FIX Antenna HFT Quick Start Guide

Overview

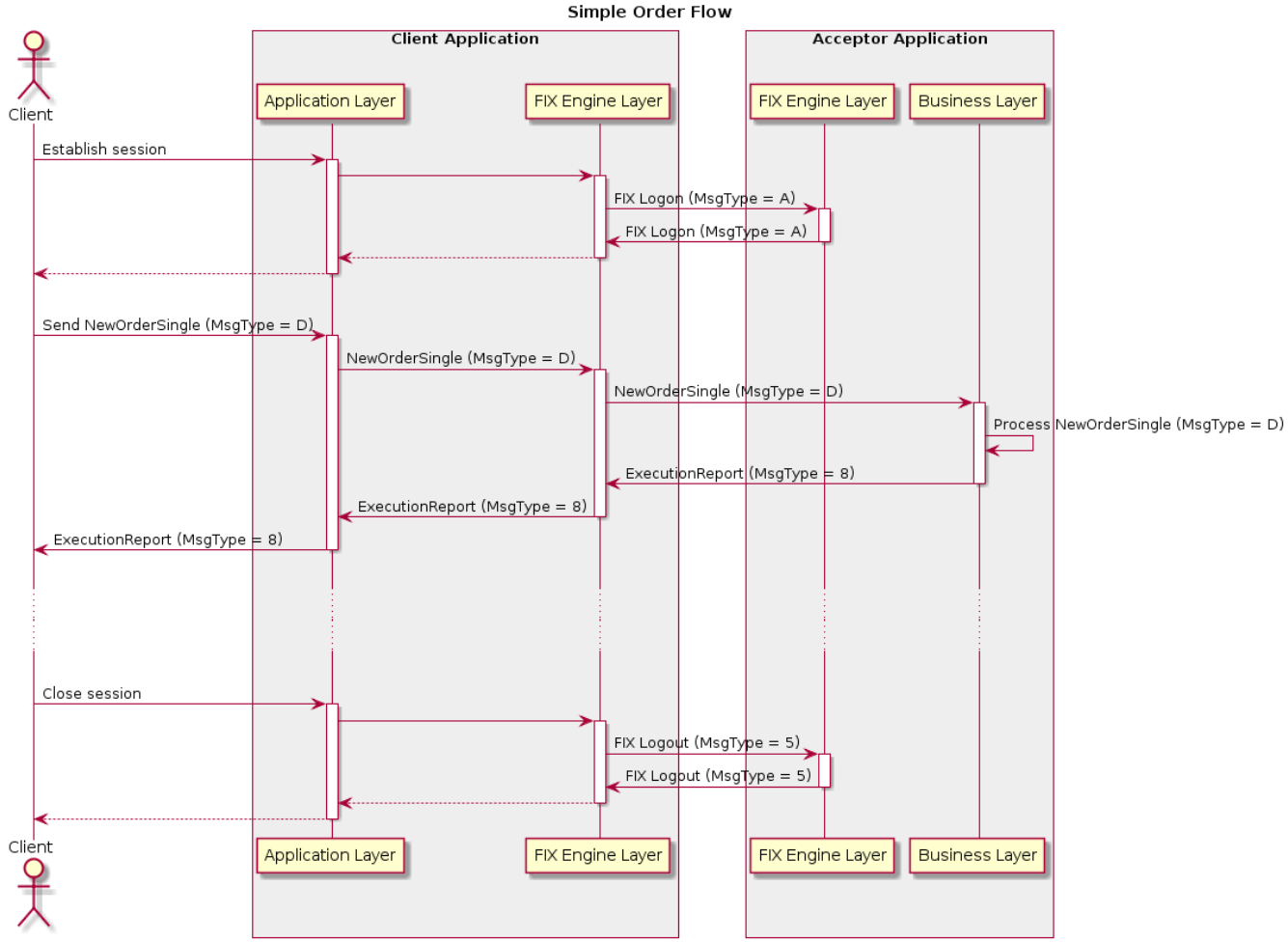
This article would be interesting for those of you who has faced with FIX Antenna for the first time and want to touch and feel it.

It will help you to build your first application based on FIX Antenna so that you will be aware of possibilities FIX Antenna provides.

Application

Typically FIX application provides FIX connectivity to multiple clients through transport protocols and also offers possibilities to install, configure, administrate and monitor trading information flows.

For our case assume that we need to build an app which will be able to establish and terminate sessions and support the simplest Order Flow presented below:



The main steps which need to be implemented in order to cover declared functionality are:

- Initialize FIX engine
- Create session (initiator or acceptor)
- Send messages
- Process incoming messages
- Close session
- Destroy engine (release resources)

This section describes how to create the application step-by-step with code examples. Follow instructions below to write, compile and run your first application with FIX Antenna C++.

Engine initialization

Use the instruction below to initialize FIX engine.

```
// Initialize engine.
FixEngine::init();
```

The "engine.properties" file contains common FIX engine configuration parameters. If full path is not specified FIX engine is looking in the current directory otherwise FIX engine uses full path to locate properties file.

```
// Initializes engine.
FixEngine::init("engine.properties");
```

If error occurs during initialization (e.g. the properties file cannot not be found, a required property is missing etc.) the exception is thrown.

```
// Initialize engine.
try {
   FixEngine::init("engine.properties");
}
catch( const Utils::Exception& ex ) {
   cout << "ERROR: " << ex.what() << endl;
}
```

For more information about FIX engine description refer to the section Engine description.

Session creation

Creation of session-acceptor

You can create a session acceptor in three steps:

- Create an application object (observer entity for session to receive events; see Processingincomingmessage for details)
- Create an Engine::Session object
- Call the Engine::Session::connect() session method

```
// Declare Application - FIX session observer
class MyApplication : public Engine::Application {
    // Override several virtual methods
    virtual bool process(const Engine::FIXMessage& msg, const Engine::Session& sn) {
        return true;
    }
    virtual bool onResend(const Engine::FIXMessage& msg, const Engine::Session& sn) {
        return true;
    }

    virtual void onLogonEvent(const Engine::LogonEvent* event, const Engine::Session& sn) {}
    virtual void onLogoutEvent(const Engine::LogoutEvent* event, const Engine::Session& sn) {}
    virtual void onUnexpectedMessageEvent(const Engine::UnexpectedMessageEvent* apEvent, const Engine::
Session& aSn) {}
    virtual void onSequenceGapEvent(const Engine::SequenceGapEvent* event, const Engine::Session& sn) {}
    virtual void onSessionLevelRejectEvent(const Engine::SessionLevelRejectEvent* event, const Engine::
Session& sn) {}
    virtual void onMsgRejectEvent(const Engine::MsgRejectEvent* event, const Engine::Session& sn) {}
    virtual void onHeartbeatWithTestReqIDEvent(const Engine::HeartbeatWithTestReqIDEvent& event, const
Engine::Session& sn) {}
    virtual void onResendRequestEvent(const Engine::ResendRequestEvent& event, const Engine::Session& sn) {}
    virtual void onNewStateEvent(const Engine::NewStateEvent& event, const Engine::Session& sn) {}
    virtual void onUnableToRouteMessage(const Engine::UnableToRouteMessageEvent& event, const Engine::
Session& sn) {}
};
// Create instance of Application
MyApplication application;
// Create instance of FIX session
Engine::Session* pSA =  Engine::FixEngine::singleton()->createSession(&application, "Sender", "Target",
Engine::FIX44);
// Connect session as acceptor
pSA->connect();
```

For more information about FIX session in general and FIX session acceptors description refer to the sections Session description.

Creation of session-initiator

You can create a session initiator in three steps:

- Create an application object (observer entity for session to receive events, see Processingincomingmessage for details)
- Create an Engine::Session object
- Call the Engine::Session::connect session method passing the initiator's specific parameters

```
// Declare Application - FIX session observer
class MyApplication : public Engine::Application {
    // Override several virtual methods
    virtual bool process(const Engine::FIXMessage& msg, const Engine::Session& sn) {
        return true;
    }
    virtual bool onResend(const Engine::FIXMessage& msg, const Engine::Session& sn) {
        return true;
    }

    virtual void onLogonEvent(const Engine::LogonEvent* event, const Engine::Session& sn) {}
    virtual void onLogoutEvent(const Engine::LogoutEvent* event, const Engine::Session& sn) {}
    virtual void onUnexpectedMessageEvent(const Engine::UnexpectedMessageEvent* apEvent, const Engine::
Session& aSn) {}
    virtual void onSequenceGapEvent(const Engine::SequenceGapEvent* event, const Engine::Session& sn) {}
    virtual void onSessionLevelRejectEvent(const Engine::SessionLevelRejectEvent* event, const Engine::
Session& sn) {}
    virtual void onMsgRejectEvent(const Engine::MsgRejectEvent* event, const Engine::Session& sn) {}
    virtual void onHeartbeatWithTestReqIDEvent(const Engine::HeartbeatWithTestReqIDEvent& event, const
Engine::Session& sn) {}
    virtual void onResendRequestEvent(const Engine::ResendRequestEvent& event, const Engine::Session& sn) {}
    virtual void onNewStateEvent(const Engine::NewStateEvent& event, const Engine::Session& sn) {}
    virtual void onUnableToRouteMessage(const Engine::UnableToRouteMessageEvent& event, const Engine::
Session& sn) {}
};
// Create instance of Application
MyApplication application;
// Create instance of FIX session
Engine::Session* pSI =  Engine::FixEngine::singleton()->createSession(&application, "Sender", "Target",
Engine::FIX44);
// Connect session as initiator
pSI->connect(30, "127.0.0.1", 9106);
```

For more information about FIX session in general and FIX session initator description refer to the sections Session description.

Creating new order

FIX Antenna provides two interfaces for FIX messages manipulations:

1. Flat model
2. Object model

The flat model is based on a simple get/set FIXMessage interface and demonstrates high performance.

```
// create FIX 4.4 New Order Single skeleton
Engine::FIXMessage* pOrder = Engine::FIXMsgFactory::singleton()->newSkel(Engine::FIX44, "D");
// set ClOrdID
pOrder->set( FIXField::ClOrdID, "USR20000101" );
// get ClOrdID
Engine::FIXFieldValue clOrdID;
bool isPresent = pOrder->set( FIXField::ClOrdID, &clOrdID );
assert( isPresent );
// set trading session
// create repeating group with 1 entry
pOrder->set( FIXField::NoTradingSessions, 1 );
// get pointer to the repeating group
Engine::FIXGroup *pGroup = pOrder->getGroup(FIXFields::NoTradingSessions);
// set trading session ID for the 1st entry
pGroup->set( FIXField::TradingSessionID, "PRE-OPEN", 0 );
// set all required fields ...
```

The object model is a typified interface that is less efficient but more user-friendly than the flat one.

```
// create FIX 4.4 New Order Single
FIX44::NewOrderSingle order( Engine::FIXMsgFactory::singleton()->newSkel(Engine::FIX44, "D") );
// set ClOrdID
order.ClOrdID().set( "USR20000101" );
// get ClOrdID
bool isPresent = !order.ClOrdID().isEmpty();
string clOrdID = order.ClOrdID();
```

For more information about FIX message and repeating groups refer to the sections Message description, Repeating Groups description.

Sending order

To send a message to the session you should call Engine::Session::put method. In fact this method does not send message but puts it to the queue, from which message will be sent out by separate thread. The method in its turn returns immediately. In other words sending is an asynchronous process in FIX Antenna, which also means that after method returns message is not necessarily sent yet.

```
// Send order (flat model) to session initiator
pSI->put( pOrder);
// Send order (object model) to session acceptor
pSA->put( order.get() );
```

For more information about sending messages refer to the sections Send message and Send message.

Processing incoming message

The Application class is responsible for:

- Processing incoming messages from the remote FIX Engine

```
virtual bool Engine::Application::process(const Engine::FIXMessage &, const Engine::Session &aSn)
```

- Processing session events (required only if default behavior is not suitable)

```
virtual void Engine::Application::onResendRequestEvent(const Engine::ResendRequestEvent &,const
Engine::Session &)
...
```

Create a new class derived from the Application class and override the Engine::Application::process method in this class to process incoming messages. If an incoming message is successfully processed, the method should return "true", if the message cannot be processed at the moment, it should be copied for further processing by application.

Other useful methods to override are: Engine::Application::onLogonEvent, Engine::Application::onLogoutEvent,Engine::Application::onSequenceGapEvent, Engine::Application::onSessionLevelRejectEvent, etc. These are the call-back methods called to notify about the corresponding session-level events. Note that all pure virtual methods of Application must be overridden (implementation can be just empty body). The Engine::Application::process method is called only when application-level message is received. All session-level messages (Heartbeats, Test Requests, Resend Requests, etc.) are handled by FIX Antenna. However you can modify default behavior overriding call-back methods in Application and providing your own logic.

if you need to keep FIX message for future processing create a copy of the message and save a copy instead of saving original message (refer to the Engine::FIXMsgProcessor::clone method). Below is an example of custom implementation of Engine::Application interface:

```
class Appl : public Engine::Application{
public:
    virtual bool process(const Engine::FIXMessage & aMsg, const Engine::Session& aSn) {
        std::clog << "aMsg: " << *aMsg.toString('|') << std::endl;
        return true;
    }
    virtual void onLogonEvent(const Engine::LogonEvent* apEvent, const Engine::Session& aSn) {
        std::clog << "LogonEvent, the Logon message was received: " << apEvent->m_pLogonMsg-
>toString() << std::endl;
    }
    virtual void onLogoutEvent(const Engine::LogoutEvent* apEvent, const Engine::Session& aSn) {
        std::clog << "LogoutEvent, the Logout message was received: " << apEvent->m_pLogoutMsg-
>toString() << std::endl;
    }
    virtual void onSequenceGapEvent(const Engine::SequenceGapEvent* apEvent, const Engine::Session&
aSn) {
        std::clog << "SequenceGapEvent" << std::endl;
    }
    virtual void onSessionLevelRejectEvent(const Engine::SessionLevelRejectEvent* apEvent, const
Engine::Session& aSn) {
        std::clog << "SessionLevelRejectEvent" << std::endl;
    }
    virtual void onMsgRejectEvent(const Engine::MsgRejectEvent* event, const Engine::Session& sn){
        std::clog << "MsgRejectEvent" << std::endl;
    }
    virtual void onResendRequestEvent(const Engine::ResendRequestEvent &,const Engine::Session &) {
        std::clog << "ResendRequestEvent" << std::endl;
    }
    virtual void onNewStateEvent(const Engine::NewStateEvent &,const Engine::Session &) {
        std::clog << "NewStateEvent" << std::endl;
    }
    virtual void onUnableToRouteMessage(const Engine::UnableToRouteMessageEvent &,const Engine::
Session &) {
        std::clog << "UnableToRouteMessage" << std::endl;
    }
    virtual bool onResend(const Engine::FIXMessage &,const Engine::Session &) {
        std::clog << "Resend" << std::endl;
        return true;
    }
    virtual void onHeartbeatWithTestReqIDEvent(const Engine::HeartbeatWithTestReqIDEvent &,const
Engine::Session &) {
        std::clog << "HeartbeatWithTestReqIDEvent" << std::endl;
    }
};
```

Do not delete the registered Application until you unregister it. For more information about processing incoming messages and Application class refer to the section Application description.

Closing session

Use the following methods to close the session:

- `Engine::Session::disconnect( bool forcefullyMarkAsTerminated = false )`
- `Engine::Session::disconnect( const std::string &logoutText, bool forcefullyMarkAsTerminated = false )`

For more information about disconnect refer to the sections Session description, disconnect, and disconnect.

Full sample

The sample below illustrates all aforementioned instructions combined in one application.

```
#include <iostream>
#include <memory>
#include <B2BITS_V12.h>
using namespace std;
class MyApp : public Engine::Application {
public:
    MyApp() {}
    virtual bool process(const Engine::FIXMessage& fixMsg, const Engine::Session& aSn) {
        cout << *fixMsg.toString('|') << endl;
        return true;
    }
    virtual void onLogonEvent(const Engine::LogonEvent* apEvent, const Engine::Session& aSn) {
        cout << "onLogonEvent" << endl;
    }
    virtual void onLogoutEvent(const Engine::LogoutEvent* apEvent, const Engine::Session& aSn) {
        cout << "onLogoutEvent" << endl;
    }
    virtual void onMsgRejectEvent(const Engine::MsgRejectEvent* event, const Engine::Session& sn) {
        cout << "onMsgRejectEvent" << endl;
    }
    virtual void onSequenceGapEvent(const Engine::SequenceGapEvent* apEvent, const Engine::Session& aSn) {
        cout << "onSequenceGapEvent" << endl;
    }
    virtual void onSessionLevelRejectEvent(const Engine::SessionLevelRejectEvent* apEvent, const Engine::
Session& aSn) {
        cout << "onSessionLevelRejectEvent" << endl;
    }
    virtual void onHeartbeatWithTestReqIDEvent(const Engine::HeartbeatWithTestReqIDEvent& event, const
Engine::Session& sn) {
        cout << "onHeartbeatWithTestReqIDEvent" << endl;
    }
    virtual void onResendRequestEvent(const Engine::ResendRequestEvent& event, const Engine::Session& sn) {
        cout << "onResendRequestEvent" << endl;
    }
    virtual void onNewStateEvent(const Engine::NewStateEvent& event, const Engine::Session& sn) {
        cout << "onNewStateEvent" << endl;
    }
    virtual void onUnableToRouteMessage(const Engine::UnableToRouteMessageEvent& event, const Engine::
Session& sn) {
        cout << "onUnableToRouteMessage" << endl;
    }
    virtual bool onResend(const Engine::FIXMessage& msg, const Engine::Session& sn) {
        cout << "onResend" << endl;
        return true;
    }
};
int main( int argc, char* argv[] )
{
    try
    {
        // Initialize engine.
        Engine::FixEngine::init();
        // Create Application instance
        MyApp application;
        // Create FIX session instance
        Engine::Session* pSA =  Engine::FixEngine::singleton()->createSession(&application, "Sender",
"Target", Engine::FIX44);
        // Connect session as acceptor
        pSA->connect();
        // Create FIX session instance
        Engine::Session* pSI =  Engine::FixEngine::singleton()->createSession(&application, "Target",
"Sender", Engine::FIX44);
        // Connect session as initiator
        pSI->connect(30, "127.0.0.1", Engine::FixEngine::singleton()->getListenPort());
        // create FIX 4.4 New Order Single
        auto_ptr<Engine::FIXMessage> pOrder(pSI->newSkel("D"));
```

```
        // set ClOrdID
        pOrder->set(FIXField::ClOrdID, "USR20000101" );
        pOrder->set(FIXField::HandlInst,    "2");
        pOrder->set(FIXField::Symbol,     "IBM");
        pOrder->set(FIXField::OrderQty,   "200");
        pOrder->set(FIXField::Side,         "1");
        pOrder->set(FIXField::OrdType,      "5");
        pOrder->set(FIXField::TimeInForce,  "0");
        //++++++++++++++++++++
        // Send order to session initiator
        pSI->put(pOrder.get());
        // Close sessions
        pSI->disconnect();
        pSA->disconnect();
        // release resources
        pSA->registerApplication(NULL);
        pSI->release();
        pSA->registerApplication(NULL);
        pSA->release();
        Engine::FixEngine::destroy();
    }
    catch(std::exception const& ex)
    {
        cout << " ERROR: " << ex.what() << endl;
        return -1;
    }
}
```