Market Technologies Pty Ltd

Fix8Pro© Java™ Bindings

A. Introduction

This document provides a brief introduction to the Java bindings available with FIx8Pro.

The Fix8Pro Java Bindings provide a lightweight JNI wrapper around the core Fix8Pro engine enabling Java applications to use Fix8Pro with ease and at near native performance.

B. System Design

The FIx8Pro Java interface uses the Simplified Application Interface (SAI) which provides a lightweight wrapper around the Fix8Pro C++ library. 3rd party application libraries designed to provide cross platform and cross language support can easily write to the SAI specification.

The Java interface in Fix8Pro is used in conjunction with longname support built into Fix8Pro. This interface is a fully native Java via JNI methods.

Since the JNI interface accesses the library via direct library calls, bypassing the Java VM, performance is on a par with native execution.

B.1 Comparison to native Java FIX implementations

The Fix8Pro Java binding can best be thought of as a intelligent compromise between the flexibility of Java and the performance of C++.

While most native Java frameworks offer the functionality expected of a modern FIX component, the Fix8Pro Java binding offers the additional advantage of super high performance - performance that cannot be matched in a Java virtual machine.

	Fix8Pro SAI Java	**Other 3rd party products**
Codec performance	Near native C++ performance	Subject to VM indeterminacies
Memory	Uses glibc or native memory allocator	Uses VM allocator
Garbage collection	Completely bypassed	Subject to GC latency
FIX versions	Works with any, specified at runtime	Version specific
Language support	C++, Java, C#, Python and more	Java

B.2 Full schema driven support

The Fix8Pro Java binding offers the additional flexibility of dynamic loading of FIX schema libraries. This means that changes to the FIX variant spoken can be added without major changes to your applications.

Fix8Pro also supports custom on-the-fly field additions so that critical new fields can be used straight away.

C. Application Interface

The FIx8Pro Java interface uses the Simplified Application Interface (SAI) which provides a lightweight wrapper around the Fix8Pro C++ library.

1. The session interface

Your application must derive a session from either the `SimplifiedClient` or `SimplifiedServer` classes. These classes already extend the SAI `sa_interface`. The following definition shows how `SimplfiedClient` extends `sa_interface`.

```
package com.fix8;

public class SimplifiedClient extends sa_interface
{
  protected SimplifiedClient(long cPtr, boolean cMemoryOwn)
  {...}
  protected void finalize()
  {...}
  public synchronized void delete()
  {...}

  /* ctor */
  public SimplifiedClient(String libpath)
  {...}

  /* send a message */
  public boolean send(FIXMessage msg)
  {...}

  /* test if the session has shutdown */
  public boolean is_shutdown()
  {...}

  /* stop the session */
  public void stop()
  {...}

  /* create a FIX message with deep construction */
  public FIXMessage create_msg(String name, boolean deepctor)
  {...}

  /* create a message */
  public FIXMessage create_msg(String name)
  {...}

  /* start the session */
  public boolean start(String ns, String conf, String session)
  {...}
}
```

i. Extract of the Fix8 Java interface

The following definition shows how a client extends `SimplfiedClient`.

```
package com.fix8jsai.client;

import com.fix8jsai.FIXField;
import com.fix8jsai.FIXMessage;
import com.fix8jsai.SWIGTYPE_p_void;

import java.io.IOException;
import java.util.function.Predicate;

import static java.lang.Thread.sleep;

public class MyClient extends com.fix8jsai.SimplifiedClient
{
   /* ctor */
   public MyClient(String libpath)
   {...}

   /* your inbound message handler */
   public boolean inbound_handler(FIXMessage msg, SWIGTYPE_p_void closure)
   {...}

   /* create a NewOrderSingle message */
   public FIXMessage generate_new_order_single()
   {...}
}
```

ii. Sample user interface

2. Creating and sending messages

Using the message longname, your application can instantiate any message that has been defined in your xml schema, using create_msg(). You can then add fields to the message with add_field(). Note the field values can be either string literals, typed values or symbolic values defined in the xml enum (such as “BUY”). The field can be eitehr the symbolic field longname or the FIX field tag identifier. The following example creates a NewOrderSingle, and adds some fields (using a few of the different options described above). In this example we also demonstrate how you can add repeating groups and nested repeating groups to your message:

```
FIXMessage generate_new_order_single()
{
   FIXMessage nos = create_msg("NewOrderSingle");
   if (nos != null
       && nos.add_field("TransactTime", "now") // special value sets time to now
       && nos.add_field(38, 100.0) // OrderQty
       && nos.add_field("Price", 500.5)
       && nos.add_field("ClOrdID",
(++_oid).ToString(CultureInfo.InvariantCulture))
       && nos.add_field("Symbol", "BHP")
       && nos.add_field("OrdType", 1)
       && nos.add_field("Side", "BUY")
       && nos.add_field("TimeInForce", "GOOD_THROUGH_CROSSING")
       && nos.add_field("NoPartyIDs", 0)
       && nos.add_field("NoUnderlyings", 3)
       && nos.add_field("NoUnderlyings:1/UnderlyingSymbol", "BLAH")
       && nos.add_field("NoUnderlyings:1/UnderlyingQty", 100.0)
       && nos.add_field("NoUnderlyings:2/UnderlyingSymbol", "FOO")
       && nos.add_field("NoUnderlyings:2/NoUnderlyingSecurityAltID", 2)
       && nos.add_field(
"NoUnderlyings:2/NoUnderlyingSecurityAltID:1/UnderlyingSecurityAltID",
"UnderBlah")
       && nos.add_field(
"NoUnderlyings:2/NoUnderlyingSecurityAltID:2/UnderlyingSecurityAltID", "OverFoo")
       && nos.add_field("NoUnderlyings:3/UnderlyingSymbol", "BOOM")
       && nos.add_field("NoUnderlyings:3/NoUnderlyingStips", 5)
       && nos.add_field(
"NoUnderlyings:3/NoUnderlyingStips:1/UnderlyingStipType", "Reverera")
       && nos.add_field(
"NoUnderlyings:3/NoUnderlyingStips:2/UnderlyingStipType", "Orlanda")
       && nos.add_field(
"NoUnderlyings:3/NoUnderlyingStips:3/UnderlyingStipType", "Withroon")
       && nos.add_field(
"NoUnderlyings:3/NoUnderlyingStips:4/UnderlyingStipType", "Longweed")
       && nos.add_field(
"NoUnderlyings:3/NoUnderlyingStips:5/UnderlyingStipType", "Blechnod")
       && nos.add_field("NoAllocs", "1")
       && nos.add_field("NoAllocs:1/AllocAccount", "Account1")
       && nos.add_field("NoAllocs:1/NoNestedPartyIDs", 1)
       && nos.add_field("NoAllocs:1/NoNestedPartyIDs:1/NestedPartyID",
"nestedpartyID1")
       && nos.add_field("NoAllocs:1/NoNestedPartyIDs:1/NoNestedPartySubIDs", 1)
       && nos.add_field(
"NoAllocs:1/NoNestedPartyIDs:1/NoNestedPartySubIDs:1/NestedPartySubID",
"subnestedpartyID1"))
       return nos;

   FIXMessage.destroy(nos);
   return null;
}
```

iii. Creating a message and adding fields

We can now send the message by calling `send()`.

3. Handling inbound messages

Your session class must provide an override to the method `inbound_handler()`. When the session receives a message from the remote session, this method is called with the fully decoded message, on the session receiver thread. The following example shows a sample implementation:

```
package com.fix8j;

public class MyClient extends SimplifiedClient
{
   public boolean inbound_handler(FIXMessage msg, SWIGTYPE_p_void closure)
   {
     if (is_shutdown())
         return false;

      string result;
      msg.print(out result);
      System.out.println(result);
      try
      {
         if (msg.is_longname("ExecutionReport"))
         {
            System.out.println(msg.as_string("OrderID"));
            FIXField ordid = msg.find_field("OrderID");
            if (ordid.is_valid())
            {
               string oid;
               if (ordid.get_field(out oid))
                  System.out.println(":{0}", oid);
            }
         }
      }
      catch (Exception e)
      {
         System.out.println("exception: {0}", e.Message);
      }
      return true;
   }
}
```

iv. Handling an inbound message

4. FixPro compiler extensions

The Fix8Pro compiler provides an additional extension that can generate schema based enumerations. These can be included in your java application allowing you to use symbolic names for FIX values. For example:

```
//------------------------------------------------------------------------------
---
package TEX.Consts; // example generated from TEX FIX schema

//------------------------------------------------------------------------------
---
public final class AdvSide
{
   public final char BUY = 'B';
   public final char SELL = 'S';
   public final char TRADE = 'T';
   public final char CROSS = 'X';
}
//------------------------------------------------------------------------------
---
public final class Side
{
   public final char BUY = '1';
   public final char SELL = '2';
   public final char BUY_MINUS = '3';
   public final char SELL_PLUS = '4';
   public final char SELL_SHORT = '5';
   public final char SELL_SHORT_EXEMPT = '6';
   public final char UNDISCLOSED = '7';
   public final char CROSS = '8';
   public final char CROSS_SHORT = '9';
   public final char CROSS_SHORT_EXEMPT = 'A';
   public final char AS_DEFINED = 'B';
   public final char OPPOSITE = 'C';
   public final char SUBSCRIBE = 'D';
   public final char REDEEM = 'E';
   public final char LEND = 'F';
   public final char BORROW = 'G';
}
...
```

D. Using and configuring Fix8Pro

Configuring your application under Java is identical to Fix8Pro. The session xml configuration files are used identically.

All the Fix8Pro services, such as logging, persistence, session management and so forth are also available to your Java application.

Please contact us for pricing and licensing options.