AquaQ Analytics

FIX Shared Library

email:
support@aquaq.co.uk

web:
`www.aquaq.co.uk`

AquaQ Analytics

Revision History

Revision	Date	Author(s)	Description
0.1	May 6, 2015	Mark Rooney	Initial release of FIX engine & documentation

Contents

Chapter 1

Company Overview

AquaQ Analytics Limited is a provider of specialist data management, data analytics and data mining services. We also provide strategic advice, training and consulting services in the area of market-data collection to clients within the capital markets sector. Our domain knowledge, combined with advanced analytical techniques and expertise in best-of-breed technologies, helps our clients get the most out of their data.

The company is currently focussed on four key areas, all of which are conducted either on client site or near-shore:

- Kdb+ Consulting Services: Development, Training and Support;
- Real Time GUI Development Services;
- SAS Analytics Services;
- Providing IT consultants to investment banks with Java, .NET and Oracle experience.

The company currently has a headcount of 30 consisting of both full time employees and contractors and is actively hiring additional resources. Some of these resources are based full-time on client site while others are involved in remote/near-shore development and support work from our Belfast headquarters. To date we have MSAs in place with 6 major institutions across the UK and the US.

Chapter 2

Overview

2.1 Prerequisites

2.2 Features

This FIX engine provides many standard features including:

Easy KDB+ integration - Getting access to a full suite of FIX engine features is as simple as loading a library into your KDB+ application.

Supports Standard FIX Versions - The FIX engine supports all of the FIX standards out of the box and even allows you to use multiple different message formats at the same time.

Message Validation - All messages that are received by the FIX engine are validated against a specification before being processed. This ensures that client use the correct format and it simplifies your business logic.

Custom Message Formats - The engine supports custom specifications via XML. The specification format enables automatic message validation.

Guaranteed Delivery -

Message Replay - The engine support message replaying and .

Logging -

Runs on Linux & Windows - The adapter is designed so that it will easily build & run on both Windows and Linux.

Flexible API -

2.3 Architecture

Chapter 3

System Setup

3.1 Requirements

3.1.1 CMake

This project uses CMake 2.6+ [1]to support building across multiple platforms. The CMake toolchain will generate the build artefacts required for your platform automatically. This will typically leave just one or two commands to run manually depending on the platform that you are using.

3.1.2 Python (Optional)

A q file containing some mappings that can be useful when building FIX messages is provided in the bin/ folder. This file has been generated based on the constants that are present in the quickfix libary headers using the CppHeaderParser[2] library. you should make sure that either Python 2.7+ or Python 3.3+ is installed if you wish to regenerate these constants based on the contents of the header files.

3.1.3 KDB+

3.2 Building the Shared Library

The first step is to build the quickfix library itself for the target platform. The instructions for this can be found either on the GitHub page or on the QuickFIX website[3]. It may be difficult to build a 32 bit binary on a 64 bit environment and vice versa.

This library uses CMake in order to enable cross platform builds.

[1] `http://www.cmake.org/`

[2] `https://pypi.python.org/pypi/CppHeaderParser/`

[3] `http://www.quickfixengine.org/quickfix/doc/html/building.html`

3.3 Configuration

3.3.1 Adding New Message Schemas

After building and installing the shared library, you should see a bin/config folder that sits alongside the shared library. This contains all of the core configuration for both acceptors and initiators. The `spec` folder contains all of the schemas for each version of FIX that is supported.

q/kdb+

```
~/fixengine/bin/config/spec> ls
FIX.4.0.xml
FIX.4.1.xml
FIX.4.2.xml
FIX.4.3.xml
FIX.4.4.xml
FIX.5.0.xml
FIX.5.0.SP1.xml
CUSTOM.xml
```

The format of the schema file will be explained by examining the FIX 4.0 XML file.

xml

```
<fix major='4' type='FIX' servicepack='0' minor='0'>
        <header>
                <!-- Header Fields (Described in more detail below) -->
        </header>
        <messages>
                <!-- Message Fields (Described in more detail below) -->
        </messages>
        <trailer>
                <field name='SignatureLength' required='N' />
                <field name='Signature' required='N' />
                <field name='CheckSum' required='N' />
        </trailer>
</fix>
```

q/kdb+

```
/ Creating an acceptor that will validate all of its messages against the FIX.4.4.xml schema file.
.fix.create[`acceptor;(enlist `version)!enlist `FIX.4.4]
/ Creating a second acceptor that is on a different port and
```

3.3.2 Modifying the default configuration file

There are two configuration files in the `config` directory of the FIX engine. The first is `acceptor-config.xml` which contains configuration that is specific to all the acceptors and the `initiator-config.xml` which contains configuration that is specific to all the initiators. Both of these xml files are made up of the same sections `[DEFAULTS]` and `[SESSION]`.

The `DEFAULTS` section of the of the configuration is where you can specify the common settings for all the acceptors that will be launched in the q session.

ini/config
```
# Defaults that should be shared across all sessions
# (they can be overridden on a per-session basis)
[DEFAULT]
BeginString=FIX.4.4
ConnectionType=acceptor
ReconnectInterval=60
SenderCompID=sellside
TargetCompID=buyside1
SocketNodelay=Y
```

The `SESSION` sections of the configuration correspond to a specific acceptor that should be launched automatically when the library is loaded. You may have multiple sessions configured as long as they do not share a SessionID or try to bind to the same port. The SessionID is made up of the `BeginString`, `SenderCompID` and `TargetCompID` and the binding port is defined by `SocketAcceptPort`. For initiators, you can have multiple connections to the same SessionID open as long as your specify the SessionQualifier.

ini/config
```
[SESSION]
# Statically creating an acceptor from a configuration file
StartTime=00:30:00
EndTime=23:30:00
ReconnectInterval=30
HeartBtInt=15
TargetCompID=buyside1_44
SocketAcceptPort=7070
SocketReuseAddress=Y
DataDictionary=config/spec/FIX44.xml
AppDataDictionary=config/spec/FIX44.xml
FileStorePath=store
PersistMessages=Y
FileLogPath=logs
```

3.3.3 Enabling/Disabling Message Verification

3.3.4 Logging

Logging is supported via directing message to stdout, to a flat file, or to a MySQL and PostgreSQL databases. This logging can be configured in the ini files from the previous section or at runtime. If you scroll to the bottom of the configuration file, you should see three sections for each.

ini/config
```
        ##############################################
        #       Plain Text Logging Configuration
        ##############################################
        FileLogPath=logs
        FileLogBackupPath=logs/backup
        FileStorePath=store
```

ini/config
```
################################################
#       MySQL Logging Configuration
################################################
MySQLLogDatabase=quickfix
MySQLLogUser=root
MySQLLogPassword=pass
MySQLLogHost=localhost
MySQLLogPort=20017
MySQLLogUserConnectionPool=N
```

The settings for a PostgreSQL database configuration mirror those for the MySQL version. Again, it is possible to split the guaranteed delivery storage from your debug logs by explicitly specifying the variables e.g `PostgreSQLStoreDatabase` alongside `PostgreSQLLogDatabase`.

ini/config
```
################################################
#       PostgreSQL Logging Configuration
################################################
PostgreSQLStoreDatabase=quickfix
PostgreSQLStoreUser=root
PostgreSQLStorePassword=pass
PostgreSQLStoreHost=localhost
PostgreSQLStorePort=20017
PostgreSQLStoreUseConnectionPool=N
```

3.3.5 Guaranteed Delivery/Message Replay

ini/config

```
# Defaults that should be shared across all sessions
# (they can be overridden on a per-session basis)
[DEFAULT]
BeginString=FIX.4.4
ConnectionType=acceptor
ReconnectInterval=60
SenderCompID=sellside
TargetCompID=buyside1
SocketNodelay=Y

[SESSION]
# Statically creating an acceptor from a configuration file
StartTime=00:30:00
EndTime=23:30:00
ReconnectInterval=30
HeartBtInt=15
TargetCompID=buyside1_44
SocketAcceptPort=7070
SocketReuseAddress=Y
DataDictionary=config/spec/FIX44.xml
AppDataDictionary=config/spec/FIX44.xml
FileStorePath=store
PersistMessages=Y
FileLogPath=logs
```

3.4 Start Up & Testing

3.4.1 Running the Unit Tests

The run script can take a parameter that indicates whether or not the unit tests should be run. The tests will print their output to the screen and also return an exit code that indicates if they were successful. If you wish to add your own tests to the suite, just modify the tests.csv file that is located in the `test/q` directory.

3.4.2 Loading the Shared Library

In order to load the shared library, we will use the dynamic load (2:) function. This function is dyadic and takes a name/path to a library as its first argument and a list as its second argument. The list should contain the name of the function that you wish to dynamically link against and the number of arguments that it takes. The library provides a bootstrapping function: `extern "C" K load_library(K x);`

This bootstrapping function will return a dictionary that maps symbols to dynamically linked functions. The function will erase the contents of any namespace that it is assigned to, so if you want to add additional variables or functions, you will need to place them after the code that loads the shared library.

q/kdb+
```
/ We load the entire library in one go by loading the load_library
/ function and then executing it.
.fix:(`:lib/fixengine 2:(`load_library;1))`
```

q/kdb+
```
/ Link against the create_adaptor function of the shared
/ library called fixengine.[so|dll] that is located in the
/ in the lib/ directory relative to the q session.
.fix.create:`:lib/fixengine 2:(`create_adaptor;2)

/ We can then use the .fix.create function just as though it was
/ built into kdb+
.fix.create[`acceptor;()!()]
```

3.4.3 Starting Servers (Acceptors)

q/kdb+
```
/ We can then use the .fix.create function just as though it was
/ built into kdb+
.fix.create[`acceptor;()!()]
.fix.create[`acceptor;(enlist `port)!enlist 7072]
.fix.create[`acceptor;`port`version!(7072;`FIX.4.2)]
```

It is also possible to configure the acceptors from an ini file that is located in the configuration directory. The syntax for these files is described in a previous section. An example configuration file that will launch two acceptors, one on port 7070 and another on port 7071 is shown below.

ini/config
```
# Define some settings that will be common between the two individual
# sessions.
[Default]
BeginString=FIX.4.4
ConnectionType=acceptor
DataDictionary=config/spec/FIX.4.4.xml
AppDataDictionary=config/spec/FIX.4.4.xml
PersistMessages=Y
SocketNodelay=Y

# Create the first session -- remembering to define a unique TargetCompID,
# SenderCompID & SocketAcceptPort.
[Session]
TargetCompID=SessionOneSellerID
SenderCompID=SessionOneBuyerID
SocketAcceptPort=7070
StartTime=00:30:00
EndTime=23:30:00

# Create the second session -- again we need to ensure our SessionID (which is
# in the format BeginString:SenderCompID->TargetCompID) is unique and that we
# have our own port.
[Session]
TargetCompID=SessionTwoBuyerID
SenderCompID=SessionTwoSellerID
SocketAcceptPort=7071
StartTime=05:45:00
EndTime=21:00:00
```

3.4.4 Starting Clients (Initiators)

q/kdb+
```
/ The initiator is built in the same fashion as the the acceptor
/ aside from the first parameter. Most of the arguments to the
/ dictionary are the same -- any that are not relevant to the
/ initiator will be ignored.
.fix.create[`initiator;()!()]
.fix.create[`initiator;(enlist `port)!enlist 7072]
.fix.create[`initiator;`port`version!(7073;`FIX.4.2)]
```

TODO :: Add that it is possible to configure an initiatior to start up statically from the config file

3.4.5 Sending a FIX Message

In order to send a FIX message from an initiator to an acceptor, you can use the `.fix.send[sid;dictionary]` function. When you first created an initiator, the function returned an integer that represents the **session id**. You can pass this identifier as the first parameter to the `.fix.send` function to indicate who the message should be sent to.

q/kdb+
```
/ Create an acceptor and an initiator which default to
/ connecting over localhost. The initiator will return
/ an sid that can be used to send messages from the
/ initiator to the acceptor.
q) .fix.listen[`TargetCompID`SenderCompID!`BuySideID`SellSideID]
q) .fix.connect[`SenderCompID`TargetCompID!`BuySideID`SellSideID]
/ We can then send the message using the sid (assuming that
/ the message dictionary has been defined beforehand)
q) message: (8 11 35 46 ...)!(`FIX.4.4;175;`D;8 ...)
q) .fix.send[message]
```

The FIX message itself should just be a dictionary that maps integers (that correspond to the tags defined in the specification) to kdb+ types. The shared library will handle to conversion of the kdb+ type to the format expected by FIX before sending. You can also just pass symbols with the data pre-formatted to the dictionary and the library will pass it straight through to the FIX engine.

q/kdb+
```
q) message:(8 11 35 46 ...)!(`FIX.4.4;175;`D;8; ...)
```

Some extra constants are provided to make the building of FIX messages a bit easier. The example below shows how you would typically send a NewOrderSingle("D") message using the symbol format:

q/kdb+
```
.fix.send_new_single_order: {[sid]

        message:()!()
        message[.fix.tags.BeginString]: `$"FIX.4.4"
        message[.fix.tags.SenderCompID]: `SellSideID;
        message[.fix.tags.TargetCompID]: `BuySideID;
        message[.fix.tags.MsgType]: `D;
        message[.fix.tags.ClOrdID]: `$"SHD2015.04.17"
        message[.fix.tags.Side]: `1;
        message[.fix.tags.TransactTime]: `$"20150417-17:38:21";
        message[.fix.tags.OrdType]: `1;

        .fix.send[sid; message];
}
```

3.4.6 Receiving a FIX Message

To receive a FIX message, you will want to use the **.fix.fromapp** callback. This callback should be defined with a single argument: e.g. `.fix.fromapp:{[dict] ... }`. The fromapp callback will be triggered for every *non-administrative* message that is received. You should check the message type field of the FIX version and Message Type of the dictionary which will be always present.

```
q/kdb+
/ Creating a callback that will just print out the dictionary
/ to console.
q) .fix.fromapp{[dict] 0N!dict; }
...
q)
/ Sample output when the callback is triggered
8|  `FIX.4.4
9|  112
35| `D
34| 188
49| `SellSideID
...
```

Chapter 4

Shared Library API

4.1 Functions

4.1.1 .fix.setdefaults

The .fix.setdefaults function is used to override any settings in the configuration file at runtime. It takes a dictionary of symbols -> kdb+ types and will convert the keys to uppercase before storing them in the settings. This means that the keys are case insensitive and also so that it is consistent with how the rest of the settings are stored in quickfix.

```
q/kdb+
q) .fix.setdefaults[(enlist `SenderCompID)!(enlist `NewCompID)]
```

Any keys that are not relevant to the acceptors or initiators will be still stored in the defaults dictionary, but ignored by the library. As long as the keys of the dictionary are a list of symbols, then this call will always succeed.

4.1.2 .fix.getdefaults

The .fix.getdefaults functions just returns a dictionary of symbols -> symbols that contains a merged view of the configuration files and the global default set via .fix.setdefaults. The values that have been set using .fix.setdefaults take precedence over the configuration file settings.

```
q) .fix.getdefaults[]
APPDATADICTIONARY| config/spec/FIX.4.4.xml
BEGINSTRING      | FIX.4.4
CONNECTIONTYPE   | acceptor
DATADICTIONARY   | config/spec/FIX.4.4.xml
ENDTIME          | 23:30:00
FILELOGBACKUPPATH| logs/backup
FILELOGPATH      | logs
FILESTOREPATH    | store
HEARTBTINT       | 15
PERSISTMESSAGES  | Y
RECONNECTINTERVAL| 60
SOCKETACCEPTPORT | 7070
SOCKETNODELAY    | Y
STARTTIME        | 00:30:00
```

4.1.3 .fix.listen

The .fix.listen function will create an acceptor that can bind to a socket and communicate with initiators. If it is passed no arguments it will take its arguments from the configuration file and the global runtime defaults. Any arguments that are not relevant to quickfix will just be ignored.

q/kdb+

```
/ This creates an acceptor with all the default arguments.
.fix.listen[]
/ We can also override the different options
.fix.listen[(enlist `BeginString)!enlist `FIX.4.4]
.fix.listen[`SenderCompID`TargetCompID!`BuySideID`SellSideID]
```

ini/config

```
BeginString       # The FIX version to use (e.g FIX.4.4 or FIX.5.0)
SenderCompID      # Your ID as agreed with the counterparty
TargetCompID      # The counterparty ID
SessionQualifier  # Used to differentiate between multiple sessions.
ConnectionType    # Can be either acceptor or initiator.
StartTime         # The time that the session should become active every day.
EndTime           # The time that the session should deactivate at every day.
StartDay
EndDay
UseDataDictionary # If set to N, then message validation will be disabled.
DataDictionary     # The data dictionary to use for validation.
```

4.1.4 .fix.connect

The `.fix.connect` function will create an initiator that can connect to acceptors, which could be created using `.fix.listen` (section 4.1.3) or be provided by a counterparty. If the `` `BeginString `` property is set and `` `DataDictionary `` is not in the defaults, then we automatically set the `` `DataDictionary ``,`` `AppDataDictionary `` to match the begin string so we can enable automatic message validation.

q/kdb+

```
.fix.connect[]
.fix.connect[`BeginString`TargetCompId!`FIX.4.2`BuySideID]
```

Figure 4.1: *Connecting to a fix adaptor using connecting to a counterparty fix adaptor*

The function takes either no arguments (in which case it will pull all of its configuration from the default config file), or a dictionary of options that should be used in place of the defaults. If the connection is not successful, the initiator will continue to try to connect to the until.

4.1.5 .fix.send

The `.fix.send` function takes a dictionary that maps long integers to any other kdb+ type. The dictionary should contain all the correct FIX tags and a session associated with the TargetCompID and the SenderCompID should exist. The session should have been created within the same kdb+ instance using `.fix.connect` (section 4.1.4) or `.fix.listen` (section 4.1.3). If a session that is associated with the message doesn't exist, then a 'session error will be raised and a more detailed message will be written to stderr.

q/kdb+

```
q) message:(8 35 46 ...)!(`FIX.4.4;`D;`SellerID ...)
q) .fix.send[message]
```

Figure 4.2: *Sending a FIX message by creating a dictionary and filling it with tags.*

The `.fix.send` function will use the BeginString, SenderCompID, and TargetCompID fields in order to lookup which session the message is intended for. This means that these three fields must be set in every message dictionary.

If the message has an invalid header or doesn't contain all the required fields, then a ‘config error will be raised in kdb+ and a more detailed error message will be sent to stderr. The validation is performed by the FIX engine that is recieving the message and not locally.

4.1.6 .fix.onrecv

The `.fix.onrecv` function can be defined to take a dictionary and will be called for each non-administrative message that is recieved (i.e. it won't be called for heartbeats, login, logout as these should be automatically handled by the quickfix library). It will typically be called after one of the participants in a session has used `.fix.send` (section 4.1.5).

```
q/kdb+
/ Defining a handler that will just print the dictionary to
/ screen. It will be called each time a non-administrative
/ message is recieved.
q) .fix.onrecv:{[dict] show dict; }
...
8 | FIX.4.4
9 | 112
35| D
34| 250
49| SellSideID
```

Figure 4.3: *Recieving a FIX message using the .fix.onrecv handler. Each FIX field is mapped to a dictionary entry for easy consumption.*

You may add any logic that you want inside the callback function to handle the dictionary, including sending response messages, disconnecting clients or storing the data in a kdb+ database.

4.1.7 .fix.getsessions

The .fix.getsessions function returns a table of all the currently running sessions within the q process and some of their configuration settings. A session corresponds to a single acceptor or initiator that has been launched with `.fix.listen` (section 4.1.3) or `.fix.connect` (section 4.1.4). Sessions that are no longer running will be removed from this table.

```
q/kdb+
/ The .fix.getsessions function returns a standard kdb+ table containing the details
/ of all the active sessions.
q) .fix.getsessions[]
beginString senderCompID targetCompID sessionQualifier isInitiator ...
------------------------------------------------------------------ ...
FIX.4.4     BuySideID    SellSideID                    0           ...
FIX.4.4     SellSideID   BuySideID    SB2015050100     1           ...

/ We can query the contents of the table to extract all the sessions that are using
/ FIX 4.4 for example.
q) select senderCompID, targetCompID from .fix.getsessions[]
                where beginString = `$"FIX.4.4"
```

Figure 4.4: *Viewing and querying the FIX session table*

4.2 Enumerations

4.2.1 .fix.tags

The .fix.tags dictionary has been built from the C++ headers in order to make building/decoding FIX messages by hand easier. To load this dictionary just load the fixenums.q script into your session **after** loading the shared library.

```
q/kdb+
q) .fix.tags
               | ::
BeginSeqNo     | 7
BeginString    | 8
BodyLength     | 9
CheckSum       | 10
EndSeqNo       | 16
MsgSeqNum      | 34
MsgType        | 35
...

/ We can look up which integer maps to a specific tag e.g. BeginString
q) .fix.tags.BeginString
8

/ A reverse lookup can also be performed using the find (?)
/ operator as each integer maps to a single tag.
q) .fix.tags?8
`BeginString
```

Figure 4.5: *Using the .fix.tags dictionary to map between the integer form & the human readable representation*

The `fixenums.q` file has been generated by a python script located in `src/python` in the project. To create a new version of the `fixenums.q` file, copy the headers from the `include/quickfix` directory into the same directory as the python script and then run **parseheaders.py**.

4.2.2 .fix.types

maps to the types that are associated with each tag

4.2.3 .fix.values

The values dictionary contains several nested dictionarys that contain constants that have been defined in the standard. These constants should be used where possible in order to remain portable. It is also possible to use these fields to "decode" the FIX messages as before using the find operator.

q/kdb+

```
q) .fix.values.msgtype
               | ::
Heartbeat      | `0
TestRequest    | `1
ResendRequest  | `2
Reject         | `3
...
q) .fix.values.msgencoding
               | ::
ISO2022JP      | `ISO-2022-JP
EUCJP          | `EUC-JP
...
q) .fix.values.orderstatus
               | ::
New            | `0
PartiallyFilled| `1
Filled         | `2
DoneForDay     | `3
Canceled       | `4
PendingNew     | `A
```