Enterprise Transport API
C# Edition
3.3.1.L1

VALUE ADDED COMPONENTS

LSEG DATA & ANALYTICS

Contents

List of Figures

List of Tables

1   Introduction

1.1   About this Manual

This document is authored by Enterprise Transport API architects and programmers who encountered and resolved many of the issues the reader might face. Several of its authors have designed, developed, and maintained the Enterprise Transport API product and other LSEG products which leverage it. As such, this document is concise and addresses realistic scenarios and use cases.

This guide documents the functionality and capabilities of the Enterprise Transport API C# Edition. In addition to connecting to itself, Tthe Enterprise Transport API can also connect to and leverage many different LSEG and customer components. If you want the Enterprise Transport API to interact with other components, consult that specific component's documentation to determine the best way to configure for optimal interaction.

This document explains the configuration parameters for the Enterprise Messaging API (simply called the Message API). Message API configuration is specified first via compiled-in configuration values, then via an optional user-provided XML configuration file, and finally via programmatic changes introduced via the software.

Configuration works in the same fashion across all platforms.

1.2   Audience

This manual provides information and examples that aid programmers using the Enterprise Transport API C# Edition. The level of material covered assumes that the reader is a user or a member of the programming staff involved in the design, coding, and test phases for applications which will use the Enterprise Transport API. It is assumed that the reader is familiar with the data types, classes, operational characteristics, and user requirements of real-time data delivery networks, and has experience developing products using the C# programming language in a networked environment.

This manual provides information that aids software developers and local site administrators in understanding Enterprise Transport API configuration parameters. You can obtain further information from the *Enterprise Message C# Edition API Developer's Guide*.

This document provides detailed yet supplemental information for application developers writing to the Enterprise Message API.

1.3   Programming Language

The Enterprise Transport API Components are written to the C, Java, and C# languages. This guide discusses concepts related to the C# Edition. All code samples in this document and all example applications provided with the product are written accordingly.

The Enterprise Message API is written using the C# programming language taking advantage of the object oriented approach to design and development of API and applications.

1.4 Acronyms and Abbreviations

ACRONYM / TERM	MEANING
ADH	LSEG Real-Time Advanced Distribution Hub is the horizontally scalable service component within the LSEG Real-Time Distribution System providing high availability for publication and contribution messaging, subscription management with optional persistence, conflation and delay capabilities.
ADS	LSEG Real-Time Advanced Distribution Server is the horizontally scalable distribution component within the LSEG Real-Time Distribution System providing highly available services for tailored streaming and snapshot data, publication and contribution messaging with optional persistence, conflation and delay capabilities.
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
DMM	Domain Message Model
Enterprise Message API	The Enterprise Message API (EMA) is an ease of use, open source, Open Message Model API. EMA is designed to provide clients rapid development of applications, minimizing lines of code and providing a broad range of flexibility. It provides flexible configuration with default values to simplify use and deployment. EMA is written on top of the Enterprise Transport API (ETA) utilizing the Value Added Reactor and Watchlist features of ETA.
Enterprise Transport API (ETA)	Enterprise Transport API is a high performance, low latency, foundation of the LSEG Real-Time SDK. It consists of transport, buffer management, compression, fragmentation and packing over each transport and encoders and decoders that implement the Open Message Model. Applications written to this layer achieve the highest throughput, lowest latency, low memory utilization, and low CPU utilization using a binary Rssl Wire Format when publishing or consuming content to/from LSEG Real-Time Distribution Systems.
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol (Secure)
JWK	JSON Web Key. Defined by RFC 7517, a JWK is a JSON formatted public or private key.
JWKS	JSON Web Key Set, This is a set of JWK, placed in a JSON array.
JWT	JSON Web Token. Defined by RFC 7519, JWT allows users to create a signed claim token that can be used to validate a user.
OMM	Open Message Model
QoS	Quality of Service
RDM	Domain Model
DP	Delivery Platform: this platform is used for REST interactions. In the context of Real-Time APIs, an API gets authentication tokens and/or queries Service Discovery to get a list of Real-Time - Optimized endpoints using DP.
LSEG Real-Time Distribution System	LSEG Real-Time Distribution System is LSEG's financial market data distribution platform. It consists of the LSEG Real-Time Advanced Distribution Server and LSEG Real-Time Advanced Distribution Hub. Applications written to the LSEG Real-Time SDK can connect to this distribution system.
Reactor	The Reactor is a low-level, open-source, easy-to-use layer above the Enterprise Transport API. It offers heartbeat management, connection and item recovery, and many other features to help simplify application code for users.
RFA	Robust Foundation API
RMTES	A multi-lingual text encoding standard
RSSL	Source Sink Library

Table 1: Acronyms and Abbreviations

ACRONYM / TERM	MEANING
RTT	Round Trip Time, this definition is used for round trip latency monitoring feature.
RWF	Rssl Wire Format, an LSEG proprietary binary format for data representation.
SOA	Service Oriented Architecture
SSL	Sink Source Library
UML	Unified Modeling Language
UTF-8	8-bit Unicode Transformation Format

Table 1: Acronyms and Abbreviations

1.5 Additional References and Resources

- Enterprise Transport API C# Edition *LSEG Domain Model Usage Guide*

- *API Concepts Guide*

- *Enterprise Transport API C# Edition Configuration Guide*

- Enterprise Transport API *ANSI Library Reference Manuals*

- Enterprise Transport API C# Edition*Value Added Components Developers Guide*

- Enterprise Transport API C# Edition *Developers Guide*

- The LSEG Developer Community

-

1.6 Documentation Feedback

While we make every effort to ensure the documentation is accurate and up-to-date, if you notice any errors, or would like to see more details on a particular topic, you have the following options:

- Send us your comments via email at ProductDocumentation@lseg.com.

- Add your comments to the PDF using Adobe's **Comment** feature. After adding your comments, submit the entire PDF to LSEG by clicking **Send File** in the **File** menu. Use the ProductDocumentation@lseg.com address.

1.7 Document Conventions

- Typographic
- Document Structure
- Diagrams

1.7.1 Optional, Conditional, and Required

Throughout this manual, all parameters, options, functions, Structure_ObjectVARIABLEs, flags, etc., are considered optional unless explicitly marked as **Conditional** or **Required**. If marked as conditional, the item's description will describe the conditions surrounding its use.

1.7.2 Typographic

This document uses the following types of conventions:

- C# Structuresclasses, methods, in-line code snippets, and types are shown in `Courier New` font.
- Parameters, filenames, tools, utilities, and directories are shown in **Bold** font.
- Document titles and variable values are shown in *italics*.
- When initially introduced, concepts are shown in ***Bold, Italics***.
- Longer code examples are shown in Courier New font against a gray background. For example:
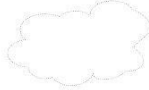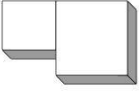
```
// decode contents into the filter list structure
        CodecReturnCode ret = fieldList.Decode(dIter, null);
        if (ret != CodecReturnCode.SUCCESS)
        {
            Console.WriteLine("DecodeFieldList() failed with return code: " + ret);
            return ret;
        }
consumer = new(new OmmConsumerConfig().OperationModel(OperationModelMode.USER_DISPATCH)
                                    .Host("localhost:14002").UserName("user"));
        consumer.RegisterClient(new RequestMsg().ServiceName("DIRECT_FEED").Name("IBM.N"),
                new AppClient(), 0);
        var endTime = System.DateTime.Now + TimeSpan.FromMilliseconds(60000);
        while(System.DateTime.Now < endTime)
        {
            consumer.Dispatch(10);      // calls to OnRefreshMsg(), OnUpdateMsg(), or
                OnStatusMsg() execute on this thread
        }
```

1.7.3 Document Structure

- General Concepts
- Detailed Concepts
- Interface Definitions
- Example Code
-

1.7.4   Diagrams

Diagrams that depict the interaction between components on a network use the following notation:

	Feed Handler, Real-Time server, or other application		Network of multiple servers
	Enterprise Transport API application		Point-to-point connection showing direction of primary data flow
	Application with local daemon		Point-to-point connection showing direction of client connecting to server
	Multicast network		Data from external source (e.g. consolidated network or exchange)
	Connection to Multicast network, no primary data flow direction		Connection to Multicast network showing direction of primary data flow

Figure 1. Network Diagram Notation

	Object
	Inheritance: object on left is like object on right
	Composition: object on left is made up of some number of objects on right
	Composition: object on left is made up of one object on right

Figure 2. UML Diagram Notation

2   Product Description and Overview

2.1   What is the Enterprise Transport API?

The Enterprise Transport API is a low-level Enterprise Transport API that provides the most flexible development environment to the application developer. It is the foundation on which all LSEG OMM-based components are built. The Enterprise Transport API allows applications to achieve the highest throughput and lowest latency available with any OMM API, but requires applications to perform all message encoding/decoding and manage all aspects of network connectivity. The Enterprise Enterprise Transport API, Enterprise Message API, and the Robust Foundation API make up the set of OMM API offerings.
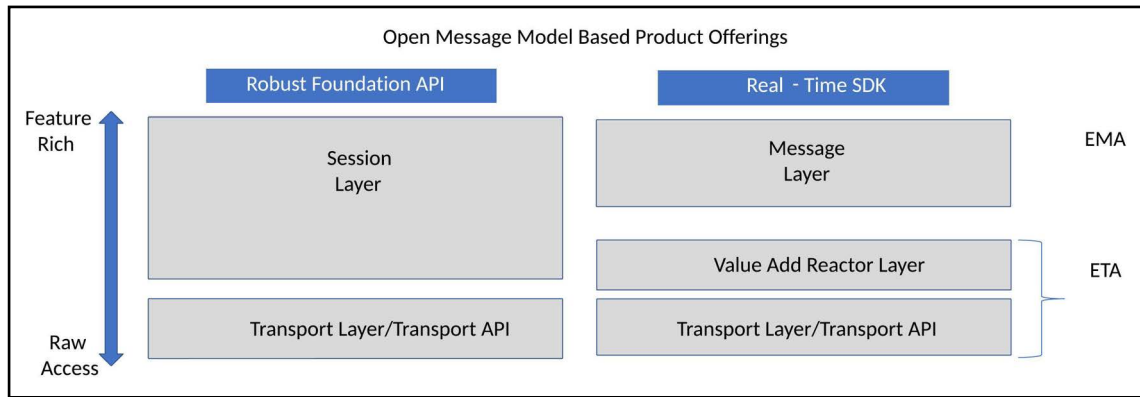
Figure 3. OMM APIs with Value Added Components

The Enterprise Transport API Value Added Components provide alternate entry points for applications to leverage OMM-based APIs with more ease and simplicity. These optional components help to offload much of the connection management code and perform encoding and decoding of some key OMM domain representations. Unlike older domain-based APIs that lock the user into capabilities or ease-of-use into the highest layer of API, Value Added components are independently implemented for use with the Enterprise Transport API and Robust Foundation API in their native languages (Example: Enterprise Transport API in C and Java, Robust Foundation API in C++ and Java). These implementations are then shipped with their respective API products as options for the application developer that may want these additional capabilities.

2.3   Transport API Reactor

The Enterprise Transport API Reactor is a connection management and event processing component that can significantly reduce the amount of code an application must write to leverage OMM in its own functions and to connect to other OMM-based devices. Consumer, interactive provider, and non-interactive provider applications can use the reactor and leverage it in managing consumer and non-interactive provider start-up processes, including user log in, source directory establishment, and dictionary download. The reactor also supports dispatching of events to user-implemented callback functions. In addition, it handles the flushing of user-written content and manages network pings on the user's behalf. The connection recovery feature allows the reactor to automatically recover from disconnects. Value Added domain representations are coupled with the reactor, allowing domain specific callbacks to be presented with their respective domain representation for easier, more logical access to content. For more information, refer to Chapter 6. This component depends on the Value Added Administration Domain Model Representation component, the Value Added Utilities, Enterprise Transport API C# Transport Package, and Enterprise Transport API C# Codec Package.

2.4   OMM Consumer Watchlist

The `Reactor` features a per-channel watchlist that provides a wealth of functionality for OMM consumer applications. The watchlist automatically performs various recovery behaviors for which developers would normally need to account.

The watchlist supports consuming from TCP-based connections (**ConnectionType.SOCKET**, etc.).

For details on configuring the `Reactor` to enable the consumer watchlist, refer to Section 6.3.2.

2.4.1   Data Stream Aggregation and Recovery

The watchlist automatically recovers data streams in response to failure conditions, such as disconnects and unavailable services, so that applications do not need special handling for these conditions. As conditions are resolved, the watchlist will re-request items on the application's behalf. Applications can also use this method to request data before a connection is fully established.

To recover from disconnects using a watchlist, enable the reactor's connection recovery. Options to reconnect disconnected channels are detailed in Section 6.4.1.2.

For efficient bandwidth usage, the watchlist also combines multiple requests for the same item into a single stream and forwards response messages to each requested stream as appropriate.

2.4.2   Additional Features

The watchlist provides additional features for convenience:

- Group and Service Status Fanout: The `Reactor` maintains a directory stream to receive service updates. As group status messages or service status messages are received, the `Reactor` forwards the status to all affected streams via `StatusMsgs`.

- Quality of Service Range Matching: The `Reactor` will accept and aggregate item requests that specify a range of `Qos`, or requests that do not specify a `Qos`. After comparing these requests with the quality of service from the providing service, the watchlist uses the best matching quality of service.

- Support for Enhanced Symbol List Behaviors: The `Reactor` supports data streams when requesting a Symbol List item. For details on requesting Symbol list data streams, refer to the Enterprise Transport API C# Edition *LSEG Domain Model Usage Guide*.

- Support for Batch Requests: The `Reactor` will accept batch requests regardless of whether the connected provider supports them.

2.4.3   Usage Notes

Applications should note the following when enabling the watchlist:

- The application must use the **ReactorSubmitMsg** and **ReactorChannel.submit(MsgBase)** methods to send messages. It cannot use **ReactorChannel.submit(TransportBuffer)**.

- Only one login stream should be opened per **ReactorChannel**.

- To prevent unnecessary bandwidth use, the watchlist will not recover a dictionary request after a complete refresh is received.

- As private streams are intended for content delivery between two specific points, the watchlist does not aggregate nor recover them.

- The **OMMConsumerRole.dictionaryDownloadMode** option is not supported when the watchlist is enabled.

2.5   Administration Domain Model Representations

The *Administration Domain Model Representations* are RDM-specific representations of the OMM administrative domain models. This Value Added Component contains classes and interfaces that represent the messages within the Login, Source Directory, and Dictionary domains. All classes follow the formatting and naming specified in the Enterprise Transport API C# Edition *LSEG Domain Model Usage Guide*, so access to content is logical and specific to the content being represented. This component also handles all encoding and decoding functionality for these domain models, so the application needs only to manipulate the message's structure members to send or receive this content. This not only significantly reduces the amount of code an application needs to interact with OMM devices (i.e., LSEG Real-Time Distribution System infrastructure), but also ensures that encoding/decoding for these domain models follow OMM-specified formatting rules. Applications can use this Value Added Component directly to help with encoding, decoding, and representation of these domain models. When using the Enterprise Transport API Reactor, this component is embedded to manage and present callbacks with a domain-specific representation of content. For more information, refer to Chapter 8. This component depends on the Enterprise Transport API C# Codec Package.

2.6   Value Added Utilities

The Value Added Utilities are a collection of common classes, mainly used by the Enterprise Transport API Reactor. For example, Value Added Utilities include a simple queue along with iterable and concurrent versions of it.

3   Building an OMM Consumer

3.1   Overview

This chapter provides an overview of how to create an OMM consumer application using the ETA Reactor and Administration Domain Model Representation Value Added Components. The Value Added Components simplify the work done by an OMM consumer application when establishing a connection to other OMM interactive provider applications, including LSEG Real-Time Distribution System, Data Feed Direct, and Real-Time — Optimized. After the Reactor indicates that the connection is ready, an OMM consumer can then consume (i.e., send data requests and receive responses) and publish data (i.e., post data).

The general process can be summarized by the following steps.

- Leverage existing or create new **`Reactor`**

- Implement callbacks and populate role

- Establish connection using **`Reactor.Connect()`**

- Issue requests and/or post information

- Log out and shut down

The **VAConsumer** example application, included with the ETA product, provides one implementation of an OMM consumer application that uses the ETA Value Added Components. The application is written with simplicity in mind and demonstrates usage of the ETA and ETA Value Added Components. Portions of functionality have been abstracted and can easily be reused, though you might need to modify it to achieve your own unique performance and functionality goals.

3.2   Leverage Existing or Create New Reactor

The **`Reactor`** can manage one or multiple **`ReactorChannel`** structures. This functionality allows the application to associate OMM consumer connections with an existing **`Reactor`**, having it manage more than one connection, or to create a new **`Reactor`** to use with the connection.

To create a new **`Reactor`**, the application must use the static method **`CreateReactor`** of the **`Reactor`** class. This will create any necessary memory and threads that the **`Reactor`** uses to manage **`ReactorChannel`**s and their content flow. If the application is using an existing **`Reactor`**, there is nothing additional to do.

For more information about the **`Reactor`** and its creation, refer to Section 6.2.1.

3.3   Implement Callbacks and Populate Role

Before creating the OMM consumer connection, the application needs to specify callback functions to use for all inbound content. The callback functions are specified on a per **ReactorChannel** basis so each channel can have its own unique callback functions or existing callback functions can be specified and shared across multiple **ReactorChannel**s.

Use of a **Reactor** requires the use of several callback functions. The application must have the following:

- **IReactorChannelEventCallback**, which returns information about the **ReactorChannel** and its state (e.g., connection up)

- **IDefaultMsgCallback**, which processes all data not handled by other optional callbacks.

In addition to the required callbacks, an OMM consumer can specify several administrative domain-specific callback functions. Available domain-specific callbacks include:

- **IRDMLoginMsgCallback**, which processes all data for the RDM Login domain.

- **IDirectoryMsgCallback**, which processes all data for the RDM Source Directory domain.

- **IDictionaryMsgCallback**, which processes all data for the RDM Dictionary domain.

The **ConsumerRole** class should be populated with all callback information for the **ReactorChannel**.

The **ConsumerRole** allows the application to provide login, directory, and dictionary request information. This can be initialized with default information. The callback functions are specified on the **ConsumerRole** class or with specific information according to the application and user. The **Reactor** will use this information when starting up the **ReactorChannel**.

For more information about the **ConsumerRole**, refer to Section 6.3.1. For information about the various callback functions and their specifications, refer to Section 6.5.2.

3.4   Establish Connection using Reactor.Connect

After populating the **ConsumerRole**, the application can use **Reactor.Connect()** to create a new outbound connection. **Reactor.Connect()** will create an OMM consumer-type connection using the provided configuration and role information.

After establishing the underlying connection, a channel event is returned to the application's **IReactorChannelEventCallback**; this provides the **ReactorChannel** and the state of the current connection. At this point, the application can begin using the **Reactor.Dispatch** function to dispatch directly on this **ReactorChannel**, or use **Reactor.Dispatch** to dispatch across all channels associated with the **Reactor**.

The **Reactor** will use the login, directory, and dictionary information specified on the **ConsumerRole** to perform all channel initialization for the user. After a user has logged in, received a source directory response, and downloaded field dictionaries, a channel event is returned to inform the application that the connection is ready.

The **Reactor.Connect()** function is described in Section 6.4.1.1. Dispatching is described in Section 6.5.

3.5   Issue Requests and/or Post Information

After the **ReactorChannel** is established, the channel can be used to request additional content. When issuing the request, the consuming application can use the **serviceId** of the desired service, along with the stream's identifying information. Requests can be sent for any domain using the formats defined in that domain model specification. Domains provided by LSEG are defined in the Enterprise Transport API C# Edition *LSEG Domain Model Usage Guide*. This content will be returned to the application via the **IDefaultMsgCallback**.

At this point, an OMM consumer application can also post information or forward generic messages to capable provider applications. All content requested, received, or posted is encoded and decoded using the ETA Codec Package described in the Enterprise Transport API C# Edition *Developers Guide*.

3.6   Log Out and Shut Down

When the consumer application is done retrieving, forwarding, or posting content, the consumer can close the `ReactorChannel` by calling `ReactorChannel.Close`. This will close all item streams and log out the user. Prior to closing the `ReactorChannel`, the application should release any unwritten pool buffers to ensure proper memory cleanup.

If the application is done with the `Reactor`, the `Reactor.Shutdown` function can be used to shutdown and clean up any `Reactor` resources.

- Closing a `ReactorChannel` is described in Section 6.4.3.

- Shutting down a `Reactor` is described in Section 6.2.2.

3.7   Additional Consumer Details

The following locations provide specific details about using OMM consumers, the ETA, and ETA Value Added Components:

- The **Consumer** application demonstrates one way of implementing of an OMM consumer application that uses ETA Value Added Components. The application's source code and Java documentation contain additional information about specific implementation and behaviors.

- Chapter 6 provides a detailed look at the ETA Reactor.

- Chapter 8 provides more information about the Administration Domain Model Representations.

- The Enterprise Transport API C# Edition *Developers Guide* provides specific ETA encoder/decoder and transport usage information.

- The Enterprise Transport API C# Edition *LSEG Domain Model Usage Guide* provides specific information about the Domain Message Models used by this application type.

4   Building an OMM Interactive Provider

4.1   Overview

This chapter provides a high-level description of how to create an OMM interactive provider application using the ETA Reactor and Administration Domain Model Representation Value Added Components. An OMM interactive provider application opens a listening socket on a well-known port allowing OMM consumer applications to connect. The ETA Value Added Components simplify the work done by an OMM interactive provider application when accepting connections and handling requests from OMM consumers.

The following steps summarize this process:

- Leverage an existing `Reactor`, or create a new one

- Create an `Server`

- Implement callbacks and populate role

- Associate incoming connections using `Reactor.Accept`

- Perform login process

- Provide source directory information

- Provide necessary dictionaries

- Handle requests and post messages

- Dispatch Round Trip Time messages

- Disconnect consumers and shut down

Included with the ETA product, the **Provider** example application provides one way of implementing an OMM interactive provider application that uses the ETA Value Added Components. The application is written with simplicity in mind and demonstrates the use of the ETA and ETA Value Added Components. Portions of the functionality are abstracted for easy reuse, though you might need to customize it to achieve your own unique performance and functionality goals.

4.2   Leverage Existing or Create New Reactor

The `Reactor` can manage one or multiple `ReactorChannel` classs. This allows the application to choose to associate OMM provider connections with an existing `Reactor`, have it manage more than one connection, or create a new `Reactor` to use with the connection.

If the application is creating a new `Reactor`, the `Reactor.CreateReactor` method is used. This will create any necessary memory and threads that the `Reactor` uses to manage `ReactorChannel`s and their content flow. If the application is using an existing `Reactor`, there is nothing additional to do.

Detailed information about the `Reactor` and its creation are available in Section 6.2.1.

4.3   Create a Server

The first step of any ETA Interactive Provider application is to establish a listening socket, usually on a well-known port so that consumer applications can easily connect. The provider uses the `Transport.Bind` method to open the port and listen for incoming connection attempts. This uses the standard ETA Transport functionality described in the Enterprise Transport API C# Edition *Developers Guide*.

Whenever an OMM consumer application attempts to connect, the provider will use the `Server` and associate the incoming connections with a `Reactor`, which will accept the connection and perform any initialization as described in Section 4.4 and Section 4.5.

4.4   Implement Callbacks and Populate Role

Before accepting an incoming connection with an OMM provider, the application needs to specify callback methods to use for all inbound content. Callback methods are specified on a per `ReactorChannel` basis so each channel can have its own unique callback methods or existing callback methods can be specified and shared across multiple `ReactorChannel`s.

The following callback methods are required for use with a `Reactor`:

- `IReactorChannelEventCallback`, which returns information about the `ReactorChannel` and its state (e.g., connection up)

- `IDefaultMsgCallback`, which processes all data not handled by other optional callbacks.

In addition to the required callbacks, an OMM provider can specify several administrative domain-specific callback methods. Available domain-specific callbacks are:

- `IRDMLoginMsgCallback`, which processes all data for the RDM Login domain.

- `IDirectoryMsgCallback`, which processes all data for the RDM Source Directory domain.

- `IDictionaryMsgCallback`, which processes all data for the RDM Dictionary domain.

The `ProviderRole` class should be populated with all callback information for the `ReactorChannel`.

Detailed information about the `ProviderRole` is in Section 6.3.1. Information about the various callback methods and their specifications are available in Section 6.6.2.

4.5   Associate Incoming Connections Using Reactor.Accept

After the `ProviderRole` is populated, the application can use `Reactor.Accept` to accept a new inbound connection. `Reactor.Accept` will accept an OMM provider connection from the passed-in `IServer` using provided configuration and role information.

When the underlying connection is established, a channel event is returned to the application's `ReactorChannelEventCallback`; this will provide the `ReactorChannel` and indicate the current connection state. At this point, the application can begin using the `Reactor.Dispatch` method to dispatch directly on this `ReactorChannel`, or continue using `Reactor.Dispatch` to dispatch across all channels associated with the `Reactor`.

The `Reactor` will perform all channel initialization and pass any administrative domain information to the application via the callbacks specified with the `ProviderRole`.

- For more details on the `Reactor.Accept` function, refer to Section 6.4.1.6.

- For more details on dispatching, refer to Section 6.6.

4.6   Perform Login Process

Applications authenticate with one another using the Login domain model. An OMM interactive provider must handle consumer Login request messages and supply appropriate responses. Login information will be provided to the application via the `IRDMLoginMsgCallback`, when specified on the `ProviderRole`.

After receiving a Login request, an interactive provider can perform any necessary authentication and permissioning.

- If the interactive provider grants access, it should send an `LoginRefresh` to convey that the user successfully connected. This message should indicate the feature set supported by the provider application.

- If the interactive provider denies access, it should send an `LoginStatus`, closing the connection and informing the user of the reason for denial.

Login messages can be encoded and decoded using themessages' `Encode` and `Decode` functions. More details and code examples are in Section 8.2.

All content requested, received, or posted is encoded and decoded using the ETA Codec Package described in the Enterprise Transport API C# Edition *Developers Guide*.

Information about the Login domain and expected content formatting is available in the Enterprise Transport API C# Edition *LSEG Domain Model Usage Guide*.

4.7   Provide Source Directory Information

The Source Directory domain model conveys information about all available services in the system. An OMM consumer typically requests a Source Directory to retrieve information about available services and their capabilities. This includes information about supported domain types, the service's state, the quality of service, and any item group information associated with the service. LSEG recommends that at a minimum, an interactive provider supply the Info, State, and Group filters for the Source Directory.

- The Source Directory Info filter contains the name and `ServiceId` for each available service. The interactive provider should populate the filter with information specific to the services it provides.

- The Source Directory State filter contains status information for the service informing the consumer whether the service is Up (available), or Down (unavailable).

- The Source Directory Group filter conveys item group status information, including information about group states, as well as the merging of groups. If a provider determines that a group of items is no longer available, it can convey this information by sending either individual item status messages (for each affected stream) or a Directory message containing the item group status information. Additional information about item groups is available in the Enterprise Transport API C# Edition *Developers Guide*.

Login messages can be encoded and decoded using the messages' `Encode` and `Decode` methods. More details and code examples are in Section 8.4.

All content requested, received, or posted is encoded and decoded using the ETA Codec Package described in the Enterprise Transport API C# Edition *Developers Guide*.

Information about the Source Directory domain and expected content formatting is available in the Enterprise Transport API C# Edition *LSEG Domain Model Usage Guide*

4.8   Provide or Download Necessary Dictionaries

Some data requires the use of a dictionary for encoding or decoding. The dictionary typically defines type and formatting information, and tells the application how to encode or decode information. Content that uses the `FieldList` type requires the use of a field dictionary (usually the **RDMFieldDictionary**, though it can instead be a user-defined or modified field dictionary).

The Source Directory message should notify the consumer about dictionaries needed to decode content sent by the provider. If the consumer needs a dictionary to decode content, it is ideal that the interactive provider application also make this dictionary available to consumers for download. The provider can inform the consumer whether the dictionary is available via the Source Directory.

If consuming from an LSEG Real-Time Advanced Distribution Hub and providing content downstream, a provider application can also download the RWFFld and RWFEnum dictionaries. Using these dictionaries, the ETA can retrieve appropriate dictionary information for providing field list content. A provider can use this feature to ensure they are using the appropriate version of the dictionary or to encode data. An LSEG Real-Time Advanced Distribution Hub that supports provider dictionary downloads sends a Login request message containing the **SupportProviderDictionaryDownload** login element. The ETA sends the dictionary request using the Dictionary domain model. For details on using the Login domain and expected message content, refer to the Enterprise Transport API C# Edition *LSEG Domain Model Usage Guide*.

Dictionary messages can be encoded and decoded using the messages' `Encode` and `Decode` methods. More details and code examples are in Section 8.5. Dictionary requests will be provided via the `IDictionaryMsgCallback`, when specified on the `ProviderRole`.

Whether loading a dictionary from file or requesting it from an LSEG Real-Time Advanced Distribution Hub, the ETA offers several utility functions for loading, downloading, and managing a properly-formatted field dictionary. The ETA also has utility functions that help the provider encode into an appropriate format for downloading or decoding downloaded dictionaries.

- All content requested, received, or posted is encoded and decoded using the ETA Codec Package described in the Enterprise Transport API C# Edition *Developers Guide*.

- Information about the Dictionary domain, dictionary utility functions, and expected content formatting is available in the Enterprise Transport API C# Edition *LSEG Domain Model Usage Guide.*

4.9   Handle Requests and Post Messages

A provider can receive a request for any domain, though this should typically be limited to the domain capabilities indicated in the Source Directory. When a request is received, the provider application must determine if it can satisfy the request by:

- Comparing `MsgKey` identification information

- Determining whether it can provide the requested quality of service

- Ensuring that the consumer does not already have a stream open for the requested information

If a provider can service a request, it should send appropriate responses. However, if the provider cannot satisfy the request, the provider should send an `StatusMsg` to indicate the reason and close the stream. All requests and responses should follow specific formatting as defined in the domain model specification. The Enterprise Transport API C# Edition *LSEG Domain Model Usage Guide* defines all domains provided by LSEG. This content will be returned to the application via the `IDefaultMsgCallback`.

The provider can specify that it supports post messages via the `LoginRefresh`. If a provider application receives a post message, the provider should determine the correct handling for the post. This depends on the application's role in the system and might involve storing the post in its cache or passing it farther up into the system. If the provider is the destination for the post, the provider should send any requested acknowledgments, following the guidelines described in the Enterprise Transport API C# Edition *Developers Guide*. Any posted content will be returned to the application via the `IDefaultMsgCallback`.

All content requested, received, or posted is encoded and decoded using the ETA Codec PackagePackage as described in the Enterprise Transport API C# Edition *Developers Guide*.

4.10   Dispatch Round Trip Time Messages

Optionally, a provider can send a Round Trip Time message to gather Round Trip Time statistics. While the ETA does not regulate rules for implementing the Round Trip Time message, the ETA provides several examples for applying this feature in provider applications. Generally, if the provider wants to support the Round Trip Time feature, the provider must provide methods for sending generic Round Trip Time messages to a consumer and extend callback methods for Round Trip Time calculation.

For detailed information, refer to the Enterprise Transport API C# Edition *RDM Usage Guide*.

4.11   Disconnect Consumers and Shut Down

If the `Reactor` application must shut down, it can either leave consumer connections intact or shut them down. If the provider decides to close consumer connections, the provider should send an `StatusMsg` on each connection's Login stream closing the stream. At this point, the consumer should assume that its other open streams are also closed.

It can then close the `ReactorChannel`s by calling `ReactorChannel.Close`. Prior to closing the `ReactorChannel`, the application should release any unwritten pool buffers to ensure proper memory cleanup.

If the application is done with the `Reactor`, the `Reactor.Shutdown` method can be used to shutdown and cleanup any `Reactor` resources.

- Closing a `ReactorChannel` is described in Section 6.4.3.

- Shutting down a `Reactor` is described in Section 6.2.2.

4.12   Additional Interactive Provider Details

For specific details about OMM interactive providers, the ETA, and ETA Value Added Component use, refer to the following locations:

- The **VAProvider** application demonstrates one implementation of an OMM interactive provider application that uses ETA Value Added Components. The application's source code and have additional information about specific implementation and behaviors.

- Chapter 6 provides a detailed look at the ETA Reactor.

- Chapter 8 provides more information about the Administration Domain Model Representations.

- The Enterprise Transport API C# Edition *Developers Guide* provides specific ETA encoder/decoder and transport usage information.

- The Enterprise Transport API C# Edition *LSEG Domain Model Usage Guide* provides specific information about the Domain Message Models used by this application type.

5   Building an OMM Non-Interactive Provider

5.1   Building an OMM Non-Interactive Provider Overview

This chapter provides an overview of how to create an OMM non-interactive provider application using the ETA Reactor and Administration Domain Model Representation Value Added Components. The Value Added Components simplify the work done by an OMM non-interactive provider application when establishing a connection to an LSEG Real-Time Advanced Distribution Hub. After the reactor indicates that the connection is ready, an OMM non-interactive provider can publish information into the LSEG Real-Time Advanced Distribution Hub cache without needing to handle requests for the information. The LSEG Real-Time Advanced Distribution Hub and other LSEG Real-Time Distribution System components can cache the information and provide it to any OMM consumer applications that indicate interest.

The general process can be summarized by the following steps.

- Leverage existing or create new **Reactor**

- Implement callbacks and populate role

- Establish connection using **ReactorConnect**

- Perform dictionary download

- Provide content

- Log out and shut down

The **NIProvider** example application, included with the ETA product, provides one implementation of an OMM non-interactive provider application that uses the ETA Value Added Components. The application is written with simplicity in mind and demonstrates usage of the ETA and ETA Value Added Components. Portions of functionality have been abstracted and can easily be reused, though you might need to modify it to achieve your own unique performance and functionality goals.

5.2   Leverage Existing or Create New Reactor

The **Reactor** can manage one or multiple **ReactorChannel** classs. This allows the application to choose to associate OMM non-interactive provider connections with an existing **Reactor**, having it manage more than one connection, or to create a new **Reactor** to use with the connection.

If the application is creating a new **Reactor**, the **CreateReactor** method is used. This will create any necessary memory and threads that the **Reactor** uses to manage **ReactorChannel** and their content flow. If the application is using an existing **Reactor**, there is nothing more to do.

Detailed information about the **Reactor** and its creation are available in Section 6.2.1.

5.3   Implement Callbacks and Populate Role

Before creating the OMM non-interactive provider connection, the application needs to specify callback methods to use for all inbound content. Callback methods are specified on a per **ReactorChannel** basis so each channel can have its own unique callback methods or existing callback methods can be specified and shared across multiple **ReactorChannel**s.

A **Reactor** requires the use of the following callback methods:

- **IReactorChannelEventCallback**, which returns information about the **ReactorChannel** and its state (e.g., connection up)

- **IDefaultMsgCallback**, which processes all data not handled by other optional callbacks.

Additionally, an OMM non-interactive provider can specify the administrative domain-specific callback method **IRDMLoginMsgCallback**, which processes all data for the RDM Login domain.

The **NIProviderRole** class should be populated with all callback information for the **ReactorChannel**. **NIProviderRole** allows the application to provide login request and initial directory refresh information. This can be initialized with default information. Callback methods are specified on the **NIProviderRole** class or with specific information according to the application and user. The **Reactor** will use this information when starting up the **ReactorChannel**.

- For detailed information on the **NIProviderRole**, refer to Section 6.3.1.

- For information on the various callback methods and their specifications, refer to Section 6.5.2.

5.4   Establish Connection using Reactor.Connect

After populating the **NIProviderRole**, the application can use **Reactor.Connect** to create a new outbound connection. **Reactor.Connect** will create an OMM non-interactive provider type connection using the provided configuration and role information.

When the underlying connection is established, a channel event will be returned to the application's **IReactorChannelEventCallback**, which provides the **ReactorChannel** and indicates the current connection state. At this point, the application can begin using the **Reactor.Dispatch** method to dispatch directly on this **ReactorChannel**, or use **Reactor.Dispatch** to dispatch across all channels associated with the **Reactor**.

The **Reactor** will use the login and directory information specified on the **NIProviderRole** to perform all channel initialization for the user. After the user is logged in and has sent a source directory response, a channel event is returned to inform the application that the connection is ready.

- For further details on the **Reactor.Connect** method, refer to Section 6.4.1.1.

- For further details on dispatching, refer to Section 6.5.

5.5   Download the Dictionary

If connected to a supporting LSEG Real-Time Advanced Distribution Hub, an OMM non-interactive provider can download the RWFFld and RWFEnum dictionaries to retrieve the appropriate dictionary information for providing field list content. An OMM non-interactive provider can use this feature to ensure they use the appropriate version of the dictionary or to encode data. To support the Provider Dictionary Download feature, the LSEG Real-Time Advanced Distribution Hub sends a Login response message containing the **SupportProviderDictionaryDownload** login element. The dictionary request is sent using the Dictionary domain model.

The ETA offers several utility functions for downloading and managing a properly-formatted field dictionary. The provider can also use utility functions to encode the dictionary into an appropriate format for downloading or decoding.

For details on using the Login domain, expected message content, and available dictionary utility functions, refer to the Enterprise Transport API C# Edition *LSEG Domain Model Usage Guide*.

5.6   Provide Content

After the **ReactorChannel** is established, it can begin pushing content to the LSEG Real-Time Advanced Distribution Hub. Each unique information stream should begin with an **RefreshMsg**, conveying all necessary identification information for the content. Because the provider instantiates this information, a negative value **streamId** should be used for all streams. The initial identifying refresh can be followed by other status or update messages.

All content is encoded and decoded using the ETA C# Codec Package described in the Enterprise Transport API C# Edition *Developers Guide*.

5.7   Log Out and Shut Down

When the Consumer application is done retrieving or posting content, it can close the **ReactorChannel** by calling **ReactorChannel.Close**. This will close all item streams and log out the user. Prior to closing the **ReactorChannel**, the application should release any unwritten pool buffers to ensure proper memory cleanup.

If the application is done with the **Reactor**, the **Reactor.Shutdown** method can be used to shutdown and cleanup any **Reactor** resources.

- For details on closing a **ReactorChannel**, refer to Section 6.4.3.

- Shutting down a **Reactor** is described in Section 6.2.2.

5.8   Additional Non-Interactive Provider Details

The following locations discuss specific details about using OMM non-interactive providers and the ETA:

- The **VANIProvider** application demonstrates one implementation of an OMM non-interactive provider application that uses ETA Value Added Components. The application's source code and have additional information about the specific implementation and behaviors.

- provides a detailed look at the ETA Reactor.

- provides more information about Administration Domain Model Representations.

- The Enterprise Transport API C# Edition *Developers Guide* provides specific ETA encoder/decoder and transport usage information.

- The Enterprise Transport API C# Edition *LSEG Domain Model Usage Guide* provides specific information about the Domain Message Models used by this application type.

6   Reactor Detailed View

6.1   Concepts

The **ETA Reactor** is a connection management and event processing component that can significantly reduce the amount of code an application must write to leverage OMM. This component helps simplify many aspects of a typical ETA application, regardless of whether the application is an OMM consumer, OMM interactive provider, or OMM non-interactive provider. The ETA Reactor can help manage consumer and non-interactive provider start up processing, including user log in, source directory establishment, and dictionary download. It also allows for dispatching of events to user-implemented callback functions, handles flushing of user-written content, and manages network pings on the user's behalf. Value Added domain representations are coupled with the reactor, allowing domain-specific callbacks to be presented with their respective domain representation for easier, more logical access to content. For a list and comparison of ETA and ETA Reactor functionalities, refer to Section 6.1.1.

The ETA Reactor internally depends on the Administration Domain Model Representation component. This allows the user to provide and consume the administrative RDM types in a more logical format. This additionally hides encoding and decoding of these domains from the Reactor user, all interaction is via a simple structural representation. More information about the Administration Domain Model Representation value added component is available in Chapter 8. The ETA Reactor also leverages several utility components, contained in the Value Added Utilities. This includes an iterable queue and a selectable bidirectional queue used to communicate events between the Reactor and Worker threads.

The ETA Reactor helps to manage the life-cycle of a connection on the user's behalf. When a channel is associated with a reactor, the reactor performs all necessary transport level initialization and alerts the user, via a callback, when the connection is up, ready for use, or is down. An application can simultaneously run multiple unique reactor instances, where each reactor instance can associate and manage a single channel or multiple channels. This functionality allows users to quickly and easily horizontally scale their application to leverage multi-core systems or distribute content across multiple connections.

Each instance of the ETA Reactor leverages multiple threads to help manage inbound and outbound data efficiently. The following figure illustrates a high-level view of the reactor threading model.
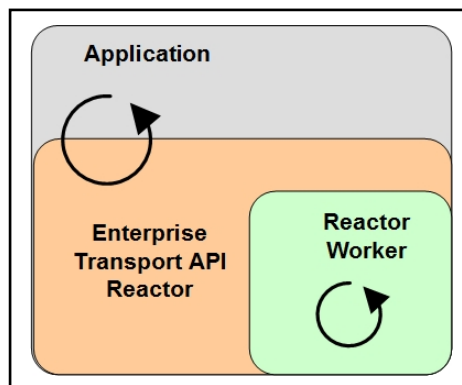
Figure 5. ETA Reactor Thread Model

There are two main threads associated with each ETA Reactor instance. The application thread is the main driver of the reactor; all event dispatching (e.g., reading), callback processing, and submitting of data to the ETA is done from this thread. Such architecture reduces latency and simplifies any threading model associated with user-defined callback methods – because callbacks happen from the application thread, a single-threaded application does not need to have additional mutex locking. The ETA Reactor also leverages an internal worker thread. The worker thread flushes any queued outbound data and manages outbound network pings for all channels associated with the Reactor.

The application drives the reactor with the use of a dispatch method. The dispatch method reads content from the network, performs some light processing to handle inbound network pings, and provides the information to the user through a series of per-channel, user-defined callback methods. Callback methods are separated based on whether they are reactor callbacks or channel callbacks. Channel callbacks are separated by domain, with a default callback where all unhandled domains or non-OMM content are provided to the user. The application can choose whether to dispatch on a single channel or across all channels managed by the reactor. The application can leverage an I/O notification mechanism (e.g. select, poll) or periodically call dispatch – it is all up to the user.

6.1.1   Functionality: Enterprise Transport API Versus Enterprise Transport API Reactor

FUNCTIONALITY	ETA	ETA REACTOR
Automatic Flushing of Data	***	X
Controlled Fragmentation and Assembly of Large Messages	X	X
Controlled Locking / Threading Model	X	X
Controlled Message Buffers with Ability to Change During Runtime	X	X
Controlled Message Packing	X	X
Downloading Field Dictionary	***	X
Loading Field Dictionary File	***	X
Network Ping Management	***	X
Programmatic Configuration	X	X
Programmatic Logging	X	X
Requesting Source Directory	***	X
Round Trip Latency Monitoring	***	For particular roles: • **c**: consumer • **nip**: Non-Interactive provider • **p**: provider
Session Management	***	X
Support for Unified and Segmented Network Connection Types	X	X
User-Defined Callbacks for Data	***	X
User Login	***	X
***: ETA users can implement this functionality themselves. They can also use or modify the ETA Reactor functionality.		

Table 2: ETA Functionality and ETA Reactor Comparison

6.1.2   Reactor Error Handling

The **ReactorErrorInfo** class is used to return error or warning information to the application. This can be returned from the various reactor methods as well as part of a callback method.

- If returned directly from a reactor method: an error occurred while processing in that method.

- If returned as part of a callback method: an error has occurred on one of the channels managed by the reactor.

ReactorErrorInfo members are as follows:

PROPERTY	DESCRIPTION
rsslErroInfoCode	An informational code about this error. Indicates whether it reports a failure condition or is intended to provide non-failure-related information to the user. For details on available codes, refer to Table 21.
Error	Returns an **Error** class (i.e., the underlying error information from the Enterprise Transport API). **Error** includes a pointer to the **IChannel** on which the error occurred, both a Enterprise Transport API and a system error number, and more descriptive error text. The **Error** and its values are described in the Enterprise Transport API C# Edition *Developers Guide*.
Location	Provides information about the file and line on which the error occurred. Detailed error text is provided via the **Error** portion of this class.

Table 3: `ErrorInfo` Properties

6.1.3   Reactor Error Info Codes

It is important that the application monitors return values from the **Reactor** callbacks and methods. Error codes indicate whether the returned **ReactorErrorInfo** is the result of a failure condition or is simply providing information regarding a successful operation.

RETURN CODE	DESCRIPTION
CHANNEL_ERROR	An error was encountered during a channel operation.
FAILURE	A general failure has occurred. The **ReactorErrorInfo** code contains more information about the specific error.
INVALID_ENCODING	The interface is attempting to write a message to the Reactor with an invalid encoding.
INVALID_USAGE	The interface is being improperly used.
NO_BUFFERS	There are no buffers available from the buffer pool. Returned from **ReactorChannel.Submit**. Use **ReactorChannel.Ioctl** to increase the pool size or wait for the reactor's worker thread to flush data and return buffers to pool. Use **IChannel.BufferUsage** or **IServer.BufferUsage** to monitor for free buffers.
PARAMETER_OUT_OF_RANGE	Indicates that a parameter was out of range.
PARAMETER_INVALID	Indicates that a parameter was invalid.
SHUTDOWN	Failure. Reactor is shut down.
SUCCESS	Indicates a success code. Used to inform the user of success and provide additional information.
WRITE_CALL_AGAIN	This is a transport success code. **ReactorChannel.Submit** is fragmenting the buffer and needs to be called again with the same buffer. In this case, **Write** was unable to send all fragments with the current call and must continue fragmenting content.

Table 4: Reactor Error Info Codes

6.1.4 Enterprise Transport API Reactor Application Lifecycle

The following figure depicts the typical lifecycle of an application using the Enterprise Transport API Reactor, as well as associated method calls. Subsequent sections in this document provide more detailed information.

Reactor Application
Lifecycle

Create Reactor using
Reactor.CreateReactor

New outbound
client connection

Accepting new
server connection

Create connection with
Reactor.Connect

Accept connection with
Reactor.Accept

Dispatch events using
Reactor.Dispatch

Send content using
ReactorChannel.Submit

Close channel using
ReactorChannel.Close

Shutdown Reactor with
Reactor.Shutdown

Figure 6. Enterprise Transport API Reactor Application Lifecycle

6.2   Reactor Use

This section describes use of **Reactor**. The **Reactor** manages **ReactorChannels** (described in Section 6.3). An understanding of both constructs is necessary for application writers.

NOTE: An application can leverage multiple **Reactor** instances to scale across multiple cores and distribute their **ReactorChannels** as needed.

PROPERTY	DESCRIPTION
EventSocket	Socket object used for Reactor-specific events to communicate with the worker thread. This socket can be polled for incoming data.
UserSpecObject	A pointer that can be set by the user of the **Reactor**. This value can be set directly or via the creation options. This information can be useful for identifying a specific instance of a **Reactor** or coupling this **Reactor** with other user-defined information.

Table 5: Reactor Class Properties

6.2.1   Creating a Reactor

The lifecycle of a **Reactor** is controlled by the application, which controls creation and destruction of each reactor instance. The following sections describe creation functionality in more detail.

6.2.1.1   Reactor Creation

The creation of a **Reactor** instance can be accomplished through the use of the following method.

FUNCTION NAME	DESCRIPTION
CreateReactor	Creates a **Reactor** instance, including all necessary internal memory and threads. After creating the **Reactor**, **ReactorChannels** can be associated, as described in Section 6.3. Options are passed in via the **CreateReactorOptions**, as defined in Section 6.2.1.2.

Table 6: Reactor Creation Method

6.2.1.2   ReactorOptions Class Properties

CLASS MEMBER	DESCRIPTION
ReissueTokenAttemptInterval	The time (in milliseconds) that the **Reactor** waits before attempting to reissue the token. The minimum interval is 1000 milliseconds, while the default setting is 5000.
ReissueTokenAttemptLimit	The maximum number of times the **Reactor** attempts to reissue the token. If set to default (i.e., -1), there is no maximum limit.
RestProxyOptions	Specifies proxy options for REST requests (**ProxyHostName**, **ProxyPort**, **ProxyUserName**, **ProxyPassword**). When **RestProxyOption** is set, it overrides the **ReactorServiceDiscoveryOptions** properties (**ProxyHostName**, **ProxyPort**, **ProxyUserName**, **ProxyPassword**). Defaults to unset.

Table 7: CreateReactorOptions Class Properties/Methods

CLASS MEMBER	DESCRIPTION
SetRestRequestTimeOut	Specifies the timeout (in seconds) for token service and service discovery request. If the request times out, the Enterprise Transport API Reactor resends the token reissue and the timeout restarts. When using the `Reactor.Connect()` method, if the request times out, the Reactor does not retry. If set to **0**, there is no timeout. Be default, the Enterprise Transport API behaves as if set to **45000** milliseconds.
SetServiceDiscoveryURL	Specifies the URL of Delivery Platform that the RTSDK API uses to discover service information such as the host and port to which the API connects to retrieve real-time data from the Real-Time solution/offering.
SetTokenExpireRatio	Specifies a ratio to multiply the access token's expiration time (in seconds) to determine the length of time the `Reactor` waits before retrieving a new access token and refreshing its connection to Real-Time - Optimized. The valid range is from **0.05** to **0.95**. By default, the Enterprise Transport API behaves as if set to **0.8**.
SetTokenServiceURL	Specifies the URL of Delivery Platform that the RTSDK API uses to obtain an authentication token.
UserSpecObject	An object that can be set by the application. This value is preserved and stored in the `UserSpecObject` of the `Reactor` returned from `CreateReactor`. This information can be useful for identifying a specific instance of a reactor or coupling this `Reactor` with other user-created information.
EnableRestLogStream	Enables logging REST request and response to an output stream. Defaults to false.
RestLogOutputStream	An output stream for logging REST request and response. Defaults to standard output.
Properties Related to XML Tracing	
XmlTracing (Bool)	Enables XML tracing for the `Reactor` to `stdout`. The `Reactor` prints the XML representation of all OMM messages when enabled. Ability to get the value of this parameter exists.
XmlTraceToFile (Bool)	Enables XML tracing for the `Reactor` to file(s). The `Reactor` prints the XML representation of all OMM messages when enabled. Ability to get the value of this parameter exists.
XmlTraceToMultipleFiles (String)	Enables XML trace to multiple files if `XmlTraceToFile` is set. If set to true, the Enterprise Message API writes the XML trace to a new file if the current file size reaches the `XmlTraceMaxFileSize`. If set to false, the Enterprise Message API stops writing the XML trace if the current file reaches the `XmlTraceMaxFileSize`. Ability to get the value of this parameter exists.
XmlTraceMaxFileSize (long)	Sets maximum file size, in bytes, for XML tracing if tracing to file is enabled. If not set, file size grows indefinitely limited by system resources. Default value is 100000000. Ability to get the value of this parameter exists.
XmlTraceFileName (String)	Sets the prefix of the filename to which to write XML trace output if tracing is enabled. Ability to get the value of this parameter exists.
XmlTraceWrite (Bool)	Used to enable or disable XML tracing for outgoing messages if tracing is enabled. Ability to get the value of this parameter exists.
XmlTraceRead (Bool)	Used to enable or disable XML tracing for incoming messages if tracing is enabled. Ability to get the value of this parameter exists.
XmlTracePing (Bool)	Used to enable or disable XML tracing for incoming and outgoing ping (only) messages if tracing is enabled. Ability to get the value of this parameter exists.

Table 7: `CreateReactorOptions` Class Properties/Methods(Continued)

6.2.1.3   ReactorOptions Utility Method

The Enterprise Transport API provides the following utility method for use with the **CreateReactorOptions**.

METHOD NAME	DESCRIPTION
Clear	Clears the **CreateReactorOptions** structure. Useful for class reuse.

Table 8: **CreateReactorOptions** Utility Method

6.2.2   Shutting Down a Reactor

The lifecycle of a **Reactor** instance is controlled by the application, which controls creation and destruction of each reactor instance. The following sections describe the shutdown method in more detail.

6.2.2.1   Reactor Shutdown Method

When the application no longer requires a **Reactor** instance, it can shutdown it using the following method.

METHOD NAME	DESCRIPTION
Shutdown	Shuts down and cleans up a **Reactor**. This also sends **ReactorChannelEvents**, indicating channel down, to all **ReactorChannel**s associated with this **Reactor**.

Table 9: **Reactor** Shutdown Method

6.2.2.2   Reactor Creation and Destruction Example

```
ReactorOptions reactorCreateOptions = new ReactorOptions();

reactorCreateOptions.Clear();

/* Create the Reactor. */
reactor = Reactor.CreateReactor(reactorCreateOptions, out var errorInfo);

/* Any use of the reactor occurs here -- see following sections for all other functionality */

/* Destroy the Reactor. */
reactor.Shutdown(out errorInfo);
```

Code Example 1: Reactor Creation and Destruction Example

6.3   Reactor Channel Use

The **ReactorChannel** class is used to represent a connection that can send or receive information across a network. This class is used to represent a connection, regardless of whether it is an outbound connection or a connection accepted by a listening socket via an **Server** instance. The **ReactorChannel** is the application's point of access, used to perform any action on the connection that it represents (e.g., dispatching events, writing, disconnecting, etc.). See the subsequent sections for more information about **ReactorChannel** and how to associate with a **Reactor**.

NOTE: Only Enterprise Transport API Reactor methods, like those defined in this chapter, should be called on a channel managed by a **Reactor**.

The following table describes the properties of the **ReactorChannel** class.

CLASS PROPERTY	DESCRIPTION
HostName	Provides the name of the host to which a consumer or NIP application connects.
MajorVersion	When a **ReactorChannel** is up (**ReactorChannelEventType.CHANNEL_UP**), this is populated with the major version number associated with the content sent on this connection. Typically only minor version increases are associated with a fully backward compatible change or extension. The Enterprise Transport API Reactor will leverage the versioning information for any content it is encoding or decoding. Proper use of versioning should be handled by the application for any other application encoded or decoded content. For more information on versioning, refer to the Enterprise Transport API C# Edition *Developers Guide*.
MinorVersion	When a **ReactorChannel** is up (**ReactorChannelEventType.CHANNEL_UP**), this is populated with the minor version number associated with the content sent on this connection. Typically, a minor version increase is associated with a fully backward compatible change or extension. The Enterprise Transport API Reactor will leverage the versioning information for any content it is encoding or decoding. Proper use of versioning should be handled by the application for any other application encoded or decoded content. For more information on versioning, refer to the Enterprise Transport API C# Edition *Developers Guide*.
OldSocket	It is possible for a socket to change over time, typically due to some kind of connection keep-alive mechanism. If this occurs, this is typically communicated via a callback indicating **ReactorChannelEventType.FD_CHANGE**. The previous socket is stored in **OldSocket** so the application can properly handle the change.
Port	Provides the server port number to which the consumer or NIP application connects.
ProtocolType	When a **ReactorChannel** is up (**ReactorChannelEventType.CHANNEL_UP**), this is populated with the **ProtocolType** associated with the content being sent on this connection. If the server indicates a **ProtocolType** that does not match the **ProtocolType** specified by the client, the connection is rejected. The Enterprise Transport API Reactor will leverage the versioning information for any content it is encoding or decoding. Proper use of versioning should be handled by the application for any other application encoded or decoded content. For more information on versioning, refer to the Enterprise Transport API C# Edition *Developers Guide*.
Channel	The underlying **IChannel** instance, as defined in the Enterprise Transport API C# Edition *Developers Guide*, mainly for reference purposes. All operations should be performed using the Enterprise Transport API Reactor functionality; the application should not use this **IChannel** instance directly with any Transport functionality.
Server	The underlying **Server** instance, as defined in the Enterprise Transport API C# Edition *Developers Guide*, mainly for reference purposes. This is populated only if the channel was created via the **Reactor.Accept** method, as described in Section 6.4.1.6.

Table 10: ReactorChannel Structure Properties

CLASS PROPERTY	DESCRIPTION
Socket	A **System.Net.Sockets.Socket** instance that corresponds to the current connection.
State	The state of the **ReactorChannel**.
UserSpecObject	An pointer that can be set by the user of the **IChannel**. This value can be set via the **ReactorConnectOption** and **ReactorAcceptOption**. This information can be useful for coupling this **ReactorChannel** with other user-created information.

Table 10: **ReactorChannel** Structure Properties (Continued)

6.3.1   Reactor Channel Roles

A **ReactorChannel** can be configured to fulfill several specific roles, which overlap with the typical OMM application types. Provided role definitions include:

- **ConsumerRole** for OMM consumer applications
- **ProviderRole** for OMM interactive provider applications
- **NIProviderRole** for OMM non-interactive provider applications

All roles have the same base class, **ReactorChannelRoleBase**.

6.3.1.1   ReactorRole Class

ReactorChannelRoleBase contains information and callback methods common to all role types and consists of the following members:

CLASS PROPERTY	DESCRIPTION
ChannelEventCallback	This **ReactorChannel**'s user-defined callback method to handle all **ReactorChannel** specific events, like **ReactorChannelEventType.CHANNEL_UP** or **ReactorChannelEventType.CHANNEL_DOWN**. This callback method is required for all role types. This callback is defined in more detail in Section 6.5.2.
DefaultMsgCallback	This **ReactorChannel**'s user-defined callback method to handle **Msg** content not handled by another domain-specific callback method. This callback method is required for all role types and is defined in more detail in Section 6.5.2.
Type	The role type enumeration value, as defined in Section 6.3.1.2.

Table 11: **ReactorChannelRoleBase** Structure Members

6.3.1.2   ReactorRoleType Enumerations

ENUMERATED NAME	DESCRIPTION
CONSUMER	Indicates that the **ReactorChannel** should act as a consumer.
NIPROVIDER	Indicates that the **ReactorChannel** should act as a non-interactive provider.
PROVIDER	Indicates that the **ReactorChannel** should act as an interactive provider.

Table 12: **ReactorRoleType**Enumerated Values

6.3.2   Reactor Channel Role: OMM Consumer

When a **ReactorChannel** is acting as an OMM consumer application, it connects to an OMM interactive provider. As part of this process it is expected to perform a login to the system. After the login is completed, the consumer acquires a source directory, which provides information about the available services and their capabilities. Additionally, a consumer can download or load field dictionaries, providing information to help decode some types of content. The messages that are exchanged during this connection establishment process are administrative RDMs and are described in the Enterprise Transport API C# Edition *LSEG Domain Model Usage Guide*

A **ReactorChannel** in a consumer role helps to simplify this connection process by exchanging these messages on the user's behalf. The user can choose to provide specific information or leverage a default populated message, which uses the information of the user currently logged into the machine running the application. In addition, the ETA Reactor allows the application to specify user-defined callback methods to handle the processing of received messages on a per-domain basis.

6.3.2.1   OMM Consumer Role

When creating a **ReactorChannel**, this information can be specified with the **ConsumerRole** class as follows:

CLASS	DESCRIPTION
DictionaryDownloadMode	Informs the **ReactorChannel** of the method to use when requesting dictionaries. Allowable modes are defined in Section 6.3.2.2.
DictionaryMsgCallback	This **ReactorChannel**'s user-defined callback method to handle dictionary message content. If not specified, all received dictionary messages will be passed to the **DefaultMsgCallback**. • For more details on this callback, refer to Section 6.5.2. • Dictionary messages are described in Section 8.5.
DirectoryMsgCallback	This **ReactorChannel**'s user-defined callback method to handle directory message content. If not specified, all received directory messages will be passed to the **DefaultMsgCallback**. • For more details on this callback, refer to Section 6.5.2. • Directory messages are described in Section 8.4.
LoginMsgCallback	This **ReactorChannel**'s user-defined callback method to handle login message content. If not specified, all received login messages will be passed to the **DefaultMsgCallback**. • For more details on this callback, refer to Section 6.5.2. • Login messages are described in Section 8.3.
RdmDirectoryRequest	The **DirectoryRequest** (defined in Section 8.4.1) sent during the connection establishment process. This can be populated with specific source directory request information or invoke the **InitDefaultRDMDirectoryRequest** method to populate with default information. • If this parameter is specified, an **RdmLoginRequest** is required. • If this parameter is empty, a directory request is not sent to the system.
ReactorOAuthCredential	Specifies the credentials for an application that makes a Delivery Platform token service request. This is required when connecting to an LSEG Real-Time Advanced Distribution Server in the cloud. See Section 6.7.2.1.
RdmLoginRequest	The **LoginRequest** (defined in Section 8.3.1) sent during the connection establishment process. This can be populated with a user's specific information or invoke the **InitDefaultRDMLoginRequest** method to populate with default information. If this parameter is empty, a login is not sent to the system; useful for systems that do not require a login.

Table 13: **ConsumerRole** Class Properties

6.3.2.2   OMM Consumer Role Dictionary Download Modes

There are several dictionary download options available to a **ReactorChannel**. The application can determine which option is desired and specify using the **ConsumerRole.dictionaryDownloadMode** parameter.

ENUMERATED NAME	DESCRIPTION
FIRST_AVAILABLE	The **Reactor** will search received directory messages for the **RDMFieldDictionary** (RWFFld) and the **enumtype.def** (RWFEnum) dictionaries. Once found, the **Reactor** will request these dictionaries for the application. After transmission is completed, the streams are closed because this content does not update.
NONE	The **Reactor** will not request dictionaries for this **ReactorChannel**. This is typically used when the application has loaded a file-based dictionary or has acquired the dictionary elsewhere.

Table 14: `ConsumerRole.dictionaryDownloadMode` Enumerated Values

6.3.2.3   OMM Consumer Role Watchlist Options

The consumer may enable an internal watchlist and configure behaviors. For more detail on the consumer watchlist feature, refer to Section 2.4.

OPTION	DESCRIPTION
enableWatchlist	Enables the watchlist.
itemCountHint	Can improve performance when used with the watchlist. If possible, set this to the approximate number of item requests the application expects to open.
maxOutstandingPosts	Sets the maximum allowable number of on-stream posts waiting for acknowledgment before the reactor disconnects.
obeyOpenWindow	Sets whether the **Reactor** obeys the OpenWindow of services advertised in a provider's Source Directory response.
postAckTimeout	Sets the time (in milliseconds) a stream waits to receive an ACK for an outstanding post before forwarding a negative acknowledgment **AckMsg** to the application.
requestTimeout	Sets the time (in milliseconds) the watchlist waits for a response to a request.

Table 15: OMM Consumer Role Watchlist Options

6.3.3   Reactor Channel Role: OMM Provider

When a **ReactorChannel** is acting as an OMM provider application, it allows connections from OMM consumer applications. As part of this process it is expected to respond to login requests and source directory information requests. Additionally, a provider can optionally allow consumers to download field dictionaries. Messages exchanged during this connection establishment process are administrative RDMs and are described in the Enterprise Transport API C# Edition *LSEG Domain Model Usage Guide*

A **ReactorChannel** in an interactive provider role allows the application to specify user-defined callback methods to handle the processing of received messages on a per-domain basis.

6.3.3.1   OMM Provider Role

When creating a **ReactorChannel**, this information can be specified with the **ProviderRole** structure, as follows:

STRUCTURE MEMBER	DESCRIPTION
DictionaryMsgCallback	This **ReactorChannel**'s user-defined callback method to handle dictionary message content. If unspecified, all received dictionary messages will be passed to the **DefaultMsgCallback**. • For further details on this callback, refer to Section 6.5.2. • Dictionary messages are described in Section 8.5.
DirectoryMsgCallback	This **ReactorChannel**'s user-defined callback method to handle directory message content. If unspecified, all received directory messages will be passed to the **DefaultMsgCallback**. • For further details on this callback, refer to Section 6.5.2. • Directory messages are described in Section 8.4.
LoginMsgCallback	This **ReactorChannel**'s user-defined callback method to handle login message content. If unspecified, all received login messages are passed to the **DefaultMsgCallback**. • For further details on this callback, refer to Section 6.5.2. • Login messages are described in Section 8.3.

Table 16: **ProviderRole** Structure Members

6.3.4   Reactor Channel Role: OMM Non-Interactive Provider

When a **ReactorChannel** acts as an OMM non-interactive provider application, it connects to an LSEG Real-Time Advanced Distribution Hub and logs into the system. After login, the non-interactive provider publishes a source directory, which provides information about the available services and their capabilities. Messages exchanged while establishing the connection are administrative RDMs and are described in the Enterprise Transport API C# Edition *LSEG Domain Model Usage Guide*

A **ReactorChannel** in a non-interactive provider role helps to simplify this connection process by exchanging these messages on the user's behalf. The user can choose to provide specific information or leverage a default populated message, which uses the information of the user currently logged into the machine running the application. In addition, the ETA Reactor allows the application to specify user-defined callback functions to handle the processing of received messages on a per-domain basis.

6.3.4.1   OMM Non-Interactive Role Members

When creating a **ReactorChannel**, this information can be specified with the **NIProviderRole** structure, as follows:

PROPERTY	DESCRIPTION
RdmLoginRequest	The **LoginRequest**, defined in Section 8.3.1, sent when establishing a connection. You can populate this with a user's specific information or invoke the **InitDefaultRDMLoginRequest** method to populate with a default set of information. If empty, a login is not sent to the system; useful for systems that do not require a login.
pDirectoryRefresh	The **DirectoryRefresh**, defined in Section 8.4.2, sent when establishing a connection. You can populate this with specific source directory refresh information or invoke the **InitDefaultRDMDirectoryRefresh** method to populate it with default information. • If this parameter is specified, an **RdmLoginRequest** is required. • If this parameter is left empty, a refresh is not automatically sent to the system.
LoginMsgCallback	The **ReactorChannel**'s user-defined callback method that handles login message content. If unspecified, all received login messages are passed to the **DefaultMsgCallback**. For further details on this callback, refer to Section 6.5.2.

Table 17: **NIProviderRole** Structure Members

6.4   Managing Reactor Channels

6.4.1   Adding Reactor Channels

A single **Reactor** instance can manage multiple **ReactorChannel**s. A **ReactorChannel** can be instantiated as an outbound client style connection or as a connection that is accepted from an **Server**. Thus, users can mix connection styles within or across Reactors and have consistent usage and behavior.

NOTE: A single **Reactor** can simultaneously manage **ReactorChannel**s from **Reactor.Connect()** and **Reactor.Accept**.

6.4.1.1   Reactor Connect

The **Reactor.Connect** method will create a new **ReactorChannel** and associate it with a **Reactor**. This method creates a new outbound connection. The **ReactorChannel** is returned to the application via a callback, as described in Section 6.5.2, at which point it begins dispatching.

Client applications can specify that **Reactor** automatically reconnect a **ReactorChannel** whenever a connection fails. To enable this, the application sets the appropriate members of the **ReactorConnectOption** class. The application can specify that **Reactor** reconnect the **ReactorChannel** to the same host, or to one from among multiple hosts.

METHOD NAME	DESCRIPTION
Reactor.Connect	Creates a **ReactorChannel** that makes an outbound connection to the configured host. This establishes a connection in a manner similar to the **Connect** method, as described in the Enterprise Transport API C# Edition *Developers Guide*. Connection options are passed in via the **ReactorConnectOption**, as defined in Section 6.4.1.2. **ReactorChannel** specific information, such as the per-channel callback functions, the type of behavior, default RDM messages, and such are passed in via the **ReactorRole**, as defined in Section 6.3.1.

Table 18: **Reactor.Connect** Method

6.4.1.2   ReactorConnectOptions Class Members

CLASS PROPERTY	DESCRIPTION
ConnectionList	Specifies **ReactorConnectInfo** as defined in Section 6.4.1.3. When used with **SetReconnectAttemptLimit()**, the **Reactor** attempts to connect to each host in the list with each reconnection attempt.
SetReconnectAttemptLimit()	The maximum number of times the **Reactor** attempts to reconnect a channel when it fails. If set to **-1**, there is no limit.
SetReconnectMinDelay()	Specifies the minimum length of time the **Reactor** waits (in milliseconds) before attempting to reconnect a failed channel. The time increases with each reconnection attempt, from **SetReconnectMinDelay()** to **SetReconnectMaxDelay()**.

Table 19: **ReactorConnectOptions** Class Members

CLASS PROPERTY	DESCRIPTION
SetReconnectMaxDelay()	Specifies the maximum length of time the **Reactor** waits (in milliseconds) before attempting to reconnect a failed channel. The time increases with each reconnection attempt, from **SetReconnectMinDelay()** to **SetReconnectMaxDelay()**.
	NOTE: The proxy settings specified in **ConnectOptions** can be used to invoke REST requests such as service discovery and obtain an authentication token. They have lower precedence than the proxy settings specified in **RsslCreateReactorOptions.restProxyOptions**. For further details, refer to Section 6.2.1.2.

Table 19: **ReactorConnectOptions** Class Members (Continued)

6.4.1.3   ReactorConnectInfo Class Members

STRUCTURE PROPERTY	DESCRIPTION
ConnectOptions	Specifies information about the host or network to which to connect, the type of connection to use, and other transport-specific configuration information associated with the underlying **Connect** method. This is described in more detail in the Enterprise Transport API C# Edition *Developers Guide*.
EnableSessionManagement	Specifies whether the channel manages the authentication token on behalf of the user used to keep the session alive. Boolean. If set to **true**, the channel obtains the authentication token and refreshes it on behalf of user to keep session active. The default setting is **false**.
SetInitTimeout()	Specifies the amount of time (in seconds) to wait to successfully establish a **ReactorChannel**. If a **ReactorChannel** is not established in this timeframe, an event is dispatched to the application to indicate that the **ReactorChannel** is down.
Location	Specifies the cloud location (e.g., **us-east-1**) of the service provider endpoint to which the RTSDK API establishes a connection. If **location** is not specified, the default setting is **us-east-1**. In any particular cloud location, the **Reactor** connects to the endpoint that provides two available zones for the location (e.g., **[us-east-1a, us-east-1b]**).
ReactorAuthTokenEventCallback	A callback function that receives **ReactorAuthTokenEvents**. The Reactor requests a token for the Consumer and NiProvider applications to send login requests and reissues with the token.

Table 20: **ReactorConnectInfo** Class Members

6.4.1.4   ReactorConnectOptions Utility Method

The Enterprise Transport API provides the following utility function for use with the **ReactorConnectOptions**.

FUNCTION NAME	DESCRIPTION
Clear	Clears the **ReactorConnectOptions** class. Useful for class reuse.

Table 21: **ReactorConnectOptions** Utility Method

6.4.1.5   Reactor.Connect Example

```
ReactorConnectOptions connectOpts = new ReactorConnectOptions();
ConsumerRole consumerRole = new ConsumerRole();

/* Configure connection options.*/
ReactorConnectInfo connectInfo = new ReactorConnectInfo();
connectOpts.connectOptions.connectionList().get(0).connectOptions().unifiedNetworkInfo().address
        ("localhost");
connectinfo.ConnectOptions.UnifiedNetworkInfo.ServiceName = ("14002");

/* Configure a role for this connection as an OMM Consumer. */
var consumerRole = new ConsumerRole();

/* Set the methods to which dispatch will deliver events. */
consumerRole.ChannelEventCallback=channelEventCallback;
consumerRole.DefaultMsgCallback=defaultMsgCallback;
consumerRole.LoginMsgCallback=loginMsgCallback;
consumerRole.DirectoryMsgCallback=directoryMsgCallback;
consumerRole.DictionaryMsgCallback=dictionaryMsgCallback;

/* Initialize a default login request. Once the channel is initialized this message will be sent. */
consumerRole.InitDefaultRDMLoginRequest();

/* Initialize a default directory request. Once the application has logged in, this message will be
        sent. */
consumerRole.InitDefaultRDMDirectoryRequest();

/* Add the connection to the Reactor. */
ret = reactor.Connect(connectOpts, consumerRole, out errorInfo);
```

Code Example 2: Reactor.Connect Example

6.4.1.6   Reactor Accept

The **Reactor.Accept** method creates a new **ReactorChannel** and associates it with an **Reactor**. This method accepts the connection from an already running **IServer**. The **ReactorChannel** will be returned to the application via a callback, as described in Section 6.5.2, at which point it can begin dispatching on the channel.

METHOD NAME	DESCRIPTION
Reactor.Accept	Creates a **ReactorChannel** by accepting it from a **IServer**. This establishes a connection in a manner similar to the **Reactor.Accept()** function, as described in the Enterprise Transport API C# Edition *Developers Guide*. • Connection options are passed in via **ReactorAcceptOption**s, as defined in Section 6.4.1.7. • **ReactorChannel**-specific information (such as the per-channel callback functions, the type of behavior, default RDM messages, and etc.) are passed in via the **ReactorRole**, as defined in Section 6.3.1.

Table 22: Reactor.Accept Function

6.4.1.7   ReactorAcceptOptions Class Members

CLASSPROPERTY	DESCRIPTION
AcceptOptions	The **AcceptOptions** associated with the underlying **Accept** method. This includes an option to reject the connection as well as a **UserSpecObject**. This is described in more detail in the Enterprise Transport API C# Edition *Developers Guide*.
SetInitTimeout	The amount of time (in seconds) to wait for the successful connection establishment of a **ReactorChannel**. If a timeout occurs, an event is dispatched to the application to indicate that the **ReactorChannel** is down.

Table 23: ReactorAcceptOptions Class Members

6.4.1.8   ReactorAcceptOptions Utility Function

The Enterprise Transport API provides the following utility method for use with the **ReactorAcceptOptions**.

METHOD NAME	DESCRIPTION
Clear	Clears the **ReactorAcceptOptions** instance. Useful for instance reuse.

Table 24: ReactorAcceptOptions Utility Method

6.4.1.9 Reactor.Accept Example

```
ReactorAcceptOptions reactorAcceptOpts = new ReactorAcceptOptions();
ProviderRole providerRole = new ProviderRole();

/* Configure accept options.*/
reactorAcceptOpts.Clear();
reactorAcceptOpts.AcceptOptions.UserSpecObject = server;

/* Configure a role for this connection as an OMM Provider. */
providerRole.Clear();
providerRole.ChannelEventCallback = channelEventCallback;
providerRole.DefaultMsgCallback = defaultMsgCallback;
providerRole.LoginMsgCallback = loginMsgCallback;
providerRole.DirectoryMsgCallback = directoryMsgCallback;
providerRole.DictionaryMsgCallback = dictionaryMsgCallback;

/* Add the connection to the Reactor by accepting it from a Server. */
ret = reactor.Accept(server, reactorAcceptOpts, providerRole, out errorInfo)
```

Code Example 3: Reactor.Accept Example

6.4.2 Reactor Channel Reconnection and Recovery Behaviors

For the client-based connections (Consumer and NIProvider), the Reactor can automatically reconnect upon loss of the underlying network connection. This behavior is controlled by the **reconnectAttemptLimit**, **reconnectMinDelay**, and **reconnectMaxDelay** configuration in .

For the delay options, for each disconnect, the reactor will start a reconnectMinDelay, and double it each subsequent connection failure until reconnectMaxDelay is reached.

For the **reconnectAttemptLimit** configuration, please see the following table for the behaviors.

For the Warm Standby feature description, see Chapter 9.

For the Preferred Host feature description, see Chapter 10.

	RECONNECTATTEMPTLIMIT VALUE		
	-1: INFINITE RECONNECT ATTEMPTS	**0: NO RECONNECTION ATTEMPTS**	**N > 0: 1 OR MORE RECONNECTION ATTEMPTS**
Single Connection	Reactor will initially attempt to connect to the connection indefinitely after the initial connection attempt or upon loss of the underlying network connection.	Reactor will attempt to connect to the configured server (first in the connection list, if specified), if this fails upon initial connect or the underlying network connection fails after establishing a connection, the channel will be in a `CHANNEL_DOWN` state, and must be cleaned up and fully closed with `reactorChannel.Close()`.	Reactor will initially attempt to connect to the single connection N times after the initial connection or upon loss of the underlying network connection. This attempt count will be reset once the Reactor Channel is able to connect to a server. Once N attempts have been reached, the channel will be in a `CHANNEL_DOWN` state, and must be cleaned up and fully closed `reactorChannel.Close()`.
Connection List	Reactor will round robin through the connection list, initially starting with the first connection and continuing to the next channel in the list if the attempt fails. After the lost of the underlying network connection, the reactor channel will attempt reconnecting to the next channel in the list. The Reactor will continue to attempt to reconnect infinitely until the channel is closed by the application. Behavior example: C1, C2, C3 configured connections Connection order: C1, C2, C3, repeat from beginning	Reactor will attempt to connect to the configured server (first in the connection list, if specified), if this fails upon initial connect or the underlying network connection fails after establishing a connection, the channel will be in a `CHANNEL_DOWN` state, and must be cleaned up and fully closed with `reactorChannel.Close()`.	Reactor will initially attempt to connect first connection in the list, and will round robin through the list, attempting N times after the initial connection attempt or upon loss of the underlying network connection. This attempt count will be reset once the Reactor Channel is able to connect to a server. Once N attempts have been reached, the channel will be in a `CHANNEL_DOWN` state, and must be cleaned up and fully closed `reactorChannel.Close()`. Behavior example: C1, C2, C3 configured connections Connection order: C1, C2, C3, repeat from beginning, until N total attempts.

Table 25: `reconnectAttemptLimit` Configuration Behaviors

6.4.3 Removing Reactor Channels

6.4.3.1 ReactorChannel.Close Method

You use the following method to remove a **`ReactorChannel`** from a **`Reactor`** instance. It can also close and clean up resources associated with the **`ReactorChannel`**.

FUNCTION NAME	DESCRIPTION
ReactorChannel.Close	Removes a `ReactorChannel` from the `Reactor` associated with the current `ReactorChannel` instance and cleans up associated resources. This additionally invokes the `IChannel.Close` method, as described in the Enterprise Transport API C# Edition *Developers Guide*, to clean up any resources associated with the underlying `IChannel`. This method can be called from either outside or within a callback.

Table 27: ReactorChannel.Close Function

6.4.3.2   ReactorChannel.Close Example

```
ReactorErrorInfo errorInfo = new ReactorErrorInfo();
/* Can be used inside or outside of a callback */
ret = reactorChannel.Close(out errorInfo);
```

Code Example 4: `ReactorChannel.Close` Example

6.5   Dispatching Data

Once an application has a `Reactor`, it can begin dispatching messages. Until there is at least one associated `ReactorChannel`, there is nothing to dispatch. When `ReactorChannel`s are available for dispatching, each channel begins seeing its user-defined per-channel callbacks being invoked. For more information about available callbacks and their specifications, refer to Section 6.5.2.

An application can choose to dispatch across all associated `ReactorChannel`s or on a particular `ReactorChannel` via `Reactor.Dispatch` method by either not setting `ReactorChannel` property of the supplied `ReactorDispatchOptions` instance or setting it to a specific `ReactorChannel` instance. If dispatching across multiple `ReactorChannel`s, the `Reactor` attempts to fairly dispatch over all channels. In either case, the application can use the dispatch call to specify the maximum number of messages that will be processed and returned via callback.

6.5.1   Reactor Dispatch Methods

NOTE: Applications should not call `DestroyReactor` or `Reactor.Dispatch` from within a callback function. All other `Reactor` functionality is safe to use from within a callback.

Events received in callback methods should be assumed to be invalid when the callback method returns. For callbacks that provide `Msg`, `LoginMsg`, `DirectoryMsg`, or `DictionaryMsg` instances, a deep copy of the object should be made if the application wishes to preserve it. To copy an `Msg`, refer to the CopyMsg method in the Enterprise Transport API C# Edition *Developers Guide*; for copying a `LoginMsg`, `DirectoryMsg`, or `DictionaryMsg` object, refer to the copy utility method for the appropriate RDM message type.

METHOD NAME	DESCRIPTION
Dispatch	This function can process events and messages across all associated `ReactorChannels` of the current `Reactor` instance or for a single `ReactorChannel` depending on the value of the provided `ReactorDispatchOptions.ReactorChannel` value. When channel information or data is available for a `ReactorChannel`, the channel's user-defined callback method is invoked (for details refer to Section 6.5.1.1).

Table 28: `Reactor.Dispatch` Method

6.5.1.1   Reactor Dispatch Options

An application can use **ReactorDispatchOptions** to control various aspects of the call to **Reactor.Dispatch**.

PROPERTY	DESCRIPTION
SetMaxMessages	Controls the maximum number of events or messages processed in this call. If this is larger than the number of available messages, **Reactor.Dispatch** will return when there is no more data to process. This value is initialized to allow up to 100 messages to be returned with a single call to **Reactor.Dispatch**.
ReadArgs	The **ReadArgs** from the underlying **Channel.Read** call.

Table 29: `ReactorDispatchOptions` Class Properties

6.5.1.2   ReactorDispatchOptions Utility Method

The Enterprise Transport API provides the following utility method for use with **ReactorDispatchOptions**.

METHOD NAME	DESCRIPTION
Clear	Clears the **ReactorDispatchOptions** instance. Useful for instance reuse.

Table 30: **ReactorDispatchOptions** Utility Method

6.5.1.3   Reactor.Dispatch Example

```
ReactorDispatchOptions dispatchOpts = new ReactorDispatchOptions();

/* Set dispatching options. */
dispatchOpts.Clear();
dispatchOpts.SetMaxMessages(200);
dispatchOpts.ReactorChannel  = reactorChannel;

/* Call Reactor.Dispatch(). It will keep dispatching events until there is nothing to read or
 * maxMessages is reached. */
ret = reactor.Dispatch(dispatchOpts, out errorInfo);
```

Code Example 5: Reactor.Dispatch Example

6.5.2   Reactor Callback Methods

A series of callback methods returns (to the application) any state information about the **ReactorChannel** connection as well as messages for that channel. Each **ReactorChannel** can define its own unique callback methods or specify callback methods that can be shared across channels.

There are several values that can be returned from a callback method implementation. These can trigger specific **Reactor** behaviors based on the outcome of the callback method. Callback return values are as follows:

RETURN CODE	DESCRIPTION
SUCCESS	Indicates that the callback function was successful and the message or event has been handled.
FAILURE	Indicates that the message or event has failed to be handled. Returning this code from any callback function will cause the **Reactor** to shutdown.
RAISE	Can be returned from any domain-specific callback (e.g., **IRDMLoginMsgCallback**). This will cause the **Reactor** to invoke the **IDefaultMsgCallback** for this message upon the domain-specific callbacks return.

Table 31: **ReactorCallbackReturnCode** Callback Return Codes

All events communicated to callback methods have the same base class, the **ReactorEvent**, which contains information common to all callback events.

CLASS PROPERTY	DESCRIPTION
ReactorChannel	The **ReactorChannel** on which the event occurred.
ReactorErrorInfo	The **ReactorErrorInfo** associated with this event.

Table 32: ReactorEvent Class Properties

6.5.3   Reactor Callback: Channel Event

The **Reactor** channel event callback communicates **ReactorChannel** and connection state information to the application. This interface has the following callback method:

```
IReactorChannelEventCallback(ReactorChannelEvent event)
```

When invoked, this returns a **ReactorChannelEvent** instance, containing more information about the event.

6.5.3.1   Reactor Channel Event

The **ReactorChannelEvent** is returned to the application via the **IReactorChannelEventCallback**.

CLASS PROPERTY	DESCRIPTION
EventType	The type of event that has occurred on the **ReactorChannel**. For a list of enumeration values, refer to Section 6.5.3.2.

Table 33: ReactorChannelEvent Class Property

6.5.3.2   Reactor Channel Event Type Enumeration Values

FLAG ENUMERATION	MEANING
CHANNEL_DOWN	Indicates that the **ReactorChannel** is not available for use. This could be a result of an initialization failure, a ping timeout, or some other kind of connection-related issue. **ReactorErrorInfo** will contain more detailed information about what occurred. There is no connection recovery for this event. To clean up the failed **ReactorChannel**, the application should call **ReactorChannel.Close**.
CHANNEL_DOWN_RECONNECTING	Indicates that the **ReactorChannel** is temporarily unavailable for use. The **Reactor** will attempt to reconnect the channel according to the values specified in **ReactorConnectOptions** when **Reactor.Connect()** was called. This only occurs on client connections because there is no connection recovery for server connections. If the watchlist is enabled, requests are recovered as appropriate when the channel successfully reconnects. Before exiting the **channelEventCallback**, the application should release any resources associated with the channel..

Table 34: ReactorChannelEventType Enumeration Values

FLAG ENUMERATION	MEANING
CHANNEL_OPENED	Indicates that the watchlist is enabled and that a channel has been created via **Reactor.Connect()**. Though the channel is still not ready for dispatch, the application can begin submitting request messages, which are sent after the channel successfully initializes.
CHANNEL_READY	Indicates that the **ReactorChannel** has successfully completed any necessary initialization processes. Where applicable, this includes exchanging any provided Login, Directory, or Dictionary content. The application should now be able to consume or provide content.
CHANNEL_UP	Indicates that the **ReactorChannel** is successfully initialized and available for dispatching. Where applicable, any specified Login, Directory, or Dictionary messages are exchanged by the **Reactor**.
FD_CHANGE	Indicates that a **Socket** change occurred on the **ReactorChannel**. If the application is using an I/O notification mechanism, unregister the **OldSocket OldSocket** and register the **Socket**, both of which can be found on the **ReactorChannel**.
INIT	Channel event initialization value. This should not be used by nor returned to the application.
WARNING	Indicates that the **ReactorChannel** has experienced an event that did not result in connection failure, but may require the attention of the application. **ErrorInfo** contains more detailed information about what occurred.

Table 34: `ReactorChannelEventType` Enumeration Values (Continued)

6.5.3.3   Reactor Channel Event Callback Example

```csharp
public ReactorCallbackReturnCode ReactorChannelEventCallback(ReactorChannelEvent evt)
        {
            ChannelInfo? chnlInfo = evt.ReactorChannel?.UserSpecObj as ChannelInfo;
            if (chnlInfo == null)
                return ReactorCallbackReturnCode.FAILURE;

            switch (evt.EventType)
            {
                case ReactorChannelEventType.CHANNEL_UP:
                    {
                        if (evt.ReactorChannel?.Socket != null)
                            Console.WriteLine("Channel Up Event: " +
                            evt.ReactorChannel.Socket.Handle.ToInt32());
                        else
                            Console.WriteLine("Channel Up Event");

                        // register selector with channel event's reactorChannel
                        RegisterChannel(evt.ReactorChannel!);

                        break;
                    }
                case ReactorChannelEventType.FD_CHANGE:
                    {
```

```csharp
                    int fdOldSocketId = evt.ReactorChannel!.OldSocket!.Handle.ToInt32();
                    int fdSocketId = evt.ReactorChannel!.Socket!.Handle.ToInt32();

                Console.WriteLine($"Channel Change - Old Channel: {fdOldSocketId}
                New Channel: {fdSocketId}");

                    // cancel old reactorChannel select
                    UnregisterSocket(evt.ReactorChannel.OldSocket);

                    // register selector with channel event's new reactorChannel
                    RegisterChannel(evt.ReactorChannel);

                    break;
                }
            case ReactorChannelEventType.CHANNEL_READY:
                {
                    // set ReactorChannel on ChannelInfo
                    chnlInfo.ReactorChannel = evt.ReactorChannel;
                    if (evt.ReactorChannel?.Socket != null)
                        Console.WriteLine("Channel Ready Event: " +
                        evt.ReactorChannel.Socket.Handle.ToInt32());
                    else
                        Console.WriteLine("Channel Ready Event");

                    if (IsRequestedServiceUp(chnlInfo))
                    {
                        CheckAndInitPostingSupport(chnlInfo);

                        if (!chnlInfo.ItemWatchList.IsEmpty)
                        {
                            chnlInfo.ItemWatchList.Clear();
                        }

                        SendMPRequests(chnlInfo);
                        SendMBORequests(chnlInfo);
                        SendMBPRequests(chnlInfo);
                        SendSymbolListRequests(chnlInfo);
                        SendYieldCurveRequests(chnlInfo);
                        chnlInfo.RequestsSent = true;
                    }

                    break;
                }
            case ReactorChannelEventType.CHANNEL_DOWN_RECONNECTING:
                {
                    if (evt.ReactorChannel?.Socket != null)
                        Console.WriteLine("\nConnection down reconnecting: Channel " +
                        evt.ReactorChannel.Socket.Handle.ToInt32());
```

```csharp
                    else
                        Console.WriteLine("\nConnection down reconnecting");

                    if (evt.ReactorErrorInfo != null && evt.ReactorErrorInfo.Error.Text != null)
                     Console.WriteLine("\tError text: " + evt.ReactorErrorInfo.Error.Text + "\n");

                    // allow Reactor to perform connection recovery

                    // unregister selectableChannel from Selector
                    if (evt.ReactorChannel?.Socket != null)
                    {
                        UnregisterSocket(evt.ReactorChannel.Socket);
                    }

                    // reset dictionary if not loaded from file
                    if (!fieldDictionaryLoadedFromFile
                        && !enumTypeDictionaryLoadedFromFile)
                    {
                        chnlInfo.Dictionary?.Clear();
                    }

                    // reset item request(s) sent flag
                    chnlInfo.RequestsSent = false;

                    // reset hasServiceInfo flag
                    chnlInfo.HasServiceInfo = false;

                    // reset canSendLoginReissue flag
                    chnlInfo.CanSendLoginReissue = false;

                    SetItemState(chnlInfo, StreamStates.CLOSED_RECOVER, DataStates.SUSPECT,
                    StateCodes.NONE);

                    break;
                }
        case ReactorChannelEventType.CHANNEL_DOWN:
            {
                if (evt.ReactorChannel!.Socket != null)
                    Console.WriteLine("\nConnection down: Channel " +
                    evt.ReactorChannel.Socket.Handle.ToInt32());
                else
                    Console
```

Code Example 6: Reactor Channel Event Callback Example

6.5.4   Reactor Callback: Default Message

The **Reactor** default message callback communicates all received content that is not handled directly by a domain-specific callback method. This callback is also invoked after any domain-specific callback that returns the **ReactorCallbackReturnCode.RAISE** value. This interface has the following callback method:

```
public ReactorCallbackReturnCode DefaultMsgCallback(ReactorMsgEvent event)
```

When invoked, this returns an **MsgEvent** class, containing more information about the event information.

6.5.4.1   Reactor Message Event

The **MsgEvent** is returned to the application via the **IDefaultMsgCallback**. This is also the base class of the **DictionaryMsgEvent**, **DirectoryMsgEvent**, and **LoginMsg** classes.

CLASS PROPERTY	DESCRIPTION
TransportBuffer	A **Buffer** instance containing the raw, undecoded message that was read and processed by the callback.
	NOTE: When the consumer watchlist is enabled, a **Buffer** is not provided, because the message might not match this buffer, or the message might be internally generated.
Msg	An **Msg** instance populated with message content by calling the **DecodeMsg** method. If not present, an error was encountered while processing the information.
	NOTE: When the consumer watchlist is enabled, is not provided to callback functions that provide RDM messages.
StreamInfo	Any information associated with a stream (only when the consumer watchlist is enabled).

Table 35: MsgEvent Structure Properties

6.5.4.2   Reactor Message Event Utility Methods

METHOD NAME	DESCRIPTION
Clear	Clears an **MsgEvent** class.

Table 36: MsgEvent Utility Method

6.5.4.3 Reactor Message Event Callback Example

```csharp
public ReactorCallbackReturnCode DefaultMsgCallback(ReactorMsgEvent evt)
        {
            ChannelInfo? chnlInfo = evt.ReactorChannel?.UserSpecObj as ChannelInfo;
            Msg? msg = evt.Msg as Msg;

            if (msg == null)
            {
```

Example 7: Reactor Message Event Callback Example

```csharp
                /* The message is not present because an error occurred while decoding it. Print
                 * the error and close the channel. If desired, the un-decoded message buffer
                 * is available in event.transportBuffer(). */
                Console.WriteLine("DefaultMsgCallback: {0}({1})",
                    evt.ReactorErrorInfo.Error.Text, evt.ReactorErrorInfo.Location);

                // unregister selectableChannel from Selector
                if (evt.ReactorChannel!.Socket != null)
                {
                    m_ReadSockets.Remove(evt.ReactorChannel.Socket);
                    m_SocketFdValueMap.Remove(evt.ReactorChannel.Socket.Handle.ToInt32());
                }

                // close ReactorChannel
                if (chnlInfo!.ReactorChannel != null)
                {
                    chnlInfo.ReactorChannel.Close(out _);
                }
                return ReactorCallbackReturnCode.SUCCESS;
            }

            // set response message
            chnlInfo!.ResponseMsg = msg;

            // set-up decode iterator if message has message body
            if (msg.EncodedDataBody != null
                && msg.EncodedDataBody.Data() != null)
            {
                // clear decode iterator
                chnlInfo.DecodeIter.Clear();

                // set buffer and version info
                chnlInfo.DecodeIter.SetBufferAndRWFVersion(msg.EncodedDataBody,
                    evt.ReactorChannel!.MajorVersion,
                    evt.ReactorChannel.MinorVersion);
            }

            ProcessResponse(chnlInfo);

            return ReactorCallbackReturnCode.SUCCESS;
```

```csharp
    }public ReactorCallbackReturnCode DefaultMsgCallback(ReactorMsgEvent evt)
    {
        ChannelInfo? chnlInfo = evt.ReactorChannel?.UserSpecObj as ChannelInfo;
        Msg? msg = evt.Msg as Msg;

        if (msg == null)
        {
            /* The message is not present because an error occurred while decoding it. Print
             * the error and close the channel. If desired, the un-decoded message buffer
             * is available in event.transportBuffer(). */
            Console.WriteLine("DefaultMsgCallback: {0}({1})",
                evt.ReactorErrorInfo.Error.Text, evt.ReactorErrorInfo.Location);

            // unregister selectableChannel from Selector
            if (evt.ReactorChannel!.Socket != null)
            {
                m_ReadSockets.Remove(evt.ReactorChannel.Socket);
                m_SocketFdValueMap.Remove(evt.ReactorChannel.Socket.Handle.ToInt32());
            }

            // close ReactorChannel
            if (chnlInfo!.ReactorChannel != null)
            {
                chnlInfo.ReactorChannel.Close(out _);
            }
            return ReactorCallbackReturnCode.SUCCESS;
        }

        // set response message
        chnlInfo!.ResponseMsg = msg;

        // set-up decode iterator if message has message body
        if (msg.EncodedDataBody != null
            && msg.EncodedDataBody.Data() != null)
        {
            // clear decode iterator
            chnlInfo.DecodeIter.Clear();

            // set buffer and version info
            chnlInfo.DecodeIter.SetBufferAndRWFVersion(msg.EncodedDataBody,
                evt.ReactorChannel!.MajorVersion,
                evt.ReactorChannel.MinorVersion);
        }

        ProcessResponse(chnlInfo);

        return ReactorCallbackReturnCode.SUCCESS;
    }
```

Example 7: Reactor Message Event Callback Example

6.5.5   Reactor Callback: RDM Login Message

The **Reactor** RDM Login Message callback is used to communicate all received RDM Login messages. This interface has the following callback method:

```
public ReactorCallbackReturnCode RdmLoginMsgCallback(RDMLoginMsgEvent event)
```

When invoked, this will return the **LoginMsgEvent** class, containing more information about the event information.

6.5.5.1   Reactor RDM Login Message Event

The **LoginMsgEvent** is returned to the application via the **RDMLoginMsgCallback**.

STRUCTURE PROPERTY	DESCRIPTION
LoginMsg	The RDM representation of the decoded Login message. If not present, an error was encountered while processing the information. This message is presented as the **LoginMsg**, described in Section 8.3.

Table 37: `LoginMsgEvent` Class Property

6.5.5.2   Reactor RDM Login Message Event Utility Method

METHOD NAME	DESCRIPTION
Clear	Clears an **LoginMsgEvent** class.

Table 38: `LoginMsgEvent` Utility Method

6.5.5.3   Reactor RDM Login Message Event Callback Example

```
public ReactorCallbackReturnCode RdmLoginMsgCallback(RDMLoginMsgEvent evt)
        {
            ChannelInfo? chnlInfo = evt.ReactorChannel?.UserSpecObj as ChannelInfo;
            LoginMsgType msgType = evt.LoginMsg!.LoginMsgType;

            if (chnlInfo == null)
                return ReactorCallbackReturnCode.FAILURE;

            switch (msgType)
            {
                case LoginMsgType.REFRESH:
                    Console.WriteLine("Received Login Refresh for Username: " +
                    evt.LoginMsg.LoginRefresh!.UserName);
                    break;
```

Table 39: Reactor RDM Login Message Event Callback Example

```
            case LoginMsgType.STATUS:
                Console.WriteLine("Received Login StatusMsg");
                break;

            case LoginMsgType.RTT:
                LoginRTT loginRTT = evt.LoginMsg.LoginRTT!;
                Console.Write("\nReceived login RTT message from Provider {0}.\n",
                    evt.ReactorChannel!.Socket!.Handle.ToInt32());
                Console.Write("\tTicks: {0}\n", Math.Round((double)loginRTT.Ticks / 1000));
                if (loginRTT.HasRTLatency)
                {
                    Console.Write("\tLast Latency: {0}\n", Math.Round((double)loginRTT.RTLatency /
                    1000));
                }
                if (loginRTT.HasTCPRetrans)
                {
                Console.Write("\tProvider side TCP Retransmissions: {0}\n", loginRTT.TCPRetrans);
                }
                Console.WriteLine("RTT Response sent to provider by reactor.\n");
                break;

            default:
                Console.WriteLine($"Received Unhandled Login Msg Type: {msgType}");
                break;
        }

        return ReactorCallbackReturnCode.SUCCESS;
    }
```

Table 39: Reactor RDM Login Message Event Callback Example

6.5.6 Reactor Callback: RDM Directory Message

The **Reactor** RDM Directory Message callback is used to communicate all received RDM Directory messages. This interface has the following callback method:

```
public ReactorCallbackReturnCode RdmDirectoryMsgCallback(RDMDirectoryMsgEvent event)
```

When invoked, this will return the **DirectoryMsgEvent** class, containing more information about the event information.

6.5.6.1   Reactor RDM Directory Message Event

The **DirectoryMsgEvent** is returned to the application via the **IDirectoryMsgCallback**.

STRUCTURE PROPERTY	DESCRIPTION
DirectoryMsg	The RDM representation of the decoded Source Directory message. If not present, an error was encountered while processing the information. This message is presented as the **DirectoryMsg**, described in Section 8.4.

Table 40: DirectoryMsgEvent Structure Property

6.5.6.2   Reactor RDM Directory Message Event Utility Method

METHOD NAME	DESCRIPTION
Clear	Clears an **DirectoryMsgEvent** instance.

Table 41: DirectoryMsgEvent Utility Method

6.5.6.3   Reactor RDM Directory Message Event Callback Example

```
public ReactorCallbackReturnCode RdmDirectoryMsgCallback(RDMDirectoryMsgEvent evt)
        {
            ChannelInfo? chnlInfo = evt.ReactorChannel?.UserSpecObj as ChannelInfo;
            DirectoryMsgType msgType = evt.DirectoryMsg!.DirectoryMsgType;

            if (chnlInfo == null)
                return ReactorCallbackReturnCode.FAILURE;

            switch (msgType)
            {
                case DirectoryMsgType.REFRESH:
                    DirectoryRefresh directoryRefresh = evt.DirectoryMsg.DirectoryRefresh!;
                    ProcessServiceRefresh(directoryRefresh, chnlInfo);
                    if (chnlInfo.ServiceInfo.Action == MapEntryActions.DELETE)
                    {
                        error.Text = "RdmDirectoryMsgCallback(): DirectoryRefresh Failed: directory
                        service is deleted";
                        return ReactorCallbackReturnCode.SUCCESS;
                    }
                    break;
                case DirectoryMsgType.UPDATE:
                    DirectoryUpdate directoryUpdate = evt.DirectoryMsg.DirectoryUpdate!;
                    ProcessServiceUpdate(directoryUpdate, chnlInfo);
                    if (chnlInfo.ServiceInfo.Action == MapEntryActions.DELETE)
                    {
                        error.Text = "RdmDirectoryMsgCallback(): DirectoryUpdate Failed: directory
                        service is deleted";
```

Table 42: Reactor RDM Directory Message Event Callback Example

```
                    return ReactorCallbackReturnCode.SUCCESS;
                }
                if (IsRequestedServiceUp(chnlInfo) && !chnlInfo.RequestsSent)
                {
                    CheckAndInitPostingSupport(chnlInfo);

                    chnlInfo.ItemWatchList.Clear();

                    SendMPRequests(chnlInfo);
                    SendMBORequests(chnlInfo);
                    SendMBPRequests(chnlInfo);
                    SendSymbolListRequests(chnlInfo);
                    SendYieldCurveRequests(chnlInfo);
                    chnlInfo.RequestsSent = true;
                }
                break;
            case DirectoryMsgType.CLOSE:
                Console.WriteLine("Received Source Directory Close");
                break;
            case DirectoryMsgType.STATUS:
                DirectoryStatus directoryStatus = evt.DirectoryMsg.DirectoryStatus!;
                Console.WriteLine("\nReceived Source Directory StatusMsg");
                if (directoryStatus.HasState)
                {
                    Console.WriteLine("\t" + directoryStatus.State);
                }
                break;
            default:
                Console.WriteLine("Received Unhandled Source Directory Msg Type: " + msgType);
                break;
        }

        return ReactorCallbackReturnCode.SUCCESS;
    }
```

Table 42: Reactor RDM Directory Message Event Callback Example

6.5.7   Reactor Callback: RDM Dictionary Message

The **Reactor** RDM Dictionary Message callback is used to communicate all received RDM Dictionary messages. This interface has the following callback method:

```
public ReactorCallbackReturnCode RdmDictionaryMsgCallback(RDMDictionaryMsgEvent event)
```

When invoked, this will return the **DictionaryMsgEvent** class, containing more information about the event information.

6.5.7.1   Reactor RDM Dictionary Message Event

The **DictionaryMsgEvent** is returned to the application via the **IDictionaryMsgCallback**.

STRUCTURE MEMBER	DESCRIPTION
RdmDictionaryMsg	The RDM representation of the decoded Dictionary message. If not present, an error was encountered while processing the information. This message is presented as the **DictionaryMsg**, described in Section 8.5.

Table 43: `DictionaryMsgEvent` Structure Member

6.5.7.2   Reactor RDM Dictionary Message Event Utility Method

METHOD NAME	DESCRIPTION
Clear	Clears a **DictionaryMsgEvent** class.

Table 44: `DictionaryMsgEvent` Utility Method

6.5.7.3   Reactor RDM Dictionary Message Event Callback Example

```
public ReactorCallbackReturnCode RdmDictionaryMsgCallback(RDMDictionaryMsgEvent evt)
        {
            ChannelInfo? chnlInfo = evt.ReactorChannel?.UserSpecObj as ChannelInfo;
            DictionaryMsgType msgType = evt.DictionaryMsg!.DictionaryMsgType;

            if (chnlInfo == null)
                return ReactorCallbackReturnCode.FAILURE;

            // initialize dictionary
            if (chnlInfo.Dictionary == null)
            {
                chnlInfo.Dictionary = m_Dictionary;
            }

            switch (msgType)
            {
                case DictionaryMsgType.REFRESH:
                    DictionaryRefresh dictionaryRefresh = evt.DictionaryMsg.DictionaryRefresh!;

                    if (dictionaryRefresh.HasInfo)
                    {
                        /* The first part of a dictionary refresh should contain information about its
                        type.
                         * Save this information and use it as subsequent parts arrive. */
                        switch (dictionaryRefresh.DictionaryType)
                        {
```

Code Example 8: Reactor RDM Dictionary Message Event Callback Example

```
                          case Dictionary.Types.FIELD_DEFINITIONS:
                              chnlInfo.FieldDictionaryStreamId = dictionaryRefresh.StreamId;
                              break;
                          case Dictionary.Types.ENUM_TABLES:
                              chnlInfo.EnumDictionaryStreamId = dictionaryRefresh.StreamId;
                              break;
                          default:
                              Console.WriteLine($"Unknown dictionary type
                              {dictionaryRefresh.DictionaryType} from message on stream
                              {dictionaryRefresh.StreamId}");
                              chnlInfo.ReactorChannel!.Close(out _);
                              return ReactorCallbackReturnCode.SUCCESS;
                  }
              }

              /* decode dictionary response */

              // clear decode iterator
              chnlInfo.DecodeIter.Clear();

              // set buffer and version info
              chnlInfo.DecodeIter.SetBufferAndRWFVersion(dictionaryRefresh.DataBody,
                          evt.ReactorChannel!.MajorVersion,
                          evt.ReactorChannel.MinorVersion);

              Console.WriteLine("Received Dictionary Response: " +
              dictionaryRefresh.DictionaryName);

              if (dictionaryRefresh.StreamId == chnlInfo.FieldDictionaryStreamId)
              {
                  if (chnlInfo.Dictionary.DecodeFieldDictionary(chnlInfo.DecodeIter,
                  Dictionary.VerbosityValues.VERBOSE, out _)
                      == CodecReturnCode.SUCCESS)
                  {
                      if (dictionaryRefresh.RefreshComplete)
                      {
                          Console.WriteLine($"Field Dictionary complete.");
                      }
                  }
                  else
                  {
                      Console.WriteLine("Decoding Field Dictionary failed: " + error.Text);
                      chnlInfo.ReactorChannel!.Close(out _);
                  }
              }
              else if (dictionaryRefresh.StreamId == chnlInfo.EnumDictionaryStreamId)
              {
                  if (chnlInfo.Dictionary.DecodeEnumTypeDictionary(chnlInfo.DecodeIter,
                  Dictionary.VerbosityValues.VERBOSE, out _)
                      == CodecReturnCode.SUCCESS)
```

Code Example 8: Reactor RDM Dictionary Message Event Callback Example

```
                    {
                        if (dictionaryRefresh.RefreshComplete)
                        {
                            Console.WriteLine("EnumType Dictionary complete.");
                        }
                    }
                    else
                    {
                        Console.WriteLine("Decoding EnumType Dictionary failed: " + error.Text);
                        chnlInfo.ReactorChannel!.Close(out _);
                    }
                }
                else
                {
                    Console.WriteLine("Received unexpected dictionary message on stream " +
                    dictionaryRefresh.StreamId);
                }
                break;
            case DictionaryMsgType.STATUS:
                Console.WriteLine("Received Dictionary StatusMsg");
                break;
            default:
                Console.WriteLine("Received Unhandled Dictionary Msg Type: " + msgType);
                break;
        }

        return ReactorCallbackReturnCode.SUCCESS;
    }
```

Code Example 8: Reactor RDM Dictionary Message Event Callback Example

6.6   Writing Data

The Enterprise Transport API Reactor helps streamline the high performance writing of content. The **Reactor** flushes content to the network so the application does not need to. The **Reactor** does so through the use of a separate worker thread that becomes active whenever there is queued content that needs to be passed to the connection.

The Enterprise Transport API Reactor functionality offers several methods for writing content: three overloads of **ReactorChannel.Submit(Msg...)** method for submitting a Codec message, an RDM message or a buffer, and **Reactor** method for submitting an RDM message. When writing applications to the Reactor, consider which is most appropriate for your needs:

Submitting Codec/RDM message

- Takes an **Msg** class as part of its options; does not require retrieval of an **Buffer** from the channel.

Submitting ITransportBuffer instance

- Takes an **Buffer** which the application retrieves from the channel.

- More efficient: the application encodes directly into the buffer, and can use buffer packing.

6.6.1　Writing Data using ReactorChannel.Submit(Msg…)

`ReactorChannel.Submit` provides a simple interface for writing `Msg`s. To send a message, the application populates an `Msg` instance, sets any other desired options on a `ReactorSubmitMsgOptions` instance, and calls `ReactorChannel.Submit(Msg...)` with the instance.

A buffer is not needed to use `ReactorChannel.Submit(Msg...)`. If the application needs to include any encoded content, it can encode the content into any available memory, and set the appropriate member of the `Msg` to point to the memory (as well as set the length of the encoded content).

6.6.1.1　ReactorChannel.Submit(Msg...) Method

METHOD NAME	DESCRIPTION
ReactorChannel.Submit	Encodes and submits an `Msg` to the Reactor. This method expects a properly populated `Msg`.

Table 45: `ReactorChannel.Submit(Msg...)` Method

6.6.1.2　Reactor Submit Options

An application can use `ReactorSubmitMsgOptions` to control various aspects of the call to `ReactorChannel.Submit`.

STRUCTURE MEMBER	DESCRIPTION
ServiceName	The application can use this instead of the `ServiceId` member specified on the `MsgKey` of an `Msg`. When used to open streams via request messages, the `Reactor` will recover using this service name. When used for other message types such as a post or generic message, the `Reactor` converts the name to its corresponding ID before writing the message. **NOTE:** This option is supported only when the watchlist is enabled.
requestMsgOptions	Provides additional functionality that may be used when using request messages to send requests.
WriteArgs.BytesWritten	If specified, will return the number of bytes to be written, including any transport header overhead and taking into account any savings from compression.
WriteArgs.Flags	Flag values that allow the application to modify the behavior of this `ReactorChannel.Submit` call. This includes options to bypass queuing or compression. More information about the specific flag values is available in the Enterprise Transport API C# Edition *Developers Guide*.
WriteArgs.Priority	Controls the priority at which the data will be written. Valid priorities are • WritePriorities.HIGH • WritePriorities.MEDIUM • WritePriorities.LOW More information about write priorities, including an example scenario, is available in the Enterprise Transport API C# Edition *Developers Guide*.
WriteArgs.UncompressedBytesWritten	If specified, will return the number of bytes to be written, including any transport header overhead but not taking into account any compression savings.

Table 46: `ReactorSubmitMsgOptions` Class Properties

6.6.1.3   ReactorChannel.Submit(Msg...) Return Codes

The following table defines the return codes that can occur when using `ReactorChannel.Submit(Msg...)`.

RETURN CODE	DESCRIPTION
ReactorReturnCode.SUCCESS	Indicates that the `ReactorChannel.Submit(Msg...)` method has succeeded.
ReactorReturnCode.NO_BUFFERS	Indicates that not enough pool buffers are available to write the message. The application can try to submit the message later, or it can use `ReactorChannel.Ioctl` to increase the number of available pool buffers and try again.
ReactorReturnCode.FAILURE	Indicates that a general failure has occurred and the message was not submitted. The `ErrorInfo` class passed to the method will contain more details.

Table 47: `ReactorChannel.Submit(Msg...)` Return Codes

6.6.1.4   ReactorRequestMsgOptions

`ReactorRequestMsgOptions` provide additional functionality when requesting items. These options are available only when the consumer watchlist is enabled.

STRUCTURE MEMBER	DESCRIPTION
UserSpecObject	A user-specified pointer that will be associated with the stream. This pointer will be provided in responses to this stream via the `WatchlistStreamInfo` provided with each message event.

Table 48: `ReactorRequestMsgOptions` Structure Members

6.6.1.5   ReactorSubmitOptions Utility Method

The Enterprise Transport API provides the following utility function for use with `ReactorSubmitMsgOptions`.

METHOD NAME	DESCRIPTION
Clear	Clears the `ReactorSubmitMsgOptions` class. Useful for class reuse.

Table 49: `ReactorSubmitMsgOptions` Utility Method

6.6.1.6   ReactorChannel.Submit(Msg...) Example

The following example shows typical use of **ReactorChannel.Submit**.

```
Msg MarketPriceRefreshMsg = new Msg();
MarketPriceRefreshMsg.Clear();
MarketPriceRefreshMsg.DomainType = (int)Rdm.DomainType.MARKET_PRICE;
MarketPriceRefreshMsg.MsgClass = MsgClasses.REFRESH;
MarketPriceRefreshMsg.ApplyHasMsgKey();
MarketPriceRefreshMsg.StreamId = msg.StreamId;
MarketPriceRefreshMsg.MsgKey.ApplyHasName();
MarketPriceRefreshMsg.MsgKey.Name = msg.MsgKey.Name;
MarketPriceRefreshMsg.MsgKey.ApplyHasNameType();
MarketPriceRefreshMsg.MsgKey.NameType = InstrumentNameTypes.RIC;
MarketPriceRefreshMsg.State.DataState(DataStates.OK);
MarketPriceRefreshMsg.State.StreamState(StreamStates.OPEN);
MarketPriceRefreshMsg.State.Code(StateCodes.NONE);
ReactorSubmitOptions submitOptions = new ReactorSubmitOptions();
submitOptions.Clear();
msgEvent.ReactorChannel.Submit((Msg)MarketPriceRefreshMsg, submitOptions, out ReactorErrorInfo
    errorInfo);
```

Code Example 9: `ReactorChannel.Submit(Msg...)` **Example**

6.6.2   Writing Data Using ReactorChannel.Submit(ITransportBuffer...)

The **Reactor** method offers efficient writing of data by using buffers retrieved directly from the Enterprise Transport API transport buffer pool. It also provides additional features not normally available from `ReactorChannel.Submit(IMsg)`, such as buffer packing. When ready to send data, the application acquires a buffer from the Enterprise Transport API pool. This allows the content to be encoded directly into the output buffer, reducing the number of times the content needs to be copied. Once content is encoded and the buffer is properly populated, the application can submit the data to the reactor. The Enterprise Transport API will ensure that successfully submitted buffers reach the network. Applications can also pack multiple messages into a single buffer by following a similar process as described above, however instead of getting a new buffer for each message the application uses the reactor's pack function instead. The following flow chart depicts the typical write process.
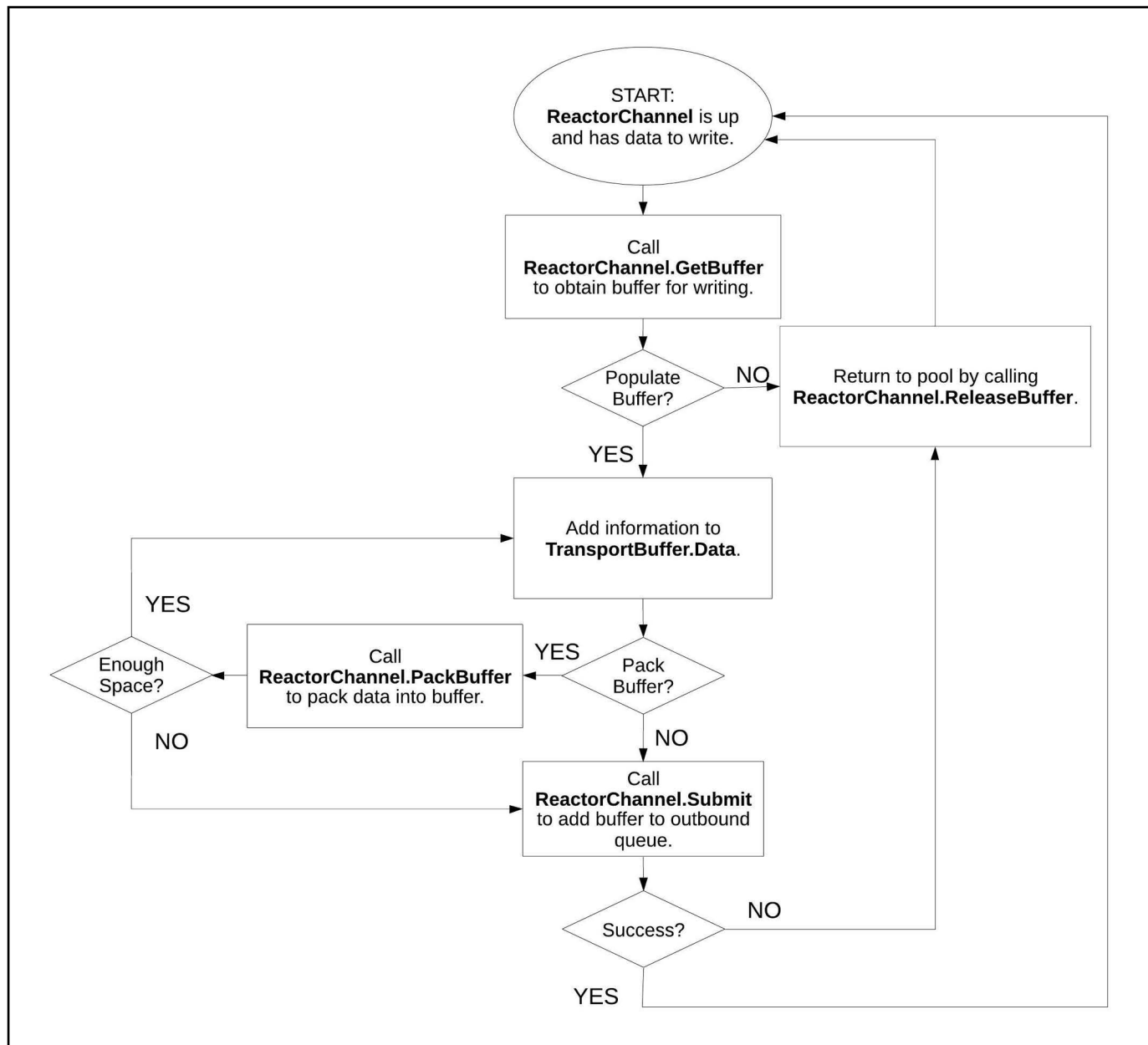
Figure 7. Flow Chart for writing data using ReactorChannel.Submit(ITransportBuffer...)

6.6.2.1   Obtaining a Buffer: Overview

Before you can submit information, you must obtain a buffer from the internal Enterprise Transport API buffer pool, as described in the Enterprise Transport API C# Edition *Developers Guide*. After acquiring the buffer via `ReactorChannel.GetBuffer`, you can populate the `Buffer.Data`. If the buffer is not used or the `Reactor` method call fails, the buffer must be released back into the pool to ensure proper reuse and cleanup. If the buffer is successfully passed to `Reactor`, the reactor will return the buffer to the pool.

The number of buffers made available to an `ReactorChannel` is configurable through the `ReactorConnectOptions` or `ReactorAcceptOptions`. For more information about available `Reactor.Connect()` and `Reactor.Accept()` options, refer to Section 6.4.1.2 and Section 6.4.1.7.

6.6.2.2   Obtaining a Buffer: ReactorChannel Buffer Management Methods

METHOD NAME	DESCRIPTION
ReactorChannel.GetBuffer	Obtains a buffer of the requested size from the buffer pool.
	If the requested size is larger than the `MaxFragmentSize`, the transport will create and return the buffer to the user. When written, this buffer will be fragmented by the `ReactorChannel.Submit` method (for further details, refer to Section 6.6.2.4).
	Because of some additional book keeping required when packing, the application must specify whether a buffer should be 'packable' when calling `ReactorChannel.GetBuffer`. For more information on packing, refer to Section 6.6.2.11.
	For performance purposes, an application is not permitted to request a buffer larger than `MaxFragmentSize` and have the buffer be 'packable.'
	If the buffer is not used or the `ReactorChannel.Submit` call fails, the buffer must be returned to the pool using `ReactorChannel.ReleaseBuffer`. If the `ReactorChannel.Submit` call is successful, the buffer will be returned to the correct pool by the transport.
	This method calls the `IChannel.GetBuffer` method which has its use and return values described in the Enterprise Transport API C# Edition *Developers Guide*.
ReactorChannel.ReleaseBuffer	Releases a buffer back to the correct pool. This should only be called with buffers that originate from `ReactorChannel.GetBuffer` and are not successfully passed to `ReactorChannel.Submit`.
	This method calls the Enterprise Transport API `ReactorChannel.ReleaseBuffer` method which has its use and return values described in the Enterprise Transport API C# Edition *Developers Guide*.

Table 50: Reactor Buffer Management Methods

6.6.2.3   Obtaining a Buffer: ReactorChannel.GetBuffer Return Values

The following table defines return and error code values that can occur while using `ReactorChannel.GetBuffer`.

RETURN CODE	DESCRIPTION
Valid buffer returned Success Case	An `Buffer` is returned to the user. The `Buffer.Length` indicates the number of bytes available to populate and the `Buffer.Data` provides a for population.
NULL buffer returned Error Code: NO_BUFFERS	NULL is returned to the user. This value indicates that there are no buffers available to the user. See `ErrorInfo` content for more details. This typically occurs because all available buffers are queued and pending flushing to the connection. The `ReactorChannel.Ioctl` function can be used to increase the number of `GuaranteedOutputBuffers` (for details, refer to Section 6.8).
NULL buffer returned Error Code: FAILURE	NULL is returned to the user. This value indicates that some type of general failure has occurred. The `ReactorChannel` should be closed.
NULL buffer returned Error Code: INIT_NOT_INITIALIZED	Indicates that the underlying Transport has not been initialized. See the `ErrorInfo` content for more details.

Table 51: `ReactorChannel.GetBuffer` **Return Values**

6.6.2.4   Writing Data: Overview

After an `Buffer` is obtained from `ReactorChannel.GetBuffer` and populated with the user's data, the buffer can be passed to the `ReactorChannel.Submit` method. This method manages queuing and flushing of user content. It will also perform any fragmentation or compression. If an unrecoverable error occurs, any `Buffer` that has not been successfully passed to `ReactorChannel.Submit` should be released to the pool using `ReactorChannel.ReleaseBuffer`. Section 6.6.2.5 describes the `ReactorChannel.Submit` method and its associated parameters.

6.6.2.5   Writing Data: ReactorChannel.Submit(ITransportBuffer...) Method

METHOD NAME	DESCRIPTION
Submit	Writes data. This method expects the buffer to be properly populated. This method calls the Enterprise Transport API `Write` method and also triggers the `Flush` method (described in the Enterprise Transport API C# Edition *Developers Guide*). This method allows for several modifications and additional parameters to be specified via the `ReactorChannel.Submit` class, defined in Section 6.6.1.2. For a list of return codes, refer to Section 6.6.2.7.

Table 52: `ReactorChannel.Submit` **Method**

6.6.2.6   Writing Data: Reactor Submit Options

For a list of submit options and their descriptions for use with **ReactorChannel.Submit**, refer to Section 6.6.1.2.

6.6.2.7   Writing Data: ReactorChannel.Submit(ITransportBuffer...) Return Codes

The following table defines the return codes that can occur when using **ReactorChannel.Submit**.

RETURN CODE	DESCRIPTION
ReactorReturnCode.SUCCESS	Indicates that the **Submit** method has succeeded. The **Buffer** will be released by the Enterprise Transport API Reactor.
ReactorReturnCode.WRITE_CALL_AGAIN	Indicates that a large buffer could not be fully written with this **Submit** call. This is typically due to all pool buffers being unavailable. The **Submit** will flush for the user to free up buffers. The application can optionally use **ReactorChannel.Ioctl** to increase the number of available pool buffers. After pool buffers become available again, the same buffer should be used to call **ReactorChannel.Submit** an additional time (using the same priority level for proper ordering of each fragment). This will continue the fragmentation process from where it left off. If the application does not subsequently pass the buffer to **ReactorChannel.Submit**, the application should release it by calling **ReactorChannel.ReleaseBuffer**.
ReactorReturnCode.FAILURE	Indicates that a general write failure has occurred. The **ReactorChannel** should be closed. The application should release the **Buffer** by calling **ReactorChannel.ReleaseBuffer**.

Table 53: **ReactorChannel.Submit** Return Codes

6.6.2.8   Writing Data: ReactorSubmitOptions Utility Function

For a details on the on the utility method for use with **ReactorChannel.Submit**, refer to Section 6.6.1.5.

6.6.2.9   Example: ReactorChannel.GetBuffer and Reactor Example

The following example shows typical use of **ReactorChannel.GetBuffer** and **ReactorChannel.Submit** methods.

```
ITransportBuffer? msgBuf = chnl.GetBuffer(STATUS_MSG_SIZE, false, out errorInfo!);
        if (msgBuf == null)
        {
            return ReactorReturnCode.FAILURE;
        }

        // encode directory close
        m_EncodeIter.Clear();
        CodecReturnCode ret = m_EncodeIter.SetBufferAndRWFVersion(msgBuf, chnl.MajorVersion,
    chnl.MinorVersion);
        if (ret != CodecReturnCode.SUCCESS)
        {
            errorInfo = new ReactorErrorInfo();
            errorInfo.Error.Text = $"EncodeIterator.SetBufferAndRWFVersion() failed with return
    code: {ret.GetAsString()}";
            return ReactorReturnCode.FAILURE;
```

```
            }

            m_DirectoryStatus.StreamId = directoryReqInfo.DirectoryRequest.StreamId;
            m_DirectoryStatus.HasState = true;
            m_DirectoryStatus.State.StreamState(StreamStates.CLOSED);
            m_DirectoryStatus.State.DataState(DataStates.SUSPECT);
            m_DirectoryStatus.State.Text().Data("Directory stream closed");
            ret = m_DirectoryStatus.Encode(m_EncodeIter);
            if (ret != CodecReturnCode.SUCCESS)
            {
                errorInfo = new ReactorErrorInfo();
                errorInfo.Error.Text = "DirectoryStatus.Encode failed";
                return ReactorReturnCode.FAILURE;
            }

            // send close status
            ReactorReturnCode reactorRet = chnl.Submit(msgBuf, m_SubmitOptions, out errorInfo!);
// check return code
switch (ret)
{
    case ReactorReturnCode.SUCCESS:
        // successful write, nothing left to do
        return ReactorReturnCode.SUCCESS;
    break;
    case ReactorReturnCode.FAILURE:
        // an error occurred, need to release buffer
        chnl.ReleaseBuffer(msgBuffer, out errorInfo);
    break;
    case ReactorReturnCode.WRITE_CALL_AGAIN:
        // large message couldn't be fully written with one call, pass it to submit again
        ret = chnl.Submit(msgBuffer, submitOpts, out errorInfo);
    break;
}
```

Code Example 10: Writing Data Using `ReactorChannel.Submit`, `ReactorChannel.GetBuffer`, and `ReactorChannel.ReleaseBuffer`

6.6.2.10   Packing Additional Data into a Buffer

If an application is writing many small buffers, it may be advantageous to combine the small buffers into one larger buffer. This can increase efficiency of the transport layer by reducing the overhead associated with each write operation, although it may add to the latency associated with each smaller buffer.

It is up to the writing application to determine when to stop packing, and the mechanism used can vary greatly. A simple algorithm can pack a fixed number of messages each time. A slightly more complex technique could use the length returned from **ITransportBuffer.Length** to determine the amount of space remaining and pack until the buffer is nearly full. Both of these mechanisms can introduce a variable amount of latency as they both depend on the rate of arrival of data (e.g., the packed buffer will not be written until enough data arrives to fill it). One way of balancing this is to employ a timer, used to limit the amount of time a packed buffer is held. If the buffer is full prior to the timer expiring, the data is written. However, when the timer expires the buffer will be written regardless of the amount of data it contains. This can help limit latency by specifying a limit to the time data is held (via use of the timer).

METHOD NAME	DESCRIPTION
ReactorChannel.PackBuffer	Packs the contents of a passed-in **Buffer** and returns the amount of available bytes remaining in the buffer for packing. An application can use the length returned to determine the amount of space available to continue packing buffers into.
	For a buffer to allow packing, it must be requested from **ReactorChannel.GetBuffer** as 'packable' and cannot exceed the **MaxFragmentSize**.
	ReactorChannel.PackBuffer return values are defined in Section 6.6.2.11.
	This method calls the **PackBuffer** method as described in the Enterprise Transport API C# Edition *Developers Guide*.

Table 54: `ReactorChannel.PackBuffer` Method

6.6.2.11   ReactorChannel.PackBuffer Return Values

The following table defines return and error code values that can occur when using **ReactorChannel.PackBuffer**.

RETURN CODE	DESCRIPTION
Positive value or ReactorReturnCode.SUCCESS Success Case	The amount of available bytes remaining in the buffer for packing.
Negative value Failure Case	This value indicates that some type of failure has occurred. If the FAILURE return code is returned, the **ReactorChannel** should be closed.

Table 55: `ReactorChannel.PackBuffer` Return Values

6.6.2.12   Example: ReactorChannel.GetBuffer, ReactorChannel.PackBuffer, and ReactorChannel.Submit

The following example shows typical use of **ReactorChannel.GetBuffer**, **ReactorChannel.PackBuffer**, and
ReactorChannel.Submit(ITransportBuffer...).

```
ITransportBuffer buffer = reactorChannel.GetBuffer(100, true, out errorInfo);
        EncodeIterator encIter = new EncodeIterator();

        encIter.SetBufferAndRWFVersion(buffer, reactorChannel.MajorVersion,
   reactorChannel.MinorVersion);
        EncodeMsgIntoBuffer(encIter, buffer);
        chnl.PackBuffer(buffer, out errorInfo);

        encIter.Clear();
        encIter.SetBufferAndRWFVersion(buffer, reactorChannel.MajorVersion,
   reactorChannel.MinorVersion);
        EncodeMsgIntoBuffer(encIter, buffer);
        chnl.PackBuffer(buffer, out errorInfo);

        encIter.Clear();
        encIter.SetBufferAndRWFVersion(buffer, reactorChannel.MajorVersion,
   reactorChannel.MinorVersion);
        EncodeMsgIntoBuffer(encIter, buffer);

        submitOptions.Clear();
        submitOptions.WriteArgs.Priority = WritePriorities.HIGH;
        submitOptions.WriteArgs.Flags = WriteFlags.DIRECT_SOCKET_WRITE;

        reactorChannel.Submit(buffer, submitOptions, out errorInfo);
```

Code Example 11: Message Packing using `ReactorChannel.PackBuffer`

6.7   Cloud Connectivity

For details on workflows and routines associated with connecting to the cloud, refer to Chapter 7.

You use the **Reactor.QueryServiceDiscovery** method to query service endpoints from the EDP-RT service.

6.7.1   Querying Service Discovery

6.7.1.1   QueryServiceDiscovery Method

METHOD	DESCRIPTION
QueryServiceDiscovery	Queries service endpoints from the Real-Time Optimized service according to the **ReactorServiceDiscoveryOptions** that you specify (listed in Section 6.7.1.2). Error handling is managed by the **ErrorInfo** class.

Table 56: **Reactor.QueryServiceDiscovery** Method

6.7.1.2   ReactorServiceDiscoveryOptions

MEMBER	DESCRIPTION
Audience	Optional and only used with Version 2 OAuth ClientCredentials with JWT logins. A **Buffer** specifies the JWT's audience claim field. By default, the Enterprise Transport API uses https://login.ciam.refinitiv.com/as/token.oauth2
ClientId	**Required**. A **Buffer** that specifies a unique ID defined for an application making a request to the token service.
ClientJwk	**Required** for Version 2 OAuth ClientCredentials with JWT logins. A **Buffer** that contains the JWK-formatted private key associated with the Service Account. For further information, refer to the ClientSecret member in this section.
ClientSecret	A **Buffer** that specifies the client secret (if one exists) used by the OAuth client to authenticate to the authorization Server.
DataFormat	Optional. An enumeration that specifies the desired data format to use when retrieving service endpoints from the service discovery. For available values, refer to Section 6.7.1.4.
ProxyHostName	Optional. A **Buffer** that specifies a proxy server hostname.
ProxyPort	Optional. A **Buffer** that specifies a proxy server port.
ProxyUserName	Optional. A **Buffer** that specifies a username to perform authorization with a proxy server.
ProxyPasswd	Optional. A **Buffer** that specifies a password to perform authorization with a proxy server.
pServiceEndpointEventCallback	A callback function that receives **ReactorServiceEndpointEvent**s. Applications can take service endpoint information from the callback to get an endpoint and establish a connection to the service.
Transport	Optional. An enumeration that specifies the desired transport protocol to retrieve service endpoints from the service discovery. For available values, refer to Section 6.7.1.3.
UserSpecObject	Optional. A user-specified pointer which is set on the **ReactorServiceEndpointEvent**. Also refer to Section 6.7.1.5.

Table 57: **ReactorServiceDiscoveryOptions** Members

6.7.1.3   QueryServiceDiscovery Transport Protocol Enumerations

ENUMERATED NAME	DESCRIPTION
ReactorDiscoveryTransportProtocol.RD_TP_INIT = 0	Specifies that the transport's protocol is unknown.
ReactorDiscoveryTransportProtocol.RD_TP_TCP = 1	Specifies that the service discovery should use the TCP transport protocol.

Table 58: `ReactorDiscoveryTransportProtocol` Enumerations

6.7.1.4   ReactorDiscoveryDataFormatProtocol Enumerations

ENUMERATED NAME	DESCRIPTION
RD_DP_INIT = 0	Specifies that the transport's data format is unknown.
RD_DP_RWF = 1	Specifies that the service discovery should use the RWF data format.
RD_DP_JSON2 = 2	Specifies that the service discovery should use the `tr_json2` data format

Table 59: `ReactorDiscoveryDataFormatProtocol` Enumerations

6.7.1.5   ReactorServiceEndpointEvent

MEMBER	DESCRIPTION
ServiceEndpointInfoList	Lists the service endpoints associated with this event. See also Section 6.7.1.6.
UserSpecObject	Optional. A user-specified pointer associated with this `ReactorServiceEndpointEvent`.

Table 60: `ReactorServiceEndpointEvent` Members

6.7.1.6   ReactorServiceEndpointInfo

`ReactorServiceEndpointInfo` represents service endpoint information.

MEMBER	DESCRIPTION
DataFormatList	A `Buffer` that contains a list of data formats used by the transport.
EndPoint	A `Buffer` that specifies the domain name of the service access endpoint.
LocationList	A `Buffer` object that specifies a list of service locations.
Port	A `Buffer` that specifies the port number used to establish connection.
Provider	A `Buffer` that specifies a public cloud provider.
Transport	A `Buffer` that specifies the transport type used to access the service.

Table 61: `ReactorServiceEndpointInfo` Members

6.7.2   OAuth Credential Management

6.7.2.1   ReactorOAuthCredential Class

You use the `ReactorOAuthCredential` structure to certify OAuth user credentials when connecting to the cloud. `ReactorOAuthCredential` includes the following properties:

PROPERTY	DESCRIPTION
Audience	Optional and only used with Version 2 OAuth ClientCredentials with JWT logins. An Buffer that specifies the JWT's audience claim field. By default, the Enterprise Transport API uses https://login.ciam.refinitiv.com/as/token.oauth2
ClientId	**Required**. A `Buffer` that specifies an authentication parameter. • Version 2 authentication: a unique ID defined for the application that makes the request and is provisioned as part or a service account for the login. For further information, refer to Section 7.4.
ClientJwk	**Required** for Version 2 OAuth ClientCredentials with JWT logins. A `Buffer` that contains the JWK-formatted private key is associated with the Service Account. For further information, refer to Section 7.4.
ClientSecret	**Required** for Version 2 OAuth ClientCredentials logins. A `Buffer` that specifies the Service Account "secret" for authentication. For further information, refer to Section 7.4.
ReactorOAuthCredentialEventCallback	A callback function that receives the `ReactorOAuthCredentialEvent` to specify the `ClientSecret`. If `ReactorOAuthCredentialEventCallback` is specified, the Value Added Components Reactor does not store the `ClientSecret`. In which case, the application must supply the `ClientSecret` whenever receiving this callback. For details on this process, refer to Section 7.4.1.
TokenScope	A `Buffer` that specifies the user's resource scope that defines the type of data the user accesses in the cloud. For further details on token scopes, refer to the Delivery Platform APIs tutorial called *Authorization - All about tokens* in the Developer Community Portal. By default, the Enterprise Transport API uses the scope: `trapi.streaming.pricing.read`.

Table 62: `ReactorOAuthCredential` Class Properties

6.7.2.2   ReactorOAuthCredentialEventCallback

The `IReactorOAuthCredentialEventCallback` is used to communicate `ReactorOAuthCredentialRenewal` in order to obtain sensitive information from the application.

MEMBER	DESCRIPTION
IReactorOAuthCredentialEventCallback	Specifies OAuth credential renewal information associated with this event.

Table 63: `ReactorOAuthCredentialEventCallback` Members

6.7.2.3   ReactorOAuthCredentialRenewal

MEMBER	DESCRIPTION
Audience	Optional and only used with Version 2 oAuth ClientCredentials with JWT logins. A **Buffer** that specifies the JWT's audience claim field. By default, the Enterprise Transport API uses https://login.ciam.refinitiv.com/as/token.oauth2
ClientId	**Required.** A **Buffer** that specifies an authentication parameter. • Version 2 authentication: a unique ID defined for the application that makes the request and is provisioned as part or a service account for the login. For further information, refer to Section 7.4.
ClientJwk	**Required** for Version 2 oAuth ClientCredentials with JWT logins. A **Buffer** that contains the JWK-formatted private key is associated with the Service Account. For further information, refer to Section 7.4.
ClientSecret	Required for Version 2 oAuth ClientCredentials logins. A **Buffer** that specifies the Service Account "secret" for authentication. For further information, refer to Section 7.4.
TokenScope	A **Buffer** that specifies the scope of the generated token.

Table 64: `ReactorOAuthCredentialRenewal` Properties

6.7.2.4   ReactorOAuthCredentialRenewal Options

OPTION	DESCRIPTION
ReactorAuthTokenEventCallback	A callback function that receives **ReactorAuthTokenEvents**. The Reactor requests a token for the Consumer (i.e., disabling watchlist) and NiProvider applications to send login requests and reissues with the token. **ReactorAuthTokenEventCallback** is needed only when changing a password without a channel in order to get a response from the request. The application does not have to send a login reissue in this case.
RenewalModes	A **ReactorOAuthCredentialRenewalMode** that specifies the mode in which the Enterprise Transport API submits OAuth credential renewals. For available ENUMs and their descriptions, refer to Section 6.7.2.5.

Table 65: ReactorOAuthCredentialRenewalOptions

6.7.2.5   ReactorOAuthCredentialRenewalModes Enum

MODE	DESCRIPTION
NONE	This value means that the renewal mode is unspecified. Use this renewal mode when normally submitting a client secret to obtain an access token
CLIENT_SECRET	Use this renewal mode when normally submitting a client secret to obtain an access token.
CLIENT_JWK	Use this renewal mode when normally submitting a client JWK to obtain an access token.

Table 66: ReactorOAuthCredentialRenewalMode Enum Values

6.8   Reactor Utility Methods

The Transport API Reactor provides several additional utility functions. These functions can be used to query more detailed information for a specific connection or change certain **ReactorChannel** parameters during run-time. These functions are described in Section 6.8.1 - Section 6.8.3.

6.8.1   General Reactor Utility Methods

METHOD NAME	DESCRIPTION
Info	Allows the application to query **ReactorChannel** negotiated parameters and settings and retrieve all current settings. This includes **MaxFragmentSize** and negotiated compression information as well as many other values. For a full list of available settings, refer to the **ReactorChannelInfo** class defined in Section 6.8.2. This method calls the Enterprise Transport API **GetChannelInfo** method which has its use and return values described in the Enterprise Transport API C# Edition *Developers Guide*.
IOCtl	Allows the application to change various settings associated with the **ReactorChannel**. The available options are defined in Section 6.8.3. This method calls the **IOCtl** method which has its use and return values described in the Enterprise Transport API C# Edition *Developers Guide*.

Table 67: Reactor Utility Methods

6.8.2   ReactorChannelInfo Class Members

The following table describes the values available to the user through using the **ReactorGetChannelInfo** method. This information is returned as part of the **ReactorChannelInfo** class.

STRUCTURE MEMBER	DESCRIPTION
ChannelInfo	Returns the underlying **IChannel** information. This includes **MaxFragmentSize**, number of output buffers, compression information, and more. The **ChannelInfo** method class is fully described in the Enterprise Transport API C# Edition *Developers Guide*.
	Structure containing the current channel's Preferred Host configuration, as well as the remaining time (in seconds) before the next configured preferred host timer will be triggered. See Section 6.4.3.

Table 68: ReactorChannelInfo Structure Members

6.8.3   ReactorChannel.IOCtl Option Values

There are currently no **Reactor** or **ReactorChannel** specific codes for use with the **IOCtl** method. Reactor-specific codes may be added in the future. The application can still use any of the codes allowed with **IOCtl**, which are documented in the Enterprise Transport API C# Edition *Developers Guide*.

7   Consuming Data from the Cloud

7.1   Overview

You can use the Enterprise Transport API to consume data from a cloud-based LSEG Real-Time Advanced Distribution Server. The API interacts with cloud-based servers using the following workflows:

- Credential Management (for details, refer to Section 7.3)

- Service Discovery (for details, refer to Section 7.5)

- Consuming Market Data (for details, refer to Section 7.6)

There are two versions of login credentials for the Delivery Platform: RTSDK C# only supports Version 2 authentication.

Version 2 Authentication, also known as "V2 auth", "OAuthClientCredentials" or "V2 Client Credentials", uses OAuth2.0 Client Credentials grant to obtain an access token. Requires a Service Account consisting of client ID and client Secret. For details, refer to Section 7.4.

NOTE: Version 2 Authentication is available as an **Early Access** feature to API developers to preview changes required to use this new authentication mechanism. Please note that the ability to setup Service Accounts to use this authentication is forthcoming.

The Enterprise Transport API will determine which authentication version to use based on the inputs. By default, for cloud connections the Enterprise Transport API connects to a server in the **us-east-1** cloud location.

For further details on Real-Time as it functions in the cloud, refer to the *Real-Time — Optimized: Installation and Configuration for Client Use*.

7.2   Encrypted Connections

When connecting to an LSEG Real-Time Advanced Distribution Server in the cloud, you must use an encrypted connection type (for details on connection types, refer to the *ETA C# Developer Guide*).

7.3   Credential Management

By default, the Enterprise Transport API will store all credential information. In order to use secure credential storage, a callback function can be specified by the user. If a callback function is specified, credentials are not stored in API; instead, application is called back whenever credentials are required.

When configuring the **ReactorChannel**, if **ReactorOAuthCredential.ReactorOAuthCredentialEventCallback** is specified, the API will call the user back whenever credentials are required. This callback must call **Reactor.SubmitOAuthCredentialRenewal** to submit the updated credentials.

7.4   Version 2 Authentication Using OAuth Client Credentials

Version 2 OAuth Client Credentials requires a client ID and client secret, or private JWK for JWT, or a client ID and private client JWK for OAuth Client Credentials with JWT. Version 2 will generate an Access Token.

Once connected to Real-Time — Optimized Advanced Distribution Server, the login session to the Advanced Distribution Server will remain valid until the consumer disconnects or is disconnected from Real-Time — Optimized. The API will only re-request an Access Token in the following cases:

- When the consumer disconnects and goes into a reconnection state.

- If the **Channel** stays in reconnection long enough to get close to the expiry time of the Access Token.

Due to the above changes, credentials are managed independently per reactor channel. Channels do not share credentials.

7.4.1   Configuring and Managing Version 2 Credentials

The client ID and client secret or private JWK must be set on the `ReactorOAuthCredential` as described in Section 6.7.2.1 of the *Enterprise Transport API C# Edition Value Added Developers Guide*. The `Reactor` will handle the credentials with a `ReactorOAuthCredentialEvent` callback for credentials if the user does not wish for the `Reactor` to store them.

7.4.1.1   JWT Credentials Handling

Version 2 OAuth Client Credentials with JWT requires a JWK public/private pair to be generated and registered with LSEG via the Platform Admin UI. The API will use a private JWK to create and sign a JWT request, which will be sent to retrieve an access token. The JWK will be handled by the API the exact same way as a client secret above. For more information about the Platform Admin UI, refer to the Real-Time — Optimized documentation in the LSEG Developers portal.

NOTE: Follow best practices for securely storing and retrieving JWK.

7.4.2   Version 2 OAuth Client Credentials Token Lifespan

Version 2 will produce a single Access Token, which will be valid for the length of the entire `expires_in` field in the token. This Access Token is used by the API to perform service discovery, and to connect to Real-Time — Optimized.

The API will re-request a token on reconnect, and will use that token for all reconnect attempts until a short time prior to expiry. At that time, the API will get a new token for reconnection use.

7.5   Service Discovery

After obtaining a token (for details, refer to), the Enterprise Transport API can perform a service discovery against the Delivery Platform to obtain connection details for the Real-Time — Optimized. The Enterprise Transport API C# Edition uses the `Reactor.QueryServiceDiscovery` method to submit a service discovery (refer to Section 6.2.1 for a description of this reactor method).

In response to a service discovery, the Delivery Platform returns transport and data format protocols and a list of hosts and associated ports for the requested service(s) (i.e., an LSEG Real-Time Advanced Distribution Server running in the cloud or endpoint). LSEG provides multiple cloud locations based on region, which is significant in how the Enterprise Transport API chooses the IP address and port to use when connecting to the cloud.

From the list sent by the Delivery Platform, the Enterprise Transport API identifies a Real-Time — Optimized endpoint with built-in resiliency whose regional location matches the API's location setting in `ReactorConnectInfo` (for details, refer to Section 6.4.1.3). If you do not specify a location, the Enterprise Transport API defaults to the **us-east-1** cloud location. An endpoint with built-in resiliency lists multiple locations in its location field (e.g., `location: [us-east-1a, us-east-1b]`). If multiple endpoints are configured for failover, the Enterprise Transport API chooses to connect to the first endpoint listed.
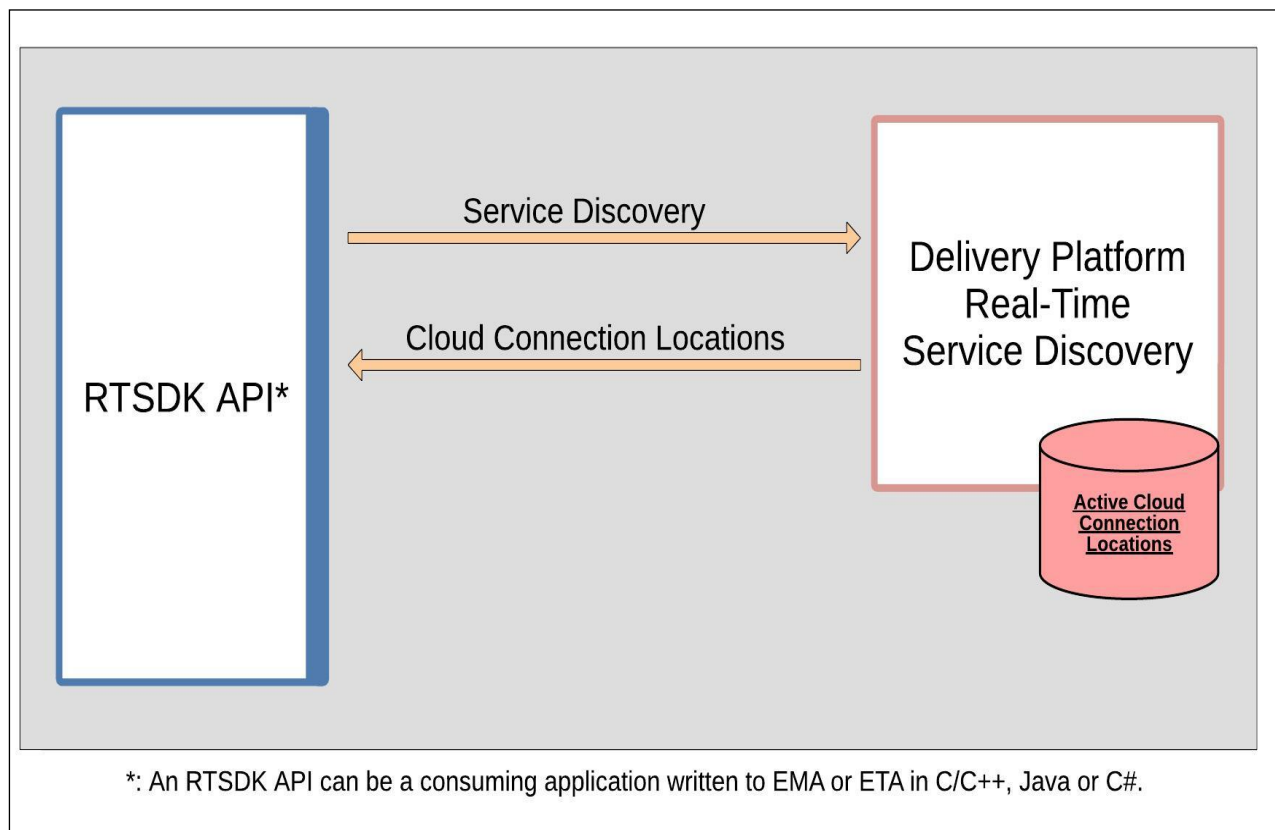
Figure 8. Service Discovery

7.6 Consuming Market Data

After obtaining its login token (for details, refer to) and running a service discovery (for details, refer to Section 7.5), the API can connect to the LSEG Real-Time Advanced Distribution Server in the cloud and obtain market data.

*: An RTSDK API can be a consuming application written to EMA or ETA in C/C++, Java or C#.

DP: Delivery Platform
ADS: LSEG Real-Time Advanced Distribution Server
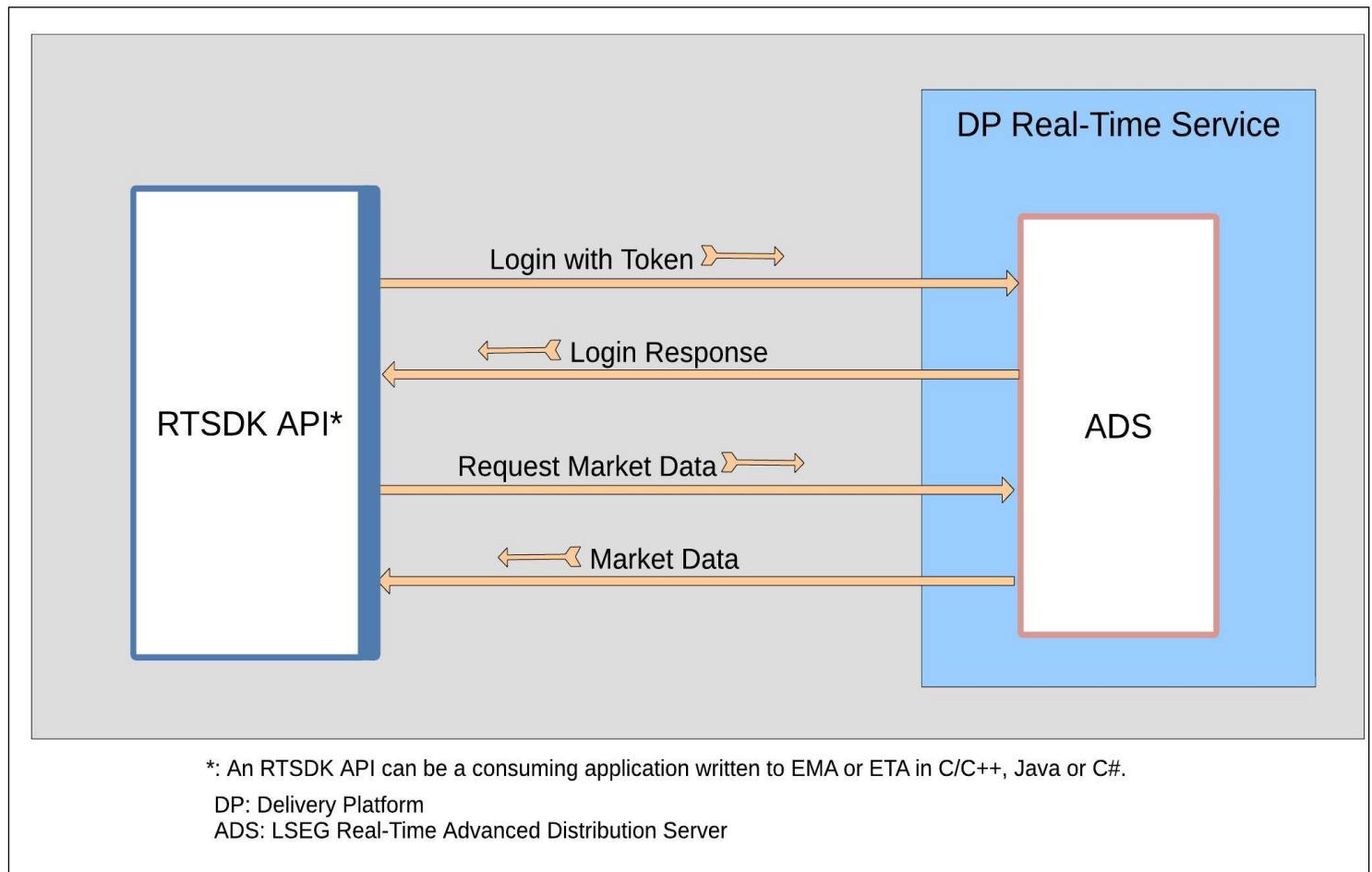
7.7 HTTP Error Handling for Reactor Token Reissues

The Enterprise Transport API supports handling for the following HTTP error codes from the API gateway:

- 300 Errors:
  - Perform URL redirect for 301, 302, 307 and 308 error codes
  - Retry the request to the API gateway for all other error codes
- 400 Errors:
  - Stop retry the request for error codes 403, 404, 410, and 451
  - Retry the request to the API gateway for all other error codes
- 500 Errors:
  - Retry the request to the API gateway for all error codes

7.8 Cloud Connection Use Cases

You can connect to the cloud and consume data according to the following use cases:

- Start to finish session management using Reactor watchlist (for details, refer to Section 7.8.1)

- Session management using Reactor with watchlist disabled (for details, refer to Section 7.8.2Section 6.8.2)

- Explicit service discovery use case (for details, refer to Section 7.8.3)

7.8.1 Session Management Using Reactor Watchlist

In this use case, when using Enterprise Transport API Reactor with watchlist enabled in conjunction with session management, the API manages the entire connection from start to finish. To enable session management, set **EnableSessionManagement** in `ReactorConnectInfo`.

The API exhibits the following behavior when enabling both session management and watchlist features:

1. Obtains a token (according to the details in Section 7.4).

2. Performs an implicit service discovery and chooses and endpoint based on specified or defaulted location (us-east-1) to connect to (according to the details in Section 7.5).

3. Consumes market data from either discovered or specified endpoint (according to the details in Section 7.6).

Possible ways to use session management with watchlist enabled include:

- By default, a default location is used to do implicit service discovery as detailed above and a LSEG resilient endpoint is automatically chosen for connectivity as detailed above. See Section 7.5 for details on service discovery.

- Application may specify a location to override default location. An LSEG resilient endpoint is automatically chosen from the specified region.

- Application may also choose to not consider results of automatic implicit service discovery by setting host and port to connect to. In this case, service discovery still occurs, however, results are ignored. Host and port information must be set in `ReactorConnectInfo` instance from `ReactorConnectOptions.ConnectionList` list.

7.8.2 Session Management Using Reactor with Watchlist Disabled

When connecting to an LSEG Real-Time Advanced Distribution Server in the cloud with the watchlist disabled (the default), the API:

- Obtains a token (according to the details in Section 7.4)

- If needed, queries service discovery (according to the details in Section 7.5)

Reactor initially handles the RDM Login request, with the application handling subsequent Login Reissues using renewed access tokens.

To support this use case, you must configure session management (i.e., in `ReactorConnectInfo` objects, set **EnableSessionManagement**).

7.8.3 Explicit Service Discovery Use Case

Application has the option to do a service discovery, parse the results, and choose an endpoint to pass into API. Depending on whether or not the watchlist is enabled, API will behave according to use cases already described in this section to manage session:

1. Obtains a token (according to the details in Section 7.4).

2. Queries service discovery (according to the details in Section 7.5).

7.9   Logging of Authentication and Service Discovery Interaction

you must enable the **ReactorOptions.EnableRestLogStream** option to enable logging REST request and response to a stream which specifies a stream with the **ReactorOptions.RestLogOutputStream** option.

7.9.1   Logged Request Information

With logging turned on in the fashion mentioned in Section 7.9, the Enterprise Transport API writes the following request information in the log:

```
Request:
- Time stamp
- The Name of the class and method that made the request
- Request method
- URI
- Request headers
- Proxy information (if used)
- Body of request as set of pairs parameter_name: parameter_value
```

NOTE: If the request contains the **client_secret** parameter, the Enterprise Transport API uses a placeholder instead of the real value of the respective parameter (thus indicating that the value was present).

7.9.2   Logged Response Information

With logging turned on in the fashion mentioned in Section 7.9, the Enterprise Transport API writes the following response information in the log:

```
Response:
- Time stamp
- The Name of the class and method that received the response
- Response status code
- Response headers
- Body of response in string format
```

8   Administration Domain Models Detailed View

8.1   Concepts

Administration Domain Model Representations are RDM-specific representations of OMM administrative domain models. This Value Added Component contains classes and interfaces that represent messages within the Login, Source Directory, and Dictionary domains (discussed in Table 69). This component also handles all encoding and decoding functionality for these domain models, so the application needs only to manipulate the message's class members to send or receive content. Such functionality significantly reduces the amount of code an application needs to interact with OMM devices (i.e., LSEG Real-Time Distribution System infrastructure), and also ensures that encoding/decoding for these domain models follow OMM-specified formatting rules. Applications can use this Value Added Component directly to help with encoding, decoding, and representation of these domain models.

Where possible, the members of an Administration Domain Model Representation are represented in the structure with the same `DataType` that is specified for the element by the domain model. In cases where multiple elements are part of a more complex container such as a `Map` or `ElemenList`, the elements are added to these complex container types.

The Enterprise Transport API C# Edition *LSEG Domain Model Usage Guide* defines and describes all domain-specific behaviors, usage, and details.

DOMAIN	PURPOSE
Dictionary	Provides dictionaries that may be needed when decoding data. Though use of the Dictionary domain is optional, LSEG recommends that provider applications support the domain's use. The Dictionary domain is considered an administrative domain. Many LSEG components require this content and expect it to follow the domain model definition. For further details refer to Section 8.5.
Login	Authenticates users and advertises/requests features that are not specific to a particular domain. Use of and support for this domain is required for all OMM applications. Login is considered an administrative domain. Many LSEG components require this content and expect it to conform to the domain model definition. For further details refer to Section 8.3.
Source Directory	Advertises information about available services and their state, quality of service, and capabilities. This domain also conveys any group status and group merge information. Interactive and non-interactive provider applications require support for this domain. LSEG strongly recommends that consumers request this domain. Source Directory is considered an administrative domain, and many LSEG components expect this content and require it to conform to the domain model definition. For further details, refer to Section 8.4.

Table 69: Domains Representations in the Administration Domain Model Value Added Component

8.2   Message Base

`MsgBase` is the root abstract class for all Administration Domain Model Representation interfaces. It provides methods for stream identification as well as methods that are common for all domain representations.

8.2.1   Message Base Members

The `MsgBase` abstract class includes the following members and methods:

MEMBER	DESCRIPTION
StreamId	**Required**. A unique signed-integer identifier associated with all messages flowing in the stream. • Positive values indicate a consumer-instantiated stream, typically via a request message. • Negative values indicate a provider-instantiated stream, often associated with Non-Interactive Providers.

Table 70: `MsgBase` Members

8.2.2   Message Base Method

METHOD	DESCRIPTION
Clear()	Clears the object for reuse.

Table 71: MsgBase Method

8.2.3   RDM Message Types

The following table provides a reference mapping between the administrative domain type and the representations provided in this component.

DOMAIN TYPE	MESSAGE TYPE	
DomainType.LOGIN (`LoginMsg`) Refer to Section 8.3	LoginMsgType.REQUEST	LoginRequest
	LoginMsgType.REFRESH	LoginRefresh
	LoginMsgType.STATUS	LoginStatus
	LoginMsgType.CLOSE	LoginClose
	LoginMsgType.CONSUMER_CONNECTION_STATUS	LoginConsumerConnectionStatus
	LoginMsgType.RTT	LoginRTT
DomainType.SOURCE (`DirectoryMsg`) Refer to Section 8.4	DirectoryMsgType.REQUEST	DirectoryRequest
	DirectoryMsgType.REFRESH	DirectoryRefresh
	DirectoryMsgType.UPDATE	DirectoryUpdate
	DirectoryMsgType.STATUS	DirectoryStatus
	DirectoryMsgType.CLOSE	DirectoryClose
	DirectoryMsgType.CONSUMER_STATUS	DirectoryConsumerStatus
DomainType.DICTIONARY (`DictionaryMsg`) Refer to Section 8.5	DictionaryMsgType.REQUEST	DictionaryRequest
	DictionaryMsgType.REFRESH	DictionaryRefresh
	DictionaryMsgType.STATUS	DictionaryStatus
	DictionaryMsgType.CLOSE	DictionaryClose

Table 72: Domain Representations Message Types

8.3 RDM Login Domain

The Login domain registers a user with the system, after which the user can request[1], post[2], or provide[3] OMM content.

- A consumer application must log into the system before it can request or post content.

- A non-interactive provider (NIP) application must log into the system before providing content. An interactive provider application must handle login requests and provide login response messages, possibly using the Data Access Control System to authenticate users.

Section 8.3.1 - Section 8.3.12 detail the layout and use of each message interface in the Login portion of the Administration Domain Message Component.

8.3.1 Login Request

A *Login Request* message is encoded and sent by OMM consumer and non-interactive provider applications. This message registers a user with the system. After receiving a successful login response, applications can then begin consuming or providing additional content. An OMM provider can use the login request information to authenticate users with the Data Access Control System.

The `LoginRequest` represents all members of a login request message and allows for simplified use in OMM applications that leverage RDMs. This structure follows the behavior and layout that is defined in the Enterprise Transport API C# Edition *LSEG Domain Model Usage Guide*.

8.3.1.1 Login Request Properties

MEMBER	DESCRIPTION
LoginAttrib	**Optional**. Contains additional login attribute information. If present, a **Flags** value of **LoginRequestFlags.HAS_ATTRIB** should be specified. For further details, refer to Section 8.3.9.1.
AuthenticationExtended	Optional. If present, a flags value of **LoginRequestFlags.HAS_AUTHNENTICATION_EXTENDED** should be specified. When populated, `AuthenticationExtended` contains additional content that will be passed to the token authenticator as an additional means to verifying a user's identity.
DownloadConnectionConfig	**Optional**. If present, a **Flags** value of **LoginRequestFlags.HAS_DOWNLOAD_CONN_CONFIG** should be specified. If absent, a default value of **0** is assumed. Enabling this option allows the application to download information about other providers on the network. You can use such downloaded information to load balance connections across multiple providers. • **1**: Indicates that the user wants to download connection configuration information. • **0**: Indicates that the user does not want to download connection information.
Flags	**Required**. Indicates presence of optional login request members. For details, refer toSection 8.3.1.2.

Table 73: `LoginRequest` Members

---

1. Consumer applications can request content after logging into the system.
2. Consumer applications can post content (similar to contributions or unmanaged publications) after logging into the system.
3. Non-interactive provider applications.

MEMBER	DESCRIPTION
InstanceId	**Optional**. If present, a **Flags** value of **LoginRequestFlags.HAS_INSTANCE_ID** should be specified. You can use the **InstanceId** to differentiate applications running on the same machine. However, because **InstanceId** is set by the user logging into the system, it does not guarantee uniqueness across different applications on the same machine.
Password	**Optional**. If present, a **Flags** value of **LoginRequestFlags.HAS_PASSWORD** should be specified. Sets the password for logging into the system.
Role	**Optional**. If present, a **Flags** value of **LoginRequestFlags.HAS_ROLE** should be specified. If absent, a default value of **Rdm.Login.RoleTypes.CONS** is assumed. Indicates the role of the application logging onto the system. • **0**: **Rdm.Login.RoleTypes.CONS**, indicates application is a consumer. • **1**: **Rdm.Login.RoleTypes.PROV**, indicates application is a provider.
UserName	**Required**. Populate this member with the username, email address, or user token based on the **UserNameType** specification. If you initialize **LoginRequest** using **InitDefaultRequest**, it uses the name of the user currently logged into the system on which the application runs.
UserNameType	**Optional**. If present, a **Flags** value of **LoginRequestFlags.HAS_USERNAME_TYPE** should be specified. If absent, a default value of **Rdm.Login.UserIdTypes.NAME** is assumed. Possible values: • **Rdm.Login.UserIdTypes.NAME == 1** • **Rdm.Login.UserIdTypes.EMAIL_ADDRESS == 2** • **Rdm.Login.UserIdTypes.TOKEN == 3** • **Rdm.Login.UserIdTypes.COOKIE == 4** • **Rdm.Login.UserIdTypes.AUTHN_TOKEN==5** A type of **Rdm.Login.UserIdTypes.NAME** typically corresponds to a Data Access Control System user name and can to authenticate and permission a user. **Rdm.Login.UserIdTypes.TOKEN** is specified when using the AAA ('triple A') API. The user token is retrieved from the Authentication Manager application. To validate users, a provider application passes this user token to the AAA Gateway. This type of token periodically changes: when it changes, an application can send a login reissue to pass information upstream. For more information, refer to documentation specific to the AAAAPI. **Rdm.Login.UserIdTypes.AUTHN_TOKEN** is specified when using LSEG Real-Time Distribution System Authentication. The authentication token should be specified in the **UserName** member. This type of token can periodically change: when it changes, an application can send a login reissue to pass information upstream. For more information, refer to the *LSEG Real-Time Distribution System Authentication User Manual*.[a]

Table 73: `LoginRequest` **Members (Continued)**

a. For further details on LSEG Real-Time Distribution System Authentication, refer to the *LSEG Real-Time Distribution System Authentication User Manual*, accessible on [MyAccount](#) in the Data Access Control System product documentation set.

8.3.1.2   Login Request Flag Enumeration Values

FLAG ENUMERATION	MEANING
LoginRequestFlags.HAS_ATTRIB	Indicates the presence of **LoginAttrib**.
LoginRequestFlags.HAS_AUTHN_EXTENDED	Indicates the presence of **AuthenticationExtended**.
LoginRequestFlags.HAS_DOWNLOAD_CONN_CONFIG	Indicates the presence of **DownloadConnectionConfig**. If absent, a value of **0** should be assumed.
LoginRequestFlags.HAS_INSTANCE_ID	Indicates the presence of **InstanceId**.
LoginRequestFlags.HAS_PASSWORD	Indicates the presence of **Password**.
LoginRequestFlags.HAS_ROLE	Indicates the presence of **Role**. If absent, a role of **Rdm.Login.RoleTypes.CONS** is assumed.
LoginRequestFlags.HAS_USERNAME_TYPE	Indicates the presence of **UserNameType**. If not present, a **UserNameType** of **Rdm.Login.UserIdTypes.NAME** should be assumed.
LoginRequestFlags.NO_REFRESH	Indicates that the consumer application does not require a login refresh for this request. This typically occurs when resuming a stream or changing a AAA token. In some instances, a provider can still deliver a refresh message, however if such a message is not explicitly asked for by the consumer, it is considered unsolicited.
LoginRequestFlags.PAUSE_ALL	Indicates that the consumer wants to pause all streams associated with the logged in user. For more information on pause and resume behavior, refer to the Enterprise Transport API C# Edition *Developers Guide*.

Table 74: **LoginRequest** Flags

8.3.1.3   Login Request Utility Method

METHOD NAME	DESCRIPTION
InitDefaultRequest	Clears a **LoginRequest** class and populates **UserName**, **Position**, **ApplicationId**, and **ApplicationName** with default values.

Table 75: **LoginRequest** Utility Method

8.3.2   Login Refresh

A *Login Refresh* message is encoded and sent by an OMM interactive provider application and responds to a Login Request message. A login refresh message indicates that the user's Login is accepted. A provider can use information from the login request to authenticate users with the Data Access Control System. After authentication, a refresh message is sent to convey that the login was accepted. If the login is rejected, a login status message should be sent as described in Section 8.3.3.

The `LoginRefresh` represents all members of a login refresh message and allows for simplified use in OMM applications that leverage RDMs. This structure follows the behavior and layout that is defined in the Enterprise Transport API C# Edition *LSEG Domain Model Usage Guide.*

8.3.2.1   Login Refresh Properties

MEMBER	DESCRIPTION
LoginAttrib	**Optional**. Contains additional login attribute information. If present, a `Flags` value of **LoginRefreshFlags.HAS_ATTRIB** should be specified. For details, refer to Section 8.3.9.1.
AuthenticationErrorCode	**Optional.** If present, a `Flags` value of **LoginRefreshFlags.HAS_AUTHN_ERROR_CODE** should be specified. `AuthenticationErrorCode` is specific to an LSEG Real-Time Distribution System Authentication environment, where **0** indicates an error-free condition. For further information, refer to the *LSEG Real-Time Distribution System Authentication User Manual.*[a]
AuthenticationErrorText	**Optional.** If present, a `Flags` value of **LoginRefreshFlags.HAS_AUTHN_ERROR_TEXT** should be specified. `AuthenticationErrorText` specifies any error text that accompanies an `AuthenticationErrorCode`. For further information, refer to the *LSEG Real-Time Distribution System Authentication User Manual.*[a]
AuthenticationExtendedResp	**Optional.** If present, a `Flags` value of **LoginRefreshFlags.HAS_AUTHN_EXTENDED_RESP** should be specified. `AuthenticationExtendedResp` contains additional, customer-defined data associated with the authentication token sent in the original request. For further information, refer to the *LSEG Real-Time Distribution System Authentication User Manual.*[a]
AuthenticationTTReissue	**Optional.** If present, a `Flags` value of **LoginRefreshFlags.HAS_AUTHN_TT_REISSUE** should be specified. Indicates when a new authentication token needs to be reissued (in UNIX Epoch time). For more information, refer to the *LSEG Real-Time Distribution System Authentication User Manual.*[a]
SupportedFeatures	**Optional.** Indicates a set of features supported by the provider of the login refresh message. If present, a `Flags` value of **LoginRefreshFlags.HAS_FEATURES** should be specified. For details, refer to Section 8.3.2.3.
Flags	**Required**. Indicate the presence of optional login refresh members. For details, see Section 8.3.2.2.
SequenceNumber	**Optional**. A user-specified, item-level sequence number which can be used by the application for sequencing messages within this stream. If present, a `Flags` value of **LoginRefreshFlags.HAS_SEQ_NUM** should be specified.

Table 76: `LoginRefresh` Properties

MEMBER	DESCRIPTION
State	**Required**. Indicates the state of the login stream. Defaults to a **StreamState** of **StreamStates.OPEN** and a **DataState** of **DataStates.OK**. For more information on **State**, refer to the Enterprise Transport API C# Edition *Developers Guide*.
UserName	**Optional**. If present, a **Flags** value of **LoginRefreshFlags.HAS_USERNAME** should be specified. Contains content appropriate to the corresponding **UserNameType** specification. If populated, this should match the **UserName** contained in the login request.
UserNameType	**Optional**. If present, a **Flags** value of **LoginRefreshFlags.HAS_USERNAME_TYPE** should be specified. If absent, a default value of **Rdm.Login.UserIdTypes.NAME** is assumed. Possible values: • **Rdm.Login.UserIdTypes.NAME == 1** • **Rdm.Login.UserIdTypes.EMAIL_ADDRESS == 2** • **Rdm.Login.UserIdTypes.TOKEN == 3** • **Rdm.Login.UserIdTypes.COOKIE==4** • **Rdm.Login.UserIdTypes.AUTHN_TOKEN==5** A type of **Rdm.Login.UserIdTypes.NAME** typically corresponds to a Data Access Control System user name and can be used to authenticate and permission a user. **Rdm.Login.UserIdTypes.TOKEN** is specified when using the AAA ('triple A') API. The user token is retrieved from the Authentication Manager application. To validate users, a provider application passes this user token to the AAA Gateway. This type of token periodically changes: when it changes, an application can send a login reissue to pass information upstream. For more information, refer to documentation specific to the AAAAPI. **Rdm.Login.UserIdTypes.AUTHN_TOKEN** is specified when using LSEG Real-Time Distribution System Authentication. The authentication token should be specified in the **UserName** property. This type of token can periodically change: when it changes, an application can send a login reissue to pass information upstream. For more information, refer to the *LSEG Real-Time Distribution System Authentication User Manual*.[a]

Table 76: LoginRefresh Properties(Continued)

a. For further details on LSEG Real-Time Distribution System Authentication, refer to the *LSEG Real-Time Distribution System Authentication User Manual*, accessible on MyAccount in the Data Access Control System product documentation set.

8.3.2.2   Login Refresh Flag Enumeration Values

FLAG ENUMERATION	DESCRIPTION
LoginRefreshFlags.CLEAR_CACHE	Indicates to clear stored payload information associated with the login stream. This might occur if some portion of data is known to be invalid.
LoginRefreshFlags.HAS_ATTRIB	Indicates the presence of **LoginAttrib**.
LoginRefreshFlags.HAS_AUTHN_ERROR_CODE	Indicates the presence of **AuthenticationErrorCode**.
LoginRefreshFlags.HAS_AUTHN_ERROR_TEXT	Indicates the presence of **AuthenticationErrorText**.
LoginRefreshFlags.HAS_AUTHN_EXTENDED_RESP	Indicates the presence of **AuthenticationExtendedResp**.
LoginRefreshFlags.HAS_AUTHN_TT_REISSUE	Indicates the presence of **AuthenticationTTReissue**.
LoginRefreshFlags.HAS_CONN_CONFIG	Indicates the presence of connection configuration information.

Table 77: LoginRefresh Flags

FLAG ENUMERATION	DESCRIPTION
LoginRefreshFlags.HAS_SEQ_NUM	Indicates the presence of **SequenceNumber**.
LoginRefreshFlags.HAS_USERNAME	Indicates the presence of **UserName**.
LoginRefreshFlags.HAS_USERNAME_TYPE	Indicates the presence of **UserNameType**. If absent, a **UserNameType** of **Rdm.Login.UserIdTypes.NAME** should be assumed.
LoginRefreshFlags.SOLICITED	• If present, this flag indicates that the login refresh is solicited (e.g., it is in response to a request). • If this flag is absent, this refresh is unsolicited.

Table 77: `LoginRefresh` Flags (Continued)

8.3.2.3   Login Support Feature Set Properties

For detailed information on posting, batch requesting, dynamic view use, and 'pause and resume,' refer to the *Transport API C# Edition Edition Developers Guide*.

PROPERTY	DESCRIPTION
Flags	**Required**. Indicates the presence of optional login refresh members. For further flag details, refer to Section 8.3.2.4.
SupportBatchCloses	**Optional**. Indicates whether the provider supports batch close functionality. • **1**: The provider supports batch closing. • **0**: The provider does not support batch reissuing. If present, a flags value of **LoginSupportFeaturesFlags.HAS_SUPPORT_BATCH_CLOSES** should be specified. If absent, a default value of **0** is assumed.
SupportBatchReissues	**Optional**. Indicates whether the provider supports batch reissue functionality. • **1**: The provider supports batch reissuing. • **0**: The provider does not support batch reissuing. If present, a flags value of **LoginSupportFeaturesFlags.HAS_SUPPORT_BATCH_REISSUES** should be specified. If absent, a default value of **0** is assumed.
SupportBatchRequests	**Optional**. Indicates whether the provider supports batch request functionality, which allows a consumer to specify multiple items, all with matching attributes, in the same request message. • **1**: The provider supports batch requesting. • **0**: The provider does not support batch requesting. If present, a **Flags** value of **LoginSupportFeaturesFlags.HAS_SUPPORT_BATCH_REQUESTS** should be specified. If absent, a default value of **0** is assumed.
SupportEnhancedSymbolList	**Optional**. Indicates whether the provider supports enhanced symbol list features: • **1**: The provider supports enhanced symbol list features. • **0**: The provider does not support enhanced symbol list features. If present, a **Flags** value of **LoginSupportFeaturesFlags.HAS_SUPPORT_ENH_SL** should be specified. If absent, a default value of **0** is assumed.

Table 78: `LoginSupportFeatures` Properties

PROPERTY	DESCRIPTION
SupportOptimizedPauseResume	**Optional**. Indicates whether the provider supports the optimized pause and resume feature. Optimized pause and resume can pause/resume individual item streams or all item streams by pausing the login stream. • **1**: The server supports optimized pause and resume. • **0**: The server does not support optimized pause and resume. If present, a **Flags** value of **LoginSupportFeaturesFlags.HAS_SUPPORT_OPT_PAUSE** should be specified. If absent, a default value of **0** is assumed.
SupportOMMPost	**Optional**. Indicates whether the provider supports OMM Posting: • **1**: The provider supports OMM Posting and the user is permissioned. • **0**: The provider supports the OMM Post feature, but the user is not permissioned. • If this element is not present, then the server does not support OMM Post feature. If present, a **Flags** value of **LoginSupportFeaturesFlags.HAS_SUPPORT_POST** should be specified. If absent, a default value of **0** is assumed.
SupportProviderDictionaryDownload	**Optional**. Indicates whether the non-interactive provider can request dictionaries from the LSEG Real-Time Advanced Distribution Hub: • **1**: The non-interactive provider can request dictionaries from the LSEG Real-Time Advanced Distribution Hub. • **0**: The non-interactive provider cannot request dictionaries from the LSEG Real-Time Advanced Distribution Hub. If present, a **Flags** value of **LoginSupportFeaturesFlags.HAS_SUPPORT_PROVIDER_DICTIONARY_DOWNLOAD** should be specified. If absent, a default value of **0** is assumed.
SupportStandby	**Optional**. Indicates whether the provider supports warm standby functionality. If supported, a provider can run as an active or a standby server, where the active will behave as usual. The standby will respond to item requests only with the message header and will forward any state changing information. When informed of an active's failure, the standby begins sending responses and takes over as active. • **1**: The provider supports a role of active or standby in a warm standby group. • **0**: The provider does not support warm standby functionality. If present, a **Flags** value of **LoginSupportFeaturesFlags.HAS_SUPPORT_STANDBY** should be specified. If absent, a default value of **0** is assumed.
SupportViewRequests	**Optional**. Indicates whether the provider supports dynamic view functionality, which allows a user to request specific response information. • **1**: The provider supports dynamic view functionality. • **0**: The provider does not support dynamic view functionality. If present, a **Flags** value of **LoginSupportFeaturesFlags.HAS_SUPPORT_VIEW** should be specified. If absent, a default value of **0** is assumed.

Table 78: `LoginSupportFeatures` **Properties**

8.3.2.4   Login Support Feature Set Flag Enumeration Values

For detailed information on batch functionality, posting, 'pause and resume,' and views, refer to the *Transport API C# Edition Developers Guide*.

FLAG ENUMERATION	MEANING
LoginSupportFeaturesFlags.HAS_SUPPORT_BATCH_CLOSES	Indicates the presence of **SupportBatchCloses**. If absent, a value of **0** is assumed.
LoginSupportFeaturesFlags.HAS_SUPPORT_BATCH_REISSUES	Indicates the presence of **SupportBatchReissues**. If absent, a value of **0** is assumed.
LoginSupportFeaturesFlags.HAS_SUPPORT_BATCH_REQUESTS	Indicates the presence of **SupportBatchRequests**. If absent, a value of **0** is assumed.
LoginSupportFeaturesFlags.HAS_SUPPORT_ENH_SL	Indicates the presence of **SupportEnhancedSymbolList**. If absent, a value of **0** is assumed.
LoginSupportFeaturesFlags.HAS_SUPPORT_OPT_PAUSE	Indicates the presence of **SupportOptimizedPauseResume**. If absent, a value of **0** is assumed.
LoginSupportFeaturesFlags.HAS_SUPPORT_POST	Indicates the presence of **SupportOMMPost**. If absent, a value of **0** is assumed.
LoginSupportFeaturesFlags.HAS_SUPPORT_PROVIDER_DICTIONARY_DOWNLOAD	Indicates the presence of **SupportProviderDictionaryDownload**. If absent, a value of **0** is assumed.
LoginSupportFeaturesFlags.HAS_SUPPORT_STANDBY	Indicates the presence of **SupportStandby**. If absent, a value of **0** is assumed.
LoginSupportFeaturesFlags.HAS_SUPPORT_VIEW	Indicates the presence of **SupportViewRequests**. If absent, a value of **0** is assumed.

Table 79: `LoginSupportFeaturesFlags`

8.3.2.5   Login Connection Config Properties

PROPERTY	DESCRIPTION
NumStandByServers	**Required**. Indicates the number of servers in the **ServerList** that the consumer can use as standby servers when using warm standby.
ServerList	**Required**. A list of servers to which the consumer may connect when using warm standby.

Table 80: `LoginConnectionConfig` Properties

8.3.2.6   Login Connection Config Methods

METHOD NAME	DESCRIPTION
Clear	Clears a **LoginConnectionConfig** object for reuse.
Copy	Performs a deep copy of a **LoginConnectionConfig** object.

Table 81: `LoginConnectionConfig` Methods

8.3.2.7   Server Info Properties

MEMBER	DESCRIPTION
Flags	**Required**. Indicates the presence of optional server information members. For details, refer to Section 8.3.2.8.
Hostname	**Required**. Indicates the server's **Hostname**.
LoadFactor	**Optional**. Indicates the load information for this server. If present, a **Flags** value of **ServerInfoFlags.HAS_LOAD_FACTOR** should be specified.
Port	**Required**. Indicates the server's port number for connections.
ServerIndex	**Required**. Provides the index value to this server.
ServerType	**Optional**. Indicates whether this server is an active or standby server. If present, a **Flags** value of **ServerInfoFlags.HAS_TYPE** should be specified, populated by **Rdm.Login.ServerTypes**.

Table 82: `ServerInfo` Members

8.3.2.8   Server Info Flag Enumeration Values

FLAG ENUMERATION	DESCRIPTION
ServerInfoFlags.HAS_LOAD_FACTOR	Indicates presence of **LoadFactor** information.
ServerInfoFlags.HAS_TYPE	Indicates presence of **ServerType**.

Table 83: `ServerInfo` Flags

8.3.2.9   Server Info Methods

METHOD NAME	DESCRIPTION
Clear	Clears a **ServerInfo** instance. Useful for instance reuse.
Copy	Performs a deep copy of a **ServerInfo** instance.

Table 84: `ServerInfo` Methods

8.3.3   Login Status

OMM provider and non-interactive provider applications use the *Login Status* message to convey state information associated with the login stream. Such state information can indicate that a login stream cannot be established or to inform a consumer of a state change associated with an open login stream.

The login status message can also reject a login request or close an existing login stream. When a status message closes a login stream, any other open streams associated with the user are also closed.

The `LoginStatus` represents all members of a login status message and allows for simplified use in OMM applications that leverage RDMs. This structure follows the behavior and layout defined in the Enterprise Transport API C# Edition *LSEG Domain Models Usage Guide*.

8.3.3.1   Login Status Properties

MEMBER	DESCRIPTION
AuthenticationErrorCode	**Optional**. If present, a `Flags` value of **LoginStatusFlags.HAS_AUTHN_ERROR_CODE** should be specified. `AuthenticationErrorCode` is specific to deployments using LSEG Real-Time Distribution System Authentication, and specifies an error code. A code of **0** indicates no error condition. For further information, refer to the *LSEG Real-Time Distribution System Authentication User Manual*.[a]
AuthenticationErrorText	**Optional**. If present, a `Flags` value of **LoginStatusFlags.HAS_AUTHN_ERROR_TEXT** should be specified. Specifies any text associated with the specified `AuthenticationErrorCode`. For further information, refer to the *LSEG Real-Time Distribution System Authentication User Manual*.[a]
Flags	**Required**. Indicates the presence of optional login status members. For details, refer to Section 8.3.3.2.
stateState	**Optional**. If present, a flags value of **LoginStatusFlags.HAS_STATE** should be specified. Indicates the state of the login stream. When rejecting a login the state should be: • `StreamState` = **StreamStates.CLOSED** or **StreamStates.CLOSED_RECOVER** • `DataState` = **DataStates.SUSPECT** • `StateCode` = **StateCodes.NOT_ENTITLED** For more information on `State`, refer to the Enterprise Transport API C# Edition *Developers Guide*.

Table 85: `LoginStatus` Members

MEMBER	DESCRIPTION
UserNameType	**Optional**. If present, a **Flags** value of **LoginStatusFlags.HAS_USERNAME_TYPE** should be specified. If absent, a default value of **Rdm.Login.UserIdTypes.NAME** is assumed. Possible values: • **Rdm.Login.UserIdTypes.NAME == 1** • **Rdm.Login.UserIdTypes.EMAIL_ADDRESS == 2** • **Rdm.Login.UserIdTypes.TOKEN == 3** • **Rdm.Login.UserIdTypes.COOKIE == 4** • **Rdm.Login.UserIdTypes.AUTHN_TOKEN == 5** A type of **Rdm.Login.UserIdTypes.NAME** typically corresponds to a Data Access Control System user name and can be used to authenticate and permission a user. **Rdm.Login.UserIdTypes.TOKEN** is specified when using the AAA ('triple A') API. The user token is retrieved from the Authentication Manager application. To validate users, a provider application passes this user token to the AAA Gateway. This type of token periodically changes: when it changes, an application can send a login reissue to pass information upstream. For more information, refer to documentation specific to the AAAAPI. **Rdm.Login.UserIdTypes.AUTHN_TOKEN** is specified when using LSEG Real-Time Distribution System Authentication. The authentication token should be specified in the **UserName** member. This type of token can periodically change: when it changes, an application can send a login reissue to pass information upstream. For more information, refer to the *LSEG Real-Time Distribution System Authentication User Manual*.[a]
UserName	**Optional**. If present, a **Flags** value of **LoginStatusFlags.HAS_USERNAME** should be specified. When populated, this should match the **UserName** in the login request.

Table 85: `LoginStatus` Members (Continued)

a. For further details on LSEG Real-Time Distribution System Authentication, refer to the *LSEG Real-Time Distribution System Authentication User Manual*, accessible on [MyAccount](#) in the Data Access Control System product documentation set.

8.3.3.2   Login Status Flag Enumeration Values

FLAG ENUMERATION	MEANING
LoginStatusFlags.CLEAR_CACHE	Indicates whether the receiver of the login status should clear any associated cache information.
LoginStatusFlags.HAS_AUTHN_ERROR_CODE	Indicates the presence of **AuthenticationErrorCode**.
LoginStatusFlags.HAS_AUTHN_ERROR_TEXT	Indicates the presence of **AuthenticationErrorText**.
LoginStatusFlags.HAS_STATE	Indicates the presence of **State**. If absent, any previously conveyed state continues to apply.
LoginStatusFlags.HAS_USERNAME	Indicates the presence of **UserName**.
LoginStatusFlags.HAS_USERNAME_TYPE	Indicates the presence of **UserNameType**. If absent a **UserNameType** of **Rdm.Login.UserIdTypes.NAME** is assumed.

Table 86: `LoginStatus` Flags

8.3.4   Login Close

A *Login Close* message is encoded and sent by OMM consumer applications. This message allows a consumer to log out of the system. Closing a login stream is equivalent to a *Close All* type of message, where all open streams are closed (i.e., all streams associated with the user). A provider can log off a user and close all of that user's streams via a login status message, see Section 8.3.3.

8.3.5   Login Consumer Connection Status

The *Login Consumer Connection Status* informs an interactive provider of its role in a *Warm Standby* group, either as an *Active* or *Standby* provider. An active provider behaves normally; however a standby provider responds to requests only with a message header (allowing a consumer application to confirm the availability of requested data across active and standby servers), and forwards any state-related messages (i.e., unsolicited refresh messages, status messages). A standby provider aggregates changes to item streams whenever possible. If a provider changes from Standby to Active via this message, all aggregated update messages are passed along. If aggregation is not possible, a full, unsolicited refresh message is passed along.

The consumer application is responsible for ensuring that items are available and equivalent across all providers in a warm standby group. This includes managing state and availability differences as well as item group differences.

The `LoginConsumerConnectionStatus` relies on the `GenericMsg` and represents all members necessary for applications that leverage RDMs. This structure follows the behavior and layout that is defined in the Enterprise Transport API C# Edition *LSEG Domain Model Usage Guide*.

8.3.5.1   Login Consumer Connection Status Members

MEMBER	DESCRIPTION
Flags	**Required**. Indicate the presence of optional login consumer connection status members. For details, refer to Section 8.3.5.2.
WarmStandbyInfo	**Optional**. Includes `LoginWarmStandbyInfo` to convey the state of the upstream provider. For details, refer to Section 8.3.5.3. If present, a flags value of **LoginConsumerConnectionStatusFlags.HAS_WARM_STANDBY_INFO** should be specified.

Table 87: `LoginConsumerConnectionStatus` Structure Members

8.3.5.2   Login Consumer Connection Status Flag Enumeration Value

FLAG ENUMERATION	DESCRIPTION
LoginConsumerConnectionStatusFlags.HAS_WARM_STANDBY_INFO	Indicates presence of `WarmStandbyInfo`.

Table 88: `LoginConsumerConnectionStatus` Flags

8.3.5.3   Login Warm Standby Info Properties

MEMBER	DESCRIPTION
Action	**Required**. Indicates how a cache of Warm Standby content should apply this information. For information on `MapEntry` actions, refer to the Enterprise Transport API C# Edition *Developers Guide*.
WarmStandbyMode	**Required**.

Table 89: `LoginWarmStandbyInfo` Members

8.3.5.4   Login Warm Standby Info Methods

METHOD NAME	DESCRIPTION
Clear	Clears a `LoginWarmStandbyInfo` instance for reuse.
Copy	Performs a deep copy of a `LoginWarmStandbyInfo` instance.

Table 90: Methods

8.3.6   Login Round Trip Time Message Use

Interactive Provider applications use Login Round Trip Time messages to measure the full roundtrip latency time between the provider and consumer. You enable Round Trip Time by setting the `LoginAttribFlags.HAS_CONSUMER_SUPPORT_RTT` flag on the initial Login RDM request message. When `LoginAttribFlags.HAS_CONSUMER_SUPPORT_RTT` is set on the consumer, the Reactor attempts to reflect the Round Trip Time message whenever the provider sends a Round Trip Time message, and the login callback will contain the incoming Round Trip Time message for informational purposes. The Consumer application does not need to take further action to handle Round Trip Time messages.

For more specific usage information about this message type, refer to the Enterprise Transport API C# Edition *RDM Usage Guide*.

8.3.6.1   Login Round Trip Time Properties

STRUCTURE MEMBER	DESCRIPTION
Flags	**Required**. Indicates the presence of optional Round Trip Time members. For details, refer to Section 8.3.6.2.
RTLatency	Specifies the previous Round Trip Latency value (in microseconds) calculated by the provider.
TcpRetrans	Indicates the total number of TCP retransmissions.
Ticks	**Required**. Specifies the tick count sent by the provider. After receiving `Ticks`, the consumer must reflect this value back to the provider using this element.

Table 91: Login Round Trip Time Members

8.3.6.2   Login Round Trip Time Flag Enumeration Values

FLAG ENUMERATION	DESCRIPTION
LoginRTTFlags.HAS_TCP_RETRANS	Indicates the presence of the **TcpRetrans** member.
LoginRTTFlags.HAS_LATENCY	Indicates the presence of the **RTLatency** member.

Table 92: Login Round Trip Time Flag Enumeration Values

8.3.6.3   Login Round Trip Time Utility Methods

The Enterprise Transport API provides the following utility method for use with Login Round Trip Time Messages.

METHOD NAME	DESCRIPTION
Clear	Clears an **LoginRTT** class for reuse.
Copy	Performs a deep copy of an **LoginRTT** class.

Table 93: Login Round Trip Time Utility Methods

8.3.7   Login Post Message Use

OMM consumer applications can encode and send data for any item via Post messages on the item's login stream. This is known as *off-stream posting* because items are posted without using that item's dedicated stream. Posting an item on its own dedicated stream is referred to as *on-stream posting*.

When an application is off-stream posting, **MsgKey** information is required on the **PostMsg**. For more details on posting, refer to the Enterprise Transport API C# Edition *Developers Guide*.

8.3.8   Login Ack Message Use

OMM provider applications encode and send Ack messages to acknowledge the receipt of Post messages. An Ack message is used whenever a consumer posts and asks for acknowledgments. For more details on posting, see the Enterprise Transport API C# Edition *Developers Guide*.

8.3.9   Login Attributes

On a Login Request or Login Refresh message, the `LoginAttrib` can send additional authentication information and user preferences between components.

8.3.9.1   Login Attrib Properties

The following table lists the elements available on a `LoginAttrib`.

PROPERTY	DESCRIPTION
AllowSuspectData	Indicates how the consumer application wants to handle suspect data. • **1**: Indicates that the consumer application allows for suspect `StreamState` information. If absent, a default value of **1** is assumed. • **0**: Indicates that the consumer application wants suspect data to cause the stream to close with a **StreamStates.CLOSED_RECOVER** state. If present, a `Flags` value of **LoginAttribFlags.HAS_ALLOW_SUSPECT_DATA** should be specified.
ApplicationId	Indicates the application ID. • If populated in a login request, `ApplicationId` should be set to the DACS `ApplicationId`. If the server authenticates with DACS, the consumer application may be required to pass in a valid application id. If initializing `LoginRequest` using `InitDefaultRequest`, an `ApplicationId` of **256** will be used. • If populated in a login refresh, `ApplicationId` should match the `ApplicationId` used in the login request. If present, a `Flags` value of **LoginAttribFlags.HAS_APPLICATION_ID** should be specified.
ApplicationName	Indicates the application name. • If populated in a login request, the `ApplicationName` identifies the OMM consumer or OMM non-interactive provider. If initializing `LoginRequest` using `InitDefaultRequest`, the `ApplicationName` is set to **upa**. • If populated in a login refresh, the `ApplicationName` identifies the OMM provider. If present, a `Flags` value of **LoginAttribFlags.HAS_APPLICATION_NAME** should be specified.
Flags	**Required**. Indicates the presence of optional login attribute members. For details, refer to Section 8.3.9.3.
Position	Indicates the DACS position. • When populated in a login request, `Position` should match the `Position` contained in the login request and the DACS `Position` (if using DACS). If the server is authenticating with DACS, the consumer application might be required to pass in a valid position. If initializing `LoginRequest` using `InitDefaultRequest`, the IP address of the system on which the application runs will be used. • When populated in a login refresh, this should match the `Position` contained in the login request If present, a `Flags` value of **LoginAttribFlags.HAS_POSITION** should be specified.

Table 94: `LoginAttrib` Properties

PROPERTY	DESCRIPTION
ProvidePermissionExpressions	Indicates whether the consumer wants permission expression information. Permission expressions allow for items to be proxy-permissioned by a consumer via content-based entitlements. • **1**: Requests that permission expression information be sent with responses. If absent, a default value of **1** is assumed. • **0**: Indicates that the consumer does not want permission expression information. If present, a **Flags** value of **LoginAttribFlags.HAS_PROVIDE_PERM_EXPR** should be specified.
ProvidePermissionProfile	Indicates whether the consumer desires the permission profile. An application can use the permission profile to perform proxy permissioning. If present, a **Flags** value of **LoginAttribFlags.HAS_PROVIDE_PERM_PROFILE** should be specified. • **1**: Indicates that the consumer wants the permission profile. If absent, a default value of **1** is assumed. • **0**: Indicates the consumer does not want the permission profile.
SingleOpen	Indicates which application the consumer wants to drive stream recovery. If present, a **Flags** value of **LoginAttribFlags.HAS_SINGLE_OPEN** should be specified. • **1**: Indicates that the consumer application wants the provider to drive stream recovery. If absent, a default value of **1** is assumed. • **0**: Indicates that the consumer application drives stream recovery.
SupportProviderDictionaryDownload	Indicates whether the interactive provider can request dictionaries from the LSEG Real-Time Advanced Distribution Hub: • **1**: The interactive provider can request dictionaries from the LSEG Real-Time Advanced Distribution Hub. • **0**: The interactive provider cannot request dictionaries from the LSEG Real-Time Advanced Distribution Hub. If present, a **Flags** value of **LoginAttribFlags.HAS_PROVIDER_SUPPORT_DICTIONARY_DOWNLOAD** should be specified. If absent, a default value of **0** is assumed.
SupportRTTMonitoring	Indicates whether the consumer supports the Round Trip Time monitoring feature in the login stream. If present, for both provider and consumer, a **Flags** value of **LoginAttribFlags.HAS_CONSUMER_SUPPORT_RTT** should be specified. A value of **2** indicates support for the Round Trip Time monitoring feature.

Table 94: LoginAttrib Properties (Continued)

8.3.9.2   Login Attrib Methods

METHOD NAME	DESCRIPTION
Clear	Clears a **LoginAttrib** instance for reuse.
Copy	Performs a deep copy of a **LoginAttrib** instance.

Table 95: `LoginAttrib` Methods

8.3.9.3   Login Attrib Flag Enumeration Values

FLAG ENUMERATION	MEANING
HAS_ALLOW_SUSPECT_DATA	Indicates the presence of **AllowSuspectData**. If absent, a value of **1** is assumed.
HAS_APPLICATION_ID	Indicates the presence of **ApplicationId**.
HAS_APPLICATION_NAME	Indicates the presence of **ApplicationName**.
HAS_POSITION	Indicates the presence of **Position**.
HAS_PROVIDE_PERM_EXPR	Indicates the presence of **ProvidePermissionExpressions**. If absent, a value of **1** is assumed.
HAS_PROVIDE_PERM_PROFILE	Indicates the presence of **ProvidePermissionProfile**. If absent, a value of **1** is assumed.
HAS_PROVIDER_SUPPORT_DICTIONARY_DOWNLOAD	Indicates the presence of **SupportProviderDictionaryDownload**.
HAS_SINGLE_OPEN	Indicates the presence of **SingleOpen**. If absent, a value of **1** is assumed.

Table 96: `LoginAttribFlags`

8.3.10   Login Message

LoginMsg is the class that encapsulates all Login message types. It is provided for use with general login-specific functionality. The following table summarizes different login messages.

CLASS	DESCRIPTION
LoginClose	RDM Login Close.
LoginConsumerConnectionStatus	RDM Login Consumer Connection Status.
LoginRefresh	RDM Login Refresh
LoginRequest	RDM Login Request.
LoginStatus	RDM Login Status.
LoginRTT	RDM Login Round Trip Time

Table 97: `LoginMsg` Classes

（空白行）

8.3.11   Login Message Utility Method

FUNCTION NAME	DESCRIPTION
Copy	Performs a deep copy of a **LoginMsg** object.

Table 98: `LoginMsg` Utility Method

8.3.12   Login Encoding and Decoding

8.3.12.1   Login Encoding and Decoding Methods

METHOD NAME	DESCRIPTION
Decode	Decodes a Login message.The decoded message may refer to encoded data from the original **Message**. If you want to store the message, use the appropriate copy method for the decoded message to create a full copy.
	Each login subinterface overrides this method to decode specific login message.

Table 99: Login Encoding and Decoding Methods

8.3.12.2   Encoding a Login Request

```
LoginRequest reqRDMMsg = new LoginRequest();
reqRDMMsg.Clear();
reqRDMMsg.StreamId = streamId;
reqRDMMsg.UserName.Data("User");
reqRDMMsg.HasUserNameType = true;
reqRDMMsg.UserNameType = UserIdTypes.NAME;

reqRDMMsg.HasAttrib = true;
reqRDMMsg.LoginAttrib.HasApplicationId = true;
reqRDMMsg.LoginAttrib.ApplicationId.Data("AppId");
reqRDMMsg.LoginAttrib.HasApplicationName = true;
reqRDMMsg.LoginAttrib.ApplicationName.Data("AppName");
reqRDMMsg.HasPassword = true;
reqRDMMsg.Password.Data(password);
reqRDMMsg.LoginAttrib.HasPosition = true;
reqRDMMsg.LoginAttrib.Position.Data(position);

Buffer membuf = new();
membuf.Data(new ByteBuffer(1024));

EncodeIterator encIter = new EncodeIterator();
encIter.Clear();
encIter.SetBufferAndRWFVersion(membuf, Codec.Codec.MajorVersion(), Codec.Codec.MinorVersion());

reqRDMMsg.Encode(encIter);
```

Code Example 12: Login Request Encoding Example

8.3.12.3   Decoding a Login Request

```csharp
DecodeIterator decIter = new DecodeIterator();
LoginRequest loginRequest = new LoginRequest();
decIter.Clear();

// membuf contains encoded message
decIter.SetBufferAndRWFVersion(membuf, Codec.Codec.MajorVersion(), Codec.Codec.MinorVersion());
msg.Decode(decIter);
CodecReturnCode ret = loginRequest.Decode(decIter, msg);
```

Code Example 13: Login Request Decoding Example

8.3.12.4   Encoding a Login Refresh

```csharp
EncodeIterator encodeIter = new EncodeIterator();
LoginRefresh loginRefresh = new LoginRefresh();

/* Clear the Login Refresh object. */
loginRefresh.Clear();

/* Set stream id. */
loginRefresh.StreamId = streamId;

/* Set flags indicating presence of optional members. */
loginRefresh.HasAttrib = true;
loginRefresh.HasUserName = true;

/* Set UserName. */
loginRefresh.UserName.Data("username");

/* Set ApplicationName */
loginRefresh.LoginAttrib.HasApplicationName = true;
loginRefresh.LoginAttrib.ApplicationName.Data("eta");

/* Set ApplicationId */
loginRefresh.LoginAttrib.HasApplicationId = true;
loginRefresh.LoginAttrib.ApplicationId.Data("256");

/* Set Position */
loginRefresh.LoginAttrib.HasPosition = true;
loginRefresh.LoginAttrib.Position.Data("127.0.0.1/net");

/* Clear the encode iterator, set its RWF Version, and set it to a buffer for encoding into. */
encodeIter.Clear();
ret = encodeIter.SetBufferAndRWFVersion(msgBuf,channelMajorVersion, channelMinorVersion);

/* Encode the message. */
ret = loginRefresh.Encode(encodeIter);
```

Code Example 14: Login Refresh Encoding Example

8.3.12.5   Decoding a Login Refresh

```
//msgBuf is a buffer that contains the encoded message
Msg msg = new Msg();
m_DecIter.Clear();
m_DecIter.SetBufferAndRWFVersion(msgBuf, chnl.MajorVersion, chnl.MinorVersion);
msg.Decode(m_DecIter);
if (msg.MsgClass == MsgClasses.REQUEST)
{
m_LoginRequest.Clear();
if (m_LoginRequest.Decode(dIter, msg) != CodecReturnCode.SUCCESS)
{
Console.WriteLine("Failed decoding Login Request message");
}
}
```

Code Example 15: Login Refresh Decoding Example

8.4   RDM Source Directory Domain

The Source Directory domain model conveys information about:

- All available services and their capabilities, their supported domain types, services' states, quality of service, and item group information (associated with any particular service). Each service is associated with a unique **ServiceId**.

- Item group status, allowing a single message to change the state of all associated items. Thus, using the Source Directory domain an application can send a mass update for multiple items instead of sending a status message for each individual item. The consumer is responsible for applying any changes to its open items. For details, refer to Section 8.4.10.

- Source Mirroring between an LSEG Real-Time Advanced Distribution Hub and OMM interactive provider applications. The Source Directory exchanges this information via a specifically-formatted generic message as described in Section 8.4.6.

8.4.1   Directory Request

An OMM consumer application encodes and sends *Directory Request* messages to request information from an OMM provider about available services. A consumer may request information about all services by omitting the **ServiceId** member, or request information about a specific service by setting it to the ID of the desired service.

The **DirectoryRequest** represents all members of a directory request message and is easily used in OMM applications that leverage RDMs. This structure follows the behavior and layout that is defined in the Enterprise Transport API C# Edition *LSEG Domain Model Usage Guide*.

8.4.1.1   Directory Request Properties

MEMBER	DESCRIPTION
Filter	**Required**. Indicates the service information in which the consumer is interested. The available flags are: • **Rdm.Directory.ServiceFilterFlags.INFOFILTER == 0x01** • **Rdm.Directory.ServiceFilterFlags.STATEFILTER == 0x02** • **Rdm.Directory.ServiceFilterFlags.GROUPFILTER == 0x04** • **Rdm.Directory.ServiceFilterFlags.LOADFILTER == 0x08** • **Rdm.Directory.ServiceFilterFlags.DATAFILTER == 0x10** • **Rdm.Directory.ServiceFilterFlags.LINKFILTER == 0x20** In most cases, you should set the **Rdm.Directory.ServiceFilterFlags.INFOFILTER**, **Rdm.Directory.ServiceFilterFlags.STATEFILTER**, and **Rdm.Directory.ServiceFilterFlags.GROUPFILTER**.
Flags	**Required**. Indicates the presence of optional directory request members. For details, refer to Section 8.4.1.2.
ServiceId	**Optional**. • If not present, this indicates the consumer wants information about all available services. • If present, this indicates the ID of the service about which the consumer wants information. Additionally, a **Flags** value of **DirectoryRequestFlags.HAS_SERVICE_ID** should be specified.

Table 100: `DirectoryRequest` Members

8.4.1.2   Directory Request Flag Enumeration Values

FLAG ENUMERATION	DESCRIPTION
DirectoryRequestFlags.HAS_SERVICE_ID	Indicates the presence of **ServiceId**.
DirectoryRequestFlags.STREAMING	Indicates that the consumer wants to receive updates about directory information after the initial refresh.

Table 101: `DirectoryRequest` Flags

8.4.1.3   Directory Request Methods

METHOD NAME	DESCRIPTION
Clear	Clears a **DirectoryRequest** class. Useful for class reuse.
Copy	Performs a deep copy of a **DirectoryRequest** class.

Table 102: `DirectoryRequest` Utility Methods

8.4.2   Directory Refresh

A *Directory Refresh* message is encoded and sent by OMM provider and non-interactive provider applications. This message can provide information about the services supported by the provider application.

The `DirectoryRefresh` represents all members of a directory refresh message and is easily used in OMM applications that leverage RDMs. This structure follows the behavior and layout that is defined in the Enterprise Transport API C# Edition *LSEG Domain Model Usage Guide*.

8.4.2.1   Directory Refresh Properties

MEMBER	DESCRIPTION
Filter	**Required**. Indicates the information being provided about supported services. This should match the `Filter` of the consumer's `DirectoryRequest`. The available flags are: • **Rdm.Directory.ServiceFilterFlags.INFOFILTER == 0x01** • **Rdm.Directory.ServiceFilterFlags.STATEFILTER == 0x02** • **Rdm.Directory.ServiceFilterFlags.GROUPFILTER == 0x04** • **Rdm.Directory.ServiceFilterFlags.LOADFILTER == 0x08** • **Rdm.Directory.ServiceFilterFlags.DATAFILTER == 0x10** • **Rdm.Directory.ServiceFilterFlags.LINKFILTER == 0x20**
Flags	**Required**. Indicates the presence of optional directory refresh members. Refer to Section 8.4.2.2.
SequenceNumber	**Optional**. If present, a `Flags` value of **DirectoryRefreshFlags.HAS_SEQ_NUM** should be specified. `SequenceNumber` is a user-specified, item-level sequence number that the application can use to sequence messages in the stream.
ServiceId	**Optional**. If present, a `Flags` value of **DirectoryRefreshFlags.HAS_SERVICE_ID** should be specified, which should match the `ServiceId` of the consumer's `DirectoryRequest`.
ServiceList	**Optional**. Contains a of information about available services.
State	**Required**. Indicates stream and data state information. For further details on `State`, refer to the Enterprise Transport API C# Edition *Developers Guide*.

Table 103: `DirectoryRefresh` Members

8.4.2.2   Directory Refresh Flag Enumeration Values

FLAG ENUMERATION	DESCRIPTION
DirectoryRefreshFlags.CLEAR_CACHE	Indicates that any stored payload information associated with the directory stream should be cleared. This might happen if some portion of data is known to be invalid.
DirectoryRefreshFlags.HAS_SEQ_NUM	Indicates the presence of `SequenceNumber`.
DirectoryRefreshFlags.HAS_SERVICE_ID	Indicates the presence of `ServiceId`.
DirectoryRefreshFlags.SOLICITED	If present, this flag indicates that the directory refresh is solicited (i.e., it is in response to a request). The absence of this flag indicates that the refresh is unsolicited.

Table 104: `DirectoryRefresh` Flags

8.4.3   Directory Update

A *Directory Update* message is encoded and sent by OMM provider and non-interactive provider applications. This message can provide information about new or removed services, or changes to existing services.

The `DirectoryUpdate` represents all members of a directory update message and allows for simplified use in OMM applications that leverage RDMs. This structure follows the behavior and layout that is defined in the Enterprise Transport API C# Edition *LSEG Domain Model Usage Guide*.

8.4.3.1   Directory Update Properties

MEMBER	DESCRIPTION
Filter	**Optional**. Indicates what information is provided about supported services. This should match the **Filter** of the consumer's **DirectoryRequest**. If present, a **Flags** value of **DirectoryUpdateFlags.HAS_FILTER** should be specified. Available flags are: • **Rdm.Directory.ServiceFilterFlags.INFOFILTER == 0x01** • **Rdm.Directory.ServiceFilterFlags.STATEFILTER == 0x02** • **Rdm.Directory.ServiceFilterFlags.GROUPFILTER == 0x04** • **Rdm.Directory.ServiceFilterFlags.LOADFILTER == 0x08** • **Rdm.Directory.ServiceFilterFlags.DATAFILTER == 0x10** • **Rdm.Directory.ServiceFilterFlags.LINKFILTER == 0x20**
Flags	**Required**. Indicates the presence of optional directory update members. For details refer to Section 8.4.3.2.
SequenceNumber	**Optional**. A user-specified, item-level sequence number which the application can use to sequence messages in this stream. If present, a **Flags** value of **DirectoryUpdateFlags.HAS_SEQ_NUM** should be specified.
ServiceId	**Optional**. This member's value must match the **ServiceId** of the consumer's **DirectoryRequest**. If present, a **Flags** value of **DirectoryUpdateFlags.HAS_SERVICE_ID** should be specified.
ServiceList	**Optional**. Contains a list of information about available services.

Table 105: `DirectoryUpdate` Members

8.4.3.2   Directory Update Flag Enumeration Values

FLAG ENUMERATION	DESCRIPTION
DirectoryUpdateFlags.HAS_FILTER	Indicates the presence of **Filter**.
DirectoryUpdateFlags.HAS_SEQ_NUM	Indicates the presence of **SequenceNumber**.
DirectoryUpdateFlags.HAS_SERVICE_ID	Indicates the presence of **ServiceId**.

Table 106: `DirectoryUpdate` Flags

8.4.4   Directory Status

OMM providers and non-interactive providers use the *Directory Status* message to convey state information associated with the directory stream. Such state information can indicate that a directory stream cannot be established or to inform a consumer of a state change associated with an open directory stream. An application can also use the Directory Status message to close an existing directory stream.

The `DirectoryStatus` represents all members of a directory status message and allows for simplified use in OMM applications that leverage RDMs. This structure follows the behavior and layout that is defined in the Enterprise Transport API C# Edition *LSEG Domain Model Usage Guide*.

8.4.4.1   Directory Status Properties

MEMBER	DESCRIPTION
Filter	**Optional**. If present, a `Flags` value of **DirectoryStatusFlags.HAS_FILTER** should be specified. Indicates what information is being provided about supported services. This should match the `Filter` of the consumer's `DirectoryRequest`. The available flags are: • **Rdm.Directory.ServiceFilterFlags.INFOFILTER == 0x01** • **Rdm.Directory.ServiceFilterFlags.STATEFILTER == 0x02** • **Rdm.Directory.ServiceFilterFlags.GROUPFILTER == 0x04** • **Rdm.Directory.ServiceFilterFlags.LOADFILTER == 0x08** • **Rdm.Directory.ServiceFilterFlags.DATAFILTER == 0x10** • **Rdm.Directory.ServiceFilterFlags.LINKFILTER == 0x20**
Flags	**Required**. Indicates the presence of optional directory status members. For details, refer to Section 8.4.4.2.
ServiceId	**Optional**. If present, a `Flags` value of **DirectoryStatusFlags.HAS_SERVICE_ID** should be specified. This member should match the `ServiceId` of the consumer's `DirectoryRequest`.
State	**Optional**. Indicates the state of the directory stream. If present, a `Flags` value of **DirectoryStatusFlags.HAS_STATE** should be specified. For more information on `State`, refer to the Enterprise Transport API C# Edition *Developers Guide*.

Table 107: `DirectoryStatus` Members

8.4.4.2   Directory Status Flag Enumeration Values

FLAG ENUMERATION	DESCRIPTION
DirectoryStatusFlags.CLEAR_CACHE	Indicates that any stored payload data associated with the directory stream should be cleared. This might happen if some portion of data is known to be invalid.
DirectoryStatusFlags.HAS_FILTER	Indicates the presence of `Filter`.
DirectoryStatusFlags.HAS_SERVICE_ID	Indicates the presence of `ServiceId`.
DirectoryStatusFlags.HAS_STATE	Indicates the presence of `State`. If not present, any previously conveyed state should continue to apply.

Table 108: `DirectoryStatus` Flags

8.4.4.3   Directory Status Methods

METHOD NAME	DESCRIPTION
Clear	Clears a **DirectoryStatus** instance. Useful for instance reuse.
Copy	Performs a deep copy of a **DirectoryStatus** instance.

Table 109: `DirectoryStatus` Utility Methods

8.4.5   Directory Close

A *Directory Close* message is encoded and sent by OMM consumer applications. This message allows a consumer to close an open directory stream. A provider can close the directory stream via a Directory Status message; for details refer to Section 8.4.4.

8.4.6   Directory Consumer Status

The *Directory Consumer Status* is sent by OMM consumer applications to inform a service of how the consumer is used for *Source Mirroring*. This message is primarily informational.

The **DirectoryConsumerStatus** relies on the **GenericMsg** and represents all members necessary for applications that leverage RDMs. This structure follows the behavior and layout that is defined in the Enterprise Transport API C# Edition *LSEG Domain Model Usage Guide*.

8.4.6.1   Directory Consumer Status Properties

MEMBER	DESCRIPTION
ConsumerServiceStatusList	**Optional**. Contains a list of **ConsumerStatusService** objects.

Table 110: `DirectoryConsumerStatus` Members

8.4.6.2   Directory Consumer Status Service Members

MEMBER	DESCRIPTION
Action	**Required**. Indicates how a cache of Source Mirroring content should apply this information. For information on **MapEntry** actions, refer to the Enterprise Transport API C# Edition *Developers Guide*.
ServiceId	**Required**. Indicates the service associated with this status.
SourceMirroringMode	**Required**. Indicates how the consumer is using the service. Available enumerations are: • **Rdm.Directory.SourceMirroringMode.ACTIVE_NO_STANDBY == 0**, • **Rdm.Directory.SourceMirroringMode.ACTIVE_WITH_STANDBY == 1**, • **Rdm.Directory.SourceMirroringMode.STANDBY == 2**

Table 111: `ConsumerStatusService` Members

8.4.7   Directory Service

A **Service** class conveys information about a service. A list of **Services** forms the **serviceServiceList** member of the **DirectoryRefresh** and **DirectoryUpdate** messages.

The members of a **Service** represent the different filters used to categorize service information.

8.4.7.1   Service Members

MEMBER	DESCRIPTION
Action	**Required**. Indicates how a cache of the service should apply this information. For information on **MapEntry** actions, refer to the Enterprise Transport API C# Edition *Developers Guide*.
Data	**Optional**. Contains data that applies to the items requested from the service and represents the Source Directory Data Filter. If present, a **Flags** value of **ServiceFlags.HAS_DATA** should be specified.
Flags	**Required**. Indicates the presence of optional service members. For details, refer to Section 8.4.7.2.
GroupStateList	**Optional**. Contains a list of elements indicating changes to item groups and represents the Source Directory Group filter.
Info	**Optional**. Contains information related to the Source Directory Info Filter. If present, a **Flags** value of **ServiceFlags.HAS_INFO** should be specified.
Link	**Optional**. Contains information about upstream sources that provide data to this service and represents the Source Directory Link Filter. If present, a **Flags** value of **ServiceFlags.HAS_LINK** should be specified.
Load	**Optional**. Contains information about the service's operating workload and represents the Source Directory Load Filter. If present, a **Flags** value of **ServiceFlags.HAS_LOAD** should be specified.
ServiceId	**Required**. Indicates the service associated with this **Service**.
State	**Optional**. Contains information related to the Source Directory State Filter. If present, a **Flags** value of **ServiceFlags.HAS_STATE** should be specified.

Table 112: `Service Members`

8.4.7.2   Service Flag Enumeration Values

FLAG ENUMERATION	DESCRIPTION
ServiceFlags.HAS_DATA	Indicates the presence of **Data**.
ServiceFlags.HAS_INFO	Indicates the presence of **Info**.
ServiceFlags.HAS_LINK	Indicates the presence of **Link**.
ServiceFlags.HAS_LOAD	Indicates the presence of **Load**.
ServiceFlags.HAS_STATE	Indicates the presence of **State**.

Table 113: `Service Flags`

8.4.7.3   Service Methods

METHOD NAME	DESCRIPTION
Clear	Clears a **Service** instance. Useful for instance reuse.
Copy	Performs a deep copy of a **Service** instance.

Table 114: Service Utility Method

8.4.8   Directory Service Info Filter

A **ServiceInfo** class conveys information that identifies the service and the content it provides. The **ServiceInfo** class represents the Source Directory Info filter. More information about the Info filter is available in the Enterprise Transport API C# Edition *LSEG Domain Model Usage Guide*.

8.4.8.1   Service Info Properties

MEMBER	DESCRIPTION
AcceptingConsumerStatus	**Optional**. Indicates whether this service supports accepting **DirectoryConsumerStatus** messages for Source Mirroring. Available values are: • **1**: The service will accept Consumer Status messages. If not present, a value of **1** is assumed. • **0**: The service will not accept Consumer Status messages. If present, a **Flags** value of **ServiceInfoFlags.HAS_ACCEPTING_CONS_STATUS** should be specified.
Action	**Required**. Indicates how a service info cache should apply this information. For information on **FilterEntryActions**, refer to the Enterprise Transport API C# Edition *Developers Guide*.
CapabilitiesList	**Required**. Contains a list of capabilities that the service supports. Populated by domain types.
DictionariesProvidedList	**Optional**. Contains a list of elements that identify dictionaries that can be requested from this service. If present, a **Flags** value of **ServiceInfoFlags.HAS_DICTS_PROVIDED** should be specified.
DictionariesUsedList	**Optional**. Contains a list of elements that identify dictionaries used to decode data from this service. If present, a **Flags** value of **ServiceInfoFlags.HAS_DICTS_USED** should be specified.
Flags	**Required**. Indicates the presence of optional service info members. For details, refer to Section 8.4.8.2.
IsSource	**Optional**. Indicates whether the service is provided directly by a source or represents a group of sources. • **1**: The service is provided directly by a source • **0**: The service represents a group of sources. If absent, a value of **0** is assumed. If present, a **Flags** value of **ServiceInfoFlags.HAS_IS_SOURCE** should be specified.
ItemList	**Optional**. Specifies a name that can be requested on the **DomainType.SYMBOL_LIST** domain to get a list of all items available from this service. If present, a **Flags** value of **ServiceInfoFlags.HAS_ITEM_LIST** should be specified.

Table 115: ServiceInfo Members

MEMBER	DESCRIPTION
QosList	**Optional**. Contains a list of elements that identify the available Qualities of Service. If present, a **Flags** value of **ServiceInfoFlags.HAS_QOS** should be specified.
ServiceName	**Required**. Indicates the name of the service.
SupportsOutOfBandSnapshots	**Optional**. Indicates whether this service supports making snapshot requests even when the **OpenLimit** is reached. Available values are: • **1**: Snapshot requests are allowed. If not present, a value of **1** is assumed. • **0**: Snapshot requests are not allowed. If present, a **Flags** value of **ServiceInfoFlags.HAS_SUPPORT_OOB_SNAPSHOTS** should be specified.
SupportsQosRange	**Optional**. Indicates whether this service supports specifying a range of Qualities of Service when requesting an item. For further information, refer to the **Qos** and **WorstQos** members of the **RequestMsg** in the Enterprise Transport API C# Edition *Developers Guide*. Available values are: • **1**: Quality of Service Range requests are supported. • **0**: Quality of Service Range requests are not supported. If not present, a value of **0** is assumed. If present, a **Flags** value of **ServiceInfoFlags.HAS_SUPPORT_QOS_RANGE** should be specified.
Vendor	**Optional**. Identifies the vendor of the data. If present, a **Flags** value of **ServiceInfoFlags.HAS_VENDOR** should be specified.

Table 115: `ServiceInfo` Members (Continued)

8.4.8.2   Service Info Flag Enumeration Values

FLAG ENUMERATION	DESCRIPTION
ServiceInfoFlags.HAS_ACCEPTING_CONS_STATUS	Indicates the presence of **AcceptingConsumerStatus**.
ServiceInfoFlags.HAS_DICTS_PROVIDED	Indicates the presence of **DictionariesProvidedList**.
ServiceInfoFlags.HAS_DICTS_USED	Indicates the presence of **DictionariesUsedList**.
ServiceInfoFlags.HAS_IS_SOURCE	Indicates the presence of **IsSource**.
ServiceInfoFlags.HAS_ITEM_LIST	Indicates the presence of **ItemList**.
ServiceInfoFlags.HAS_QOS	Indicates the presence of **QosList**.
ServiceInfoFlags.HAS_SUPPORT_OOB_SNAPSHOTS	Indicates the presence of **SupportsOutOfBandSnapshots**.
ServiceInfoFlags.HAS_SUPPORT_QOS_RANGE	Indicates the presence of **SupportsQosRange**.
ServiceInfoFlags.HAS_VENDOR	Indicates the presence of **Vendor**.

Table 116: `ServiceInfo` Flags

8.4.8.3   Service Info Methods

METHOD NAME	DESCRIPTION
Clear	Clears a `ServiceInfo` instance. Useful for instance reuse.
Copy	Performs a deep copy of a `ServiceInfo` instance.

Table 117: `ServiceInfo` **Utility Methods**

8.4.9   Directory Service State Filter

A `ServiceState` class conveys information about service's current state. It represents the Source Directory State filter. For more information about the State filter, refer to the Enterprise Transport API C# Edition *LSEG Domain Model Usage Guide*.

8.4.9.1   Service State Members

MEMBER	DESCRIPTION
AcceptingRequests	Indicates whether the immediate provider (to which the consumer is directly connected) can handle the request. Available values are: • **1**: The service will accept new requests. • **0**: The service does not currently accept new requests. If present, `Flags` value of **ServiceStateFlags.HAS_ACCEPTING_REQS** should be specified.
Action	**Required**. Indicates how a cache of the service state should apply this information. For details on `FilterEntryActions`, refer to the Enterprise Transport API C# Edition *Developers Guide*.
Flags	**Required**. Indicates the presence of optional service state members. For details refer to Section 8.4.9.2.
ServiceState	**Required**. Indicates whether the original provider of the data can respond to new requests. Requests can still be made if so indicated by `AcceptingRequests`. Available values are: • **1**: The original provider of the data is available. • **0**: The original provider of the data is not currently available.
Status	This status should be applied to all open items associated with this service. If present, `Flags` value of **ServiceStateFlags.HAS_STATUS** should be specified.

Table 118: `ServiceState` **Members**

8.4.9.2   Service State Flag Enumeration Values

FLAG ENUMERATION	DESCRIPTION
ServiceStateFlags.HAS_ACCEPTING_REQS	Indicates the presence of `AcceptingRequests`.
ServiceStateFlags.HAS_STATUS	Indicates the presence of `Status`.

Table 119: `ServiceState` **Flags**

8.4.9.3   Service State Methods

METHOD NAME	DESCRIPTION
Clear	Clears a `ServiceState` instance. Useful for instance reuse.
Copy	Performs a deep copy of a `ServiceState` instance.

Table 120: `ServiceState` Utility Methods

8.4.10   Directory Service Group Filter

A `ServiceGroup` class is used to convey status and name changes for an item group. It represents the Source Directory Group filter. For further details about the Group filter, refer to the Enterprise Transport API C# Edition *LSEG Domain Model Usage Guide*.

8.4.10.1   Service Group Properties

MEMBER	DESCRIPTION
Action	**Required**. Indicates how a cache of the service group should apply this information. For further details on `FilterEntryActions`, refer to the Enterprise Transport API C# Edition *Developers Guide*.
Flags	**Required**. Indicates the presence of optional service group members. For details, refer to Section 8.4.10.2.
Group	**Required**. Identifies the name of the item group being changed.
MergedToGroup	**Optional**. Specifies the new group name. All items of the specified `Group` are put into this new group. If present, a `Flags` value of **ServiceGroupFlags.HAS_MERGED_TO_GROUP** should be specified.
Status	**Optional**.Specifies the status to apply to all open items associated with the group specified by `Group`. If present, a `Flags` value of **ServiceGroupFlags.HAS_STATUS** should be specified.

Table 121: `ServiceGroup` Members

8.4.10.2   Service Group Flag Enumeration Values

FLAG ENUMERATION	DESCRIPTION
ServiceGroupFlags.HAS_MERGED_TO_GROUP	Indicates the presence of `MergedToGroup`.
ServiceGroupFlags.HAS_STATUS	Indicates the presence of `Status`.

Table 122: `ServiceGroup` Flags

8.4.10.3   Service Group Methods

METHOD NAME	DESCRIPTION
Clear	Clears a `ServiceGroup` instance. Useful for instance reuse.
Copy	Performs a deep copy of a `ServiceGroup` instance.

Table 123: `ServiceGroup` Utility Methods

8.4.11   Directory Service Load Filter

A `ServiceLoad` class conveys the workload of a service. It represents the Source Directory Load filter. For further details on the Service Load filter, refer to the Enterprise Transport API C# Edition *LSEG Domain Model Usage Guide*.

8.4.11.1   Service Load Members

MEMBER	DESCRIPTION
Action	**Required**. Indicates how a cache of the service load should apply this information. For information on `FilterEntryActions`, refer to the Enterprise Transport API C# Edition *Developers Guide*.
Flags	**Required**. Indicates presence of optional service load members. For details, refer to Section 8.4.11.2.
LoadFactor	If present, `Flags` value of **ServiceLoadFlags.HAS_LOAD_FACTOR** should be specified. Indicates the current workload on the source that provides data. A higher load factor indicates a higher workload. For more information, refer to the Enterprise Transport API C# Edition *LSEG Domain Model Usage Guide*.
OpenLimit	Specifies the maximum number of streaming requests that the service allows. If present, `Flags` value of **ServiceLoadFlags.HAS_OPEN_LIMIT** should be specified.
OenWindow	Specifies the maximum number of outstanding requests (i.e., requests awaiting a refresh) that the service allows. If present, `Flags` value of **ServiceLoadFlags.HAS_OPEN_WINDOW** should be specified.

Table 124: `ServiceLoad` Members

8.4.11.2   Service Load Flag Enumeration Values

FLAG ENUMERATION	DESCRIPTION
ServiceLoadFlags.HAS_LOAD_FACTOR	Indicates the presence of `LoadFactor`.
ServiceLoadFlags.HAS_OPEN_LIMIT	Indicates the presence of `OpenLimit`.
ServiceLoadFlags.HAS_OPEN_WINDOW	Indicates the presence of `OpenWindow`.

Table 125: `ServiceLoad` Flags

8.4.11.3 Service Load Methods

METHOD NAME	DESCRIPTION
Clear	Clears a **ServiceLoad** instance. Useful for instance reuse.
Copy	Performs a deep copy of a **ServiceLoad** instance.

Table 126: `ServiceLoad` Utility Methods

8.4.12 Directory Service Data Filter

A **ServiceData** class conveys the data to apply to all items of a service. It represents the Source Directory Data filter. For further details on the Data filter, refer to the Enterprise Transport API C# Edition *LSEG Domain Model Usage Guide*.

8.4.12.1 Service Data Members

MEMBER	DESCRIPTION
Action	**Required**. Indicates how a cache of the service data should apply this information. For further details on **FilterEntryActions**, refer to the Enterprise Transport API C# Edition *Developers Guide*.
Data	**Optional**. Contains the encoded **Buffer** representing the data. The type of the data is given by **DataType**. If present, a **Flags** value of **ServiceDataFlags.HAS_DATA** should be specified.
DataType	**Optional**. Specifies the **DataType** of the data. For information on **DataType**s, refer to the Enterprise Transport API C# Edition *Developers Guide*. If present, a **Flags** value of **ServiceDataFlags.HAS_DATA** should be specified.
Flags	**Required**. Indicates the presence of optional service data members. For details, refer to Section 8.4.12.2.
Type	**Optional**. Indicates the type of content present in **Data**. Available enumerations are: • **Rdm.Directory.DataTypes.TIME == 1** • **Rdm.Directory.DataTypes.ALERT == 2** • **Rdm.Directory.DataTypes.HEADLINE == 3** • **Rdm.Directory.DataTypes.STATUS == 4** If present, **Flags** value of **ServiceDataFlags.HAS_DATA** should be specified.

Table 127: `ServiceData` Members

8.4.12.2 Service Load Data Enumeration Values

FLAG ENUMERATION	DESCRIPTION
ServiceDataFlags.HAS_DATA	Indicates the presence of **Type**, **DataType**, and **Data**.

Table 128: `ServiceData` Flags

8.4.12.3   Service Data Methods

METHOD NAME	DESCRIPTION
Clear	Clears a **ServiceData** instance. Useful for instance reuse.
Copy	Performs a deep copy of a **ServiceData** instance.

    Table 129: `ServiceData` Methods

8.4.13   Directory Service Link Info Filter

A **ServiceLinkInfo** class conveys information about upstream sources that form a service. It represents the Source Directory Link filter. More information about the Service Link filter content is available in the Enterprise Transport API C# Edition *LSEG Domain Model Usage Guide*.

The **ServiceLinkInfo** class contains a list of **ServiceLink** classs that each represents an upstream source.

8.4.13.1   Service Link Info Members

MEMBER	DESCRIPTION
Action	**Required**. Indicates how a cache of the service link information should apply this information. For further information on **FilterEntryActions**, refer to the Enterprise Transport API C# Edition *Developers Guide*.
LinkList	**Optional**. Contains a list of **ServiceLink** class, each representing a source.

    Table 130: `ServiceLinkInfo` Members

8.4.13.2   Service Link Info Methods

METHOD NAME	DESCRIPTION
Clear	Clears a **ServiceLinkInfo** instance. Useful for instance reuse.
Copy	Performs a deep copy of a **ServiceLinkInfo** instance.

    Table 131: `ServiceLinkInfo` Methods

8.4.14   Directory Service Link

A **ServiceLink** class conveys information about an upstream source. It represents an entry in the Source Directory Link filter and is used by the **LinkList** member of the **ServiceLinkInfo** class. For further details on Service Link filter content, refer to the Enterprise Transport API C# Edition *LSEG Domain Model Usage Guide*.

8.4.14.1   Service Link Properties

MEMBER	DESCRIPTION
Action	**Required**. Indicates how a cache of the service link should apply this information. For information on **MapEntryActions**, refer to the Enterprise Transport API C# Edition *Developers Guide*.
Flags	**Required**. Indicates the presence of optional service link members. For details, refer to Section 8.4.14.2.
LinkCode	**Optional**. Indicates additional information about the status of a source. Available enumerations are: • **Rdm.Directory.LinkCodes.NONE == 0** • **Rdm.Directory.LinkCodes.OK == 1** • **Rdm.Directory.LinkCodes.RECOVERY_STARTED == 2** • **Rdm.Directory.LinkCodes.RECOVERY_COMPLETED == 3** If present, a **Flags** value of **ServiceLinkFlags.HAS_CODE** should be specified.
LinkState	**Required**. Indicates whether the source is up or down. Available values are: • **Directory.LinkStates.DOWN = 0** • **Directory.LinkStates.UP = 1**
Name	**Required**. Specifies the name of the source. Sources with identical names are typically load-balanced sources.
Text	**Optional**. Gives additional status details regarding the source. If present, a **Flags** value of **ServiceLinkFlags.HAS_TEXT** should be specified.
Type	**Optional**. Specifies whether the source is interactive or broadcast. Available enumerations are: • **Rdm.Directory.LinkTypes.INTERACTIVE == 1** • **Rdm.Directory.LinkTypes.BROADCAST == 2** If present, a **Flags** value of **ServiceLinkFlags.HAS_TYPE** should be specified.

Table 132: `ServiceLink` Members

8.4.14.2   Service Link Enumeration Values

FLAG ENUMERATION	DESCRIPTION
ServiceLinkFlags.HAS_CODE	Indicates the presence of **Code**.
ServiceLinkFlags.HAS_TEXT	Indicates the presence of **Text**.
ServiceLinkFlags.HAS_TYPE	Indicates the presence of **Type**.

Table 133: `ServiceLink` Flags

8.4.14.3   Service Link Methods

METHOD NAME	DESCRIPTION
Clear	Clears a **ServiceLink** instance. Useful for instance reuse.
Copy	Performs a deep copy of a **ServiceLink** instance.

Table 134: ServiceLink Methods

8.4.15   Directory Message

DirectoryMsg is the general purpose class that encapsulates all directory messages. Different directory messages are summarized in the following table.

INTERFACE	DESCRIPTION
DirectoryClose	RDM Directory Close.
DirectoryConsumerConnectionStatus	RDM Directory Consumer Status.
DirectoryRefresh	RDM Directory Refresh.
DirectoryRequest	RDM Directory Request.
DirectoryStatus	RDM Directory Status.
DirectoryUpdate	RDM Directory Update.

Table 135: DirectoryMsg Types

8.4.16   Directory Message Utility Methods

METHOD NAME	DESCRIPTION
Copy	Performs a deep copy of a **DirectoryMsg** instance.

Table 136: DirectoryMsg Utility Methods

8.4.17   Directory Encoding and Decoding

8.4.17.1   Directory Encoding and Decoding Methods

METHOD NAME	DESCRIPTION
Encode	Encodes a source directory message. This method takes the EncodeIterator as a parameter into which the encoded content is populated.
Decode	Decodes a source directory message. The decoded message may refer to encoded data from the original message. If the message is to be stored for later use, use the copy method of the decoded message to create a full copy.

Table 137: Directory Encoding and Decoding Methods

8.4.17.2   Encoding a Source Directory Request

```
ITransportBuffer msgBuf = chnl.GetBuffer(1024, false, out error);
DirectoryRequest directoryRequest = new DirectoryRequest();

directoryRequest.Clear();
directoryRequest.StreamId = 2;
directoryRequest.Filter = ServiceFilterFlags.INFO | ServiceFilterFlags.STATE |
    ServiceFilterFlags.GROUP;
directoryRequest.Streaming = true;

m_EncIter.Clear();
m_EncIter.SetBufferAndRWFVersion(msgBuf, chnl.MajorVersion, chnl.MinorVersion);
CodecReturnCode ret = m_DirectoryRequest.Encode(m_EncIter);
```

Code Example 16: Directory Request Encoding Example

8.4.17.3   Decoding a Source Directory Request

```
//msgBuf is a buffer contains the encoded message
Msg msg = new Msg();

m_DecIter.Clear();
m_DecIter.SetBufferAndRWFVersion(msgBuf, chnl.MajorVersion, chnl.MinorVersion);
msg.Decode(m_DecIter);
if (msg.MsgClass == MsgClasses.REQUEST)
{
   m_DirectoryRequest.Clear();
   if (m_DirectoryRequest.Decode(dIter, msg) != CodecReturnCode.SUCCESS)
    {
        Console.WriteLine("Failed decoding Directory Request message");
    }
}
```

Code Example 17: Directory Request Decoding Example

8.4.17.4   Encoding a Source Directory Refresh

```
ITransportBuffer msgBuf = chnl.GetBuffer(REFRESH_MSG_SIZE, false, out error);

// encode source directory request
m_DirectoryRefresh.Clear();
m_DirectoryRefresh.StreamId = 2;

// clear cache
m_DirectoryRefresh.ClearCache = true;
m_DirectoryRefresh.Solicited = true;

// state information for response message
m_DirectoryRefresh.State.Clear();
m_DirectoryRefresh.State.StreamState(StreamStates.OPEN);
m_DirectoryRefresh.State.DataState(DataStates.OK);
m_DirectoryRefresh.State.Code(StateCodes.NONE);

// attribInfo information for response message
m_DirectoryRefresh.Filter = Directory.ServiceFilterFlags.INFO | Directory.ServiceFilterFlags.STATE |
    Directory.ServiceFilterFlags.GROUP;

// populate the Service
m_Service.Clear();
m_Service.Action = MapEntryActions.ADD;

// set the service Id (map key)
m_Service.ServiceId = ServiceId;

m_Service.HasInfo = true;
m_Service.Info.Action = FilterEntryActions.SET;

// vendor
m_Service.Info.HasVendor = true;
m_Service.Info.Vendor.Data(Vendor);

// service name - required
m_Service.Info.ServiceName.Data(ServiceName);

// Qos Range is not supported
m_Service.Info.HasSupportQosRange = true;
m_Service.Info.SupportsQosRange = 0;

m_Service.Info.CapabilitiesList.Add((long)Rdm.DomainType.MARKET_PRICE);
m_Service.Info.CapabilitiesList.Add((long)Rdm.DomainType.MARKET_BY_ORDER);
m_Service.Info.CapabilitiesList.Add((long)Rdm.DomainType.MARKET_BY_PRICE);
m_Service.Info.CapabilitiesList.Add((long)Rdm.DomainType.DICTIONARY);

// qos
m_Service.Info.HasQos = true;
Qos qos = new Qos();
qos.Rate(QosRates.TICK_BY_TICK);
qos.Timeliness(QosTimeliness.REALTIME);
m_Service.Info.QosList.Add(qos);

// isSource = Service is provided directly from original publisher
m_Service.Info.HasIsSource = true;
m_Service.Info.IsSource = 1;
```

```
// accepting customer status = no
m_Service.Info.HasAcceptingConsStatus = true;
m_Service.Info.AcceptConsumerStatus = 0;

// supports out of band snapshots = no
m_Service.Info.HasSupportOOBSnapshots = true;
m_Service.Info.SupportsOOBSnapshots = 0;

m_Service.HasState = true;
m_Service.State.Action = FilterEntryActions.SET;

// service state
m_Service.State.ServiceStateVal = 1;

// accepting requests
m_Service.State.HasAcceptingRequests = true;
m_Service.State.AcceptingRequests = 1;

m_Service.HasLoad = true;
m_Service.Load.Action = FilterEntryActions.SET;

// open limit
m_Service.Load.HasOpenLimit = true;
m_Service.Load.OpenLimit = OPEN_LIMIT;

m_Service.HasLink = true;
m_Service.Link.Action = FilterEntryActions.SET;
ServiceLink serviceLink = new ServiceLink();

// link name - Map Entry Key
serviceLink.Name.Data(LinkName);

// link type
serviceLink.HasType = true;
serviceLink.Type = Rdm.Directory.LinkTypes.INTERACTIVE;

// link text
serviceLink.HasText = true;
serviceLink.Text.Data("Link state is up");
m_Service.Link.LinkList.Add(serviceLink);

m_DirectoryRefresh.ServiceList.Add(m_Service);

// encode directory request
m_EncodeIter.Clear();
CodecReturnCode ret = m_EncodeIter.SetBufferAndRWFVersion(msgBuf, chnl.MajorVersion,
    chnl.MinorVersion);

ret = m_DirectoryRefresh.Encode(m_EncodeIter);
```

Code Example 18: Directory Refresh Encoding Example

8.4.17.5   Decoding a Source Directory Refresh

```
//msgBuf is a buffer contains the encoded message
Msg msg = new Msg();

m_DecIter.Clear();
m_DecIter.SetBufferAndRWFVersion(msgBuf, chnl.MajorVersion, chnl.MinorVersion);
msg.Decode(m_DecIter);
if (msg.MsgClass == MsgClasses.REFRESH)
{
    m_DirectoryRefresh.Clear();
    if (m_DirectoryRefresh.Decode(dIter, msg) != CodecReturnCode.SUCCESS)
    {
        Console.WriteLine("Failed decoding Directory Request message");
    }

}
```

Code Example 19: Directory Refresh Decoding Example

8.5   RDM Dictionary Domain

The Dictionary domain model conveys information needed for parsing published data. Dictionaries provide additional meta-data, such as that necessary to decode the content of a `FieldEntry` or additional content related to its `FieldId`. For more information about the different types of dictionaries and their usage, refer to the Enterprise Transport API C# Edition *LSEG Domain Model Usage Guide*.

This domain's interface makes it easier to use the existing utilities for encoding, decoding, and caching dictionary information. For more information on these utilities, see the Enterprise Transport API C# Edition *LSEG Domain Model Usage Guide*.

8.5.1   Dictionary Request

A *Dictionary Request* message is encoded and sent by OMM consumer applications. This message requests a dictionary from a service.

The `DictionaryRequest` represents all members of a dictionary request message and is easily used in OMM applications that leverage RDMs. This class follows the behavior and layout that is defined in the Enterprise Transport API C# Edition *LSEG Domain Model Usage Guide*.

8.5.1.1   Dictionary Request Properties

MEMBER	DESCRIPTION
DictionaryName	**Required**. Indicates the name of the dictionary being requested.
Flags	**Required**. Indicates the presence of optional dictionary request members. For details, refer to Section 8.5.1.2.
RdmMsgBase	**Required**.Specifies the message type (i.e., Dictionary). For a dictionary request, send **DictionaryMsgType.REQUEST**.
ServiceId	**Required**. Specifies the service from which to request the dictionary.
Verbosity	**Required**. Indicates the amount of information desired from the dictionary. Available enumerations are: • **Rdm.Dictionary.VerbosityValues.INFO == 0x00**: Version information only • **Rdm.Dictionary.VerbosityValues.MINIMAL == 0x03**: Provides information needed for caching • **Rdm.Dictionary.VerbosityValues.NORMAL == 0x07**: Provides all information needed for decoding • **Rdm.Dictionary.VerbosityValues.VERBOSE == 0x0F**: Provides all information (including comments) Providers are not required to support the **MINIMAL** and **VERBOSE** filters.

Table 138: `DictionaryRequest` Members

8.5.1.2   Dictionary Request Flag Enumeration Values

FLAG ENUMERATION	DESCRIPTION
DictionaryRequestFlags.STREAMING	Indicates that the dictionary stream should remain open after the initial refresh. An open stream can listen for status messages that indicate changes to the dictionary version. For more information, see the Enterprise Transport API C# Edition *LSEG Domain Model Usage Guide*.

Table 139: `DictionaryRequest` Flag

8.5.2   Dictionary Refresh

A *Dictionary Refresh* message is encoded and sent by OMM provider applications. This message transmits dictionary content in response to a request.

The `DictionaryRefresh` represents all members of a dictionary refresh message and is easy to use in OMM applications that leverage RDMs. This class follows the behavior and layout that is defined in the Enterprise Transport API C# Edition *LSEG Domain Model Usage Guide*.

8.5.2.1   Dictionary Refresh Properties

MEMBER	DESCRIPTION
DataBody	When decoding, this points to the encoded data buffer with dictionary content. This buffer should be set on a `DecodeIterator` and passed to the appropriate decode method according to the `type`. Not used when encoding. The dictionary is retrieved from the `DataDictionary`.
Dictionary	Conditional (**required** when encoding). Points to a `DataDictionary` object that contains content to encode. For more information on the `DataDictionary` class, refer to the Enterprise Transport API C# Edition *LSEG Domain Model Usage Guide*. Not used when decoding.
DictionaryName	**Required**. Indicates the name of the dictionary being provided.
Flags	**Required**. Indicates the presence of optional dictionary refresh members. For details, refer to Section 8.5.2.2.
RdmMsgBase	**Required**. Specifies the message type (i.e., dictionary message). For a dictionary refresh, set to **DictionaryMsgType.REFRESH**.
SequenceNumber	**Optional**. A user-specified, item-level sequence number that the application can use to sequence messages in this stream. If present, a `Flags` value of **DictionaryRefreshFlags.HAS_SEQ_NUM** should be specified.
ServiceId	**Required**. Indicates the service ID of the service from which the dictionary is provided.
StartFid	Maintains the state when encoding a dictionary across multiple messages. ⚠️ **Warning!** To ensure that all dictionary content is correctly encoded, the application should not modify this.
State	**Required**. Indicates the state of the dictionary stream. Defaults to a `StreamState` of **StreamStates.OPEN** and a `DataState` of **DataStates.OK**. For more information on `State`, refer to the Enterprise Transport API C# Edition *Developers Guide*.

Table 140: `DictionaryRefresh` Members

MEMBER	DESCRIPTION
DictionaryType	**Required**. Indicates the type of dictionary being provided. The dictionary encoder and decoder support the following types: • **Rdm.Dictionary.VerbosityValues.FIELD_DEFINITIONS == 1** • **Rdm.Dictionary.VerbosityValues.ENUM_TABLES == 2**
Verbosity	**Required**. Indicates the amount of information desired from the dictionary. Available enumerations are: • **Rdm.Dictionary.VerbosityValues.INFO == 0x00**: Provides version information only • **Rdm.Dictionary.VerbosityValues.MINIMAL == 0x03**: Provides information needed for caching • **Rdm.Dictionary.VerbosityValues.NORMAL == 0x07**: Provides all information needed for decoding • **Rdm.Dictionary.VerbosityValues.VERBOSE == 0x0F**: Provides all information (including comments) Providers do not need to support the **MINIMAL** and **VERBOSE** filters.

Table 140: `DictionaryRefresh` Members (Continued)

8.5.2.2   Dictionary Refresh Flag Enumeration Values

FLAG ENUMERATION	DESCRIPTION
DictionaryRefreshFlags.CLEAR_CACHE	Indicates that stored payload information associated with the dictionary stream should be cleared. This might happen if some portion of data is known to be invalid.
DictionaryRefreshFlags.HAS_INFO	Indicates the presence of `DictionaryType`. Not used when encoding. The `Encode` method adds information to the encoded message when appropriate.
DictionaryRefreshFlags.HAS_SEQ_NUM	Indicates the presence of `SequenceNumber`.
DictionaryRefreshFlags.IS_COMPLETE	Indicates that this is the final fragment and that the consumer has received all content for this dictionary. Not used when encoding. The `Encode` method adds information to the encoded message when appropriate.
DictionaryRefreshFlags.SOLICITED	Indicates that the directory refresh is solicited (e.g., it is a response to a request). If the flag is not present, this refresh is unsolicited.

Table 141: `DictionaryRefresh`

8.5.3   Dictionary Status

OMM provider and non-interactive provider applications use the *Dictionary Status* message to convey state information associated with the dictionary stream. Such state information can indicate that a dictionary stream cannot be established or to inform a consumer of a state change associated with an open dictionary stream. The Dictionary status message can also indicate that a new dictionary should be retrieved. For more information on handling Dictionary versions, see the Enterprise Transport API C# Edition *LSEG Domain Model Usage Guide*.

The `DictionaryStatus` represents all members of a dictionary status message and allows for simplified use in OMM applications that leverage RDMs. This class follows the behavior and layout that is defined in the Enterprise Transport API C# Edition *LSEG Domain Model Usage Guide*.

8.5.3.1   Dictionary Status Members

MEMBER	DESCRIPTION
Flags	**Required**. Indicate the presence of optional dictionary status members. For details, refer to Section 8.5.3.2.
State	**Optional**. Indicates the state of the dictionary stream. For more information on **State**, refer to the Enterprise Transport API C# Edition *Developers Guide*. If present, a **flags** value of **DictionaryStatusFlags.HAS_STATE** should be specified.

Table 142: `DictionaryStatus` Members

8.5.3.2   Dictionary Status Flag Enumeration Value

FLAG ENUMERATION	DESCRIPTION
DictionaryStatusFlags.CLEAR_CACHE	Indicates that any stored payload information associated with the dictionary stream should be cleared. This might happen if some portion of data is known to be invalid.
DictionaryStatusFlags.HAS_STATE	Indicates the presence of **State**. If absent, any previously conveyed state continues to apply.

Table 143: `DictionaryStatus` Flags

8.5.4   Dictionary Close

A ***Dictionary Close*** message is encoded and sent by OMM consumer applications. This message allows a consumer to close an open dictionary stream. A provider can close the directory stream via a Dictionary Status message; for details, refer to Section 8.5.3.

8.5.5   Dictionary Messages

DictionaryMsg is the base interface for all Dictionary messages. It is provided for use with general dictionary-specific functionality.

INTERFACE	DESCRIPTION
DictionaryClose	RDM Dictionary Close.
DictionaryRefresh	RDM Dictionary Refresh.
DictionaryRequest	RDM Dictionary Request.
DictionaryStatus	RDM Dictionary Status.

Table 144: `DictionaryMsg` Types

8.5.6   Dictionary Message: Utility Methods

FUNCTION NAME	DESCRIPTION
Copy	Performs a deep copy of a **DictionaryMsg** instance.

Table 145: `DictionaryMsg` Utility Methods

8.5.7   Dictionary Encoding and Decoding

8.5.7.1   Dictionary Encoding and Decoding Methods

METHOD NAME	DESCRIPTION
Encode	Encodes a dictionary message. This method takes the EncodeIterator as a parameter into which the encoded content is populated.
Decode	Decodes a dictionary message. The decoded message may refer to encoded data from the original message. If the message is to be stored for later use, use the copy method of the decoded message to create a full copy.

Table 146: Dictionary Encoding and Decoding Methods

8.5.7.2   Encoding a Dictionary Request

```
ITransportBuffer msgBuf = chnl.GetTransportBuffer(TRANSPORT_BUFFER_SIZE_REQUEST, false, out error);

DictionaryRequest dictionaryRequest = new DictionaryRequest();

dictionaryRequest.DictionaryName.Data("RWFFld");
dictionaryRequest.StreamId = 4;
dictionaryRequest.ServiceId = ServiceId;
dictionaryRequest.Verbosity = Rdm.Dictionary.VerbosityValues.NORMAL;

encIter.Clear();
encIter.SetBufferAndRWFVersion(msgBuf, chnl.Channel!.MajorVersion, chnl.Channel.MinorVersion);

CodecReturnCode ret = dictionaryRequest.Encode(encIter);
```

Code Example 20: Dictionary Request Encoding Example

8.5.8   Decoding a Dictionary Request

```
//msgBuf is a buffer contains the encoded message
Msg msg = new Msg();

m_DecIter.Clear();
m_DecIter.SetBufferAndRWFVersion(msgBuf, chnl.MajorVersion, chnl.MinorVersion);
msg.Decode(m_DecIter);
if (msg.MsgClass == MsgClasses.REQUEST)
{
   m_DictionaryRequest.Clear();
   if (m_DictionaryRequest.Decode(dIter, msg) != CodecReturnCode.SUCCESS)
    {
        Console.WriteLine("Failed decoding Directory Request message");
    }
   else
   {
      Console.WriteLine(#"DictionaryRequest.StreamId: {m_DictionaryRequest.StreamId}");
      Console.WriteLine(#"DictionaryRequest.DictionaryName: {m_DictionaryRequest.DictionaryName}");
   }
}
```

Code Example 21: Dictionary Request Decoding Example

8.5.8.1   Encoding a Dictionary Refresh

```
m_DictionaryRefresh.Clear();

m_DictionaryRefresh.StreamId = 4;
m_DictionaryRefresh.DictionaryType = Rdm.Dictionary.Types.FIELD_DEFINITIONS;
m_DictionaryRefresh.DataDictionary = Dictionary;
m_DictionaryRefresh.State.StreamState(StreamStates.OPEN);
m_DictionaryRefresh.State.DataState(DataStates.OK);
m_DictionaryRefresh.State.Code(StateCodes.NONE);
m_DictionaryRefresh.Verbosity = Rdm.Dictionary.VerbosityValues.NORMAL;
m_DictionaryRefresh.ServiceId = ServiceId;
m_DictionaryRefresh.DictionaryName.Data("RWFFld");
m_DictionaryRefresh.Solicited = true;

ITransportBuffer msgBuf = chnl.GetBuffer(MAX_FIELD_DICTIONARY_MSG_SIZE, false, out error);
m_EncodeIter.Clear();
m_EncodeIter.SetBufferAndRWFVersion(msgBuf, chnl.MajorVersion, chnl.MinorVersion);
CodecReturnCode ret = m_DictionaryRefresh.Encode(m_EncodeIter);
```

Code Example 22: Dictionary Refresh Encoding Example

8.5.8.2   Decoding a Dictionary Refresh

```
//msgBuffer contains the encoded message
Msg msg = new Msg();

m_DecIter.Clear();
m_DecIter.SetBufferAndRWFVersion(msgBuf, chnl.MajorVersion, chnl.MinorVersion);
msg.Decode(m_DecIter);

CodecReturnCode ret = m_DictionaryRefresh.Decode(dIter, msg);

if (m_DictionaryRefresh.HasInfo)
{
   if (m_DictionaryRefresh.DictionaryType != Rdm.Dictionary.Types.FIELD_DEFINITIONS &&
   m_DictionaryRefresh.DictionaryType != Rdm.Dictionary.Types.ENUM_TABLES)
      Console.WriteLine("Invalid Dictionary Type");
}

if (m_DictionaryRefresh.StreamId == FIELD_DICTIONARY_STREAM_ID)
{
    Console.WriteLine("Received Dictionary Refresh for field dictionary");

    IRefreshMsg refreshMsg = msg;
   ret = Dictionary.DecodeFieldDictionary(dIter, Rdm.Dictionary.VerbosityValues.VERBOSE, out
   CodecError codecError);
    if (m_DictionaryRefresh.RefreshComplete)
      Console.WriteLine("Field Dictionary complete, waiting for Enum Table...");
}
```

Code Example 23: Dictionary Refresh Decoding Example

Appendix A   Value Added Utilities

Value Added Utilities are a collection of common classes used mainly by the Transport API Reactor. Included is a selectable, bidirectional queue that can communicate events between the Reactor and Worker threads. Other Value Added Utilities include a simple queue along with iterable and concurrent versions of it.

The Value Added Utilities are internally leveraged by the Transport API Reactor and cache so applications need not be familiar with their use.

Document ID: ETACSharp331UMVAC.240
Date of issue: December 2024