

CAPLIN

Caplin Xaqua 1.0

Integrating Caplin Xaqua With A Trading System

February 2011

CONFIDENTIAL

Contents

1	Preface.....	1
1.1	What this document contains.....	1
	About Caplin document formats	1
1.2	Who should read this document.....	1
1.3	Related documents.....	2
1.4	Typographical conventions.....	3
1.5	Feedback.....	3
1.6	Acknowledgments.....	4
2	Overview.....	5
3	Trading concepts.....	7
3.1	Trade Models.....	7
3.2	Trade Channels	7
3.3	Trades.....	7
3.4	Trade Events.....	8
3.5	Blotter Channels	8
4	Example Trade Models.....	10
4.1	Example Executable Streaming Price (ESP).....	10
4.2	Example Request for Stream (RFS).....	11
4.3	Example Order (ORD).....	12
4.4	Request for Quote (RFQ) with timeouts	13
5	Configuring Trade Models.....	14
5.1	RFQ example.....	15
6	Using the Trading DataSource Java API.....	16
6.1	Initialization.....	16
6.2	New Trade Channels.....	17
6.3	New Trades	17
6.4	Dealing with Events.....	18
6.5	Closing Trades.....	19
6.6	Closing Channels.....	19
6.7	Handling Blotter Channels.....	20
	Registering the BlotterTradeListener	20
	Implementing the BlotterTradeListener interface	22
6.8	Handling Trade Restorations.....	24

Restoring Trades that still exist on the client	25
7 The Java Trading DataSource Example.....	26
8 Using the Trading DataSource C++ API.....	27
8.1 Initialization.....	27
Logging	28
8.2 New Trades.....	29
8.3 New Trade Channels.....	29
8.4 Dealing with Events.....	30
8.5 Closing Trades.....	31
8.6 Closing Channels.....	31
8.7 Handling Blotter Channels.....	32
Registering the BlotterListener	32
Implementing the BlotterListener interface	33
8.8 Handling Trade Restorations.....	34
Restoring Trades that still exist on the client	35
9 The C++ Trading DataSource Example.....	36
10 Configuring Caplin Liberator for trading.....	37
10.1 Mapping trade messaging objects in Liberator.....	38
10.2 Routing trade messages to the Trading DataSource	40
10.3 Trading performance and integrity.....	41
Optimizing client reconnection time	41
Throttling	41
10.4 Security.....	43
Session IDs	43
Single sign-on and KeyMaster	43
11 Appendix: Overview of throttling.....	44
12 Glossary of terms and acronyms.....	46

1 Preface

1.1 What this document contains

This document describes how the Caplin Trading DataSource allows you to integrate Caplin Xaqua with your existing trading system. It explains trading concepts (such as Trade Models), how to implement a Trading DataSource, and how to configure Caplin Liberator for trading.

Trading DataSources can be implemented in Java™ and C++. The document describes how to use both the Java and C++ APIs for this purpose.

Note: The diagrams and examples in this document show the Caplin Xaqua client to be a Caplin Trader application, but the Trading DataSource can serve any Caplin Xaqua client application.

About Caplin document formats

This document is supplied in three formats:

- ◆ Portable document format (*.PDF* file), which you can read on-line using a suitable PDF reader such as Adobe Reader®. This version of the document is formatted as a printable manual; you can print it from the PDF reader.
- ◆ Web pages (*.HTML* files), which you can read on-line using a web browser. To read the web version of the document navigate to the *HTMLDoc_m_n* folder and open the file *index.html*.
- ◆ Microsoft HTML Help (*.CHM* file), which is an HTML format contained in a single file. To read a *.CHM* file just open it – no web browser is needed.

For the best reading experience

On the machine where your browser or PDF reader runs, install the following Microsoft Windows® fonts: Arial, Courier New, Times New Roman, Tahoma. You must have a suitable Microsoft license to use these fonts.

Restrictions on viewing .CHM files

You can only read *.CHM* files from Microsoft Windows.

Microsoft Windows security restrictions may prevent you from viewing the content of *.CHM* files that are located on network drives. To fix this either copy the file to a local hard drive on your PC (for example the Desktop), or ask your System Administrator to grant access to the file across the network. For more information see the Microsoft knowledge base article at <http://support.microsoft.com/kb/896054/>.

1.2 Who should read this document

This document is intended for Technical Managers, System Architects, and Developers who need to understand the trading concepts used in Caplin Xaqua. The second half of the document also explains how Developers can implement a Trading DataSource in Java or C++, and how to configure Caplin Liberator for trading.

1.3 Related documents

- ◆ **Caplin Xaqua: Overview**

Provides a business and technical overview of Caplin Xaqua and includes an explanation of its architecture.

- ◆ **Caplin DataSource Overview**

A technical overview of Caplin DataSource.

- ◆ **Caplin Xaqua: Trading DataSource Java API Reference**

This is the detailed Java API documentation for the Caplin Trading DataSource.

- ◆ **Caplin Xaqua: Trading DataSource C++ API Reference**

This is the detailed C++ API documentation for the Caplin Trading DataSource.

- ◆ **Caplin Xaqua: Trade Model Configuration XML Reference**

This document defines the XML tags and attributes used to define Trade Models.

(Older versions of this document are called
Caplin Trader: Trade Model Configuration XML Reference.)

- ◆ **Caplin DataSource for C API Reference**

The API reference documentation for the C DataSource SDK.

- ◆ **Caplin Trader: API Specification**

Documents the JavaScript libraries that allow developers to extend Caplin Trader by writing custom JavaScript code.

- ◆ **Caplin KeyMaster Overview**

KeyMaster integrates Caplin Liberator with an existing single sign-on system, so that end-users do not have to explicitly log in to the Liberator server in addition to logging in to the enterprise's single sign-on server.

1.4 Typographical conventions

The following typographical conventions are used to identify particular elements within the text.

Type	Uses
aMethod	Function or method name
<i>aParameter</i>	Parameter or variable name
<i>/AFolder/Afile.txt</i>	File names, folders and directories
<div>Some code;</div>	Program output and code examples
The value=10 attribute is...	Code fragment in line with normal text
Some text in a dialog box	Dialog box output
Something typed in	User input – things you type at the computer keyboard
Glossary term	Items that appear in the “Glossary of terms and acronyms”
XYZ Product Overview	Document name
◆	Information bullet point
■	Action bullet point – an action you should perform

Note: Important Notes are enclosed within a box like this.
Please pay particular attention to these points to ensure proper configuration and operation of the solution.

Tip: Useful information is enclosed within a box like this.
Use these points to find out where to get more help on a topic.

Information about the applicability of a section is enclosed in a box like this.
For example: “This section only applies to version 1.3 of the product.”

1.5 Feedback

Customer feedback can only improve the quality of our product documentation, and we would welcome any comments, criticisms or suggestions you may have regarding this document.

Visit our feedback web page at <https://support.caplin.com/documentfeedback/>.

1.6 Acknowledgments

Adobe, *Adobe® Reader*, and *Flex* are either registered trademarks or trademarks of Adobe Systems Incorporated is a registered trademark of Adobe Systems Incorporated in the United States and/or other countries.

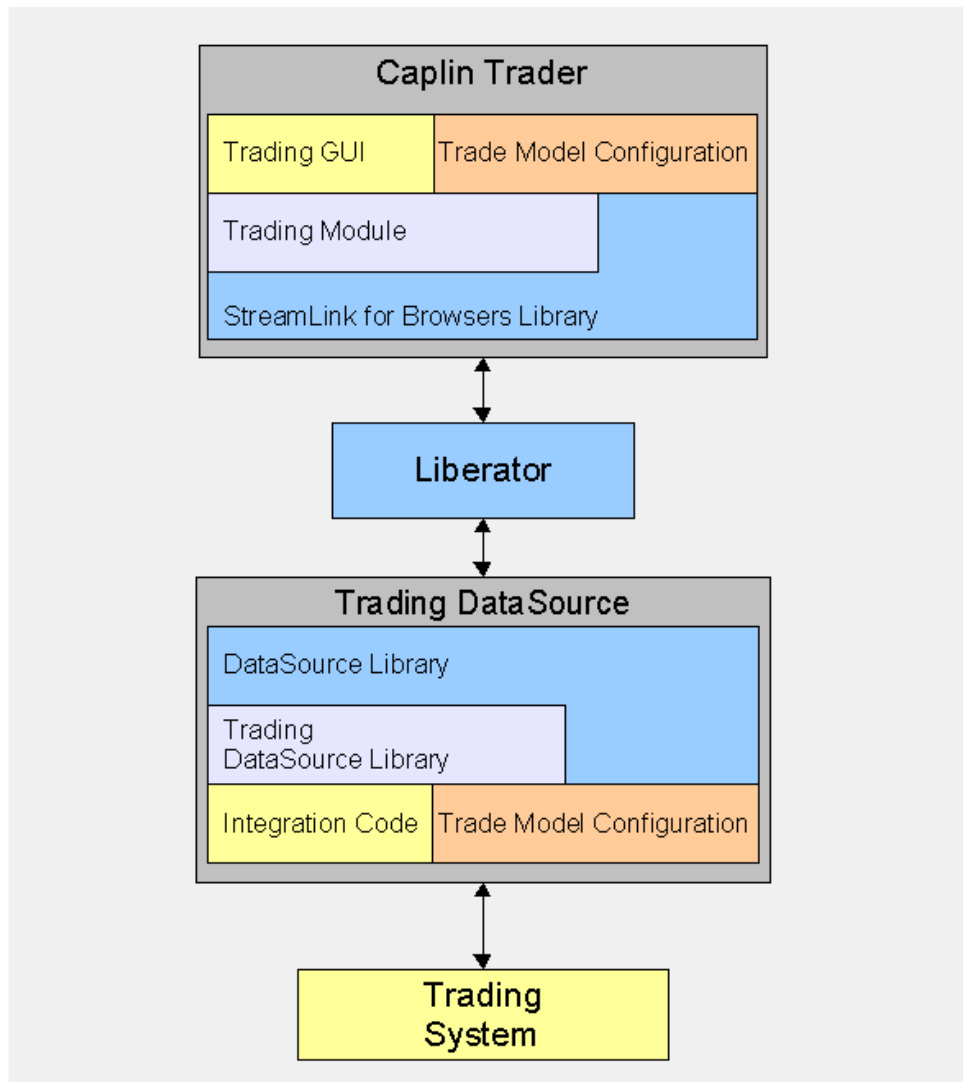
Windows is a registered trademark of Microsoft Corporation in the United States and other countries.

Java is a trademark of Sun Microsystems, Inc. in the U.S. or other countries.

2 Overview

Caplin Xaqua consists of a number of components (see the **Caplin Xaqua: Overview**). The main components used to integrate Caplin Xaqua with your trading system are the Trading DataSource and the Trading GUI.

The following diagram shows the basic architecture of the trading integration components and how they fit into Caplin Xaqua.



Simplified Caplin Xaqua architecture showing only trading integration components

Trading system

The trading system represents your systems that support trade capture and execution.

Trading DataSource

The Trading DataSource is the interface between Caplin Xaqua and the trading system. Its job is to enable communication between clients and the trading system. It sits between Caplin Liberator and the trading system, handling messages sent between clients and the trading system, via Caplin Liberator. The Trading DataSource consists of the standard Caplin DataSource Library, the Trading DataSource Library, and the custom code required to integrate with your trading system.

The Trading DataSource provides a simple API that can be used to communicate with your trading system or can be integrated directly into it. This means the DataSource can be a stand alone process or part of an existing one. The Trading DataSource API is available in both Java and C++ and is built on top of the Caplin DataSource library. It gives full access to all the functionality of the Caplin DataSource library, which allows you to send and receive custom messages in addition to trade messaging.

Trading GUI

The Trading GUI is the part of Caplin Trader that displays Trade Tickets and Trade Tiles. It can be customized to contain the correct information and understand the types of trades that can be performed. The Trading GUI consists of custom presentation code that interacts with the Trading DataSource through the Liberator using the client-side Trading Module and the StreamLink for Browsers Library.

The provided Trade Tickets and Trade Tiles can be used as a starting point for customization, or completely new trading displays can be created using the supplied APIs.

Trade Model configuration

The Trade Model configuration is a set of XML files defining the Trade Models that are to be used by the Trading DataSource and Trading GUI. These definitions represent the trade life-cycle and provide an interface between the end-user and the trading system. The same Trade Model configuration is used by both components to ensure they communicate and maintain a consistent state with one another.

Caplin Xaqua is not tied to any particular Trade Model; it can be configured to match your existing Trade Models along with any new Trade Models you wish to develop. Once configured with Trade Models, the Trading DataSource Library and Trading Module will control and verify the states and transitions allowed. This simplifies the integration needed as most of the logic is handled for you and is defined by your configuration.

3 Trading concepts

Caplin Xaqua uses a number of concepts to represent trading: trade models, trade channels, trades, and trade events.

3.1 Trade Models

A Trade Model represents a type of Trade, for example a Request for Quote (RFQ) or Executable Streaming Price (ESP). Trade Models consist of a number of states and transitions, and are defined by XML configuration. The Trade Model controls the flow of a Trade by defining all the possible states the Trade can be in, and the messages that cause transitions from one state to another.

3.2 Trade Channels

A Trade Channel represents a single end-user's communication between Caplin Xaqua and the Trading DataSource. It is a private channel for bidirectional messaging, and all messages relating to Trades for an end-user will be sent and received on the end-user's channel.

The Caplin Trader application opens a Trade Channel by subscribing to an object. Caplin Liberator maps this subscription to a unique object name for that end-user (see [Mapping trade messaging objects in Liberator](#) ⁽³⁸⁾) and subscribes to the object from the Trading DataSource. When the Trading DataSource responds to this subscription, a private channel is effectively created for messages in both directions between the client and the Trading DataSource; this is the Trade Channel.

Many deployments would use a single Trade Channel. However, sometimes it is useful to have multiple Trading DataSources, each handling different asset classes. In this case the end-user could have a separate Trade Channel for each Trading DataSource; the client would be set up to subscribe to different object names for the different channels.

3.3 Trades

A Trade represents a single trade for an end-user. This could be an RFQ, an execution on a streaming price, or any other type of trade. A Trade is typically initiated by the client. The Trading DataSource processes events from the client and the trading system that transition the Trade between different states.

Multiple Trades can be in operation on the same Trade Channel, either concurrently or one after the other. Each Trade has an associated RequestId set by the client and a TradeId set by the trading system. These ids are set by the first message sent by either side and are then included in subsequent messages to link the messages to the correct Trade.

A Trade is tied to a Trade Model; this relationship is determined by the first message sent by the client. Once the Trade Model for a Trade has been set, the state of the Trade transitions from the initial state to a final state according to the definition of the Trade Model.

3.4 Trade Events

A Trade Event typically represents an action by a client or an event from the trading system. An event originating from the trading system can represent an action by a dealer or an automated action. A Trade Event is raised either by receiving a message from a client, or directly which causes the DataSource to send a message to a client. Events are tied to a Trade and cause the Trade to move from one state to another, as defined by the Trade Model.

A Trade Event contains a number of fields and values, which map directly onto a message sent or received by the Trading DataSource. Some message fields are mandatory and are part of the Trading API; for example, all Trade Events have a type which is represented by the MsgType field in the underlying message. Other fields are optional, some of which may be required by the Trade Model being used.

3.5 Blotter Channels

A Blotter Channel is a software channel that the Trading DataSource uses to send information to the client for display in a blotter. In a Caplin Trader application the blotter is updated when a Trade changes state; for example; when a quote is requested, when a Trade is executed, and when a Trade is canceled.

The Caplin Trader application opens a Blotter Channel by subscribing to an object whose subject name begins with the string "/TRADEBLOTTER", or "/BLOTTER/", or where the subject name contains the string "/FT/TRADEHISTORY/". If the subject starts with "/TRADEBLOTTER", a container based blotter is created. If the subject starts with "/BLOTTER/", or contains the string "/FT/TRADEHISTORY/", a record-based blotter channel is created that works with older versions of the Caplin Trader grid.

The Liberator maps the Blotter Channel subscription to a unique object name for the Caplin Trader end-user and subscribes to the object from the Trading DataSource. The Trading DataSource responds to this subscription by creating a unique Blotter Channel.

Subsequently, whenever the end-user interacts with the trading subsystem, the Trading DataSource creates a Blotter Event each time the client is notified of a Trade Event. As a result it may send a blotter message to the client across the Blotter Channel. The message typically contains information about the state of the Trade.

The following pictures show the blotter in a Caplin Trader application being updated at successive stages in the execution of an FX Trade using the RFS Trade Model. (For simplicity only the left hand side of the blotter is shown; in reality there are more fields on the right hand side of the blotter.)

1. When the end-user requests a quote the blotter entry is created and its status is set to Opened.

FX Blotter							
ID	Currency Pair	Dealt Currency	Status	B/S	Amount	Rate	S/D
1235491222769	AUDUSD	AUD	Opened	2-WAY	500,000		26 Feb 2009

2. As price quotes are streamed to the client the status changes to Price Update.

FX Blotter							
ID	Currency Pair	Dealt Currency	Status	B/S	Amount	Rate	S/D
1235491222769	AUDUSD	AUD	Price Update	2-WAY	500,000		26 Feb 2009

3. When the end-user clicks the Sell button on the Trade Ticket, the blotter entry status becomes Executing.

FX Blotter							
ID	Currency Pair	Dealt Currency	Status	B/S	Amount	Rate	S/D
1235491222769	AUDUSD	AUD	Executing	SELL	500,000	0.9180	26 Feb 2009

4. Finally the Trade is confirmed and the final sale details are sent to the blotter with status set to Done.

FX Blotter							
ID	Currency Pair	Dealt Currency	Status	B/S	Amount	Rate	S/D
1235491222769	AUDUSD	AUD	Done	SELL	500,000	0.9180	26 Feb 2009

Successive updates to a blotter entry in Caplin Trader

The client should normally subscribe to a separate Blotter Channel for each asset class traded—for example, an FX blotter and an FI blotter — since the type of information that needs to be displayed on the blotter varies according to asset class. The client would be set up to subscribe to different object names for the different Blotter Channels, such as “/BLOTTER/FX” and “/BLOTTER/FI”.

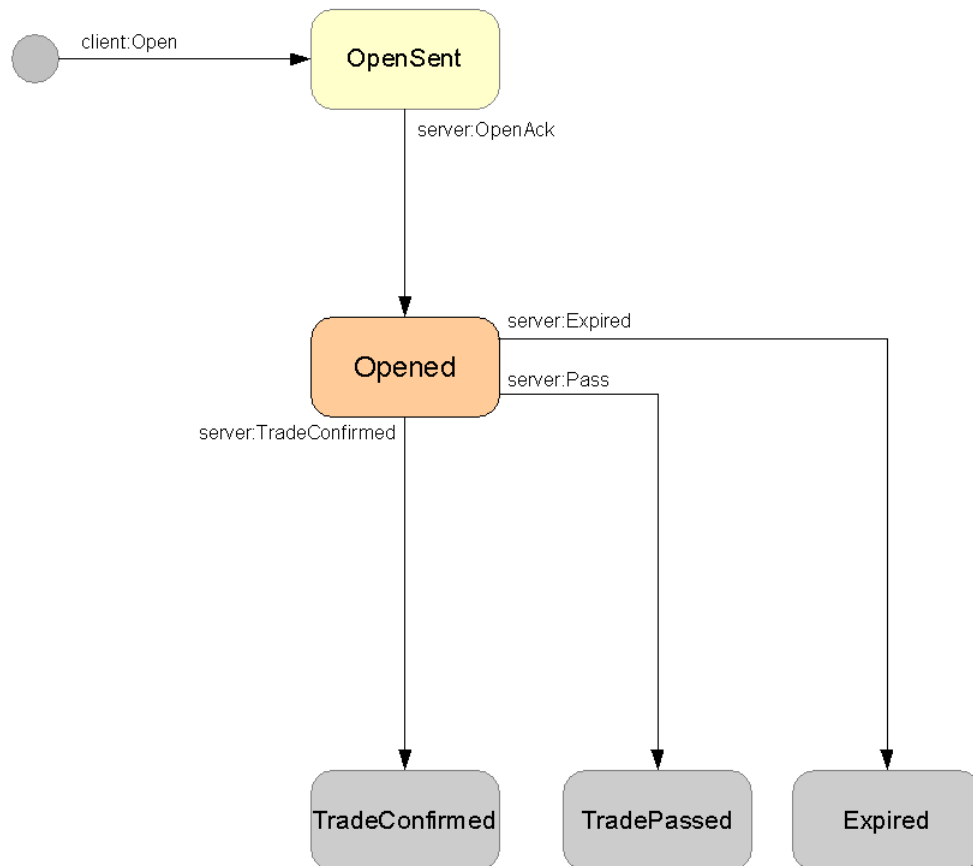
4 Example Trade Models

The following sections show examples of Trade Models that can be used with Caplin Xaqua. More complicated Trade Models can be used with little extra complexity of integration.

4.1 Example Executable Streaming Price (ESP)

This state diagram shows the example Executable Streaming Price (ESP) Trade Model.

This model is provided with Caplin Xaqua and is used by the demonstration Trading DataSource and the Caplin Trader Reference Implementation, for one-click trading via the Trade Tile.



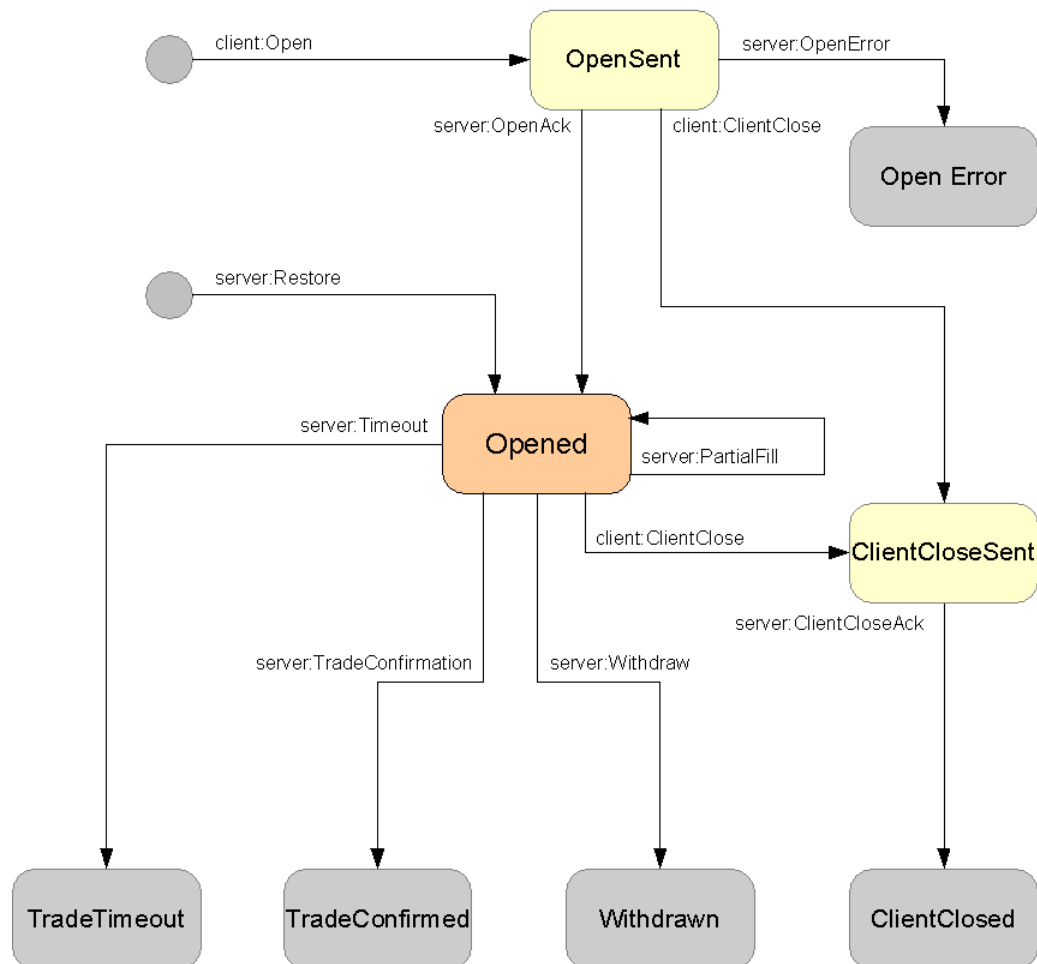
State diagram for ESP Trade Model

4.3 Example Order (ORD)

This state diagram shows the example Order (ORD) Trade Model.

This model differs from the ESP and RFS models in that it has two different transitions from the initial state. The standard transition is the client open event (client:Open), but the additional initial state transition (server:Restore) is used when the server restores a Trade from the trading system.

This model is provided with Caplin Xaqua and is used by the demonstration Trading DataSource. It is not currently used by Caplin Trader, but may be included in a forthcoming version of the Caplin Trader Reference Implementation for ticket-based order Trades.



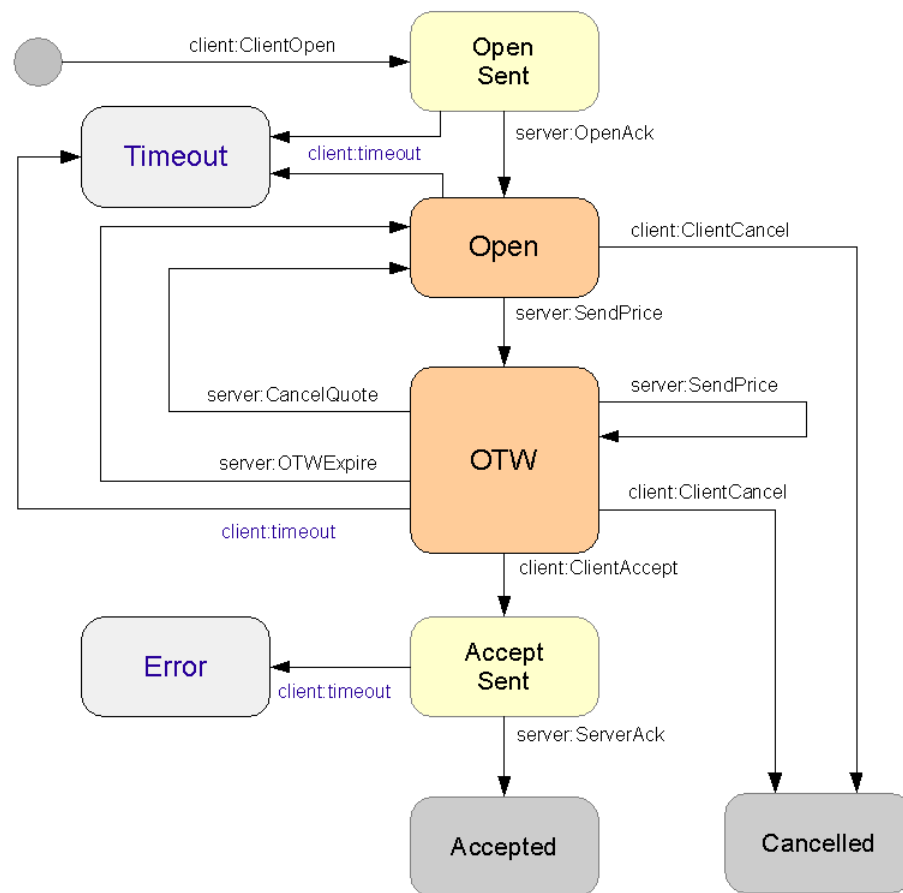
State diagram for Example ORD Trade Model

4.4 Request for Quote (RFQ) with timeouts

This state diagram shows a typical simple Request for Quote Trade Model.

This particular model is not used by the Caplin Trader Reference Implementation, but is typical of an RFQ workflow. It includes timeouts on events; these timeouts are implemented on the client side only, to ensure that the client does not hang if there is no response from the server. Additional states can easily be added before the Open state to account for credit checks and other validation steps as required.

The XML configuration that describes this trade model is shown in the [RFQ example](#)^[15] section of [Configuring Trade Models](#)^[14].



State diagram for RFQ Trade Model

5 Configuring Trade Models

Caplin Xaqua is designed to work with any Trade Model; it can be configured to match your existing Trade Models or new models being developed.

Trade Models are configured using an XML definition file which defines:

- ◆ The possible states a Trade can be in.
- ◆ The transitions a Trade can take from one state to another.
- ◆ The checks that are made before the transition can be made.

Any number of Trade Models can be configured and Trades can automatically pick out the relevant model to use.

The Trading DataSource kit includes example Trade Model XML definition files. These are Request for Stream (RFS), Executable Streaming Price (ESP), and Order (ORD). The XML definitions can be used as they are, or can be adapted to your requirements.

For the detailed definition of the Trade Model XML, see the document **Caplin Xaqua: Trade Model Configuration XML Reference**.

5.1 RFQ example

The following example is the XML configuration for the RFQ Trade Model shown in [Request for Quote \(RFQ\) with timeouts](#)^[13]. Note that the `timeout` and `timeoutState` attributes on some of the `<state>` tags only apply to the Trade Model when it executes on a client. The timeouts ensure that the client does not hang if there is no response from the server.

```
<tradeModels>
  <tradeModel name="RFQ" initialState="Initial">
    <state name="Initial">
      <transition target="OpenSent"
        trigger="ClientOpen"
        source="client" />
    </state>

    <state name="OpenSent" timeout="10" timeoutState="Timeout">
      <transition target="Open"
        trigger="OpenAck"
        source="server" />
    </state>

    <state name="Open" timeout="60" timeoutState="Timeout">
      <transition target="OTW"
        trigger="PriceUpdate"
        source="server" />
      <transition target="Cancelled"
        trigger="ClientCancel"
        source="client" />
    </state>

    <state name="OTW" timeout="60" timeoutState="Timeout">
      <transition target="AcceptSent"
        trigger="ClientAccept"
        source="client" />
      <transition target="OTW"
        trigger="PriceUpdate"
        source="server" />
      <transition target="Open"
        trigger="OTWExpire"
        source="server" />
      <transition target="Cancelled"
        trigger="ClientCancel"
        source="client" />
    </state>

    <state name="AcceptSent" timeout="10" timeoutState="Error">
      <transition target="Accepted"
        trigger="AcceptAck"
        source="server" />
    </state>

    <state name="Accepted" />
    <state name="Cancelled" />
    <state name="Timeout" />
    <state name="Error" />
  </tradeModel>
</tradeModels>
```

6 Using the Trading DataSource Java API

This section provides a brief description of the main parts of the Trading DataSource Java API (see the **Caplin Java Trading DataSource: API Documentation** for full details).

To implement a Trading DataSource you set up listener objects to handle events occurring on Trades and create events to be sent back into the system.

6.1 Initialization

When your Trading DataSource application starts, it should create an instance of `TradingDataSource` and register itself as a `TradingApplicationListener`. The `TradingDataSource` starts up a `DataSource`, which then connects to Liberator. The `DataSource` also sets up the necessary `DataSource` listeners internally to handle the trade messaging.

The following is a simple code extract showing the creation of a `TradingDataSource` within a custom application.

Creation of a TradingDataSource

```
public class MyTradingApp implements TradingApplicationListener
{
    static void main(String[] args)
    {
        new MyTradingApp();
    }

    MyTradingApp()
    {
        // Create a factory object for generating the trading state machines.
        StateMachineFactory myTradingStateMachineFactory = new StateMachineFactory();

        // Load the required Trade Models into the factory.
        myTradingStateMachineFactory.loadModels(new File("conf/MyESPStateModel.xml"));
        myTradingStateMachineFactory.loadModels(new File("conf/MyRFSStateModel.xml"));

        // Create the Trading DataSource.
        // This will create a DataSource object internally to manage the
        // communication with other DataSources, such as Caplin Liberator
        // and hence client applications.
        // TradingDataSource implements the standard DataSource callbacks,
        // which allow you to use the Trading API to communicate with
        // clients in the form of trade messages.

        tradingDataSource = new TradingDataSource
            (this, //Reference to this TradingApplicationListener
            "MyDataSourceConfig.xml", // Config file for this DataSource
            myTradingStateMachineFactory
            );

        // Once the TradingDataSource has been created
        // it has to be started explicitly.
        tradingDataSource.start();
    }

    // ...
}
```

The `TradingDataSource` processes the flow of a Trade by following the states defined by the Trade Model for the particular type of Trade. This processing is carried out by a state machine, implemented as a `StateMachine` object. Before creating the `TradingDataSource`, the Trading DataSource application creates a `StateMachineFactory`, and loads the factory with the required Trade Models. The models are defined in the XML configuration files discussed in [Configuring Trade Models](#)^[14]. The `StateMachineFactory` is then passed to the `TradingDataSource` constructor, so that the DataSource can create a state machine for each loaded Trade Model.

The `TradingDataSource` constructor is also supplied with an XML format configuration file for the DataSource. This file defines the connections that the DataSource makes with the other Caplin Xaqua components, such as Liberator.

6.2 New Trade Channels

The `TradingApplicationListener` is notified when new Trade Channels are created; this allows you to perform any necessary user specific initialization. You must also add a `ChannelListener` to the channel. The `ChannelListener` is an interface you must implement; it could be a new instance for each channel or a single global instance. Its job is to handle notifications on the channel about newly created Trades and Trades being closed.

The following code extract shows part of a sample implementation of a `TradingApplicationListener`. It shows a custom `ChannelListener` being added to the channel to handle Trades, and then a method being called on a hypothetical `tradingSystem` object to log in the end-user for the channel.

Example implementation of `TradingApplicationListener.channelCreated()`

```
public void channelCreated(TradeChannel channel)
{
    channel.setChannelListener(new MyChannelListener(channel));

    // Handle new channel/user.
    // For example:
    tradingSystem.loginUser(channel.getUser());
}
```

6.3 New Trades

The `ChannelListener` is notified of new Trades when a client initiates them; the Trade is passed to it as a Trade object. When a Trade is created the `ChannelListener` can perform any initialization needed with the trading system and also add a `TradeListener` to the newly created Trade object. The `TradeListener` is an interface you must implement and could be a new instance for each Trade or a single global instance; its job is to handle all events for a Trade. You normally have different implementations of `TradeListener` that handle different trade types.

The following code extract shows part of a sample implementation of a `ChannelListener`. It shows a different custom `TradeListener` being added to the Trade object to handle its events, depending on the Trade Model used for the Trade.

Example implementation of ChannelListener.tradeCreated()

```
public void tradeCreated(Trade trade)
{
    // Check the Trade Model used
    // and create an appropriate listener to handle it.
    if (trade.getType().equals("ORD"))
    {
        trade.setTradeListener(new ORDTradeListener(trade));
    }
    else if (trade.getType().equals("RFQ"))
    {
        trade.setTradeListener(new RFQTradeListener(trade));
    }
}
```

How the new Trade is handled depends on the trading system to which the Trading DataSource is connected, and the nature of the API to that system.

The Trade object must be retained, so that it can be referred to when the trading system responds with an event (see [Dealing with events](#) ^[18]). Assume, for example, the trading system API supports a listener style interface, with a listener object for each Trade. TradeListener.tradeCreated() can store the Trade object in a trading system listener object before calling the trading system. When the trading system subsequently raises an event on the Trade, it will call the listener, which can then refer to the Trade object as required (for example to create an event to pass on to the client).

Alternatively the trading system may not support a listener interface. For example, it may, just pass back an “event” with an ID relating to the Trade. In this case TradeListener.tradeCreated() would have to store the Trade object in a suitable data structure (say a hash table). This structure must be accessible by the code that handles events from the trading system. This code would typically use the ID returned in the trading system event as the key to extract the Trade object.

6.4 Dealing with Events

The TradeEvent object represents a Trade Event, which typically encapsulates a message between the client and the Trading DataSource. A TradeEvent has a type, which represents the type of the message, for example “Open”, “PriceUpdate” or “Execute”. It also has a number of fields to represent all the necessary information for that message, for example “BidPrice” or “Amount”.

The TradeListener is responsible for handling TradeEvents; it is notified when new events are received from the client. When a message is received from the client, it is processed by the Trading DataSource to verify that the event is valid based on the Trade Model before notifying the TradeListener of the event. The Trade and TradeEvent objects are passed to the TradeListener. The Trade has been updated with the data from the TradeEvent and can be used to create and send new TradeEvents. The TradeListener would then typically send a message on to the trading system or handle the event in some other way.

The following code extract shows an implementation of the TradeListener method to receive events.

Example implementation of TradeListener.receiveEvent()

```
public void receiveEvent(TradeEvent event)
{
    // Talk to trading system
}
```

Events raised by the trading system can be pushed into the Trading DataSource. This is done by creating a `TradeEvent` from the relevant `Trade` object, setting the necessary attributes, and asking the `Trade` object to send it. At this point the Trading DataSource will verify, through the Trade Model, that the event is allowed and contains all the necessary information, before sending the message off to the client.

The following code extract shows the typical custom code that would be written in the Trading DataSource to create an event and send it to a client.

Custom code to create an event

```
TradeEvent myEvent = trade.createEvent("PriceUpdate");
myEvent.addField("BidPrice", bidPrice);

// Add more fields
// ...

// Then send the event on to the client.
trade.sendEvent(myEvent);
```

6.5 Closing Trades

When a Trade reaches a final state it is closed. This could happen when the end-user or the trading system cancels the Trade, when the Trade is successfully executed, or when it is rejected. The final states are defined by the Trade Model and are the states that have no transitions to another state. The `ChannelListener` is notified when a Trade has reached this state, which allows the application to clean up any resources associated with that Trade.

The following code extract shows the implementation of the `ChannelListener` method for notifying closed Trades.

Example implementation of `ChannelListener.tradeClosed()`

```
public void tradeClosed(Trade trade)
{
    // Clean up
}
```

6.6 Closing Channels

When an end-user logs off the system the Trade Channel for that end-user is closed. This could also happen if the client application is designed to close Trade Channels when they are not in use. The `TradingApplicationListener` will be notified when a channel is closed, which allows any resources associated with that channel to be cleaned up. Once the Trade has been closed it can no longer be used to create or send events.

The following code extract shows the implementation of the `TradingApplicationListener` method for notifying closed channels.

Example implementation of `TradingApplicationListener.channelClosed()`

```
public void channelClosed(TradeChannel channel)
{
    // Clean up
}
```

6.7 Handling Blotter Channels

To handle Blotter Channels in the trading DataSource:

- Implement code in the `TradingApplicationListener` interface to register and deregister a `BlotterTradeListener`.
- Implement the `BlotterTradeListener` interface to construct and send blotter messages.
This interface provides notification of Blotter Events through the life cycle of a Trade. Blotter events (class `BlotterEvent`) are created when the Trading DataSource has validated a state transition in the Trade Model and has sent a `TradeEvent` to the client.

In the code fragments shown in the next sections, the implementation of `BlotterTradeListener` is the example listener `AutoBlotterTradeListener` provided with the example Trading DataSource:

Registering the BlotterTradeListener

When implementing the `TradingApplicationListener` interface (see [Initialization](#) ¹⁶), add code to register and deregister a `BlotterTradeListener`.

Registering and deregistering BlotterTradeListener in TradingApplicationListener

```
public class MyTradingApp implements TradingApplicationListener
{
    ...
    private BlotterTradeListener autoBlotterTradeListener
        = AutoBlotterTradeListener.getInstance();
    ...

    // Called when the TradingDataSource has created a BlotterChannel
    // as a result of receiving a request for a blotter subject.

    public void blotterChannelCreated(BlotterChannel blotterChannel)
    {
        tradingDataSource.addBlotterTradeListener(
            blotterChannel,
            autoBlotterTradeListener
        );
    }

    // Called when a BlotterChannel is closed.
    // The channel is closed when the originally requested blotter
    // subject is discarded, or the connection to the Trading DataSource's
    // peer is lost, or the Trading DataSource is being shut down.

    public void blotterChannelClosed(BlotterChannel blotterChannel)
    {
        tradingDataSource.removeBlotterTradeListener(
            blotterChannel,
            autoBlotterTradeListener
        );
    }
}
```

Here is more detailed explanation of the previous code fragment:

- ◆ The `BlotterTradeListener` is the example listener `AutoBlotterTradeListener` provided with the example Trading DataSource. The static method `AutoBlotterTradeListener.getInstance()` returns an instance of the listener that can be used with this Blotter Channel:

```
private BlotterTradeListener autoBlotterTradeListener
    = AutoBlotterTradeListener.getInstance();
```

- Code the `blotterChannelCreated()` method to register the `BlotterTradeListener` with the Trading DataSource when the Blotter Channel is created, by calling `addBlotterTradeListener()`:

```
public void blotterChannelCreated(BlotterChannel blotterChannel)
{
    tradingDataSource.addBlotterTradeListener(
        blotterChannel,
        autoBlotterTradeListener
    );
}
```

- Code the `blotterChannelClosed()` method to deregister the `BlotterTradeListener` from the Trading DataSource when the Blotter Channel is closed:

```
public void blotterChannelClosed(BlotterChannel blotterChannel)
{
    tradingDataSource.removeBlotterTradeListener(
        blotterChannel,
        autoBlotterTradeListener
    );
}
```


Implementing the BlotterTradeListener interface

The `BlotterTradeListener` has one method `receiveBlotterEvent()` that is called when the Trading DataSource has sent a Trade Event to the client. Note that the `BlotterEvent` is not an event in the execution of a Trade Model; it is merely a message containing both the `TradeEvent` for which a blotter entry is to be constructed and the `Blotter Channel` to send the entry on.

Here is a simple example of the `receiveBlotterEvent()` method in the `AutoBlotterTradeListener` implementation of `BlotterTradeListener`.

AutoBlotterTradeListener (example of BlotterTradeListener)

```
public class AutoBlotterTradeListener implements BlotterTradeListener
{
    // Called after a Trade Event has been validated and sent to the client.

    public void receiveBlotterEvent(BlotterEvent blotterEvent)
    {
        BlotterChannel blotterChannel = blotterEvent.getBlotterChannel();
        TradeEvent tradeEvent = blotterEvent.getTradeEvent();
        Trade trade = tradeEvent.getTrade();

        BlotterMessage blotterMessage = blotterChannel.createBlotterMessage();

        blotterMessage.addField("L1_AMOUNT", trade.getField("L1_AMOUNT"));
        blotterMessage.addField("ACCOUNT", trade.getField("ACCOUNT"));
        blotterMessage.addField("TRADING_TYPE", "demo");
        blotterMessage.addField("USER_NAME", trade.getChannel().getUser());
        ...

        blotterChannel.sendBlotterMessage(blotterMessage);
    }
}
```

Here is more detailed explanation of the previous code fragment:

- Obtain the `BlotterChannel` from the `BlotterEvent` that the Trading DataSource passed to the listener. From the `BlotterEvent` obtain the `TradeEvent` and the `Trade` to which the Trade Event relates.

```
BlotterChannel blotterChannel = blotterEvent.getBlotterChannel();
TradeEvent tradeEvent = blotterEvent.getTradeEvent();
Trade trade = tradeEvent.getTrade();
```

- Create a new blotter message on the `Blotter Channel`.

```
BlotterMessage blotterMessage = blotterChannel.createBlotterMessage();
```

- Populate the blotter message with the required fields and their values. Typically the field values are obtained from the Trade and from the user information associated with the Trade Channel.

```
blotterMessage.addField("L1_AMOUNT", trade.getField("L1_AMOUNT"));
blotterMessage.addField("ACCOUNT", trade.getField("ACCOUNT"));
blotterMessage.addField("TRADING_TYPE", "demo");
blotterMessage.addField("USER_NAME", trade.getChannel().getUser());
...
```

- Send the newly constructed blotter message to the client via the Blotter Channel.

```
blotterChannel.sendBlotterMessage(blotterMessage);
```

6.8 Handling Trade Restorations

When an end-user logs in to Caplin Trader, they may be interested in receiving trades that they had opened in the previous session. The Trading DataSource facilitates this by allowing you to restore trade objects and their associated events, and send them to the client.

The Trading DataSource's `TradingApplicationListener` has a `channelCreated()` method (see [New Trade Channels](#)⁽¹⁷⁾). In the implementation of this method, add the code that restores trade channels, as follows.

For each Trade to be restored, the code must invoke `restoreTrade()` on the Trade Channel, create a Trade Event for the Trade (`createRestoreEvent()`), and send the event to the client. At run time, when the client constructs a trade channel, `channelCreated()` is invoked in the Trading DataSource, which results in Trade Events being sent to the client for all the Trades to be restored.

The following example shows in more detail how to do this. It assumes that the trading system with which the Trading DataSource communicates (`tradingSystem` in the example code) is able to supply the information about trades that can be restored. For simplicity, the example is restricted to FX trading; in a real implementation you may need to handle trading in more than one asset class.

Example: Restoring Trades

```
public class MyTradingApp implements TradingApplicationListener
{
    public void channelCreated(TradeChannel channel)
    {
        // Get from the trading system a list of FX Trades
        // that are restorable for this end-user.
        List<TradingFXSystemTrade> trades =
            tradingSystem.getFXTrades(channel.getUser());
        for (TradingSystemTrade tradingSystemTrade: trades)
        {
            // Get the trade Model for the Trade.
            String tradeModel = tradingSystemTrade.getTradeProtocol();

            // We need to pass to the client an id that uniquely identifies
            // this restored Trade. We get this from the trading system.
            String restoredTradeId = tradingSystemTrade.getTradeId();

            // Now create the restored Trade on the Trade Channel.
            String assetClass = "FX";
            Trade restoredTrade = channel.restoreTrade(assetClass, tradeModel,
                restoredTradeId);
            restoredTrade.setTradeListener(new FxTradeListener());

            // Get the state of the trade being restored.
            String tradeState = tradingSystemTrade.getState();

            // Create a TradeEvent for the restored Trade,
            // with the correct trading state.
            TradeEvent restoreEvent =
                restoredTrade.createRestoreEvent(tradeState);

            // Copy the data relating to the trade into the TradeEvent.
            for (String fieldName: tradingSystemTrade.getFields().keySet())
            {
                String fieldValue = tradingSystemTrade.getFieldValue(fieldName);
                restoreEvent.addField(fieldName, fieldValue);
            }
            // Send the restored TradeEvent on to the client.
            restoredTrade.sendEvent(restoreEvent);
        }
    }
}
```

Restoring Trades that still exist on the client

When a Trade is restored it may already exist on the client. For example, the end-user may have opened a Trade Ticket, and then the client may fail over to another Liberator. When the failover is complete, the Trading DataSource restores the Trades in progress and sends them to the client as Trade Events. In this situation, when the client subsequently receives the Trade Event for a restored trade and still has the Ticket open for that trade, it must be able to associate the Trade Event with the relevant open Ticket.

To facilitate this, when the Trade is first created, the Trading DataSource must send the client an event containing a Trade Restoration id. The client retains the Trade Restoration id against the Trade. If the Trading DataSource subsequently restores the Trade, the client receives a Restored Trade event containing the matching Trade Restoration id.

The Trade Restoration ids would typically be created and managed by the trading system (as shown in the code example in [Handling Trade Restorations](#)^[24]).

The following example shows typical Trading DataSource code for creating the Trade Restoration id.

Creating a Trade Restoration id

```
// A Trade has been newly opened at the client,  
// so create the corresponding Trade Event.  
TradeEvent event = trade.createEvent("OpenAck");  
  
//Create the Trade in the trading system.  
TradingSystemTrade tradingSystemTrade = tradingSystem.createTrade();  
  
// Send a Trade Restoration id to the client  
// for possible use later if the trade is restored.  
event.setRestorationId(tradingSystemTrade.getTradeId());  
...  
trade.sendEvent(event);
```

7 The Java Trading DataSource Example

The Java Trading DataSource is supplied with example code to help you get started. This is in the Java package `example`, which is located in the folder `examples/source`. The `example` package contains commented example code that shows how to use the Trading DataSource API, as described in [Using the Trading DataSource Java API](#)¹⁶. The package is set up to handle Request For Stream (RFS) and Executable Streaming Price (ESP) Trade Models, but could be easily adapted to handle any Trade Models you configure.

The example is also used as the Reference Implementation Trading DataSource for ESP and RFS Trades made through the Caplin Trader Reference Implementation, and shares the Trade Model definitions with the client.

8 Using the Trading DataSource C++ API

This section provides a brief description of the main parts of the Trading DataSource C++ API (see the **Caplin Xaqua: Trading DataSource C++ API Reference** documentation for full details).

To implement a Trading DataSource you make use of callbacks on a class of your choice, to be notified of events occurring on Trades and to create events to be sent back into the system.

8.1 Initialization

When your Trading DataSource application starts, it should create an instance of `TradingDataSource` and register itself as a `TradingApplicationListener`.

The `TradingDataSource` starts up a `DataSource`, which then connects to the Liberator. The `DataSource` also sets up the necessary `DataSource` listeners internally to handle the trade messaging.

The following is a simple code extract showing the creation of a `TradingDataSource` within a custom application.

Creation of a TradingDataSource

```
using namespace Caplin::TradingDataSource;

class MyTradingApp : public TradingApplicationListener
{
public:
    MyTradingApp(const TradingDataSource::TradingDataSource& tradingDataSource,
                 const Logger& log)
        : m_tradingDataSource(tradingDataSource) m_log(log)
    {
        // ...
    private:
        TradingDataSource m_tradingDataSource;
        Logger m_log; //For logging errors and events.
    }

int main(int argc, char *argv[])
{
    // Prepare a vector of filenames of the trade model configuration files to use.
    std::vector<std::string> tradeModelConfigFiles;
    tradeModelConfigFiles.push_back("conf/MyESPStateModel.xml");
    tradeModelConfigFiles.push_back("conf/MyRFSStateModel.xml");

    // Create and initialize the Trading DataSource.
    // By default, it uses the logger that is provided with
    // the DataSource for C library underlying the Trading DataSource API.
    TradingDataSource::TradingDataSource myTradingDataSource(
        // Config file for this DataSource
        "MyDataSourceConfig.conf",
        tradeModelConfigFiles);

    // Create the Trading DataSource application, passing it the Trading DataSource
    // object and the logger (so we can log application errors to the same place as
    // errors raised in the Trading DataSource API).
    MyTradingApp myTradingApp(myTradingDataSource, myTradingDataSource.getLogger());

    // Connect to DataSource peers and start receiving callbacks.
    myTradingDataSource.start(&myTradingApp);
    // ...
}
```

The `TradingDataSource` processes the flow of a Trade by following the states defined by the Trade Model for the particular type of Trade. This processing is carried out by a state machine, implemented internally as a `StateMachine` object. Before creating the `TradingDataSource`, the Trading DataSource application creates a `StateMachineFactory` internally and loads the factory with the required Trade Models. The models are defined in the XML configuration files discussed in [Configuring Trade Models](#)^[14] and their paths are passed in to the constructor of `TradingDataSource` either as a vector of strings or a single string.

The `TradingDataSource` constructor is also supplied with a configuration file for the DataSource. This file defines the connections that the DataSource makes with the other Caplin Xaqua components, such as Liberator.

Logging

By default the C++ Trading DataSource uses the logging facilities that are provided with the DataSource for C library underlying the Trading DataSource API. This means that log messages are written to standard DataSource log files, along with any other messages and events that are logged by that library.

In place of the default logging, you can use your own logger class (or wrap a third party one); this must be an implementation of `LoggerImpl`. You can pass the implementation of `LoggerImpl` into the `TradingDataSource` constructor.

The `getLogger()` method of the `TradingDataSource` allows you to access the logger (the default one, or your custom implementation), so that your application can log errors to the same place as errors raised in the Trading DataSource API.

Note: If you implement a custom logger, only the Trading DataSource code will use it. Any errors or events logged by the code in the underlying DataSource for C library go through this library's logger.

For more information about the default logger, see the [DataSource for C API Reference](#).

Example of a custom Trading DataSource logger that uses the standard C++ logger:

```
class Log4cxxLogger : public Caplin::LoggerImpl
{
public:
    Log4cxxLogger() : m_log(log4cxx::Logger::getRootLogger())
    {
        log4cxx::xml::DOMConfigurator::configure("log4cxx_config.xml");
    }

    void debug(const std::string& msg) { m_log->debug(msg); }
    void error(const std::string& msg) { m_log->error(msg); }
    void critical(const std::string& msg) { m_log->fatal(msg); }
    void info(const std::string& msg) { m_log->info(msg); }
    void warn(const std::string& msg) { m_log->warn(msg); }

private:
    log4cxx::LoggerPtr m_log;
};
```

8.2 New Trades

The `ChannelListener` is notified of new Trades when a client initiates them; the Trade is passed to it as a `Trade` object. When a Trade is created the `ChannelListener` can perform any initialization needed with the trading system and also add a `TradeListener` to the newly created `Trade` object. The `TradeListener` is an interface you must implement and could be a new instance for each Trade or a single global instance; its job is to handle all events for a Trade. You normally have different implementations of `TradeListener` that handle different trade types.

The following code extract shows part of a sample implementation of a `ChannelListener`. It shows a different custom `TradeListener` being added to the `Trade` object to handle its events, depending on the Trade Model used for the Trade.

Example implementation of `ChannelListener::tradeCreated()`

```
void MyTradeChannelListener::tradeCreated(Trade& trade)
{
    if (trade.isType("ESP"))
    {
        trade.setTradeListener(m_pESPTradeListener);
    }
    else if (trade.isType("RFS"))
    {
        trade.setTradeListener(m_pRFSTradeListener);
    }
}
```

How the new Trade is handled depends on the trading system to which the Trading DataSource is connected, and the nature of the API to that system.

The `Trade` object must be retained, so that it can be referred to when the trading system responds with an event (see [Dealing with events](#)^[30]). Assume, for example, the trading system API supports a listener style interface, with a listener object for each Trade. `TradeListener.tradeCreated()` can store the `Trade` object in a trading system listener object before calling the trading system. When the trading system subsequently raises an event on the Trade, it will call the listener, which can then refer to the `Trade` object as required (for example to create an event to pass on to the client).

Alternatively the trading system may not support a listener interface. For example, it may, just pass back an "event" with an ID relating to the Trade. In this case `TradeListener.tradeCreated()` would have to store the `Trade` object in a suitable data structure (say a hash table). This structure must be accessible by the code that handles events from the trading system. This code would typically use the ID returned in the trading system event as the key to extract the `Trade` object.

8.3 New Trade Channels

The `TradingApplicationListener` is notified when new Trade Channels are created; this allows you to perform any necessary user specific initialization. You must also add a `ChannelListener` to the channel. The `ChannelListener` is an interface you must implement; it could be a new instance for each channel or a single global instance. Its job is to handle notifications on the channel about newly created Trades and Trades being closed.

The following code extract shows part of a sample implementation of a `TradingApplicationListener`. It shows a custom `ChannelListener` being added to the channel to handle Trades, and then a method being called on a hypothetical `tradingSystem` object to log in the end-user for the channel.

Example implementation of TradingApplicationListener::channelCreated()

```
void MyTradingApplicationListener::channelCreated(TradeChannel& channel)
{
    channel.setChannelListener(m_pTradeChannelListener);

    // Handle new channel/user.
    // For example:
    tradingSystem.loginUser(channel.getUser());
}
```

8.4 Dealing with Events

The `TradeEvent` object represents a Trade Event, which typically encapsulates a message between the client and the Trading DataSource. A `TradeEvent` has a type, which represents the type of the message, for example "Open", "PriceUpdate" or "Execute". It also has a number of fields to represent all the necessary information for that message, for example "BidPrice" or "Amount".

The `TradeListener` is responsible for handling `TradeEvents`; it is notified when new events are received from the client. When a message is received from the client it is processed by the Trading DataSource to verify that the event is valid based on the Trade Model before notifying the `TradeListener` of the event. The `Trade` and `TradeEvent` objects are passed to the `TradeListener`. The `Trade` has been updated with the data from the `TradeEvent` and can be used to create and send new `TradeEvents`. The `TradeListener` would then typically send a message on to the trading system or handle the event in some other way.

The following code extract shows an implementation of the `TradeListener` method to receive events.

Example implementation of TradeListener::receiveEvent()

```
void RFQTradeListener::receiveEvent(Trade& trade,
                                    const TradeEvent& tradeEvent)
{
    // Talk to trading system
}
```

Events raised by the trading system can be pushed into the Trading DataSource. This is done by creating a `TradeEvent` from the relevant `Trade` object, setting the necessary attributes, and asking the `Trade` object to send it. At this point the Trading DataSource will verify, through the Trade Model, that the event is allowed and contains all the necessary information, before sending the message off to the client.

The following code extract shows the typical custom code that would be written in the Trading DataSource to create an event and send it to a client.

Custom code to create an event

```
TradeEvent myEvent = trade.createEvent("PriceUpdate ");
myEvent.addField("BidPrice", bidPrice);

// Add more fields
// ...

// Then send the event on to the client.
trade.sendEvent(myEvent);
```

8.5 Closing Trades

When a Trade reaches a final state, it is closed. This could happen when the end-user or the trading system cancels the Trade, when the Trade is successfully executed, or when it is rejected. The final states are defined by the Trade Model and are the states that have no transitions to another state. The `ChannelListener` is notified when a Trade has reached this state, which allows the application to clean up any resources associated with that Trade. Once the Trade has been closed it can no longer be used to create or send events.

The following code extract shows the implementation of the `ChannelListener` method for notifying closed Trades.

Example implementation of `ChannelListener.tradeClosed()`

```
void MyTradeChannelListener::tradeClosed(Trade& trade)
{
    // Clean up
    // ...
}
```

8.6 Closing Channels

When an end-user logs off the system the Trade Channel for that end-user is closed. This could also happen if the client application is designed to close Trade Channels when they are not in use. The `TradingApplicationListener` will be notified when a channel is closed, which allows any resources associated with that channel to be cleaned up.

The following code extract shows the implementation of the `TradingApplicationListener` method for notifying closed channels.

Example implementation of `TradingApplicationListener.channelClosed()`

```
void MyTradingApplicationListener::channelClosed(TradeChannel& channel)
{
    // Clean up
    // ...
}
```

8.7 Handling Blotter Channels

To handle Blotter Channels in the trading DataSource:

- Implement code in the `TradingApplicationListener` class to register and deregister a `BlotterListener`.
- Implement the `BlotterListener` class to construct and send blotter messages.

This interface provides notification of Blotter Events through the life cycle of a Trade. Blotter events are created when the Trading DataSource has validated a state transition in the Trade Model and has sent a `TradeEvent` to the client.

Registering the BlotterListener

When implementing the `TradingApplicationListener` interface (see [Initialization](#) ^[27]), add code to register and deregister a `BlotterListener`.

- Code the `blotterChannelCreated()` method to register the `BlotterListener` with the Trading DataSource when the Blotter Channel is created, by calling `setBlotterListener()`.

Registering the BlotterListener in TradingApplicationListener

```
void DemoTradingSource::blotterChannelCreated(BlotterChannel& blotterChannel)
{
    m_tradingDataSource->setBlotterListener(blotterChannel,
                                           this //The BlotterListener
                                           );
}
```

In this example the `DemoTradingSource` implements both `TradingApplicationListener` and `BlotterListener`, so the `BlotterListener` argument of `setBlotterListener` is passed as `this`.

- Code the `blotterChannelClosed()` method to deregister the `BlotterListener` from the Trading DataSource when the Blotter Channel is closed:

Deregistering the BlotterListener in TradingApplicationListener

```
void DemoTradingSource::blotterChannelClosed(BlotterChannel& blotterChannel)
{
    m_tradingDataSource->removeBlotterListener(blotterChannel);
}
```

Implementing the BlotterListener interface

The `BlotterListener` has one method `receiveBlotterEvent()` that is called when the Trading DataSource has validated a state transition in the Trade Model and has sent a `TradeEvent` to the client.

Here is a simple example of the `receiveBlotterEvent()` method. In this example the `DemoTradingSource` implements `BlotterListener`, and so contains the code of `receiveBlotterEvent()`.

Example of `BlotterListener::receiveBlotterEvent()`

```
void DemoTradingSource::receiveBlotterEvent (
    TradingDataSource::BlotterChannel& blotterChannel,
    const TradingDataSource::Trade& trade,
    const TradingDataSource::TradeEvent& tradeEvent
)
{
    BlotterMessage blotterMessage = blotterChannel.createMessage();

    blotterMessage.addFields(trade.getFields());

    std::string status = tradeEvent.getField("Status");
    blotterMessage.addField("Status", status);
    blotterMessage.addField("TradeDate", "20080808");
    blotterMessage.addField("TimeStamp", "123456");
    blotterMessage.addField("UserName", trade.getUser());
    ...

    blotterChannel.sendMessage(blotterMessage);
}
```

Here is more detailed explanation of the previous code fragment:

- Create a new blotter message on the Blotter Channel.

```
BlotterMessage blotterMessage = blotterChannel.createMessage();
```

- Populate the blotter message with the required fields and their values. Typically the field values are obtained from the Trade.

```
blotterMessage.addFields(trade.getFields());

std::string status = tradeEvent.getField("Status");
blotterMessage.addField("Status", status);
blotterMessage.addField("TradeDate", "20080808");
blotterMessage.addField("TimeStamp", "123456");
blotterMessage.addField("UserName", trade.getUser());
...
```

- Send the newly constructed blotter message to the client via the Blotter Channel.

```
blotterChannel.sendMessage(blotterMessage);
```

8.8 Handling Trade Restorations

When an end-user logs in to Caplin Trader, they may be interested in receiving trades that they had opened in the previous session. The Trading DataSource facilitates this by allowing you to restore trade objects and their associated events, and send them to the client.

The Trading DataSource's `TradingApplicationListener` has a `channelCreated()` method (see [New Trade Channels](#) ^[29]). In the implementation of this method, add the code that restores trade channels, as follows.

For each Trade to be restored, the code must invoke `restoreTrade()` on the Trade Channel, create a Trade Event for the Trade (`createRestoreEvent()`), and send the event to the client. At run time, when the client constructs a trade channel, `channelCreated()` is invoked in the Trading DataSource, which results in Trade Events being sent to the client for all the Trades to be restored.

The following example shows in more detail how to do this. It assumes that the trading system with which the Trading DataSource communicates (`tradingSystem` in the example code) is able to supply the information about trades that can be restored. For simplicity, the example is restricted to FX trading; in a real implementation you may need to handle trading in more than one asset class.

Example: Restoring Trades

```
void DemoTradingSource::channelCreated(TradeChannel& channel)
{
    // Get from the trading system the FX Trades
    // that are restorable for this end-user.
    vector<TradingFXSystemTrade> trades =
        m_tradingSystem.getFXTrades(channel.getUser());

    for (vector<TradingFXSystemTrade>::iterator
        tradingSystemTrade = trades.begin();
        tradingSystemTrade != trades.end(); ++tradingSystemTrade)
    {
        // Get the trade Model for the Trade.
        string tradeModel = tradingSystemTrade->getTradeProtocol();

        // We need to pass to the client an id that uniquely identifies
        // this restored Trade. We get this from the trading system.
        string restoredTradeId = tradingSystemTrade->getTradeId();

        // Now create the restored Trade on the Trade Channel.
        string assetClass = "FX";
        Trade restoredTrade = channel.restoreTrade(assetClass, tradeModel,
                                                    restoredTradeId);
        restoredTrade.setTradeListener(&m_tradeListener);

        // Get the state of the trade being restored.
        string tradeState = tradingSystemTrade->getState();

        // Create a TradeEvent for the restored Trade,
        // with the correct trading state.
        TradeEvent restoreEvent = restoredTrade.createRestoreEvent(tradeState);

        // Copy the data relating to the trade into the TradeEvent.
        restoreEvent.addFields(tradingSystemTrade->getFields());

        // Send the restored TradeEvent on to the client.
        restoredTrade.sendEvent(restoreEvent);
    }
}
```

Restoring Trades that still exist on the client

When a Trade is restored it may already exist on the client. For example, the end-user may have opened a Trade Ticket, and then the client may fail over to another Liberator. When the failover is complete, the Trading DataSource restores the Trades in progress and sends them to the client as Trade Events. In this situation, when the client subsequently receives the Trade Event for a restored trade and still has the Ticket open for that trade, it must be able to associate the Trade Event with the relevant open Ticket.

To facilitate this, when the Trade is first created, the Trading DataSource must send the client an event containing a Trade Restoration id. The client retains the Trade Restoration id against the Trade. If the Trading DataSource subsequently restores the Trade, the client receives a Restored Trade event containing the matching Trade Restoration id.

The Trade Restoration ids would typically be created and managed by the trading system (as shown in the code example in [Handling Trade Restorations](#)^[34]).

The following example shows typical Trading DataSource code for creating the Trade Restoration id.

Creating a Trade Restoration id

```
// A Trade has been newly opened at the client,  
// so create the corresponding Trade Event.  
TradeEvent event = trade.createEvent("OpenAck");  
  
//Create the Trade in the trading system.  
TradingSystemTrade tradingSystemTrade = m_tradingSystem.createTrade();  
  
// Send a Trade Restoration id to the client  
// for possible use later if the trade is restored.  
event.setRestorationId(tradingSystemTrade.getTradeId());  
...  
trade.sendEvent(event);
```

9 The C++ Trading DataSource Example

The C++ Trading DataSource is supplied with an example application to help you get started.

- ◆ The windows version has a Visual Studio 2005 project file for the example in the folder *examples\DemoTradingSource*.
- ◆ The Linux version has a make file for the example in the folder *examples\DemoTradingSource*.

The source files *DemoTradingSource.cpp* and *DemoTradingSource.h* contain the example code that shows how to use the Trading DataSource API as described in this document. The example is set up to handle Request For Stream (RFS) and Executable Streaming Price (ESP) Trade Models, but could be easily adapted to handle any Trade Models you configure.

10 Configuring Caplin Liberator for trading

Caplin Liberator is an integral part of Caplin Xaqua (see the diagram in the [Overview](#)^[5]) and therefore must be correctly configured to support trading activity.

The following aspects of trading activity are determined through Liberator configuration:

- ◆ Associating a unique end-user with a trade message and ensuring that one end-user cannot trade on behalf of another; see [Mapping trade messaging objects in Liberator](#)^[38].
- ◆ [Routing of trade messages](#)^[40] to the correct Trading DataSource.
- ◆ [Trading performance and integrity](#)^[41].
- ◆ [Security](#)^[43].

The following table lists the Liberator configuration items relevant to trading:

Configuration item	Parameter	See section
add-object	<i>name</i>	Mapping trade messaging objects in Liberator ^[38]
add-object	<i>throttle-times</i>	Throttling ^[41]
add-object	<i>discard-timeout</i>	Throttling ^[41]
add-peer	—	Routing trade messages to the Trading DataSource ^[40]
add-data-service	—	Routing trade messages to the Trading DataSource ^[40]
object-map	—	Mapping trade messaging objects in Liberator ^[38]
output-queue-size	—	Optimizing client reconnection time ^[41]
session-id-len	—	Session IDs ^[43]

Note: The configuration items discussed here specifically relate to using Liberator in a trading environment, and in particular to support Caplin Trader applications. There are many other aspects of Liberator functionality and performance that also need to be configured in an implementation of Caplin Xaqua. For more information about configuring Liberator, see the **Liberator Administration Guide**.

Tip: The examples of Liberator configuration shown in the following sections are derived from the configuration file supplied with the Caplin Trader Reference Implementation. This file is *rttpe.conf*, located in `$CT_INSTALL_DIR/apps/caplin/Liberator/etc/`, where `$CT_INSTALL_DIR` is the directory in which Caplin Xaqua has been installed.

10.1 Mapping trade messaging objects in Liberator

Trade messages are passed between client and Liberator as updates to subscriptions. The subject of the update identifies it as a trade message. The Liberator configuration must define a base subject name for these subscriptions, as a “built-in” Liberator object that is created when Liberator starts up. For example, in the Caplin Trader Reference Implementation, the Liberator configuration file defines an object called `/FT/TRADE`, so that all trade messages sent between client and Liberator have a subject name that starts with `/FT/TRADE`.

```
add-object
    name      /FT/TRADE
    ...
end-object
```

Tip: The **add-object** configuration item can also take additional optional parameters, as shown by the ... in the previous example. For more information, see the Configuration Reference section of the **Liberator Administration Guide**, and the section on [Throttling](#)⁴¹.

A client opens a Trade Channel by subscribing to one of the built-in Liberator trade message objects, for example, `/FT/TRADE`. Because Liberator and the Trading DataSource need to manage many end-users who are simultaneously trading, the Liberator maps the generic trade message objects onto user specific object names. This defines the unique channel over which each end-user trades. The mapping is defined using the **object-map** configuration item.

Example object mapping

```
object-map /FT/TRADE/%1 /FT/TRADE/%1/%U
```

`%U` is the unique end-user name (Liberator login name) associated with the Liberator session.

`%1` represents any variable length string appearing in the subject name of the subscription.

When an end-user called “UserA” connects to Liberator and trades, the trade messages sent between the client and Liberator have a subject name of the form:

`/FT/TRADE/FX/<identification-of-this-trade>`

Before passing an incoming trade message to the Trading DataSource, Liberator maps the subject name in the message according to the **object-map** configuration, so that the subject of the passed on message is:

`/FT/TRADE/FX/< identification-of-this-trade>/UserA-1`

where **UserA-1** is the unique end-user name assigned to this Liberator login of “UserA”.

This transformation allows the Trading DataSource to distinguish between trade messages from “UserA” and trade messages relating to other end-users. The Trade Channel for “UserA” is uniquely defined by the combination of the strings `/FT/TRADE/FX/` and `UserA-1` in the message subject.

Similarly, when Liberator receives a trade message with subject
“`/FT/TRADE/FX/< identification-of-this-trade>/UserA-1`”

from the Trading DataSource, it can readily determine the client connection over which it should forward the message to the client, and the forwarded message is given the subject
“`/FT/TRADE/FX/< identification-of-this-trade>/`”.

Preventing identity theft

The %1 parameter in the **object-map** helps to prevent an end-user from trading using another end-user's identity. For example, if "UserB", should attempt to fake a trade message so that it appears to come from "UserA", Liberator will reject the message. The faked trade message from the "UserB" client would have the subject

```
/FT/TRADE/FX/<identification-of-this-trade>/UserA
```

However the object mapping in the Liberator transforms this subject name to

```
/FT/TRADE/FX/<identification-of-this-trade>/UserA/UserB-1
```

which Liberator recognizes as a subject for which no subscription exists, so the message is rejected.

10.2 Routing trade messages to the Trading DataSource

The Liberator configuration defines the DataSource peers to which Liberator is connected and the data services that Liberator provides to subscribing clients. In Caplin Xaqua this configuration must include the Trading DataSources and their associated data services.

Tip: For more information about data services, see the **Caplin DataSource Overview**.

For example, in the Caplin Trader Reference Implementation, the Liberator configuration defines connections to two Trading DataSources, one for FX trading and one for FI trading:

Example of Trading DataSource configuration in Liberator

```
# fxtradesource
add-peer
  remote-id      17
  remote-type    active
  remote-name    fxtradesrc
  label          fxtradesrc
end-peer

# fitradesource
add-peer
  remote-id      18
  remote-type    active
  remote-name    fitradesrc
  label          fitradesrc
end-peer

...

add-data-service
  service-name    fx-trade-data
  include-pattern ^/FT/TRADE/FX

  add-source-group
    required
    add-priority
    label          fxtradesrc
    end-priority
  end-source-group
end-data-service

add-data-service
  service-name    fi-trade-data
  include-pattern ^/FT/TRADE/FI

  add-source-group
    required
    add-priority
    label          fitradesrc
    end-priority
  end-source-group
end-data-service
```

The **add-peer** configuration items define connections to two Trading DataSources, one for FX Trading (`fxtradesrc`) and one for FI Trading (`fitradesrc`).

There is a data service for each of these Trading DataSources, defined by the **add-data-service** configuration item. The FX service (`service-name fx-trade-data`) has an *include-pattern* parameter that ensures all trade message subscriptions whose subject begins with `/FT/TRADE/FX` are directed to the FX Trading DataSource. Similarly the FI Service definition (`service-name fi-trade-data`) directs to the FI Trading DataSource any trade message subscription whose subject begins with `/FT/TRADE/FI`.

Note that both the *include-patterns* start with the base subject name for trade messages as defined in an **add-object** configuration item; see [Mapping trade messaging objects in Liberator](#)^[38].

10.3 Trading performance and integrity

Liberator configuration items are used to:

- ◆ Optimize client reconnection time.
- ◆ Reduce the performance impact of high update rates through throttling.

Optimizing client reconnection time

If a client session becomes disconnected, Liberator will store update messages for the client until the client reconnects. This optimizes the reconnect time – on reconnection the stored updates are sent to the client as though the connection had not been lost.

The **output-queue-size** configuration item defines the maximum number of such messages that Liberator will store for each client. If the client reconnects after this limit has been reached, Liberator effectively discards the messages in the queue and instead sends the client the full image data for each subscribed object, which can take a significant time.

In a trading environment, the client is likely to be subscribed to a large number of instruments that are updating frequently, so the default **output-queue-size** value of 64 may be too small to optimize reconnection, even for transient connection losses. It is therefore suggested that **output-queue-size** be increased to **512** messages.

Throttling

Liberator's throttling feature is a mechanism for reducing the performance impact of high update rates – see the appendix [Overview of throttling](#)^[44]. For optimal performance it is highly desirable to throttle the instrument data updates that are streamed to clients, but:

Note: Messages on Trade Channels must **not** be throttled.

Trade messages must not be throttled because two trade messages sent within a single throttle period would be merged into one message. The resulting behavior would be undefined – for example, the Trade could fail or could execute incorrectly.

To meet these two requirements, throttling instrument data updates but not throttling trade messages, define separate Liberator objects for the instrument data subscriptions and for the messaging subscriptions. You can then set separate throttle levels for the messages relating to these two types of subscription.

Example of throttle settings in the Caplin Trader Reference Implementation:

```
add-object
  name      /FX
  type      20
  only-changed-fields
  throttle-times 0.25 2
end-object

add-object
  name      /FI
  type      20
  only-changed-fields
  throttle-times 0.25 2
end-object

...

add-object
  name      /FT/TRADE
  type      20
  throttle-times 0
  discard-timeout 0
end-object

...
```

This configuration shows that streamed data about FX and FI instruments is sent to clients as updates to the subscriptions in the Liberator directories /FX and /FI respectively. For example, updates to the FX currency pair EURUSD would be sent as messages with the subject name /FX/EURUSD.

Both FX and FI updates are subject to throttling (*throttle-times 0.25 2*), where the default throttle time is 0.25 seconds (the first entry in the list). The configuration parameter *only-changed-fields* ensures that just the changed fields in an update are forwarded to the client, thus optimizing the performance of the real time instrument display.

The **add-object** configuration item for trade messaging (*name /FT/TRADE*) has different parameter settings:

- ◆ *throttle-times 0* ensures that the objects used for trade messaging (*/FT/TRADE/...*) do not have throttling enabled.

Note: The throttle time *must* be explicitly set to zero here, so as to override any globally defined throttle time (configuration item **object-throttle-times**).

- ◆ The *only-changed-fields* parameter is false (by default), so all trade messages are sent in their entirety.

This is a performance optimization. The additional processing needed on Liberator to send the client all fields in each trade message is less than the processing needed to reconstruct messages in the client if the Liberator only sent the updated fields.

- ◆ *discard-timeout 0* ensures that Liberator removes a trade message subscription from its cache as soon as the Trade Channel is closed (the end-user has logged off Liberator and has therefore unsubscribed from /FT/TRADE/).

10.4 Security

The recommendations in the following sections explain how to maximize the security of Trades.

Also see [Preventing identity theft](#)^[39] in [Mapping trade messaging objects in Liberator](#)^[38].

Session IDs

When a client connects to Liberator, the Liberator generates a unique session ID. This identifies the session in subsequent (RTTP) message exchanges between the client and Liberator across this connection. The client includes the session ID in every message sent to Liberator. The session ID is generated using a cryptographically secure pseudo-random number generator, the major feature of such a generator being that is difficult for a third party to predict its output by observing its previous outputs.

The Liberator configuration item **session-id-len** defines the length in characters of the unique identifier for a session. Its default value is 12, for backwards compatibility with older versions of Liberator and client StreamLink libraries, where the length of the session identifier cannot be changed.

Note: It is recommended that in Caplin Xaqua installations **session-id-len** be increased to **22** characters. This makes it extremely unlikely that a third party could successfully guess a session ID so as to impersonate a legitimate end-user.

Single sign-on and KeyMaster

If you have a single sign-on system in place, Caplin recommends using the Caplin KeyMaster product in conjunction with this system, so that Caplin Trader end-users can access Liberator in a secure manner through the single sign-on. For more information, see the **Caplin KeyMaster Overview**.

11 Appendix: Overview of throttling

In a fast moving market, data updates can be generated more frequently than an end-user can actually notice. For example, an end-user does not actually need to see every update if an item updates ten times in a second. Additionally, system resources and performance can be adversely impacted by such high update rates being fed through to client applications.

Liberator's throttling feature is a mechanism for reducing the performance impact of high update rates. When throttling is enabled, Liberator accumulates all the updates for a data item during an interval called the throttle time. At the end of this interval Liberator sends just the latest updated values of the item to the subscribed clients. The rate at which updates are sent to clients is therefore reduced or "throttled".

The following table shows how throttling works.

Consider a data item containing the bid and ask prices for stock in company ABC. Clients are subscribed to the item /ABC consisting two fields "bid" and "ask" (the bid price and the ask price). The table shows the succession of field updates that Liberator receives from the price feed via a DataSource adapter, the values that it holds in its cache, and the throttled updates that are actually sent to clients subscribed to /ABC.

Time	Updates received by Liberator	/ABC in Liberator cache		Updates sent to clients
		bid=	ask=	
T0 Start of throttle period 1		51.160	52.032	
T1	bid=51.162, ask=52.037	51.162	52.037	
T2	bid=51.155	51.155	52.037	
T3	bid=51.158	51.158	52.037	
T4 End of throttle period 1		51.158	52.037	bid=51.158, ask=52.037
T4 Start of throttle period 2		51.158	52.037	
T5	ask=52.041	51.158	51.041	
T6	ask=52.040	51.158	51.040	
T7 End of throttle period 2		51.158	51.040	ask=51.040

During the first throttle period the bid price changes three times (at times T1, T2, and T3) and the ask price changes just once (at time T1). At the end of this throttle period (time T4) Liberator sends just the most recently updated bid and ask prices to the clients, so the clients do not see the bid price updates at times T1 and T2.


During the second throttle period the ask price changes twice (at times T5 and T6), but the bid price does not change at all, so at the end of this period (time T7) Liberator sends only the most recently updated ask price to the clients.

If an item is not updated at all during a throttle period, the very next time the item is updated in any subsequent throttle period Liberator immediately sends the updated values to its subscribed clients. This ensures that throttling does not introduce unnecessary delay in propagating updates to clients.

Throttling can be configured per data item; this allows all the items in a particular directory or even an individual item in a directory to be throttled by specific amounts.

Liberator can supply the same item to multiple end-users at different throttle levels. This allows end-users who need to view a large number of items, but who have low specification computers or slow network connections to the server, to receive data at a speed that suits their environment.

Client applications can change the level of throttling for specified items, groups of items, or all items globally.

Note: Messages on Trade Channels must **not** be throttled; see [Throttling](#) .

Tip: In some older Caplin documents throttling is sometimes called conflation.

12 Glossary of terms and acronyms

This section contains a glossary of terms, abbreviations, and acronyms used in this document.

Term	Definition
API	<u>A</u> pplication <u>P</u> rogramming <u>I</u> nterface
Caplin Trader	A web application framework for financial trading. An application constructed with Caplin Trader (a Caplin Trader application) is a Caplin Xaqua client application. Caplin Trader was formerly called "Caplin Trader Client".
Caplin Trader application	A Caplin Xaqua client that has been built using Caplin Trader .
Caplin Trader Reference Implementation	Caplin Trader ships with a Reference Implementation. This is a Caplin Trader application that provides a fully-functional, single-dealer trading client that can easily be modified, customized, and extended to create bespoke offerings.
Caplin Xaqua	A framework for building single-dealer platforms that enables banks to deliver multi-product trading direct to client desktops. Caplin Xaqua was formerly called "the Caplin Platform".
Caplin Xaqua client	A client desktop application that interfaces with Caplin Xaqua to deliver multi-product trading to end-users. The application can be implemented in any technology that is supported by Caplin Xaqua; for example Ajax, Microsoft .NET, Microsoft Silverlight™, Adobe Flex™, and Java™.
Blotter	A record of the details of Trades made by an end-user of Caplin Xaqua . In a Caplin Trader application the blotter is displayed in a dedicated panel and is dynamically updated as the end-user progresses through a Trade .
Blotter Channel	See Blotter Channels ^[8] .
Data service	An abstraction layer within a DataSource that allows the DataSource to request objects, based on their subject names, without needing to know the specific DataSources from which the objects originate. A data service is defined through configuration. For more information see the Caplin DataSource Overview .
DataSource	DataSources are software adapters within Caplin Xaqua that connect Xaqua to external sources of real time data and external trading systems . In other Caplin documents DataSources are also called DataSource adapters.
ESP	<u>E</u> xecutable <u>S</u> teaming <u>P</u> rice Trade Model .
Liberator	Caplin Liberator is a real-time financial internet hub that delivers trade messages and market data to and from subscribers over any network.
RFQ	<u>R</u> equest for <u>Q</u> uote Trade Model .
SDK	<u>S</u> oftware <u>D</u> evelopment <u>K</u> it
Throttling	A Liberator feature for reducing the performance impact of high update rates for data streamed to clients. See Appendix: Overview of throttling ^[44] .

Term	Definition
Trade	In this document the term “Trade” represents a single trade for an end-user. This could be an RFQ , an execution on a streaming price (ESP), or any other type of trade. See Trades ^[7] .
Trade Channel	A single end-user’s communication between the client application and the Trading DataSource . See Trade Channels ^[7] .
Trade Event	An action by a client or an event from the trading system . See Trade Events ^[8] .
Trading DataSource	The DataSource used to integrate Caplin Xaqua with a trading system .
Trading DataSource API	The API to the Trading DataSource that allows the DataSource to be integrated with a trading system .
Trade Model	A Trade Model represents the workflow when trading a financial instrument. Examples are Request for Quote (RFQ) and Executable Streaming Price (ESP). Trade Models consist of a number of states and transitions between those states. See Trade Models ^[7] .
Trading system	The term used in this document to refer to systems that support trade capture.

Index

- A -

- Abbreviations, definitions 46
- Acronyms, definitions 46
- add-data-service
 - Liberator configuration example 40
 - Liberator configuration item 37
- add-object
 - discard-timeout parameter 41
 - Liberator configuration example 38
 - Liberator configuration item 37
 - throttle-times parameter 41
- add-peer
 - Liberator configuration example 40
 - Liberator configuration item 37
- add-source-group
 - Liberator configuration example 40
- API
 - C++ Trading DataSource, using 27
 - Java Trading DataSource, using 16
 - Trading DataSource, overview 5
 - Trading DataSource, reference documentation 2
- API (C++)
 - ChannelListener 31
 - ChannelListener interface 29
 - restoring Trades 34
 - TradeEvent object 30
 - TradeListener interface 29
 - TradeListener object 30
 - TradingApplicationListener 27, 29
 - TradingDataSource 27
- API (Java)
 - ChannelListener 19
 - ChannelListener interface 17
 - restoring Trades 24
 - TradeEvent object 18
 - TradeListener interface 17
 - TradeListener object 18
 - TradingApplicationListener 16, 17
 - TradingDataSource 16
- Architecture
 - of Caplin Xaqua 2
- asset class

handled by Trading DataSource 7

- B -

- Blotter channel
 - definition of and example 8
 - handling in C++ 32
 - handling in Java 20
- BlotterTradeListener (C++)
 - registering with TradingApplicationListener 32
- BlotterTradeListener (Java)
 - registering with TradingApplicationListener 20
- BlotterTradeListener interface
 - implementing in C++ 33
 - implementing in Java 22

- C -

- C++
 - API for Trading DataSource, examples 27
- Cache
 - in Liberator 44
- Caplin Trading DataSource
 - API reference documentation 2
- Caplin Xaqua
 - overview and architecture 2
- Channel
 - blotter channel 8
 - trade channel 7
- ChannelListener (C++)
 - notification of new trades 29
- ChannelListener (Java)
 - notification of new trades 17
- ChannelListener (C++)
 - adding to a trade channel 29
 - notifying channel closure 31
 - notifying trade closure 31
- ChannelListener (Java)
 - adding to a trade channel 17
 - notifying channel closure 19
 - notifying trade closure 19
- Configuration
 - of Liberator for trading 37
 - of RFQ trade model using XML 15
 - of trade model 5, 14

- Configuration file for DataSource
 - in C++ TradingDataSource constructor 27
- Conflation 44
- Custom logger
 - in C++ Trading DataSource 28
- D -**
- Data service
 - in Liberator, for trading 40
- DataSource Configuration file
 - in C++ TradingDataSource constructor 27
- DataSource listener (C++)
 - handling trade messages 27
- DataSource listener (Java)
 - handling trade messages 16
- Default logger
 - in C++ Trading DataSource 28
- discard-timeout
 - parameter of add-object 41
- E -**
- ESP Trade Model
 - in example package of Java Trading DataSource kit 26
 - in Trading DataSource kit 14
 - state diagram 10
- Event
 - definition for trading 8
 - from trading system (C++) 29
 - from trading system (Java) 17
 - handling by listener object 16, 27
 - represented by TradeEvent object (C++) 30
 - represented by TradeEvent object (Java) 18
- Examples
 - closing a trade (C++) 31
 - closing a trade (Java) 19
 - closing a trade channel (C++) 31
 - closing a trade channel (Java) 19
 - creating a new trade channel (C++) 29
 - creating a new trade channel (Java) 17
 - creation of trade event (C++) 30
 - creation of trade event (Java) 18
 - implementation of ChannelListener (C++) 29
 - implementation of ChannelListener (Java) 17
 - implementation of TradeListener (C++) 30
 - implementation of TradeListener (Java) 18
 - trade model configuration (RFQ) 15
 - Trading DataSource example (C++) 36
 - Trading DataSource example (Java) 26
- Examples (C++)
 - initialization of Trading DataSource 27
- Examples (Java)
 - initialization of Trading DataSource 16
- Executable Streaming Price (ESP)
 - example of trade model 7
 - state diagram 10
- F -**
- Factory (C++)
 - for trading state machines 27
- Factory (Java)
 - for trading state machines 16
- Field
 - in trade event 8
- G -**
- Glossary 46
- Guard 7
- I -**
- Identity theft
 - preventing 38
- Integrity
 - of trading 41
- J -**
- Java
 - API for Trading DataSource, examples 16
 - trademark 4

- K -

KeyMaster 43

- L -

Liberator

- access through KeyMaster 43
- cache 44
- mapping trade messaging objects 38
- optimizing client reconnection time 41
- routing trade messages 40
- session ID 43
- throttling 41

Liberator configuration for trading

- summary of configuration items 37

Listener interface

- in trading system (C++) 29
- in trading system (Java) 17

Listener object

- ChannelListener (C++) 29, 31
- ChannelListener (Java) 17, 19
- TradeListener (C++) 29, 30
- TradeListener (Java) 17, 18

Listener object (C++)

- TradingApplicationListener 27

Listener object (Java)

- TradingApplicationListener 16

LoggerImpl

- in C++ Trading DataSource 28

Logging

- in C++ Trading DataSource 28

- M -

Message

- encapsulated by TradeEvent (C++) 30
- encapsulated by TradeEvent (Java) 18

Messages

- causing state transitions 7
- custom 5
- for trading 5
- on blotter channel 8
- on trade channels 7
- raising trade event 8
- transmitting RequestId and TradeId 7

MsgType field

- in trade event 8

- N -

Note: 44

- General Liberator configuration 37
- Setting for session-id-len 43
- Throttling trade messages 41
- Throttle-time setting for trade messages 41

- O -

object-map

- Liberator configuration example 38
- Liberator configuration item 37
- use in preventing identity theft 38

ORD trade model

- in Trading DataSource kit 14
- state diagram 12

Order (ORD)

- state diagram 12

output-queue-size

- Liberator configuration – definition 41
- Liberator configuration item 37

Overview

- of Caplin Xaqua 2

- P -

Performance

- of trading 41

- R -

Readership 1

Reconnection time

- optimizing for client 41

Reference Implementation Trading DataSource 26

Request for Quote (RFQ)

- example of trade model 7
- state diagram 13

Request for Stream (RFS)

- state diagram 11

RequestId 7

- Restored Trade event
 - C++ 35
 - Java 25
- Restoring Trades
 - C++ 34
 - Java 24
- RFQ Trade Model
 - example XML configuration 15
 - state diagram 13
- RFS Trade Model
 - in example package of Java Trading DataSource kit 26
 - in Trading DataSource kit 14
 - state diagram 11
- S -**
- Security
 - of Trades 43
- Session ID
 - in Liberator 43
 - size of 43
- session-id-len
 - Liberator configuration item 37
 - recommended size 43
- Single sign-on system 43
- State
 - concept, in trade model 7
 - defining in XML configuration 15
- State machine 16, 27
- T -**
- Terms, glossary of 46
- throttle-times
 - parameter of add-object 41
- Throttling
 - overview 44
 - settings for trading 41
- timeout
 - in Trade Model on client 13
 - in Trade Model XML configuration 15
- Tip:
 - Location of reference Liberator configuration file 37
 - Use of the term “conflation” 44
- Trade
 - closing (C++) 31
 - closing (Java) 19
 - definition of 7
 - handling new Trade using ChannelListener (C++) 29
 - handling new trade using ChannelListener (Java) 17
 - security of 43
- Trade Channel
 - definition of 7
 - Liberator configuration 38
 - notifying closure (C++) 31
 - notifying closure (Java) 19
 - notifying creation (C++) 29
 - notifying creation (Java) 17
- Trade Event
 - definition of 8
 - handling by listener object 16, 27
- Trade messaging
 - configuration for routing to Trading DataSource 40
 - role of TradeEvent object (C++) 30
 - role of TradeEvent object (Java) 18
 - role of TradeListener object (C++) 30
 - role of TradeListener object (Java) 18
 - setting discard-timeout in Liberator 41
 - setting throttle-times in Liberator 41
- Trade messaging (C++)
 - handling via DataSource listeners 27
- Trade messaging (Java)
 - handling via DataSource listeners 16
- Trade messaging object
 - in Liberator 38
- Trade model
 - configuration overview 5
 - configuring 14
 - definition of 7
 - ESP state diagram 10
 - ORD state diagram 12
 - relationship to trade 7
 - relationship to Trading DataSource 5
 - RFQ state diagram 13
 - RFS state diagram 11
 - supported in Trading DataSource kit 14
- Trade object 17, 18, 29, 30
- Trade Restoration id
 - C++ 34, 35
 - Java 24, 25
- Trade System
 - events raised by 18, 30

- Trade System
 - messages from TradeListener (C++) 30
 - messages from TradeListener (Java) 18
- TradeEvent object (C++) 30
- TradeEvent object (Java) 18
- Tradeld 7
- TradeListener (Java)
 - custom 17
- TradeListener (C++)
 - custom 29
 - handling trade events 30
- TradeListener (Java)
 - handling trade event 18
- Trading DataSource
 - API overview 5
 - example of Liberator configuration for 40
 - overview and architecture 5
 - relationship to trade models 5
 - standard trade models in kit 14
 - using the C++ API 27
 - using the Java API 16
- Trading GUI
 - overview 5
- Trading integrity 41
- Trading performance 41
- Trading state machine (C++) 27
- Trading state machine (Java) 16
- Trading System
 - cancelling a trade (C++) 31
 - cancelling a trade (Java) 19
 - events from 8
 - events from (C++) 29
 - events from (Java) 17
 - initialization within (C++) 29
 - initialization within (Java) 17
 - listener interface (C++) 29
 - listener interface (Java) 17
 - overview 5
 - processing events 7
- TradingApplicationListener (C++)
 - notifying when trade channel created 29
- TradingApplicationListener (C++)
 - example 27
 - registering BlotterTradeListener 32
- TradingApplicationListener (Java)
 - example 16
 - notifying when trade channel created 17
 - registering BlotterTradeListener 20
- TradingDataSource (C++)
 - creating in custom application 27
 - logging 28
- TradingDataSource (Java)
 - creating in custom application 16
- Transition
 - concept, in trade model 7
 - defining in XML configuration 15
 - of trade state 7
- V -**
- Value
 - in trade event 8
- X -**
- XML configuration
 - example for RFQ trade model 15
 - in TradingDataSource kit 14
 - overview 5

Contact Us

Caplin Systems Ltd
Cutlers Court
115 Houndsditch
London EC3A 7BR
Telephone: +44 20 7826 9600
www.caplin.com

The information contained in this publication is subject to UK, US and international copyright laws and treaties and all rights are reserved. No part of this publication may be reproduced or transmitted in any form or by any means without the written authorization of an Officer of Caplin Systems Limited.

Various Caplin technologies described in this document are the subject of patent applications. All trademarks, company names, logos and service marks/names ("Marks") displayed in this publication are the property of Caplin or other third parties and may be registered trademarks. You are not permitted to use any Mark without the prior written consent of Caplin or the owner of that Mark.

This publication is provided "as is" without warranty of any kind, either express or implied, including, but not limited to, warranties of merchantability, fitness for a particular purpose, or non-infringement.

This publication could include technical inaccuracies or typographical errors and is subject to change without notice. Changes are periodically added to the information herein; these changes will be incorporated in new editions of this publication.

Caplin Systems Limited may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time.

This publication may contain links to third-party web sites; Caplin Systems Limited is not responsible for the content of such sites.