CAPLIN

StreamLink 6.0

Overview

February 2013

Contents

1 Preface

1.1 What this document contains

This document gives a technical overview of Caplin StreamLink version 6.0.

It aims to provide an understanding of:

- What StreamLink is and how it can be used.
- Fundamental StreamLink concepts & features.
- The relationship between StreamLink and the Caplin Platform.
- How StreamLink can integrate with your own client application software to provide streaming data display and support online trading of financial instruments.

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 anyone who requires an introduction to Caplin StreamLink.

Typical readers include:

- Technical Managers
- System Architects
- System Administrators
- Software Developers

1.3 Related documents

- **Caplin Platform Overview**

  Provides a technical overview of the Caplin Platform, including an explanation of its architecture.
- **StreamLink JS API Documentation**

  The reference documentation for the JavaScript implementation of StreamLink.
- **StreamLink Java API documentation**

  The reference documentation for the JavaScript implementation of StreamLink.
- **StreamLink Android API documentation**

  The reference documentation for the Android implementation of StreamLink.
- **StreamLink.NET API Documentation**

  The reference documentation for the .NET implementation of StreamLink.
- **StreamLink Silverlight API Documentation**

  The reference documentation for the Silverlight implementation of StreamLink.
- **StreamLink iOS API Documentation**

  The reference documentation for the iOS implementation of StreamLink.
- **Caplin DataSource Overview**

  A technical overview of Caplin DataSource.
- **Caplin Liberator Administration Guide**

  Describes Caplin Liberator 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.

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® Reader 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.

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

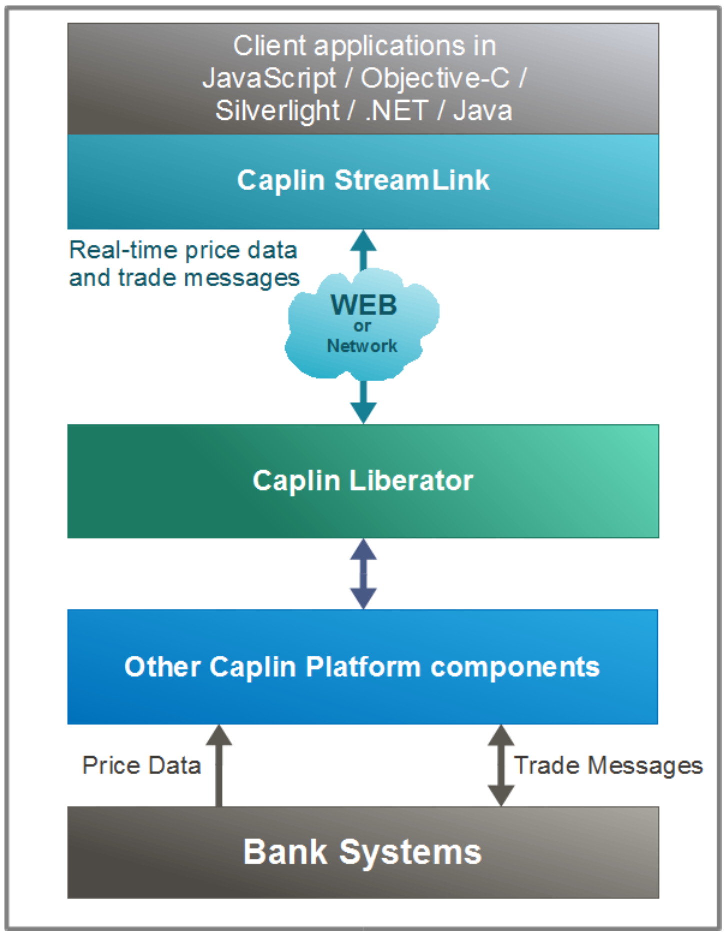
Java and *JavaScript* are trademarks or registered trademarks of Oracle® Corporation in the U.S. and other countries.

Objective-C is a trademark of Apple Inc., registered in the U.S. and other countries.

Android is a trademark of Google Inc.

2 What is Caplin StreamLink?

StreamLink is an API and underlying code library that allows client applications to exchange real-time financial data and trade messages with the **Caplin Platform**. This is shown in the following diagram.


Client applications in JavaScript / Objective-C / Silverlight / .NET / Java
Caplin StreamLink
Real-time price data and trade messages
WEB or Network
Caplin Liberator
Other Caplin Platform components
Price Data
Trade Messages
Bank Systems

StreamLink exchanging financial data and trade messages

The client application can be implemented in a variety of technologies. It can be:

- A browser-based (JavaScript™) application.
- A Java™ applet, Java application, or Java Web Start application.
- A Microsoft Silverlight or .NET application (typically implemented in C# or Visual Basic).
- An Objective-C application running under iOS.
- An Android™ application

StreamLink acts as the interface between the client application and a Liberator server. Caplin Liberator is A financial internet hub that delivers data and messages in real time to and from subscribers over any network.

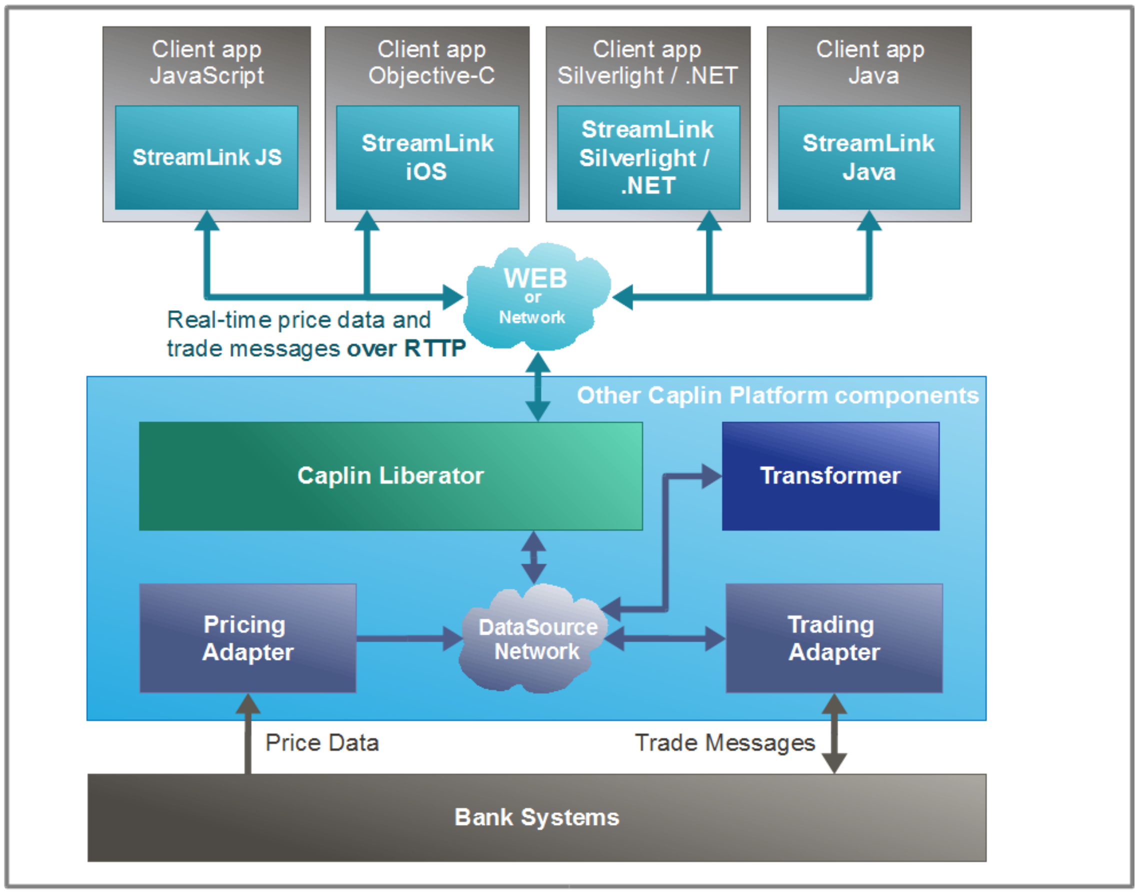
Using StreamLink, the client can connect to the Liberator and receive streaming price data and other market data that is updated in real time. With a suitably coded client, end users can place trades based on the displayed data. StreamLink passes the trade messages to the Liberator for onward transmission to a trading system and receives the response messages via the Liberator, forwarding them to the client application.

3 Key concepts

This section introduces some key concepts about StreamLink.

3.1 StreamLink and the Caplin Platform

StreamLink is a component of the Caplin Platform that resides in client applications. The following diagram shows in a little more detail how it fits in with the rest of the Platform.



StreamLink within the Caplin Platform Architecture

StreamLink communicates with a Liberator server using Caplin's Real Time Text Protocol, RTTP 8, typically across an Internet connection tunneled over HTTP or HTTPS. The StreamLink API conveniently hides from the client application the details of how to handle the RTTP connection and protocol.

Liberator acts as the gateway for all data flows between the Caplin Platform and the StreamLink API in clients. The Liberator communicates with other Platform components using an internal messaging infrastructure called **DataSource**. Liberator obtains data from and sends data to external systems via components called **Integration Adapters** (often just called "DataSources" in older Caplin documents).

In a typical trading application the real-time financial data displayed by clients is obtained from one or more external data feeds, via dedicated Integration Adapters ("Pricing Adapter" in the diagram). The Liberator messages about trades pass between clients and external trading system, via the Liberator and other dedicated Integration Adapters ("Trading Adapter" in the diagram) .

For more information about the Caplin Platform and the components comprising it, see the **Caplin Platform Overview**. For more information about DataSource and Integration Adapters, see the **DataSource Overview**.

3.2 RTTP

RTTP stands for **R**eal **T**ime **T**ext **P**rotocol. This is Caplin's protocol for streaming real-time financial data from Liberator servers to client applications, and for transmitting trade messages between clients and Liberator in both directions.

RTTP is a robust text-based protocol that operates over Intranets, the Web, and over private networks (directly via TCP/IP). When operating over **WANs**, Intranets and the Web, RTTP is typically encapsulated in HTTP, or in HTTPS, to provide the highly secure messaging required by many financial applications.

RTTP has the following features:

- Automatically tunnels between clients and Liberator, through firewalls and proxy servers, with no special TCP/IP port requirements.
- Uses persistent virtual connections, providing rapid and seamless recovery from transient connection loss.
- Is designed to ensure that communication between a Liberator and its clients utilizes bandwidth extremely efficiently.
- Can handle information in numerous formats, including:
  - Structured records.
  - News headlines and news stories.
  - Containers.
  - Directories.

For more information about RTTP data formats, see Data Types 18 in About the data 17.

RTTP connection types

RTTP works (via StreamLink) with various client application technologies: Java, JavaScript, .NET and Silverlight applications, and Objective-C running on iOS. Accordingly it has several connection types to support different combinations of application and network technology, as shown in the following table.

RTTP Connection type	**Client type**	**Characteristics**
Direct	Java, Android applications iOS applications, .NET applications	This is a direct persistent connection via TCP/IP. The client connects to the server via a TCP/IP socket, and the server streams data directly to the client across this connection.
HTTP Streaming	Java, Android applications, JavaScript iOS applications, .NET applications, Silverlight applications	This is a persistent connection that uses tunneling over HTTP or HTTPS. The client connects to the server by requesting a URL. The server holds the requested page open and streams data to the client until the client closes the connection. In browsers, the entirety of the requested page is buffered in the browser, which means that the browser must be periodically closed and reconnected to stop memory consumption.
HTTP Polling	Java, Android applications, JavaScript, .NET applications, Silverlight applications	The client polls the server for data, issuing a fresh HTTP or HTTPS request each time. Because the polling mechanism is relatively inefficient and resource hungry, and can increase message latency, it is usually only used as a fallback mode when other types of connection cannot be established.
WebSocket	Java, Android applications, JavaScript, iOS applications	This is an HTML5 connection type that provides a bidirectional connection between the client and the Liberator. It is the best type of connection to use in browsers.

StreamLink can be configured to define which connection types should be used by a particular client; see Configuration 40.

3.3 StreamLink APIs and SDKs

StreamLink provides an object oriented API that allows applications to access RTTP functionality without needing to understand the details of the protocol itself. It is supplied as an SDK containing the API and an underlying software library. There is an SDK for each type of client technology that can access the Caplin Platform.

Using the appropriate StreamLink SDK, you can build applications in JavaScript, Java, Java for Android, .NET, Silverlight, and Objective-C on iOS. The following table shows which StreamLink SDK is used for which client technology.

Client application technology	StreamLink SDK
JavaScript/Ajax/HTML	**StreamLink JS**
Objective-C on iOS	**StreamLink iOS**
Java	**StreamLink Java**
Java on the Android platform.	**StreamLink Android**
.NET applications	**StreamLink.NET**
Silverlight applications	**StreamLink Silverlight**

Also see the StreamLink Architecture 26.

3.4 Caplin Liberator

Caplin Liberator is a real-time financial internet hub that delivers trade messages and market data to clients over any network that supports RTTP. StreamLink connects client applications to a Liberator and exchanges data with the Caplin Platform (and with other clients) via the Liberator.

Liberator contains a high performance publishing engine capable of delivering millions of updates per second from a single server to multiple clients. It also provides standard Web server functionality to clients using HTTP and HTTPS connections.

Liberator supports all the RTTP data formats – structured records, directories, and so on (see RTTP 8, and Data types 18 in About the data 17).

For more information about Liberator see the **Caplin Platform Overview** and the **Caplin Liberator Administration Guide**.

3.5 Subscriptions and real-time updating

StreamLink uses a publish and subscribe model. A Liberator server manages data which it publishes to clients. Clients *subscribe* through StreamLink to selected data. For example, a client may subscribe to a financial instrument, such as a foreign exchange currency pair – say EURUSD. The Liberator manages the subscriptions requested by all the clients connected to it.

The StreamLink API has a set of subscription classes that allow a client to name the data it wishes to subscribe to and to issue subscription requests. When a client sends a subscription request to the Liberator, if the Liberator does not have the required data item in its cache (see Caching data 12), it will in turn issue a subscription request for the item from an Integration Adapter (for example a Pricing Adapter that supplies indicative price information).

When Liberator has the data available, it sends the client's StreamLink an initial image of the complete data item. StreamLink then makes the data available to the client application. For example, this could be a record with containing indicative bid and ask prices for EURUSD.

As the Liberator receives further updates to the data item, it updates its cache and pushes the updates to all clients that are subscribed to the item. Generally only the parts of the data item that have changed are sent out. For example if just the bid price of a currency pair changes the Liberator sends only the new bid price field to the clients that have subscribed to the currency pair.

The types of data that can be subscribed to are described in Data types 18. Subscriptions can specify parameters that restrict how much data is returned (see Making subscriptions more specific using parameters 33).

StreamLink clients can also publish data, so that it is available to the Liberator, other Integration Adapters, and other subscribing clients (see Creating, updating, and deleting data) 11.

3.6 Creating, updating, and deleting data

Besides receiving data from Liberator, clients can use StreamLink to send data back to the Liberator and/ or its DataSources.

Note: Some Caplin documents use the term “contribution” to describe data that a client sends back to a Liberator.

A client can create new data items, update existing items, and delete existing items, using the appropriate StreamLink commands:

- **Create** – to create data items
- **Publish** – to send updates to data items
- **Delete** – to delete data items.

In practice, whether or not a client can do these things, and where resulting data changes are visible, depends on which Caplin component *owns* the item. The client must also have write permission granted on the data items.

Item is owned by Liberator

If a data item is not owned by a data service (see below), it is owned by the Liberator. In this case the client can create, update, and delete the item (provided it has write access to it). The changes are cached in the Liberator and are immediately available to other clients.

This type of data modification is used for client-to-client chat, for example.

Item is owned by a data service

(For an explanation of data services see the **Caplin DataSource Overview**.)

When a data service owns a data item*, clients cannot create or delete the item. A client can *update* the item, provided it has write access to it. The Liberator does not keep the update in its cache, rather it just sends it on to the Integration Adapter(s) providing the data service. The owning Adapter may keep the updated information to itself, to forward it to an external system for example.

This type of data modification is typically used in trade messaging, where the data items are trade messages exchanged privately between a particular client and a Trading Adapter. Effectively a client sends a trade message to the Integration Adapter by publishing an update that other clients cannot see.

Alternatively the Integration Adapter could send the update back to the Liberator, whence the Liberator will cache the new data so that other clients (having read access permission to the data) will be able to see the new values.

* A data item is owned by a data service if its subject name matches the namespace configured for the data service, and the Integration Adapter(s) that provide the data service are configured to accept updates.

3.7 Caching data

To ensure optimum performance and reduce network load, data is cached within the Liberator server. Liberator maintains a cache of all the items that clients have subscribed to. The cache is updated as updates are received from the Integration Adapters connected to the Liberator, or from StreamLink clients. Liberator records which clients have subscribed to which items, so as clients subsequently unsubscribe, it can delete items from its cache when they are no longer subscribed to by any client.

Liberator can also cache multiple levels of data (see Type 2 data 20), and record history for time-series replay (see Type 3 data 21).

3.8 StreamLink in trading applications

StreamLink can be used within trading applications. As well as being the mechanism by which a client trading application obtains price information, StreamLink can provide the bi-directional transport for the message flows generated when the end-user executes trades.

For more information see Trade messages 25.

4 StreamLink features

This is the list of StreamLink features. At present the StreamLink SDKs that support the StreamLink 6 API are StreamLink.NET (**SLN**), StreamLink Silverlight (**SLS**), StreamLink JS (**SLJS**), StreamLink iOS (**SLi**), StreamLink Android (**SLA**), and StreamLink Java (**SLJ**).

Y = Feature currently available.

L = Limited implementation of this feature

N = Feature not currently available

Many of the features in the list are currently available in version 4.n of other StreamLink SDKs (StreamLink Java, and StreamLink for Browsers).

Note that the version 4.n SDKs do not have the same API architecture as StreamLink 6 (see StreamLink architecture 26). StreamLink JS is an improved implementation of StreamLink for Browsers that conforms to the version 6 architecture.

Supported data types

Features	SLN	SLS	SLJS	SLi	SLA	SLJ
◆ Records (see About the data 17):						
– Type 1: subject + fields.	Y	Y	Y	Y	Y	Y
– Type 2: Supports level 2 quote data.	Y	Y	Y	Y	Y	Y
– Type 3: Holds history of updates.	Y	Y	Y	Y	Y	Y
◆ Containers.	Y	Y	Y	Y	Y	Y
◆ Directories	Y	Y	Y	N	Y	Y
◆ Pages.	Y	Y	N	N	N	N
◆ News Headlines.	Y	Y	Y	N	Y	Y
◆ News Stories.	Y	Y	Y	N	Y	Y
◆ Chat.	Y	Y	N	N	Y	Y
◆ Permissions.	Y	Y	Y	Y	Y	Y

Supported connection types (to server)

Features	SLN	SLS	SLJS	SLi	SLA	SLJ
There are several connection types to support different combinations of application and network technology. See RTTP connection types 9.						
◆ Direct via TCP/IP.	Y	N	N	Y	Y	Y
◆ HTTP/HTTPS tunneled.	Y	Y	Y	Y	Y	Y

Data transmission

Features	SLN	SLS	SLJS	SLi	SLA	SLJ
◆ Real-time data updates.	Y	Y	Y	Y	Y	Y
◆ Data snapshots (see Obtaining Data snapshots 30).	N	N	Y	Y	Y	Y
◆ Trade messaging support.	Y	Y	Y	Y	N	N
◆ Batching of requests and discards – one request message can subscribe to many data items or discard many subscriptions.	Y	Y	Y	Y	Y	Y

Data manipulation

Features	SLN	SLS	SLJS	SLi	SLA	SLJ
◆ Ability to specify server-side data filtering from the client (See Filtering data 34):						
– Record image filtering.	Y	Y	Y	N	Y	Y
– Record update filtering.	Y	Y	Y	Y	Y	Y
– News filtering.	Y	Y	Y	N	Y	Y

Commands to modify data

Features	SLN	SLS	SLJS	SLi	SLA	SLJ
◆ The client can modify data on the server, on Integration Adapters, and on other clients:						
– Create new data items.	Y	Y	Y	Y	Y	Y
– Delete existing data items.	Y	Y	Y	Y	Y	Y
– Publish to existing data items (update them).	Y	Y	Y	Y	Y	Y

Performance

Features	SLN	SLS	SLJS	SLi	SLA	SLJ
◆ Efficient data transmission through the RTTP protocol.	Y	Y	Y	Y	Y	Y
◆ Dynamically configurable throttling levels: global and per subject (see Throttling 38).	Y	Y	Y	Y	Y	Y

Resilience

Features	SLN	SLS	SLJS	SLi	SLA	SLJ
◆ Rapid and seamless recovery from transient connection loss.	Y	Y	Y	Y	Y	Y
◆ NOOP connection monitoring: RTTP detects connection loss even when TCP/IP sockets are not closed properly.	Y	Y	Y	Y	Y	Y
◆ **Failover** modes of arbitrary complexity through configuration (see Resilience, failover, and load balancing 41) 41.	Y	Y	Y	Y	Y	Y

Security

Features	SLN	SLS	SLJS	SLi	SLA	SLJ
◆ Single sign-on support through Caplin KeyMaster integration.	Y	Y	Y	Y	Y	Y

Status reporting

Features	SLN	SLS	SLJS	SLi	SLA	SLJ
◆ Liberator Data Service notifications and object status updates.	Y	Y	Y	Y	Y	Y
◆ Access to connection statistics.	Y	Y	L	Y	L	L

Configuration

Features	SLN	SLS	SLJS	SLi	SLA	SLJ
◆ Configuration through JSON, Configuration Objects, or XML (see Configuration 40).	Y	Y	Y	N	Y	Y

Logging

Features	SLN	SLS	SLJS	SLi	SLA	SLJ
◆ Set log level and categories of information to log.	Y	Y	Y	Y	Y	Y
◆ Set up custom logger class.	Y	Y	Y	Y	Y	Y

Instrumentation

Features	SLN	SLS	SLJS	SLi	SLA	SLJ
◆ Message latency measurement.	Y	Y	Y	N	Y	Y

5 About the data

StreamLink and Liberator have a common view of the messages they pass between each other, based upon a set of standard data formats used by RTTP. These formats are designed to support efficient transmission of information for financial applications. The same formats are used internally to the Caplin Platform within DataSource messages (see **DataSource Overview**).

Note: Within Liberator and other Caplin Platform components, such as Integration Adapters, the RTTP and DataSource data items are often described as "objects".

5.1 Subjects, symbols, and fields

An RTTP message is identified by its **subject.** Where the message is about a financial instrument, the subject often contains a **symbol –** a letter or sequence of letters used to identify the financial instrument. A message can be a record 18, consisting of a subject and a number of fields.

Simple record message:

Subject	Fields
/FX/EURUSD	price=1.334

In this example the record sent by Liberator to StreamLink contains foreign exchange (FX) price information. The subject /FX/EURUSD contains the symbol EURUSD, which defines the currency pair Euros and US Dollars. The price field contains the indicative exchange rate in US dollars per Euro.

Where did the price information in this example come from?
The prices would typically originate from a Foreign Exchange price feed. An Integration Adapter in the Caplin Platform converts the prices information into DataSource messages, which are passed to a Liberator. The Liberator caches the records and pushes them out to subscribing clients via RTTP and StreamLink.

Subjects can have an arbitrary format, but they must start with a "/" as shown in the example above, and are usually organized in a "/" separated directory structure (see Directories 24 in the Data types 18 section).

The actual symbol structure and naming conventions used depend on the type of system that the Caplin Platform forms a part of (for example a foreign exchange trading system, or a securities market data system), and the naming conventions used by the external systems connected to the Integration Adapters. Integration Adapters may map the symbols of records received from external systems into formats used internally to the Caplin Platform, and for presentation to end users via StreamLink.

A **field** is a specific piece of information relating to the subject. For example, the fields for a record containing a foreign exchange quote might include "Bid" (the bid price), "Ask" (the asking price), and "Amount" (the maximum amount of the base currency that can be traded at the quoted price).

When the Liberator sends an update message to a client, it normally* only includes in the message the fields that have changed.
For example, when a client first subscribes to Liberator to obtain a quote for /FX/EURUSD, the record returned will contain *all* the relevant fields:

/FX/EURUSD Bid=1.3442 Ask=1.3448 Amount=10000

If the Ask price subsequently changes, the Liberator will send the client an update containing just the new value of the Ask field:

/FX/EURUSD Ask=1.3450

The range of fields available for a particular financial instrument depends on the Integration Adapter from which the data about that instrument has been obtained.

* Alternatively Liberator can be configured to always send *all* the subscribed fields of a data item to the client whenever the item is updated.

Also see Directories 24 in Data types 18.

5.2 Data types

StreamLink and RTTP can handle the following types of data:

- Records
- Containers
- Directories
- Pages
- News headlines and news stories
- Chat
- Permissions

Note: Other Caplin components, such as Liberator often refer to these data types as “object types”.

Records

A record is a means of storing and displaying information. Records are composed of one or more fields which may be of different types. For example, a record containing foreign exchange data could have several price fields (such as the last bid and ask prices) together with time and date fields, whereas an index record would have a price field but no bid or ask values.

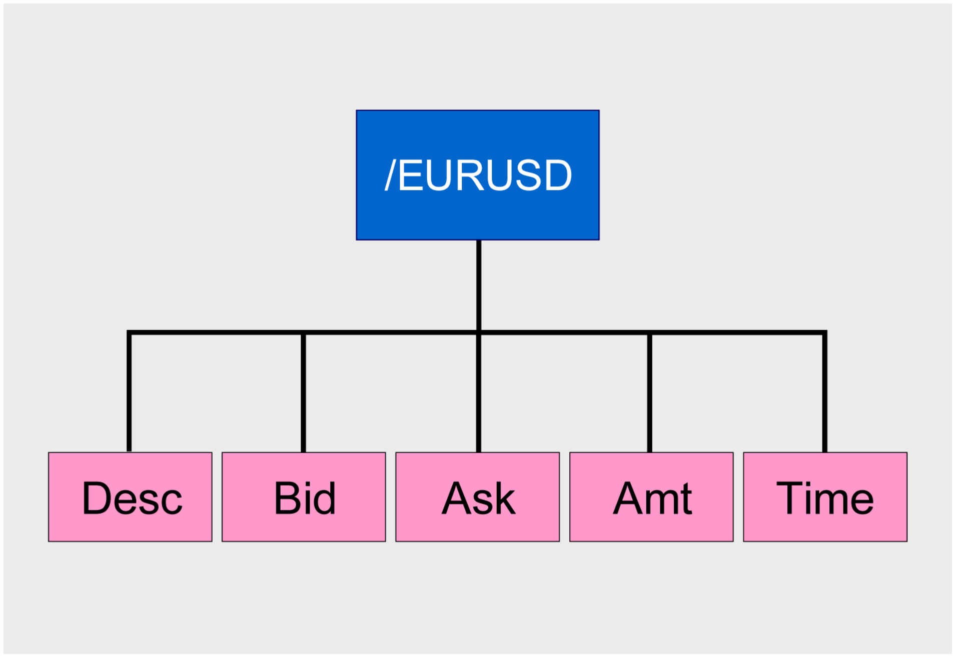
The data in a record can be structured in one of three different ways, known as Type 1 data, Type 2 data, or Type 3 data.

Records can be used to hold trade messages 25 as well as financial market data.

Type 1 data

The majority of record based data is structured as Type 1 data. This means there is only one level of fields under the subject that identifies the record.

The following diagram shows an example field structure for a quotation in a foreign exchange trading system.



Record containing Type 1 data

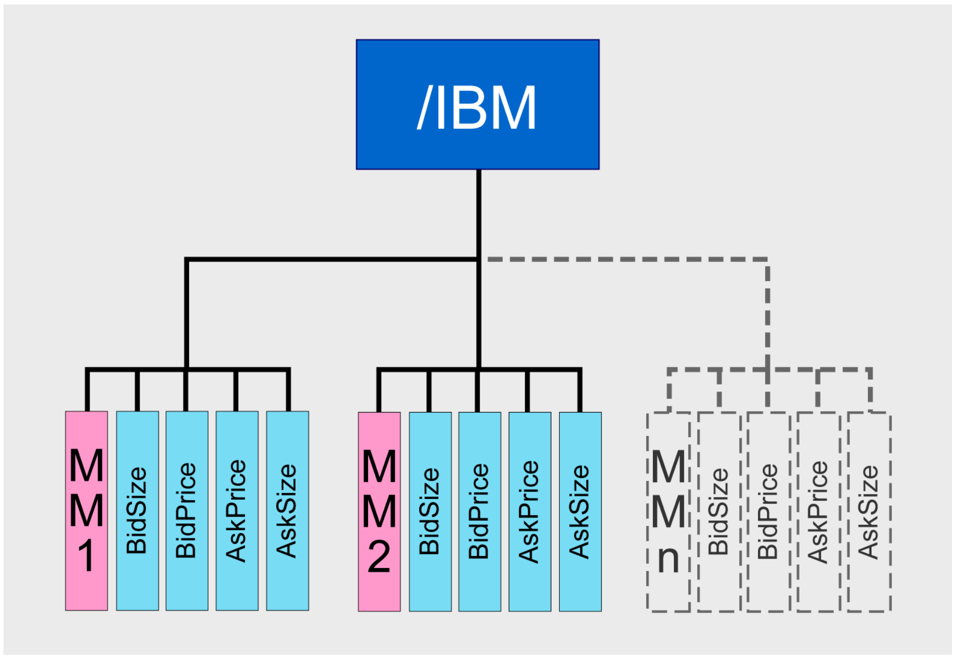
Here the record /EURUSD contains quote data for conversion between Euros and US Dollars. It has one level consisting of five fields: Desc (description), Bid (bid price), Ask (ask price), Amt (maximum amount of dollars that can be traded at the quoted bid and ask prices), and Time (the date and time of the quote).

When a client first subscribes to /EURUSD, StreamLink receives a type 1 record containing all the fields with their latest values (assuming the client didn't exclude any of the fields from the subscription). Subsequently whenever StreamLink receives an update to /EURUSD, the record contains just the fields whose values have changed.

Type 2 data

Type 2 data is often referred to as “level 2” data, as it is used for level 2 quotes. Level 2 quote data enables several price quotes per subject (coming from different market makers or traders) to be available at all times.

The field structure shown in the following diagram might be applicable for a simple level 2 display for equity data (in this case IBM stock), where there are several active market makers.



Record containing Type 2 data

In this case the /IBM subject (primary key) has a secondary key of Market Maker (MM). The record contains quote data for each of the market makers providing quotes (MM1, MM2, and so on). This allows a subscriber to see the full set of quotes in the market. An update to the record will always have a market maker associated with it, so only the fields with that market maker as a secondary key will be overwritten.

A typical use for Type 2 data is to feed the display of a market order book that is in a tabular format:

/IBM:

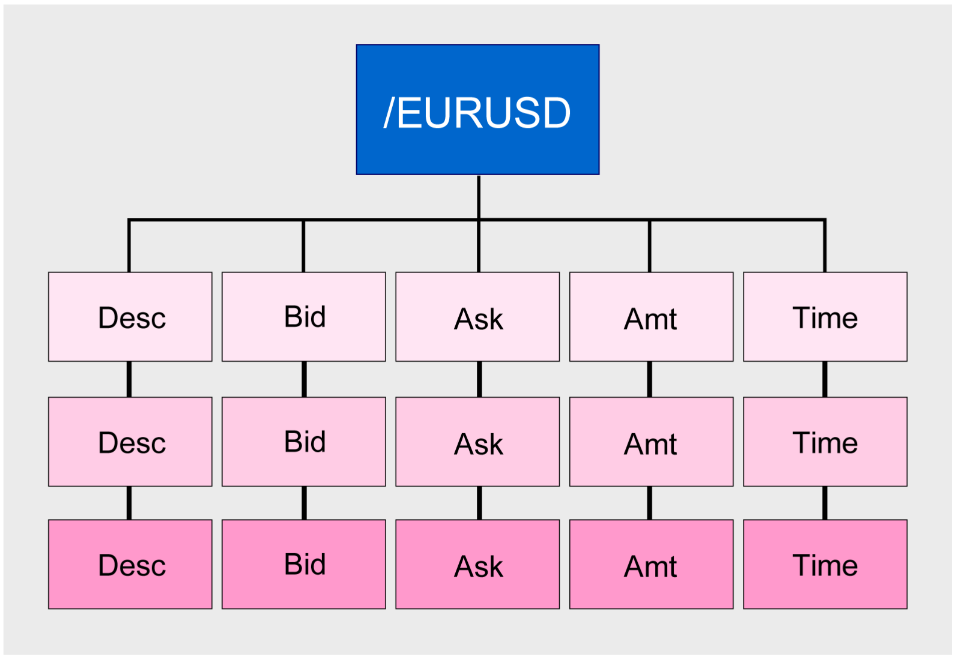
MM (Market Maker)	BidSize	BidPrice	AskPrice	AskSize
MM1	1000	1.9814	1.9823	1000
MM2	2000	1.9926	1.9999	2000
MM3	1000	1.9602	1.9613	1000

A single field in the table can be easily updated from the record: for example, Key="MM1" AskSize="2500".

Type 3 data

Type 3 data stores the history of updates to the record. A common use for data of this type is holding and viewing daily trade activity, where typically this mechanism will only be used for a day at a time before the Liberator's cache is deleted and the update list starts again.

The following diagram shows a record containing Type 3 data that records the history of quotations in a foreign exchange trading system.



Record containing Type 3 data

This particular record holds the history of quotes for conversion between Euros and US Dollars. The boxes running from left to right are the fields of the record; these are Desc (description), Bid (bid price), Ask (ask price), Amt (maximum amount of dollars that can be traded at the quoted bid and ask prices), and Time (the date and time of the quote).

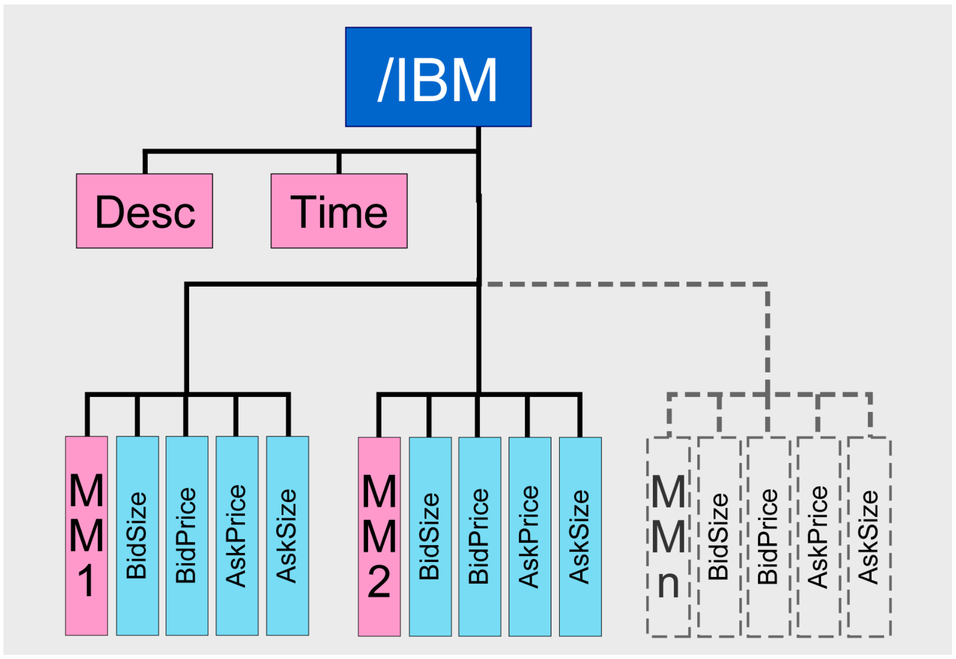
Each new record update is inserted at the end of the list. So the topmost line of fields contains the oldest known value of the record, and the bottom line of fields contains the most recent value of the record.

Caplin Liberator can maintain type 3 data structures, and it allows you to configure the amount of update history to be retained in the structure. When subscribing to a record containing type 3 data, the StreamLink client will immediately receive from Liberator all cached updates, followed by any new updates as they occur.

Type 1 and Type 2 data combined

A single record can contain both Type 1 and Type 2 data.

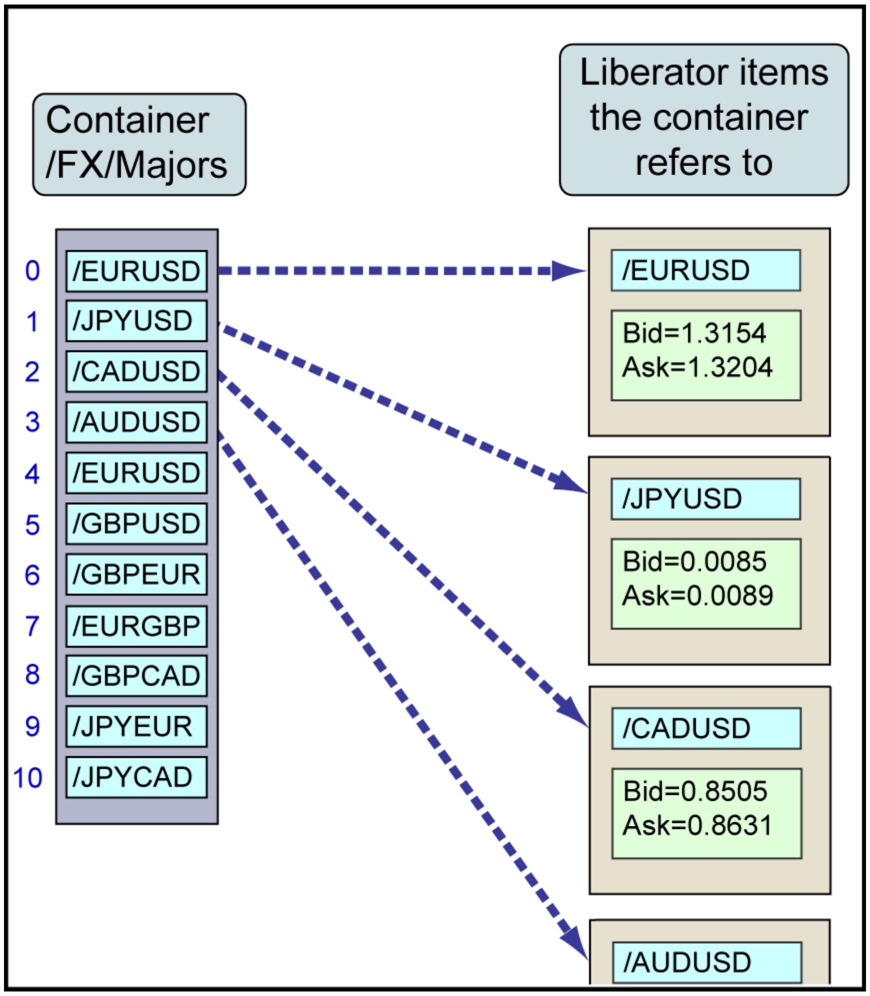
The following record is for equity data (in this case IBM stock), containing Type 1 data consisting of a description (Desc) and the time of the quote (Time), followed by Type 2 data listing price quotes from several different market makers (MM1, MM2, and so on).



Record containing Type 1 and Type 2 data

Containers

A container holds a set of references to other data items in the Liberator:



Structure of a container

When a client requests a container item from StreamLink, it is automatically subscribed to the linked items as well. Any parameters on the container subscription (such as fields or filters) are passed on to the linked item subscriptions.

Containers are managed on the server side (that is, by the Liberator) on behalf of all the subscribing clients, so there is less work for the clients to do. For example, the Liberator handles the addition and deletion of items in the container; it automatically adjusts client subscriptions accordingly and communicates the changes to the clients via StreamLink. In contrast, when a client subscribes to a directory (see Directories 24), it is not automatically subscribed to the items in the directory; the client must explicitly subscribe to them as required.

A client can request the Liberator to provide a windowed view of the items in the container, which can help reduce the processor load and memory usage on the client – for more information see Specifying container parameters 37.

Directories

StreamLink, RTTP, and Liberator understand and utilize the concept of a directory based hierarchical name space for data items. This is realized in the use of the “/” delimiter within the subject of the record.

For example the subject /FX/EURUSD comprises

- the root directory “/” (the first “/” in /FX/EURUSD)
- the directory “FX” (Foreign Exchange) underneath “/”
- the symbol “EURUSD” (Euros and US Dollars) underneath “/FX/”

A client can subscribe to a directory and StreamLink will then inform the client when items are created or deleted within that directory. Subscribing to a directory does not automatically subscribe to the items in the directory; the client must explicitly subscribe to them as required.

Pages

A page is a free format piece of text made up of rows. This data type is normally used to display information that was originally formatted for terminals that only display text. Typical sizes are 14 rows of 64 characters (“Reuters small page”) and 25 rows of 80 characters (“Reuters large page”).

News headlines and news stories

A news headline is a relatively short message containing free text, with a link to the more detailed news story behind the headline, and a date. The story link can be in any format; for example, it could be a link to an RTTP news story, or a URL pointing to an external web page. A headline can also have tags (or codes) associated with it, for use in searches.

A request for a news headline may contain a filter string using a simple logical syntax. The filter allows a client application to limit the headline updates it receives.

A news story is an arbitrary length text item, referred to by one or more news headlines.

Chat

RTTP chat items allow users logged into a Liberator to chat in real-time. Each chat item represents a virtual chat room for 2 or more users. A client sends a message on a chat “channel” by updating (publishing to) the associated chat item.

Permissions items

Clients can receive updates from Liberator about changes to access permissions. These updates are sent in permissions items, which have the same structure as Type 2 data 20. A client can use the updated permission information to modify the way the application behaves.

For more information see Permissions objects 40 in the Authentication and permissioning 39 section.

5.3 Trade messages

When the Caplin Platform is deployed as a trading platform, a client can use StreamLink to pass messages relating to trade transactions between the client and the trading system behind the Liberator. The client uses the StreamLink Publish command to send update messages to the trading system, and the trading system responds by sending updates back to the client. (In practice, the Publish command is hidden under a richer Trading API that the client actually uses.)

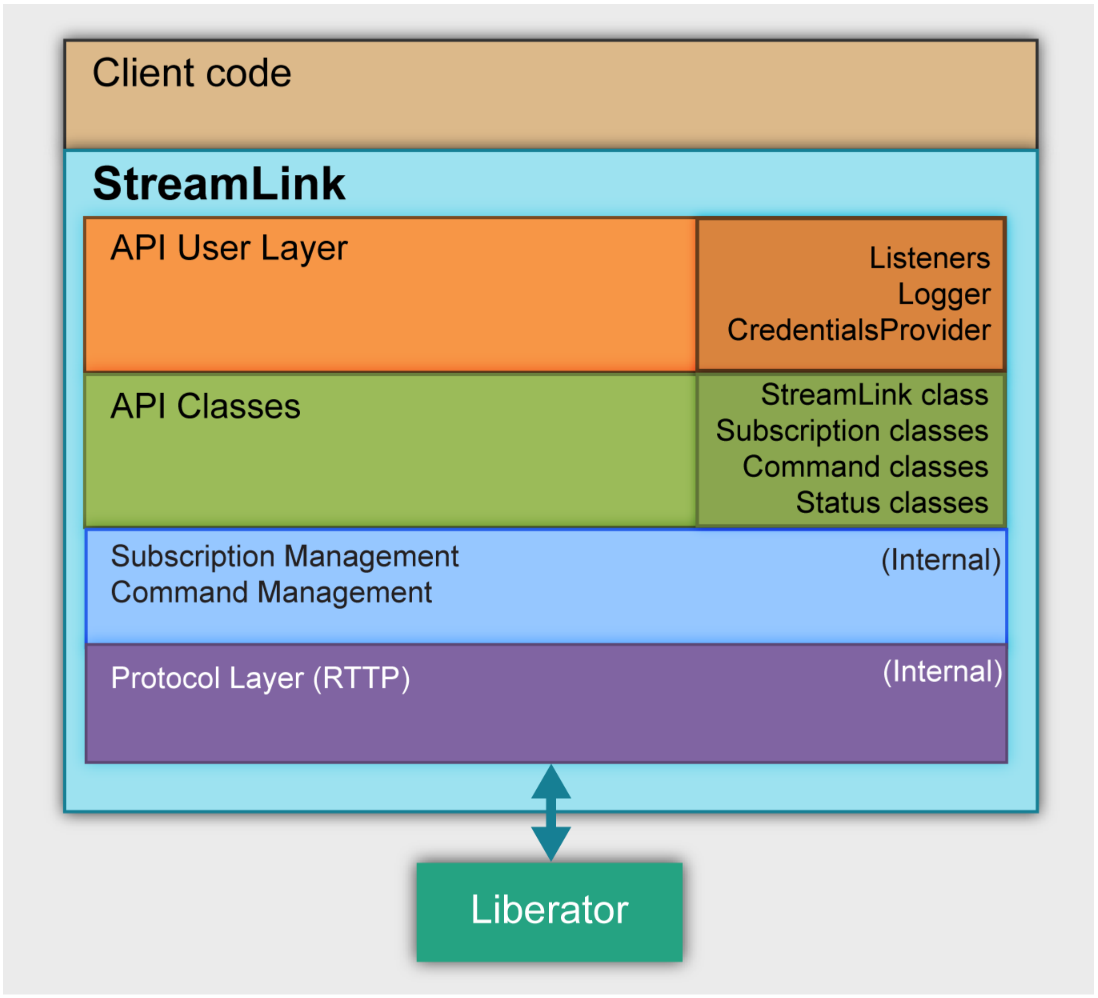
Typically the client issues an “open trade” request, which automatically subscribes the client to a subject dedicated to trade messaging (for example /TRADE/FX). The trading system sends the client (via a Trading Integration Adapter and the Liberator) an update to /TRADE/FX containing the request acknowledgement. The client and the trading system then exchange further messages as updates to /TRADE/FX, according to the particular trade model being executed, until the trade is complete.

6 StreamLink architecture

The StreamLink 6 APIs are provided as a set of classes and interfaces in the implementation language of the client application (see StreamLink APIs and SDKs 10). The StreamLink code is divided into four layers:

- The API User layer
- The API Classes
- Subscription Management and Command Management
- The Protocol layer

These are shown in the following architecture diagram



StreamLink architecture

API User Layer

The API User Layer defines interfaces that developers using StreamLink must implement as concrete classes in the client code.

The most important of these interfaces are the Listeners which define callback methods to handle events returned by StreamLink. For example, **SubscriptionListeners** handle the data (images and updates) received from Liberator when the client has subscribed to a particular item. The actual implementations of such callbacks depend upon what the client code needs to do with the received data.

There is also a **Logger** interface, whose implementation enables the client to record StreamLink activity in an appropriate manner.

The **CredentialsProvider** interface is for supplying login information to the Liberator. The implementation of a **CredentialsProvider** could obtain this information from a database, a web site, or from a single sign-on system.

API Classes

The API Classes layer contains the public classes and methods that client code uses to initiate and control interactions with StreamLink.

The main class is called **StreamLink**; every client must create an instance of this class to use the functionality provided by StreamLink. There are also **Subscription** classes for setting up subscriptions to data, and **Command** classes used to send commands to Liberator. Commands include, Create (to create data items), Publish (to send updates to data items), and Delete (to delete data items).

The **Status** classes allow a client to obtain information about the status of RTTP connections and Liberator's data services. (For an explanation of data services see the **Caplin DataSource Overview**.)

Subscription Management and Command Management layer

This layer is internal to StreamLink. It implements the functionality of the StreamLink API; managing subscriptions, implementing the commands issued from the API User Layer, making calls on the RTTP protocol layer, and handling data and status information received from RTTP for onward transmission to the callback methods defined in the API User Layer.

Protocol Layer

This layer is also internal to StreamLink. It manages the connections to the Liberator server, and implements the RTTP protocol that handles bi-directional communication with Liberator.

7 How to use StreamLink

This section contains some simple examples of how client applications can use the StreamLink 6.n API. The examples contain some simple code fragments, illustrating how the API classes and methods are used. Since the StreamLink API is written in a number of different object oriented languages (see StreamLink APIs and SDKs), the code fragments are in pseudo object oriented code – *the actual implementation details will vary from one StreamLink SDK to another*.

The rest of the section contains more detailed information about using StreamLink's features.

7.1 Asynchronous operation

The StreamLink API operates asynchronously. This means that when the client code issues a subscription request or a command to StreamLink, the response is not returned immediately through the method used to issue the request or command. Instead, you must set up a listener object containing one or more callback methods, and then supply this listener to StreamLink.

StreamLink calls the appropriate callback method(s) on the listener object, to communicate data and command responses back to the client code.

This method of operation is shown in the following examples.

7.2 Subscribing to data and receiving updates

To subscribe to data and receive updates to the data you need to

- implement (code) some interfaces from the API User Layer,
- call various methods from the API classes layer to connect to a Liberator and set up the subscription.

Implementing the interfaces

Before using StreamLink you need to implement as concrete classes an appropriate set of listener and other interfaces defined in the API User Layer. These implementations integrate StreamLink with your client application.

As a minimum, you must implement a **SubscriptionListener**.

StreamLink calls the **SubscriptionListener** to handle subscription events. For example, to deal with events concerning subscriptions to record-structured data, you need to implement on the **SubscriptionListener** the methods relating to record data. Your class should contain implementations of the methods **onRecordUpdate()**, **onRecordType2Update()**, **onRecordType3Update()**, to handle updates to type1, type2, and type 3 records respectively in a manner suitable for the client application. You can also implement methods for receiving permissions, containers, and directories. At run time, StreamLink passes in to each method both the subscription to which the event relates, and an event containing the updated fields as name-value pairs.

The **SubscriptionListener** may also implement an **onSubscriptionError()** method to deal with errors in subscriptions, and an **onSubscriptionStatus()** method to handle changes to the status of the subscription. (A status message relates to the state of the Liberator data services that handle the subscription, such as “Stale”, “Limited”, “OK”. For more information about data services see the **Caplin DataSource Overview**.)

Depending on how Liberator is configured, you may need to prove that you have the right to connect to it. The typical way of doing this is with a username and password, which you can provide to the **StreamLinkFactory.create** method. In a production implementation the credentials would normally be provided by an implementation of a **CredentialsProvider** which could be much more sophisticated. For example, it could obtain login credentials from a database, a web site, or a single sign-on system. The credentials could take the form of a digitally signed secure token.

Calling StreamLink

The following simple example shows how to use the StreamLink API to subscribe to a simple record containing a share price. The subject is /MSFT. This pattern is used for subscribing to all types of data.

- The application will issue a subscription request to StreamLink, so the first thing to do is implement a **SubscriptionListener** that receives the updates resulting from the subscription. In the pseudo code fragments below the record subscription listener implementation is called **MySubscriptionListener**.
- Get an instance of the StreamLink class:

```
StreamLink streamLink = StreamLinkFactory.create({
                            liberator_urls: "rttp://liberator1:8080",
                            username: "demouser",
                            password: "demopass"
                            });
```

  In StreamLink Java, there are variants of the constructor that enable you to obtain configuration from a properties file. The **create()** method also lets you set less common configuration options by passing in an optional **StreamLinkConfiguration** object. For more information, see Configuration 40.

- Set up the record subscription listener:

```
// Create an instance of a class that implements SubscriptionListener.
mySubsListener = new MyRecordSubscriptionListener();
```

- Create a subscription to /MSFT via the **streamLink** that you acquired earlier, supplying the subscription listener that will receive updates, events, and errors relating to the subscribed data item:

```
recordSubscription = streamLink.subscribe("/MSFT", mySubsListener);
```

- Connect to the Liberator:

```
streamLink.connect();
```

  StreamLink connects to the Liberator defined in its configuration and logs in to the Liberator using the details previously set up in **credentialsProvider**.

- Some time later Liberator returns an image of the /MSFT record through a call to the **onRecordUpdate()** method of **mySubsListener** (instantiation of **MyRecordSubscriptionListener**). Subsequently, each time StreamLink receives an update to /MSFT, it calls **onRecordUpdate()**, passing the updated fields and their new values.

  You only need to connect to the Liberator once; subsequent subscription requests are passed to the Liberator immediately. You can make subscription requests *before* connecting to the Liberator; StreamLink queues them until a connection is established.

7.3 Subscribing to more than one data item

The StreamLink API supports subscriptions to multiple data items using a single connection to the server. So there is no need to create multiple instances of the **StreamLink** class in order to make multiple requests.

For example, you can subscribe to multiple records and receive updates into a single instance of **RecordSubscriptionListener**. However, StreamLink also allows you to register a different listener for each subscribed item.

- The following example shows how to subscribe to both /MSFT and /YHOO and receive updates for both of these subjects into the same **RecordSubscriptionListener**.

```
msftRecordSubscription = streamLink.subscribe("/MSFT", mySubsListener);
yhooRecordSubscription = streamLink.subscribe("/YHOO", mySubsListener);
```

Note: StreamLink automatically batches multiple subscription requests into a single request call to maximize efficiency “over the wire”.

7.4 Obtaining data snapshots

StreamLink JS, StreamLink Java, StreamLink iOS, and StreamLink Android clients can also request all the data for a subject as at a single moment; this is called a snapshot. Snapshots return only the data currently in the Liberator’s cache, so the subject needs to be subscribed to first (if it is from an active DataSource). The snapshot API, **StreamLink.Snapshot()**, is similar to the subscription API, **StreamLink.Subscribe()**, except that it does not return a **Subscription** object.

The following example shows how to obtain a snapshot of the data for the subject /MSFT. This example assumes that, as described in Subscribing to data and receiving updates 28, you have connected to Liberator, set up the record subscription listener (`mySubsListener`), and created a subscription to /MSFT

- Request the snapshot:

```
streamLink.snapshot("/MSFT", mySubsListener);
```

- Some time later Liberator returns an image of the /MSFT record through a call to the **onRecordUpdate()** method of **mySubsListener** (instantiation of **MyRecordSubscriptionListener**).

- If you wish to obtain another snapshot of /MFST, you must request it again.

7.5 Sending data to the Liberator

This example shows how a client application can use StreamLink to modify data on a Liberator server. In this case the application creates a new record data item. This pattern is used for issuing all StreamLink commands.

- The application will need to issue a *command* to StreamLink, so the first thing to do is implement a **CommandListener** to receive the results of the command. In the pseudo code fragments below the command listener implementation is called **MyCommandListener**.
- Get an instance of the **StreamLink** class, as in Subscribing to data and receiving updates 28:

```
StreamLink streamLink = StreamLinkFactory.create({
                           liberator_urls: "rttp://liberator1:8080",
                           username: "demouser",
                           password: "demopass"
                           });
```

- Set up the command listener:

```
// Create an instance of a class that implements CommandListener.
myCommandListener = new MyCommandListener();
```

- Create the fields that you want to publish. In JavaScript, you can represent these in object literal notation:

```
fields = {"dBestBid": "1.2345", "dBestAsk": "1.2466"};
```

- Issue a “Publish To Subject” command to **streamLink** to send the fields. Assuming the application is already connected to the Liberator, StreamLink sends the command immediately.

```
streamLink.publishToSubject("/MyRecord", fields, myCommandListener);
```

  When the command has executed successfully, StreamLink calls the **onCommandOk()** method of **myCommandListener** (instantiation of **MyCommandListener**). If the command fails, for example because the user does not have permission to create record subjects on Liberator, StreamLink calls **onCommandError()** instead.

Also see Creating, updating, and deleting data 11.

Persistence

When a StreamLink application publishes data to Liberator, the data does not normally persist within Liberator if the connection to the Liberator is lost. You can use the **persistent** configuration attribute to ensure that the data does persist across reconnections. When **persistent** is set to `true`, if the application reconnects to the Liberator (or to a different Liberator, depending on the failover configuration), StreamLink automatically resends the data.

For example, consider that Liberator has an object `/MyMode` representing a mode that the end-user is in, and the default mode of this object is `"A"`. When StreamLink logs an end-user in to Liberator, the mode starts off as `"A"`. To change the end-user to mode `"B"` you would simply send the following publish command:

```
myCommandListener = new MyCommandListener();
fields = {"mode": "B"};
streamLink.publishToSubject("/MyMode", fields, myCommandListener);
```

If the end-user is disconnected from the Liberator, the user session is lost. With this coding, when StreamLink reconnects, the end-user's mode will have defaulted back to mode `"A"`. If you want the mode to be automatically set to `"B"` again when StreamLink reconnects, specify the persistent configuration attribute as `true`:

```
myCommandListener = new MyCommandListener();
fields = {"mode": "B"};
myCommandParameters = {"persistent": true}
streamLink.publishToSubject("/MyMode",
                             fields,
                             myCommandListener,
                             myCommandParameters);
```

This code ensures that the object `/MyMode` with field `"mode=B"` is reliably sent to the Liberator whenever a reconnect occurs.

If you subsequently no longer want the `/MyMode` state to persist in mode `"B"`, simply remove the persistence, as follows:

```
myCommandListener = new MyCommandListener();
fields = {"mode": "B"};
myCommandParameters = {"persistent": true}

myCommandSubscription = streamLink.publishToSubject("/MyMode",
                                                    fields,
                                                    myCommandListener,
                                                    myCommandParameters);
...
/* Some time later, myMode no longer needs to be persisted...*/

myCommandSubscription.unPersist();
```

Note that after the call to `unPersist()`, `/MyMode` remains in mode `"B"` until such time as StreamLink reconnects to the Liberator.

7.6 Discarding data

When a client no longer wishes to receive updates to a subscribed data item it can *unsubscribe* from the item. This effectively discards it as far as the client is concerned. However, the item is still present on the Liberator, so the client can subsequently subscribe to it again if required.

A client can also permanently delete an item from Liberator by executing the StreamLink Delete command. For this to succeed the client must have write access permission to that item on the Liberator .

7.7 Making subscriptions more specific using parameters

The client can supply various types of parameter to a subscription request, to ensure that only specifically required data is returned. For example, you can specify which fields of a record are to be returned (see Specifying fields 33), you can further restrict the date using filters (see Filtering data 34), and you can control which data is returned within a container (see Specifying container parameters 37).

Specifying fields

When the client subscribes to a record it can specify which fields of the record are to be returned, so the fields that are irrelevant to the application are ignored (Liberator does not send them across to the client).

Filtering data

When a client subscribes to data it can specify a filter that restricts the information returned, so the client only receives values that it is interested in. The filtering is done on the Liberator, rather than in the client, so that network traffic and client side processing are reduced.

The filter is defined as an expression based on the fields in the data item.

For example:

- An FX trading client application subscribes to the currency pair /EURUSD, requesting updates for the Bid field. The subscription request includes a filter "Bid > 1.3440", meaning that the client (and hence the end-user) is only interested in receiving updates for /EURUSD where the Bid price is greater than 1.3440
- A subscription to news headlines includes a filter that selects only headlines containing the word “industrials”.

Record filtering

Data structured as records (see Records 18) can be filtered using filter expressions containing the following operators:

Character	Meaning
`|`	or
`&`	and
`=`	equal to
`!=`	not equal to
`<`	less than
`>`	greater than
`<=`	less than or equal to
`>=`	greater than or equal to
`~`	Provides matching for items in comma-separated string lists. For example, `~abc` matches a field in `"xyz,abc,def"` but not in `"xyzabcdef"`
`( )`	Parenthesis – perform these filters first

For example, the filter expression

```
(BidSize > 1000) & (AskSize > 1000)
```

selects only those records where both the `BidSize` and `AskSize` fields each contain a value greater than 1,000.

There are two sorts of record filter: update filters and record filters.

An **update filter** is applied to the data *within* a particular update of the data item. An update record does not necessarily contain all the fields that the client originally subscribed to within the item. If the filter is dependent on a field that is not present within an update, the filter operation will result in “no match”, and the Liberator will not send the update to the client.

An **image filter** is applied *after* the data within a particular update has been applied to the image of the data item. The filter may specify a field that is not present within the update. When this happens, the Liberator uses its cached value of the field to determine whether the update matches the filter.

The following table shows an example of the difference in behavior between update filters and image filters. In both cases the filter used is
`(BidSize > 1000) & (AskSize > 1000).`

Each row of the table represents the result of an update, where the updates have been applied to the same record item in the order 1, 2, 3, 4, 5.
{none} means the field in question was not updated.

Updated record fields:

Update no	**Bid**	**BidSize**	**Ask**	**AskSize**	**Matches Update Filter?**	**Matches Image Filter?**
1.	54.25	1500	55.00	2000	Yes	Yes
2.	54.00	2000	{none}	{none}	No	Yes
3.	54.50	500	{none}	{none}	No	No
4.	54.25	1500	54.75	500	No	No
5.	54.00	2000	54.75	1500	Yes	Yes

News filtering

News headlines can be filtered by specifying a news filter when creating an object. The filter expression consists of news codes and/or whole words to search for. A news filter expression can contain the following operators:

Character	Meaning
`[space]`	or
`|`	or
`+`	and
`&`	and
`-`	and
`=`	equal to
`!`	not
`~`	not
`'`	start or end of free text search string
`"`	start or end of free text search string
`( )`	parenthesis – perform these filters first

For example, the filter expression

```
UKX | ('stock rises' & (!'preference'))
```

selects only those news headlines with code `UKX`, or where the headline contains the phrase “stock rises” and does not contain the phrase “preference”.

News codes can only contain capitals and numbers, and have a maximum length defined in the Liberator configuration. Capitalized items in the headline that are longer than the maximum news code length are treated as text for the purposes of searching, as are items in mixed case and lower case. Use `'` or `"` to force a text search. For example `'UKX'` selects news headlines containing the text “UKX”.

Specifying container parameters

When a client subscribes to a container it can specify a window into the container, that is, the start and end rows of the container data that are to be returned. The Liberator only sends the client updates for the items currently in the window.

The client can ask StreamLink to move the window on the container (for example because the end-user has scrolled down through the displayed view of the container window). StreamLink will then ask Liberator to supply the data for the items that are currently in the scope of the window.

This server-side paging capability helps reduce the processor and memory requirements on the client in situations where the container refers to a large number of items, but the end-user views just a few of them at a time through a small scrollable window. Rather than handling updates for the entire list, even though only a small part of it is on view, the client merely has to manage updates for the few items in the container window (which maps onto the display window).

7.8 Monitoring the connection

StreamLink can notify the client application of life cycle events relating to the connection between the client and the Liberator. This is done through an implementation of the **ConnectionListener** interface.

7.9 Reachability on mobile devices

On mobile devices the network may not always be available, and therefore a StreamLink app running on the device may not be able to reach the Liberator. If the app cannot connect because the network is not available, or loses its connection to the Liberator for the same reason, it should not try to reconnect but should effectively stop.

StreamLink for iOS automatically detects and handles changes to network availability. In both StreamLink Java and StreamLink Android, the **StreamLink** object has the **networkAvailable()** and **networkUnavailable()** methods that the app can call to inform StreamLink about changes to network availability.

For more information about how to handle changes to network availability in StreamLink Android apps,

see the **StreamLink Android API Documentation**.

7.10 Throttling

Using StreamLink you can limit the rate at which updates are sent to a client; this is called **throttling**. There are a number of reasons for doing this:

- To reduce network usage levels, both on leaving the Liberator and entering the client.
- To reduce the load on the client, for instance when too high a screen update rate would overload the client machine.
- To reduce the load on the Liberator, by reducing the rate at which it needs to send updates to its clients.

The amount of throttling is defined as a time interval. For example, if the throttle time for a data item is 1 second then Liberator will send an update for that item to subscribed clients at most every second. So if there are three updates to the item within a second, only the third one will be sent to the clients, at the end of the one second interval.

Decreasing the throttle time increases the maximum frequency of updates received by the client, whereas increasing the throttle time decreases the maximum frequency of received updates.

When Liberator identifies that a data item is updating less frequently than the throttle time, it does not activate throttling and sends the updates to the subscribed clients immediately.

Throttle times are set up in the Liberator's configuration; these are global settings which apply to all data items unless overridden for specific items. The configuration is defined as a set of throttle levels. For example, an item could have five throttle levels:

1. No throttling.
2. Throttling at 0.5 seconds.
3. Throttling at 1 second.
4. Throttling at 2 seconds.
5. The stopped state (no updates are forwarded to clients).

The Liberator can have a default throttle level applying to all data items when the client initially subscribes. This is typically the lowest level, but it could be set to one of the other levels. A client will start at the default throttling level when the user logs in and the client subscribes to data via StreamLink.

The client can then change the throttle level by sending a command to StreamLink. A client cannot set a custom throttle time, as the times are configured in the Liberator, but the client can move up or down a throttle level, go to the minimum or maximum level, or stop receiving updates altogether and subsequently resume them. A client can also alter the throttle level for a particular data item it has subscribed to, so the level is different to that applying to other items.

You can make throttle levels persistent, so that they are maintained when StreamLink reconnects to Liberator (by default throttle settings do not persist across connection failures); see Persistence 32.

Note: Liberator's throttling behavior is also known as “conflation” because several update values spread across time are effectively combined into one value (the latest one).
StreamLink also “conflates” messages it sends to Liberator, but this is a different sort of optimization, where multiple requests/discards are batched up into one message for transmission.

7.11 Authentication and permissioning

StreamLink provides a public interface called **CredentialsProvider** that is responsible for providing the credentials that Liberator requires to be sure that the user is allowed to log in. These credentials are often a username and password. StreamLink developers can implement this interface themselves to perform custom logic, such as integrating with a single sign on system, or retrieving a password from an external system.

The StreamLink library includes several sample implementations of **CredentialsProvider** depending on the technology in use:

- **PasswordCredentialsProvider**

  A basic implementation that provides user credentials consisting of a set username and password.

- **StandardKeyMasterCredentialsProvider**

  An implementation that attempts to retrieve a password token from Caplin KeyMaster. It can also optionally poll a specified URI on the KeyMaster server in order to keep the HTTP session alive.

- **AuthenticatingKeyMasterCredentialsProvider**

  An implementation that attempts to retrieve a password token from Caplin KeyMaster server that requires basic HTTP authentication. This implementation has the same behavior as **StandardKeyMasterCredentialsProvider** except that it attaches the user credentials to each web request that it makes.

Note: **StreamLink Silverlight**
AuthenticatingKeyMasterCredentialsProvider is not available in StreamLink Silverlight, because (at the time of publication) Silverlight does not allow web requests to include user credentials.

Permissions objects

A permissions object allows changes in user permissions on objects to be sent in real time between Permissioning Adapters, and between Liberator and client applications.

In StreamLink, subscribing to a permissions change is just like subscribing to a normal record update (see Subscribing to data and receiving updates 28), but the callback method to implement from **SubscriptionListener** is **onPermissionUpdate()**.

The client application will then receive the permission object, and any subsequent updates to it, and can use this information as appropriate. For example, the client could modify the way the application behaves according to changes in the permission object, such as enabling or denying trading on particular instruments.

For more information about permissions objects, see the **Caplin DataSource Overview**.

7.12 Logging

StreamLink includes a console logger class to aid application development. You can programmatically set the logging level and categories of items to be logged. At run time StreamLink will then display on the screen diagnostic information about its operation. Developers can implement custom logger classes, by implementing the **Logger** interface.

In StreamLink implementations other than StreamLink JS and StreamLink Silverlight, there is also a file logger that captures the diagnostic information in a file. This is primarily intended to help Caplin Support staff diagnose and fix StreamLink related problems encountered by customers.

7.13 Configuration

StreamLink is configured using a variety of definition formats, depending on the specific technology of each StreamLink SDK. The formats include JSON and Configuration Objects. The configuration allows you to define in detail the Liberators you wish to connect to, the connection types you would like to use, and other settings relevant to each particular technology.

For more information about configuring a particular StreamLink implementation, refer to the API documentation for the StreamLink SDK you are using.

7.14 Resilience, failover, and load balancing

StreamLink supports highly resilient operation by providing the ability to connect to alternative Liberator servers (failover) and by trying alternative types of RTTP connection. These capabilities are defined through configuration (see Configuration 40).

Configurable failover strategy

When a client's connection to a Liberator server fails (either because there is a persistent network failure or because the Liberator has failed), the client can connect to an alternative server according to a configurable failover scheme. StreamLink also uses this scheme when it first connects the client to a server.

The configuration technique is flexible and allows you to define sophisticated failover schemes. Liberator servers can be collected into 'ordered' groups which are nested within a 'balance' group.

'ordered' group

On first connection or during failover, StreamLink tries each of the servers or groups in an 'ordered' group in turn, in the order they have been declared within the configuration.

'balance' group

The 'ordered' groups within a 'balance' group are tried at random.

Example failover configuration (JSON syntax):

```
[
   //1st Group
   ["rttp://primary1.example.com", "rttp://backup1.example.com"],
   //2nd Group
   ["rttp://primary2.example.com", "rttp://backup2.example.com"]
]
```

This configuration defines two 'ordered' groups (“1st group” and “2nd group”) within an outer 'balance' group. This means that when StreamLink first tries to connect to a server, it randomly chooses between “ 1st group” and “2nd group”.

Each 'order' group defines a primary server (`"primaryN.example.com"`) and a backup server (`"backupN.example.com"`). Within the selected group StreamLink first attempts to connect to the primary server; if this fails, it attempts to connect to the backup server. If neither server can be reached, StreamLink attempts to connect to the servers in the other group.

So the sequence of servers to be tried is either

primary1 > backup1 > primary2 > backup2

or

primary2 > backup2 > primary1 > backup1

In a failover scenario, StreamLink first attempts to reconnect to the next server in the current 'order' group, followed by the servers in the other 'order' group. Once all possible connections have been tried, repeats the sequence of connection attempts after the configured reconnect delay.

Load balancing

'Balance' groups in the configuration allow you to implement server load balancing. If there are several servers (or server groups) within a 'balance' group, each client connects at random to one of the servers, so when there are many connected clients, the connections are distributed fairly evenly across the available servers.

A configuration that implements server load balancing would look like this:

Example: Server load balancing configuration (JSON syntax)

```
//Load balance across four servers

[
   ["rttp://liberator1.example.com"],
   ["rttp://liberator2.example.com"],
   ["rttp://liberator3.example.com"],
   ["rttp://liberator4.example.com"]
]
```

Alternative RTTP connection types

When a client attempts to connect to a Liberator, StreamLink refers to an ordered list of RTTP connection types 9. It tries each type of connection in sequence until one succeeds. If none of them succeed, StreamLink tries to connect to another Liberator, according to the configured failover scheme.

By default, StreamLink tries to connect via its default connection types, as defined by the browser or particular StreamLink technology. However, you can explicitly configure the connection types to try, as shown in the following example.

Example: Configuration defining connection types (JSON syntax)

```
[
   ["rttp://liberator1.example.com"],
   ["poll://liberator2.example.com",
    "http://liberator2.example.com",
    "ws://liberator2.example.com"]
]
```

The example shows that StreamLink is configured to connect to liberator1 using the default StreamLink connection types; the configuration specifies this using the generic connection type `rttp`. In contrast, when StreamLink attempts to connect to liberator 2, it first tries a Polling connection (`poll`), then an HTTP Streaming connection (`http`), and finally a WebSocket connection (`ws`).

8 Glossary of terms and acronyms

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

Term	Definition
Caplin Integration Suite (CIS)	A set of **APIs**, and tools for creating adapters that integrate the **Caplin Platform** with external systems.
Caplin Liberator	A financial internet hub that delivers data and messages in real time to and from subscribers over any network. StreamLink communicates with Liberator servers.
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.
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.
Failover	The transfer of operation from a hardware or software component that has failed to an alternative copy of the component, to ensure uninterrupted provision of service.
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**.
Snapshot	A request for all the data for a subject as at a single moment. See Obtaining data snapshots 30.

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

StreamLink 6.0: Overview, February 2013, Release 2