CAPLIN

# Caplin Trader 1.3

## Integrating Caplin Trader With A Trading System

November 2008

# Contents

# 1 Preface

## 1.1 What this document contains

This document describes how the Caplin Trading DataSource allows you to integrate Caplin Trader with your existing Trading System.

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

### 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.

  **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, Enterprise Architects, and System Architects, who require an overview of the Caplin Trading DataSource and its Java and C++ APIs.

## 1.3 Related documents

- **Caplin Trader Architecture**

  This document describes the architecture of Caplin Trader. It focuses on the use of the Caplin Platform in trading applications. It also identifies the areas in which the Platform can be integrated with your company's own and third-party systems.

- **Caplin Java Trading DataSource: API Documentation**

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

- **Caplin C++ Trading DataSource: API Documentation**

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

- **Caplin Trader Trade Model XML Reference**

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

## 1.4 Typographical conventions

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

| *Type* | *Uses* |
|---|---|
| **aMethod** | Function or method name |
| *aParameter* | Parameter or variable name |
| */AFolder/Afile.txt* | File names, folders and directories |
| `Some code;` | 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 |
| **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.

## 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.

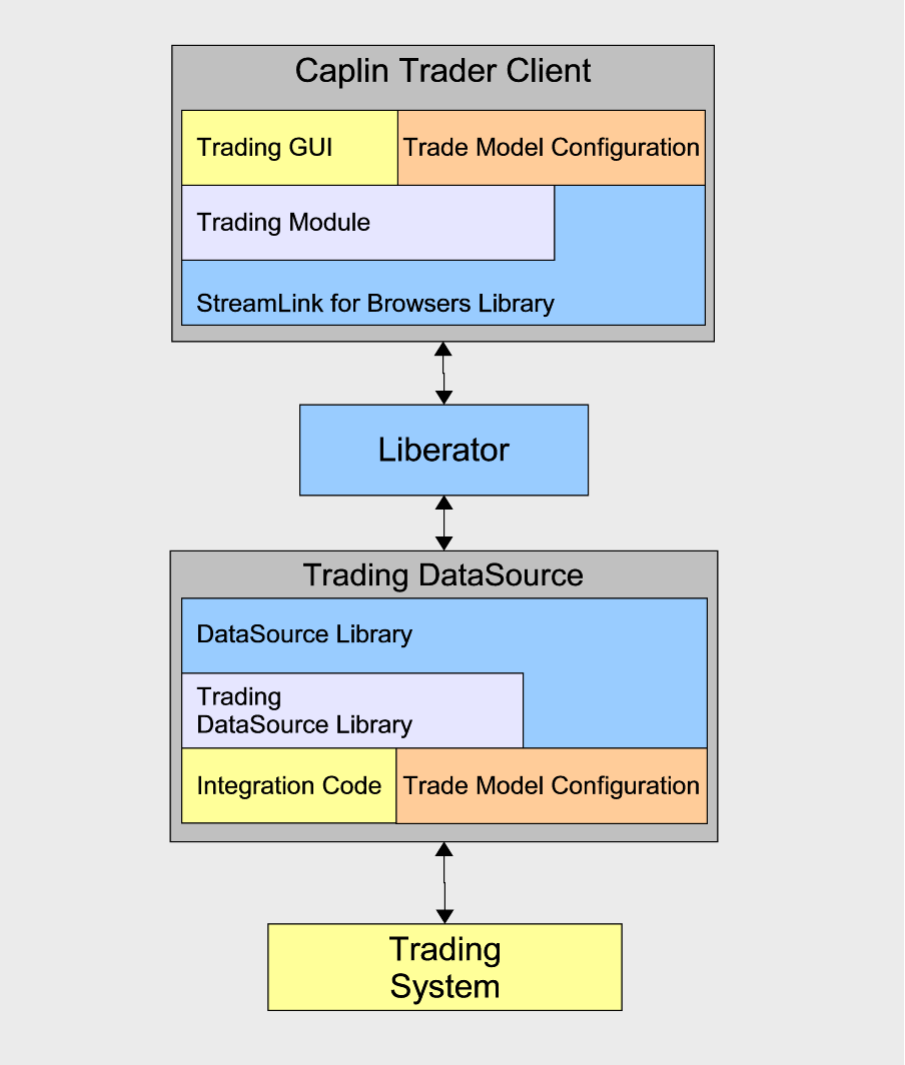
Please email your thoughts to documentation@caplin.com.

## 1.6 Acknowledgments

# 2 Overview

Caplin Trader consists of a number of components (see the **Caplin Trader Architecture**). The main components used to integrate Caplin Trader 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 Trader.




**Simplified Caplin Trader 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 Trader 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 SDK. It gives full access to all the functionality of Caplin DataSource SDK; this allows you to send and receive custom messages in addition to Trade messaging.

## Trading GUI

The Trading GUI is the part of the Caplin Trader Client that displays trading tickets and quick 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 quick 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 Trader 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 Trader uses a number of concepts to represent trading; models, channels, trades and 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 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. It also defines guards on transitions, which are checks to see if a transition can be performed; for example checking that certain fields are present in an event.

## 3.2 Trade Channels

A Trade Channel represents a single user's communication between the Caplin Trader Client and the Trading DataSource. It is a private channel for bidirectional messaging and all messages relating to trades for a user will be sent and received on the user's channel.

The Caplin Trader Client opens a Trade Channel by subscribing to an object. The Caplin Liberator maps this subscription to a unique object name for that user 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 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 a 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 and then 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.

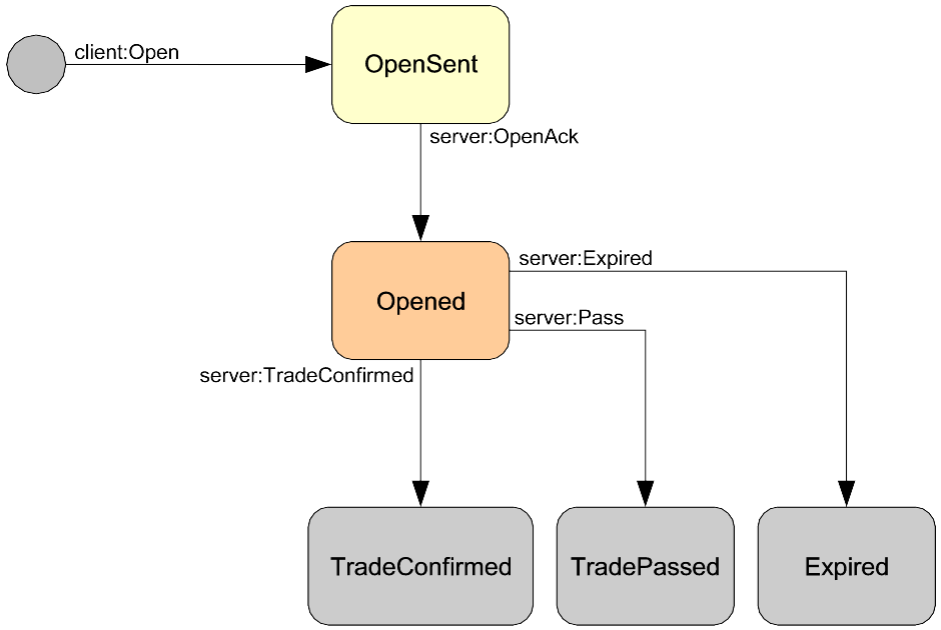
# 4 Example Trade Models

The following sections show examples of Trade Models that can be used with Caplin Trader. 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 Trader and is used by the demonstration TradingDataSource and the Caplin Trader Client Reference Implementation, for one-click trading via the Trade Tile.

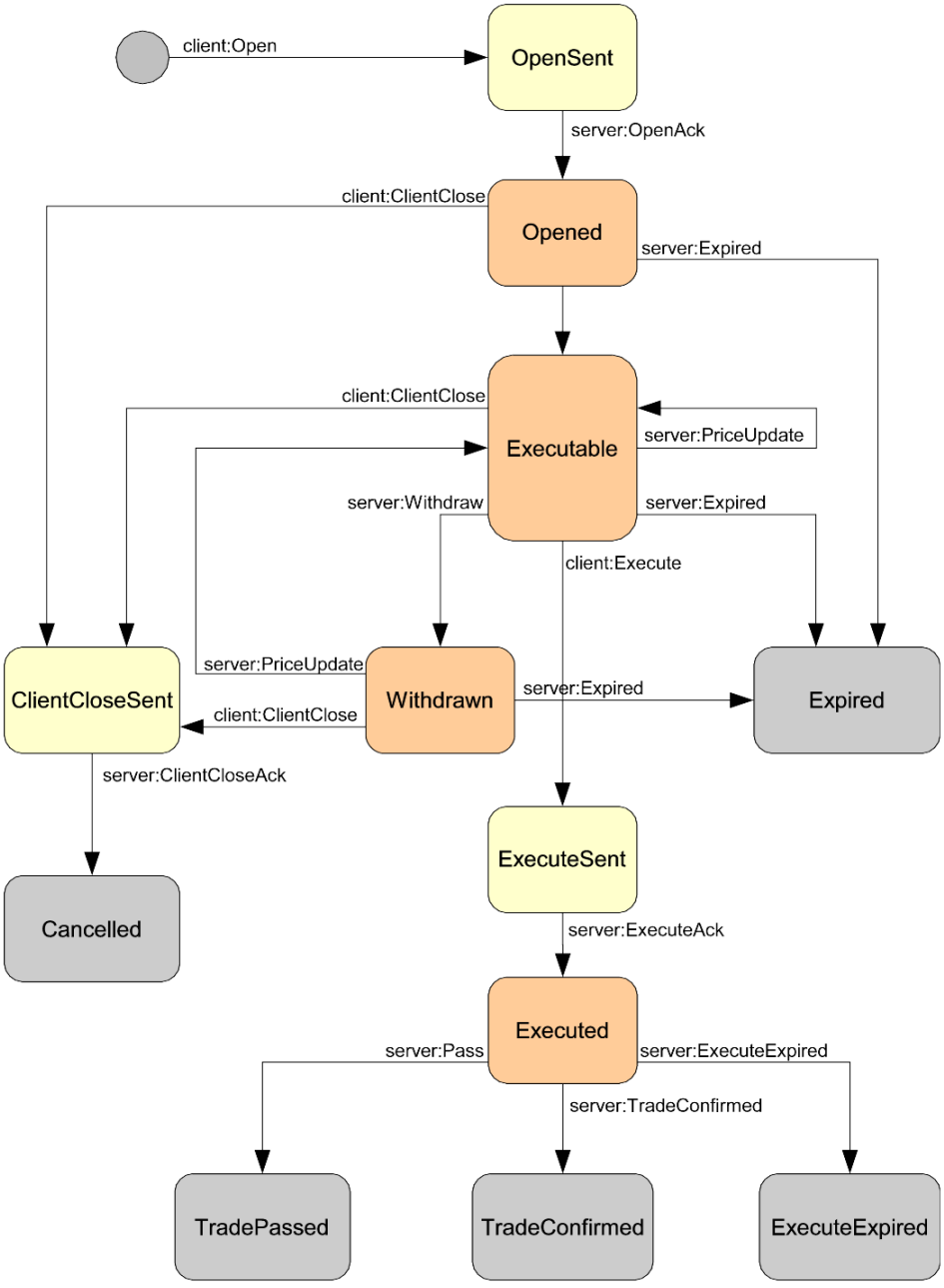



**State diagram for ESP Trade Model**

## 4.2 Example Request for Stream (RFS)

This state diagram shows the example Request For Stream (RFS)

Trade Model. This model is provided with Caplin Trader and is used by the demonstration TradingDataSource and the Caplin Trader Client Reference Implementation, for ticket-based trades.



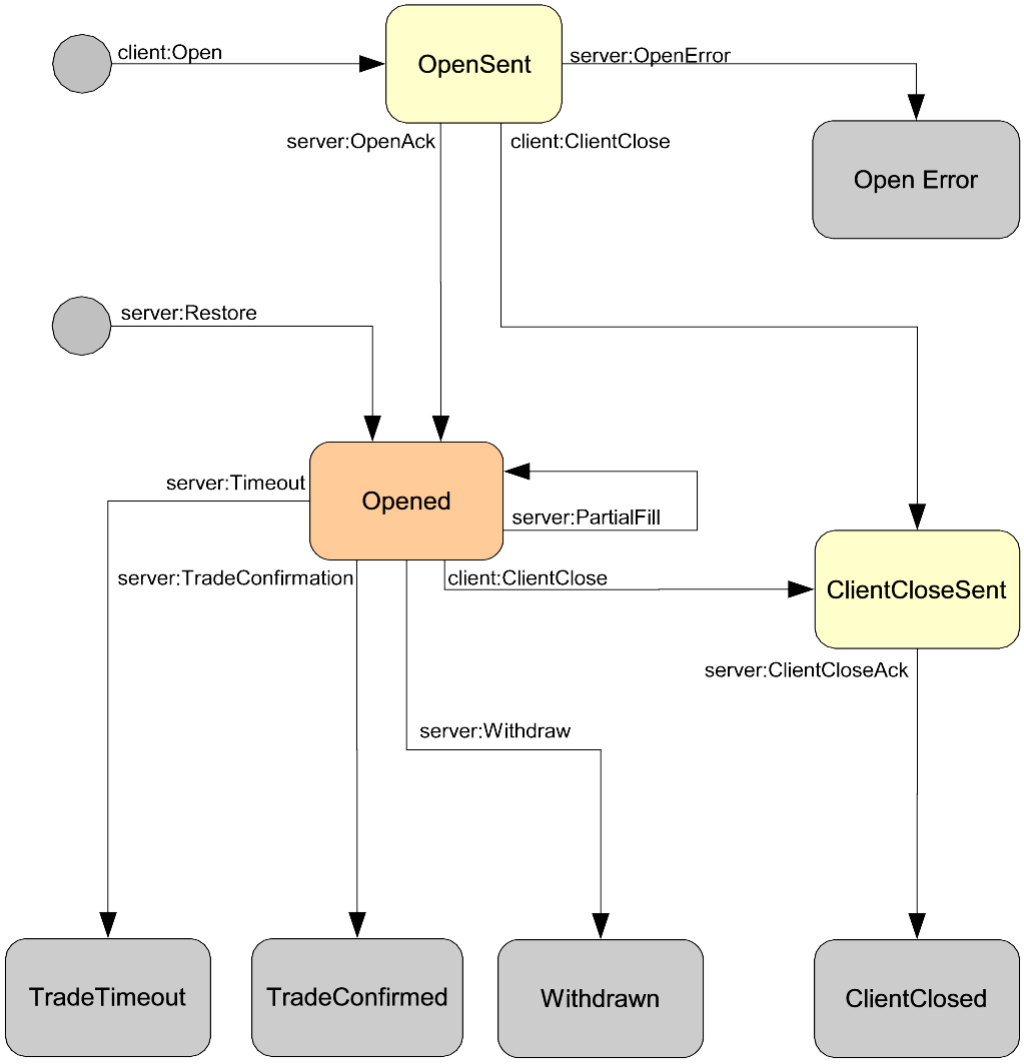

**State diagram for Example RFS Trade Model**

## 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 Trader and is used by the demonstration TradingDataSource. It is not currently used by the Caplin Trader Client, but will be included in a forthcoming version of the Caplin Trader Client Reference Implementation for ticket-based order trades.

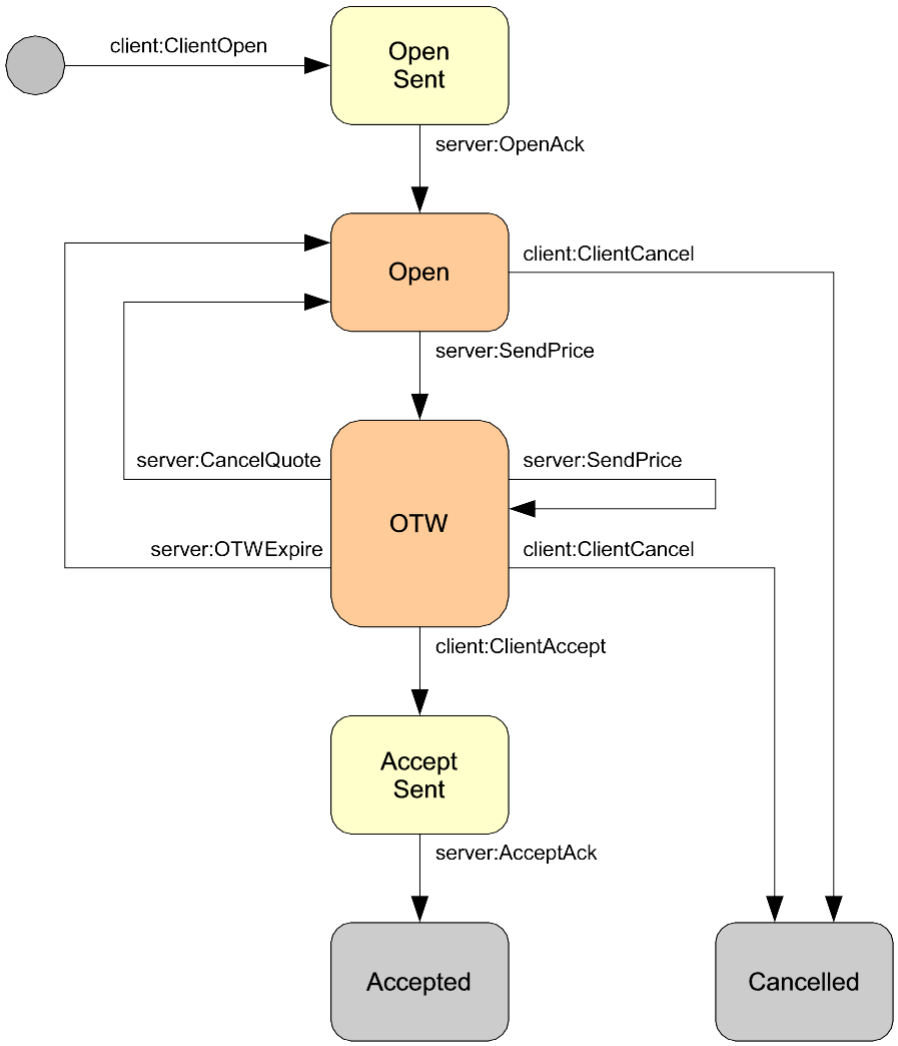



**State diagram for Example ORD Trade Model**

## 4.4 Simple Request for Quote (RFQ)

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

This particular model is not used by the Caplin Trader Client Reference Implementation, but is typical of an RFQ workflow. Additional states can easily be added before the Open state to account for credit checks and other validation steps as required.




**State diagram for RFQ Trade Model**

# 5 Configuring Trade Models

Caplin Trader 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 they can be adapted to your requirements.

For the detailed definition of the Trade Model XML, see the document **Caplin Trader Trade Model XML Reference**.

## 5.1 Simple RFQ Example

The following is a simple example of the XML configuration for a simple RFQ Trade Model. This example only shows states and transitions and does not check any events for required fields. See Simple Request for Quote (RFQ) 11 to see the Trade Model that this configuration represents.

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

    <state name="OpenSent">
      <transition target="Open" trigger="OpenAck" source="server" />
    </state>

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

    <state name="OTW">
      <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">
      <transition target="Accepted" trigger="AcceptAck" source="server" />
    </state>

    <state name="Accepted">
    </state>

    <state name="Cancelled">
    </state>

  </tradeModel>
</stateModels>
```

## 5.2 Checking Fields

The XML configuration allows fields to be checked before a transition will take place. You can define different types of checks, such as whether a field exists, whether it is a certain value, or whether it is greater than a certain value.

A transition definition can define fields to check. In the example below the fields `RequestId` and `TradeId` are both checked to make sure they exist.

```
  <state name="OpenSent">
    <transition target="Open" trigger="OpenAck" source="server">
      <fields>
        <field name="RequestId" exists=”true” />
        <field name="TradeId" exists=”true” />
      </fields>
    </transition>
  </state
```

Sets of fields to be checked against can be defined outside the scope of a particular Trade Model. This allows multiple models to make use of the same sets of field checks. In the following example the Open state has a transition that will check the `FieldSet1` fields definition, where `FieldSet1` is defined outside the scope of the model.

```
<fieldsDef id="FieldSet1">
  <field name="MsgType" exists="true" />
  <field name="MsgVersion" greaterThan="0" />
  <field name="RequestId" exists="true" />
</fieldsDef>

<tradeModel name=”RFQ”>

  <state name="Open">
    <transition target="OTW" trigger="PriceUpdate" source="server">
      <fields id="FieldSet1" />
    </transition>
  </state>

</tradeModel>
```

Sets of fields can also reference other sets of fields; this is useful when multiple messages have a common set of fields to check. The following example shows `FieldSet1` referencing `StdFields`, this has the effect of combining both field definitions at runtime so all fields are checked.

```
<fieldsDef id="StdFields">
  <field name="MsgType" exists="true" />
  <field name="MsgVersion" greaterThan="0" />
  <field name="RequestId" exists="true" />
</fieldsDef>

<fieldsDef id="FieldSet1">
  <fields id=”StdFields” />
  <field name="Price" exists="true" />
  <field name="PriceVersion" greaterThan=”0” />
</fieldsDef>
```

Additionally, models can parameterize their use of field sets, which allows a single model to be used by multiple Trade types, such as Spots, Swaps, Forwards, and so on, and different sets of fields are checked accordingly. The following example shows a single fields definition, `PriceUpdateFields`, used by the `PriceUpdate` transition in the `Open` state. The fields checked depend on a run-time parameter, the `TradingType` field, which defines whether the trade is a Spot or a Swap.

```
<fieldsDef id="PriceUpdateFields">
  <param value="SPOT">
    <field name="AskPrice" greaterThan="0" />
    <field name="BidPrice" greaterThan="0" />
  </param>
  <param value="SWAP">
    <field name="L1_AskPrice" greaterThanField=”L1_BidPrice” />
    <field name="L1_BidPrice" greaterThan="0" />
    <field name="L2_AskPrice" greaterThanField="L2_BidPrice" />
    <field name="L2_BidPrice" greaterThan="0" />
  </param>
</fieldsDef>

<tradeModel name=”RFQ”>

  <state name="Open">
    <transition target="OTW” trigger="PriceUpdate" source="server">
      <fields id="PriceUpdateFields"
              paramType="context" paramField="TradingType" />
    </transition>
  </state>

</tradeModel>
```

# 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 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**

```
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.

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

      // Once the TradingDataSource has been created
      // it has to be started explicitly.
      myTradingDataSource.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 12. The `StateMachineFactory` is then passed to the `TradingDataSource` constructor, so that the DataSource can create a the 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 Platform components, such as Liberator.

## 6.2 New Channels

The `TradingApplicationListener` is notified when new 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 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("RFS"))
   {
      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 trade 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 trade 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 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 `Trade Events`; 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 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 a user logs off the system the trade channel for that 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
}
```

# 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 Client 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 C++ Trading DataSource: API 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()
   {
      // Prepare a vector of filenames of the config files to use
       std::vector<std::string> configFiles;
       configFiles.push_back("conf/MyESPStateModel.xml");
       configFiles.push_back("conf/MyRFSStateModel.xml");

      // Create the Trading DataSource, passing in a pointer to a class that
      // implements TradingApplicationListener (so our 'this' pointer is fine)
      m_tradingDataSource.reset(new TradingDataSource(this,
                                                      "MyDataSourceConfig.conf",
                                                      configFiles));

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

   // ...
private:
   std::auto_ptr<TradingDataSource> m_tradingDataSource;
}
```

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 12 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 Platform components, such as Liberator.

## 8.2 New Channels

The `TradingApplicationListener` is notified when new 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 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());
}
```

## 8.3 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()**

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

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.

## 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 `Trade Events`; 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 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 a user logs off the system the trade channel for that 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
  // ...

}
```

# 9 The C++ Trading DataSource Example

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

The Visual Studio 2005 project file for the example is 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 Glossary of terms and acronyms

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

| Term | Definition |
|---|---|
| **API** | Application Programming Interface |
| **Caplin Platform** | The Caplin Platform is a suite of software products for on-line financial trading and Web delivery of real-time market data. |
| **Caplin Trader** | Caplin Trader is a complete platform and toolkit for building multi-product trading portals. It is built on the **Caplin Platform**. |
| **Caplin Trader Client** | Caplin Trader Client is a platform neutral Web application that provides a rich trading workstation in a browser. It is based on an Ajax framework called **webcentric**, into which you can place any Web content created in HTML, Ajax, Adobe Flex™, or any other similar technology. |
| **DataSource** | DataSources are software adapters within the **Caplin Platform** that connect the Platform to external sources of real time data and external **Trading Systems**. In other Caplin documents DataSources are also called DataSource adapters. |
| **ESP** | Executable Streaming Price **Trade Model**. |
| **Liberator** | Caplin Liberator is a bidirectional streaming push server designed to deliver market data and trade messages over any network that supports Web traffic. |
| **RFQ** | Request for Quote **Trade Model**. |
| **SDK** | Software Development Kit |
| **Trade** | In this document the term Trade (with a capital T) represents a single trade for a user. This could be an **RFQ**, an Execution on a streaming price (**ESP**), or any other type of trade. See Trades 6. |
| **Trade Channel** | A single user's communication between the client application and the **Trading DataSource**. See Trade Channels 6. |
| **Trade Event** | An action by a client or an event from the Trading System. See Trade Events 7. |
| **Trading DataSource** | The **DataSource** used to integrate **Caplin Trader** 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 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. See Trade Models 6. |
| **Trading System** | The term used in this document to refer to systems that support trade capture. |

# Index

## - A -



## - C -



## - D -



## - E -

## - F -



## - G -



## - J -



## - L -



## - M -



## - O -



## - R -

## - S -



## - T -

## - V -



## - X -

CAPLIN THE REAL TIME WEB COMPANY

## Contact Us

Caplin Systems Ltd
Triton Court
14 Finsbury Square
London EC2A 1BR
Telephone: +44 20 7826 9600
Fax: +44 20 7826 9610
**www.caplin.com**