InfoReach

Engineered for what's next

InfoReach FIX Engine

FIX C++ Programmer's Guide

Version 8.3

Contents

Introduction

InfoReach FIX Engine provides you with a C++ API that can be used by client C++ applications that are developed for establishing FIX connections, and sending and receiving FIX messages.

Engine Proxy Initialization

Methods of the ELTEngineProxy class make the engine functionality available to client applications. The ELTEngineProxy has to be initialized at the beginning of each client application process.

Example 1: ELTEngineProxy Initializing

```
#include <eltraderEngine/ELTEngineProxy.h>

…
…
    try
    {

        // Creating configuration URL
        ELTUrl url
("java:engineConsole:./cppcfg/cppbootstrap.xml");

        // Initializing engine proxy.
        ELTEngineProxy::initialize(url);

         // Here we have full functional ELTEngineProxy
    }
    catch(ELTEngineException& ex)
    {
        std::cout << "Exception was caught " << ex.what() <<
std::endl;
    }
….
```

The configuration URL specifies how and where the configuration for ELTEngineProxy can be obtained. It should always be started with a "java:" string. The last part of the URL points to the bootstrap file where the configuration of the processes is located. The middle part is the name of the process configuration that will be used. In short, it has to be:
"java:<ProccessName>:<PathToBootstrapFile>"

The bootstrap file contains parameters for the process. This XML file contains the element <Processes> that in turn contains elements that represent named settings for a process. In the following example you can see the configuration for only one process with the name "engineConsole".

The most important parts of the process configuration contain these attributes:

bootstrapDataFile – specifies the "java's" bootstrap file, which is used by java processes.
type – type of the process in the FIXEngine's processes.xml file.
name – name of the process in the FIXEngine's processes.xml file.

All of these 3 attributes are used to bootstrap internal implementations in the same way java processes do. Refer to "InfoReach FIX JAVA Programmers Guide.doc" for details about bootstrapping java processes.

Example 2: C++ Bootstrap file.

```
<Bootstrap>
     <Processes>
          <engineConsole
               bootstrapDataFile="../../../cfg/bootstrap.xml"
               type="CppEngineConsole"
               name="CppEngineConsoleDemo"
          >
               <JVMSettings>
                    <property
                         name="-DnotificationLevel"
                         value="4"
                    />
                    <property … />
                    <property … />
                    <property … />
               </JVMSettings>
          </engineConsole>
     </Processes>
</Bootstrap>
```

Sending FIX messages

There are two ways to send FIX messages through **ELTEngineProxy**:

- Send a prepared ELTMessage object.
- Send a plain FIX string.

Example 3: Sending a Programmatically Prepared Field Group

```
ELTFieldGroupPtr fieldGroup(new ELTFieldGroup());
ELTValue fieldValue;
fieldValue.setType(ELTValue::typeAlphaNumeric);
fieldValue = "FIX.4.1";
fieldGroup->setFieldValue(ELT.fieldBeginString, fieldValue);
fieldValue = "D";
fieldGroup->setFieldValue(ELT.fieldMsgType, fieldValue);
fieldValue = "TestOrder";
fieldGroup->setFieldValue(ELT.fieldClOrdID, fieldValue);
fieldValue = "InfoReach";
fieldGroup->setFieldValue(ELT.fieldSymbol, fieldValue);

fieldValue = '1';
fieldValue.setType(ELTValue::typeChar);
fieldGroup->setFieldValue(ELT.fieldOrdType, fieldValue);
fieldGroup->setFieldValue(ELT.fieldHandlInst, fieldValue);
fieldGroup->setFieldValue(ELT.fieldSide, fieldValue);

fieldValue = '0';
fieldGroup->setFieldValue(ELT.fieldSettlmntTyp, fieldValue);
fieldGroup->setFieldValue(ELT.fieldTimeInForce, fieldValue);

fieldValue = "400";
fieldValue.setType(ELTValue::typeQty);
fieldGroup->setFieldValue(ELT.fieldOrderQty, fieldValue);

fieldValue = "USD";
fieldValue.setType(ELTValue::typeCurrency);
fieldGroup->setFieldValue(ELT.fieldCurrency, fieldValue);

time_t t;
time(&t);
fieldValue = t;
fieldValue.setType(ELTValue::typeUTCTimestamp);
fieldGroup->setFieldValue(ELT.fieldCurrency, fieldValue);

ELTMessagePtr message(new ELTMessage(fieldGroup));

ELTEngineProxy::getInstance().sendMessage(
    ELTUserTicket::getSystemTicket(),
    std::string("ConnectionName"),
    message);
```

Example 4: Sending FIX-Formatted String

```
            ELTEngineProxy::getInstance().sendMessage(
                 ELTUserTicket::getSystemTicket(),
                 std::string("ClientA_toBrokerX"),

std::string("8=FIX.4.1□40=1□38=400□35=D□63=0□21=1□5677=20011003-
18:43:35□60=20010505-
11:25:00□59=0□58=TIMESTAMP□15=USD□55=MSFT□54=1□11=Order.2□"));
```

Receiving FIX Messages

In order to receive messages you need to subscribe to a component that implements the **IELTEventListener** interface as a listener to the **ELTEngineProxy (**Example 5**)**. The event received from the engine can be examined and processed according to the application's needs inside the **processEvent()** method.

Example 5: processEvent

```
class ClientProcess : public IELTEventListener
{
      virtual void processEvent(const ELTEventPtr event)
            {
                  // to do something with the event
                  std::cout << "Received event:" << std::endl
<< event << std::endl;
            }
}
```

This class should be subscribed for engine events via the **ELTEngineProxy subscribeDirect()** method:

Example 6: ELTEngineProxy subscribeDirect()

```
      IELTEventListenerPtr client(new ClientProcess());
            ELTEngineProxy::getInstance().subscribeDirect(
                ELTUserTicket::getSystemTicket(),
                std::string("ConnectionName"),
                true, // for App message events
                false, // for Addmin message events
                false, // for connection state events
                client);
```

Method **processEvent(const ELTEventPtr event)** is called every time an application event type is received from a connection with the "ConnectionName" name.

To stop the subscription, use the ELTEngineProxy unsubscribeDirect(…) method shown here:

Example 7: ELTEngineProxy unsubscribeDirect(…)

```
            ELTEngineProxy::getInstance().unsubscribeDirect(
                ELTUserTicket::getSystemTicket(),
                std::string("ConnectionName"),
                true, // from App message events
                false, // from Admin message events
                false, // from connection state events.
                Client);
```

Build Instructions

WIN32

System requirements:

- MS Visual Studio 6.0.
- MS Visual Studio.NET 2003.
- MS Visual Studio.NET 2005.

Important note: engine C++ libraries require EXACT matching of the runtime environment that was used for libraries builds and for user application builds.
That means that if you use MS VS 6.0 for your application, you have to use libraries built with MS VS 6.0.
In order to support all above-mentioned platforms, libraries are provided built with each of the supported compilers. Depending on Debug/Release settings and compiler libraries, names may have the following suffixes:

- _vs60_d – debug libraries for MS VS 6.0;
- _vs60 – release libraries for MS VS 6.0;
- _vs71_d – debug libraries for MS VS .NET 2003;
- _vs71 – release libraries for MS VS .NET 2003;
- _vs80_d – debug libraries for MS .NET 2005;
- _vs80 – release libraries for MS .NET 2005;

In order to create an application that works correctly with InfoReach FIX Engine, you need to configure your project to follow these requirements:

- Configure you compiler to use RTTI (Run-Time Type Information)
- Include a path to **$(INSTALLATION_HOME)\win32\include** in your project
- Modify your **PATH** environment variable to include the path to **$(INSTALLATION_HOME)\win32\native**

- Link your project with:
  'Debug Multithreaded DLL' option when you are building DEBUG version of you application;
  'Multithreaded DLL" for release version;

- Link your application with the corresponding to your compiler infra.lib, eltraderEngine.lib and xmlengineutils.lib located in **$(INSTALLATION_HOME)\win32\native**

engineConsole Sample

Sample code is located in **$(INSTALLATION_ROOT)\cpp\samples**.

Source path: **$(INSTALLATION_ROOT)\cpp\samples\engineConsole\src**

This is a simple command line tool that demonstrates the use of basic engine functionality from a client application. This includes subscribing/unsubscribing for certain types of messages, sending/receiving messages, resetting connections, and using a message-retrieving API.

Building engineConsole

The **engineConsole** comes with a pre-compiled executable. It can also be built with MS VisualStudio 6.0, .NET 2003 and .NET 2005 on Win32.

The path to workspace and project files is:

$(INSTALLATION_ROOT)\samples\cpp\engineConsole

Running engineConsole

This sample works only with the default engine demo configuration. Before starting the engine console, you need to start the following processes in the sequence illustrated in Table 1:

Table 1: Start Sequence

```
WIN32:
$(INSTALLATION_ROOT)\win32\bin\ELTService.bat
$(INSTALLATION_ROOT)\win32\bin\FIXServer.bat
$(INSTALLATION_ROOT)\win32\bin\FIXClient.bat
```

Then, you can run:

```
WIN32
$(INSTALLATION_ROOT)\samples\cpp\engineConsole\EngineConsoleWin32.bat
```