FIX Antenna Quick Start Guide

- Overview
- Application
  - Engine initialization
  - Session creation
    - Creation of session-acceptor
    - Creation of session-initiator
  - Creating new order
  - Sending order
  - Processing incoming message
  - Closing session
  - Full sample
  - Scheduler QuickStart Guide

Overview

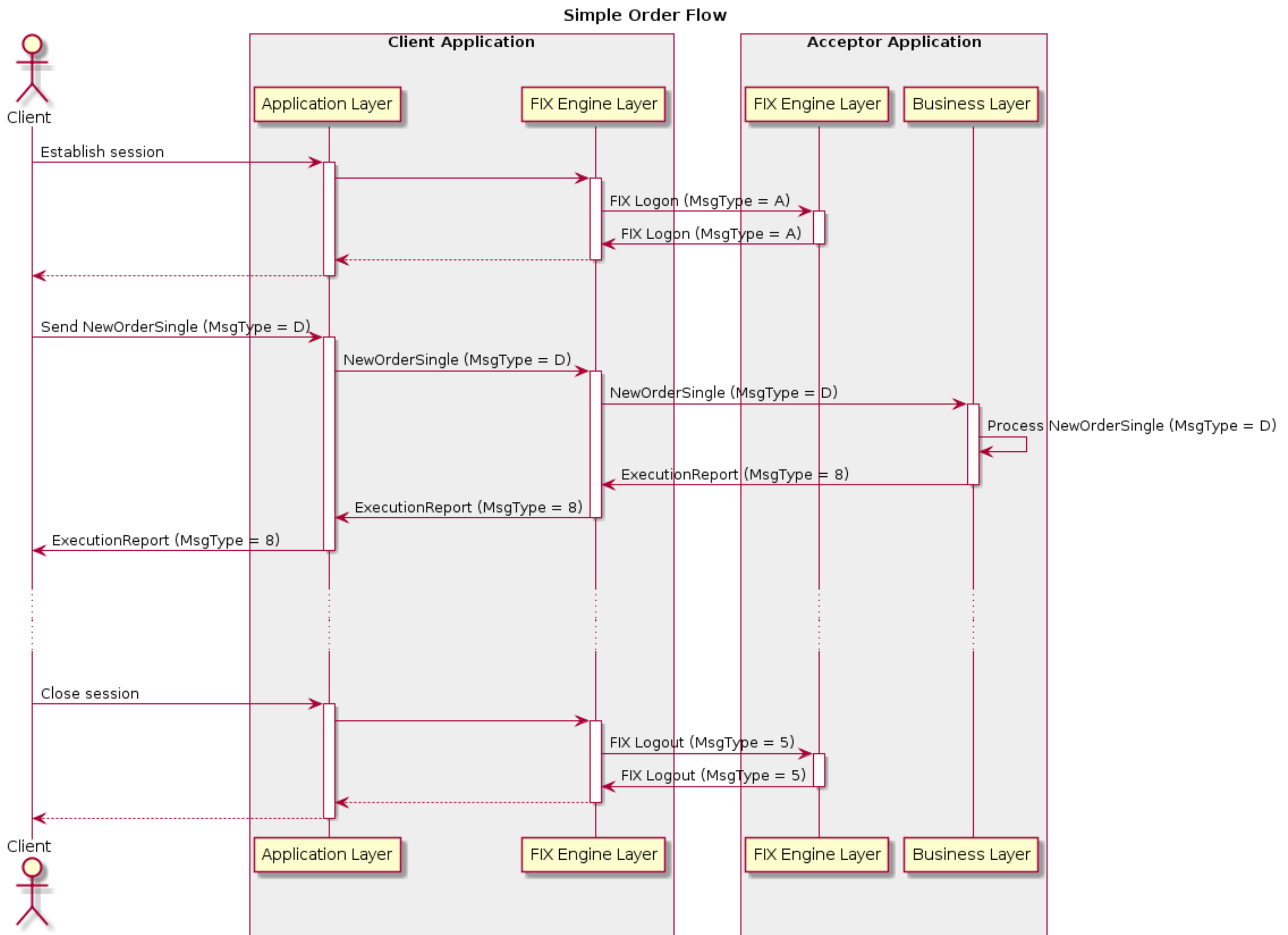
This article will be of interest to those who are coming face to face with FIX Antenna for the first time, and want to get a hands-on understanding of it.

It will help those building their first FIX Antenna-based application be aware of the possibilities FIX Antenna provides.

Application

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

For example, assume that a user needs to build an app that will be able to establish and terminate sessions, and support a simple Order Flow, presented below:



The main steps that need to be implemented in order to cover these functionalities are:

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

The following section describes how to create the application step-by-step with code examples. Follow the 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 the full path is not specified, the FIX Engine is looking in the current directory. Otherwise, the FIX Engine uses the full path to locate the properties file.

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

If an error occurs during initialization (for example, the properties file cannot not be found, or 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 the application object (an observer entity for a session to receive events; see Processingincomingmessage for details)
- Create the 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 sessions in general and a FIX session acceptor description, refer to the sections: Session description and Creating FIX session acceptor.

Creation of session-initiator

You can create a session initiator in three steps:

- Create the application object (observer entity for session to receive events, see Processingincomingmessage for details)
- Create the 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 sessions in general and a FIX session initiator description, refer to the sections: Session description and Creating FIX session initiator.

Creating new order

FIX Antenna provides two interfaces for FIX message manipulations:

1. Flat model
2. Object model

The flat model is based on a simple get/set FIX Message 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 messages and repeating groups, refer to the sections Message description, Repeating Groups description and FIX Message.

Sending order

To send a message to a session you should call on the Engine::Session::put method. This method does not send a message but instead puts it in a queue, from which the message will be sent out via a 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 the method returns the message it 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 acceptor message and Send initatior message.

Processing incoming message

The application class is responsible for:

- Processing incoming messages from a 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 &)
...
```

To process incoming messages, create a new class derived from the Application class and override the Application::process method. If an incoming message is successfully processed, the method should return "true". If a message cannot be processed at the moment, it should return "false". If the Engine::Application::process method does not return "true" (i.e. returns "false" or throws an exception), the engine passes the same incoming message to the Application::process again and again until either the Engine::Application::process returns "true", or the permitted number of attempts is exceeded (DelayedProcessing.MaxDeliveryTries, refer to Common parameters for more information). The interval between attempts is specified by the DelayedProcessing.DeliveryTriesInterval property (refer to Common parameters for more information). If the number of unsuccessful attempts exceeds the amount specified by MaxDeliveryTries, a Logout message stating "Application is not available" is sent to the counter-party and the session is closed.

Other useful methods to override are: Engine::Application::onLogonEvent, Engine::Application::onLogoutEvent,Engine::Application::onSequenceGapEvent, Engine::Application::onSessionLevelRejectEvent, etc. These are the callback methods called on to notify about the corresponding session-level events. Note that all pure virtual methods of Application must be overridden (implementation can just be empty body). The Engine::Application::process method is called only when an 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 and override callback methods in Application and provide your own logic.

If you need to keep a FIX message for future processing, create and save a copy of the message instead of saving the original message. (Refer to the Engine::FIXMsgProcessor::clone method). Below is an example of a custom implementation of the 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 the Application class, refer to the section with the 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 disconnecting, refer to the sections: Session description, Disconnect, and Disconnect.

Full sample

The sample below illustrates all instructions mentioned above, combined into 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;
    }
}
```

Scheduler QuickStart Guide

FIX Antenna Scheduler is a main events execution component to carry out scheduled actions, usually periodic ones (for example, every day at 9:30 AM), that defines the timeline of events. This component schedules events as required and notifies schedules when needed about the events that have occurred (refer to the Scheduler section of the FA Programmer's Guide).

The sample below shows how a FIX Antenna C++ user can create and configure the FIX Scheduler.

Sample

```
// Initialize FE
...

// Get engine's maintained scheduler instance.
Scheduler* pScheduler = FixEngine::singleton()->getScheduler();

// Get configured sessions list.
Engine::FAProperties::ConfiguredSessionsMap sessions = FixEngine::singleton()->getProperties()-
>getConfiguredSessions();
// Iterate over sessions configured and bind them to schedules defined.
for(Engine::FAProperties::ConfiguredSessionsMap::const_iterator it = sessions.begin(); it != sessions.end();
++it)
{
    // Find Schedule
    SessionsSchedulePtr pSchedule;
    try
    {
        pSchedule = SessionsSchedulePtr(pScheduler->getSchedule(it->second.scheduleId_));
    }catch(const System::SchedulerException&)
    {
        // Not found
        continue;
    }

    if(!pSchedule)
    {
        // The schedule is not Sessions schedule derived instance.
        continue;
    }

    //Get session controller
    FixEngineSessionsControllerPtr pSessionController = FixEngineSessionsControllerPtr (pSchedule-
>getSessonsController());
    if(!pSessionController)
    {
        // Sessions controller is not FixEngineSessionsController derived instance. So we can't use it.
        continue;
    }

    // Register the session within controller
    // myApp - instance of Engine::Application derived class that handles session callbacks.
    pSessionController->registerSession(it->first, myApp, it->second);
}
```

To use sessions scheduling, one should configure schedule(s) via the engine.properties file or via API.
For more details see: Programmers Guide