CAPLIN

Trading 6.0

Integrating The Caplin Platform With A Trading System

March 2013

Contents

1 Preface

1.1 What this document contains

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

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

Note: The Java examples shown in Using the Trading Integration API for Java 19 apply to the **Caplin Trading Integration API** in the Caplin Integration Suite 6.0 or later. For examples that use version 4.x of the API (called the **Trading DataSource API**), refer to the section “Using the Trading DataSource Java API” in the document **Caplin Xaqua 1.0: Integrating Caplin Xaqua With A Trading System**.

Note: The diagrams and examples in this document show the **client application** to be a Caplin Trader application, but the Trading Adapter can serve any **client application** that is built to interact with the Caplin Platform via **Caplin Liberator**.

About Caplin document formats

This document is supplied in two 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* folder and open the file *index.html*.

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.

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 the Caplin Platform. The second half of the document also explains how Developers can implement a Trading Adapter in Java or C++, and how to configure Caplin Liberator for trading.

1.3 Related documents

- **Caplin Platform Overview**

  Provides a technical overview of the Caplin Platform, including an explanation of its architecture.

- **Caplin DataSource Overview**

  A technical overview of Caplin DataSource.

- **Caplin Integration Suite for Java API Documentation: Trading Integration API section**

  The reference documentation for the Caplin Trading Integration API for Java.

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

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

- **Caplin Trading: 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 Xaqua: Trade Model Configuration XML Reference
  and **Caplin Trader: Trade Model Configuration XML Reference**.)

- **Caplin Integration Suite for Java API Documentation: DataSource API section**

  The reference documentation for the Java DataSource API.

- **Caplin Integration Suite: How To Create A Platform Java Blade**

  This document explains how to create new Java-based Caplin Platform blades, including Trading Adapter blades, using the Caplin Integration Suite Toolkit.

- **Caplin DataSource for C API Documentation**

  The reference documentation for the C DataSource API.

- **Caplin Trader: API Specification**

  Documents the JavaScript libraries for implementing client applications that use Caplin Trader.

- **Caplin Liberator Administration Guide**

  Describes the Caplin Liberator server and its place within the Caplin Platform. Explains how to install, configure, and manage the Liberator. Includes configuration reference information, and a list of Liberator's log and debug messages.

- **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.

- **Caplin Permissioning: How To Create A Permissioning Adapter**

  Describes how you can use the Permissioning Integration API in the Caplin Integration Suite for Java to create a Permissioning Adapter.

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
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

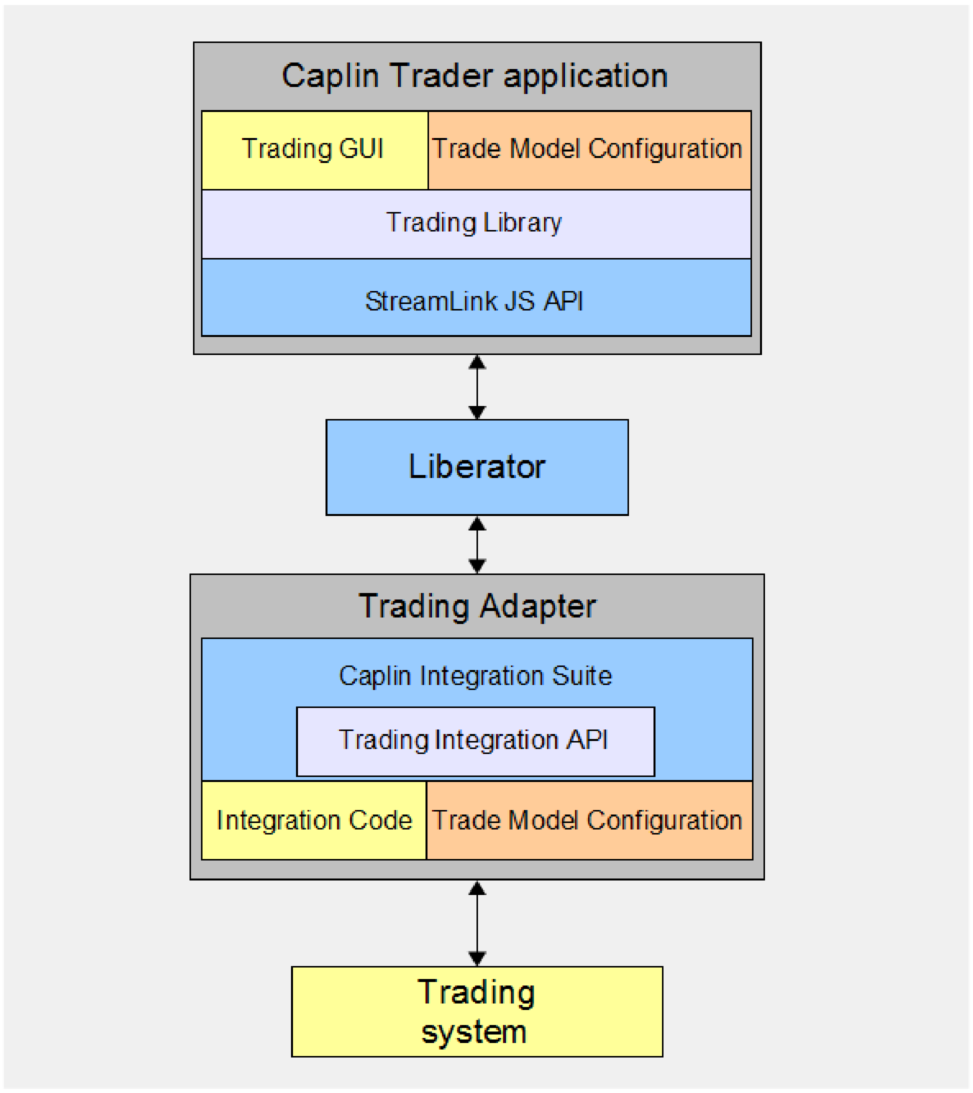
Adobe, *Adobe® Reader*, and *Flex* are either registered trademarks or trademarks of Adobe Systems Incorporated is a registered trademark of Adobe Systems Incorporated in the United States and/or other countries.

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

Java is a trademark or registered trademark of Oracle® Corporation in the U.S. and other countries.

2 Overview

The **Caplin Platform** consists of a number of components (see the **Caplin Platform Overview**).
The main components used to integrate the Caplin Platform with your trading system are the Trading Adapter and, if your trading client is a Caplin Trader application, the Trading GUI. The following diagram shows the basic architecture of the trading integration components and how they fit into the Caplin Platform and **Caplin Trader**.



Simplified Caplin Platform and Caplin Trader architecture showing only Trading Integration components

Trading system

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

Trading Adapter

The Trading Adapter is an **Integration Adapter** that constitutes the interface between the Caplin Platform 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 that are sent between clients and the trading system via the Liberator.

The Trading Adapter is implemented using the **Caplin Integration Suite**, and in particular the suite's **Trading Integration API** that exchanges trade messages with other Caplin Platform components such as Caplin Liberator. The Adapter includes the custom code required to integrate with your trading system. It can be a standalone process or part of an existing one.

The Trading Integration API is available in both Java (see Using the Trading Integration API for Java 19) and C++ (see Using the Trading Integration API for C++ 49).

The Caplin Integration Suite includes a Code Generator that helps you to produce correct Java Trading Adapter code more quickly. For more about this, see Using the Code Generator 40 in Using the Trading Integration API for Java 19.

Trading GUI

The Trading GUI is the part of a Caplin Trader application 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 Adapter via the Trading Library, the **StreamLink JS API**, and Liberator.

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 Adapter 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.

The Caplin Platform 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 Integration Library and Trading Library will control and verify the states and transitions allowed. This simplifies the integration process, as most of the logic is handled for you and is defined by your configuration.

3 Trading concepts

The Caplin Platform 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 the Caplin Platform and the Trading Adapter. 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 60) and subscribes to the object from the Trading Adapter. When the Trading Adapter responds to this subscription, a private channel is effectively created for messages in both directions between the client and the Trading Adapter; this is the Trade Channel.

Many deployments would use a single Trade Channel. However, sometimes it is useful to have multiple Trading Adapters, each handling different asset classes. In this case the end-user could have a separate Trade Channel for each Trading Adapter; 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 Adapter 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 Trading Adapter 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 Adapter. 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 Adapter 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.

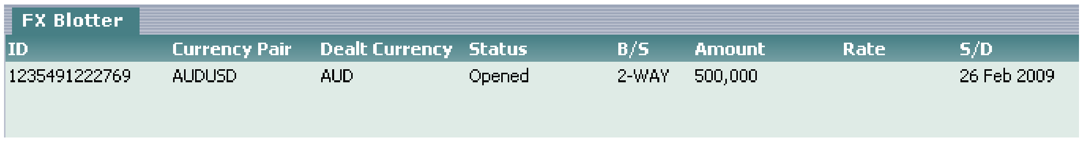
The Caplin Trader application opens a Blotter Channel by subscribing to an object whose subject name matches one of the Blotter Channel patterns defined for the Trading Adapter. (For detailed information on how to configure the Trading Adapter Channel patterns, see Configuring Trade subjects 31.) If 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, otherwise a container-based blotter channel is opened.

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 Adapter. The Trading Adapter responds to this subscription by creating a unique Blotter Channel.

Subsequently, whenever the end-user interacts with the trading subsystem, the Trading Adapter 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.

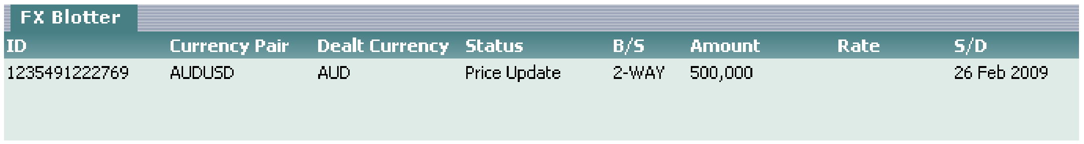
The following pictures show the blotter in a Caplin Trader application being updated at successive stages in the execution of an FX Trade using the RFS Trade Model. (For simplicity only the left hand side of the blotter is shown; in reality there are more fields on the right hand side of the blotter.)

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


FX Blotter

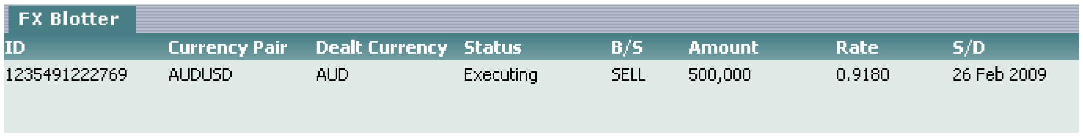
ID	Currency Pair	Dealt Currency	Status	B/S	Amount	Rate	S/D
1235491222769	AUDUSD	AUD	Opened	2-WAY	500,000		26 Feb 2009

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

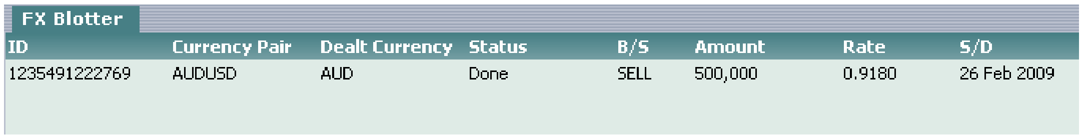
3. When the end-user clicks the Sell button on the Trade Ticket, the blotter entry status becomes Executing.



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 a Caplin Trader application

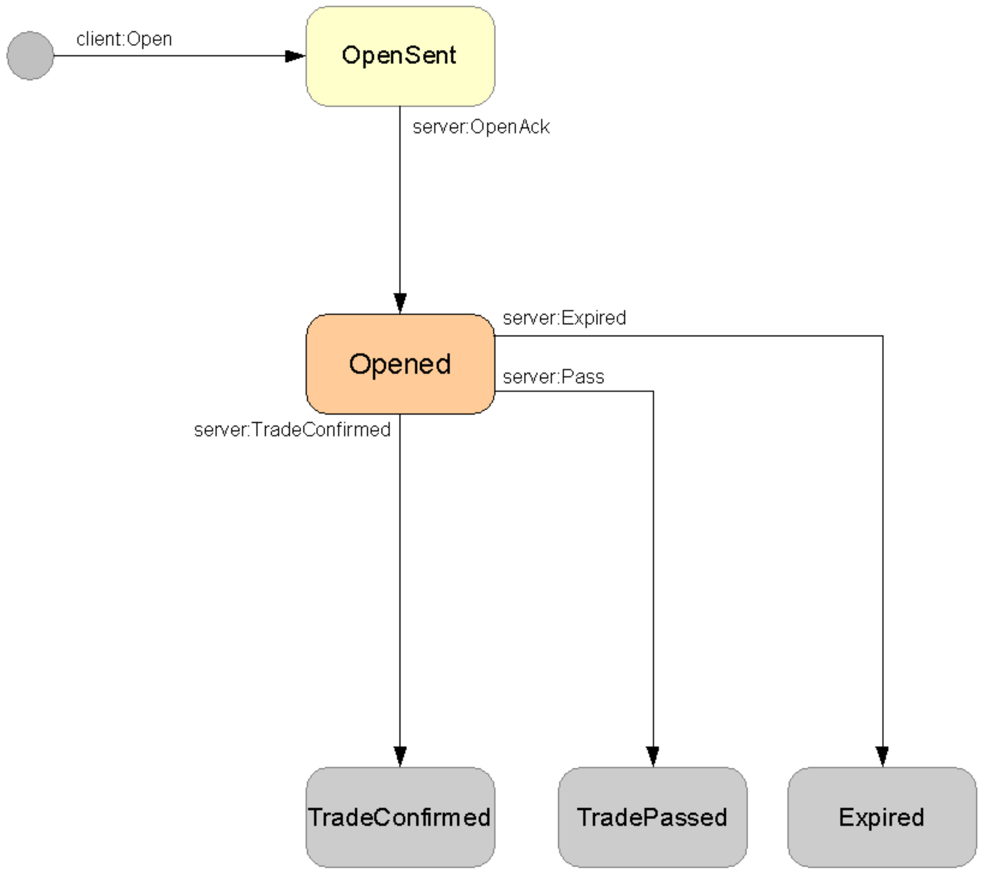
The client should normally subscribe to a separate Blotter Channel for each asset class traded—for example, an FX blotter and an FI blotter — since the type of information that needs to be displayed on the blotter varies according to asset class. The client would be set up to subscribe to different object 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 the Caplin Platform. More complicated Trade Models can be used with little extra complexity of integration.

4.1 Example Executable Streaming Price (ESP)

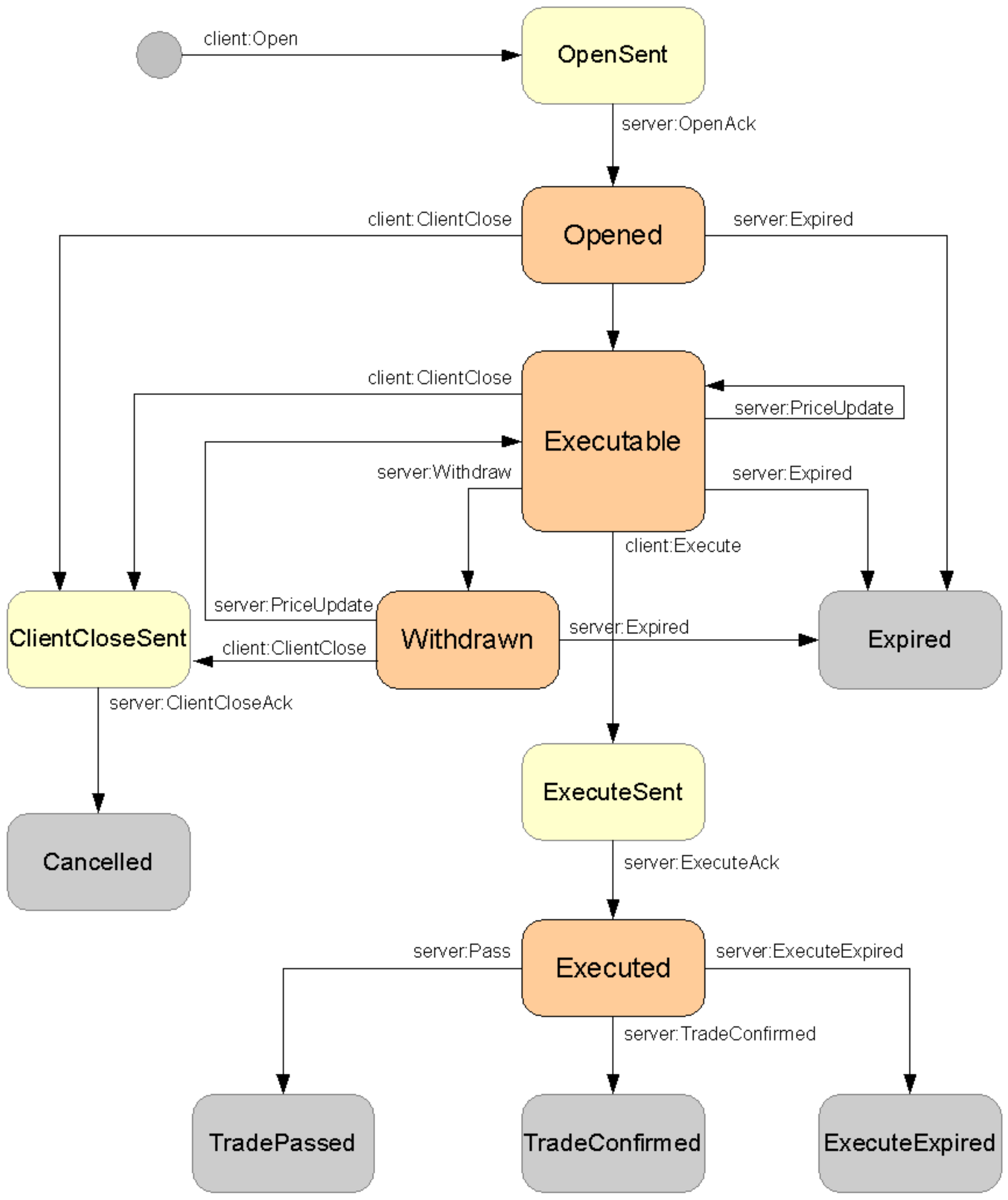
This state diagram shows an example Executable Streaming Price (ESP) Trade Model.



State diagram for ESP Trade Model

4.2 Example Request for Stream (RFS)

This state diagram shows an example Request For Stream (RFS) Trade Model that is typically used for ticket-based Trades.

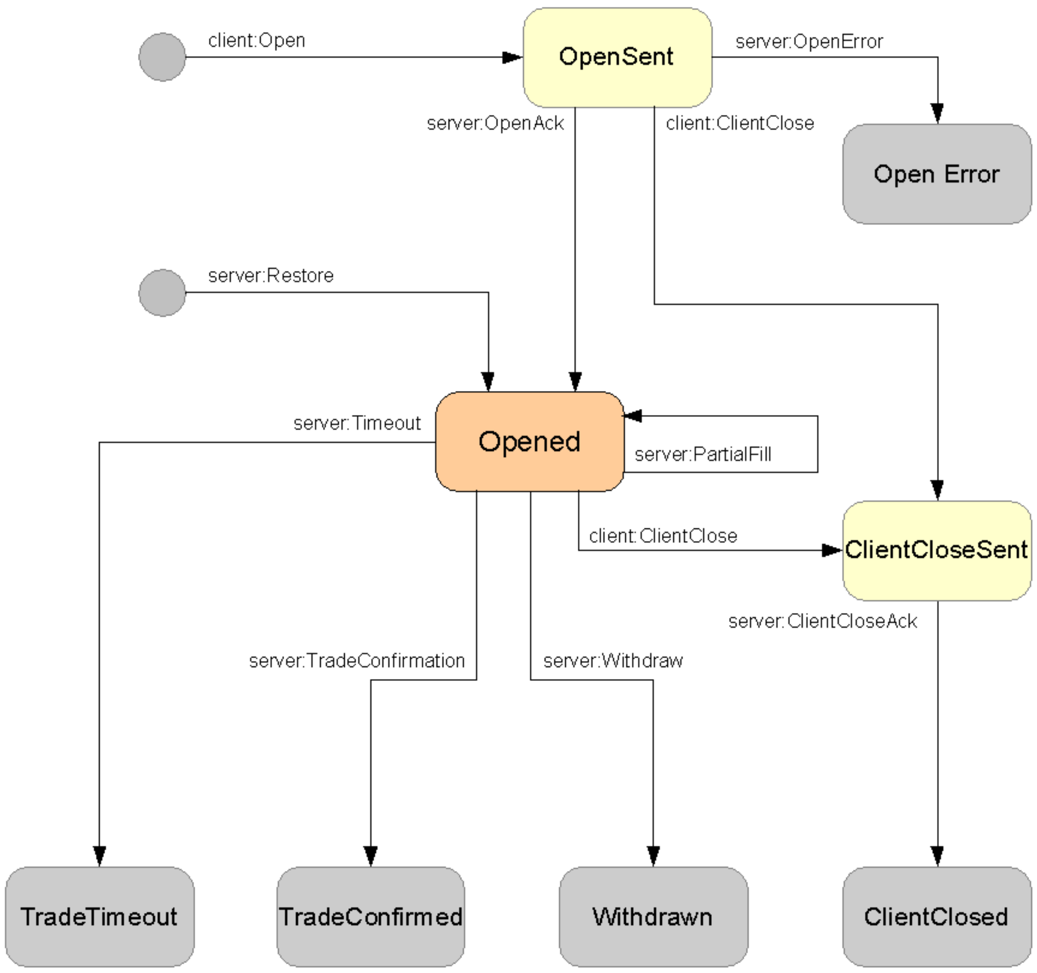


State diagram for Example RFS Trade Model

4.3 Example Order (ORD)

This state diagram shows an 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.



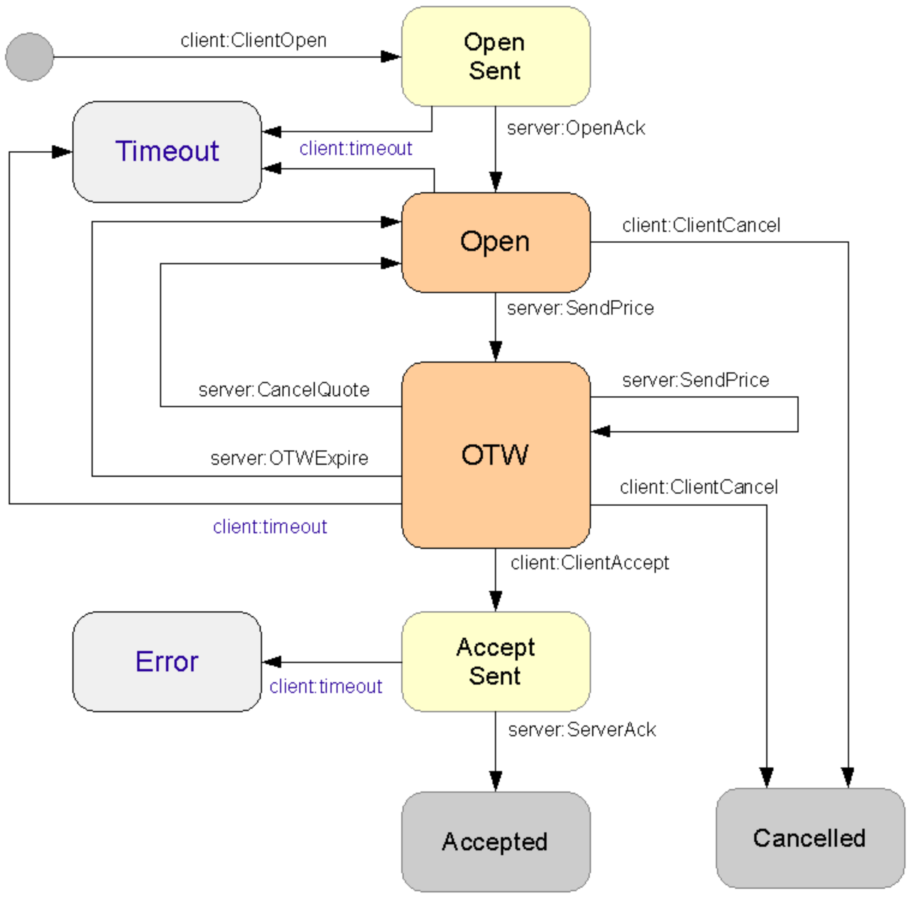
State diagram for Example ORD Trade Model

4.4 Request for Quote (RFQ) with timeouts

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

This model 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 16 section of Configuring Trade Models 15.



State diagram for RFQ Trade Model

5 Configuring Trade Models

The Caplin Platform 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.

In Trading Adapters that are based on the Trading Integration API for Java, the Trade Model configuration XML files are specified in a Java properties file (property **trading.models**), together with other configuration. For an example of such a properties file, see Defining the subject pattern properties 31 in Configuring Trade subjects 31. For a full description of the properties, see the “Trading Integration API” section of the document **Caplin Integration Suite for Java API Documentation**.

For the detailed definition of the Trade Model XML, see the document **Caplin Trading: 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 14. 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>
```

5.2 ESP Example with field definitions

The following example is the XML configuration for an ESP Trade Model, where the `<transition>` tags contain field definitions (`<field name="xxx" />`). The `<field>` tags define the DataSource message fields that are relevant to the trade model state transition in which the tags appear.

```
<tradeModels>
  <tradeModel name="ESP" initialState="Initial">
    <state name="Initial">
      <transition target="OpenSent" trigger="Open" source="client">
        <field name="StateModelName" description="ESP model name"
               required="true" />
        <field name="RequestID" description="Request ID"
               required="true" />
        <field name="QuoteID" description="Quote ID"
               required="true" />
        <field name="InstrumentName" description="Instrument Name"
               required="true" />
        <field name="DealtCurrency" description="Dealt currency"
               required="true" />
        <field name="BuySell" description="BuySell indicator"
               required="true" />
        <field name="Price" description="Quote ID" />
      </transition>
    </state>
    <state name="Timeout" />
    <state name="OpenSent" timeout="10" timeoutState="Timeout">
        <transition target="Opened" trigger="OpenAck" source="server">
        <field name="RequestID" description="Request ID" required="true" />
        <field name="StateModelName" description="ESP model name"
               required="true" default="ESP" />
        <field name="BuySell" description="BuySell indicator"
               required="true" default="BUY" />
      </transition>
    </state>
    <state name="Opened" timeout="10" timeoutState="Timeout">
      <transition target="TradeConfirmed" trigger="TradeConfirmation"
                  source="server">
        <field name="RequestID" description="Request ID" required="true" />
        <field name="TradeID" description="Request ID" required="true" />
      </transition>
      <transition target="TradePassed" trigger="Pass" source="server" />
      <transition target="TradeExpired" trigger="Expired" source="server" />
    </state>
    <state name="TradeConfirmed" />
    <state name="TradePassed" />
    <state name="TradeExpired" />
  </tradeModel>
</tradeModels>
```

In this example, the `"Open"` message from the client that triggers the transition from the `Initial` state to the state `OpenSent` has several relevant fields: `StateModelName`, `RequestID`, `QuoteID`, `InstrumentName`, `DealtCurrency`, `BuySell`, and `Price`. All these fields, apart from `Price`, are defined as `"required"`, which means that they must be present in the DataSource message for the transition to occur.

Fields are also defined for the `OpenAck` trigger on the state `OpenSent`, and for the `TradeConfirmation` trigger on the state `Opened`.

Tip: It is recommended that in your Trade Model definitions you define all the relevant message fields for each state transition, particularly the mandatory fields. This creates a well defined contract between client and server; it helps you ensure that client applications and Trading Adapters work properly together.

At run time, the library code behind the Trading Integration API checks the fields in messages received from clients against the XML definitions for the Trade Model. These checks ensure the messages are valid for triggering state transitions in the trade model. The API also makes the same checks on messages generated for transmission to clients. In the XML example, if a message from the client that should normally trigger a transition to the `OpenSent` state does not contain all the fields defined with the XML attribute `required="true"`, the Trading Library raises an error.

Tip: If you use the Caplin Integration Suite's Code Generator to create Trading Adapter code for a Trade Model in which you have defined the message fields, the generator creates for each `<transition>` a `TradeEvent` class with getters and setters for the fields. This helps you to produce correct Adapter code more quickly; you do not have to manipulate a dictionary of fields, and your application always has access to the correct field names.
For more about this, see Using the Code Generator 40, and in particular, What code is generated? 42.

6 Using the Trading Integration API for Java

This section provides a brief description of the main parts of the Trading Integration API for Java (For full details, see the "Trading Integration API" section of the document **Caplin Integration Suite for Java API Documentation**). The Trading Integration API for Java is part of the **Caplin Integration Suite**.

With the Caplin Integration suite, you can implement a Java-based Trading Adapter as a Trading Adapter blade. To create such a blade, use the Caplin Integration Adapter Blade project wizard. For more information about creating Java-based platform blades, see the document **Caplin Integration Suite: How To Create A Platform Java Blade**.

Note: The Java examples shown in the following sections apply to the Caplin Trading Integration API in the Caplin Integration Suite for Java 6.0 or later. For examples that use version 4.x of the Trading Integration API (called the Trading DataSource API), refer to the section "Using the Trading DataSource Java API" in the document **Caplin Xaqua 1.0: Integrating Caplin Xaqua With A Trading System**.

Tip: With the Caplin Integration Suite for Java, you can create an Integration Adapter that uses both the Trading Integration API for Java and the Permissioning Integration API for Java. This allows the same application to act both as a Trading Adapter and as a **Permissioning Adapter**. For details of how to include permissioning functionality in your Integration Adapter, see the document **Caplin Permissioning: How To Create A Permissioning Adapter**.

To implement a Trading Adapter you must set up listener objects to handle events occurring on Trades and create events to be sent back into the system, as shown in the following sections. The Caplin Integration Suite includes a Code Generator that helps you to produce correct Java Trading Adapter code more quickly. The generated code includes many of the listener interfaces. For more about this, see Using the Code Generator 40.

6.1 Initialization

When your Trading Adapter starts, it should create an instance of `TradingProvider` and register itself as a `TradingApplicationListener`. The `TradingProvider` uses a `DataSource` object (from the underlying DataSource library), which connects to Liberator. The `DataSource` also sets up the necessary DataSource listeners internally to handle the trade messaging.

The `TradingProvider` is configured through a Java properties file given to the `DataSource`.
This file specifies:

- The XML configuration files that define the Trade Models the `TradingProvider` is to use.

  See Configuring Trade Models 15.

- The subjects of the Trade objects that represent messages on Trade Channels, messages on Blotter Channels, and Blotter Items.

  See Defining the subject pattern properties 31 in Configuring Trade subjects 31.

- The location and name of the audit log file, if you want to change the default setting.

  The default location of the audit log is the *logs* directory of your Trading Adapter installation. The default name of the log file is defined by a pattern. For details, see the definition of the `trading.audit.logger.pattern` property in the "Trading Integration API" section of the document **Caplin Integration Suite for Java API Documentation**.

The following code extract shows the creation of a `TradingProvider` within a custom Trading Adapter.

Creation of a TradingProvider

```
public class MyTradingApp implements TradingApplicationListener
{
      static void main(String[] args) throws IOException, SAXException
   {
      // Create an argument array. In practice, these arguments
      // would typically be command line arguments.
      String[] args = new String[3];
      args[0] = "--config-file=conf/DataSource.xml"; // DataSource configuration
      args[1] = "--fields-file=conf/Fields.xml";     // DataSource message field defs
      args[2] = "--trading-property-file=conf/trading-provider.properties";
                                       // The properties file for the TradingProvider

      new MyTradingApp(args);
   }

   MyTradingApp(String[] arguments) throws IOException, SAXException
   {
      // Instantiate a DataSource to manage the communication with
      // other DataSource applications, such as Caplin Liberator,
      // and hence with client applications.

      DataSource dataSource = DataSourceFactory.createDataSource(arguments);

      // Instantiate a TradingProvider.
      // This allows you to use the Trading Integration API to communicate
      // with clients by means of trade messages.

      TradingProvider tradingProvider = new TradingProvider
                       (this,        // Reference to this TradingApplicationListener
                        dataSource); // The DataSource that this TradingProvider uses

      // Once the TradingProvider has been instantiated
      // the DataSource has to be started explicitly.

      dataSource.start();
   }

   // ...
}
```

The `arguments` parameter passed to the `DataSource` constructor, provides arguments that specify:

- An XML format configuration file for the `DataSource`.

  This file defines the connections that the Trading Adapter makes with the other Caplin Platform components, such as Liberator.

- An XML format file that defines the field names within the trade messages and other messages.
- The location of the properties file that configures the `TradingProvider`.

The `TradingProvider` processes the flow of a Trade by following the states defined by the Trade Model for the particular type of Trade. The Trade Models are defined in the XML configuration files referred to by the `TradingProvider`'s properties file. The Trade Model XML is discussed in Configuring Trade Models 15.

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 `TradeChannelListener` to the channel; this interface handles notifications on the channel about Trades which have been created or closed. You can add a new instance for each channel or a single global instance. You must also implement the interface.

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.setTradeChannelListener(new MyTradeChannelListener(channel));

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

6.3 New Trades

A new Trade, initiated by a client, is passed to the `TradeChannelListener` as a Trade object. The `TradeChannelListener` can then do the following:

- Carry out any initialization needed within the trading system.
- Add a `TradeListener` to the newly-created `Trade` object, either as a new instance for each Trade or as a single global instance.

The `TradeListener` handles all events for a Trade. You must implement this interface. Different trade types are normally handled by different implementations of `TradeListener`.

The following code extract shows part of a sample implementation of a `TradeChannelListener`. 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 TradeChannelListener.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 Adapter is connected, and the nature of that system's API.

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 22). Assume, for example, that 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.

In some cases, 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 store the `Trade` object in a suitable data structure (for example, a hash table). This data structure must be accessible by the code that handles events from the trading system, as this code would typically use the ID returned by 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 Adapter. A `TradeEvent` object has a type, which represents the type of the message, for example "Open", "PriceUpdate" or "Execute". It also contains a number of fields to represent all the necessary information for that message, for example "BidPrice" or "Amount".

When a message is received from the client, it is processed by the Trading Integration Library to verify whether the event is valid based on the Trade Model, and is then passed to a `TradeListener`.
The `TradeListener` is responsible for handling events relating to a Trade (`TradeEvent`s, `InvalidTransitionEvent`s and `InvalidFieldsEvent`s).

If the event is valid, the Trading Integration library creates a new `TradeEvent`, updates the `Trade` object with the data from the `TradeEvent`, and sends this to the `TradeListener` (by calling `TradeListener.receiveEvent()`). The `TradeListener` would then typically send a message on to the trading system.

If the event is not a valid transition, or the message contains invalid fields, the trade on the client may not be in the same state as the server. In this case, the Trading Integration library creates an `InvalidTransitionEvent` or `InvalidFieldsEvent` as appropriate, and sends it to the `TradeListener` (by calling `TradeListener.receiveInvalidTransitionEvent()` or `TradeListener.receiveInvalidFieldsEvent()`). The `TradeListener` should respond by sending an event that is valid for both the client and server states; typically this would be an error event that closes the trade.

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

Example implementation of TradeListener

```
private class MyTradeListener implements TradeListener
{
   public void receiveEvent(TradeEvent event) throws TradeException
   {
      // Send a message to the trading system.
   }

   public void receiveInvalidTransitionEvent(InvalidTransitionEvent event)
   {
      // Handle error in trade model state transition.
   }

   public void receiveInvalidFieldsEvent(InvalidFieldsEvent event)
   {
      // Handle invalid fields error in event.
   }
```

Events raised by the trading system can be pushed into the Trading Adapter. 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 Adapter will verify through the Trade Model that the event is allowed and that it contains all the necessary information, before sending the message to the client.

The following code extract shows the typical custom code that would be written in the Trading Adapter 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

A Trade is closed when the end-user or the trading system cancels the Trade, when the Trade is successfully executed, or when it is rejected. These final states are defined by the Trade Model: they are states that have no further transitions to another state. The `TradeChannelListener` is notified when a Trade has reached one of these final states, which allows the application to clean up any resources associated with that Trade.

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

Example implementation of TradeChannelListener.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 Adapter:

- 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 Adapter has validated a state transition in the Trade Model and has sent a `TradeEvent` to the client.

Registering the BlotterTradeListener

When implementing the `TradingApplicationListener` interface (see Initialization 19), you must add code to register and deregister a `BlotterTradeListener`, as shown in the following example.

Registering and deregistering BlotterTradeListener in TradingApplicationListener

```
public class MyTradingApp implements TradingApplicationListener
{
   ...
   private BlotterTradeListener blotterTradeListener
           = new MyBlotterTradeListener();
   ...

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

   public void blotterChannelCreated(BlotterChannel blotterChannel)
   {
      tradingProvider.addBlotterTradeListener(
                           blotterChannel,
                           blotterTradeListener);
   }

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

   public void blotterChannelClosed(BlotterChannel blotterChannel)
   {
      tradingProvider.removeBlotterTradeListener(
                           blotterChannel,
                           blotterTradeListener);
   }
```

In the previous code fragment:

- `MyBlotterTradeListener` represents an implementation of `BlotterTradeListener`. (For details, see Implementing the BlotterTradeListener interface 27).

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

```
public void blotterChannelCreated(BlotterChannel blotterChannel)
   {
      tradingProvider.addBlotterTradeListener(
                              blotterChannel,
                              blotterTradeListener);
   }
```

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

```
   public void blotterChannelClosed(BlotterChannel blotterChannel)
   {
      tradingProvider.removeBlotterTradeListener(
                              blotterChannel,
                              blotterTradeListener);
   }
```

Implementing the BlotterTradeListener interface

The `BlotterTradeListener` has one method `receiveBlotterEvent()` that is called when the Trading Adapter 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.

The following example shows the `receiveBlotterEvent()` method in a simple implementation of the `BlotterTradeListener` interface named `MyBlotterTradeListener`.

AutoBlotterTradeListener (example of BlotterTradeListener)

```
public class MyBlotterTradeListener 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 a more detailed explanation of the previous code fragment:

- Obtain the `BlotterChannel` from the `BlotterEvent` that the Trading Adapter 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 a Caplin Trader application, they may be interested in receiving Trades that they had opened in the previous session. The Trading Adapter facilitates this by allowing you to restore trade objects and their associated events, and send them to the client.

The Trading Adapter's `TradingApplicationListener` has a `channelCreated()` method (see New Trade Channels 21). In the implementation of this method, you can 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 Adapter, 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 Adapter 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 = tradingSystem.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: tradingSystem.getFields().keySet())
         {
            String fieldValue = tradingSystem.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 Adapter 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 Adapter sends the client an event containing a Trade Restoration ID. The client retains the Trade Restoration ID against the Trade. If the Trading Adapter subsequently restores the Trade, the client receives a Restored Trade event containing the matching Trade Restoration ID.

The Trade Restoration IDs are typically created and managed by the trading system (as shown in the code example in Handling Trade Restorations 29).

The following example shows typical Trading Adapter code for creating a 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);
```

6.9 Configuring Trade subjects

The Trading Adapter must be configured to recognize the subjects of Trade objects that represent:

- Messages on a Trade Channel.
- Messages on a Blotter Channel.
- Blotter Channel items.

A subject pattern is a pattern that is used to map to the subject of an incoming event to a Trade Channel, Blotter Channel, or Blotter Item, through the use of special placeholders. To define the subject patterns, the following properties must be set in the Java properties file that is supplied to the `DataSource` (see Initialization 19):

`tradechannel.patterns`

`blotterchannel.patterns`

`blotteritem.pattern`

The **`blotteritem.pattern`** property defines how the Trading Adapter must construct the subject for each Blotter Item to be sent on a Blotter Channel.

Example:

`tradechannel.patterns` is set to `/PRIVATE/%U/TRADE`

This means that any Trade object whose subject begins with `/PRIVATE/john-0/TRADE` is identified as a message for the user john's Trade Channel `/PRIVATE/john-0/TRADE`

Defining the subject pattern properties

You should define the pattern properties in a properties file as shown in this example:

Properties file containing properties for matching subject patterns

```
#  Specify the XML configuration files defining
#  the trade models to be used by the Trading Adapter.
trading.models=conf/MyESPStateModel.xml,conf/MyRFSStateModel.xml

# Define subject patterns for TradeChannels,
# BlotterChannels, and BlotterItems.
tradechannel.patterns=/PRIVATE/%U/TRADE/
blotterchannel.patterns=/PRIVATE/%U/TRADEBLOTTER/%1
blotteritem.pattern=/TRADE/%U/BLOTTER/%1
```

Supply the location of the file to the `DataSource` constructor (see Initialization 19).

Tip: For a full description of the properties, see the "Trading Integration API" section of the document **Caplin Integration Suite for Java API Documentation**.

Format of subject patterns

The pattern in a subject pattern property consists of fixed text, with optional placeholders of the form `%<some-character>`. A placeholder marks a position in the pattern where variable text in a subject can be matched or inserted.

For example, the subject `/PRIVATE/TRADE/12345` matches the pattern `/PRIVATE/TRADE/%1`, where `%1` is `12345`

If there is more than one section of variable text to be matched, use a different character in each placeholder; say `%1`, `%2`, … as in the following example:

The subject `/PRIVATE/TRADE/LON/FX/12345`, matches the pattern `/PRIVATE/TRADE/%1/FX/%2` where `%1` is `LON` and `%2` is `12345`

Special placeholders

There are two built-in placeholders with special meanings:

- `%U` (uppercase letter 'U') represents the Liberator session username of the trading user.

  For example, `john-0`
  (If a user is allowed to have multiple concurrent logins on the Liberator, each login is identified by a unique session username. For example, `john-0`, `john-1`, …)

- `%u` (lowercase letter 'u') represents the Liberator login name of the trading user.

  For example, `john`

Note: The subjects of Trade Channels, Blotter Channels, and Blotter Items must contain the Liberator session username (a requirement of the Caplin Trading Integration API). This means that the `%U` placeholder must always be present in the configured **`tradechannel.patterns`**, **`blotterchannel.patterns`**, and **`blotteritem.pattern`**, as shown in the examples in the following sections.

Also see Rules and restrictions for pattern properties 37

Configuring subject patterns for Trade Channels

`tradechannel.patterns` property

To allow the Trading Adapter to determine which Trade objects belong to which Trade Channels, you should define the **`tradechannel.patterns`** property.

A typical definition would be:

```
tradechannel.patterns=/PRIVATE/%U/TRADE
```

When the Trading Adapter receives a request for the subject `/PRIVATE/john-0/TRADE`, the subject matches the pattern, and this causes the Trading Adapter to create a Trade Channel for Trades with the subject `/PRIVATE/john-0/TRADE`

Assume the Trading Adapter subsequently receives an object representing a Trade, whose subject is `/PRIVATE/john-0/TRADE/FX`. This object also matches the Trade Channel pattern, and is therefore identified as being a Trade on the Trade Channel `/PRIVATE/john-0/TRADE/FX`

Liberator configuration

The corresponding **add-object** and **object-map** definitions for Trade Channel subjects must be set up in the Liberator configuration file (see Mapping trade messaging objects in Liberator 60).

For example:

```
add-object
      /PRIVATE/TRADE
      ...
end-object

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

Private Trade Channels

In the previous examples, the fixed prefix `/PRIVATE`, together with the Liberator session username (for example, `john-0`), allow the Liberator's Permissioning Auth Module to validate the object as belonging to the private Trade Channel of the user 'john'. This ensures that only the genuine user 'john' can create Trades on this channel, and that no other user can subscribe or contribute to it.

In practice, the prefix could be any string, but if you want to implement Trade Channels that are private to individual users (as Caplin recommends), you must configure the Permissioning Auth Module to recognize and validate objects with your chosen subject pattern as belonging to a private channel. The subject must contain the Liberator session name (`%U` in the pattern); this is a requirement of the Caplin Trading Integration API, but the Permissioning Auth Module also uses it to ensure that the channel is private to the user's session.

Multiple Trade Channels

The Trading Adapter will automatically create multiple Trade Channels for a user, providing that the **`tradechannel.patterns`** property is defined appropriately.

In the previous example, where the property is defined as the pattern `/PRIVATE/%U/TRADE`, a request for the subject `/PRIVATE/john-0/TRADE/FX` matches this pattern, and the Trading Adapter creates an FX Trade Channel to handle Trades with subjects that begin with `/PRIVATE/john-0/TRADE/FX`

A subsequent request for the subject `/PRIVATE/john-0/TRADE/FI` also matches the pattern, but then the Trading Adapter creates a separate Trade Channel for FI; this channel handles Trades with subjects that begin with `/PRIVATE/john-0/TRADE/FI`

Multiple subject patterns for Trade Channels

The **`tradechannel.patterns`** property can contain more than one pattern, separated by commas. This capability can be used to create multiple Trade Channels for a user when the subjects for the channels have incompatible naming conventions.

For example:

```
tradechannel.patterns=/PRIVATE/%U/TRADE/FX,/PRIVATE/%U/FI/TRADE
```

Here, FX Trades have subjects that start with

`/PRIVATE/<session-username>/TRADE/FX`

whereas FI Trades have subjects starting with

`/PRIVATE/<session-username>/FI/TRADE`

A request for the subject `/PRIVATE/john-0/TRADE/FX` matches the first pattern of the property, so the Trading Adapter creates an *FX* Trade Channel for the user session `john-0`. A request for the subject `/PRIVATE/john-0/FI/TRADE` matches the second pattern of the property, so the Trading Adapter creates a separate *FI* Trade Channel for `john-0`.

Configuring subject patterns for Trade Blotters

`blotterchannel.patterns` property

The `blotterchannel.patterns` property allows the Trading Adapter to determine which Trade objects are requests to open (create) Blotter Channels.

A typical definition would be:

```
blotterchannel.patterns=/PRIVATE/%U/TRADEBLOTTER/%1
```

An object whose subject is `/PRIVATE/john-0/TRADEBLOTTER/FX` matches this pattern, and therefore is identified as a request to create a Blotter Channel `/PRIVATE/john-0/TRADEBLOTTER/FX` for the user session `john-0`.

Liberator configuration

The corresponding **add-object** and **object-map** definitions for Blotter Channel subjects must be set up in the Liberator configuration file (see Mapping trade messaging objects in Liberator 60).

For example:

```
add-object
      /PRIVATE/TRADEBLOTTER
      ...
end-object

object-map /PRIVATE/TRADEBLOTTER/%1 /PRIVATE/%U/TRADEBLOTTER/%1
```

Private Blotter Channels

The fixed prefix `/PRIVATE`, together with the Liberator session user name (say `john-0`), allows the Liberator's Permissioning Auth Module to validate the object as belonging to the private Blotter Channel of the user 'john'. This ensures that only the genuine user 'john' can create this Blotter Channel and insert Blotter Items into it. No other user is allowed to subscribe or contribute to it.

See "Private Trade Channels" in Configuring subject patterns for Trade Channels 32 for information on using prefixes other than `/PRIVATE`.

`blotteritem.pattern` property

When the Trading Adapter constructs a Blotter Item on completion of a Trade, it gives the item a subject. The **`blotteritem.pattern`** property defines how the Trading Adapter must construct this subject.

Assume the Blotter Channel is `/PRIVATE/john-0/TRADEBLOTTER/FX`,
based on the **`blotterchannel.patterns`** setting `/PRIVATE/%U/TRADEBLOTTER/%1`

Then for this channel:

```
    %U = john-0
    %1 = FX
```

Assume **`blotteritem.pattern`** is defined as `/PRIVATE/TRADE/%U/BLOTTER/%1`

When the Trading Adapter constructs a Blotter Item for an FX Trade, it obtains the placeholder values (`%U`, `%1`) for the item's subject from the corresponding placeholder settings in the FX blotter channel, and appends a trade-id to the subject. The subject of the generated Blotter Item is therefore:

```
/PRIVATE/TRADE/john-0/BLOTTER/FX/<trade-id>
```

For example:

```
/PRIVATE/TRADE/john-0/BLOTTER/FX/123456
```

Also see Rules and restrictions for pattern properties 37.

Multiple Blotter Channels

The Trading Adapter will automatically create multiple blotter channels for a user, providing that the `blotterchannel.patterns` property is defined appropriately.

The following example shows how the patterns can be defined to support a Blotter Channel for FX Trades, and a separate Blotter Channel for FI Trades.

`blotterchannel.patterns` is:	`/PRIVATE/%U/TRADEBLOTTER/%1`
`blotteritem.pattern` is:	`/PRIVATE/TRADE/%U/BLOTTER/%1`

1) Blotter Channel for FX Trades:

An object whose subject is `/PRIVATE/john-0/TRADEBLOTTER/FX` matches **`blotterchannel.patterns`** so the Trading Adapter creates an FX Blotter Channel with this subject.

For this channel:

```
    %U = john-0
    %1 = FX
```

The **`blotteritem.pattern`** setting causes a Blotter Item that is sent on this channel to have a subject of the form:

```
/PRIVATE/TRADE/john-0/BLOTTER/FX/<trade-id>
```

The Trading Adapter constructs this subject by substituting the values of the `%U` and `%1` placeholders from the Blotter Channel subject into the equivalent placeholders in the **`blotteritem.pattern`**.

2) Blotter Channel for FI Trades:

An object whose subject is `/PRIVATE/john-0/TRADEBLOTTER/FI` also matches **`blotterchannel.patterns`**, so the Trading Adapter creates an FI Blotter Channel with this subject.

For this channel:

```
    %U = john-0
    %1 = FI
```

The `blotteritem.pattern` setting causes a Blotter Item that is sent on this channel to have a subject of the form:

```
/PRIVATE/TRADE/john-0/BLOTTER/FI/<trade-id>
```

Note: Make sure that the prefix to the subject of a Blotter Item (the part before the `<trade-id>`) is different for each Blotter Channel. If the Liberator needs to clear down a particular blotter channel for recovery purposes, this ensures that it can do so without affecting other blotter channels for the same user.

Multiple subject patterns for Blotter Channels

The **`blotterchannel.patterns`** property can contain more than one pattern, separated by commas. This capability can be used to create multiple Blotter Channels for a user, when the subjects for the channels have incompatible naming conventions.

For example:

```
blotterchannel.patterns=/PRIVATE/%U/TRADEBLOTTER/%1/,/PRIVATE/%U/%1/TRADEBLOTTER
```

Here, Blotter Channels for FX Trades have subjects that start with

`/PRIVATE/<session-username>/`**`TRADEBLOTTER/FX`**

whereas Blotter Channels for FI Trades have subjects starting with

`/PRIVATE/<session-username>/`**`FI/TRADEBLOTTER`**

A request for the subject `/PRIVATE/john-0/TRADEBLOTTER/FX` matches the first pattern of the property, so the Trading Adapter creates an FX Blotter Channel for the user session `john-0`.
A request for the subject `/PRIVATE/john-0/FI/TRADEBLOTTER` matches the second pattern of the property, so the Trading Adapter creates a separate FI Blotter Channel for `john-0`.

Sending items on the same Blotter Channel from multiple Trading Adapters

If you are using multiple Trading Adapters you may need them to share a Blotter Channel, so that in the client, all Blotter Items of a particular type are shown in the same blotter, regardless of which Trading Adapters originated those items.

Each Trading Adapter must define the same **`blotterchannel.patterns`** property.

For example:

```
blotterchannel.patterns=/PRIVATE/%U/TRADEBLOTTER/%1
```

The subject of every Blotter Item must be stamped with the name referring to the Trading Adapter that generated it. To do this, include the name in the **`blotteritem.pattern`** property of each relevant Trading Adapter, as in this example:

```
blotteritem.pattern=/PRIVATE/TRADE/%U/BLOTTER/<trading-adapter-name>/%1
```

Hence:

Trading Adapter “TA-1”: `blotteritem.pattern`

```
blotteritem.pattern=/PRIVATE/TRADE/%U/BLOTTER/TA-1/%1
```

Trading Adapter "TA-2": `blotteritem.pattern`

```
blotteritem.pattern=/PRIVATE/TRADE/%U/BLOTTER/TA-2/%1
```

By stamping the Blotter Items with the identification of the originating Trading Adapter, the relevant items in the Blotter can be recovered if one of the Trading Adapters fails and is restarted. The Liberator maintains a container representing the Blotter Channel. When the failed Trading Adapter restarts, the Liberator clears down the container's Blotter Items that originated from this Trading Adapter, to ensure the consistency and completeness of the items (an item may be lost because it was being transmitted as the connection went down). The identification stamp allows the Liberator to determine which items to clear down, leaving Blotter Items that originated from the other Trading Adapters untouched.

Rules and restrictions for pattern properties

This section describes some rules and restrictions that apply when specifying pattern properties for configuring Trade subjects.

Trade Blotter subject patterns

- **`blotteritem.patterns` can only contain *one* subject pattern.**

  (**`blotterchannel.patterns`** can contain more than one subject pattern – see "Multiple subject patterns for Blotter Channels" in Configuring subject patterns for Trade Blotters 34)

- **The patterns in `blotterchannel.patterns` must contain a `%U` placeholder**.

  See the Note in Format of subject patterns 32.

- **The placeholders in `blotteritem.pattern` must be present in every subject pattern in `blotterchannel.patterns`.**

  For example:

  `blotteritem.pattern`:`

  `/PRIVATE/TRADE/%U/BLOTTER/%1/%2`

  `blotterchannel.patterns`:

  `/PRIVATE/%U/%1/TRADEBLOTTER/FX/%2,/PRIVATE/%U/%1/TRADEBLOTTER/FI/%2`

Rules applying to all subject patterns

- **You cannot have two adjacent placeholders in a pattern.**

  Multiple placeholders in a pattern must be separated by a literal string value.

  Examples:

  The pattern `/PRIVATE/TRADE/%U/%1` is permitted, because a forward slash ('`/`') separates the `%U` placeholder from the `%1` placeholder.

  The pattern `/PRIVATE/TRADE/%U%1` is not permitted, because the Trading Adapter would not be able to determine where in the received subject the value for `%U` ends and the value for `%1` begins.

- **Be careful with placeholders at the end of patterns.**

  If a pattern ends with a placeholder, the subject being matched cannot have any content after the placeholder value.

  Example:

  Pattern: `/PRIVATE/TRADE/FX/%U`

  Subject: `/PRIVATE/TRADE/FX/john-0/SPOT`

  The subject matches the pattern, but because the pattern ends with a parameter (`%U`), the Liberator session username is incorrectly interpreted as `john-0/SPOT`

  Note that a subject *can* have trailing content if the end of the pattern is a literal string.

  Example:

  Pattern: `/PRIVATE/TRADE/%U/FX`

  Subject: `/PRIVATE/TRADE/john-0/FX/SPOT`

  The subject correctly matches the pattern because the pattern ends with the literal string `/FX`

- **Placeholder values in a pattern must not clash with literal strings in the pattern.**

  The values extracted for a placeholder in a subject must not clash with the literal string that immediately follows it in the pattern.

  Example:

  Pattern: `/PRIVATE/TRADE/%1/%U`

  Subject: `/PRIVATE/TRADE/FX/SPOT/john-0`

  The matching of placeholder values is based upon the boundaries defined by the literal string values in the pattern, so in the above subject:

  – `/PRIVATE/TRADE/` matches the pattern `/PRIVATE/TRADE/`

  – `FX/` matches the pattern `%1/`, so `%1` is extracted as `FX` (correct).

  – `SPOT/john-0` matches `%U`, so `%U` is extracted as `SPOT/john-0` (incorrect).

  The problem here is that the forward slash in the value for `%1` (the forward slash in `FX/`) matches the forward slash between the `%1` and the `%U` in the pattern.

  This problem can be avoided for the pattern in this example if the client application provides the value `FX-SPOT` rather than `FX/SPOT` for the `%1` parameter, so the subject is now:

  `/PRIVATE/TRADE/FX-SPOT/john-0`

  – `/PRIVATE/TRADE/` matches the pattern `/PRIVATE/TRADE/`

  – `FX-SPOT/` matches the pattern `%1/`, as before, so `%1` is extracted as `FX-SPOT` (correct).

  – `john-0` matches `%U`, so `%U` is now extracted as `john-0` (correct).

- **Values of placeholders cannot be empty strings.**

  A subject is not matched if the value extracted for one of the pattern's placeholders is an empty (zero-length) string.

  Example:

  Pattern: `/PRIVATE/TRADE/%U/%1`

  Subject: `/PRIVATE/TRADE//FX`

  The subject does not match the pattern, because the `%U` value in the subject is an empty (zero-length) string.

  Similarly, the subject `/PRIVATE/TRADE/john-0/` does not match the above pattern, because the `%1` value in the subject is an empty string.

Tip: The most likely reason for problems with empty placeholders is incorrect configuration, probably in a component other than the Trading Adapter (such as Liberator or Transformer)

6.10 Using the Code Generator

A Code Generator is supplied with the Caplin Integration Suite that helps you to produce correct Java Trading Adapter code more quickly. It parses the XML defining one or more Trade Models, and uses this information to generate:

- A `TradeListener` interface for each Trade Model, and an associated adapter class.
- Implementations of the `TradeEvent` interface.

  A `TradeEvent` implementation is generated for each `<transition>` in the Trade Model that has message fields defined in `<field>` tags (see ESP Example with field definitions 17). Each implementation contains getter and setter methods for the fields defined in the `<transition>` XML tag.

Using the Code Generator, together with Trade Model XML configuration that defines the message fields relevant to each trade model state transition, gives you the following benefits:

- Skeleton code is generated to handle all the state changes in a Trade Model.
- You do not have to manipulate your own dictionary of fields; your application always has access to the correct message field names, provided you define them correctly in the Trade Model XML.
- Field names are accessed through generated getters and setters, avoiding errors resulting from typing field name strings incorrectly in code.

How to run the Code Generator

- To run the Code Generator, type the following in a command window:

```
java -cp <code-generator-jar> com.caplin.trading.CodeGenerator
<src> <package> <templates-dir> <model-file> [<model-name>]
```

where:

`<code-generator-jar>` is the location and name of the jar file containing the Code Generator. The name of this jar starts with `trading-datasource-code-generator`, and the file is located in the *tools* directory of the CIS kit.

`<src>` is the root location where the generated source should be saved. If this is a relative path, it is relative to the directory from where the command is run.

`<package>` is the name of the java package to be used for the generated java files.

`<templates-dir>` is the directory containing the code templates* that the Code Generator uses. If this is a relative path, it is relative to the directory from where the command is run.

These templates are normally located in the *templates/java* directory relative to the directory where the Caplin Integration kit was unpacked.

`<model-file>` is the location of the Trade Model XML file containing the definition of one or more Trade Models. If this is a relative path, it is relative to the directory from where the command is run.

`<model-name>` is the name of the Trade Model in the `<model-file>` for which its code is to be generated. This command parameter is optional; if not supplied, code is generated for all the Trade Models within the `<model-file>`.

Example Code Generator command:

The following example assumes the Code Generator is run from the directory where the Caplin Integration Suite kit was unpacked.

```
java -cp <code-generator-jar> com.caplin.trading.CodeGenerator
"src" "com.novobank.trading.adapter" "templates/java"
"models/NovoBankTradeModels.xml" "ESP"
```

This command would generate in the directory *src/com/novobank/trading/adapter* directory the Java source files for the ESP trade model defined in *models/NovoBankTradeModels.xml*.

Note: Each time you run the Code Generator with the same `<src>` directory, it overwrites any previously generated Java source files with new versions.

*The Code Generator uses the open-source Apache Velocity template engine.
For more about this, see http://velocity.apache.org

What code is generated?

The Code Generator takes the Trade Model XML as its source data, and for each Trade Model defined in the XML file, it generates the following source files:

*<**ModelName**>TradeListener.java*	A model specific `TradeListener` interface called `<ModelName>TradeListener`
*<**ModelName**>TradeListenerAdapter.java*	An associated adapter class called `<ModelName>TradeListener`
Source files called *<**TransitionTriggerName**>TradeEvent.java*	For each `<transition>` tag in the Trade Model XML that has `<field>` tags, the Code Generator produces a class that implements the `TradeEvent` interface. The class contains `get()` and `set()` methods for the fields defined in the transition. There is one source file per class.

For example, if you run the Code Generator on the ESP trade model shown in ESP Example with field definitions 17, it generates the following files:

***ESP**TradeListener.java*	`TradeListener` interface called **`ESPTradeListener`**
***ESP**TradeListenerAdapter.java*	`TradeListener` adapter called **`ESPTradeListenerAdapter`**
***Open**TradeEvent.java*	**`OpenTradeEvent`** interface: the `TradeEvent` interface for transition trigger **`"Open"`**
***OpenAck**TradeEvent.java*	**`OpenAckTradeEvent`** interface the `TradeEvent` interface for transition trigger **`"OpenAck"`**
***TradeConfirmation**TradeEvent.java*	**`TradeConfirmationTradeEvent`** interface: the `TradeEvent` interface for transition trigger **`"TradeConfirmation"`**

Using the TradeListener classes

Given the 'ESP' trade model 17 referred to above, you would write an implementation of the generated `TradeListener` interface, `ESPTradeListener`, calling it, say, `MyESPTradeListener`. You then use `MyESPTradeListener` and the generated `ESPTradeListenerAdapter` class in your implementation of `TradingApplicationListener`, like this:

```
public void tradeCreated(Trade trade) throws TradeException
{
   MyESPTradeListener listener = new MyESPTradeListener();
   ESPTradeListenerAdapter adapterListener =
      new ESPTradeListenerAdapter(listener);

   trade.setTradeListener(adapterListener);
}
```

Each `<Model>TradeListener` interface contains an `on<TriggerName>()` method for each client transition (that is, for each `<transition>` tag in the Trade Model XML that has the attribute `source="client"`). For example, the generated `ESPTradeListener` interface would contain the method `onOpen()`:

```
public interface ESPTradeListener
{
   public void onOpen(OpenTradeEvent ev);

   public void receiveInvalidTransitionEvent(InvalidTransitionEvent event);

   public void receiveInvalidFieldsEvent( InvalidFieldsEvent event );

   ...
}
```

(The `receiveInvalidTransitionEvent()` and `receiveInvalidFieldsEvent()` method declarations are always generated.)

The line of the trade Model XML that caused the `onOpen()` code to be generated is:

```
      <transition target="OpenSent" trigger="Open" source="client">
```

The trigger name (`trigger="Open"`) is used to define both the name of the method, `onOpen()`, and the name of the event class passed as a parameter, `OpenTradeEvent`. Your implementation of `ESPTradeListener` would contain implementations of `onOpen()`, `receiveInvalidTransitionEvent()`, and `receiveInvalidFieldsEvent()`

If there are no fields defined in the XML for a particular client transition trigger, the `<ModelName>TradeListener()` method generated for that trigger just uses a `TradeEvent` parameter:

```
public interface XYZTradeListener
{
   public void onOpen(OpenTradeEvent ev);

   public void onPass(TradeEvent ev); // No fields are defined in
                                      // the XML for the "Pass" trigger

   public void receiveInvalidTransitionEvent(InvalidTransitionEvent event);

   public void receiveInvalidFieldsEvent(InvalidFieldsEvent event);

   ...
}
```

The Trade Model's TradeListenerAdapter

The generated `<ModelName>TradeListenerAdapter` class is a `TradeListener` that wraps the implementation of the `<ModelName>TradeListener` interface.

Here is an example showing the code that would be generated for the `ESPTradeListenerAdapter`:

```
public class ESPTradeListenerAdapter implements TradeListener
{
   private ESPTradeListener listener;

   public ESPTradeListenerAdapter(ESPTradeListener listener)
   {
      this.listener = listener;
   }

   @Override
   public void receiveEvent(TradeEvent ev) throws TradeException
   {
      if ( ev.getType().equals("Open") )
      {
         listener.onOpen(new OpenTradeEvent(ev));
      }
   }
   ...
}
```

In this code, the `if` statement has been generated to handle the ESP Trade Model's 17 client transition trigger `"Open"`:

```
      <transition target="OpenSent" trigger="Open" source="client">
```

When the received client event is for the transition `"Open"`, the generated code creates an instance of the corresponding `TradeEvent` (`OpenTradeEvent`), and passes it to the `onOpen()` method of your `ESPTradeListener` implementation.

The adapter provides you with the convenience of not having to implement for yourself the `TradeListener.receiveEvent()` code to identify which client transition trigger is in the received `TradeEvent`; in particular, it provides all the string comparison code needed to do this.

The <TransitionTriggerName>TradeEvents

For each `<transition>` in the Trade Model XML that contains `<field>` tags, the Code Generator creates a class called `<TransitionTriggerName>TradeEvent` that implements the `TradeEvent` interface. The name of the class is derived from the `trigger` attribute of the `<transition>` tag. For example, the XML transition `<transition target="OpenSent" trigger="`**`Open`**`" ...>` generates a class called **`Open`**`TradeEvent`.

The generated class contains a `get()` and `set()` method for each field. The corresponding `on<TransitionTriggerName>()` method in your implementation of `<ModelName>TradeListener` is passed a `<TransitionTriggerName>TradeEvent` instance.

Additionally the `<TransitionTriggerName>TradeEvent` constructor that takes a `Trade` parameter sets any default field values that are defined in the transition's `<field>` tags; these defaults are used when processing events generated from the server.

The following example shows the generated `OpenTradeEvent` class relating to the ESP trade model 17. The relevant initial line of XML in the Trade Model is:

```
<transition target="OpenSent" trigger="Open" source="client">
```

Generated OpenTradeEvent class

```
public class OpenTradeEvent implements TradeEvent
{
   private TradeEvent backingEvent;

   public OpenTradeEvent(TradeEvent tradeEvent) throws TradeException
   {
      backingEvent = tradeEvent;
   }

   public OpenTradeEvent(Trade trade) throws TradeException
   {
      backingEvent = trade.createEvent("Open");
      setupDefaultFields()
   }

   ...

   private void setupDefaultFields()
   {
   }

   /**
    * Get ESP model name
    */
   public String getStateModelName()
   {
      return backingEvent.getField("StateModelName");
   }

   /**
    * Set ESP model name
    */
   public void setStateModelName(String value)
   {
      backingEvent.addField("StateModelName",value);
   }
   /**
    * Get Request ID
    */
   public String getRequestID()
   {
      return backingEvent.getField("RequestID");
   }

   /**
    * Set Request ID
    */
   public void setRequestID(String value)
   {
      backingEvent.addField("RequestID",value);
   }
   ...
   // And similar for fields QuoteID, InstrumentName, DealtCurrency,
   // BuySell, and Price.
}
```

The Code Generator obtains the comment describing each `get()` and `set()` method from the description attribute of the corresponding `<field>` tag in the Trade Model XML.

The next example shows the generated `OpenAckTradeEvent` class relating to the ESP trade model 17. The relevant lines of XML in the Trade Model are:

```
<transition target="Opened" trigger="OpenAck" source="server">
  <field name="RequestID" description="Request ID" required="true" />
  <field name="StateModelName" description="ESP model name" required="true"
         default="ESP" />
  <field name="BuySell" description="BuySell indicator" required="true"
         default="BUY" />
</transition>
```

In this `<transition>`, default values are defined for the fields `StateModelName` and `BuySell`, so the generated class sets up these defaults in the (server originated) message:

Generated OpenAckTradeEvent class

```
public class OpenAckTradeEvent implements TradeEvent
{
   private TradeEvent backingEvent;

   public OpenAckTradeEvent(TradeEvent tradeEvent) throws TradeException
   {
      backingEvent = tradeEvent;
   }

   public OpenAckTradeEvent(Trade trade) throws TradeException
   {
      backingEvent = trade.createEvent("OpenAck");
      setupDefaultFields()
   }

   ...

   private void setupDefaultFields()
   {
      this.setStateModelName("ESP");
      this.setBuySell("BUY");
   }

   /**
    * Get Request ID
    */
   public String getRequestID()
   {
      return backingEvent.getField("RequestID");
   }

   /**
    * Set Request ID
    */
   public void setRequestID(String value)
   {
      backingEvent.addField("RequestID",value);
   }

   /**
    * Get ESP model name
    */
   public String getStateModelName()
   {
      return backingEvent.getField("StateModelName");
   }

   /**
    * Set ESP model name
    */
   public void setStateModelName(String value)
   {
      backingEvent.addField("StateModelName",value);
   }

   /**
    * Get BuySell indicator
    */
   public String getBuySell()
   {
      return backingEvent.getField("BuySell");
   }

   /**
    * Set BuySell indicator
    */
   public void setBuySell(String value)
   {
      backingEvent.addField("BuySell",value);
   }
}
```

How to ensure the generated code is valid Java

Use Java names in the Trade Model XML

To ensure the generated code is valid Java the names used for models, states, transitions, and fields in the Trade Model XML must be valid Java names. In particular:

- Java object names must only contain valid characters. In particular, do not define names that contain spaces.
- The first character of each word making up the name should start with a capital letter, and the rest of the characters in the name should be lower case

Examples:

✓ `DealtCurrency`

✗ `dealtCurrency`

✗ `DEALTCURRENCY`

✗ `dealtcurrency`

✗ `dealt currency`

Defining fields in the Trade Model XML

In the Trade Model XML the same trigger can apply to more than one state transition; that is, two or more `<transition>` tags can have the same `trigger` attribute. This typically happens when the Trade Model uses common error and timeout triggers.

If you need to define fields related to a set of such transitions, you must only put the `<field>` tags in *one* of the `<transition>` tags. (When the Code Generator examines the Trade Model, it raises an XML parsing exception if this rule is violated.)

This rule means that all transitions in the Trade Model that have the same trigger name must use the same set of fields. If you want two or more such transitions to have differing sets of related fields, you must define the `<transition>` tags to have different trigger names.

7 Using the Trading Integration API for C++

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

Note: At the time of publication of this document, the Trading Integration API for C++ was still released under its old name of Trading DataSource C++ API.

To implement a Trading Adapter 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.

7.1 Initialization

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

The `TradingDataSource` uses a `DataSource` object from the underlying DataSource for C library, which 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 Adapter.
   // 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 Adapter application, passing it the Trading Adapter
   // object and the logger (so we can log application errors to the same place as
   // errors raised in the Trading Adapter API).
   MyTradingApp myTradingApp(myTradingDataSource, myTradingDataSource.getLogger());

   // Connect to DataSource applications (Liberator and/or Transformer)
   // 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 Adapter 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 15 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 application. This file defines the connections that the application makes with the other Caplin Platform components, such as Liberator.

Logging

By default the Trading Integration API for C++ uses the logging facilities that are provided with the DataSource for C library underlying the C++ 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` class 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 Integration API.

Note: If you implement a custom logger, only the Trading Integration library and the Trading Adapter code that you write 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 Adapter 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;
};
```

7.2 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 Adapter 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 53). 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.

7.3 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());
}
```

7.4 Dealing with Events

The `TradeEvent` object represents a Trade Event, which typically encapsulates a message between the client and the Trading Adapter. 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 `TradeEvent`s; 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 Adapter 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 Adapter. 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 Adapter 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 Adapter 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);
```

7.5 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
  // ...
}
```

7.6 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
  // ...

}
```

7.7 Handling Blotter Channels

To handle Blotter Channels in the Trading Adapter:

- 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 Adapter 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 49), add code to register and deregister a `BlotterListener`.

- Code the `blotterChannelCreated()` method to register the `BlotterListener` with the Trading Adapter 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 Adapter 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 Adapter 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);
```

7.8 Handling Trade Restorations

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

The Trading Adapter's `TradingApplicationListener` has a `channelCreated()` method (see New Trade Channels 52). 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 Adapter, 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 Adapter 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 = tradingSystem->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 Adapter 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 Adapter must send the client an event containing a Trade Restoration ID. The client retains the Trade Restoration ID against the Trade. If the Trading Adapter 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 57).

The following example shows typical Trading Adapter 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);
```

8 Configuring Caplin Liberator for trading

Caplin Liberator is an integral part of the Caplin Platform (see the diagram in the Overview 6) and therefore must be correctly configured to support trading activity. When you use the Caplin Integration Adapter Blade project wizard to create a Trading Adapter blade, it automatically creates the configuration that enables Liberator to trade. This configuration is applied when you deploy the blade.

For more information about creating Java-based platform blades, such as Trading Adapter blades, see the document **Caplin Integration Suite: How To Create A Platform Java Blade**.

Note: If you do not intend to implement your Trading Adapter as a Caplin Platform Blade, you must manually configure the Liberator as described in the rest of this section and its subsections.

If you are implementing your Trading Adapter as a Caplin Platform Blade, you do not need to read the following material.

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 60.
- Routing of trade messages 62 to the correct Trading Adapter.
- Trading performance and integrity 63.
- Security 65.

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

Configuration item	Parameter	See section
add-object	*name*	Mapping trade messaging objects in Liberator 60
add-object	*throttle-times*	Throttling 63
add-object	*discard-timeout*	Throttling 63
add-object	*no-batching*	Throttling 63
add-peer	–	Routing trade messages to the Trading Adapter 62
add-data-service	–	Routing trade messages to the Trading Adapter 62
object-map	–	Mapping trade messaging objects in Liberator 60
output-queue-size	–	Optimizing client reconnection time 63
session-id-len	–	Session IDs 65

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 the Caplin Platform. For more information about configuring Liberator, see the **Caplin Liberator Administration Guide**.

8.1 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, the base subject name could be `/PRIVATE/TRADE`, so that all trade messages sent between client and Liberator have a subject name that starts with /PRIVATE/TRADE.

```
add-object
        name    /PRIVATE/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 **Caplin Liberator Administration Guide**, and the section on Throttling 63.

A client opens a Trade Channel by subscribing to one of the built-in Liberator trade message objects, for example, `/PRIVATE/TRADE`. Because Liberator and the Trading Adapter 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 /PRIVATE/TRADE/%1 /PRIVATE/%U/TRADE/%1
```

`%U` is the Liberator session username of the trading user.

`%1` is a placeholder representing any variable length string appearing in the subject name of the subscription.

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

    /PRIVATE/TRADE/FX

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

    /PRIVATE/john-0/TRADE/FX

where **`john-0`** is the Liberator session username assigned to this Liberator login of 'john'.

This transformation allows the Trading Adapter to distinguish between trade messages from 'john' and trade messages relating to other end-users. The Trade Channel for 'john' is uniquely defined by the first part of the message subject: `/PRIVATE/john-0/TRADE/FX`

Similarly, when Liberator *receives* a trade message with the subject
`/PRIVATE/john-0/TRADE/FX`
from the Trading Adapter, 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
`/PRIVATE/TRADE/FX`

Note: If you are implementing a Java-based Trading Adapter, the trade messaging subject names and object maps defined in Liberator must be compatible with the subject patterns configured in the Trading Adapter. See Configuring Trade subjects 31.

8.2 Routing trade messages to the Trading Adapter

The Liberator configuration defines the DataSource applications (also called “peers”) to which Liberator is connected, and the **data services** that Liberator provides to subscribing client applications. In the Caplin Platform, this configuration must include the Trading Adapters and their associated data services.

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

The following example configuration for Liberator defines connections to two Trading Adapters, one for FX trading and one for FI trading:

Example of Trading Adapter configuration in Liberator

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

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

...

add-data-service
   service-name        fx-trade-data
   include-pattern     ^/PRIVATE/.*/TRADE/FX

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

add-data-service
   service-name        fi-trade-data
   include-pattern     ^/PRIVATE/.*/TRADE/FI

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

The **add-peer** configuration items define connections to two Trading Adapters, one for FX Trading (`fxtradeadapter`) and one for FI Trading (`fitradeadapter`).

There is a data service for each of these Trading Adapters, 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 `/PRIVATE/<some-string>/TRADE/FX` are directed to the FX Trading Adapter. Similarly the FI Service definition (`service-name fi-trade-data`) directs to the FI Trading Adapter any trade message subscription whose subject begins with `/PRIVATE/<some-string>/TRADE/FI`

Note that both the *include-pattern*s contain the base subject name for trade messages as defined in an **add-object** configuration item; see Mapping trade messaging objects in Liberator 60. `<some-string>` would typically be the Liberator session username of the trading user.

8.3 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 66. 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:

```
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    /PRIVATE/TRADE
        type    20
        throttle-times 0
        discard-timeout 0
        no-batching
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 `/PRIVATE/TRADE`) has different parameter settings:

- *throttle-times 0* ensures that the objects used for trade messaging (`/PRIVATE/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 `/PRIVATE/TRADE`).
- *no-batching* ensures that trade messages are never batched together (it negates the settings of the global configuration item **burst-max**).

8.4 Security

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

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 Platform 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 end-users of a Caplin Trader application can access Liberator in a secure manner through the single sign-on. For more information, see the **Caplin KeyMaster Overview**.

9 Appendix: Overview of throttling

This appendix provides an explanation of Throttling, to clarify the information given in the Throttling 63 section of Trading performance and integrity 63 ,

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

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 illustrates 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 Pricing 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 63.

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

10 Glossary of terms and acronyms

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

Term	Definition
Alerts Integration API	An integration API that allows you to send alert notifications into the **Caplin Platform** from other systems. The Alerts Integration API is part of the **Caplin Integration Suite**.
App	An application that runs in a web browser or on a mobile device.
Blade	A re-usable software module containing the code and resources needed to implement a business feature.
Blade toolkit	A set of commands to create, build and export **Caplin Platform Blades**.
Caplin Integration Suite (CIS)	A set of APIs and tools for creating adapters that integrate the Caplin Platform with external systems. It includes: ◆ a **Pricing Integration API** ◆ a **Trading Integration API** ◆ a **Permissioning Integration API** ◆ an **Alerts Integration API** ◆ the **Blade toolkit**
Caplin Liberator	A financial internet hub that delivers data and messages in real time to and from subscribers over any network.
Caplin Platform	An integrated suite of software that supports the services and distribution capabilities needed for web trading. It consists of **Caplin Liberator**, Caplin Transformer, Caplin KeyMaster, Caplin Director, and Caplin Management Console. For more information, see the **Caplin Platform Overview**.
Caplin Platform Blade	A **Blade** designed for use with the Caplin Platform. A Caplin Platform Blade can be an Adapter Blade, Config Blade, or Service Blade.
Caplin Trader	A complete development suite for creating HTML5 trading **apps**.
Caplin Trader application	A **client application** that has been built using **Caplin Trader**.
Client application	A browser-based or desktop application that communicates with the **Caplin Platform** (**Caplin Liberator**) through the **StreamLink API**.
Blotter	A record of the details of Trades made by an end-user of a **Client application**. 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 9.
Data service	A set of rules used by the **Caplin Platform** to determine how a requested data item should be sourced based on its subject. A data service can incorporate priority, failover and load balancing. For more information see the **Caplin DataSource Overview**.

Term	Definition
DataSource	DataSource is the messaging infrastructure used by the **Caplin Platform** and **Integration Adapters**. In some older documents, DataSource is also used as a synonym (but *non-preferred term*) for **DataSource application**.
DataSource API	An API that allows server applications (including **Integration Adapters**) to communicate with the **Caplin Platform**.
DataSource application	An application that uses the **DataSource API**. **Caplin Liberator**, **Caplin Transformer**, and **Integration Adapters** are all DataSource applications. In some older documents, DataSource applications are called **DataSource peers**.
ESP	Executable Streaming Price **Trade Model**.
Integration Adapter	A server application that allows an external system to communicate with the **Caplin Platform**. An Integration Adapter is a **DataSource application** and is created using the **Caplin Integration Suite**.
Liberator	Short for **Caplin Liberator**.
Permissioning	The process of determining the access rights that an end-user has to resources, such as data and functionality. Also known as **authorization**.
Permissioning Adapter	An **Integration Adapter** that uses the **Permissioning Integration API** to integrate the **Caplin Platform** with a **permissioning system** .
Permissioning Integration API	An API for sending permissioning information to the **Caplin Platform**. The Permissioning Integration API is part of the **Caplin Integration Suite**.
Permissioning system	An external source of permissioning information.
Pricing Adapter	An **Integration Adapter** that uses the **Pricing Integration API** to integrate the **Caplin Platform** with a **pricing system**.
Pricing Integration API	An API for sending pricing information to the **Caplin Platform**. The Pricing Integration API is part of the **Caplin Integration Suite**.
Pricing system	An external source of financial price data.
RFQ	Request for Quote **Trade Model**.
StreamLink API	An API that allows a **client application** to communicate with a **Caplin Liberator**. There are StreamLink APIs for various technologies; for example, Java, JavaScript, .NET and Silverlight applications, and Objective-C running on iOS.
StreamLink JS	The **StreamLink API** for JavaScript.
Throttling	A **Caplin Liberator** feature for reducing the performance impact of high update rates for data streamed to clients. See Appendix: Overview of throttling 66.
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 8.

Term	Definition
Trade Channel	A single end-user's communication between the client application and a**Trading Adapter**. See Trade Channels 8.
Trade Event	An action by a client or an event from a **trading system**. See Trade Events 9.
Trade Model	The definition of a particular trading workflow. It specifies all the states that a trade can be in, and the transitions between those states. See Trade Models 8.
Trading Adapter	An **Integration Adapter** that uses the **Trading Integration API** to integrate the **Caplin Platform** with a **trading system**.
Trading DataSource API	An older term for the **Trading Integration API**.
Trading Integration API	An API for creating an **Integration Adapter** for trade messages that intelligently manages the **Trade model** for each trade. The Trading Integration API is part of the **Caplin Integration Suite**. Formerly known as the **Trading DataSource API**.
Trading system	A system, external to the **Caplin Platform**, that supports trading of financial instruments.

CAPLIN
THE WEB TRADING TECHNOLOGY COMPANY

Contact Us

Caplin Systems Ltd

Cutlers Court

115 Houndsditch

London EC3A 7BR

Telephone: +44 20 7826 9600

www.caplin.com