrTicket,
                         ELTConnectionName connName,
                         String logoutText)
      throws ELTConnectionUnknownException,
             ELTConnectionUnavailableException,
             SecurityException
  ```

  Whenever the application wants to terminate the live FIX session by issuing a logout request, it can call the **disconnect()** method with the right connection name. This method also accepts the user ticket and the text message that will be placed inside the FIX logout request sent to the counter party:

```
proxy.disconnect(GlobalSystem.getUserTicket(),
                 new ELTConnectionName("ClientConnAtoB"),
                 "End of day logout");
```

Applications can terminate server-mode connections as well as client-mode connections.

-
  ```
  boolean isConnected(UserTicket usrrTicket, ELTConnectionName connName)
      throws ELTConnectionUnknownException,
             ELTConnectionUnavailableException,
             SecurityException
  ```

  This method will return **true** if the session for connection '**connName**' is active, and **false** otherwise. Again, both client-mode and server-mode connections can be tested this way.

Listing 1 presents a sample sequence of calls that the application can make to instantiate the engine proxy, initiate client connections, test the connection's state, and terminate the connection after it has finished:

```
// Initialize InfoReach FIX Engine component:
ELTEngineComponent.initialize(mainArguments);

// Get handle to engine proxy:
ELTEngineProxy proxy = ELTSystem().getEngineProxy();

…
…
// Initiate connection ClientConnAtoB:
proxy.connect(GlobalSystem.getUserTicket(),
              new ELTConnectionName("ClientConnAtoB"));

// If initiated successfully do some work:
if (proxy.isConnected(GlobalSystem.getUserTicket(),
                      new ELTConnectionName("ClientConnAtoB")))
{
    // Do some work:
    ...

    // Disconnect:
    proxy.disconnect(GlobalSystem.getUserTicket(),
                      new ELTConnectionName("ClientConnAtoB"),
                     "End of day logout");
}
```

Listing 1.

2.1.1 Authenticating Logon Requests

When the server engine receives a logon message with a request to initiate a FIX connection, the server engine passes this logon message to the **Authenticator** for authentication. The **Authenticator** object can be registered with the engine network (one central authenticator per engine network) by specifying its class name in the configuration. This is done by specifying property `authenticatorImplementationClass` inside the **enginenetwork.xml** file, or by setting it in the **Engine Network Configuration** tool's **General Properties** tab. The resulting xml section resembles the following:

```
<EngineNetwork
      authenticatorImplementationClass = "com.company.Authenticator"
      …
      …
/>
```

The ELTService process instantiates the **Authenticator**, and it will be called by **InfoReach** server engines. This will happen every time a session-initiating logon request is received from the counterparty. The **Authenticator** can either accept or reject a FIX session initiation request.

To implement an **Authenticator** object, the application creates an object. The object type is derived from **ELTAuthenticationAdapter** (package **com.inforeach.eltrader.message**). It implements the following abstract method:

```
public int authenticate(ELTAdminMessage m);
```

This method should return 0 (zero) if the authentication of a logon message *m* is successful and any other integer. For example:

```
public class CustomAuthenticator extends
      ELTAuthenticationAdapter
{
      public int authenticate(ELTAdminMessage m)
      {
            if (m.senderCompId().equals("ClientB"))
                  return 0;
            else
                  return -1;
      }
}
```

2.2 Creating and Sending Messages

Section 1.2 states that the user's application uses the **ELTFieldGroup** class for creating and sending messages through **InfoReach's** engines. An application would instantiate the **ELTFieldGroup** object, populate the required number of fields using **ELTFieldGroup's** interface, and then instruct its engine proxy to send it to the required destination through a specific FIX connection. Internally the right type message object with the correctformat conforming to the connection's protocol version is created and sent. The following sections contain a description of **ELTFieldGroup** class and related examples.

2.2.1 ELTFieldGroup's Interface and ELTValue Class

ELTFieldGroup class has a simple interface for setting and retrieving field values. The field tag numbers act as hash keys, making access to individual field values efficient. Since different fields may be different types, the **ELTValue** class (package **com.inforeach.eltrader.message.value**) is used as a placeholder for individual field values.

Presented below is a part of **ELTFieldGroup's** interface. You can use this for setting and retrieving field values. The full interface is given in the reference.

```
// returns number of field values inside the ELTFieldGroup
public int getFieldCount()

// returns ELTValue for the specified field tag. If field
// is not set returns null
public ELTValue getFieldValueIfSet(int fieldTag)

// set specific field's value to prepared ELTValue object
public void setFieldValue(int fieldTag, ELTValue value)

// set specific field's value to a primitive type,
// ELTValue object will be prepared internally:
public void setFieldValue(int fieldTag, int value)
public void setFieldValue(int fieldTag, long value)
public void setFieldValue(int fieldTag, double value)
public void setFieldValue(int fieldTag, char value)
public void setFieldValue(int fieldTag, boolean value)
public void setFieldValue(int fieldTag, String value)

// returns true if field is present inside ELTFIeldGroup and is set,
// returns false otherwise
public boolean isFieldSet(int fieldTag)
```

InfoReach's ELTFieldGroup class also supports storing the repeating subgroups within the group. It does this by implementing additional methods for adding and retrieving subgroups. The **fieldTag** parameter that's passed to these methods is the tag of the field that specifies the number of repeating subgroups in the message:

```
public int getGroupCount(int fieldTag)
public ELTFieldGroup getFieldGroup(int fieldTag, int index)
public void addFieldGroupAs(int fieldTag, ELTFieldGroup gr)
```

Whenever the **ELTValue** object is obtained by calling **ELTFieldGroup's getFieldValueIfSet()**method, one of the following methods can be used to retrieve the actual value of the right type:

```
public int     intValue()
public long    longValue()
public double  doubleValue()
public char    charValue()
public boolean booleanValue()
public String  stringValue()
```

It should be noted that the **IllegalFieldUsageException** or **NumberFormatException** may be thrown by **ELTValue's** method. This happens if the type of the value stored inside is different from the return type of the called method, which makes conversion impossible.

2.2.2 Loading Messages from File

Instead of creating messages programmatically, it is possible to load messages from a file using one of **ELTMessage's valueOf()** methods. The application needs only to take care of reading appropriately formatted strings representing a FIX message from any resource (file, db, messaging framework), and then call the **ELTMessage.valueOf()** method to create the correct type of message. The string should be in the same format as required by FIX, excluding some session-specific fields like body length, check sum, etc. The string must contain a message type field. The message type field should follow the protocol version field or it should be first if the protocol version field is absent. In case the protocol version field is absent from the string, users can call the **valueOf()** method that takes the protocol version as a parameter; otherwise **valueOf()** that only takes string can be called. The following are valid strings that can be converted into a message:

`"8=FIX.4.1 35=D 11=OrderX 55=MSFT 54=1 15=USD 38=1000 40=1 21=1 59=3 "`
or
`"35=D 11=OrderX 55=MSFT 54=1 15=USD 38=1000 40=1 21=1 59=3 "`

And these are invalid strings:

`"11=OrderX 55=MSFT 54=1 15=USD 38=1000 40=1 21=1 59=3 "` --- no message type
or
`"11=OrderX 55=MSFT 54=1 15=USD 38=1000 40=1 21=1 35=D 59=3 "` --- message type is not first

To use **ELTMessage's valueOf()** methods one will do:

```
      String msgString =
       "8=FIX.4.1 35=D 11=Ord1 55=MO 54=1 15=USD 38=1000 40=1 21=1 59=3 ";
      ELTMessage m = ELTMessage.valueOf(msgString);
```

or

```
      String msgString =
       "35=D 11=Ord1 55=MO 54=1 15=USD 38=1000 40=1 21=1 59=3 ";
      ELTMessage m = ELTMessage.valueOf(msgString, "FIX.4.1");
```

2.2.3 Sample Code

The application can perform the following steps in order to send a **FIX Allocation** message through the engine. An **Allocation** message is chosen as an example to demonstrate the use of nested repeating subgroups.

1. Create the supergroup and populate it with all the required and necessary conditional fields. Note that there are two ways to create a field group. :

```
// create new empty ELTFieldGroupObject:
// Less efficient but more flexible way.
// It allows to send messages to ANY connection
// regardless of connection's protocol version.
// ELTFieldGroup allocFields = new ELTFieldGroup();
// allocFields.setFieldValue(ELT.fieldMsgType, ELT.msgAllocation);
// More efficient but less flexible way.
// It requires the message to be sent only to connections
// with corresponding protocol version.
// ELTFieldGroup allocFields = new ELTFieldGroup(ELT.msgAllocation,
protocolVersion);

// set Allocation's fields:
allocFields.setFieldValue(ELT.fieldAllocID, "Alloc1");
allocFields.setFieldValue(ELT.fieldAllocTransType,
                          ELT.ALLOCTRANSTYPE_NEW);
allocFields.setFieldValue(ELT.fieldSymbol, "CSCO");
allocFields.setFieldValue(ELT.fieldSide, ELT.SIDE_BUY);
allocFields.setFieldValue(ELT.fieldShares, 2000);
allocFields.setFieldValue(ELT.fieldAvgPx, 100.25);
allocFields.setFieldValue(ELT.fieldCurrency, "USD");
allocFields.setFieldValue(ELT.fieldTradeDate, "19990326");
```

2. Let's say we would like this message to contain 2 subgroups representing 2 orders that we need to allocate. Then we need to prepare 2 ELTFieldGroups, and then insert them into the supergroup:

```
// prepare first order subgroup:
ELTFieldGroup order1 = new ELTFieldGroup();
order1.setFieldValue(ELT.fieldClOrdID, "ClOrdId-1");
order1.setFieldValue(ELT.fieldSecondaryOrderID, "X23");

// prepare second order subgroup:
ELTFieldGroup order2 = new ELTFieldGroup();
order2.setFieldValue(ELT.fieldClOrdID, "ClOrdId-2");
order2.setFieldValue(ELT.fieldSecondaryOrderID, "X24");

// add subgroups to the supergroup:
allocFields.addFieldGroupAs(ELT.fieldNoOrders, order1);
allocFields.addFieldGroupAs(ELT.fieldNoOrders, order2);
```

3. Let's say we also need one subgroup representing allocation. We need to set it up in the same way we set up the order subgroups above, but since the allocation subgroup can itself have its own subgroups, we would have to do a little more work:

```
// prepare allocation subgroup:
ELTFieldGroup alloc = new ELTFieldGroup();
alloc.setFieldValue(ELT.fieldAllocAccount, "Account123");
alloc.setFieldValue(ELT.fieldAllocShares, 2000);

// prepare first misc fee subsubgroup:
ELTFieldGroup fee1 = new ELTFieldGroup();
fee1.setFieldValue(ELT.fieldMiscFeeAmt, 12.25);
fee1.setFieldValue(ELT.fieldMiscFeeCurr, "USD");
fee1.setFieldValue(ELT.fieldMiscFeeType, ELT.MISCFEETYPE_MARKUP);

// prepare second misc fee subsubgroup:
ELTFieldGroup fee2 = new ELTFieldGroup();
fee2.setFieldValue(ELT.fieldMiscFeeAmt, 10.25);
fee2.setFieldValue(ELT.fieldMiscFeeCurr, "USD");
fee2.setFieldValue(ELT.fieldMiscFeeType, ELT.MISCFEETYPE_REGULATORY);
```

```
// insert fee subsubgroups into allocation subgroup:
alloc.addFieldGroupAs(ELT.fieldNoMiscFees, fee1);
alloc.addFieldGroupAs(ELT.fieldNoMiscFees, fee2);

// finally insert allocation subgroup into supergroup:
allocFields.addFieldGroupAs(ELT.fieldNoAllocs, alloc);
```

4. At this point, we have completely filled the ELTFieldGroup object with fields required by the FIX Allocation message. To send the FIX message to a required destination, we can either pass the ELTFieldGroup to the proxy’s sendMessage() call:

   ```
   proxy.sendMessage(UserTicket usrTicket,
                     ELTConnectionName connName,
                     Object m)
   ```

   like this:

   ```
   proxy.sendMessage(GlobalSystem.getUserTicket(),
                     new ELTConnectionName(“ClientConnAtoB”),
                     allocFields);
   ```

   or we can call the ELTMessageFactory ourselves to create the message first and then send it through the dispatcher:

   ```
   // create empty message of the right type:
   ELTMessage msg = ELTMessageFactory.createMessage(ELT.msgAllocation,
                                                    "FIX.4.1");

   // assign ELTFieldGroup to it:
   msg.giveFields(allocFields);

   // send the message through proxy:
   proxy.sendMessage(GlobalSystem.getUserTicket(),
                     new ELTConnectionName(“ClientConnAtoB”),
                     msg);
   ```

   In the first case the engine itself will create the message in the correct format according to the protocol version of the connection. In the second case, we need to specify the protocol version ourselves and make sure that it matches the protocol version of the connection.

The fields specifying the number of repeating subgroups in this example (NoOrders, NoAllocs, NoMiscFees) are never explicitly set. These fields will be automatically set to the correct value based on how many subgroups are added to the **supergroup**.

In this example, constants from the class ELT (package **com.inforeach.eltrader**) are used everywhere in place of the message type identifiers and field tag numbers. Also, constants are used to specify field values in case the FIX fields can have only a limited value set. All constants of the ELT class are described in the reference. It is recommended that they be used instead of actual values because it will limit the number of application modifications due to underlying protocol changes.

2.2.4 Send Message API

-
  ```
  public void sendMessage(UserTicket usrTicket,
                          ELTConnectionName connName,
                          Object m)
      throws ELTConnectionUnknownException,
             ELTConnectionUnavailableException,
             ELTNotConnectedException,
             ELTInvalidMessageException,
             PersistentStorageException,
             SecurityException
  ```

  Applications use this method to send a prepared FIX message(s) on the 'connName' connection. The application can pass the object of type **ELTMessage**, or type **ELTFieldGroup**, or the **String** that represents the FIX message as a parameter *m*. If the **ELTFieldGroup** or **String** is sent as a parameter, then it's internally converted into an **ELTMessage** object with the correct sender and target values dictated by the connection's properties. If the **ELTMessage** is sent as a parameter, then its sender and target values will still be overwritten with the appropriate ones. The **UserTicket** parameter may be passed for logging and security checks and can be null.

  NOTE: When the **ELTFieldGroup** is created, it can be populated with any set of field values. When it is passed to the **sendMessage()** method before the message is actually sent on the wire to the counter party, all the fields that do not belong to the message according to its meta-data are stripped off. However, the application itself receives all field values in the message event when the engine notifies the application that the message was sent. Also, all the fields will be logged in the persistent storage.

-
  ```
  public void sendMultipleMessages(UserTicket usrTicket,
                                   ELTConnectionName connName,
                                   Object[] m)
      throws ELTConnectionUnknownException,
             ELTConnectionUnavailableException,
             ELTNotConnectedException,
             ELTInvalidMessageException,
             PersistentStorageException,
             SecurityException
  ```

  This method is the same as above, the only difference being that the engine proxy can be instructed to send an array of messages in a preset order instead of a single message.

  NOTE: When the **ELTFieldGroup** is created, it can be populated with any set of field values. When it is passed to the **sendMessage()** method before the message is actually sent on the wire to the counterparty, all fields that do not belong to the message according to its meta-data are stripped off. However, when the engine notifies the application that the message was sent, the application receives all the field values in the message event. Also, all the fields will be logged in the persistent storage.

-
  ```
  public void postInternalMessage(UserTicket usrTicket,
                                  ELTConnectionName connName,
                                  Object m)
      throws ELTConnectionUnknownException,
             ELTConnectionUnavailableException,
  ```

```
        ELTInvalidMessageException,
        PersistentStorageException,
        SecurityException
```

Applications use this method to post a prepared FIX message on the connection '**connName**'. The subscribers listening to the message flow on this connection are notified about the message, but the message itself isn't sent. Applications can use this method to simulate received messages without actually reading them from the wire. As with the sendMessage() call, type **ELTMessage**, **ELTFieldGroup** objects, or **String** can be passed as parameter *m*. The difference is that if a type **ELTMessage** object is passed by the applications, it will not be altered in before notifying the subscribers.

-
  ```
  public void postMultipleInternalMessages(UserTicket usrTicket,
                                           ELTConnectionName connName,
                                           Object[] m)
      throws ELTConnectionUnknownException,
             ELTConnectionUnavailableException,
             ELTInvalidMessageException,
             PersistentStorageException,
             SecurityException
  ```

  This method is the same as the one above, the only difference is that the engine proxy can be instructed to post an array of messages in a preset order instead of a single message.

-
  ```
  public void sendAdminMessage(UserTicket usrTicket,
                               ELTConnectionName connName,
                               Object m)
      throws ELTConnectionUnknownException,
             ELTConnectionUnavailableException,
             ELTNotConnectedException,
             ELTInvalidMessageException,
             PersistentStorageException,
             SecurityException
  ```

  This method should be used by applications to send administrative message. If the application needs to post an administrative message to listeners, then the same **postMultipleInternalMessage** can be used for application messages.

  NOTE: When the **ELTFieldGroup** is created it can be populated with any set of field values. When it is passed to the **sendMessage()** method before the message is actually sent on the wire to the counter party, all fields that do not belong to the message according to its meta-data are stripped off. However, when the engine notifies the application that the message was sent, the application receives all the field values in the message event. Also, all the fields will be logged in the persistent storage.

-
  ```
  public void resendMessages(UserTicket usrTicket,
                             ELTConnectionName connName,
                             Date startingTime,
                             Date endingTime,
                             ELTMessageBoolExpr filter)
      throws ELTConnectionUnknownException,
             ELTConnectionUnavailableException,
             ELTNotConnectedException,
             PersistentStorageException,
  ```

```
            SecurityException
```

Connections that have DB logging enabled can be instructed to resend all messages that were sent or posted through it between a specified start-time and end-time. This is necessary during application recovery if the message replay to listeners is needed. It's also useful for message flow recovery between counterparties during critical malfunctions. The filter parameter can be used to filter the set of messages that will be replayed.

The `ELTMessageBoolExpr` class can be used by simply calling its constructor with string representing a Boolean expression as a parameter. The elements of the expression will be simple Boolean operators on FIX field names or their combination. For example the following call would create a filter that can filter out all messages of type 'F' or 'G':

```
 ELTMessageBoolExpr filter = new ELTMessageBoolExpr
                                ("MsgType <> 'F' OR MsgType <> 'G'");
```

This call will filter out all messages that were sent before the required time:

```
ELTMessageBoolExpr filter = new ELTMessageBoolExpr
                         ("SendingTime > '20010314-14:35:46'");
```

In all the cases where filtering is not required, *null* can be passed in place of the filter.

2.2.5 API for Snapshot Monitoring of FIX Connections

-
  ```
  public boolean isConnected (UserTicket userTicket,
                              ELTConnectionName connectionName)
      throws ELTConnectionUnknownException,
             ELTConnectionUnavailableException,
             SecurityException
  ```

This method can be used to check if the FIX connection is still alive on both TCP/IP and FIX session levels.

-
  ```
  public boolean pingConnection(UserTicket userTicket,
                                ELTConnectionName connectionName
                                int timeout)
      throws ELTConnectionUnknownException,
             ELTConnectionUnavailableException,
             SecurityException
  ```

Use this method to send a test request, and instruct the engine to wait for a given timeout heartbeat. If the heartbeat or any other message arrives from the counterparty within this timeout, this method returns **true**; otherwise, it returns false.

-
  ```
  public ELTConnectionState getConnectionState
                                  (UserTicket userTicket,
                                   ELTConnectionName connectionName)
      throws ELTConnectionUnknownException,
             ELTConnectionUnavailableException,
  ```

```
            SecurityException
```

This method can be used to obtain the full state of the connection. The `ELTConnectionState` class and its methods are described in the on-line reference.

2.2.6 API for Retrieving Logged Messages

Applications can use a proxy API to retrieve messages logged by connections that had their DB logging enabled. In all the following methods, *null* can be passed in place of a filter for all cases where filtering is not required.

-
  ```
  public ELTAppMessage[] retrieveMessages(UserTicket usrTicket,
                                          ELTConnectionName connName,
                                          Date startingTime,
                                          Date endingTime,
                                          ELTMessageBoolExpr filter,
                                          bool convertToPublishFormat)
      throws ELTConnectionUnknownException,
             ELTConnectionUnavailableException,
             PersistentStorageException,
             SecurityException
  ```

  Use this method to retrieve all messages that were sent or posted on the connection between the start-time and end-time. The filter parameter can be used in the same manner as in the `resendMessages()` method. The `convertToPublishFormat` parameter controls whether the messages will be returned to the caller in the same format in which they were persisted (wire format), or in the format expected by the application's listener components (publish format). If both wire and publish formats are the same, no conversion will be performed (see section 3.2).

-
  ```
  public ELTAppMessage[] retrieveMessages(UserTicket usrTicket,
                                          ELTConnectionName connName,
                                          int sentRecievedFlag,
                                          Date startingTime,
                                          Date endingTime,
                                          ELTMessageBoolExpr filter,
                                          bool convertToPublishFormat)
      throws ELTConnectionUnknownException,
             ELTConnectionUnavailableException,
             PersistentStorageException,
             SecurityException
  ```

  The difference between these two methods is that the second method accepts a sent or received flag as a parameter. If this flag is equal to one (1), then only sent messages will be retrieved and then filtered. If it is equal to zero (0), then only received messages will be retrieved and then filtered. The `convertToPublishFormat` parameter controls whether the messages will be returned to the caller in the same format in which they were persisted (wire format), or in the format expected by the application's listener components (publish format). If both wire and publish formats are the same, no conversion will be performed (see section 3.2).

-
  ```
  public ELTAppMessage[] retrieveMessages(UserTicket usrTicket,
                                          ELTConnectionName connName,
                                          int sentRecievedFlag,
                                          int lowerBoundMessageId,
                                          int upperBoundMessageId,
                                          ELTMessageBoolExpr filter,
                                          bool convertToPublishFormat)
      throws ELTConnectionUnknownException,
             ELTConnectionUnavailableException,
             PersistentStorageException,
             SecurityException
  ```

  All messages that go through the **InfoReach** engine are assigned network-wide unique message IDs that are independent of FIX-required sequence numbers. These IDs allow applications to detect gaps in communication, and try to recover lost messages (see section 6.3 for a full description). Applications can use this method to recover messages with message IDs within a given range. The `convertToPublishFormat` parameter controls whether the messages will be returned to the caller in the same format that they were persisted (wire format), or in the format expected by the application's listener components (publish format). If both wire and publish formats are the same, no conversion will be performed (see section 3.2).

-
  ```
  public ELTAppMessage[] retrieveMessages(UserTicket usrTicket,
                                          ELTConnectionName connName,
                                          int sentRecievedFlag,
                                          int messageIds,
                                          ELTMessageBoolExpr filterб
                                          bool convertToPublishFormat)
      throws ELTConnectionUnknownException,
             ELTConnectionUnavailableException,
             PersistentStorageException,
             SecurityException
  ```

  Same as above only instead of a message ID range, it takes an array of required message IDs as its parameter.

-
  ```
  public ELTMessage[] retrieveMessagesForResend(UserTicket userTicket,
                                         ELTConnectionName connectionName,
                                         int beginSeqNum,
                                         int endSeqNum,
                                         ELTMessageBoolExpr filter)
        throws ELTConnectionUnknownException,
               ELTConnectionUnavailableException,
               SecurityException,
               PersistentStorageException
  ```

  Retrieves <u>sent</u> messages within a specified sequence number range. These messages have their poss dup flag set to 'Y'; this ensures that they can be resent by the application. This method should be used by applications when the resend requests are handled by the business level, and not by the engine.

-
  ```
  public Object[][] retrieveMessageLogData(UserTicket userTicket,
                                        ELTConnectionName connectionName,
                                        int sentReceivedFlag,
                                        String selectClause,
                                        String whereClause,
                                        String advancedClauses)
        throws ELTConnectionUnknownException,
               ELTConnectionUnavailableException,
               SecurityException,
               PersistentStorageException
  ```

  Allows execution of arbitrary advanced engine log queries. This can include aggregate functions and clauses such as: math functions, GROUP BY clauses, HAVING clauses, ORDER BY clauses, and others.

The complete **InfoReach** engine proxy API is provided in the on-line reference.

2.3 Listening to Message Flow

All application level components that need to be registered with the engine proxy as event listeners have to implement the **IEventListener** interface. This interface has only one method:

```
void onEvent(Object event);
```

The example application in ***Appendix A: Sample FIX*** Message Processing Application shows how implementation of the **onEvent()** method should be structured in order to handle different kinds of events. The events that can be passed from **InfoReach** engines to this method are :

1. ELTAdminMessageEvent --- will contain the administrative message that can be accessed by calling the getMessage() method:

   ```
   ELTAdminMessage adminMessage =
                          ((ELTAdminMessageEvent)event).getMessage();
   ```

2. ELTAppMessageEvent --- will contain the application message that can be accessed by calling the getMessage() method:

   ```
   ELTAppMessage appMessage =
                          ((ELTAppMessageEvent)event).getMessage();
   ```

3. ELTConnectionStateEvent --- will contain the ELTConnectionState object representing the full state of some connection. Connections publish their state during critical points of their lifetime such as logon, disconnect, error, etc. The ELTConnectionState object can be obtained from this event by calling the getConnectionState () method:

   ```
   ELTConnectionState state =
                ((ELTConnectionStateEvent)event).getConnectionState();
   ```

The engine proxy API contains two methods to subscribe and unsubscribe the application level components as event listeners:

-
  ```
  public void subscribe(UserTicket usrTicket,
                        IEventListener eventListener,
                        XMLData listenerProperties)
  ```

  When the application calls this method, it should pass the component object that will listen to the message flow in place of the '**eventListener**' parameter. The '**listenerProperties**' parameter will contain the list of connection names and the type of events in which the listening component is interested. For example, if a component, **MessageListener**, needs to be registered with the engine proxy, '**EngineProxyForApplicationX**,' and is interested in both application and administrative FIX messages going through connections '**ClientConnAtoB**' and '**ServerConnAtoB**', then the application can make this sequence of calls:

  ```
  // setup listenerProperties XMLData object:
  XMLData listenerProps = new XMLData();

  // only interested in these connections:
  listenerProps.setAttributeValue("connectionNames",
                                  "ClientConnAtoB, ServerConnAtoB");

  // interested in FIX application messages:
  listenerProps.setAttributeValue("subscribeForAppMsgs", true);

  // interested in FIX admin messages:
  listenerProps.setAttributeValue("subscribeForAdminMsgs", true);

  // now subscribe with engine proxy:
  proxy.subscribe(GlobalSystem.getUserTicket(),
                  MessageListener,
                  listenerProps);
  ```

  The full list of listener properties is given in the table below.

Listener Properties	
Property	**Description**
connectionNames	String containing a comma separated list of connection names. These connections should be visible to the application through the engine proxy.
subscribeForAppMsgs	Valid values are "**true**" or "**false**". If "**true**" then the listening component will be notified of the FIX application messages going through the connections (depending on connection settings sent or received or all messages can trigger the notification process).
subscribeForAdminMessages	Valid values are "**true**" or "**false**". If "**true**" then the listening component will be notified of the FIX admin messages going through the connections (depending on

	connection settings sent or received or all messages can trigger the notification process).
subscribeForConnInfo	Valid values are "**true**" or "**false**". If "**true**" then the listening component will be notified of the connection events (e.g. connect, disconnect, etc.).
subscribeDirectly	Valid values are "**true**" or "**false**". Default value is set to inversed value of 'useSendingQueue' attribute of the connection to which listener subscribes.

<MessageFilter>	This element is applicable only for non-direct listeners (i.e. attribute subscribeDirectly=false should be set in order to use filtered subscription).
	This element can have one of the two allowed attributes:
	- msgTypes - coma separated list of message types. E.g. msgTypes="D,F,G"
	- condition – a boolean expression based on message fields. If it evaluates to true message will be published to this listener, otherwise, they will be filtered out. E.g. condition="MsgType = 'D'"
	
	Example of setting up a filetered subscription via API:
	
	`// Create listener's properties`
	`XMLData listenerProperties_ = new XMLData();`
	
	`listenerProperties_.getDocumentRoot().setNodeValue("ListenerProperties");`
	
	`listenerProperties_.setAttributeValue(ELTEngineDefs.CFG_SUBSCRIBE_DIRECTLY, false);`
	
	`listenerProperties_.setAttributeValue(ELTEngineDefs.CFG_SUBSCRIBE_FOR_APP_MSGS, true);`
	
	`listenerProperties_.setAttributeValue(ELTEngineDefs.CFG_SUBSCRIBE_FOR_ADMIN_MSGS, true);`
	
	`// Create filter's properties.`
	`XMLData filterProperties_ = listenerProperties_.createNewElement(ELTEngineDefs.CFG_MESSAGE_FILTER);`
	
	`listenerProperties_.getDocumentRoot().appendChild(filterProperties_);`
	
	//If you need to use BoolExpr then do:
	filterProperties_.setAttribute(ELTEngineDefs.CFG_MESSAGE_FILTER_CONDITION, "Field='Value'");
	
	// OR, if you just want to filter by message types (which is much faster then BoolExpr filter):
	filterProperties_.setAttribute(ELTEngineDefs.CFG_MES

	SAGE_FILTER_MSG_TYPES, "D,F,G");

-
  ```
  public void unsubscribe(UserTicket usrTicket,
                          IEventListener eventListener,
                          XMLData listenerProperties)
  ```

  Use this method to unsubscribe listeners from specific connections and event types. For example, if the **MessageListener** component does not require notification of events happening on connection **ServerConnAtoB** and is not interested in any FIX administrative messages at some point during run time, then the following sequence of calls can be made:

  ```
  // setup listenerProperties XMLData object:
  XMLData unsubscribeProps = new XMLData();

  // not interested in these connections any more:
  unsubscribeProps.setAttributeValue("connectionNames",
                                     "ServerConnAtoB");

  // not interested in FIX admin messages any more:
  unsubscribeProps.setAttributeValue("subscribeForAdminMsgs", true);

  // now unsubscribe with engine proxy:
  proxy.unsubscribe(GlobalSystem.getUserTicket(),
                    MessageListener,
                    unsubscribeProps);
  ```

Upon receiving a message inside the event, the application can examine several custom fields that the engine populates in order to provide additional information about the message. These fields include:

- Tag 16536 --- this message field contains the message format string name. Currently it can be "FIX", "FIXML" or empty (for "FIX"). Use the **ELT.fieldMessageFormat** constant to access this field.

- Tag 16575 --- this message field contains the character 'Y' if this message was received from a counterparty and it will be not set or set to 'N' if this message was sent or posted locally. Use the **ELT.fieldIsFromCounterParty** constant to access this field.

- Tag 16576 --- this message field contains the connection string name through which this message was received, sent or posted. This field is extremely useful for applications processing message flow. Use the **ELT.fieldConnectionName** constant to access this field.

- Tag 16580 --- this message field contains the connection-wide unique integer message id. This id should be persisted by applications that use **ELTGuaranteedMessageListener**. Use the **ELT.fieldConnectionMessageId** constant to access this field.

- Tag 16581 --- this message field contains the engine network-wide unique integer message ID. Use the **ELT.fieldGlobalMessageId** constant to access this field.

- Tag 16584 --- this received message field contains a string explanation of errors encountered at message validation time. The application may examine this field every time it is notified of the received message. It does this in order to ensure received message validity. Use the **ELT.fieldMessageErrors** constant to access this field.

- Tag 16585 --- indicates that the received message has sequence number greater then expected. See **`maxOutOfSequnceMessagesBeforeDisconnect`** connection parameter for further information. Use **ELT.fieldMessageErrors** constant to access this field.

- Tag 16589 --- optional field which can be considered as connection-wide unique sequence number (unlike FIX sequence numbers it is direction-independent). All messages regardless of the direction (incoming or outgoing) will have this field set to unique continuously-incremented value. In other words, it can be used to sort incoming and outgoing messages in the same order as they were processed by the system Without using this field there is no way to do it because FIX incoming and outgoing sequence numbers are maintained independently and messages can be exchanged at a very high rate so sending time field (correct up to the millisecond) cannot guarantee the order of the messages. Available since FIX Engine 6.2. Usage of this field can be disabled via -**DuseConnectionUniformMessageId** option (by default 'true'). Use **ELT.fieldConnectionUniformMessageId** constant to access this field.

NOTE: *There may be cases when it is desirable to process different engine events on separate thread, for example by subscribing several listening components. Each listener is responsible for its own set of events, and each has a different **subscribe()** call. **InfoReach FIX Engine** notifies each subscribed component on a separate thread. Hence, concurrency of event processing may be completely controlled by application developers through manipulating the number of subscribed components. One component can listen to multiple connections' flows. Alternately, there can be one component per connection, or multiple components responsible for different events of the same connection. This note is true for subscribers which have attribute ELTEngineDefs.CFG_SUBSCRIBE_DIRECTLY set to false in their properties or when attribute ELTEngineDefs.CFG_SUBSCRIBE_DIRECTLY is omitted and given connection uses sending queue (i.e. connection should not have in its properties attribute* `useSendingQueue="false"`*).*

2.4 Using IELTEngineProxyActivationHandler

If an application needs to subscribe or do certain initialization/uninitialization procedures after the engine is already initialized but before it starts operating, a custom implementer of **com.inforeach.eltrader.engine.IELTEngineProxyActivationHandler** should be registered with the engine proxy using the proxy's **setEngineProxyActivationHandler()** API call before calling **ELTEngineComponent.initialize(…)**. None of the engine API should be used prior **calling ELTEngineComponent.initialize(…)**. But after this call the engine is already operational, i.e. schedules are setup and task could be running, server connections start accepting clients and message flow could be started, etc. Often it's critical to be able to subscribe for messages right before engine is activated and it can be done in **IELTEngineProxyActivationHandler** implementer.

IELTEngineProxyActivationHandler's methods:

`void onPreEngineProxyActivation();`

This method is called right before the engine proxy gets activated. It's guaranteed that at this time local connections are not activated yet (i.e. any scheduled tasks have not run and server engine sockets are not accepting client connections yet). Application should subscribe with the engine here.

`void onPostEngineProxyActivation();`

The method is called right after the engine proxy gets activated.
If engine proxy is a part of a fault tolerance cluster successful call of **ELTEngineComponent.initialize(…)** does not guarantee that proxy is activated – it could be in stand-by mode, waiting for a primary (current active member of the cluster) to fail.

`void onPreEngineProxyDeactivation();`

This method is called whenever proxy is deactivated.
When an engine proxy is a member of a fault tolerance cluster due to some conditions, its process might be deactivated by engine manager or it might exit by itself in case connection with engine manager is lost.
At this point engine API is available for an application layer and it can, for example, unsubscribe from the engine.

`void onPostEngineProxyDeactivation();`

This method is called whenever proxy is deactivated.
When engine proxy is a member of a fault tolerance cluster, its process might be deactivated by engine manager or it might exit by itself in case connection with engine manager is lost.
At this point engine API is not available for an application layer.

There is an abstract class com.inforeach.eltrader.engine.util.ELTAbstractEngineProxyActivationHandler which might be used to simplify the process of implementing proxy activation handler.
Below is an example how it might be used:

```
        ELTEngineProxy.setEngineProxyActivationHandler(new
ELTAbstractEngineProxyActivationHandler()
            {
               public void onPreEngineProxyActivation()
               {
                  // your initialization code
                  // Do not put it right after ELTEngineComponent.initialize(…);
               }
               public void onPostEngineProxyDeactivation()
               {
                  // your uninitialization code
        }
    });
 ELTEngineComponent.initialize(…);
```

See samples/java/FIXClientServer/SampleClient.java for complete example.

2.5 Putting server-mode connection on hold

Sometimes it's needed to change minor connection properties (e.g. sender ID, target ID, message validation flag, configure message preprocessor, etc) without shutting down other connections. One can use **reloadConfig** command in ELTService process but configuration changes would not be applied for active connections. In order to be able to apply changes for client-mode connections, they can be simply disconnected (either via API or using **EngineView** GUI tool). However this is not the case for server-mode connections which accept incoming client connections. In order to restrict server-mode connections from connecting clients, the connections can be **PutOnHold**. Once a server-mode connection is **PutOnHold**, it stops accepting any incoming client connections and sends **Logout(MsgType=5)** message with field **Text(58)** set to the reason for the connection being on hold.
This can be achieved either via **EngineView** GUI tool or via API of **ELTEngineProxy** class:

```
public void putOnHold(
    UserTicket        userTicket,
    ELTConnectionName connectionName,
    String            reason)
    throws ELTConnectionUnknownException,
           ELTConnectionUnavailableException,
           SecurityException
```

Puts connection on hold and sends **Logout(MsgType 5)** message to the connected clients with field **Text(58)** set to the **reason**.

```
public String getHoldReason(
    UserTicket        userTicket,
    ELTConnectionName connectionName)
    throws ELTConnectionUnknownException,
           ELTConnectionUnavailableException,
           SecurityException
```

Returns a description for on-hold reason for a given connection.

```
public void removeFromHold(
    UserTicket        userTicket,
    ELTConnectionName connectionName)
    throws ELTConnectionUnknownException,
           ELTConnectionUnavailableException,
           SecurityException
```

Removes connection from hold.

```
public boolean isOnHold(
    UserTicket        userTicket,
    ELTConnectionName connectionName)
    throws ELTConnectionUnknownException,
           ELTConnectionUnavailableException,
           SecurityException
```

Returns true if connection is on hold.

2.6 Using IELTMessageProprocessor

When all messages for a given connection need to be adjusted in some way, a custom implementer of **com.inforeach.eltrader.engine.IELTMessagePreprocessor** should be provided. For example, this can be useful when a counterparty's custom protocol requirements cannot be expressed via InfoReach FIX Engine's protocol meta-data. Custom implementer can be registered for any connection by setting additional attribute in **<Connection>** element:

```
engineEventPublisherPreprocessor="com.vendor.XXXMessagePreprocessor"
```

IELTMessagePreprocessor interface provides callbacks which are invoked as certain actions occur inside the engine. See java docs for more details.

com.inforeach.eltrader.engine.util.ELTBaseMessagePreprocessor class is an abstract class intended to simplify implementing custom message preprocessors which can be configured connection.

CAUTION: It is prohibited to send/post messages inside IELTMessagePreprocessor's method implementations.

2.6.1 Deafult implementation of IELTMessagePreprocessor

Since version 6.2 default, rule based implementation of the IELTMessagePreprocessor interface included in the standard package. It can be configured via ELTEngineNetworkConfig GUI tool. In enginenetwork.xml file these rules can be specified right in the connection properties in separate element called **`MessagePreprocessorRules`**. This element can have such optional sub elements:

- **`PreprocessSentAppMessageRule`**
- **`PreprocessReceivedAppMessageRule`**
- **`PreprocessPostedAppMessageRule`**
- **`PreprocessSentAdminMessageRule`**
- **`PreprocessReceivedAdminMessageRule`**
- **`PreprocessBeforeSendingRule`**
- **`PreprocessAfterReceivingRule`**
- **`PreprocessAfterRetrievalRule`**

Each of those sub elements can have a list of rules which will be applied to message at the corresponding time. The general XML format of a rule is

```
<Rule
      [bool expression]
```

```
       [invert = "true"|"false"]
>
      [operation1]
          … … …
       [operationN]
</Rule>
```

There are multiple formats for expressions and also several different operations that can be performed (see below). The *invert* attribute reverses the result of the expression. It is false by default.

Expression Formats

- Empty expression: unconditional rule.
  <Rule>

- Single field - single value matching expression. Examples (expression is highlighted):
  <Rule ***field="fieldName" value="value"*** >
  <Rule ***field="fieldName" value="value" invert="true"*** >

- Single field – multiple values matching expression. Example:
  <Rule ***field=" fieldName" values="value1|value2|…|valueN"*** >
  <Rule ***field=" fieldName" values="value1|value2|…|valueN" invert="true"*** >

- General regular expression. Example:
  <Rule **expression =** ***"fieldName1 = 'value1' AND (fieldName2 <> 'value2' OR NOT fieldName3 > 'value3')"*** >
  The general regular expressions in the rules are less efficient than special matching rules and therefore special matching rules should be used whenever possible.

- Custom expression evaluation class. Example:
  <Rule ***expressionImplClass="com.somepackage.SomeClassName"*** >
  The custom class name can be used to evaluate the expression. The class should implement interface *IDefaulterRuleCondition* with one method
  boolean evaluate(IDefaulterDataProvider provider). The entity to which the rule is being applied is passed to this method as a data provider. The implementation can thus examine current fields in the entity and return true or false to indicate whether the rule should be applied.

Operations

***Set* operations**:

- Setting single field to a specific value:
  <Set field="fieldName" value="value" />

- Setting formatted string value:
  <Set field="fieldName" formattedText="formatted text pattern" />
  The formatted text pattern can include field values with $() notation. For example, pattern "Order qty $(OrderQty)" will result in the field value being set to "Order qty 1000" if the OrderQty field of the entity was set to 1000.

- Setting single field to a value returned by custom value provider:
  <Set field="fieldName" valueProviderImplClass=" com.somepackage.SomeClassName" param="param string" />
  The custom class name can be used to calculate the value. The class should implement interface *IFieldValueProvider* with one method
  Object getValue(Object fieldId, IDefaulterDataProvider provider, Object param). The param string is passed to this method and also the entity to which the rule is being applied is passed to this method as a data provider. The implementation can thus examine current fields in the entity and return calculated value. The ***param*** attribute is optional.

- Setting single field to a value returned by custom XML-configured value provider:
  <Set field="fieldName" valueProviderImplClass=" com.somepackage.SomeClassName" param="param string" >
  <AdancedParams>
  </AdvancedParams>
  </Set>
  The custom class name can be used to calculate the value. The class should implement interface *IFieldValueConfigurableProvider* with two methods
  Object getValue(Object fieldId, IDefaulterDataProvider) and *void initialize(Object param, XMLData advancedParams).* The param string is passed to initialize() method and also the XMLData object representing the <AdvancedParameters> element is passed to the initialize method. The implementation can thus examine current fields in the entity and return calculated value based on the parameters. The ***param*** attribute and the <AdvancedParameters> element are optional.

- Setting single field to a value calculated by an expression:
  <Set field="fieldName" expression="math expression" />
  For example, <Set field="OrderValue" expression="OrderQty * Price" />

All set operations support optional "override" attribute. It allows specifying whether the operation can actually change the value of the field and not just set the value of the empty field. By default the set operations do not override existing values. If an operation should override a value even if it is already set then the "override" attribute should be explicitly set to "true". For example:
<Set field="fieldName" value="value" override="true"/>.

***Copy* operations:**

- Copying value of one field into another field:
  <Copy field="fieldName1" from="fieldName2" />
  <Copy field="fieldName1" from="fieldName2" override="true" />

***Clear* operations:**

- Clear the value of the field:
  <Clear field="fieldName1" />

***Check* operations:**

Check operations are used for validation of entities. Presence or absence of fields, fields values, values that have to be based on other fields may be validated. If validation fails the specified error message or warning message will be displayed/logged. Failed check rules do not preclude other rules from being evaluated and applied.

- Checking if field is set:
  <Check field="fieldName" errorMessage = "error text"/>

- Check if specific field has specific value:
  <Check field="fieldName" value="value" errorMessage = "error text"/>

- Check if specific field has one of the values:
  <Check field="fieldName" values="validValue1|…|validValueN" errorMessage = "error text"/>

- Check if specific field has one of the values returned from the custom valu provider:
  < Check field="fieldName" validValuesProviderImplClass =" com.somepackage.SomeClassName" param="param string" />
  The custom class name can be used to calculate the valid values list. The class should implement interface *IFieldValidValuesProvider* with one method *String[] getValidValues(Object fieldId, IDefaulterDataProvider provider, Object param).* The param string is passed to this method and also the entity to which the rule is being applied is passed to this method as a data provider. The implementation can thus examine current fields in the entity and return calculated values. The ***param*** attribute is optional.

- Check if specific field has one of the values returned from the custom valu provider:
  <Check field="fieldName" validValuesProviderImplClass =" com.somepackage.SomeClassName" param="param string" />
  <AdancedParams>
  </AdvancedParams>
  </Check>
  The custom class name can be used to calculate the valid values list. The class should implement interface *IFieldValidValuesProvider* with two methods *String[] getValidValues(Object fieldId, IDefaulterDataProvider* and *void initialize(Object param, XMLData advancedParams).* The param string is passed to initialize() method and also the XMLData object representing the <AdvancedParameters> element is passed to the initialize method. The implementation can thus examine current fields in the entity and return calculated value based on the parameters. The ***param*** attribute and the <AdvancedParameters> element are optional.

All check operations support optional “warning” attribute. It allows specifying whether the failed check rule should result in the error message or in the warning message (error is default). For example:
<Check field="fieldName" value="value" errorMessage = “text” warning = “true”/>.

***Error* operations:**

The check operations are normally used by unconditional rules. The conditional rules that need to post an error message as an action use error operation which just a kind of empty check rule:

- Sendng an error message:
  <Error errorMessage = “error text”/>

For example:

<Rule
  expression = “OrderQty > 10000”
>
        <Error errorMessage = “OrderQty is too large”/>
</Rule>

Example:

```
    <Connection ….>
        <MessagePreprocessorRules>
            <PreprocessSentAppMessageRule>
                <Rule>
                    <Set field="Account" value="MyAccount#" override="true" />
                </Rule>
            </PreprocessSentAppMessageRule>
            <PreprocessSentAdminMessageRule>
                <Rule expression="MsgType = &apos;2&apos;">
                    <Set field="EndSeqNo" value="0" override="true" />
                </Rule>
                <Rule expression="MsgType = &apos;4&apos;">
                    <Copy field="OrigSendingTime" from="SendingTime"
override="false" />
                </Rule>
        </PreprocessSentAdminMessageRule>
</MessagePreprocessorRules>
```

3 Engine Network Deployment

3.1 Overview

There are a variety of engine deployment options that accommodate the available hardware, licensing, and system requirements for message throughput. Engine deployment can support any number of FIX connections, on any number of machines. You can do this by setting the necessary parameters in the engine network configuration file. Engine network configuration consists of a set of XML files with self-describing XML tags. They can be edited manually, or modified through the **Engine Network Configuration Tool** (see the User's Guide for procedures). **InfoReach** initialization routines use the information that's contained in these files whenever they are called by higher-level Java components.

Conceptually, deploying InfoReach FIX engines for communicating message flow with counter parties can be described in these steps:

1. Configuring FIX connection properties including connection name, mode (client or server), company ID, target company ID, protocol version, and more.
2. Grouping configured connections by their mode, company ID, and company sub-ID by associating connections with configured engines.
3. Assigning engines to be **local** engines to some engine proxy. Each user process can instantiate one engine proxy and thus all engines assigned to this proxy as **local** will be instantiated in this process. Distribution is achieved by associating FIX connections with engines that are assigned as **local** to different engine proxies. The connections are established and supported in different processes; they could even run on separate machines. Each process can listen to the event flow from FIX connections instantiated in another process. This is achieved by assigning engines of the other process as **remote** to this process' engine proxy. Each engine in the network can only be local to one proxy; but it can be remote to any set of other proxies in the network.
4. Configuring processes to instantiate one engine proxy with given name. The engine proxy will be instantiated with name read from configuration when the client application calls the ELTEngineComponent.initialize(…) .
5. Starting the processes on dedicated machine(s).

3.2 InfoReach FIX Connection Properties

Each FIX connection has its own configuration file section that contains its properties. FIX connections could be one of three types: **'client'**, **'server'** or **'loopback'**. FIX **client** connections initiate the logon sequence and therefore must have properties for the IP and port of the server they are connecting to. Each client connection section contains a **ServerList** section. The section describes each server the connection can logon to. Besides the IP and port properties of the counterparty server, the client connection can contain properties for the local IP and port through which the client connection will be initiated. This is useful in case the machine hosting the client engine has multiple IPs. If these local bind properties are not specified the host machine default IP will be used. Loopback connections do not have a physical connection with a counterparty, and will only be used by applications to simulate message flow. Listing 2 shows a sample

ConnectionList section that contains properties for a client connection named '**ClientConnAtoB**', a server connection named '**ServerConnAtoB**', and a loopback connection, '**AtoBConnSimulator**'. **ClientConnAtoB** can connect to a primary or secondary server with the name '**ServerB**'. Notice that all property values are in double quotes:

```
<ConnectionList>

<Connection
      name = "ClientConnAtoB "
      connectionMode = "client"
      companyId = "ClientA"
      counterCompanyId = "ServerB"
      autoReconnect = "false"
      protocolVersion = "FIX.4.1"
      validateMessages = "false"
>
      <ServerList>
      <Server
            name = "primary"
            clientRemoteAddress = "199.16.21.916"
            clientRemotePort = "8081"
            clientLocalAddress = "LocalIPorHostName"
            clientLocalPort = "330"
            maxConnectAttempts = "2"
      ></Server>
      <Server
            name = "secondary"
            clientRemoteAddress = "199.16.21.917"
            clientRemotePort = "8081"
            maxConnectAttempts = "2"
      ></Server>
      </ServerList>
</Connection>

<Connection
      name = "ServerConnAtoB"
      connectionMode = "server"
      companyId = "ServerA"
      counterCompanyId = "ClientB"
>
      <ClientList>
              <Client ipAddress="196.222.111.*" port="*" />
      </ClientList>
</Connection>

<Connection
      name = "AtoBConnSimulator"
      connectionMode = "loopback"
      companyId = "ClientA"
      counterCompanyId = "ServerB"
></Connection>

</ConnectionList>
```

Listing 2.

Table 1 contains the full list of properties for client and server FIX connections. Common properties appear first in the list. Additional properties for client FIX connections are listed under their corresponding headings:

Table 1: FIX Connection Properties

Properties Common to the Client and Server FIX Connection	
Property	**Description**
name	Unique connection name.
companyId	This appears in the **SenderCompID** field of every message sent on this connection
companySubId	This appears in the **SenderSubID** field of every message sent on this connection.
counterCompanyId	This appears in the **TargetCompID** field of every message sent on this connection.
counterCompanySubId	This appears in the **TargetSubID** field of every message sent on this connection
connectionMode	Mode of the connection. Valid values are "**client**", "**server**", or "**loopback**"
protocolVersion	FIX protocol version that's "spoken" over this connection.
heartbeatInterval	FIX's millisecond heartbeat interval (see FIX protocol for details). **NOTE:** It's not recommended to set a long heartbeat interval since heartbeats can trigger recovery from out-of-sequence situations. The interval, however, should not be too short. If it is, heartbeats may flood the communication.
reasonableTransmissionTime	This is a reasonable transmission time parameter that can be set per connection (in milliseconds). For example, if it is set to 3 sec., and heartbeat is 30 sec., the engine will think that something is wrong after 33 sec. passes since the last received message. This parameter should usually be set to smaller numbers.
logoutTimeout	Maximum interval in milliseconds to wait for confirmation of the logout.

releasePendingMessagesOnConnect	Valid values are "**true**" or "**false**". If set to "**true**" then all the messages sent by the application will be recorded even if a physical connection to the counterpart is not established. Recorded messages will be released as soon as the connection is established
publishPendingAppMessages	Valid values are "**true**" or "**false**". If set to "**true**" then the engine listeners will be notified of outgoing messages before messages are added to the sending queue in the engine
publishSentAppMessages	Valid values are "**true**" or "**false**". If set to "**true**" then engine listeners will be notified of outgoing messages after the messages are sent on this connection. Note that if both **publishPendingAppMessages** and **publishSentAppMessages** are set to "**true,**" then the engine listeners would be notified twice.
resendInEngine	Valid values are "**true**" or "**false**". If set to "**true**" then the engine implementing this connection will reply to the FIX resend request. **NOTE:** Engine database persistence must be enabled to support this behavior. If it's not, the engine will send a fill gap message. If this property is set to "**false**" then the engine will reply with a gap fill, and the application is responsible for retrieving necessary messages from the log, and deciding which messages it wants to resend. In this case, the Proxy's **retrieveMessagesForResend()**method can be used. **WARNING: The application has to handle the resend request with care. It must not resend messages that can potentially be out of order with other messages that it sent previous to the resend request notification. If this happens it's because the counter party is not aware that the resent messages are a reply to its resend request since the engine already sent a gap fill message before.**
disconnectOnDBError	If the connection is configured to log messages into the database, then this connection can be instructed to terminate if there is a db logging error. This can be useful if it is absolutely critical that everything that goes though this connection gets logged

disableDBLogging	If **MessageLogFacility** is configured for connection, this connection can be instructed not to log a message by setting this flag to "**true**". By default this flag is "**false**", however no logging will be done with this flag set to "**false**" if message log facility for this connection is not configured.
publishFormat	The format of the messages sent to application components listening for event flow. Currently supported values for this property are "FIX" and "FIXML". By default it is "FIX". Automatic conversion will be done by the **InfoReach** engine between FIX and FIXML format if necessary.
wireFormat	The format of the messages to be sent over the wire to the counter parties. Currently supported values for this property are "FIX" and "FIXML". By default it is "FIX". Automatic conversion will be done by the **InfoReach** engine between FIX and FIXML format if necessary
messageLogFacility	This is a sub-element which itself consists of multiple properties. It is described in section 3.2.1.
pendingQueueThreshold	When the connection message volume is very high, there is a possibility that the size of the engine's 'send' queue will grow beyond the allotted memory requirement. The **pendingQueueThreshold** property can be set to limit the size of the send queue. If the engine attempts to place a message in the queue after the threshold size was reached, the message will be written to persistent storage and will be read later; after the queue size is reduced. The optimal value for this property is contingent on the available hardware and the message volume that is sent on the connection.
connectionResetTimes	This is a semicolon-separated list of times. The times values represent when this connection should be reset. All times should be in GMT HH:MM:SS format. Example: **connectionResetTimes="9:41:00;19:41:30"** **InfoReach** engine automatically sends a logon message with reset sequence number flag set to '**Y**' at the **connectionResetTimes** times.
validateIncomingMessages	Valid values are "**on**", "**off**" or "**requiredOnly**" Default value is "**requiredOnly**". Controls the validation of incoming FIX message.

validateOutgoingMessages	Valid values are "**on**", "**off**" or "**requiredOnly**" Default value is "**on**". Controls the validation of outgoing FIX message.
watchActivity	Valid values are "**true**" or "**false**". Default value is "**false**". This parameter is very useful for troubleshooting. Very often if processes are running in production there is nobody watching it and if for some reason (e.g. firewall keeps socket connection, or DB process the query too long time) the FIX connection disconnects, nobody can make thread dump from the process, which is the only way to find out real cause of such problems. To solve it, the **InfoReach** engine has the ability to make thread dumps programmatically and if the 'watchActiviy' attribute is set to 'true,' then if no messages were sent/received during time of 2*heartbeatInterval seconds then engine does 3 consequent thread dumps with an interval of 2 seconds.
forceDisconnectionIfInactive	Valid values are "**true**" or "**false**". Default value is "**false**". NOTE: applicable only if watchActiviy=true. This option works in ensemble with 'watchActivity'. When 'watchActivity' is set to 'true' engine monitors messages going through connection in order to determine that connection is working as it supposed to. When no activity detected then activity watcher component just cases JVM to print out threads dump in order to help diagnose the cause of the situation. But it does not affect connection state. So, if connection gets stuck on network call or DB call engine still considers that connection as 'live' one and will reject all attempt to connect it (if it is client connection) and will reject all attempts to reconnect (if it is server mode connection). This parameter instructs activity watcher component to disconnect connection forcedly in order to make connection available again.
useSendingQueue	Valid values are "**true**" or "**false**". Default value is "**true**". Specifies whether the connection will use the sending queue or not. When the connection doesn't use the sending queue, then there will be no ability to work with the connection 'off line', i.e. there will be no pending messages. The only way an application can send messages to such connections is when connection is up.

endSeqNo41	Valid values are "**true**" or "**false**". Default value is "**false**". Valid for connections which use protocol versions higher then 4.2 (including). Some counterparties have old engines that implement ResendRequest message handling according to FIX.4.1 specifications while using messages from FIX4.2 and which require BeginString (8) field to be set to FIX.4.2. For such situations, this flag has to be set to 'true'. By default it is 'false'.
requestResendTillInfinity	Valid values are "**true**" or "**false**". Default value is "**true**". Some counterparties can handle ResendRequest(msgType=2) messages only with EndSeqNo(tag 16) set to the infinity marker (i.e. 999999 for FIX protocol prior 4.1 and 0 for FIX versions starting from FIX 4.2).
maxOutOfSequnceMessagesBeforeDisconnect	Default value: 100 This parameter allows to force the automatic FIX connection disconnection in case it goes into the infinite cycle of Resend Requests. This use case is not covered by the FIX specification and therefore this behavior might not be compatible with other engines. In event when the ResendRequest is lost or a reply to it is lost all incoming messages with seqNum greater than the engine expects are marked with special field IsOutOfSequence (tag 16575, constant ELT.fieldIsFromCounterParty) set to 'Y'. When max. number of out-of-sequence messages is received then the connection disconnects. Manual intervention is required.
cme1xMode	Valid values are "**true**" or "**false**". Default value is "**false**". CME 1.x introduced several features which are not FIX compliant. 1. Sending 2nd TestRequest if there was no reply for the first one. - unlike standard FIX the CME FIX REQUIRES us not to terminate the session in case the first TestRequest was unanswered but send a second one and terminate session only if that 2nd one was not answered. 2. Sending 2nd ResendRequest. - unlike standard FIX the CME FIX REQUIRES second

	ResendRequest to be sent in case correct message with sequence number greater then we were excepting was received after sending first ResendRequest.
limitIncomingSpeed	Measured in msgs/sec. Valid values: any number greater then 0. Default value: 0 (no limit). It guarantees that the number of messages received by this connection at any given second will be no more than the specified number.
limitOutgoingSpeed	Measured in msgs/sec. Valid values: any number greater then 0. Default value: 0 (no limit). It guarantees that the number of messages sent by this connection at any given second will be no more than the specified number.
ResendRequestController	This is a sub-element which itself consists of multiple properties. It is described in section 3.2.2.
ReleasePendingOnConnectController	This is a sub-element which itself consists of multiple properties. It is described in section 3.2.3.
addDebugInfoToMessages	Valid values are "**true**" or "**false**". Default value is "**false**". If attribute is "true" and some messages were sent/received through connection then sender/receiver thread name, thread id and message's instance hashCode will be set in user defined field "ELTDebugInfo".
Additional Properties for the Client FIX Connection	
logonTimeout	Maximum interval in milliseconds to wait for confirmation of the logon.
connectToLastUsedServer	Valid values are "**true**" or "**false**". If set to "**true**" and multiple servers are specified in the **ServerList** section of the client connection, then it can be forced to connect to the server with which the last connection was established instead of the first one in the list.
autoReconnect	Valid values are "**true**" or "**false**". If set to "**true**" it will try to reconnect automatically after a connection was dropped.

connectTimeout	Time in milliseconds within which the connection should be established with any of the servers from the **ServerList** before giving up.
maxConnectAttempts	Number of times to retry to logon to each of the target servers in the server list.
serverList	Connection sub-element containing multiple sub-elements describing the counterparty's servers to which this client connection can logon. See below for properties describing the servers.
connectionIntervals	This is a list of semicolon-separated pairs of **startConnection**, **endConnection** times. It establishes and/or terminates FIX connections at specific times without manual intervention. All times should be in GMT HH:MM:SS format **Example:** **connectionIntervals = "9:41:00,19:41:30;19:42:00,none"** The start and end times are separated by a comma. If disconnect is not automatic, specify '**none**' in place of the disconnect time.
Properties for a Counter Party Server of a Client FIX Connection (target server)	
name	Name of the target server (should be unique within the **ServerList** section)
clientRemoteAddress	Target server IP address.
clientRemotePort	Target server IP port.
clientLocalAddress	If host machine has multiple IPs, use this property to specify that the client socket connection is initiated from the desired IP. If this property is not specified, the socket connection will be initiated from the host's default IP.
clientLocalPort	This property can be specified to make sure client socket connection is initiated from the desired port. If this property is not specified the socket connection will be initiated from the anonymous port
maxConnectAttempts	Number of times to retry logging on to this target server

Properties of Server-mode FIX Connection for client IP filtering (Client element of ClientList)	
ipAddress	IP mask for allowed clients. (e.q. 225.225.*.*)
port	Port mask for allowed clients. (e.q. 123*)

3.2.1 MessageLogFacility

Each FIX connection can persist FIX messages that pass through it into the database. The **MessageLogFacility** section that contains all database logging properties can be specified separately for each connection, or as a default for all connections associated with same engine. Also, when applicable, it can be specified as a default for all connections in the engine network The connection **MessageLogFacility** settings have precedence over settings that are specified for an engine that the connection is associated with, which in turn has precedence over the default engine network **MessageLogFacility**. Listing 3 shows the connection section for the connection 'ServerConnAtoB' rewritten from Listing 2 so that it has its own MessageLogFacility section:

```
<Connection
      name = "ServerConnAtoB"
      connectionMode = "server"
      companyId = "ServerA"
      counterCompanyId = "ClientB"

      <MessageLogFacility
            logLevel = "2"
            databaseAlias = "sybase"
            incomingTableNamePrefix     = "In_"
            outgoingTableNamePrefix     = "Out_"
            pendingTableNamePrefix      = "Pending_"
            incomingTableDBConnectionId = "1"
            outgoingTableDBConnectionId = "1"
            pendingTableDBConnectionId  = "1"
            useSingleDBConnection       = "true"
      ></MessageLogFacility>

></Connection>
```

Listing 3.

Table 2 describes the **MessageLogFacility** section properties :

Table 2: MessageLogFacility Properties

MessageLogFacilityProperties	
Property	**Description**

logLevel	Depending on the log facility's *log_level* property, different message types may be omitted from being written into the database. 4 log levels are currently supported: • *log_level "0":* Messages are not persisted. **NOTE:** The FIX Engine will not be able to respond to resend requests if this log level is specified • *log_level "1":* All messages except heartbeats and test requests are persisted. This is the **recommended** level since it does not require the engine to store information that is not required for either recovery or FIX session management. Even though heartbeats and test requests are not stored, their sequence number is remembered and persisted so that the session can be correctly recovered. • *log_level "2":* same as *log_level "1".* It is reserved for future use • *log_level "3":* all messages including heartbeats and test requests are persisted
databaseAlias	This is the name of the database resource described in the **GlobalSystem's** configuration. The database resource description in the G**lobalSystem's** configuration will contain the database's URL, JDBC driver name, userId, password, etc.
incomingTableNamePrefix	All FIX messages persisted into the database by the log facility are, by default, recorded into a table named '**MessageLog**'. For efficiency it's sometimes necessary to split the load across several tables. If the '**incomingTableNamePrefix**' property is specified, then the incoming messages on the FIX connection will be recorded into a separate table whose name is prefixed with the value of this property (e.q. In_MessageLog)
outgoingTableNamePrefix	If the '**outgoingTableNamePrefix**' property is specified, then the outgoing messages on the FIX connection will be recorded into a separate table whose name is prefixed with the value of this property (e.q. **Out_MessageLog**)
pendingTableNamePrefix	If the '**pendingTableNamePrefix**' property is specified, then the outgoing messages on the FIX connection not sent (because the connection was down) will be recorded into a separate table. The name is prefixed with the value of this property (**e.q. Pending_MessageLog**)

incomingTableDBConnectionId	This is a numeric id for the dedicated connection used for accessing the incoming messages table out of the pool of dedicated connections of the database resource. For example if the database resource's '**numOfDedicatedConnections**' property was set to **5**, then there will be **5** dedicated connections with IDs **1** through **5**, and any of them can be used here. If this property is not specified, then the next connection out of the pool of available non-dedicated connections is used.
outgoingTableDBConnectionId	This is a numeric id of the dedicated connection used for accessing the outgoing messages table out of the pool of dedicated connections of the database resource. If this property is not specified, then a random free connection out of the pool of available non-dedicated connections is used
pendingTableDBConnectionId	This is a numeric ID of the dedicated connection to be used for accessing the pending messages table out of the pool of dedicated connections of the database resource. If this property is not specified, then a random free connection out of the pool of available non-dedicated connections is used
useSingleDBConnection	Valid values are "**true**" or "**false**". If set to "**true,**" then the same connection out of the pool of non-dedicated connections is always used whenever a non-dedicated connection is required. If set to "**false,**" then the next connection is picked out of the pool each time.
implementationClass	This property can be used when it is necessary to substitute the default log facility with the custom one. Then the value for this property should contain the name of the class that implements interface **IELTMessageLogFacility**, for example: `implementationClass="com.brokerX.engine.LogMsg”` When this property is specified, then the rest of the properties become irrelevant since they are only used by the default log facility. The custom log facility should take care of configuring itself with appropriate parameters. One can choose to extend auxiliary class **ELTMessageLogSubstitute** and overwrite only necessary methods instead of implementing the **ELTMessageLogFacility** interface from scratch.
Index	This is an optional sub-element for index creation in incoming, outgoing and pending messages tables. A MessageLogFacility can have a few indexes. See below for properties describing the index.
Properties for an Index of a MessageLogFacility	

fields	Indexable fields in following format: **"field1,field2,..."**.
indexName	Name of the created index. This attribute is optional. If it is not specified index name is generated in following format: **"field1_field2_..._fieldN_tableName_idx"** In PostgreSQL index name must be unique across DB, so this attribute should not be specified to have it autogenerated with different name for incoming/outgoing/pending tables.

3.2.2 ResendRequest controller

Sometimes business requires that some messages not be resent if connection was down for some time. For this purpose, the engine has the ability to control which messages will be resent and which not.

```
<ResendRequestController>
      <MessageFilter
            condition="NO MsgType IN ( &apos;D&apos;,&apos;G&apos;)"
      />
</ResendRequestController>
```

The example above will just not resend messages 'D' and 'G'.

See Table 3: `ResendRequestController` Properties for full details.

Table 3: `ResendRequestController` Properties

Property	**Description**
`disabled`	Valid values are "**true**" or "**false**". Default value is "**false**".
`implementationClass`	If not specified, default implementation is used, otherwise new instance of specified class will be instantiated. The class has to implement the com.inforeach.eltrader.message.filter.IELTMessageFilter interface. By default it is not specified (i.e. default implementation is used).
MessageFilter - internal element. Configure default implementation.	
`condition`	Boolean expression that will be applied to every message. Only messages that satisfy this expression will be resent/released.
`failedAction`	Valid values are "**postInternalReject**" or "**doNothing**". Default value is "**doNothing**". If a message failed, the 'condition' default implementation will not resent/release it. Also, it has the ability to notify application

	about the fact that it was not resent/released by posting an internal reject (fix message 8).

3.2.3 Release Pending Messages On Connect Controller

The 'Release Pending Messages on Connect' controller can filter messages which have to be released after the connection gets connected.

```
<ReleasePendingOnConnectController>
      <MessageFilter
            condition="NO MsgType IN ( &apos;D&apos;,&apos;G&apos;)"
      />
</ReleasePendingOnConnectController>
```

The example above will not release pending messages with types 'D' and 'G'.
Its attributes are absolutely the same as the ResendRequest controller; therefore, for full details see Table 3: `ResendRequestController` Properties.

3.3 InfoReach FIX Engine Properties

The **InfoReach FIX Engine** component is instantiated to host one or more FIX connections. The **AssociatedConnectionList** for each engine will specify the names of all the connections that the engine will implement. The only restriction on the **AssociatedConnectionList** for server-mode engines is:

1. Server-mode engine cannot include connections with the same set of ***{companyId, companySubId, counterCompanyId, counterCompanySubId}*** properties.

Table 4 shows a sample **EngineList** section that contains properties and the **AssociatedConnectionLists** for the client engine '**ClientA'**, server engine '**ServerA'**, and loopback engine '**ClientASimulator'**.

```
      <EngineList>

            <Engine
                  name = "EngineClientA"
                  connectionMode = "client"
                  companyId = "ClientA"
            >
                  <AssociatedConnectionList>
                        <AssociatedConnection
                              name = "ClientConnAtoB"
                        ></AssociatedConnection>
                  </AssociatedConnectionList>
            </Engine>

            <Engine
                  name = "EngineServerA"
                  connectionMode = "server"
```

```
                companyId = "ServerA"
                serverPort = "8081"
                serverAddress = "LocalIPorHostName"
          >
                <AssociatedConnectionList>
                      <AssociatedConnection
                            name = "ServerConnAtoB"
                      ></AssociatedConnection>
                </AssociatedConnectionList>
          </Engine>

          <Engine
                name = "ClientASimulator"
                connectionMode = "loopback"
                companyId = "RES"
          >
                <AssociatedConnectionList>
                      <AssociatedConnection
                            name = "AtoBConnSimulator"
                      ></AssociatedConnection>
                </AssociatedConnectionList>
          </Engine>

    </EngineList>
```

Table 4 contains a full list of properties for the client, server, and loopback FIX engines. Common properties are listed first. Additional properties for the server FIX engines are listed under their corresponding headings:

Table 4: Engine Properties

Properties Common to Client and Server FIX Engines	
Property	**Description**
Name	Unique engine name.
companyId	This has to match the **companyId** property of every connection associated with this engine
companySubId	This has to match the **companySubId** property of every connection associated with this engine
connectionMode	Engine mode. Valid values are "**client**", "**server**", or "**loopback**". **Client** connections can only be associated with a **client** engine, **server** connections with **server** engines, and **loopback** connections with **loopback** engines.
Additional Properties for a Server FIX Engine	
serverPort	Port on which the server engine will listen for incoming logon requests and subsequent messages.

serverAddress	If host machine has multiple IPs, use this property to specify that the server socket is bound to the desired IP. If this property is not specified the socket will be bound to the host's default IP.
Properties for Connections Associated with the Engine	
Name	Connection name.

3.4 *InfoReach Engine Proxies*

Applications using the engine network will never directly interface with InfoReach FIX engine components; instead, they will instantiate the engine proxy and use its interface to communicate with the FIX engine network. Engines are made visible to the applications through the engine proxy by registering the engine's name in the engine proxy's *local engines list* or in the engine proxy's *remote engine list*. Users can specify whether they want to use **RMI** or **CORBA** for communicating with out-of-process engines (engines visible to application through the proxy's remote engine list). The **defaultObjectMiddleware** property can be set to either "**CORBA**" or "**RMI.**" When the application wants to communicate with some engine out of process, this engine should be instantiated on the network by some other process to which this engine is local (belongs to other process's proxy's local engine list).

The sample **EngineProxyList** section in ***Listing 5*** contains entries for two engine proxies. Both have engines "**ClientA**" and "**ServerA**" registered as accessible through them; however, "**ClientA**" is in-process for proxy "**EnginesForApplicationX**" and "**ServerA**" is in-process for "**EnginesForApplicationY**".

```
<EngineProxyList>

      <EngineProxy
            name = "EnginesForApplicationX"
      >
            <LocalEngineList>
                  <Engine name="ClientA"
            </LocalEngineList>

            <RemoteEngineList>
                  <Engine name="ServerA"
            </LocalEngineList>

      </EngineProxy>

      <EngineProxy
            name = "EnginesForApplicationY"
      >
            <LocalEngineList>
                  <Engine name="ServerA"
```

```
                </LocalEngineList>

                <RemoteEngineList>
                      <Engine name="ClientA"
                </LocalEngineList>

          </EngineProxy>

      </EngineProxyList>
```

Listing 5

Given the above configuration, FIX engine "**ClientA**" will be instantiated in one process where that instantiates proxy "**EnginesForApplicationX**", and engine "**ServerA**" will be instantiated in another process (possibly on different machine) with proxy "**EnginesForApplicationY.**" Both engines, however, will be visible to any both processes instantiating either of the proxies--one engine will run in the same process as the process itself and the other will be available through the middleware.

3.5 Global System Configuration

Besides engine network configuration, which controls engine network deployment parameters, the global system configuration is required to identify a license key resource, database resources, and an output logging resource available in the system.

3.5.1 Database Resource List

Table 1 shows a sample database resource list that contains a description of two database resources. Because the **InfoReach** components use JDBC for database communication, heterogeneous databases can be used. All that is required for a successful database connection is the name of the JDBC driver and database URL.

In addition, **InfoReach** supports flat file persistence. You need to specify the URL in the format "***filedb:****path_to_some_folder*". You do not have to specify user, password and numbers of Dedicated/Non-dedicated DB connections. Flat file persistence can be configured to write to multiple locations simultaneously. This can be achieved by specifying additional URLs in `mirrorUrlList`. Flat file stored FIX Messages are in binary format. You can use *catfiledb utility* tool to dump the content of flat files.

NOTE: `BackupList` is supported only if you use the JDBC based persistence; `mirrorUrlList` is supported only if you use flat file based persistence.

```
<DatabaseList>

        <Database
              alias = "sybase"
              driver = "com.sybase.jdbc.SybDriver"
              url = "jdbc:sybase:Tds:server:1433/recdb"
              user = "admin"
              password = "admin"
              numberOfDedicatedConnections = "3"
```

```
            numberOfNonDedicatedConnections = "5"
        ></Database>

        <Database
            alias = "mssql"
            driver = " weblogic.jdbc.mssqlserver4.Driver"
            url = "jdbc:weblogic:mssqlserver4:recdb@server:1433"
            user = "admin"
            password = "admin"
            numberOfNonDedicatedConnections = "1"
        ></Database>

        <Database
            alias = "mssqlWithBackup"
            driver = " weblogic.jdbc.mssqlserver4.Driver"
            url = "jdbc:weblogic:mssqlserver4:recdb@server:1433"
            user = "admin"
            password = "admin"
            numberOfNonDedicatedConnections = "1"
            reconnectAttempts= "2"
            reconnectInterval= "1000"
        >
            <BackupList>
                   <Backup
                         url = "jdbc:weblogic:mssqlserver4:recdb@bk1:1433"
                         user = "admin"
                         password = "admin"
                         reconnectAttempts = "3"
                         reconnectInterval = "500"
                   />
                   <Backup
                         url = "jdbc:weblogic:mssqlserver4:recdb@bk2:1433"
                         user = "admin"
                         password = "admin"
                         reconnectAttempts = "3"
                         reconnectInterval = "500"
                   />
            </BackupList>
        </Database>
        <Database
            alias="filedb"
            url="filedb:../../data"
        >
            <mirrorUrlList>
                 <mirrorUrl url="filedb:../../data" />
                 <mirrorUrl url="filedb:c:\backup" />
            </mirrorUrlList>
        </Database>

</DatabaseList>
```

Listing 6.

The following table describes all the attributes that have to be set for a particular database resource:

Table 5: Database Resource Attributes

Database Resource Properties	
Property	**Description**
Alias	Unique name of the resource.
Driver	Java class name of the JDBC driver used to access this database.
url	Driver-specific URL of the database.
User	Valid DB user with read/write permissions.
Password	Password of the DB user.
numberOfDedicatedConnections	Number of DB connections that will be dedicated for specific tasks. These connections will have IDs 1 through N. Components using this resource will specify the ID of dedicated connection to use for their specific database operations. NOTE: this parameter is ignored by filedb. Also it is ignored if Oracle is used with CLOBs. See special note below about Oracle and CLOBs.
numberOfNonDedicatedConnections	Size of the pool of non-dedicated DB connections. One free connection will be picked from the pool whenever this resource is used for some database operation. NOTE: this parameter is ignored by filedb. Also it is ignored if Oracle is used with CLOBs.See special note below about Oracle and CLOBs.
reconnectAttempts	Number of times the engine will try to establish a connection to the database.
reconnectInterval	Wait time (in milliseconds) between connection attempts.
reconnectOnAnyException	Disconnects from the database upon any kind of (true|false).
dbErrorsForReconnect	The list of comma-separated database error codes that will initiate database connection recovery. Database vendor specific. (e.g. "10021, 100333").
failoverOnAnyException	Forces a disconnect from the current database and initiates connection to the next available backup database in case of any database error. (true | false)

dbErrorsForFailover	The list of comma-separated database error codes that will initiate database connection recovery to a backup database. Database vendor specific.
Backup properties NOTE: Backup database should be of the same type as primary, i.e. if primary database is MSSQL7.0, all backup databases should be also MSSQL7.0	
url	Database URL. JDBC driver-specific.
User	Valid database user with read/write permissions.
Password	Database user password.
reconnectAttempts	Number of database reconnect attempts.
reconnectInterval	Wait time (in milliseconds) between database connection attempts.

Because Oracle does not support TEXT data type by default it is configured to use VARCHAR(4000) for the Message and NonMetaDataFields columns in DB tables. In case 4K is not enough specifying the -DoracleUsesClob=true parameter will cause those columns to be created as CLOBs. When this parameter is specified and Oracle is configured to be used with CLOBs some DB alias attributes are ignored. These attributes are: numberOfDedicatedConnections and numberOfNonDedicatedConnections. If the message log facility is configured to use single DB connection then one new DB connection will be created that can work with CLOB data types. This connection will not be from the pool of non-dedicated DB connection. If different DB connections are configured for in_, out_ and pending_ tables then as many new CLOB-compatible DB connections will be created as needed. E.g. for configuration like [inTableDedicatedConnectionId=1, outTableDedicatedConnectionId=2, pendTableDedicatedConnectionId=3] three new CLOB-compatible DB connectiosn to Oracle server will be created. For configuration like this [inTableDedicatedConnectionId=1, outTableDedicatedConnectionId=1, pendTableDedicatedConnectionId=1] one CLOB-compatible DB connection will be created. For configuration [inTableDedicatedConnectionId=1, outTableDedicatedConnectionId=1, pendTableDedicatedConnectionId=3] two CLOB-compatible DB connections will be created.

3.5.2 OutputLogFacility Resource List

Table 6 shows a sample output log facility list containing the description of one log facility. Because the **InfoReach** components distinguish between error, warning, or debug information, the log facilities can be configured to record different types of logging information into different output destinations by specifying a different 'logger' component for each type. Each logger section contains an output destination and also a prefix to attach to each output record so that log records of different types can be distinguished in case they go to the same output destination.

```
<OutputLogFacilityList>

      <OutputLogFacility
            name = "OutputLogger"
      >
            <ErrorLogger
                  prefix = "*** error: "
                  outputDestination = "errors.out"
                  enabled = "true"
                  outputLevel = "2"
                  timeStampEnabled = "true"
                  threadInfoEnabled = "true"
                  alsoToStandardError = "true"
                  publishToEventService = "true"
            />

            <WarningLogger
                  prefix = "! warning: "
                  outputDestination = "warnings.out"
                  enabled = "true"
                  outputLevel = "2"
                  timeStampEnabled = "true"
                  threadInfoEnabled = "true"
                  alsoToStandardError = "true"
            />

            <DebugLogger
                  prefix = "    debug: "
                  outputDestination = "debug.out"
                  enabled = "true"
                  outputLevel = "2"
                  timeStampEnabled = "true"
                  threadInfoEnabled = "true"
                  alsoToStandardOutput = "true"
            />

            <DefaultLogger
                  prefix = "     info: "
                  outputDestination = "debug.out"
                  enabled = "true"
                  outputLevel = "2"
                  timeStampEnabled = "true"
                  threadInfoEnabled = "true"
                  alsoToStandardOutput = "true"
            />

      </OutputLogFacility>

</OutputLogFacilityList>
```

Listing 7.

Table 6 describes all properties that have to be set for a logger section within the output log facility:

Table 6: Output Log Facility Resource Properties

Output Log Facility Resource Properties	
Property	**Description**
prefix	Text that will be attached at the beginning of each log record of this logger's type made through this log facility.
outputDestination	Name of the resource that will contain log records.
enabled	Valid values are "**true**" or "**false**". If set to "**false,**" log entries of this logger's type will not be recorded.
outputLevel	Level of the log information to be recorded.
timeStampEnabled	Valid values are "**true**" or "**false**". If set to "**true,**" a timestamp will be attached to the log record.
threadInfoEnabled	Valid values are "**true**" or "**false**". If set to "true," the name of the thread making the entry will be attached to the log record.
alsoToStandardOutput	Valid values are "**true**" or "**false**". If set to "**true,**" the record will also be displayed on the standard output (console).
alsoToStandardError	Valid values are "**true**" or "**false**". If set to "**true,**" the record will also be displayed on the standard error (console).

The global system configuration will normally reside in the **system.xml** resource. This resource will be referenced by each process's configuration.

3.5.3 Enabling publishing process information to EngineView tool

In order to make a process publish its process information (like used/free/available memory), a number of threads for such elements should be present in the process's configuration:

```
<GlobalSystem>
      <ProcessOperator
            processInfoPostingInterval = "10"
      />
</GlobalSystem>
```

The '`processInfoPostingInterval`' attribute specifies a time period (in seconds) of how often the process's information will be send out.

3.6 Dynamic Reloading of Configuration

Some changes to the **InfoReach** FIX Engine configuration parameters can take effect at run-time. You can add/delete/modify FIX connections and engines at runtime with some restrictions.

To apply changes you made to FIX engine, type the following command at the ELTService command prompt:
reloadConfig – reload system configuration.

Restrictions applied on 'reloadConfig' command:

- Configurations of connections which are currently in use will not be updated;
- 'server' mode engines properties cannot be changed, although connections to server engine can be added/removed;
- Changing connection's 'useSendingQueue' attribute requires whole system restart to be applied.

4 Starting Up the Engine(s)

Once the engine network configuration is prepared, it has all the FIX engines that could be instantiated on the network described and registered as local to some engine proxy. To instantiate the engine, the application only needs to instantiate this engine's **EngineProxy** by issuing one call in its **main()**:

```
ELTEngineComponent.initialize(bootstrapData,processType,processName);
```

Where:

1. `bootstrapData - xml formatted document which specifies initial process configuration.`

2. `processType – name of the entry in the /Directory/ElTrader/Engine/Config/processes.xml file which will be used as actual process configuration (it's different from initial bootstrap resource).`

3. `processName – the name which will be assigned to the process.` Actual process handle is a composition of processType::processName.

As stated in section 3.1, processes can be instructed to use one of the pre-configured engine proxies by passing specific process names and process types to the `ELTEngineComponent.initialize()` call. The process type and process name parameters are used as keys in order to find the appropriate configuration section in the configuration files. This section will contain the name of the engine proxy that will be instantiated. The sample process section from the processes of type 'ClientApp' and name 'ApplicationX' are shown in Table 7. At every process section, it consists of two sub-sections **GlobalSystem** and **ELTSystem**. At this point it is only important to note that the **ELTSystem** element contains the name of the engine proxy that will be initialized inside this process.

Table 7: Sample Process Section ‘ClientApp’ & ‘ApplicationX’

```
<ClientApp
      name = "ApplicationX”
>
      <GlobalSystem
            configDataResource = "system.xml"
            logFacilityName = "OutputLogger"
      />

      <ELTSystem
            logFacilityName = "OutputLogger"
      >
            <Engine>
                  <EngineProxy name = "EnginesForApplicationX"/>
            </Engine>
      </ELTSystem>
</ClientApp>
```

4.1.1 Process Section Attributes

The header tag for each process section identifies the type of the process (e.q. ‘**ClientApp**’ in Table 7), and it can also contain the attribute for the process’s name and an arbitrary set of attributes that can be used by the application at run time. For example, if the process of type ‘ClientApp’ with name ‘ApplicationX’ will need several configuration parameters to use at run time, they can be added to the process section’s header in the bootstrap file like this:

```
<ClientApp
      name = ”ApplicationX”
      parameter1 = ”value1”
      parameter2 = ”value2”
      ...
      parameterN = ”valueN”
>
```

To access additional parameters specified in the process section header at run time, the application will get a handle to the system’s bootstrap configuration data structure by issuing a call:

```
XMLData bootstrapData = GlobalSystem.getBootstrapConfigData();
```

(It is assumed that call **ELTEngineComponent.initialize()** was already made).

Then, any configuration parameter value stored in the bootstrap Data structure can be retrieved by using **XMLData** class API (**getAttributeXXXValue()** and **getDocumentPart()** methods; see the reference).

4.1.2 GlobalSystem Sub-section

The main purpose of the **GlobalSystem** sub-section is to specify global level parameters, which will be used by the **InfoReach** global level components. The table below contains a description of all attributes that can be placed inside the **GlobalSystem** sub-section for the **InfoReach** components:

GlobalSystem Attributes	
Property	**Description**
configDataResource	Global system configuration resource name. This resource will contain lists of the database resources and log facility resources.
logFacilityName	Name of the log facility to be used by the application level components. The log facility must be one of the log facilities described in the **OutputLogFacility** list of the **GlobalSystem** configuration

4.1.3 ELTSystem Sub-section

The main purpose of the **ELTSystem** sub-section is to hold the description of the engine proxy that will be instantiated by the process during initialization phase.

In addition to the engine proxy description, the **ELTSystem** sub-section will also contain the **logFacilityName** attribute (just like the **GlobalSystem** sub-section) pointing to the log facility that will be used by the **InfoReach** level components.

5 Security and Encryption

5.1 Using Secure Socket Layer

InfoReach FIX Engine can be configured to use secure sockets on a particular connection. Additional information only needs to be specified in the engine network configuration file telling the engine which secure socket factory to use on a particular connection. The configuration of the secure socket factories themselves will reside in the global system configuration.

For client connections, the **<Server>** entry of the client connection must contain the attribute **socketFactoryName**, which references a specific socket factory in the **<SocketFactoryList>** of the **system.xml**, for example:

```
<Connection
    name="ClientC_toBrokerZ"
    ...
>
    <ServerList>
        <Server
            name="BrokerZ"
```

```
                  clientRemoteAddress="localhost"
                  clientRemotePort="8083"
                  ...
                  socketFactoryName="ClientSSL"
              />
          </ServerList>
      </Connection>
```

This means that each client connection can use different socket factory and thus different certificates. For the server connection, the situation is a little different. For server engines, the socketFactoryName attribute is added to the <Engine> element as follows:

```
      <Engine
          companyId="BROKER_Z"
          connectionMode="server"
          name="BrokerZ"
          serverPort="8083"
          socketFactoryName="ServerSSL"
      >
      ...
      </Engine>
```

Thus, all server connections associated with the same engine will use the same secure socket factory meaning that they will use the same certificates.

The following is an example configuration of a couple of secure socket factories:

```
       <GlobalSystem>
            ...
            <SocketFactoryList>
                <SocketFactory
                   name = "ClientSSL"
                   protocolType = "SSLv3"
                   implementation = "SunJSSE"
                >
                   <KeyStore
                      keystoreResource =
        "InfoReach/ElTrader/EngineDemo/cfg/client.keystore"
                      keystorePassword = "password"
                      keystoreType = "JKS"
                   />

                   <SocketParameters
                      cipherSuites =
                           "SSL_DH_anon_WITH_DES_CBC_SHA,
                            SSL_DH_anon_WITH_3DES_EDE_CBC_SHA,
                            SSL_DHE_DSS_WITH_DES_CBC_SHA,
                            SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA,
                            SSL_DH_anon_EXPORT_WITH_DES40_CBC_SHA,
                            SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA,
                            SSL_RSA_WITH_RC4_128_MD5,
                            SSL_RSA_WITH_RC4_128_SHA,
                            SSL_RSA_WITH_DES_CBC_SHA,
                            SSL_RSA_WITH_3DES_EDE_CBC_SHA,
                            SSL_DH_anon_WITH_RC4_128_MD5,
                            SSL_RSA_EXPORT_WITH_RC4_40_MD5,
                            SSL_RSA_WITH_NULL_MD5,
                            SSL_RSA_WITH_NULL_SHA,
                            SSL_DH_anon_EXPORT_WITH_RC4_40_MD5"
                      needClientAuthentication = "true"
                   />
                </SocketFactory>

                <SocketFactory
                   name = "ServerSSL"
                   protocolType = "SSLv3"
                   implementation = "SunJSSE"
                >
                   <KeyStore
                      keystoreResource =
        "InfoReach/ElTrader/EngineDemo/cfg/server.keystore"
                      keystorePassword = "password"
                      keystoreType = "JKS"
                   />

                   <SocketParameters
                      cipherSuites =
                           "SSL_DH_anon_WITH_DES_CBC_SHA,
                            SSL_DH_anon_WITH_3DES_EDE_CBC_SHA,
                            SSL_DHE_DSS_WITH_DES_CBC_SHA,
                            SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA,
                            SSL_DH_anon_EXPORT_WITH_DES40_CBC_SHA,
                            SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA,
                            SSL_RSA_WITH_RC4_128_MD5,
                            SSL_RSA_WITH_RC4_128_SHA,
                            SSL_RSA_WITH_DES_CBC_SHA,
                            SSL_RSA_WITH_3DES_EDE_CBC_SHA,
                            SSL_DH_anon_WITH_RC4_128_MD5,
                            SSL_RSA_EXPORT_WITH_RC4_40_MD5,
                            SSL_RSA_WITH_NULL_MD5,
                            SSL_RSA_WITH_NULL_SHA,
                            SSL_DH_anon_EXPORT_WITH_RC4_40_MD5"
                      needClientAuthentication = "true"
                   />
                </SocketFactory>

            </SocketFactoryList>
       </GlobalSystem>
```

The keystore file referenced by the **keystoreResource** attribute in the **SockectFactory** configuration element is constructed using **keytool.exe** provided by the **JDK**:

a. Create a keystore using a keytool with the -genkey option (as an example).
b. Submit a CRS to a certified CA and obtain a certificate.
c. Import the certificate authenticating the CA's public key.
d. Import the certificate reply supplied by the CA.

The **truststore** that JSSE uses to authenticate received certificates can be set using the system property **-Djavax.net.ssl.trustStore;** otherwise, it will default to the **cacerts** file located in the **$JRE_HOME/lib/security** directory.

5.2 Using PGP-DES-MD5 Encryption Schema

InfoReach FIX Engine supports the **PGP-DES-MD5** message encryption schema described in FIX protocol documentation (**EncryptMethod=5**). Under this schema, the initial logon messages are encrypted with **PGP** and exchange **DES** encryption keys. After that, each message is encrypted with **DES** encryption and validated through use of **MD5** signatures. To use this method, counterparties must exchange the public keys for PGP encryption and decryption. Also, the machine where the engine runs must have **PGP** software. When running the engine process, the path to the **PGP** executable and the environment variables pointing to the **PGP** key ring files should be defined.

For win32-based systems (example):

```
-DpgpProgramPath="D:\Program Files\Network Associates\pgpnt\pgp.exe"
-DUSERPROFILE="C:\Documents and Settings\userX"
```

For unix-based systems (example):

```
-DpgpProgramPath=/usr/local/pgp/pgp
-DPGPPATH=/home/userX/pgp
```

If PGP keys are correctly generated and imported, and the paths are correctly specified, all that remains is to change engine network configuration to indicate that on a particular connection encryption method 5 must be used.

For client connections, the **<Encryption>** section is added inside the connection properties section:

```
<Connection
    name="ClientC_toBrokerZ"
    ...
>
        ...

    <Encryption
         encryptionMethod="5"
         pgpId="fixclient"
         pgpPassword="fixclientpass"
         counterPgpId="fixserver"
    />
</Connection>
```

This means that each client connection can use a different PGP ID, and it can encrypt its logon message for different PGP counterparties. For the server connection, the situation is a little different. For server engines, the **<Encryption>** section is added to the **<Engine>** element as follows:

```
<Engine
    companyId="BROKER_Z"
        ...
>
        ...

        <Encryption
              encryptionMethod="5"
              pgpId="fixserver"
              pgpPassword="fixserverpass"
        />
</Engine>
```

All server connections associated with the same engine use the same **PGP ID** to encrypt their logon reply. Notice that in engine's encryption section the **counterPgpId** property is not specified. This is because multiple clients with different PGP IDs may be connecting to the server connections associated with the same server engine. Upon receiving encrypted logon messages, the server connection deduces the counterparty's **PGP ID** from the message signatures and then encrypts its reply accordingly.

5.3 DES Encryption Schema

InfoReach FIX Engine supports the **DES** message encryption schema described in FIX protocol documentation (**EncryptMethod=2**). Under this schema, all messages are encrypted using DES encryption. In order to use this method, both the client and server **DES** keys must be exactly the same. Therefore, counterparties must first exchange the value of the key via a secure medium.

To use **DES** encryption, you must make the following configuration changes:

1) Use the KeyGenerator tool to generate the DES master key.
KeyGenerator - tool which provides convenient way to generate "not weak" keys for DES.

2) Change the client connection settings and appropriate server engine:

```
<Encryption
        encryptionMethod="2"
        DESMasterKey="abcdef0123456789"
        DESMasterIV="abcdef0123456789"
/>
```

encryptionMethod="2" - will make engine use DES encryption
DESMasterKey - the place where your secret (generated by KeyGenerator tool) key should be placed.
DESMasterIV - "initial vector" - DES specific parameter. It could be any set of 16th characters.

NOTE: DESMasterKey and DESMasterIV parameters have to be the same on both ends of connection.

6 Fault Tolerance and Recoverability

6.1 High Availability

InfoReach's FIX engine library is a robust component that can process a large volume of sent and received messages on multiple connections. In case of an invalid FIX message, the engine components send a FIX reject message and continue their operation. They will also try to skip any '**garbage'** caused by the line noise if it occurs in between FIX messages.

All situations resulting in exceptions are handled gracefully within the engines, while they continue sending and receiving consequent FIX messages without interruption. The engines can be instructed to terminate FIX connections if message logging problems occur. It can be critical to have a trace of everything that went through the connections.

If a connection is lost and the application still tries to send messages on this connection using the engine proxy, the messages are persisted with a specific internal flag. This marks them undelivered. When the broken connection is re-established, the engine sends all undelivered messages in one wave. The sequence number remains intact across the two sessions. This feature can be enabled or disabled for each connection (see section 3.2).

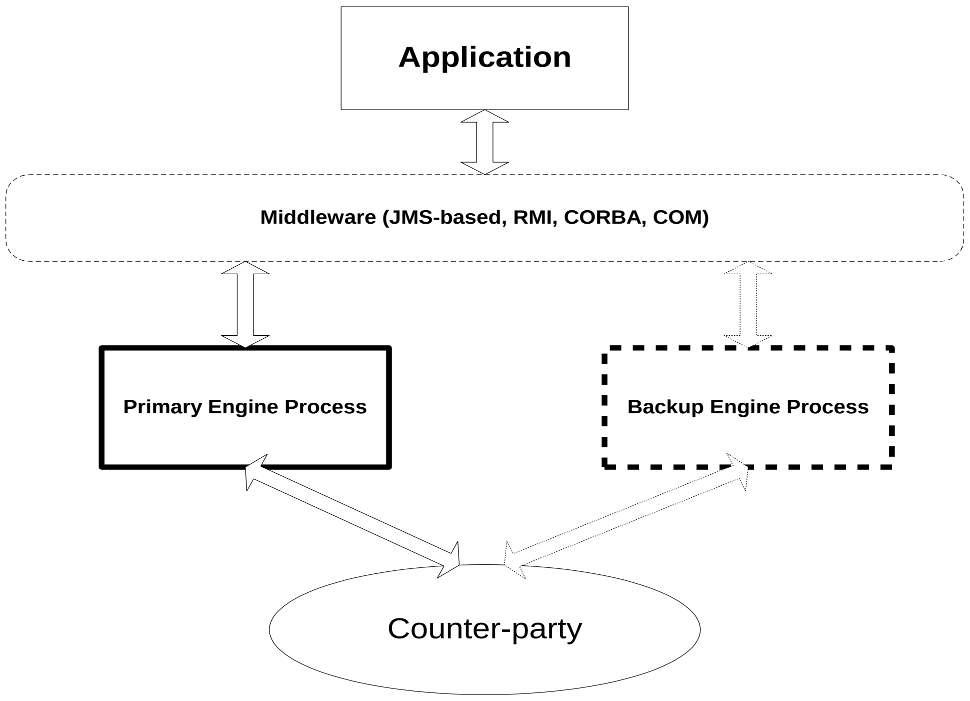
Because the engines can persist all messages into a database, the processes that instantiate the engines can be brought down, and then restarted without any loss of information. All connections initiated after the recovery will start with the correct sequence number after the necessary FIX-level resends and tests are performed.

The situation may occur where an application is unable to process messages, and the engine continues sending and receiving them. In this case, an application needs to restore the correct state with respect to the messages sent or received by the engine. **InfoReach's Engine Proxy API** provides methods that applications can use to restore their state with respect to FIX engine communication (see section 2.2.6).

6.2 Fault Tolerance

If a client application runs in a separate process space than the engine, it's possible to resolve any engine failure so the application isn't effected by service interruptions. The idea is to run a parallel back-up engine process that seamlessly takes over the primary engine process. If the failed engine process contains only FIX client connections, they would be automatically re-connected by the back-up process upon activation. If, however, there were any FIX server connections than this schema, the engine will rely on the ability of the counterparty to reestablish FIX connection with a different IP and port.

Figure 2: Fault Tolerance



InfoReach FIX Engine fault tolerance is achieved by deploying two or more proxies to form a fault tolerant cluster. All proxy members of the cluster must be configured in exactly the same way, each containing the same engines that have the same associated connections. Each cluster member is assigned a ranking starting from zero to determine the fail over succession. Zero is the highest ranking. Each proxy from the cluster should be instantiated by a different process, and all the processes should run at the same time. This ensures the backup process can be instructed to take over the connections of the failed process. Upon system startup, the highest-ranking cluster member that successfully registers with the engine manager is designated as the primary for the cluster--thus, the order in which the processes instantiating the cluster proxies are started may determine which process is designated as the primary one. The designated primary will be the only proxy active at any given time. All other proxies in the cluster will remain deactivated until a failure condition triggers a primary re-assignment. This selection process is performed within the engine manager.

The fault tolerant cluster entry in the engine network configuration looks like this:

```
<FaultToleranceClusterList>
    <FaultToleranceCluster
        name = "Cluster1"
        reasonableProxyDeactivateTime = "5000"
        reasonableProxyActivateTime = "60000"
        reasonableClusterInitTime = "120000"
        reasonableClusterFailoverTime = "120000"
    >
        <EngineProxy
            name = "Primary"
            rank = "0"
        />
        <EngineProxy
            name = "Backup1"
```

```
                rank = "1"
            />
            <EngineProxy
                name = "Backup2"
                rank = "2"
            />
        </FaultToleranceCluster>
    </FaultToleranceClusterList>

    <EngineProxyList>
        <EngineProxy
            name = "Primary"
        >
            <LocalEngineList>
                <Engine
                    name = "TestEngine"
                />
            </LocalEngineList>
        </EngineProxy>

        <EngineProxy
            name = "Backup1"
            proxyToReplicate = "Primary"
        />

        <EngineProxy
            name = "Backup2"
            proxyToReplicate = "Primary"
        />
    </EngineProxyList>
```

There are four cluster attributes that control the fail over behavior:

1. **`reasonableProxyActivateTime`** - This is the maximum time permitted for a proxy to complete the activation process. If a proxy fails to activate during this time, the engine manager considers it faulty. It will deactivate the proxy, and then attempt to activate the next member in the cluster. The default is 2.5 minutes.

2. **`reasonableProxyDeactivateTime`** - This is the amount of time the engine manager will wait for a proxy to deactivate. If the manager determines that the primary of a given cluster has faulted due to an **AppAdmin** ping failure, it will invoke the deactivate method on the proxy before attempting to activate the next designated alternate in the cluster. This minimizes state losses between the old and new primary proxy. The default is five seconds.

3. **`reasonableClusterInitTime`** - During system startup, this is the amount of time the engine manager will wait for all cluster members to register before designating a primary engine proxy. If the engine proxy with the highest possible rank of 0 registers, it will immediately be designated as the primary. Cluster members may still register at any time; however, once a primary is designated it will only be replaced if it un-registers or experiences a failure. A subsequent registration of an engine proxy with a higher rank will not replace the current designated engine proxy. The default is three minutes.

4. **`reasonableClusterFailoverTime`** - This is the amount of time client adapters will wait for the engine network to transition to a new primary engine proxy. The default is zero. This means that it will wait indefinitely.

There are two ways to detect failures. The first is by understanding the polling done by the component that runs inside the **ELTService** process of all cluster processes. When a poll fails, the engine manager is notified and activates the next designated alternate. The second method is by the engine proxy, which monitors itself for failure conditions. For example, if the database logging is specified as critical for any of the proxy's connections, then any database operation failure will cause the proxy to un-initialize the process and un-register from the engine manager.

The engine manager works with an application environment management system (e.q. "Tivoli", "Veritas"). The manager accomplishes this by writing the cluster states to a file. This is an example of the file:

```
<FaultToleranceClusterList>
    <FaultToleranceCluster
        name = "Cluster1"
        activeProxy="Backup1"
    >
        <EngineProxy
            name = "Primary"
            serverAddress="primaryServer"
        />
        <EngineProxy
            name = "Backup1"
            serverAddress="backupServer1"
        />
        <EngineProxy
            name = "Backup2"
            serverAddress="backupServer2"
        />
    </FaultToleranceCluster>
</FaultToleranceClusterList>
```

The option of writing the state of the cluster to the file is configured through the engine manager process's configuration entry with the attribute `clusterStatePersistenceResource`. This attribute contains the resource name in which to write the cluster state.

The engine proxy can notify the application layer that it has activated and deactivated via a handler. The application specific handler will implement the interface **com.inforeach.eltrader.engine.IELTEngineProxyActivationHandler** and is registered with the engine proxy using the proxy's **setEngineProxyActivationHandler()** API call.

6.3 Guaranteed Message Delivery

All messages that go through InfoReach FIX connections are assigned unique network-wide message IDs that are independent of FIX message sequence numbers. These IDs allow applications to detect communication gaps and try to recover lost messages. InfoReach provides a convenience class **ELTGuaranteedEngineMessageListener** that applications can use when they don't require any special processing during message-gap recovery; rather they simply rely on messages replay from the El`Trader logs.

The InfoReach message ID sequence is split into five ranges for different types of messages:

1. Administration messages received from the counterparty.
2. Administration messages sent to the counterparty.
3. Application messages received from the counterparty.
4. Application messages sent to the counterparty.
5. Messages resent to the counterparty as a result of a resend request that it receives from the counterparty.

The maximum and minimum values for each range are enumerated as static constants of **ELTEngineDefs** class. The values are:

1. INITIAL_ADMIN_MESSAGE_FROM_COUNTER_PARTY_ID and MAX_ADMIN_MESSAGE_FROM_COUNTER_PARTY_ID

2. INITIAL_ADMIN_MESSAGE_TO_COUNTER_PARTY_ID and MAX_ADMIN_MESSAGE_TO_COUNTER_PARTY_ID

3. INITIAL_APP_MESSAGE_FROM_COUNTER_PARTY_ID and MAX_APP_MESSAGE_FROM_COUNTER_PARTY_ID

4. INITIAL_APP_MESSAGE_TO_COUNTER_PARTY_ID and MAX_APP_MESSAGE_TO_COUNTER_PARTY_ID

5. INITIAL_RESENT_MESSAGE_ID and MAX_RESENT_MESSAGE_ID

To use the **ELTGuaranteedEngineMessageListener** class applications, it is necessary to construct an object like the one described above with their own **IEventListener** interface implementer as a parameter to this classes' constructor. Instead of registering their own listener component with engine proxy, they would register the newly created **ELTGuaranteedEngineMessageListener** object. This ensures that InfoReach's guaranteed message delivery component processes all message events before they're passed to the application's listener component. If InfoReach's component detects gaps, it will automatically use the proxy's recovery API (see section 2.2.6) to retrieve and replay missing messages. Initially, applications should inform InfoReach's component of the last message ID consumed by this application for each of the message types.
ELTGuaranteedEngineMessageListener provides methods that allow the application to set the last message ID consumed by the application. This is the way the application indicates to the **ELTGuaranteedEngineMessageListener** that for a given connection name and message type it has to monitor the message stream for gaps. If the application does not wish to incur the overhead of guaranteed message delivery for a given connection and message type, it will not call the corresponding **setLastConsumedXXXMessageId()** method. It's the application's responsibility to determine the last consumed message ID in order to set a given connection name and message type. This is because the criterion for message consumption is application specific. Also, some schema for persisting the last consumed message IDs is required on the application level.

After the last message IDs that are consumed have been set for all the connections and message types, the application can be resynchronized in two ways.

- The application can explicitly call **retrievePotentiallyLostMessages()**. All messages in the engine with message IDs greater than the last consumed message ID set for the all the given connection and message types will be synchronously retrieved and sent as updates to the application's own listener.

  Neither the **ELTGuaranteedEngineMessageListener,** nor the application's listener needs to be subscribed with the engine proxy for this method to function.

- Otherwise, **ELTGuaranteedEngineMessageListener** will be registered as a listener and the first message received will trigger this retrieval behavior for the given connection name and message type. It is important to remember that the retrieval behavior will not be performed for connections and message types where the application did not set the last consumed message ID.

If the application does not have the last consumed message ID, it can use the **ELTEngineDefs** constants for the initial values that are enumerated above.

The **ELTGuaranteedEngineMessageListener** detects message gaps by keeping track of the last message ID received for the given connection name and message type. A lost message condition is triggered when the message ID of the received message is not less than or equal to the last consumed message ID plus one. If a message with a message ID smaller than the last consumed message ID plus one is received, a warning is logged but the last consumed message ID is not changed.

When a lost message condition is detected, message consumption is temporarily suspended while a synchronous retrieval of the lost messages is performed. The lost messages are then sent as normal updates (in their correct order) to the application listener. Normal event consumption resumes. Therefore, the application event **IEventListener** is not aware that a message disruption occurred. (See ***Appendix A: Sample*** FIX Message Processing Application for an example.).

7 IBM WebSphere MQ Integration

7.1 Overview

InfoReach FIX Engine can be easily integrated with IBM WebSphere MQ. One queue is used for messages received from the counterparty to be passed to an application. Another queue is used for messages sent from the application.

In order to guarantee message delivery from the application to the counterparty, it's required that a given FIX connection is configured to have the "Release pending on connect" option enabled.

7.2 MQ Messages Format

All MQ queues operate with MQ messages.
Each MQ Message consists of two parts:

1. integer field which contains the length of FIX string
2. array of bytes containing FIX string itself encoded with ASCII symbols

7.3 Configuration

MQ Gate should be configured in the **enginenetwork.xml** file in the appropriate `<EngineProxy>` section. Example:

```
<EngineProxy name="ServerEngineProxy">
    <InwardAdapterList>
    …
    </InwardAdapterList>

    <LocalEngineList>
    …
    </LocalEngineList>

    <RemoteEngineList />

    <MQGate
        host="gorkon"
        channel="S_gorkon"
        qManager="QM_gorkon"
        persisterImpl="jdbc"
        persisterUrl="mssql2000:MSG_IDS"
    >
    <Connection
        name="Server_42"
        readQueue="FIX_IN_1"
        writeQueue="FIX_OUT_1"
    />
    <Connection
        name="Server_41"
        readQueue="FIX_IN_2"
        writeQueue="FIX_OUT_2"
    />
    </MQGate>
</EngineProxy>
```

MQGate attributes		
Property	**Required**	**Description**
host	Y	Name or IP of a host machine where MQ server is running.
channel	Y	Name of a channel to which the gate will be connected.
qManager	Y	Name of the MQ Queue Manager
persisterImpl	N	Implementation of the persister. Persister is responsible to store message IDs of messages successfully posted to the MQ write queue. Possible values: "jdbc" – will store message IDs in the JDBC compliant DB. "file" – will store message IDs either in file system or in the Directory.
persisterUrl	N	Configuration for persister. Format of the configuration depends on "persisterImpl" attribute value: - For "jdbc" it should be in form: <DBAlias>:<tableName> DBAlias – one of the DB descriptions from system.xml file. - For "file" it could be in form: <[file|directory]>://<pathToFile>.
port	N	Listening port of MQ listener. If not specified it's left to MQ library to decide which port to use. (IBM documentation specifies port 1414).
user	N	User name to be used to connect to MQ. If not specified it's left to MQ library to decide what name to use.
password	N	Password to be used to connect to MQ. If not specified it's left to MQ library to decide what password to use.
mqTraceLevel	N	Allows enabling of MQ internal tracing facility. Possible values: 1, 2, 3, 4, 5. If not specified – MQ tracing disabled.
readTimeoutInterval	N	Number of milliseconds to wait on "get" method. Default value is 3000 ms.
writeRetryInterval	N	Number of milliseconds to wait before the next attempt to put message in the queue. Default value is 5000 ms.
writeQueuePingInterval	N	Number of milliseconds in ping interval. This attribute will be applied to error queue also. Some firewalls can close network connections which were idle for a long time. To prevent it, this attribute was

		introduced. Implementation details: method of the queue "getInhibitPut()" is used to ping the queue. Default value is 0 ms.

MQGate's Connection attributes		
Property	**Required**	**Description**
name	Y	Name of the FIX connection
readQueue	Y	Name of the queue FROM which MQ gate will get messages for FIX connection.
writeQueue	Y	Name of the queue TO which MQ gate will post messages from FIX connection.
errorQueue	N	Name of the queue TO which MQ gate will post notifications about errors.
persisterImpl	N	See description for it in <MQGate> element documentation. NOTES: - if it's not specified on the <Connection> level, then settings from <MQGate> element will be used. - if it's omitted in both <MQGate> and <Connection> it's treated as severe error.
persisterUrl	N	See description for it in <MQGate> element documentation NOTES: see NOTES for persisterImpl attribute.
handleAppMessages	N	Specifies whether or not FIX application messages should be forwarded to MQ. Possible values: [true|false]. Default value is "true".
correlationId	N	If specified, only messages with such correlationId will be read from READ queue. All the rest will remain on the READ queue. Also, all messages posted to WRITE and ERROR queue will have correlationId set to this value.
handleAdminMessages	N	Specifies whether or not FIX administrative messages should be forwarded to MQ. Possible values: [true|false]. Default value is "false".
readTimeoutInterval	N	Number of milliseconds to wait on "get" method. Default value is specified in parent MQGate element.
writeRetryInterval	N	Number of milliseconds to wait before the next attempt to put message in the queue. Default value is specified in parent MQGate element.
writeQueuePingInterval	N	Number of milliseconds in ping interval. This attribute will be applied to error queue also. Some firewalls can close network connections which were idle for a long time. To prevent it, this attribute

		was introduced. Implementation details: method of the queue “getInhibitPut()” is used to ping the queue. Default value is specified in parent MQGate element.

8 Using InfoReach FIX Engine behind a Firewall

8.1 Overview

InfoReach's FIX Engine can be deployed on servers protected by firewalls with GUI tools accessing and monitoring engines from outside the firewall protection. To achieve this, FIX Engine processes can be configured to use a predefined set of ports that can be opened by a firewall administrator. This configuration consists of a single line where port numbers or port ranges are listed: e.g. "2000, 2001, 2010-2030" specifies that engine should use ports 2000, 2001, 2010, 2011, 2012, …, 2030. Also, system administrator should open one more port which is specified in the **bootstrap.xml** file for ELTService process as an attribute of the <Directory> element, called "initRefProviderPort".
Also, very often a machine which has to be accessed from behind the firewall has more than one network interface (either it has multiple network cards, or it can be accessed via VPN). In such cases, the engine has to know which of the available local IPs to use, expose to firewall. This is controlled by JAVA –D options:

```
set LOCAL_IP=hostLocalIP
set OPTS=%OPTS% -DlocalHost=%LOCAL_IP%
set OPTS=%OPTS% -Djava.rmi.server.hostname==%LOCAL_IP%
```

8.2 Configuring socket pool via bootstrap.xml file

To make a process use socket pool, "socketPoolPorts" attribute should be specified for the <Bootstrap> element in the process's bootstrap.xml file. E.g.

```
<Bootstrap
    bootstrapResource="ElTrader/Engine/Config/processes.xml"
    socketPoolPorts="2000-2100"
…
/>
```

8.3 Configuring socket pool via Java's –D option

Another way to configure a process to use socket pool is via "–DsocketPoolPorts" option. For instance, you can add this line to ELTService.bat:

```
set OPTS=%OPTS% -DsocketPoolPorts="2000-2100"
```

Then ELTService process will use only ports in the range of 2000-2100, plus the initial reference provider port (by default 5678).

9 Utility classes

InfoReach's FIX Engine comes with several utility classes that are implemented on top of the basic API.

9.1 Detecting connected/disconnected FIX connection states

Very often an application just needs to know if a connection is connected and can send messages directly to the counterparty. Such information can be obtained by subscribing to connection state events and analyzing received states. Since a lot of intermediate states get published, it requires more than several simple lines of code to detect when connection actually went down or when it is ready to accept messages for transmitting. The class **ELTConnectionUpDownTrigger** is intended exactly for such situations.
Here is the sample how it could be used:

```
import com.inforeach.eltrader.engine.util.*;

….
            new ELTConnectionUpDownTrigger("YOUR_CONNECTION_NAME")
            {
                protected void up()
                {
                    ELTSystem.getLogFacility().info().write("\n\n !!!!! 
"+getConnectionName()+" is UP !!!!\n\n");
                }

                protected void down(int reason)
                {
                    ELTSystem.getLogFacility().info().write("\n\n !!!!! 
"+getConnectionName()+" is DOWN. Reason: "+getDefaultReasonName(reason)+" 
!!!!\n\n");
                }
            }.startListening();
….
```

9.2 Last consumed messages' ID persisting helpers

When the application uses the guaranteed message delivery mechanism, it is required that the application maintains the tracking of the last consumed messages' IDs. To simplify these tasks, two auxiliary classes were created that allow storing consumed message IDs either in the JDBC compliant DB or in the XML formatted file.
In order to achieve greater flexibility both implementations use the same interface. The only difference is how actual implementation is created and initialized. Here are examples of how to create and initialize JDBC and file persisters:

```
import com.inforeach.eltrader.engine.util.*;
```

```
 IELTLastConsumedMessageIdPersister jdbcPersister =
ELTLastConsumedMessageIdPersisterFactory.getInstance().createPersister(ELTLas
tConsumedMessageIdPersisterFactory.IMPL_SPEC_JDBC);
jdbcPersister.initialize("myOracle:ConsumedIds");

 IELTLastConsumedMessageIdPersister filePersister =
ELTLastConsumedMessageIdPersisterFactory.getInstance().createPersister(ELTLas
tConsumedMessageIdPersisterFactory.IMPL_SPEC_FILE);
filePersister.initialize("file://c:\temp\consumedIds.xml");
```

Then the application, after processing the message, can just call:

```
jdbcPersister.persistConsumedMessageId(eltMessage);
```

Then the persister will store the message ID.

When the application needs to restore data from the persister it could use appropriate 'get' methods from the `IELTLastConsumedMessageIdPersister` interface and 'set' methods of the **ELTGuaranteedEngineMessageListener**. Or, the application can use static method of the **ELTAbstractLastConsumedMessageIdPersister** class:

```
ELTAbstractLastConsumedMessageIdPersister.configureGuaranteedMessageListener(
myConnectionName, myGuaranteedMessageListener, myPersister);
```

9.3 Getting FIX protocol version for connection

In order to create message objects or field group objects in a most efficient way it is required to know the specific FIX protocol version of the connection where the messages will be sent. One possible way of doing it is getting it from the connection properties. Because of the possibility of dynamic reloading of FIX engine configuration at run time it is not correct to cache protocol version once at program start up and use it later. Use method `com.inforeach.eltrader.engine.ELTEngineUtil.getProtocolVersion(ELTConnectionName connectionName)` to get protocol version at run time.

Appendix A: Sample FIX Message Processing Application

```
// Import section
import java.io.*;
import com.inforeach.util.*;
import com.inforeach.util.log.*;
import com.inforeach.util.xml.*;
import com.inforeach.eltrader.*;
import com.inforeach.eltrader.message.*;
import com.inforeach.eltrader.engine.*;
import com.inforeach.eltrader.engine.esevent.*;
import com.inforeach.eltrader.engine.fix.FIX;
import com.inforeach.eltrader.engine.util.*;
import com.inforeach.eltrader.service.*;

// Main application class:
public class FIXMessagingApp
{
    public static void main(final String args[])
    {
        try
        {
          ELTEngineProxy.setEngineProxyActivationHandler(new
ELTAbstractEngineProxyActivationHandler()
              {
                public void onPreEngineProxyActivation()
                {
                  try
                  {

                     String needGuaranteedDelivery;
                     if (args.length > 3)
                         needGuaranteedDelivery = args[3];
                     else
                         needGuaranteedDelivery = "0";

                      // get engine proxy handle:
                      ELTEngineProxy proxy = ELTSystem.getEngineProxy();

                      // setup listenerProperties XMLData object:
                      XMLData listenerProps = new XMLData();

                      // interested in all visible connections:
                      // alternatively could specify comma-separated list of connections
                      // like "connection1,connection1":
                      listenerProps.setAttributeValue(
                                       ELTEngineDefs.CFG_CONNECTION_NAMES,
                                       "*");

                      // interested in FIX application messages:
                      listenerProps.setAttributeValue(
                                       ELTEngineDefs.CFG_SUBSCRIBE_FOR_APP_MSGS,
                                       true);

                      // interested in FIX admin messages:
                      listenerProps.setAttributeValue(
                                       ELTEngineDefs.CFG_SUBSCRIBE_FOR_ADMIN_MSGS,
                                       true);

                      // interested in Connection State messages:
                      listenerProps.setAttributeValue(
                                       ELTEngineDefs.CFG_SUBSCRIBE_FOR_CONN_INFO,
                                       true);

                      // now subscribe MessageProcessor to engine proxy:
                      if (needGuaranteedDelivery.equals("1"))
                      {
                          // subscribe with support for guaranteed delivery
                          ELTSystem.getLogFacility().info().write("Registering guaranteed
message listener.");

                          // instantiate guaranteed delivery component:
                          ELTGuaranteedEngineMessageListener guaranteedMsgListener =
                              new  ELTGuaranteedEngineMessageListener(
                                                             new MessageProcessor());

                          // get array of all visible connections from the proxy.
                          // we cannot get them out of the listenerProps object
                          // because we used '*' to denote all visible connections:
                          ELTConnectionName[] connectionNames =
                              proxy.getConnectionNameList(GlobalSystem.getUserTicket());

                          // for each of the connections set last consumed sent and
                          // received app message id:
                          for (int i = connectionNames.length - 1; i >= 0; i--)
                          {
                              // set last consumed message id of received app message
                              guaranteedMsgListener.
                                       setLastConsumedAppMessageFromCounterPartyId(
                                           connectionNames[i],

ELTEngineDefs.INITIAL_APP_MESSAGE_FROM_COUNTER_PARTY_ID);

                              // set last consumed message id of sent app message
                              guaranteedMsgListener.
                                       setLastConsumedAppMessageToCounterPartyId(
                                           connectionNames[i],

ELTEngineDefs.INITIAL_APP_MESSAGE_TO_COUNTER_PARTY_ID);
                          }

                          // here this call is made for demo purposes only since
                          // last consumed message ids are set to initial values
                          // and, thus, no message gaps exist right now:
                          guaranteedMsgListener.retrievePotentiallyLostMessages();

                          // finally subscribe with guaranteedMsgListener as listener:
                          proxy.subscribe(GlobalSystem.getUserTicket(),
                                      guaranteedMsgListener,
                                      listenerProps);
                      }
                      else
                      {
                          // subscribe without overhead of guaranteed delivery:
                          proxy.subscribe(GlobalSystem.getUserTicket(),
                                      new MessageProcessor(),
                                      listenerProps);
                      }
                  }
                  catch (Exception ex)
                  {
                    ex.printStackTrace();
                  }
                }
              });

            // get main parameters including bootstrap file:
            String bootstrap_file_name = args[0];
            String processType = args[1];
            String processName = args[2];

            // initialize Inforeach FIX engine component.
            ELTEngineComponent.initialize(bootstrapData, processType, processName);

            GlobalSystem.getCmdInterpreter().start();
            System.out.println(GlobalSystem.getCmdInterpreter().getHelp());

            Thread.currentThread().join();
        }
        catch(Exception ex)
        {
            ex.printStackTrace();
        }
    }

    // Class implementing IEvenListener:
    private static
    class MessageProcessor implements IEventListener
    {
        public void onEvent(Object event)
          {
            ILogFacility log = ELTSystem.getLogFacility();
            if (null == event)
            {
              log.error().write("MessageProcessor received null event");
              return;
            }
            else if (event instanceof ELTAppMessageEvent)
            {                     ////
              ELTAppMessage m = ((ELTAppMessageEvent) event).getMessage();
              if (ELTAppMessage.isOriginalOrder(m))
              {
                  log.info().write(">>> Processed order: " +
                                    m.getFieldValue(ELT.fieldClOrdID));
                  // Do something if necessary...
              }
              else if (ELTAppMessage.isOrderConfirmation(m))
              {
                  log.info().write(">>> Received confirmation for: " +
                                    m.getFieldValue(ELT.fieldClOrdID));
                  // Do something if necessary...
              }
              else if (ELTAppMessage.isOrderFillReport(m))
              {
                  log.info().write(">>> Received fill for: " +
                                    m.getFieldValue(ELT.fieldClOrdID));
                  // Do something if necessary...
              }
              else
              {
                  log.info().write(">>> Received app message: " + m);
                  // Do something if necessary...
              }
            }
            else if (event instanceof ELTAdminMessageEvent)
            {
              ELTMessage m = ((ELTAdminMessageEvent) event).getMessage();
              log.info().write(">>> Processing admin message: " + m);
              // Do something if necessary...
            }
            else if (event instanceof ELTConnectionStateEvent)
            {
              ELTConnectionState state = ((ELTConnectionStateEvent)event).
                                              getConnectionState();
              log.info().write(">>> Processing connection state event: " +
                               state);
              // Do something if necessary...
            }
        }
    }
}
```

Appendix B. Customizing Field and Message Sets

The **InfoReach** engine library includes a collection of XML files that contain FIX field and message descriptions for FIX Protocol 4.x. The **Metadata Editor** tool that's distributed with **InfoReach** is a visual tool. It helps create user-defined fields, messages, or even custom protocol versions quickly. Whenever new user-defined fields and/or messages are required for some FIX connection they can be added quickly to the **UserDefined** section through the **MetadataEditor** tool. If numerous field or message modifications are required to fields or messages that are already defined by a specific version of FIX protocol, the original version can be saved using the MetadataEditor tool. You would need to save the version under different name. After saving, the modifications to the newly created copy can be made. Later, appropriate FIX connections can be configured to have the name of the newly created version as their protocol version (see section 3.2). This ensures that message validation for these connections are in accordance with the newly configured field and message formats.

Some of the tags in the user-defined field range are reserved for use by **InfoReach** and should not be modified. The set of reserved fields contain:

- Tag 16536 --- this field in the message contains the message format string name. Currently it can be "FIX" or "FIXML". Use **ELT.fieldMessageFormat** constant to access this field.

- Tag 16575 --- this field in the message contains the character 'Y' if this message came from the counterparty. It is not set or set to 'N' if this message was sent or posted locally. Use **ELT.fieldIsFromCounterParty** constant to access this field.

- Tag 16576 --- this field in the message contains the string name of the connection through which this message was received, sent or posted. This field is useful for applications that process message flow. Use **ELT.fieldConnectionName** constant to access this field.

- Tag 16580 --- this field in the message contains a connection-wide unique integer message ID. This is the ID that should be persisted by those applications that use **ELTGuaranteedMessageListener**. Use **ELT.fieldConnectionMessageId** constant to access this field.

- Tag 16581 --- this field in the message contains an engine network-wide unique integer message ID. Use **ELT.fieldGlobalMessageId** constant to access this field.

- Tag 16584 --- this field in the received message contains a string explanation of errors that were encountered at the time of message validation. The application may examine this field every time it is notified of the received message to make sure the received message is valid. Use the **ELT.fieldMessageErrors** constant to access this field.

- Tags 16502, 16503, 16504, 16537, 16582 and 16583 are used internally by InfoReach.

Appendix C. Runtime Properties

When starting a process that hosts InfoReach engines, the following environment properties can be set with –D flag (for example –**DoptimizeForSpeed="true"**):

Property	**Description**
`optimizeForSpeed`	Possible values are "**true**" or "**false**". If set to "**true**" the **InfoReach Value** objects work faster but also take more memory. Depending on the nature of the application, this property should be set in order to optimize either the speed, or the memory consumption.
`autoReconnectDelay`	If the process hosts client engine(s), this property could be used to control how long the engine waits before attempting to reconnect to the counterparty after the connection was dropped. The values should be specified in milliseconds. Default reconnect delay is 2000 (2 seconds).
`pgpProgramPath`	If encryption method **5** is used by the engine, use this property to specify the PGP executable path(see 5.2).
`localHost`	Specifies which of available IPs should be used for socket binding.
`java.rmi.server.hostname`	Specifies which of available IPs will be used for RMI objects.
`noPublishSent`	Valid values '**true**' or '**false**'. Default value is '**false**'. If set to '**true**' then connection option '**publishSentAppMessages**' will be ignored and EngineNetworkConfig tool will not show this as an option at all.
`standardVersionEditable`	Valid values '**true**' or '**false**'. Default value is '**false**'. If set to '**true**' makes MetaDataEditor tool to allow edit standard FIX protocol versions (which is disabled by default).
`currentDBalias`	Specifies DB alias from system.xml which will be used to substituted `'databaseAlias'` attribute in <`MessageLogFacility`> element of connection properties if it has value `'%currentDBalias%'`.
`nativeThreadDumpEnabled`	Valid values '**true**' or '**false**'. Specifies whether or not programmatic thread dump available on this particular platform.
`disableCmdInterpreter`	Valid values '**true**' or '**false**'. Default value is '**false**'. Useful when process is running on background and std in is unavailable. Sometimes on unix platforms it causes serious problems.
`disableLogFacilityOutputDestinations`	Valid values '**true**' or '**false**'. Default value is '**false**'. If process's output is redirected this option makes the engine ignore

	`<OutputLogFacilityList>/<XXLogger>/outputDestination` settings and skip writing to the file and just put debug/info/error/warning print outs to std out.
`directoryCacheRoot`	If some processes are running far from the ELTService process, it makes sense to cache directory resources, which improves start up time significantly. The value of this option specifies path on local drive where cached resources will be saved.
`useConnectionUniformMessageId`	Since of Engine version 6.2 it assigns sequentially incremented ID (connection wide) to every message regardless of its direction. This solves two problems: 1) If hosting machine has running time synchronization software then we cannot rely on timestamps during connection state recovery. Because these IDs are time independent Engine can rely on them to restore real sequence of messages regardless of time jumps 2) These IDs can be used on GUIs to sort messages in order to see the real sequence of messages. Prior this there was no way to do it because FIX sequence numbers are maintained separately for incoming and outgoing connections and sending time field do not guarantee sequence of event in case if messages were exchanged at high rate (more then several messages per millisecond).
`oracleUsesClob`	Valid values '**true**' or '**false**'. Default value is '**false**'. Controls DB type for "Message" column in DB tables. If 'false' then varchar(xx) will be used (see description for 'oracleMessageSize'). If 'true' then that column will be CLOB.
`oracleMessageSize`	Specifies size of varchar() type for "Message" column. Default value: 4000 (max allowed by Oracle).
`tradingDayStartTime`	Format: HH:mm:ss It is always in GMT. Example: -DtradingDayStartTime=20:00:00 Specifies the time which is considered as start of the trading day. It is relevant only for connections configured to release pending messages on connect. Message that are pending before DtradingDayStartTime are not released on connect.