Enterprise Transport API Java Edition 3.6.7.L1

VALUE ADDED COMPONENTS

Document Version: 3.6.7
Date of issue: October 2022
Document ID: ETAJ367UMVAC.220

REFINITIV™

Contents

1 Value Added Components Developers Guide Introduction 1
1.1 About this Manual 1
1.2 Audience 1
1.3 Programming Language 1
1.4 Acronyms and Abbreviations 1
1.5 Additional References and Resources 3
1.6 Documentation Feedback 3
1.7 Document Conventions 3
1.7.1 Optional, Conditional, and Required 3
1.7.2 Typographic 3
1.7.3 Document Structure 4
1.7.4 Diagrams 4

2 Product Description and Overview 7
2.1 What is the Enterprise Transport API? 7
2.2 What are Enterprise Transport API Value Added Components? 8
2.3 Transport API Reactor 9
2.4 OMM Consumer Watchlist 9
2.4.1 Data Stream Aggregation and Recovery 9
2.4.2 Additional Features 9
2.4.3 Usage Notes 10
2.5 Administration Domain Model Representations 10
2.6 Value Added Utilities 10
2.7 Payload Cache 10

3 Building an OMM Consumer 11
3.1 Building an OMM Consumer Overview 11
3.2 Leverage Existing or Create New Reactor 11
3.3 Implement Callbacks and Populate Role 12
3.4 Establish Connection using Reactor.connect 12
3.5 Issue Requests and/or Post Information 12
3.6 Log Out and Shut Down 13
3.7 Additional Consumer Details 13

4 Building an OMM Interactive Provider 14
4.1 Building an OMM Interactive Provider Overview 14
4.2 Leverage Existing or Create New Reactor 14
4.3 Create a Server 14
4.4 Implement Callbacks and Populate Role 15
4.5 Associate Incoming Connections Using Reactor.accept 15
4.6 Perform Login Process 15
4.7 Provide Source Directory Information 16
4.8 Provide or Download Necessary Dictionaries 16
4.9 Handle Requests and Post Messages 17
4.10 Dispatch Round Trip Time Messages 17
4.11 Disconnect Consumers and Shut Down 17
4.12 Additional Interactive Provider Details 18

5 Building an OMM Non-Interactive Provider 19
5.1 Building an OMM Non-Interactive Provider Overview 19

5.2 Leverage Existing or Create New Reactor 19
5.3 Implement Callbacks and Populate Role 20
5.4 Establish Connection using Reactor.connect 20
5.5 Download the Dictionary 20
5.6 Provide Content 21
5.7 Log Out and Shut Down 21
5.8 Additional Non-Interactive Provider Details 21

6 Reactor Detailed View 22
6.1 Concepts 22
6.1.1 Functionality: Enterprise Transport API Versus Enterprise Transport API Reactor 23
6.1.2 Reactor Error Handling 23
6.1.3 Reactor Error Info Codes 24
6.1.4 Enterprise Transport API Reactor Application Lifecycle 25
6.2 Reactor Use 25
6.2.1 Creating a Reactor 26
6.2.2 Destroying a Reactor 28
6.3 Reactor Channel Use 29
6.3.1 Reactor Channel Roles 30
6.3.2 Reactor Channel Role: OMM Consumer 32
6.3.3 Reactor Channel Role: OMM Provider 34
6.3.4 Reactor Channel Role: OMM Non-Interactive Provider 36
6.4 Managing Reactor Channels 37
6.4.1 Adding Reactor Channels 37
6.4.2 Removing Reactor Channels 42
6.5 Reporting on Channel Statistics 43
6.6 Dispatching Data 43
6.6.1 Reactor Dispatch Methods 43
6.6.2 Reactor Callback Methods 46
6.6.3 Reactor Callback: Channel Event 46
6.6.4 Reactor Callback: Default Message 50
6.6.5 Reactor Callback: RDM Login Message 51
6.6.6 Reactor Callback: RDM Directory Message 52
6.6.7 Reactor Callback: RDM Dictionary Message 56
6.7 Writing Data 57
6.7.1 Writing Data using ReactorChannel.submit(Msg…) 57
6.7.2 Writing Data Using ReactorChannel.submit(TransportBuffer...) 62
6.8 Creating and Using Tunnel Streams 72
6.8.1 Authenticating a Tunnel Stream 73
6.8.2 Opening a Tunnel Stream 73
6.8.3 Negotiating Stream Behaviors: Class of Service 75
6.8.4 Tunnel Stream Callback Methods and Event Types 79
6.8.5 Code Sample: Opening and Managing a Tunnel Stream 81
6.8.6 Accepting Tunnel Streams 83
6.8.7 Receiving Content on a TunnelStream 88
6.8.8 Sending Content on a TunnelStream 88
6.8.9 Closing a Tunnel Stream 91
6.9 Cloud Connectivity 92
6.9.1 Querying Service Discovery 92
6.9.2 OAuth Credential Management 94
6.10 JSON to RWF Protocol Conversion for WebSocket Support 96
6.10.1 Consumer and WebSocket Support 97
6.10.2 Interactive Provider and WebSocket Support 99
6.10.3 ReactorJsonConverterOptions 101
6.10.4 ReactorJsonConversionEventCallback Function 102

6.10.5 ReactorServiceNameToIdCallback Function ... 103
6.10.6 ReactorServiceNameToIdEvent Class ... 103
6.10.7 ReactorJsonConversionEvent Class ... 103
6.11 Reactor Utility Methods ... 104
6.11.1 General Reactor Utility Methods ... 104
6.11.2 ReactorChannelInfo Class Members ... 104
6.11.3 ReactorChannel.ioctl Option Values ... 104
6.11.4 Debugging for Reactor ... 104

7 Consuming Data from the Cloud ... 106
7.1 Overview ... 106
7.2 Encrypted Connections ... 106
7.3 Credential Management ... 106
7.4 Version 1 Authentication Using oAuth Password and Refresh_Token ... 107
7.4.1 Client_ID (AppKey) ... 107
7.4.2 Obtaining Initial Access and Refresh Tokens ... 107
7.4.3 Refreshing the Access Token and Sending a Login Reissue ... 108
7.4.4 Session Management per User Credential ... 108
7.5 Version 2 Authentication Using oAuth Client Credentials ... 109
7.5.1 Configuring and Managing Version 2 Credentials ... 109
7.5.2 Version 2 oAuth Client Credentials Token Lifespan ... 109
7.6 Service Discovery ... 110
7.7 Consuming Market Data ... 111
7.8 HTTP Error Handling for Reactor Token Reissues ... 111
7.9 Cloud Connection Use Cases ... 112
7.9.1 Session Management Use Case ... 112
7.9.2 Disabling the Watchlist ... 112
7.9.3 Query Service Discovery ... 112
7.10 Logging of Authentication and Service Discovery Interaction ... 113
7.10.1 Logged Request Information ... 113
7.10.2 Logged Response Information ... 113

8 Administration Domain Models Detailed View ... 114
8.1 Concepts ... 114
8.2 Message Base ... 115
8.2.1 Message Base Members ... 115
8.2.2 Message Base Method ... 115
8.2.3 RDM Message Types ... 116
8.3 RDM Login Domain ... 117
8.3.1 Login Request ... 117
8.3.2 Login Refresh ... 122
8.3.3 Login Status ... 132
8.3.4 Login Close ... 134
8.3.5 Login Consumer Connection Status ... 134
8.3.6 Login Round Trip Time Message Use ... 135
8.3.7 Login Post Message Use ... 136
8.3.8 Login Ack Message Use ... 136
8.3.9 Login Attributes ... 137
8.3.10 Login Message ... 139
8.3.11 Login Message Utility Method ... 140
8.3.12 Login Encoding and Decoding ... 140
8.4 RDM Source Directory Domain ... 145
8.4.1 Directory Request ... 145
8.4.2 Directory Refresh ... 147
8.4.3 Directory Update ... 148

8.4.4 Directory Status ... 149
8.4.5 Directory Close ... 150
8.4.6 Directory Consumer Status ... 151
8.4.7 Directory Service ... 152
8.4.8 Directory Service Info Filter ... 153
8.4.9 Directory Service State Filter ... 155
8.4.10 Directory Service Group Filter ... 156
8.4.11 Directory Service Load Filter ... 157
8.4.12 Directory Service Data Filter ... 158
8.4.13 Directory Service Link Info Filter ... 159
8.4.14 Directory Service Link ... 160
8.4.15 Directory Message ... 161
8.4.16 Directory Message Utility Methods ... 161
8.4.17 Directory Encoding and Decoding ... 162
8.5 RDM Dictionary Domain ... 168
8.5.1 Dictionary Request ... 168
8.5.2 Dictionary Refresh ... 169
8.5.3 Dictionary Status ... 170
8.5.4 Dictionary Close ... 171
8.5.5 Dictionary Messages ... 171
8.5.6 Dictionary Message: Utility Methods ... 172
8.5.7 Dictionary Encoding and Decoding ... 172
8.6 RDM Queue Messages ... 178
8.6.1 Queue Data Message Persistence ... 178
8.6.2 Queue Request ... 178
8.6.3 Queue Refresh ... 179
8.6.4 Queue Status ... 179
8.6.5 Queue Close ... 179
8.6.6 Queue Data ... 181
8.6.7 QueueDataExpired ... 184
8.6.8 Queue Ack ... 185

9 Warm Standby Feature ... 186
9.1 Overview ... 186
9.2 Warm Standby Modes ... 186
9.3 Warm Standby Configuration and Feature Details ... 188
9.3.1 Configuration Example for a Staring Server and a Standby Server ... 190
9.4 ... 191

10 Payload Cache Detailed View ... 192
10.1 Concepts ... 192
10.2 Payload Cache ... 193
10.2.1 Payload Cache Management ... 193
10.2.2 Cache Error Handling ... 193
10.2.3 Payload Cache Instances ... 194
10.2.4 Managing RDM Field Dictionaries for Payload Cache ... 194
10.2.5 Payload Cache Utilities ... 196
10.3 Payload Cache Entries ... 197
10.3.1 Managing Payload Cache Entries ... 197
10.3.2 Applying Data ... 199
10.3.3 Retrieving Data ... 200

Appendix A Value Added Utilities ... 203

List of Figures

Figure 1. Network Diagram Notation 5
Figure 2. UML Diagram Notation 6
Figure 3. OMM APIs with Value Added Components 7
Figure 4. Enterprise Transport API Value Added Components 8
Figure 5. ETA Reactor Thread Model 22
Figure 6. Enterprise Transport API Reactor Application Lifecycle 25
Figure 7. Flow Chart for writing data via ReactorChannel.submit(TransportBuffer...) 64
Figure 8. Tunnel Stream Illustration 72
Figure 9. Obtaining an Authentication Token 107
Figure 10. Login Reissue 108
Figure 11. Service Discovery 110
Figure 12. Login Based Warm Standby Order of Events in a Cutover from Active to Standby 187
Figure 13. Service Based Warm Standby Order of Events in a Cutover from Active to Standby 188
Figure 14. Consumer Application using Cache to Store Payload Data for Item Streams 192

List of Tables

Table 1: Acronyms and Abbreviations 1
Table 2: ETA Functionality and ETA Reactor Comparison 23
Table 3: `ReactorErrorInfo` Structure Members 24
Table 4: Reactor Error Info Codes 24
Table 5: `Reactor` Structure Class Members 26
Table 6: `Reactor` Creation Method 26
Table 7: `ReactorOptions` Class Members 26
Table 8: `ReactorOptions` Utility Method 27
Table 9: Reactor Debugging Levels 28
Table 10: `Reactor` Destruction Method 28
Table 11: `ReactorChannel` Class Members 29
Table 12: `ReactorRole` Class Members 31
Table 13: `ReactorRole.roleTypes` Enumerated Values 31
Table 14: `ConsumerRole` Class Members 32
Table 15: `ConsumerRole.dictionaryDownloadMode` Enumerated Values 33
Table 16: OMM Consumer Role Watchlist Options 33
Table 17: `ConsumerRole` Utility Method 34
Table 18: `ProviderRole` Class Members 34
Table 19: `ProviderRole` Utility Method 35
Table 20: `NIProviderRole` Class Members 36
Table 21: `NIProviderRole` Utility Method 36
Table 22: `Reactor.connect` Method 37
Table 23: ReactorConnectOptions Class Members 37
Table 24: `ReactorConnectInfo` Class Members 38
Table 25: `ReactorConnectOptions` Utility Method 39
Table 26: `Reactor.accept` Method 41
Table 27: `ReactorAcceptOptions` Class Members 41
Table 28: `RsslReactorAcceptOptions` Utility Method 41
Table 29: `ReactorChannel.close` Function 42
Table 30: `ReactorChannel.dispatch` Reactor Dispatch Methods 44
Table 31: `ReactorDispatchOptions` Class Members 44
Table 32: `ReactorDispatchOptions` Utility Method 44
Table 33: `RsslReactorCallbackReturnCodes` Callback Return Codes 46
Table 34: `ReactorEvent` Class Members 46
Table 35: `ReactorChannelEvent` Class Members 46
Table 36: `RsslReactorChannelEventType` Enumeration Values 47
Table 37: `ReactorChannelEvent` Utility Methods 48
Table 38: `ReactorMsgEvent` Class Members 50
Table 39: `ReactorMsgEvent` Utility Method 50
Table 40: `RDMLoginMsgEvent` Class Members 51
Table 41: `RDMLoginMsgEvent` Utility Method 52
Table 42: `RDMDirectoryMsgEvent` Class Members 54
Table 43: `RDMDirectoryMsgEvent` Utility Method 54
Table 44: `RDMDictionaryMsgEvent` Class Members 56
Table 45: `RDMDictionaryMsgEvent` Utility Method 56
Table 46: `ReactorChannel.submit(Msg...)` Method 57
Table 47: `ReactorSubmitOptions` Class Members 59
Table 48: `ReactorChannel.submit(Msg...)` Return Codes 59
Table 49: `ReactorRequestMsgOptions` Class Members 60
Table 50: `ReactorSubmitOptions` Utility Method 60
Table 51: ReactorChannel Buffer Management Methods 65

Table 52: ReactorChannel.getBuffer Return Values ... 66
Table 53: ReactorChannel.submit(TransportBuffer...) Method ... 66
Table 54: ReactorChannel.submit(TransportBuffer...) Return Codes ... 67
Table 55: ReactorChannel.packBuffer Method ... 69
Table 56: ReactorChannel.packBuffer Return Values ... 69
Table 57: TunnelStreamAuthInfo Structure Members ... 73
Table 58: ReactorChannel.openTunnelStream Method ... 73
Table 59: TunnelStreamOpenOptions ... 74
Table 60: ClassOfService.common Structure Members ... 75
Table 61: ClassOfService.authentication Structure Members ... 76
Table 62: ClassOfService.flowControl Structure Members ... 76
Table 63: ClassOfService.dataIntegrity Structure Members ... 77
Table 64: ClassOfService.guarantee Structure Members ... 78
Table 65: Tunnel Stream Callback Methods ... 79
Table 66: Tunnel Stream Callback Event Types ... 80
Table 67: TunnelStreamRequestEvent Structure Members ... 84
Table 68: ReactorChannel.acceptTunnelStream Method ... 84
Table 69: TunnelStreamAcceptOptions Options ... 85
Table 70: ReactorChannel.rejectTunnelStream Method ... 85
Table 71: TunnelStreamRejectOptions Options ... 85
Table 72: Tunnel Stream Buffer Methods ... 88
Table 73: Tunnel Stream Submit Method ... 88
Table 74: TunnelStreamSubmitOptions Structure Members ... 90
Table 75: TunnelStreamInfo Structure Members Methods ... 90
Table 76: Tunnel Closure Method ... 91
Table 77: Reactor.queryServiceDiscovery Method ... 92
Table 78: ReactorServiceDiscoveryOptions Structure Members ... 92
Table 79: ReactorDiscoveryTransportProtocol Enumerations ... 93
Table 80: ReactorDiscoveryDataFormatProtocol Enumerations ... 93
Table 81: ReactorServiceEndpointEvent Structure Members ... 93
Table 82: ReactorServiceEndpointEvent Structure Members ... 94
Table 83: ReactorOAuthCredential Structure Class Members ... 94
Table 84: ReactorOAuthCredentialEvent Structure Members ... 95
Table 85: RsslReactorOAuthCredentialRenewal Members ... 95
Table 86: ReactorOAuthCredentialRenewal Options ... 96
Table 87: ReactorOAuthCredentialRenewalMode Enums ... 96
Table 88: ReactorJsonConverterOptions Class Members ... 101
Table 89: ReactorServiceNameToIdCallback Parameters ... 103
Table 90: ReactorServiceNameToIdEvent Class Members ... 103
Table 91: ReactorJsonConversionEvent Class Members ... 103
Table 92: Reactor Utility Methods ... 104
Table 93: ReactorChannelInfo Class Members ... 104
Table 94: Reactor Methods for Debugging Handling ... 105
Table 95: Domains Representations in the Administration Domain Model Value Added Component ... 114
Table 96: MsgBase Structure Members ... 115
Table 97: MsgBase Method ... 115
Table 98: RsslRDMMsg Domain Representations Message Types ... 116
Table 99: LoginRequest Structure Members ... 117
Table 100: LoginRequest Flags ... 120
Table 101: LoginRequest Utility Methods ... 121
Table 102: LoginRefresh Structure Members ... 122
Table 103: LoginRefresh Flags ... 126
Table 104: LoginSupportFeatures Members ... 128
Table 105: LoginSupportFeaturesFlags ... 129
Table 106: LoginConnectionConfig Members ... 130

Table 107: LoginConnectionConfig Methods 130
Table 108: ServerInfo Structure Members 130
Table 109: ServerInfo Flags 131
Table 110: ServerInfo Utility Methods 131
Table 111: LoginStatus Structure Members 132
Table 112: LoginStatus Flags 133
Table 113: LoginClose Member 134
Table 114: LoginConsumerConnectionStatus Members 134
Table 115: LoginConsumerConnectionStatus Flags 134
Table 116: LoginWarmStandbyInfo Structure Members 135
Table 117: LoginConsumerConnectionStatus Utility LoginWarmStandbyInfo Methods 135
Table 118: Login Round Trip Time Members 135
Table 119: Login Round Trip Time Flag Enumeration Values 136
Table 120: Login Round Trip Time Utility Methods 136
Table 121: LoginAttrib Members 137
Table 122: LoginAttrib Methods 139
Table 123: LoginAttribFlags 139
Table 124: LoginMsg Interfaces 139
Table 125: LoginMsg Utility Method 140
Table 126: RDM Login Encoding and Decoding Methods 140
Table 127: DirectoryRequest Structure Members 145
Table 128: DirectoryRequest Flags 146
Table 129: DirectoryRequest Utility Methods 146
Table 130: DirectoryRefresh Structure Members 147
Table 131: DirectoryRefresh Flags 147
Table 132: DirectoryUpdate Structure Members 148
Table 133: DirectoryUpdate Flags 149
Table 134: DirectoryStatus Structure Members 149
Table 135: DirectoryStatus Flags 150
Table 136: DirectoryStatus Utility Methods 150
Table 137: DirectoryClose Member 150
Table 138: DirectoryConsumerStatus Structure Members 151
Table 139: ConsumerStatusService Structure Members 151
Table 140: Service Structure Members 152
Table 141: Service Flags 152
Table 142: Service Utility Methods 153
Table 143: ServiceInfo Structure Members 153
Table 144: ServiceInfo Flags 154
Table 145: ServiceInfo Utility Methods 155
Table 146: ServiceState Structure Members 155
Table 147: ServiceState Flags 156
Table 148: ServiceState Utility Methods 156
Table 149: ServiceGroupState Structure Members 156
Table 150: ServiceGroupState Flags 157
Table 151: ServiceGroupState Utility Methods 157
Table 152: ServiceLoad Structure Members 157
Table 153: ServiceLoad Flags 158
Table 154: ServiceLoad Utility Methods 158
Table 155: ServiceData Structure Members 158
Table 156: ServiceData Flags 159
Table 157: ServiceData Utility Methods 159
Table 158: ServiceLinkInfo Structure Members 159
Table 159: ServiceLinkInfo Utility Methods 159
Table 160: ServiceLink Structure Members 160
Table 161: ServiceLink Flags 160

Table 162: `ServiceLink` Utility Methods 161
Table 163: `DirectoryMsg` Interfaces 161
Table 164: `DirectoryMsg` Utility Methods 161
Table 165: Directory Encoding and Decoding Methods 162
Table 166: `DictionaryRequest` Structure Members 168
Table 167: `DictionaryRequest` Flag 168
Table 168: `DictionaryRefresh` Structure Members 169
Table 169: `DictionaryRefreshFlags` Flags 170
Table 170: `DictionaryStatus` Structure Members 171
Table 171: `DictionaryStatus` Flags 171
Table 172: `DictionaryClose` Member 171
Table 173: `DictionaryMsg` Interfaces 171
Table 174: `DictionaryMsg` Utility Methods 172
Table 175: Dictionary Encoding and Decoding Methods 172
Table 176: `QueueRequest` Members 178
Table 177: `QueueRefresh` Members 179
Table 178: `QueueStatus` Members 179
Table 179: `QueueClose` Members 180
Table 180: `QueueData` Members 181
Table 181: Queue Data Flag 181
Table 182: `RsslRDMQueueTimeoutCodes` Queue Data Message Timeout Codes 182
Table 183: Queue Data Message Encoding Methods 182
Table 184: QueueDataExpired Structure Members 184
Table 185: `RsslRDMQueueDataUndeliverableCodes` Queue Data Message Undeliverable Codes 185
Table 186: `QueueAck` Queue Ack Members 185
Table 187: `CacheError` Class Members 193
Table 188: Method for Cache Error Handling `CacheError` Utility Method 193
Table 189: Methods for Managing Cache Instances 194
Table 190: `PayloadCacheConfigOptions` Structure Members 194
Table 191: Methods for Setting Dictionary to Cache 195
Table 192: Payload Cache `PayloadCache` Utility Methods 197
Table 193: Payload Cache Entry Management Methods 197
Table 194: Methods for Applying and Retrieving Cache Entry Data 200
Table 195: Methods for Using the Payload Cursor 200

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 Refinitiv 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 Java Edition. In addition to connecting to itself, the Enterprise Transport API can also connect to and leverage many different Refinitiv 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..

1.2 Audience

This manual provides information and examples that aid programmers using the Enterprise Transport API Java Edition Value Added Components. 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 or its Value Added Components. 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 Java programming language in a networked environment. Although Transport API Value Added Components offer alternate entry points to Transport API functionality, it is recommended that users are familiar with general Enterprise Transport API usage and interfaces.

1.3 Programming Language

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

1.4 Acronyms and Abbreviations

ACRONYM / TERM	MEANING
ADH	Refinitiv Real-Time Advanced Data Hub is the horizontally scalable service component within the Refinitiv Real-Time Distribution System providing high availability for publication and contribution messaging, subscription management with optional persistence, conflation and delay capabilities.
ADS	Refinitiv Real-Time Advanced Distribution Server is the horizontally scalable distribution component within the Refinitiv 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

Table 1: Acronyms and Abbreviations

ACRONYM / TERM	MEANING
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 Refinitiv 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 Refinitiv Wire Format when publishing or consuming content to/from Refinitiv Real-Time Distribution Systems.
GC	Garbage Collection
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol (Secure)
OMM	Open Message Model
QoS	Quality of Service
RDM	Refinitiv Domain Model
Refinitiv Real-Time Distribution System	Refinitiv Real-Time Distribution System is Refinitiv's financial market data distribution platform. It consists of the Refinitiv Real-Time Advanced Distribution Server and Refinitiv Real-Time Advanced Data Hub. Applications written to the Refinitiv 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	Refinitiv Source Sink Library
RTT	Round Trip Time, this definition is used for round trip latency monitoring feature.
RWF	Refinitiv Wire Format, a Refinitiv 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

1. Enterprise Transport API Java Edition *Refinitiv Domain Model Usage Guide*
2. *API Concepts Guide*
3. Enterprise Transport API Java Edition *Developers Guide*
4. The Refinitiv 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@refinitiv.com.
- Add your comments to the PDF using Adobe's **Comment** feature. After adding your comments, submit the entire PDF to Refinitiv by clicking **Send File** in the **File** menu. Use the ProductDocumentation@refinitiv.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:

- Java classes, methods, 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 object */
if ((retVal = filterList.decode(decIter)) >= CodecReturnCodes.SUCCESS)
{
    /* create single filter entry and reuse while decoding each entry */
    FilterEntry filterEntry = CodecFactory.createFilterEntry();
```

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:

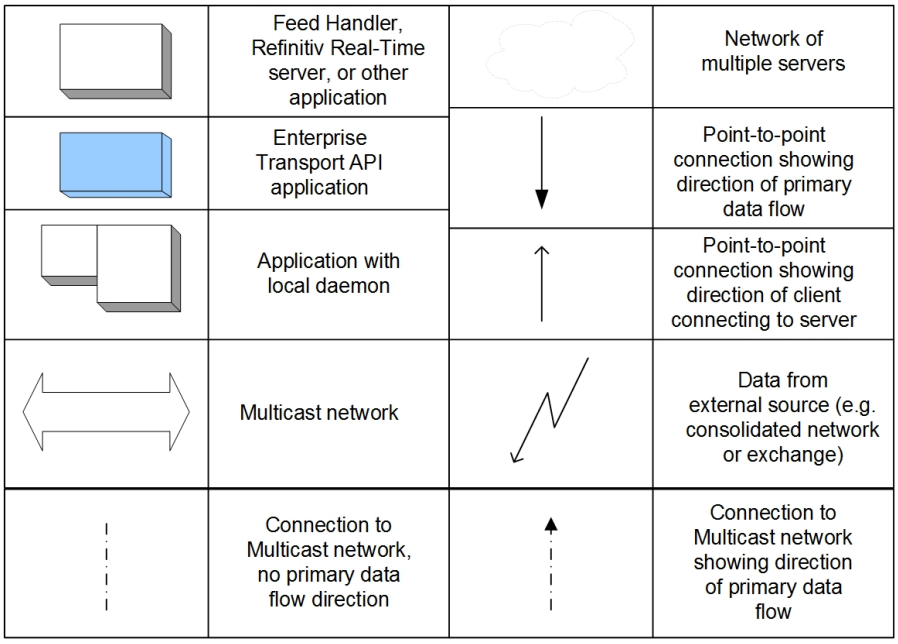


Figure 1. Network Diagram Notation

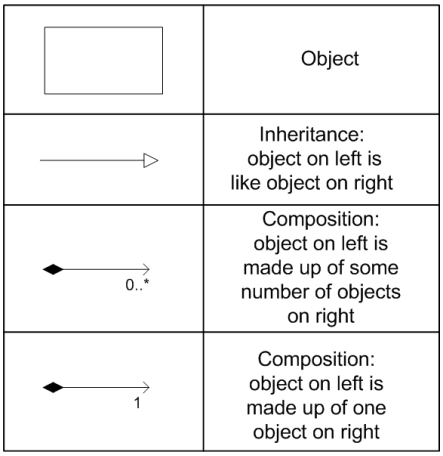


Figure 2. UML Diagram Notation

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 Refinitiv 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.

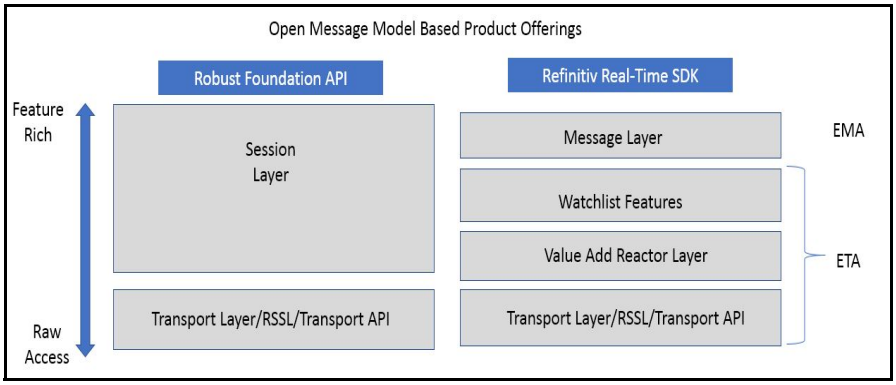


Figure 3. OMM APIs with Value Added Components

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.

2.2 What are Enterprise Transport API Value Added Components?

The Value Added Components simplify and compliment the use of the Enterprise Transport API. These components (depicted in green in Figure 4) are offered along side the Enterprise Transport API to maximize the user experience and allow for more intuitive, straight forward, and rapid creation of Enterprise Transport API applications. Applications can write directly to the Enterprise Transport API interfaces or commingle some or all Value Added Components. The choice to leverage these components is up to the application developer; you do not need to use Value Added Components to use the Enterprise Transport API. Using Enterprise Transport API Value Added Components, you can choose and customize the balance between ultra high-performance raw access and ease-of-use feature functionality. Value Added Components are written to the Enterprise Transport API interfaces and are designed to work alongside the Enterprise Transport API. Their interfaces have a similar look and feel to Enterprise Transport API interfaces to provide simple migration and consistent use between all components and the Enterprise Transport API.

All value added components provide fully supported library files ready to build into new or existing Enterprise Transport API applications. Examples and documentation are provided to show the full power and capability of the component for each layer of the Enterprise Transport API.

Some value added components provide buildable source code[1] to allow for customization and modification to suit specific user needs. This source code serves the following purposes:

- Clients may want to provide their own implementation of the component. Rather than starting from scratch, clients can modify the component to jump start their development efforts.

  NOTE: If a client customizes a component's code, the client is responsible for its support and maintenance.

- Clients might want to build a new component that has similar behaviors to an existing component. Clients can leverage the code of one component to jump start their development efforts.
- Clients may want to collaborate in troubleshooting or suggesting improvements to the component for everyone's benefit.

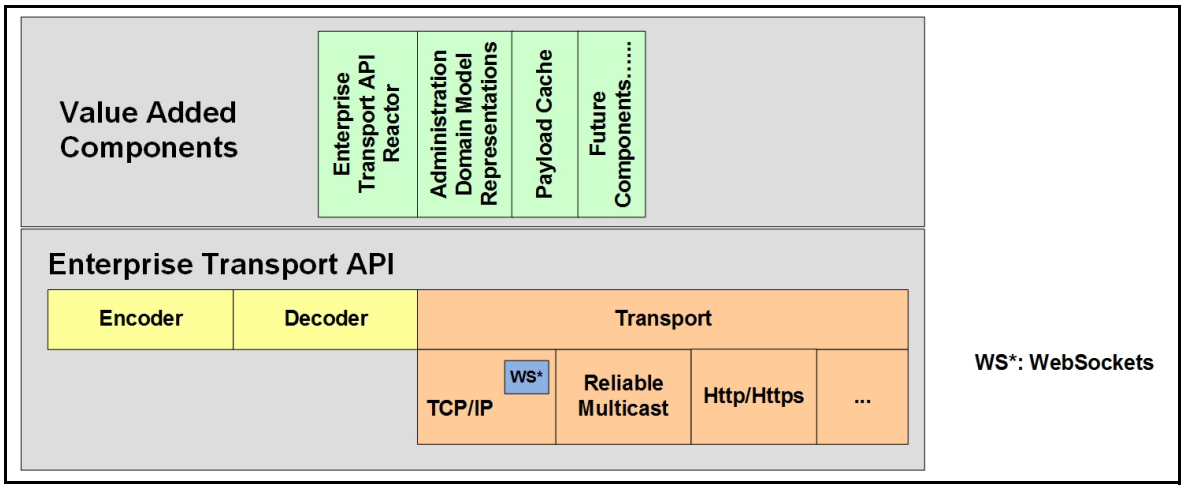


Figure 4. Enterprise Transport API Value Added Components

1. Refinitiv fully supports the use of its pre-built library files. Provided source code can help with user troubleshooting and debugging. However, the user, not Refinitiv, is responsible for supporting any modifications to the provided source.

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 Java Transport Package, and Enterprise Transport API Java 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 (**ConnectionTypes.SOCKET**).

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 APIJava Edition *Refinitiv 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 **`ReactorChannel.submit(Msg)`** 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 **`ConsumerRole.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 APIJava Edition *Refinitiv 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 class 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., Refinitiv 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 Java 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. Included is a selectable bidirectional queue used to communicate events between the Reactor and Worker threads. Other Value Added Utilities include a simple queue along with iterable and concurrent versions of it.

2.7 Payload Cache

Applications can leverage the OMM payload cache feature. Using the payload cache, an application can maintain a local store of the OMM container data it consumes, publishes, or transforms. The cache maintains the latest values of the OMM data entries: container values update to reflect the most recent refresh and update message payloads whenever the application receives them. The Enterprise Transport API retrieves data from the cache entry in the form of an encoded OMM container. The payload cache is independent of other Value Added components, and only requires the Enterprise Transport API Java Codec Package. Only library files are available for the cache component.

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 Refinitiv Real-Time Distribution System, Refinitiv Data Feed Direct, and Refinitiv 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 **Consumer** 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`** objects. 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 **`ReactorFactory.createReactor`** method. 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.

3.3 Implement Callbacks and Populate Role

Before creating the OMM consumer connection, the application needs to specify callback methods to use for all inbound content. The 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.

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

- **ReactorChannelEventCallback**, which returns information about the **ReactorChannel** and its state (e.g., connection up)
- **DefaultMsgCallback**, 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 methods. Available domain-specific callbacks include:

- **RDMLoginMsgCallback**, which processes all data for the RDM Login domain.
- **RDMDirectoryMsgCallback**, which processes all data for the RDM Source Directory domain.
- **RDMDictionaryMsgCallback**, which processes all data for the RDM Dictionary domain.

The **ConsumerRole** object 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 methods are specified on the **ConsumerRole** object or with specific information according to the application and user. The **Reactor** will use this information when starting up the **ReactorChannel**.

Detailed information about the **ConsumerRole** is in Section 6.3.1. Information about the various callback functions and their specifications are available in Section 6.6.2.

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 **ReactorChannelEventCallback**; this provides the **ReactorChannel** and the state of the current connection. At this point, the application can begin using the **ReactorChannel.dispatch** method to dispatch directly on this **ReactorChannel**, or use **Reactor.dispatchAll** 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** method is described in Section 6.4.1.1. Dispatching is described in Section 6.6.

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 Refinitiv are defined in the Enterprise Transport API Java Edition *Refinitiv Domain Model Usage Guide*. This content will be returned to the application via the **DefaultMsgCallback**.

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 PackageETA described in the Enterprise Transport API Java 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** method can be used to shutdown and clean up any **Reactor** resources.

- Closing a **ReactorChannel** is described in Section 6.4.2.
- Shutting down an **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 Javadoc 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 Java Edition *Developers Guide* provides specific ETA encoder/decoder and transport usage information.
- The Enterprise Transport API Java Edition *Refinitiv 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 a **`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`** objects. 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 **`ReactorFactory.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 APIJava 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**:

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

- **RDMLoginMsgCallback**, which processes all data for the RDM Login domain.
- **RDMDirectoryMsgCallback**, which processes all data for the RDM Source Directory domain.
- **RDMDictionaryMsgCallback**, which processes all data for the RDM Dictionary domain.

The **ProviderRole** object 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 **Server** 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 **ReactorChannel.dispatch** method to dispatch directly on this **ReactorChannel**, or continue using **Reactor.dispatchAll** 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** method, 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 **RDMLoginMsgCallback**, 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 the messages' **encode** and **decode** methods. More details and code examples are in Section 8.3.

All content requested, received, or posted is encoded and decoded using the ETA Java Codec PackageETA described in the Enterprise Transport APIJava Edition *Developers Guide*.

Information about the Login domain and expected content formatting is available in the Enterprise Transport APIJava Edition *Refinitiv 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. Refinitiv 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 Java Edition *Developers Guide*.

Source Directory 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 Java Codec PackageETA described in the Enterprise Transport API Java Edition *Developers Guide*.

Information about the Source Directory domain and expected content formatting is available in the Enterprise Transport API Java Edition *Refinitiv 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 Refinitiv **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 a Refinitiv Real-Time Advanced Data 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. A Refinitiv Real-Time Advanced Data 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 APIJava Edition *Refinitiv 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 **`RDMDictionaryMsgCallback`**, when specified on the **`ProviderRole`**.

Whether loading a dictionary from file or requesting it from a Refinitiv Real-Time Advanced Data 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 Java Codec PackageETA described in the Enterprise Transport APIJava Edition *Developers Guide*.

- Information about the Dictionary domain, dictionary utility functions, and expected content formatting is available in the Enterprise Transport APIJava Edition *Refinitiv 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 APIJava Edition *Refinitiv Domain Model Usage Guide* defines all domains provided by Refinitiv. This content will be returned to the application via the **DefaultMsgCallback**.

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 APIJava Edition *Developers Guide*. Any posted content will be returned to the application via the **DefaultMsgCallback**.

All content requested, received, or posted is encoded and decoded using the ETA Java Codec PackageETA as described in the Enterprise Transport APIJava 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 Java 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 6.4.2.
- 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 **Provider** application demonstrates one implementation of an OMM interactive provider application that uses ETA Value Added Components. The application's source code and Javadoc 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 APIJava Edition *Developers Guide* provides specific ETA encoder/decoder and transport usage information.
- The Enterprise Transport APIJava Edition *Refinitiv 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 a Refinitiv Real-Time Advanced Data Hub. After the reactor indicates that the connection is ready, an OMM non-interactive provider can publish information into the Refinitiv Real-Time Advanced Data Hub cache without needing to handle requests for the information. The Refinitiv Real-Time Advanced Data Hub and other Refinitiv 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 **`Reactor.connect`**
- 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`** objects. 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 **`ReactorFactory.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:

- **ReactorChannelEventCallback**, which returns information about the **ReactorChannel** and its state (e.g., connection up)
- **DefaultMsgCallback**, which processes all data not handled by other optional callbacks.

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

The **NIProviderRole** object 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** object 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 6.6.2.

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 **ReactorChannelEventCallback**, which provides the **ReactorChannel** and indicates the current connection state. At this point, the application can begin using the **ReactorChannel.dispatch** method to dispatch directly on this **ReactorChannel**, or use **Reactor.dispatchAll** 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 6.6.

5.5 Download the Dictionary

If connected to a supporting Refinitiv Real-Time Advanced Data 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 Refinitiv Real-Time Advanced Data 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 Java Edition *Refinitiv Domain Model Usage Guide*.

5.6 Provide Content

After the **ReactorChannel** is established, it can begin pushing content to the Refinitiv Real-Time Advanced Data 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 Java Codec PackageETA described in the Enterprise Transport API Java 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 6.4.2.
- 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 **NIProvider** application demonstrates one implementation of an OMM non-interactive provider application that uses ETA Value Added Components. The application's source code and Javadoc have additional information about the specific implementation and behaviors.
- Chapter 6 provides a detailed look at the ETA Reactor.
- Chapter 8 provides more information about Administration Domain Model Representations.
- The Enterprise Transport API Java Edition *Developers Guide* provides specific ETA encoder/decoder and transport usage information.
- The Enterprise Transport API Java Edition *Refinitiv 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.

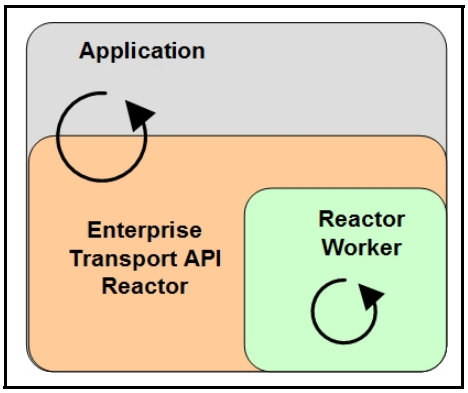


Figure 5. ETA Reactor Thread Model

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. It also attempts to recover any connections that are lost.

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`** object 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:

CLASS MEMBER	DESCRIPTION
code	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 20.
error	Returns an **`Error`** object (i.e., the underlying error information from the Enterprise Transport API). **`Error`** includes a pointer to the **`Channel`** 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 APIJava 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 object.

Table 3: `ReactorErrorInfo` Members

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
SUCCESS	Indicates a success code. Used to inform the user of success and provide additional information.
FAILURE	A general failure has occurred. The **`ReactorErrorInfo`** code contains more information about the specific error.
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.
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 **`ReactorChannel.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.

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.

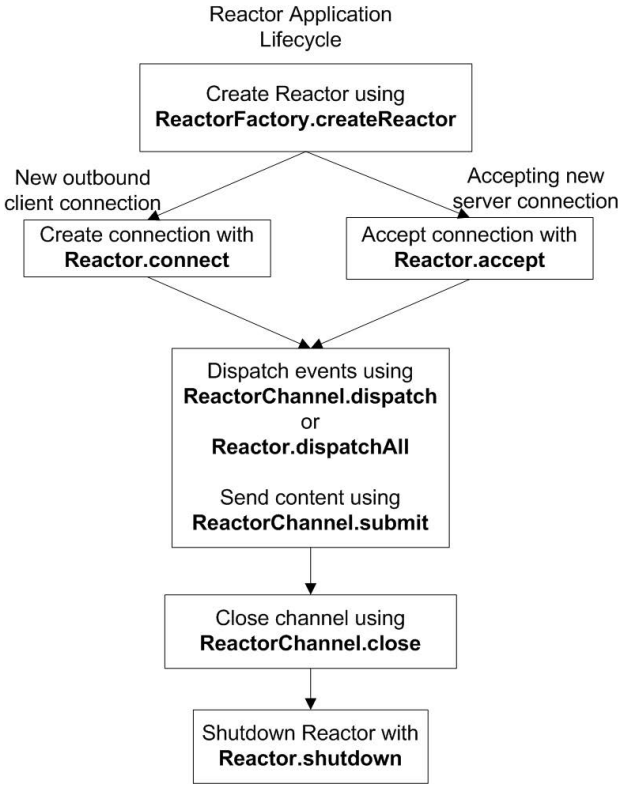


Figure 6. Enterprise Transport API Reactor Application Lifecycle

6.2 Reactor Use

This section describes use of **`Reactor`**. The **`Reactor`** manages **`ReactorChannel`**s (described in Section 6.3). An understanding of both constructs is necessary for application writers. Reactor uses the class members described in Table 5.

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

CLASS MEMBER	DESCRIPTION
reactorChannel	The reactor's internal channel used for **`Reactor`** specific events to communicate with the worker thread. The application must register this **`ReactorChannel`**'s selectable channel with its selector if using select notification. All **`ReactorChannel`** data event notification occurs on the **`ReactorChannel`**'s specific **`selectableChannel`**, as detailed in Section 6.3.
userSpecObj	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 an **`Reactor`** or coupling this **`Reactor`** with other user-defined information.

Table 5: `Reactor` Class Members

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
ReactorFactory.createReactor	Creates a **Reactor** instance, including all necessary internal memory and threads. After creating the **Reactor**, **ReactorChannel**s can be associated, as described in Section 6.3. Options are passed in via the **ReactorOptions**, as defined in Section 6.2.1.2.

Table 6: Reactor Creation Method

6.2.1.2 ReactorOptions Class Members

CLASS MEMBER	DESCRIPTION
debuggerOptions	ReactorDebuggerOptions instance that can be used to configure debugging functionality for the Reactor. By default, debugger options set no debugging level. For more details, see Section 6.2.1.4.
enableXmlTracing	Enables XML tracing for the **Reactor**. The **Reactor** prints the XML representation of all OMM messages when enabled.
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.
restRequestTimeOut	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 **rsslReactorConnect()** 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.
serviceDiscoveryURL	Specifies the URL of Refinitiv Data 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 Refinitiv Real-Time solution/offering.
tokenReissueRatio	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 Refinitiv 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**.

Table 7: ReactorOptions Class Members

CLASS MEMBER	DESCRIPTION
tokenServiceURL	Specifies the URL of Refinitiv Data Platform that the RTSDK API uses to obtain an authentication token.
userSpecObj	An object that can be set by the application. This value is preserved and stored in the **userSpecObj** of the **Reactor** returned from **ReactorFactory.createReactor**. This information can be useful for identifying a specific instance of a reactor or coupling this **Reactor** with other user-created information.

Table 7: ReactorOptions Class Members (Continued)

6.2.1.3 ReactorOptions Utility Method

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

METHOD NAME	DESCRIPTION
clear	Clears the **ReactorOptions** class. Useful for object reuse.

Table 8: ReactorOptions Utility Method

6.2.1.4 ReactorDebuggerOptions Interface

The **ReactorDebuggerOptions** interface provides methods that set debugging levels for the Reactor and the output stream for the debugging information.

The debugging levels available for the Reactor are provided in the **ReactorDebuggerLevels** class.

DEBUGGING LEVEL	DESCRIPTION
LEVEL_NONE	No debugging information is logged.
LEVEL_CONNECTION	Provides debugging points for connection events such as **CHANNEL_UP**, **CHANNEL_DOWN**, etc.
LEVEL_EVENTQUEUE	Allows obtaining information about the number of events in the Reactor queue and the number of events to be dispatched.
LEVEL_TUNNELSTREAM	Provides debugging points for **TunnelStream** events.

Table 9: Reactor Debugging Levels

By default, the output is directed to **ByteArrayOutputStream** with a default capacity of 64 KB.

If this limit is exceeded, debugging messages are no longer written into the buffer until the user requests the already logged information. In this case the buffer is cleared and more information can be logged.

For additional information, see Section 6.2.1.5 and Section 6.11.4.

6.2.1.5 ReactorDebuggerOptions Methods

METHOD	DESCRIPTION
setDebuggingLevels(int val)	Sets the levels to be debugged in batch.
enableLevel(int val)	Specifies the debugging level that should be enabled.

METHOD	DESCRIPTION
disableLevel(int val)	Specifies the debugging level that should be disabled.
debugEnabled()	Determines whether at least one connection level is enabled.
outputStream(OutputStream stream)	Sets the stream into which the information is written.
capacity(int capacity)	Specifies the capacity that should be applied to the output stream if applicable.
clear()	Clears current **ReactorDebuggerOptions** instance.

6.2.2 Destroying 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 destruction functionality in more detail.

6.2.2.1 Reactor Destruction

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

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

Table 10: Reactor Destruction Method

6.2.2.2 Reactor Creation and Destruction Example

```
ReactorOptions reactorCreateOptions = ReactorFactory.createReactorOptions();

reactorCreateOptions.clear();

/* Create the Reactor. */
reactor = ReactorFactory.createReactor(reactorCreateOptions, errorInfo);

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

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

Code Example 1: Reactor Creation and Destruction Example

6.3 Reactor Channel Use

The **ReactorChannel** object is used to represent a connection that can send or receive information across a network. This object is used to represent a connection, regardless of whether it is an outbound connection or a connection accepted by a listening socket via a **Server**. 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 an **Reactor**.

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

CLASS MEMBER	DESCRIPTION
hostname	Provides the name of the host to which a consumer or NIP application connects.
majorVersion	When a **ReactorChannel** is up (**ReactorChannelEventTypes.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 APIJava Edition *Developers Guide*.
minorVersion	When a **ReactorChannel** is up (**ReactorChannelEventTypes.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 APIJava Edition *Developers Guide*.
oldSelectableChannel	It is possible for a selectable channel 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 **ReactorChannelEventTypes.FD_CHANGE**. The previous selectable channel is stored in **oldSelectableChannel** so the application can properly unregister and then register the new **selectableChannel** with their selector I/O.
port	Provides the server port number to which the consumer or NIP application connects.
protocolType	When a **ReactorChannel** is up (**ReactorChannelEventTypes.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 APIJava Edition *Developers Guide*.
channel	The underlying **Channel** object, as defined in the Enterprise Transport APIJava 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 **Channel** directly with any Transport functionality.
server	The underlying **Server** object, as defined in the Enterprise Transport APIJava 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.
selectableChannel	Represents a selectable channel that can be used in select notification to alert users when dispatch is required on a specific **ReactorChannel**. Used to register with a selector.

Table 11: ReactorChannel Class Members

CLASS MEMBER	DESCRIPTION
state	The state of the **ReactorChannel**.
userSpecObj	An object that can be set by the user of the **Channel**. This value can be set via the **ReactorConnectOptions**s and **ReactorAcceptOption**s. This information can be useful for coupling this **ReactorChannel** with other user-created information.

Table 11: ReactorChannel Class Members (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, the **ReactorRole**.

6.3.1.1 ReactorRole Class

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

CLASS MEMBER	DESCRIPTION
channelEventCallback	This **ReactorChannel**'s user-defined callback method to handle all **ReactorChannel** specific events, like **ReactorChannelEventTypes.CHANNEL_UP** or **ReactorChannelEventTypes.CHANNEL_DOWN**. This callback method is required for all role types. This callback is defined in more detail in Section 6.6.2.
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 6.6.2.
type	The role type enumeration value, as defined in Section 6.3.1.2.

Table 12: ReactorRole Class 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 a interactive provider.

Table 13: ReactorRoleTypes 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 APIJava Edition *Refinitiv 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** object as follows:

CLASS MEMBER	DESCRIPTION
clientId	This member has been deprecated. Instead, use the **reactorOAuthCredential** member to set all Refinitiv Data Platform token service request credentials.
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 6.6.2. • 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 6.6.2. • 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 6.6.2. • 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, a **rdmDirectoryRequest** 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 Refinitiv Data Platform token service request. This is required when connecting to a Refinitiv Real-Time Advanced Distribution Server in the cloud. See Section 6.9.2.1.
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.
watchlistOptions	Configurable options for the consumer watchlist. Options are described in more detail in Section 6.3.2.3.

Table 14: ConsumerRole Class Members

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 15: 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 16: OMM Consumer Role Watchlist Options

6.3.2.4 OMM Consumer Role Utility Method

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

METHOD NAME	DESCRIPTION
clear	Clears the **ConsumerRole** object. Useful for object reuse.

Table 17: ConsumerRole Utility Method

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 APIJava Edition *Refinitiv 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** class, as follows:

CLASS 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 6.6.2. • 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 6.6.2. • 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 6.6.2. • Login messages are described in Section 8.3.
listenerCallback	This **ReactorChannel**'s user-defined callback for accepting or rejecting tunnel streams. For further details on this callback, refer to Section 6.8.6.

Table 18: ProviderRole Class Members

6.3.3.2 OMM Provider Role Utility Method

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

METHOD NAME	DESCRIPTION
clear	Clears the **ProviderRole** object. Useful for object reuse.

Table 19: ProviderRole Utility Method

6.3.4 Reactor Channel Role: OMM Non-Interactive Provider

When a **ReactorChannel** acts as an OMM non-interactive provider application, it connects to a Refinitiv Real-Time Advanced Data 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 APIJava Edition *Refinitiv 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** class, as follows:

MEMBER	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.
rdmDirectoryRefresh	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 **rdmDirectoryRefresh** is required. • If this parameter is left empty, a directory request is not 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 6.6.2.

Table 20: NIProviderRole Class Members

6.3.4.2 OMM Non-Interactive Provider Role Utility Method

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

FUNCTION NAME	DESCRIPTION
clear	Clears the **NIProviderRole** object. Useful for object reuse.

Table 21: NIProviderRole Utility Method

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 a **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 6.6.2, 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 **ReactorConnectOptions** object. The application can specify that **Reactor** reconnect the **ReactorChannel** to the same host, or to one from among multiple hosts.

Consumer applications can combine the reactor connect feature with the watchlist feature to enable recovery of item streams across connections. For more information on the watchlist feature, refer to Section 2.4.

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 **Transport.connect** method, as described in the Enterprise Transport APIJava Edition *Developers Guide*. Connection options are passed in via the **ReactorConnectOptions**, 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 22: Reactor.connect Method

6.4.1.2 ReactorConnectOptions Class Members

CLASS MEMBER	DESCRIPTION
connectionList	Specifies **ReactorConnectInfo** as defined in Section 6.4.1.3. When used with **reconnectAttemptLimit**, the **Reactor** attempts to connect to each host in the list with each reconnection attempt.
reconnectAttemptLimit	The maximum number of times the **Reactor** attempts to reconnect a channel when it fails. If set to **-1**, there is no limit.
reconnectMinDelay	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 **reconnectMinDelay** to **reconnectMaxDelay**.

Table 23: ReactorConnectOptions Class Members

CLASS MEMBER	DESCRIPTION
reconnectMaxDelay	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 **reconnectMinDelay** to **reconnectMaxDelay**.
statisticFlags	Specifies **ReactorChannelStatisticFlags** which set the type of statistics reporting (if any) to perform on the **Reactor** channel. **ReactorChannelStatisticFlags** uses the following enums: • **RSSL_RC_ST_NONE** (or **0x0000**): Turns off statistics reporting. • **RSSL_RC_ST_READ** (or **0x0001**): Turns on statistics reporting for the number of bytes read and the number of uncompressed bytes read. • **RSSL_RC_ST_WRITE** (or **0x0002**):Turns on statistics reporting for the number of bytes written and uncompressed bytes written. • **RSSL_RC_ST_PING** (or **0x0004**): Turns on statistics reporting for the number of pings received and the number of pings sent.

Table 23: ReactorConnectOptions Class Members (Continued)

6.4.1.3 ReactorConnectInfo Class Members

CLASS MEMBER	DESCRIPTION
connectOptions	Specifies information (**connectOptions**) 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 **Transport.connect** method. This is described in more detail in the Enterprise Transport APIJava 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**.
initTimeout	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]**).
loginReqIndex	This specifies the login message to be used for this connection, contained in the pointer array of **RsslReactorOMMConsumerRole.pLoginRequestList**. If **RsslReactorOMMConsumerRole.pLoginRequestList** is specified, then this connection will use that login message and login credential callback when it connects. This must be less than **RsslReactorOMMConsumerRole.pLoginRequestCount**. This is a 0 indexed array. For details, see Section 6.5.
oAuthCredentialIndex	This specifies the oAuth credential to be used for this connection, contained in the pointer array of **RsslReactorOMMConsumerRole.pOAuthCredentialList**, if **enableSessionManagement** is turned on and **RsslReactorOMMConsumerRole.oAuthCredentialCount** has been set. This must be less than the number set in **RsslReactorOMMConsumerRole.oAuthCredentialCount**. This is a 0 indexed array. For details, see Section 6.5.

Table 24: ReactorConnectInfo Class Members

CLASS MEMBER	DESCRIPTION
serviceDiscoveryRetryCount	The number of times the **Reactor** attempts to reconnect a channel before forcing the API to retry service discovery. See the description of **ReactorOptions.serviceDiscoveryURL** in Section 6.2.1.2. This functionality works when the user sets **enableSessionManagement** to true, but is ignored when the user sets the values of **UnifiedNetworkInfo.address** and **UnifiedNetworkInfo.serviceName** in **ConnectOptions** directly. For details, refer to the Enterprise Transport API Java Edition *Developers Guide*. The default value is 3. After each 3 attempts to reconnect a channel, the **Reactor** will force to retry service discovery. The **Reactor** will not retry to get an endpoint from the service discovery when the value is **0**.
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.

Table 24: ReactorConnectInfo Class Members (Continued)

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** object. Useful for object reuse.

Table 25: ReactorConnectOptions Utility Method

6.4.1.5 Reactor.connect Example

```
ReactorConnectOptions connectOpts = ReactorFactory.createReactorConnectOptions();
ConsumerRole consumerRole = ReactorFactory.createConsumerRole();

/* Configure connection options.*/
connectOpts.clear();
connectOpts.connectOptions.connectionList().get(0).connectOptions().unifiedNetworkInfo().address
        ("localhost");
connectOpts.connectOptions.connectionList().get(0).connectOptions().unifiedNetworkInfo().serviceName
        ("14002");

/* Configure a role for this connection as an OMM Consumer. */
consumerRole.clear();

/* 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, 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 **Server**. The **ReactorChannel** will be returned to the application via a callback, as described in Section 6.6.2, at which point it can begin dispatching on the channel.

METHOD NAME	DESCRIPTION
Reactor.accept	Creates a **ReactorChannel** by accepting it from a **Server**. This establishes a connection in a manner similar to the **Server.accept** function, as described in the Enterprise Transport APIJava 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 26: Reactor.accept Method

6.4.1.7 ReactorAcceptOptions Class Members

CLASS MEMBER	DESCRIPTION
acceptOptions	The **AcceptOptions** associated with the underlying **Server.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 APIJava Edition *Developers Guide*.
initTimeout	The amount of time (in seconds) to wait for the successful connection establishment of an **ReactorChannel**. If a timeout occurs, an event is dispatched to the application to indicate that the **ReactorChannel** is down.

Table 27: 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** object. Useful for object reuse.

Table 28: RsslReactorAcceptOptions Utility Method

6.4.1.9 Reactor.accept Example

```
ReactorAcceptOptions reactorAcceptOpts = ReactorFactory.createReactorAcceptOptions();
ProviderRole providerRole = ReactorFactory.createProviderRole();

/* 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, errorInfo)
```

Code Example 3: `Reactor.accept` Example

6.4.2 Removing Reactor Channels

6.4.2.1 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 passed in **`Reactor`** instance and cleans up associated resources. This additionally invokes the **`Channel.close`** method, as described in the Enterprise Transport APIJava Edition *Developers Guide*, to clean up any resources associated with the underlying **`Channel`**. This method can be called from either outside or within a callback.

Table 29: `ReactorChannel.close` Function

6.4.2.2 ReactorChannel.close Example

```
ReactorErrorInfo errorInfo = ReactorFactory.createReactorErrorInfo();
/* Can be used inside or outside of a callback */
ret = reactorChannel.close(errorInfo);
```

Code Example 4: `ReactorChannel.close` Example

6.5 Reporting on Channel Statistics

You can use the **ReactorRetrieveChannelStatistic()** method to report on channel statistics. To use this method, you must first activate channel statistics reporting in **RsslReactorConnectOptions** by setting the **statisticFlags** member (for details on this member and the types of statistics on which you can report, refer to Section 6.4.1.2).

To get statistics, create a **ReactorChannelStatistic** structure and pass it in with the method. The Enterprise Transport API responds with the data for which the **statisticFlags** expressed interest.

6.6 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 6.6.2.

An application can choose to dispatch across all associated **ReactorChannel**s (via **Reactor.dispatchAll**) or to dispatch on a particular **ReactorChannel** (via **ReactorChannel.dispatch**). If dispatching on a single **ReactorChannel**, only this channel's data is processed and returned via the channel's callback. 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.

Typically, an application registers both the **Reactor**'s internal **ReactorChannel**'s selectableChannel and each **ReactorChannel**'s selectableChannel with a select notifier. The select notifier can help inform the application when data is available on particular **ReactorChannel**s or when channel information is available from the **Reactor** (via its internal **ReactorChannel**). An application can also forgo the use of notifiers and instead periodically call the dispatch function to process data as described in Section 6.6.1.

6.6.1 Reactor Dispatch Methods

NOTE: Applications should not call **Reactor.shutdown**, **Reactor.dispatchAll**, or **ReactorChannel.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** objects, a deep copy of the object should be made if the application wishes to preserve it. To copy a **Msg**, refer to the **Msg.copy** method in the Enterprise Transport APIJava 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
Reactor.dispatchAll	This method processes events and messages across the provided **Reactor** and all of its associated **ReactorChannel**s. When channel information or data is available for a **ReactorChannel**, the channel's user-defined callback method is invoked. The application can control the maximum number of messages dispatched with a single call to **Reactor.dispatchAll**. This can be controlled through passed-in **ReactorDispatchOptions**, as described in Section 6.6.1.1.
ReactorChannel.dispatch	This method processes a specific channel's events and messages from the **Reactor**. When channel information or data is available for a **ReactorChannel**, the channel's user-defined callback method is invoked. The application can control the maximum number of messages dispatched with a single call to **ReactorChannel.dispatch**. This can be controlled through passed-in **ReactorDispatchOptions** (for details refer to Section 6.6.1.1).

Table 30: Reactor Dispatch Methods

6.6.1.1 Reactor Dispatch Options

An application can use **ReactorDispatchOptions** to control various aspects of the call to **Reactor.dispatchAll** and **ReactorChannel.dispatch**.

CLASS MEMBER	DESCRIPTION
maxMessages	Controls the maximum number of events or messages processed in this call. If this is larger than the number of available messages, **Reactor.dispatchAll** or **ReactorChannel.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.dispatchAll** or **ReactorChannel.dispatch**.
readArgs	The **ReadArgs** from the underlying **Channel.read** call.

Table 31: ReactorDispatchOptions Class Members

6.6.1.2 ReactorDispatchOptions Utility Function

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

METHOD NAME	DESCRIPTION
clear	Clears the **ReactorDispatchOptions** object. Useful for object reuse.

Table 32: ReactorDispatchOptions Utility Method

6.6.1.3 ReactorChannel.dispatch Example

```
ReactorDispatchOptions dispatchOpts = ReactorFactory.createReactorDispatchOptions();

/* Set dispatching options. */
dispatchOpts.clear();
dispatchOpts.maxMessages(200);

/* Call ReactorChannel.dispatch(). It will keep dispatching events until there is nothing to read or
 * maxMessages is reached. */
ret = reactorChannel.dispatch(dispatchOpts, errorInfo);
```

Code Example 5: ReactorChannel.dispatch Example

6.6.2 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., **RDMLoginMsgCallback**). This will cause the **Reactor** to invoke the **DefaultMsgCallback** for this message upon the domain-specific callbacks return.

Table 33: ReactorCallbackReturnCodes 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 MEMBER	DESCRIPTION
reactorChannel	The **ReactorChannel** on which the event occurred.
errorInfo	The **ReactorErrorInfo** associated with this event.

Table 34: ReactorEvent Class Members

6.6.3 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:

```
reactorChannelEventCallback(ReactorChannelEvent event)
```

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

6.6.3.1 Reactor Channel Event

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

CLASS MEMBER	DESCRIPTION
eventType	The type of event that has occurred on the **ReactorChannel**. For a list of enumeration values, refer to Section 6.6.3.2.

Table 35: ReactorChannelEvent Class Member

6.6.3.2 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**s 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, such as **TransportBuffer**s, and unregister its **selectableChannel**, if valid, from any select notifiers.
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 selectable channel change occurred on the **ReactorChannel**. If the application is using a select notification mechanism, it should unregister the **oldSelectableChannel** and register the **selectableChannel**, 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. **ReactorErrorInfo** contains more detailed information about what occurred.

Table 36: ReactorChannelEventType Enumeration Values

6.6.3.3 Reactor Channel Event Utility Methods

METHOD NAME	DESCRIPTION
clear	Clears a **ReactorChannelEvent** object.

Table 37: ReactorChannelEvent Utility Methods

6.6.3.4 Reactor Channel Event Callback Example

```
public int reactorChannelEventCallback(ReactorChannelEvent event)
{
    switch(event.eventType())
    {
        case ReactorChannelEventTypes.CHANNEL_UP:
            // register selector with channel event's reactorChannel
            event.reactorChannel().selectableChannel().register(selector, SelectionKey.OP_READ,
                    event.reactorChannel());
        break;

        case ReactorChannelEventTypes.CHANNEL_DOWN:
            // close ReactorChannel
            if (event.reactorChannel() != null)
            {
                event.reactorChannel().close(errorInfo);
            }
        break;

        case ReactorChannelEventTypes.CHANNEL_READY:
            /* Channel has exchanged its initial messages (if any were provided on the role object)
            * and is ready for use. */
            sendItemRequests(reactorChannel);
        break;

        case ReactorChannelEventTypes.FD_CHANGE:
            /* The descriptor representing this channel has changed. Normally the application only
            * needs to update its notification mechanism in response to this event. */
            // cancel old reactorChannel select
                SelectionKey key = event.reactorChannel().oldSelectableChannel().keyFor(selector);
                key.cancel();
                // register selector with channel event's new reactorChannel
            event.reactorChannel().selectableChannel().register(selector, SelectionKey.OP_READ,
                    event.reactorChannel());
        break;

    }
    return ReactorCallbackReturnCodes.SUCCESS;
}
```

Code Example 6: Reactor Channel Event Callback Example

6.6.4 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 **ReactorCallbackReturnCodes.RAISE** value. This interface has the following callback method:

```
public int defaultMsgCallback(ReactorMsgEvent event)
```

When invoked, this returns a **ReactorMsgEvent** object, containing more information about the event information.

6.6.4.1 Reactor Message Event

The **ReactorMsgEvent** is returned to the application via the **DefaultMsgCallback**. This is also the base class of the **RDMDictionaryMsgEvent**, **RDMDirectoryMsgEvent**, and **RDMLoginMsgEvent** classes.

CLASS MEMBER	DESCRIPTION
transportBuffer	A **TransportBuffer** containing the raw, undecoded message that was read and processed by the callback.
	NOTE: When the consumer watchlist is enabled, a **TransportBuffer** is not provided, because the message might not match this buffer, or the message might be internally generated.
msg	A **Msg** object populated with message content by calling **Msg.decode**. If not present, an error was encountered while processing the information.
	NOTE: When the consumer watchlist is enabled, **msg** 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 38: ReactorMsgEvent Class Members

6.6.4.2 Reactor Message Event Utility Methods

METHOD NAME	DESCRIPTION
clear	Clears a **ReactorMsgEvent** object.

Table 39: ReactorMsgEvent Utility Method

6.6.4.3 Reactor Message Event Callback Example

```
public int defaultMsgCallback(ReactorMsgEvent event)
{
    Msg msg = event.msg();

    /* Received a Msg --- or, if the decode failed, an error. */
    /* The Msg will have already been passed through Msg.decode. Only the payload requires
    additional decoding. */
    if (msg != null)
        processMsg(msg);
    else
        System.out.printf("defaultMsgCallback Error: %s(%s)\n", event.errorInfo().error().text(),
                event.errorInfo().location());
}
```

Code Example 7: Reactor Message Event Callback Example

6.6.5 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 int rdmLoginMsgCallback(RDMLoginMsgEvent event)
```

When invoked, this will return the **RDMLoginMsgEvent** object, containing more information about the event information.

6.6.5.1 Reactor RDM Login Message Event

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

CLASS MEMBER	DESCRIPTION
rdmLoginMsg	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 40: RDMLoginMsgEvent Class Member

6.6.5.2 Reactor RDM Login Message Event Utility Method

METHOD NAME	DESCRIPTION
clear	Clears a **RDMLoginMsgEvent** object.

Table 41: RDMLoginMsgEvent Utility Method

6.6.5.3 Reactor RDM Login Message Event Callback Example

```
public int rdmLoginMsgCallback(RDMLoginMsgEvent event)
{
    LoginMsg loginMsg = event.rdmLoginMsg();

    /* Received an RDM LoginMsg --- or, if the decode failed, an error. */
    /* The login message will already be fully decoded. */
    if (loginMsg != null)
    {
        switch(loginMsg.rdmMsgType())
        {
            case REFRESH:
                LoginRefresh refresh = (LoginRefresh)loginMsg;
            break;
            case STATUS:
                LoginStatus status = (LoginStatus)loginMsg;
            break;
            default:
                System.out.println("Received unhandled login message.");
            break;
        }
    }
    else
        System.out.printf("rdmLoginMsgCallback Error: %s(%s)\n",
                event.errorInfo().error().text(),
                event.errorInfo().location());
}
```

Code Example 8: Reactor RDM Login Message Event Callback Example

6.6.6 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 int rdmDirectoryMsgCallback(RDMDirectoryMsgEvent event)
```

When invoked, this will return the **`RDMDirectoryMsgEvent`** object, containing more information about the event information.

6.6.6.1 Reactor RDM Directory Message Event

The **RDMDirectoryMsgEvent** is returned to the application via the **RDMDirectoryMsgCallback**.

CLASS MEMBER	DESCRIPTION
rdmDirectoryMsg	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 42: RDMDirectoryMsgEvent Class Member

6.6.6.2 Reactor RDM Directory Message Event Utility Method

METHOD NAME	DESCRIPTION
clear	Clears an **RDMDirectoryMsgEvent** object.

Table 43: RDMDirectoryMsgEvent Utility Method

6.6.6.3 Reactor RDM Directory Message Event Callback Example

```
public int rdmDirectoryMsgCallback(RDMDirectoryMsgEvent event)
{
    DirectoryMsg directoryMsg = event.rdmDirectoryMsg();

    /* Received an RDM DirectoryMsg --- or, if the decode failed, an error. */
    /* The directory message will already be fully decoded. */
    if (directoryMsg != null)
    {
        switch(directoryMsg.rdmMsgType())
        {
            case REFRESH:
                DirectoryRefresh refresh = (DirectoryRefresh)directoryMsg;
            break;
            case UPDATE:
                DirectoryUpdate update = (DirectoryUpdate)directoryMsg;
            break;
            case STATUS:
                DirectoryStatus status = (DirectoryStatus)directoryMsg;
            break;
            default:
                System.out.println("Received unhandled directory message.");
        }
    }
    else
        System.out.printf("rdmDirectoryMsgCallback Error: %s(%s)\n",
                event.errorInfo().error().text(),
                event.errorInfo().location());
}
```

Code Example 9: Reactor RDM Directory Message Event Callback Example

6.6.7 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 int rdmDictionaryMsgCallback(RDMDictionaryMsgEvent event)
```

When invoked, this will return the **RDMDictionaryMsgEvent** object, containing more information about the event information.

6.6.7.1 Reactor RDM Dictionary Message Event

The **RDMDictionaryMsgEvent** is returned to the application via the **RDMDictionaryMsgCallback**.

CLASS 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 44: RDMDictionaryMsgEvent Class Member

6.6.7.2 Reactor RDM Dictionary Message Event Utility Method

METHOD NAME	DESCRIPTION
clear	Clears an **RDMDictionaryMsgEvent** object.

Table 45: RDMDictionaryMsgEvent Utility Method

6.6.7.3 Reactor RDM Dictionary Message Event Callback Example

```
public int rdmDictionaryMsgCallback(RDMDictionaryMsgEvent event)
{
    DictionaryMsg dictionaryMsg = event.rdmDictionaryMsg();

    /* Received an RDM DictionaryMsg --- or, if the decode failed, an error. */
    if (dictionaryMsg != null)
    {
        switch(dictionaryMsg.rdmMsgType())
        {
            case REFRESH:
                DictionaryRefresh refresh = (DictionaryRefresh)dictionaryMsg;
            break;
            case STATUS:
                DictionaryStatus status = (DictionaryStatus)dictionaryMsg;
            break;
            default:
                System.out.println("Received unhandled dictionary message.");
        }
```

```
    }
    else
        System.out.printf("rdmDictionaryMsgCallback Error: %s(%s)\n",
                event.errorInfo().error().text(),
                event.errorInfo().location());
}
```

Code Example 10: Reactor RDM Dictionary Message Event Callback Example

6.7 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 offers two methods for writing content: **ReactorChannel.submit(Msg...)** and **ReactorChannel.submit(TransportBuffer...)**. When writing applications to the Reactor, consider which is most appropriate for your needs:

ReactorChannel.submit(Msg...)

- Takes an **Msg** object as part of its options; does not require retrieval of an **TransportBuffer** from the channel.
- Must be used when the consumer watchlist is enabled.

ReactorChannel.submit(TransportBuffer...)

- Takes an **TransportBuffer** which the application retrieves from the channel.
- More efficient: the application encodes directly into the buffer, and can use buffer packing.
- Cannot be used when the consumer watchlist is enabled.

6.7.1 Writing Data using ReactorChannel.submit(Msg…)

ReactorChannel.submit(Msg...) provides a simple interface for writing **Msg**s. To send a message, the application populates an **Msg** object, sets any other desired options on a **ReactorSubmitOptions** object, and calls **ReactorChannel.submit(Msg...)** with the object.

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).

6.7.1.1 ReactorChannel.submit(Msg…) Method

METHOD NAME	DESCRIPTION
ReactorChannel.submit	Encodes and submits a **Msg** to the Reactor. This method expects a properly populated **Msg**.

Table 46: ReactorChannel.submit(Msg...) Method

6.7.1.2 Reactor Submit Options

An application can use **ReactorSubmitOptions** to control various aspects of the call to **ReactorChannel.submit**.

CLASS MEMBER	DESCRIPTION
serviceName	The application can use this instead of the **serviceId** member specified on the **MsgKey** of a **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 APIJava 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 APIJava 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 47: ReactorSubmitOptions Class Members

6.7.1.3 ReactorChannel.submit(Msg...) Return Codes

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

RETURN CODE	DESCRIPTION
ReactorReturnCodes.SUCCESS	Indicates that the **ReactorChannel.submit(Msg...)** method has succeeded.
ReactorReturnCodes.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.
ReactorReturnCodes.WRITE_CALL_AGAIN	Indicates that buffer created for the **Msg** is being fragmented and needs to be called again with the same **Msg**. This indicates that underlying write was unable to send all fragments with the current call and must continue fragmenting.
ReactorReturnCodes.FAILURE	Indicates that a general failure has occurred and the message was not submitted. The **ReactorErrorInfo** object passed to the method will contain more details.

Table 48: ReactorChannel.submit(Msg...) Return Codes

6.7.1.4 ReactorRequestMsgOptions

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

CLASS MEMBER	DESCRIPTION
userSpecObj	A user-specified object that will be associated with the stream. This object will be provided in responses to this stream via the **WatchlistStreamInfo** provided with each message event.

Table 49: ReactorRequestMsgOptions Class Members

6.7.1.5 ReactorSubmitOptions Utility Method

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

METHOD NAME	DESCRIPTION
clear	Clears the **ReactorSubmitOptions** object. Useful for object reuse.

Table 50: ReactorSubmitOptions Utility Method

6.7.1.6 ReactorChannel.submit(Msg...) Example

The following example shows typical use of **`ReactorChannel.submit(Msg...)`**.

```
RequestMsg requestMsg = (RequestMsg)CodecFactory.createMsg();
ReactorSubmitOptions opts = ReactorFactory.createReactorSubmitOptions();
ReactorErrorInfo errorInfo = ReactorFactory.createReactorErrorInfo();
int ret;

requestMsg.clear();
requestMsg.msgClass(MsgClasses.REQUEST);
requestMsg.streamId(2);
requestMsg.domainType(DomainTypes.MARKET_PRICE);
requestMsg.containerType(DataTypes.NO_DATA);
requestMsg.applyStreaming();
requestMsg.applyHasQos();
requestMsg.qos().timeliness(QosTimeliness.REALTIME);
requestMsg.qos().rate(QosRates.TICK_BY_TICK);
requestMsg.msgKey().applyHasName();
requestMsg.msgKey().applyHasServiceId();
requestMsg.msgKey().name().data("TRI.N");
requestMsg.msgKey().serviceId(1);

ret = reactorChannel.submit(requestMsg, opts, errorInfo);
```

Code Example 11: `ReactorChannel.submit(Msg...)` Example

6.7.2 Writing Data Using ReactorChannel.submit(TransportBuffer...)

The **`ReactorChannel.submit(TransportBuffer...)`** 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(Msg...)`**, 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.

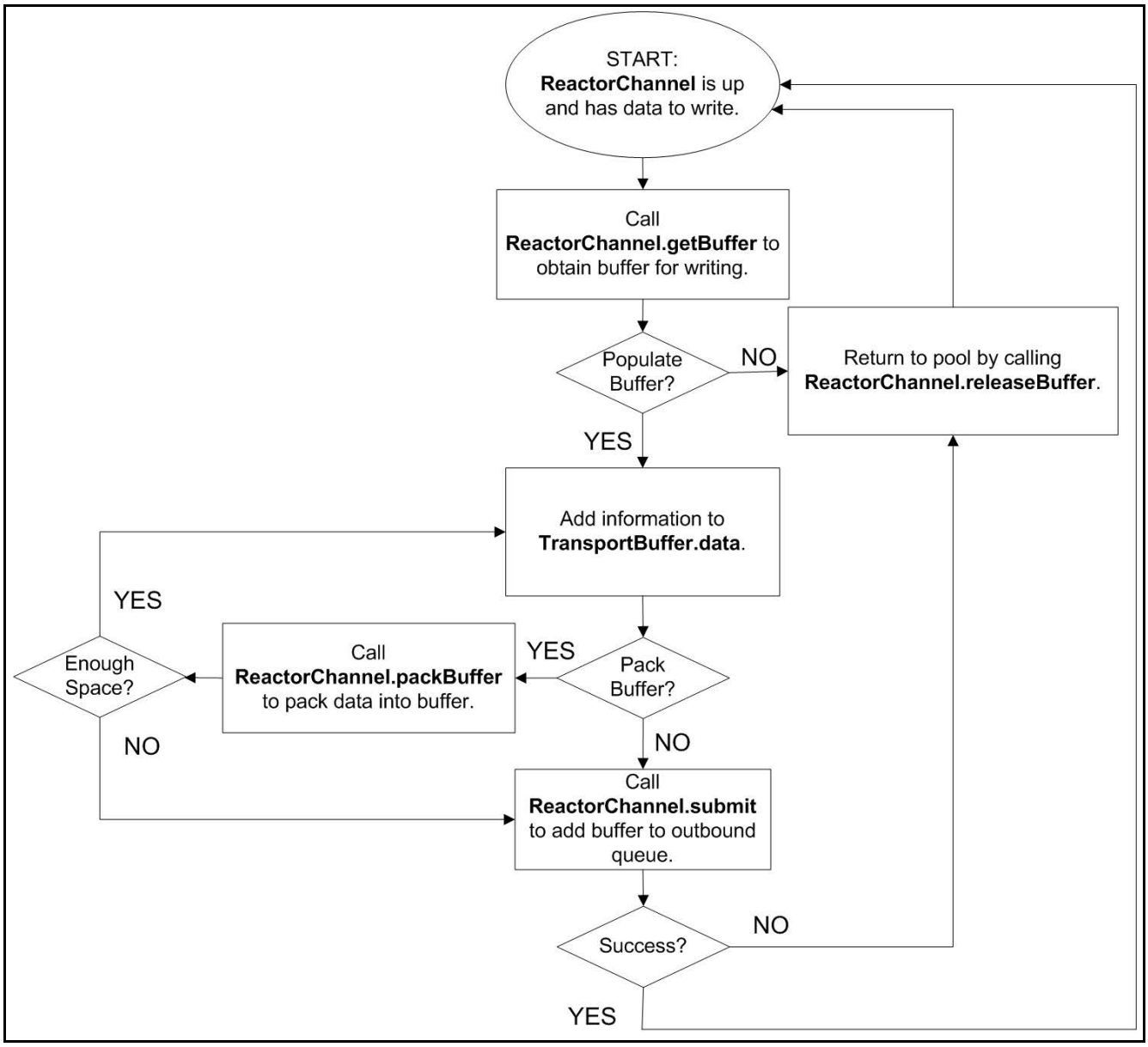


Figure 7. Flow Chart for writing data via ReactorChannel.submit(TransportBuffer...)

6.7.2.1 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 APIJava Edition *Developers Guide*. After acquiring the buffer via **`ReactorChannel.getBuffer`**, you can populate

the **TransportBuffer.data**. If the buffer is not used or the **ReactorChannel.submit(TransportBuffer...)** 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 **ReactorChannel.submit(TransportBuffer...)**, 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.

6.7.2.2 Obtaining a Buffer: ReactorChannel Buffer Management Methods

METHOD NAME	DESCRIPTION
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(TransportBuffer...)** method (for further details, refer to Section 6.7.2.4). Because of some additional book keeping required when packing, the application must specify whether a buffer should be 'packable' when calling **getBuffer**. For more information on packing, refer to Section 6.7.2.11. 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(TransportBuffer...)** call fails, the buffer must be returned to the pool using **releaseBuffer**. If the **ReactorChannel.submit(TransportBuffer...)** call is successful, the buffer will be returned to the correct pool by the transport. This method calls the **Channel.getBuffer** method which has its use and return values described in the Enterprise Transport APIJava Edition *Developers Guide*.
releaseBuffer	Releases a buffer back to the correct pool. This should only be called with buffers that originate from **getBuffer** and are not successfully passed to **ReactorChannel.submit(TransportBuffer...)**. This method calls the Enterprise Transport API **Channel.releaseBuffer** method which has its use and return values described in the Enterprise Transport APIJava Edition *Developers Guide*.
bufferUsage	Returns the number of buffers currently in use by the **ReactorChannel**, this includes buffers that the application holds and buffers internally queued and waiting to be flushed to the connection by the **Reactor**. This method calls the **Channel.bufferUsage** method which has its use and return values described in the Enterprise Transport APIJava Edition *Developers Guide*.

Table 51: ReactorChannel Buffer Management Methods

6.7.2.3 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	A **TransportBuffer** is returned to the user. The **TransportBuffer.length** indicates the number of bytes available to populate and the **TransportBuffer.data** provides a ByteBuffer 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 **ReactorErrorInfo** 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 6.11).
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 API Transport has not been initialized. See the **ReactorErrorInfo** content for more details.

Table 52: ReactorChannel.getBuffer Return Values

6.7.2.4 Writing Data: Overview

After a **TransportBuffer** is obtained from **getBuffer** and populated with the user's data, the buffer can be passed to the **ReactorChannel.submit(TransportBuffer...)** method. This method manages queuing and flushing of user content. It will also perform any fragmentation or compression. If an unrecoverable error occurs, any **TransportBuffer** that has not been successfully passed to **ReactorChannel.submit(TransportBuffer...)** should be released to the pool using **releaseBuffer**. Section 6.7.2.5 describes the **ReactorChannel.submit(TransportBuffer...)** method and its associated parameters.

6.7.2.5 Writing Data: ReactorChannel.submit(TransportBuffer...) Method

METHOD NAME	DESCRIPTION
submit	Writes data. This method expects the buffer to be properly populated. This method calls the Enterprise Transport API **Channel.write** method and also triggers the **Channel.flush** method (described in the Enterprise Transport APIJava Edition *Developers Guide*). This method allows for several modifications and additional parameters to be specified via the **ReactorSubmitOptions** object, defined in Section 6.7.1.2. For a list of return codes, refer to Section 6.7.2.8.

Table 53: ReactorChannel.submit(TransportBuffer...) Method

6.7.2.6 Writing Data: Reactor Submit Options

For a list of submit options and their descriptions for use with **ReactorChannel.submit(TransportBuffer...)**, refer to Section 6.7.1.2.

6.7.2.7 Writing Data: ReactorRequestMsgOptions

For a list of request message options and their descriptions for use with **ReactorChannel.submit(TransportBuffer...)**, refer to Section 6.7.1.4.

6.7.2.8 Writing Data: ReactorChannel.submit(TransportBuffer…) Return Codes

The following table defines the return codes that can occur when using **ReactorChannel.submit(TransportBuffer...)**.

RETURN CODE	DESCRIPTION
ReactorReturnCodes.SUCCESS	Indicates that the **ReactorChannel.submit(TransportBuffer...)** method has succeeded. The **TransportBuffer** will be released by the Enterprise Transport API Reactor.
ReactorReturnCodes.WRITE_CALL_AGAIN	Indicates that a large buffer could not be fully written with this **ReactorChannel.submit(TransportBuffer...)** call. This is typically due to all pool buffers being unavailable. The **Reactor** 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(TransportBuffer...)** 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(TransportBuffer...)**, the application should release it by calling **ReactorChannel.releaseBuffer**.
ReactorReturnCodes.FAILURE	Indicates that a general write failure has occurred. The **ReactorChannel** should be closed. The application should release the **TransportBuffer** by calling **ReactorChannel.releaseBuffer**.

Table 54: ReactorChannel.submit(TransportBuffer...) Return Codes

6.7.2.9 Writing Data: ReactorSubmitOptions Utility Function

For a details on the on the utility method for use with **ReactorChannel.submit(TransportBuffer...)**, refer to Section 6.7.1.5.

6.7.2.10 Example: ReactorChannel.getBuffer and ReactorChannel.submit(TransportBuffer…) Example

The following example shows typical use of **ReactorChannel.getBuffer** and **ReactorChannel.submit(TransportBuffer...)**.

```
TransportBuffer msgBuffer = null;
EncodeIterator encodeIter = CodecFactory.createEncodeIterator();
ReactorSubmitOptions submitOpts = ReactorFactory.createReactorSubmitOptions();

msgBuffer = reactorChannel.getBuffer(1024, false, errorInfo);

encodeIter.clear();
encodeIter.setBufferAndRWFVersion(msgBuffer, reactorChannel.majorVersion(),
        reactorChannel.minorVersion());
encodeMsgIntoBuffer(encodeIter, msgBuffer);

submitOpts.clear();
ret = reactorChannel.submit(msgBuffer, submitOpts, errorInfo);
// check return code
switch (ret)
{
    case ReactorReturnCodes.SUCCESS:
        // successful write, nothing left to do
        return ReactorReturnCodes.SUCCESS;
    break;
    case ReactorReturnCodes.FAILURE:
        // an error occurred, need to release buffer
        reactorChannel.releaseBuffer(msgBuffer, errorInfo);
    break;
    case ReactorReturnCodes.WRITE_CALL_AGAIN:
        // large message couldn't be fully written with one call, pass it to submit again
        ret = reactorChannel.submit(msgBuffer, submitOpts, errorInfo);
    break;
}
```

Code Example 12: Writing Data Using ReactorChannel.submit, ReactorChannel.getBuffer, and ReactorChannel.releaseBuffer

6.7.2.11 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 **`ReactorChannel.packBuffer`** 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 **`TransportBuffer`** 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 6.7.2.12. This method calls the **`Channel.packBuffer`** method as described in the Enterprise Transport APIJava Edition *Developers Guide*.

Table 55: `ReactorChannel.packBuffer` Method

6.7.2.12 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 ReactorReturnCodes.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 56: `ReactorChannel.packBuffer` Return Values

6.7.2.13 Example: ReactorChannel.getBuffer, ReactorChannel.packBuffer, and ReactorChannel.submit(TransportBuffer…)

The following example shows typical use of **ReactorChannel.getBuffer**, **ReactorChannel.packBuffer**, and **ReactorChannel.submit(TransportBuffer…)**.

```
int remainingLength = 0;
TransportBuffer msgBuffer = null;
EncodeIterator encodeIter = CodecFactory.createEncodeIterator();
ReactorSubmitOptions submitOpts = ReactorFactory.createReactorSubmitOptions();

/* get a packable buffer */
msgBuffer = reactorChannel.getBuffer(1024, true, errorInfo);

encodeIter.clear();
encodeIter.setBufferAndRWFVersion(msgBuffer, reactorChannel.majorVersion(),
        reactorChannel.minorVersion());
encodeMsgIntoBuffer(encodeIter, msgBuffer);

/* pack first encoded message into buffer */
remainingLength = reactorChannel.packBuffer(msgBuffer, errorInfo);

encodeIter.clear();
encodeIter.setBufferAndRWFVersion(msgBuffer, reactorChannel.majorVersion(),
        reactorChannel.minorVersion());
encodeMsgIntoBuffer(encodeIter, msgBuffer);

/* pack second encoded message into buffer */
remainingLength = reactorChannel.packBuffer(msgBuffer, errorInfo);

encodeIter.clear();
encodeIter.setBufferAndRWFVersion(msgBuffer, reactorChannel.majorVersion(),
        reactorChannel.minorVersion());

/* now write packed buffer by passing third buffer to submit */
encodeMsgIntoBuffer(encodeIter, msgBuffer);

submitOpts.clear();
ret = reactorChannel.submit(msgBuffer, submitOpts, errorInfo);
```

Code Example 13: Message Packing using ReactorChannel.packBuffer

6.8 Creating and Using Tunnel Streams

The Reactor allows users to create and use special tunnel streams. A tunnel stream is a private stream with additional behaviors, such as end-to-end line of sight for authentication and guaranteed delivery. Tunnel streams are founded on the private streams concept, and the Enterprise Transport API establishes them between consumer and provider endpoints (passing through any intermediate components, such as Refinitiv Real-Time Distribution System or a Refinitiv Real-Time Edge Device).

When creating a tunnel, the consumer indicates any additional behaviors to enforce, which is exchanged with the provider application end point. The provider end-point acknowledges creation of the stream as well as the behaviors that it will enforce on the stream. After the stream is established, the consumer can exchange any content it wants, though the tunnel stream will enforce behaviors on the transmitted content as negotiated with the provider.

A tunnel stream allows for multiple substreams to exist, where substreams follow from the same general stream concept, except that they flow and coexist within the confines of a tunnel stream.

In the following diagram, the orange cylinder represents a tunnel stream that connects the consumer application to the provider application. Notice that the tunnel stream passes directly through intermediate components: the tunnel stream has end-to-end line of sight so that the provider and consumer effectively talk to one another directly, though they traverse multiple devices in the system. Each black line flowing through the cylinder represents a different substream, where each substream transmits its own independent stream of information. Each substream could communicate different market content; for example one could be a Time Series request while another could be a request for Market Price content. A substream can also connect to a special provider application called a Queue Provider. A Queue Provider allows for persistence of content exchanged over the tunnel stream and substream, and helps provide content beyond the end-point visible to the consumer. To interact with a Queue Provider, additional addressing information is required, described in more detail in Section 8.6.

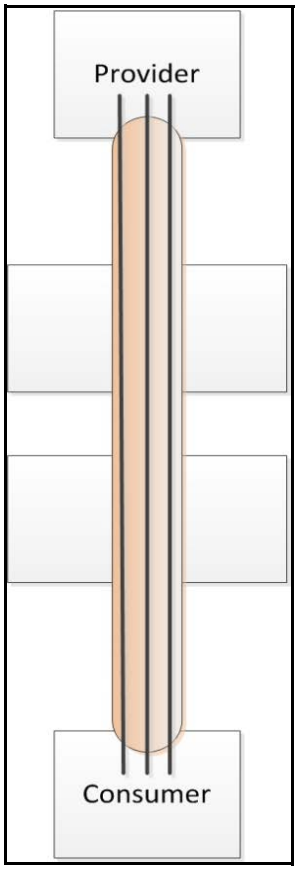


Figure 8. Tunnel Stream Illustration

6.8.1 Authenticating a Tunnel Stream

Providers might require the consumer to authenticate itself when establishing the tunnel stream. The type of authentication, if any, is given by the **ClassOfService.authentication.type**. For more information on class or service, refer to Section 6.8.3.

The **ClassOfService.authentication.type** may be set to **OMM_LOGIN**. When an OMM consumer expects this type of authentication, it should set a **LoginRequest** message on the **TunnelStreamOpenOptions.authLoginRequest** member. If the OMM consumer application does not provide it, the API will use the login request provided on the **ConsumerRole.rdmLoginRequest** when the consumer connected (refer to Section 6.3.2). The consumer must provide one of these for authentication of this type.

The login request will be sent to the provider. When the provider sends a Login response to complete the authentication, the **TunnelStreamStatusEvent** event given to the consumer will include an **TunnelStreamAuthInfo** object with more details. OMM provider applications will see the login request as a normal message within the **TunnelStream** and should respond with a login response message via **TunnelStream.submit**.

Other types of authentication might be specified, but must be performed by both the provider and consumer applications by submitting normal **TunnelStream** messages via **TunnelStream.submit**.

The **TunnelStreamAuthInfo** object contains the following member:

MEMBER	DESCRIPTION
loginMsg	The Login message sent by the tunnel stream's provider application, which resulted in this event.

Table 57: TunnelStreamAuthInfo Members

6.8.2 Opening a Tunnel Stream

The user can create one or more tunnel streams and associate them with any **ReactorChannel**, which opens the private stream connection and negotiates any specified behaviors. Prior to opening a tunnel stream, you must implement the **StatusEventCallback**, which is described in Section 6.8.4.

6.8.2.1 ReactorChannel.openTunnelStream Method

METHOD NAME	DESCRIPTION
openTunnelStream	Begins the establishment of a tunnel stream. The **TunnelStream** is returned via the **TunnelStreamStatusEventCallback** as specified on the **TunnelStreamOpenOptions**. For more details, refer to Section 6.8.2.2.

Table 58: ReactorChannel.openTunnelStream Method

6.8.2.2 TunnelStreamOpenOptions

The **TunnelStreamOpenOptions** contain event handler associations and options for use in creating a tunnel stream.

CLASS MEMBER	DESCRIPTION
domainType	Indicates the domain for which the tunnel stream is established. Set this to the domain specified on the service on which the Enterprise Transport API opens the tunnel stream.
streamId	Indicates the stream ID to use for the tunnel stream. Though substreams will flow within this stream ID, each will have their own independent stream ID. For example, a tunnel stream can have an ID of 10. If a substream is opened to retrieve TRI data, the substream can have a stream ID of 5, though it is encapsulated in the tunnel stream whose stream ID is 10.
serviceId	Indicates the service ID of the service on which you open the tunnel stream.
userSpecObject	Indicates a user-specified object passed in via these options and then associated with the **TunnelStream**.
statusEventCallback	Specifies an instance of the callback for **TunnelStreamStatusEvent**s, which provides the **TunnelStream** on initial connection, and after the tunnel stream is established, communicates the tunnel stream's state information. For further details, refer to Section 6.8.4.
queueMsgCallback	Specifies the instance of the callback used to handle Queue Messages received on this **TunnelStream**. • For details on the **TunnelStreamQueueMsgCallback**, refer to Section 6.8.4. • For details on various Queue Messages, refer to Section 8.6.
defaultMsgCallback	Specifies the instance of the callback that handles all other content received on this **TunnelStream**. For further details, refer to Section 6.8.4.
name	Specifies the tunnel stream name. **name** cannot be longer than 255 characters.
responseTimeout	Sets the duration (in seconds) to wait for a provider to respond to a tunnel stream open request. If the provider does not respond in time, a **TunnelStreamStatusEvent** is sent to the application to indicate that the tunnel stream was not opened.
guaranteedOutputBuffers	Sets the number of guaranteed output buffers available for the tunnel stream.
authLoginRequest	Specifies the login request to send if **ClassOfService.authentication.type** is set to **OMM_LOGIN**. If absent, the API uses the login request provided on the **ConsumerRole.rdmLoginRequest**.
classOfService	The class of service of the tunnel stream to be opened. For further details on **ClassOfService**, refer to Section 6.8.3.

Table 59: TunnelStreamOpenOptions

6.8.3 Negotiating Stream Behaviors: Class of Service

ClassOfService is used to negotiate **TunnelStream** behaviors. Negotiated behaviors are divided into five categories: common, authentication, flow control, data integrity, and guarantee.

- When a consumer application calls **ReactorChannel.openTunnelStream**, it sets the **TunnelStreamOpenOptions.classOfService** members to manage and control tunnel stream behaviors. The consumer passes these settings to the connected provider.
- When a provider application receives an **TunnelStreamRequestEvent**, the provider calls **TunnelStreamRequestEvent.classOfService** to retrieve the behaviors requested by the consumer.

After tunnel stream negotiation is complete, the provider and consumer each receive a **TunnelStreamStatusEvent** where each can view the negotiated behaviors on the **TunnelStream** object.

NOTE: Do not modify the **ClassOfService** member of the **TunnelStream**.

The enumerations given for members described in this section can be found in **com.refinitiv.eta.rdm.ClassesOfService**.

6.8.3.1 ClassOfService Common Member

Common elements describe options related to the exchange of messages, such as the maximum message size and desired exchange protocol.

MEMBER	DEFAULT	RANGE/ ENUMERATIONS	DESCRIPTION
maxFragmentSize	6144	1 – 2,147,483,647	The maximum size of message fragments exchanged on the tunnel stream. This value is set only by providers when accepting a tunnel stream.
maxMsgSize	614400	1 – 2,147,483,647	The maximum size of messages exchanged on the tunnel stream. This value is set only by providers when accepting a tunnel stream.
protocolMajorVersion	Codec.majorVersion()	0 – 255	The major version of the protocol specified by **protocolType**.
protocolMinorVersion	Codec.minorVersion()	0 – 255	The minor version of the protocol specified by **protocolType**.
protocolType	Codec.protocolType()	0 – 255	Identifies the protocol of the messages exchanged on the tunnel stream.

Table 60: ClassOfService.common Members

6.8.3.2 ClassOfService Authentication Member

The authentication member contains options to authenticate a consumer to the corresponding provider.

MEMBER	DEFAULT	RANGE/ ENUMERATIONS	DESCRIPTION
type	ClassesOfService. AuthenticationTypes. *NOT_REQUIRED*	ClassesOfService. AuthenticationTypes. *NOT_REQUIRED* == 0, ClassesOfService. AuthenticationTypes. OMM_LOGIN == 1	Indicates the type of authentication, if any, to perform on the tunnel stream. For further details on authentication, refer to Section 6.8.1.

Table 61: `ClassOfService.authentication` Members

6.8.3.3 ClassOfService Flow Control Members

The flow control member contains options related to flow control, such as the type and the allowed window of outstanding data.

MEMBER	DEFAULT	RANGE/ ENUMERATIONS	DESCRIPTION
type	ClassesOfService. FlowControlTypes. NONE	ClassesOfService. FlowControlTypes. NONE == 0, ClassesOfService. FlowControlTypes. BIDIRECTIONAL == 1	Indicates the type of flow control (if any) to apply to the tunnel stream.
recvWindowSize	-1	0 – 2,147,483,647	Sets the amount of data (in bytes) that the remote peer can send to the application over a reliable tunnel stream. If **`type`** is set to **NONE**, this parameter has no effect. **-1** indicates that the application wants to use the default value for the negotiated flow control type. In this case, if **`type`** is set to **BIDIRECTIONAL**, the default is **12288**.
sendWindowSize	None	0 – 2,147,483,647	Indicates the amount of data (in bytes) the application can send to the remote peer on a reliable tunnel stream. This value is provided on the **`TunnelStream`** object and does not need to be set when opening or accepting a tunnel stream. This value is retrieved from the remote end and is informational, as flow control is performed by the API. When room is available in the window, the API transmits more content as submitted by the application. If **`type`** is set to **NONE**, this parameter has no effect.

Table 62: `ClassOfService.flowControl` Members

6.8.3.4 ClassOfService Data Integrity Member

The data integrity member contains options related to the reliability of content exchanged over the tunnel stream.

MEMBER	DEFAULT	RANGE	DESCRIPTION
type	ClassesOfService. DataIntegrityTypes. BEST_EFFORT	ClassesOfService. DataIntegrityTypes. BEST_EFFORT == 0, ClassesOfService. DataIntegrityTypes. RELIABLE == 1	Sets the level of reliability for message transmission on the tunnel stream. If set to **RELIABLE**, data is retransmitted as needed over the tunnel stream to ensure that all messages are delivered in the correct order. **NOTE:** At this time, **RELIABLE** is the only supported option.

Table 63: `ClassOfService.dataIntegrity` Members

6.8.3.5 ClassOfService Guarantee Members

The guarantee member contains options related to the guarantee of content submitted over the tunnel stream.

Consumer applications performing Queue Messaging to a Queue Provider should set the **`ClassOfService.guarantee.type`** to **ClassesOfService.GuaranteeTypes.PERSISTENT_QUEUE**.

MEMBER	DEFAULT	RANGE	DESCRIPTION
type	ClassesOfService. GuaranteeTypes. NONE	ClassesOfService. GuaranteeTypes. NONE == 0, ClassesOfService. GuaranteeTypes. PERSISTENT_QUEUE == 1	Indicates the level of guarantee that will be performed on this stream. **PERSISTENT_QUEUE** is not supported for provider applications. **NOTE:** If **`type`** is set to **PERSISTENT_QUEUE** for a consumer application, the data integrity **`type`** must also be set to **RELIABLE** and the flow control **`type`** to **BIDIRECTIONAL**.
persistLocally	true	false, true	Indicates whether messages are persisted locally on the tunnel stream. When **`type`** is **NONE**, this member has no effect.
persistenceFilePath	NULL	n/a	File path where files containing persistent messages may be stored. If set to NULL, the current working directory is used. When **`type`** is **NONE**, or when **`persistLocally`** is set to **false**, this member has no effect.

Table 64: `ClassOfService.guarantee` Members

6.8.4 Tunnel Stream Callback Methods and Event Types

6.8.4.1 Tunnel Stream Callback Methods

The **TunnelStream** delivers events via the following user-implemented callback methods. These callback methods return event objects as defined in Section 6.8.4.2.

CALLBACK METHOD	DESCRIPTION
statusEventCallback	Communicates status information about the tunnel stream. Additionally, this callback delivers the **TunnelStream** object after the enhanced private stream is established. This callback provides a **TunnelStreamStatusEvent** to the application. Details about this event are available in Section 6.8.4.2.
defaultMsgCallback	Similar to the **ReactorChannel.defaultMsgCallback**, content received by the tunnel stream are returned via this callback if it is not handled by a more specific content handler, such as the **queueMsgCallback**. This callback provides a **TunnelStreamMsgEvent** to the application. Details about this event are available in Section 6.8.4.2.
queueMsgCallback	Any queue messages are delivered via this callback and presented to the user in their native queue message formats. If unspecified, queue messages are delivered via the **defaultMsgCallback**; however they are not presented in a queue message format. This callback provides a **TunnelStreamQueueMsgEvent** to the application. Details about this event are available in Section 6.8.4.2.

Table 65: Tunnel Stream Callback Methods

6.8.4.2 Tunnel Stream Callback Event Types

Various tunnel stream callbacks return their information via specific event objects. The following table defines these events.

EVENT	EVENT DESCRIPTION	CLASS MEMBER	CLASS MEMBER DESCRIPTION
TunnelStreamStatusEvent	This event presents the tunnel stream and its status.	tunnelStream	Returns the **`TunnelStream`** associated with this event. When the **`TunnelStream`** is initially opened, the initial instance of the **`TunnelStream`** is made available.
		state	Indicates status information associated with the **`TunnelStream`**. For example: • A state of **OPEN** and **OK** indicates that the tunnel stream is established and content should be flowing as expected. • A state of **CLOSED_RECOVER** or **SUSPECT** indicates that the connection or tunnel stream might be lost. However, if performing guaranteed messaging, content might be persisted by the reactor and communicated upon recovery of the tunnel stream.
		authInfo	(Consumers only) Provides information about a received authentication response.
TunnelStreamMsgEvent	This event presents content received on the **`TunnelStream`**. If a more specific handler (i.e., **`queueMsgCallback`**) is also configured, messages of that type will go to their specific handler.	tunnelStream	Returns the **`TunnelStream`** associated with this event.
		msg	Contains the content being presented to the user. • If content adheres to the OMM, it is partially decoded for user convenience. • If content is opaque, a buffer housing the contents is presented to the user via this object.
		transportBuffer	The transport buffer associated with this event.
		containerType	The container type associated with this event's transport buffer.
TunnelStreamQueueMsgEvent	This event presents any queue message content received on the **`TunnelStream`**.	tunnelStream	Returns the **`TunnelStream`** associated with this event.
		queueMsg	Contains the content being presented to the user, presented as a queue message object.

Table 66: Tunnel Stream Callback Event Types

6.8.5 Code Sample: Opening and Managing a Tunnel Stream

The following code sample is a basic example of opening a tunnel stream. The example assumes that a **Reactor** and **ReactorChannel** are already open and properly established.

```
// Basic sample for event handlers
class  Sample implements StatusEventCallback, TunnelStreamQueueMsgCallback,
        TunnelStreamDefaultMsgCallback
{
    ReactorErrorInfo _errorInfo;

    // StatusEventCallback
    public int statusEventCallback(TunnelStreamStatusEvent event)
    {
        System.out.println("Status of Tunnel Stream (" + event.tunnelStream().streamId() + ") is " +
                event.state());
        Return ReactorCallbackReturnCodes.SUCCESS;
    }

    // TunnelStreamDefaultMsgCallback
    public int TunnelStreamDefaultMsgCallback(TunnelStreamMsgEvent event)
    {
        System.out.println("Received content on Tunnel Stream (" + event.tunnelStream().streamId() +
                ")");
        Return ReactorCallbackReturnCodes.SUCCESS;
    }

    // TunnelStreamQueueMsgCallback
    public int tunnelStreamQueueMsgCallback(TunnelStreamQueueMsgEvent event)
    {
        System.out.println("Received Queue Message on Tunnel Stream (" +
                event.tunnelStream().streamId() + ")");
        Return ReactorCallbackReturnCodes.SUCCESS;
    }
}

int openTunnelStream()
{
    TunnelStreamOpenOptions _openOptions = RectorFactory.createTunnelStreamOpenOptions();

    // populate the options and enable guaranteed delivery for communication with a Queue Provider
    _openOptions.streamId(TUNNEL_STREAM_ID);
    _openOptions.domainType(DomainTypes.QUEUE_MESSAGING);
    _openOptions.serviceId(QUEUE_MESSAGING_SERVICE_ID);
    // specify the event handlers
    _openOptions.statusEventCallback(this);
    _openOptions.TunnelStreamDefaultMsgCallback(this);
    _openOptions.queueMsgCallback(this);

    if ((reactorChannel.openTunnelStream(_openOptions, _errorInfo)) != ReactorReturnCodes.SUCCESS)
```

```
    {
        System.out.println("openTunnelStream failed!");
        return ReactorReturnCodes.FAILURE;
    }

    System.out.println("openTunnelStream succeeded!");
    return ReactorReturnCodes.SUCCESS;
}
```

Code Example 14: Opening a Tunnel Stream

6.8.6 Accepting Tunnel Streams

OMM provider applications can accept tunnel streams provided on an **ReactorChannel** (enabled by specifying a **tunnelStreamListenerCallback** on the **ProviderRole**).

When a consumer opens a tunnel stream, the **tunnelStreamListenerCallback** receives an **TunnelStreamRequestEvent**. At this point, the provider should call **TunnelStreamRequestEvent.classOfService** to retrieve the **ClassOfService** requested by the tunnel stream and ensure that the parameters indicated by the members of that class of service match what the provider allows. The provider can also check the **TunnelStreamRequestEvent.classOfServiceFilter** to determine which behaviors the consumer supports. For more information on this filter, refer to Section 6.8.6.1.

- To accept a tunnel stream, the provider must call **ReactorChannel.acceptTunnelStream** with the given **TunnelStreamRequestEvent**. Further events regarding the accepted stream are provided in the specified **TunnelStreamAcceptOptions.statusEventCallback**.
- To reject a tunnel stream, the provider calls **ReactorChannel.rejectTunnelStream** with the given **TunnelStreamRequestEvent**. No further events are received for that tunnel stream.

Queue messaging (a **ClassOfService.guarantee.type** setting of **PERSISTENT_QUEUE**) is not supported for provider applications.

The API automatically rejects tunnel streams that contain invalid information. When this happens, the provider application receives warnings via a **ReactorChannelEvent**. The type will be set to **ReactorChannelEventTypes WARNING** and the **ReactorErrorInfo** in the event will contain text describing the reason for the rejection.

Warning! Ensure that the provider application calls **ReactorChannel.acceptTunnelStream** or **ReactorChannel.rejectTunnelStream** before returning from the **tunnelStreamListenerCallback**. If not, the provider application will receive a warning via an **ReactorChannelEvent** similar to the above, and the stream will be automatically rejected.

6.8.6.1 Reactor Tunnel Stream Listener Callback

Providers that want to handle tunnel streams from connected consumers can specify a **TunnelStreamListenerCallback**. This callback informs the provider application of any consumer tunnel stream requests.

The provider can specify this callback function on the **ProviderRole**, which has the following signature:

```
listenerCallback(TunnelStreamRequestEvent event);
```

For more information on the **ProviderRole**, refer to Section 6.3.3.

A **TunnelStreamRequestEvent** is returned to the application via the **TunnelStreamListenerCallback**.

MEMBER	DESCRIPTION
reactorChannel	Specifies the **ReactorChannel** on which the event was received.
streamId	Specifies the stream ID of the requested tunnel stream.
domainType	Specifies the domain type of the requested tunnel stream.
serviceId	Specifies the service ID of the requested tunnel stream.
name	Specifies the name of the requested tunnel stream.
classOfServiceFilter	Sets a filter that indicates which **ClassOfService** members are present. The provider can use this filter to determine whether behaviors are supported by the consumer and if needed, reject the tunnel stream before calling **TunnelStreamRequestEvent.classOfService** to get the full **ClassOfService**. For enumerations of the flags present in this filter, refer to **com.refinitiv.eta.rdm.ClassesOfService.FilterFlags**.
classOfService	Specifies the **ClassOfService** for the requested tunnel stream.

Table 67: TunnelStreamRequestEvent Members

6.8.6.2 ReactorChannel.acceptTunnelStream Method

METHOD NAME	DESCRIPTION
ReactorChannel.acceptTunnelStream	Accepts a tunnel stream requested by a consumer. The **TunnelStream** is returned in the **TunnelStreamStatusEventCallback** specified on the **TunnelStreamAcceptOptions**. For more information, refer to Section 6.8.6.3.

Table 68: ReactorChannel.acceptTunnelStream Method

6.8.6.3 TunnelStreamAcceptOptions

OPTION	DESCRIPTION
statusEventCallback	Specifies the instance of the callback for **`TunnelStreamStatusEvents`**, which provides the **`TunnelStream`** on initial connection and then communicates state information about the tunnel afterwards. For details on the **`TunnelStreamStatusEventCallback`**, refer to Section 6.8.4.1.
defaultMsgCallback	Specifies the instance of the callback used to handle all other content received on this **`TunnelStream`**. For details on **`TunnelStreamdefaultMsgCallback`**, refer to Section 6.8.4.1.
userSpecObject	Specifies a user-defined object passed in via these options and then associated with the **`TunnelStream`**.
classOfService	Specifies an **`ClassOfService`** with members indicating behaviors that the application wants to apply to the **`TunnelStream`**. For more information on class of service, refer to Section 6.8.3.
guaranteedOutputBuffers	Sets the number of pooled buffers available to the application when writing content to **`TunnelStream`**.

Table 69: `TunnelStreamAcceptOptions` Options

6.8.6.4 ReactorChannel.rejectTunnelStream

METHOD NAME	DESCRIPTION
ReactorChannel.rejectTunnelStream	Rejects a tunnel stream requested by a consumer. No further events will be received for this tunnel stream. For more information, refer to Section 6.8.6.5.

Table 70: `ReactorChannel.rejectTunnelStream` Method

6.8.6.5 TunnelStreamRejectOptions

OPTION	DESCRIPTION
state	A **`State`** to send to the consumer. The application can use the **`state.streamState`**, **`state.dataState`**, and **`state.text`** to indicate the nature of the rejection.
expectedClassOfService	An optional **`ClassOfService`** to send to the consumer. If rejecting the stream due to a problem with the **`ClassOfService`** parameters from the **`TunnelStreamRequestEvent`**, the provider application should populate this with the associated parameters.

Table 71: `TunnelStreamRejectOptions` Options

6.8.6.6 Accepting a Tunnel Stream Code Sample

The following code illustrates how to accept a tunnel stream requested by a consumer. The example presumes that a **Reactor** and **ReactorChannel** are already open and properly established.

```
public int listenerCallback(TunnelStreamRequestEvent event)
{
    int ret;
    TunnelStreamAcceptOptions acceptOpts = ReactorFactory.createTunnelStreamAcceptOptions();

    if (isFilterValid(event.classOfServiceFilter()) &&
        isClassOfServiceValid(event.classOfService())
    {
        acceptOpts.clear();

        // set class of service to what this provider supports
        acceptOpts.classOfService().dataIntegrity().type(ClassesOfService.DataIntegrityTypes.RELIABLE
                );
        acceptOpts.classOfService().flowControl().type((ClassesOfService.FlowControlTypes.
                BIDIRECTIONAL));

        // Set Authentication to match consumer. This provider will perform OMM Login authentication if
        // requested.
        acceptOpts.classOfService().authentication().type(event.classOfService().authentication().
                type());

        acceptOpts.statusEventCallback(this);
        acceptOpts.defaultMsgCallback(this);

        if ((ret = event.reactorChannel().acceptTunnelStream(event, acceptOpts, event.errorInfo()))
                < ReactorReturnCodes.SUCCESS)
        {
        System.out.println("acceptTunnelStream() failed with return code: " + ret + " <" +
                event.errorInfo().error().text() + ">");
        }
    }

    return ReactorCallbackReturnCodes.SUCCESS
}
```

Code Example 15: Accepting a Tunnel Stream Code Example

6.8.6.7 Rejecting a Tunnel Stream Code Sample

The following code illustrates how to reject a tunnel stream requested by a consumer. The example assumes that a **Reactor** and **ReactorChannel** are already open and properly established.

```
public int listenerCallback(TunnelStreamRequestEvent event)
{
    int ret;

    /* Now presuming that the application wishes to reject the tunnel stream because the requested
     * class of service is invalid. */

    if (!isFilterValid(event.classOfServiceFilter()) || !isClassOfServiceValid(event.classOfService())
    {
        /* Set what the class of service is expected to be. */
        ClassOfService expectedCos = ReactorFactory.createClassOfService();
        expectedCos.clear()
        expectedCos.authentication().type(ClassesOfService.AuthenticationTypes.OMM_LOGIN);
        expectedCos.flowControl().type(ClassesOfService.FlowControlTypes.BIDIRECTIONAL);
        expectedCos.dataIntegrity().type(ClassesOfService.DataIntegrityTypes.RELIABLE);
        /* ... (set additional members, based on what is desired by the provider) */

        TunnelStreamRejectOptions rejectOpts = ReactorFactory.createTunnelStreamRejectOptions();
        rejectOpts.clear();
        rejectOpts.state().streamState(StreamStates.CLOSED);
        rejectOpts.state().dataState(DataStates.SUSPECT);
        rejectOpts.state().code(StateCodes.NONE);
        rejectOpts.state().text().data("Unsupported TunnelStream class of service");
        rejectOpts.expectedClassOfService(expectedCos);

        if ((ret = event.reactorChannel().rejectTunnelStream(event, rejectOpts, event.errorInfo())) <
                ReactorReturnCodes.SUCCESS)
        {
            System.out.println("rejectTunnelStream() failed with return code: " + ret + " <" +
                    event.errorInfo().error().text() + ">");
        }
    }

    return ReactorCallbackReturnCodes.SUCCESS
}
```

Code Example 16: Rejecting a Tunnel Stream Code Example

6.8.7 Receiving Content on a TunnelStream

Invoking the **ReactorChannel.dispatch** method reads and processes inbound content, where any information received on this **TunnelStream** will be delivered to the application via the tunnel stream callback methods specified via **openTunnelStream** or **ReactorChannel.acceptTunnelStream**.

Dispatching this content works in the same manner as dispatching any other content on the reactor.

- Tunnel stream callback methods are described in Section 6.8.4.
- Tunnel stream callback methods deliver the events described in Section 6.8.4.2.

6.8.8 Sending Content on a TunnelStream

When you send content on a **TunnelStream**: get a buffer from the **TunnelStream**, encode your content into the buffer, and then use the **TunnelStream.submit** method to push the content out over the **TunnelStream**. By obtaining a buffer from the **TunnelStream**, the reactor can then properly handle any negotiated behaviors, making this functionality nearly transparent.

6.8.8.1 Tunnel Stream Buffer Methods

METHOD NAME	DESCRIPTION
getBuffer	Obtains a buffer from the **TunnelStream**. To properly enforce negotiated behaviors on content in the buffer, the Enterprise Transport API associates the buffer with the tunnel stream from which it is obtained.
info	Gets information about the Tunnel Stream by returning the **TunnelStreamInfo** structure. For details on **TunnelStreamInfo** methods, refer to Section 6.8.8.4.
releaseBuffer	Releases a buffer back to the **TunnelStream** from which it came. You should release any buffer that you do not submit. Releasing the buffer ensures it is properly recycled and can be reused.
	NOTE: If you submit a buffer properly, you do not need to release it, because the submit method automatically releases it after sending the content on the **TunnelStream**.

Table 72: Tunnel Stream Buffer Methods

6.8.8.2 Tunnel Stream Submit

The submit method is used to write content to the **TunnelStream**. This method also enforces any specified behaviors on submitted content (e.g., if guaranteed messaging is specified, this content follows all configured persistence options).

METHOD NAME	DESCRIPTION
submit	Use the submit method to pass in opaque or RDM message content (including Queue Messages) to be processed and sent over the **TunnelStream**. This method has additional options that can be specified via **TunnelStreamSubmitOptions** (refer to Section 6.8.8.3).

Table 73: Tunnel Stream Submit Method

6.8.8.3 TunnelStream.submit Option

When calling **TunnelStream.submit** with a buffer, you can use **TunnelStreamSubmitOptions** to provide the **containerType** option.

MEMBER	DESCRIPTION
containerType	Specifies the type of data in the buffer being submitted. For example: • If the submitted buffer contains a **Msg**, set **containerType** **DataTypes.MSG**. • If sending non-RWF data, set **containerType** to a non-RWF type, such as **DataTypes.OPAQUE**. For more information on possible container types, refer to the Enterprise Transport APIJava Edition *Developers Guide*.

Table 74: TunnelStreamSubmitOptions Members

6.8.8.4 TunnelStreamInfo Methods

The following table describes values available when using the **TunnelStream.info** method (for details, refer to Section 6.8.8.1). This information is returned as part of the **TunnelStreamInfo** object.

METHOD NAME	DESCRIPTION
bigBuffersUsed	Returns the number of big buffers used by user application.
buffersUsed	Returns the total number of buffers used by the Tunnel Stream for a user application.
ordinaryBuffersUsed	Returns the total number of buffers with a size less than or equal to 6144 bytes used by Tunnel Stream for a user application. Java implementation maintains an internal pool, thus you can allocate buffer space in advance and use buffer space when needed for **TunnelStream.getBuffer** calls.

Table 75: TunnelStreamInfo Methods

6.8.8.5 Submitting Content on a Tunnel Stream Code Sample

The following code sample is a basic example of writing opaque content to a tunnel stream. You can combine this example with the QueueData message samples in subsequent chapters to send content to a Queue Provider.

```
int submitMessage()
{
    TunnelStreamSubmitOptions _submitOpts = ReactorFactory.createTunnelStreamSubmitOptions();

    // gets a buffer of 50 bytes to put content into.
    TransportBuffer _buffer = tunnelStream.getBuffer(50, _errorInfo);

    // put generic content into the buffer
    _buffer.data().put("Hello World!");
    _submitOpts.containerType(DataTypes.OPAQUE);

    if ((tunnelStream.submit(_buffer, _submitOpts, _errorInfo)) != ReactorReturnCodes.SUCCESS)
    {
        System.out.println("Content submission failed!");
```

```
        // Because submission failed, we need to return the buffer to the tunnel stream
        tunnelStream.releaseBuffer(_buffer, _errorInfo);

        return ReactorReturnCodes.FAILURE;
    }

    System.out.println("Content submission succeeded!");
    // Thanks to successful submission, we do not need to release the buffer because the Reactor will.
    return ReactorReturnCodes.SUCCESS;
}
```

Code Example 17: Submitting Content on a Tunnel Stream

6.8.9 Closing a Tunnel Stream

When an application has completed its use of a **`TunnelStream`**, close it using the **`close`** method.

6.8.9.1 Tunnel Stream Close

METHOD NAME	DESCRIPTION
close	Closes a tunnel stream. After you close a tunnel stream, the Enterprise Transport API cleans up any data that was stored for guaranteed messaging or reliable delivery. The **`finalStatusEvent`** argument indicates that the application wants to receive a final **`TunnelStreamStatusEvent`** whenever the tunnel stream closes. If this is set to **true**, the tunnel stream will be cleaned up after the application receives the final **`TunnelStreamStatusEvent`** event.

Table 76: Tunnel Closure Method

6.8.9.2 Closing a Tunnel Stream Sample

The following code sample illustrates how to close a tunnel stream.

```
int closeTunnelStream()
{

    if ((tunnelStream.close(true,_errorInfo)) != ReactorReturnCodes.SUCCESS)
    {
        System.out.println("Closing tunnel stream failed!");
        return ReactorReturnCodes.FAILURE;
    }

    System.out.println("Tunnel Stream closed successfully.");
    return ReactorReturnCodes.SUCCESS;
}
```

Code Example 18: Closing a Tunnel Stream

6.9 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.

6.9.1 Querying Service Discovery

6.9.1.1 queryServiceDiscovery Method

METHOD	DESCRIPTION
Reactor.queryServiceDiscovery	Queries service endpoints from the Refinitiv Real-Time Optimized service according to the **`ReactorServiceDiscoveryOptions`** that you specify (listed in Section 6.9.1.2). Error handling is managed by the **`ReactorErrorInfo`** object.

Table 77: `Reactor.queryServiceDiscovery` Method

6.9.1.2 ReactorServiceDiscoveryOptions

MEMBER	DESCRIPTION
clientId	**Required**. A **`Buffer`** that specifies a unique ID defined for an application making a request to the token service.
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 6.9.1.4.
password	**Required**. A **`Buffer`** that specifies a password for authorization with the token service.
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.
proxyDomain	Optional. A **`Buffer`** that specifies the proxy domain of the user to authenticate. **Required** for NTLM or for Negotiate/Kerberos or for Kerberos authentication protocols.
reactorServiceEndpointEventCallback	A callback function that receives **`ReactorServiceEndpointEventCallback`**s. Applications can take service endpoint information from the callback to get an endpoint and establish a connection to the service.
tokenScope	A **`Buffer`** that specifies an optional token scope to limit the scope of the generated token from the token 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 6.9.1.3.
userName	**Required**. A **`Buffer`** that specifies a user name for authorization with the token service.
userSpecObj	Optional. A user-specified pointer which is set on the **`ReactorServiceEndpointEvent`**. Also refer to Section 6.9.1.5.

Table 78: `ReactorServiceDiscoveryOptions` Members

6.9.1.3 queryServiceDiscovery Transport Protocol Enumerations

ENUMERATED NAME	DESCRIPTION
RD_TP_INIT = 0	Specifies that the transport's protocol is unknown.
RD_TP_TCP = 1	Specifies that the service discovery should use the TCP transport protocol.
RD_TP_WEBSOCKET = 2	Specifies that the service discovery should use the Websocket transport protocol.

Table 79: `ReactorDiscoveryTransportProtocol` Enumerations

6.9.1.4 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 80: `ReactorDiscoveryDataFormatProtocol` Enumerations

6.9.1.5 ReactorServiceEndpointEvent

MEMBER	DESCRIPTION
serviceEndpointInfo	Lists the service endpoints associated with this event. See also Section 6.9.1.6.
userSpecObj	Optional. A user-specified object associated with this **`ReactorServiceEndpointEvent`**.

Table 81: `ReactorServiceEndpointEvent` Members

6.9.1.6 ReactorServiceEndpointInfo

`ReactorServiceEndpointEvent` represents service endpoint information.

MEMBER	DESCRIPTION
dataFormatList	A **`List<String>`** that contains a list of data formats used by the transport.
endPoint	A **`String`** that specifies the domain name of the service access endpoint.
locationList	A **`List<String>`** that specifies a list of service locations.
port	A **`String`** that specifies the port number used to establish connection.
provider	A **`String`** that specifies a public cloud provider.
transport	A **`String`** that specifies the transport type used to access the service.

Table 82: `ReactorServiceEndpointEvent` Members

6.9.2 OAuth Credential Management

6.9.2.1 ReactorOAuthCredential Class

You use the **`ReactorOAuthCredential`** class to certify OAuth user credentials when connecting to the cloud. **`ReactorOAuthCredential`** includes the following members:

MEMBER	DESCRIPTION
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. • Version 1 authentication (Client ID usage with oAuth Password Credentials): refer toSection 7.4.1.
clientSecret	**Required** for Version 2 oAuth ClientCredentials logins. A **`Buffer`** that specifies the Service Account "secret" for authentication. For further information, refer to Section 7.5.
password	**Required** for Version 1 oAuth Password Credentials logins. A **`Buffer`** that specifies the password used in tandem with the **`userName`** to obtain the access token.
reactorOAuthCredentialEventCallback	A callback function that receives the **`ReactorOAuthCredentialEvent`** to specify the **`password`** and/or **`clientSecret`**. If **`pOAuthCredentialEventCallback`** is specified, the Value Added Components Reactor does not store the **`password`** or **`clientSecret`**. In which case, the application must supply the **`password`** whenever receiving a new refresh token. For details on this process, refer to Section 7.4.2.
takeExclusiveSignOnControl	Optional and only used with Version 1 Password Credentials logins. A boolean that, if set to true, forces sign-out of any other applications using the same credentials. By default, this is set to true.
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 Refinitiv Data 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`**.
userName	**Required** for Version 1 oAuth Password Credentials logins. A **`Buffer`** that specifies the user name used to obtain the access token from the Refinitiv Data Platform.

Table 83: `ReactorOAuthCredential` Class Members

6.9.2.2 reactorOAuthCredentialEventCallback

Whenever the Enterprise Transport API needs a new refresh token, it needs to again supply the username, Client ID, and password. But the Enterprise Transport API stores only the username and Client ID, not the password. To obtain the password (and if available, the client secret), the Enterprise Transport API sends the **`ReactorOAuthCredentialEvent`** callback to the application.

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

Table 84: `ReactorOAuthCredentialEvent` Members

6.9.2.3 ReactorOAuthCredentialRenewal

MEMBER	DESCRIPTION
userName	Conditional. A **Buffer** that specifies the user name that the Enterprise Transport API sends to the Refinitiv Data Platform token service. The **ReactorOAuthCredentialEventCallback** also uses **userName** when returning sensitive information. **Required** except when specifying sensitive information in the **ReactorOAuthCredentialEventCallback**.
password	**Required**. A **Buffer** that specifies the password, which is sent with the **userName** to get an access token and a refresh token.
newPassword	Conditional. A **Buffer** that specifies the new password when changing the password associated with the specified **userName**. Include newPassword only when the application wants to change its password, in which case both the current (**password**) and new password (**newPassword**) are required.
clientId	A **Buffer** that specifies the unique Client ID for the application that makes the request.
clientSecret	A **Buffer** that specifies the client secret (if one exists) used by the OAuth client to authenticate to the authorization Server.
tokenScope	A **Buffer** that specifies the scope of the generated token.

Table 85: RsslReactorOAuthCredentialRenewal Members

6.9.2.4 ReactorOAuthCredentialRenewal Options

OPTION	DESCRIPTION
reactorAuthTokenEventCallback	A callback function (**reactorAuthTokenEventCallback**) 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.
proxyDomain	A **Buffer** that specifies the domain for authenticated proxies.
proxyHostName	A **Buffer** that specifies the proxy's host name.
proxyPasswd	A **Buffer** that specifies the password for authenticated proxies.
proxyPort	A **Buffer** that specifies the proxy's port.
proxyUserName	A **Buffer** that specifies the username for authenticated proxies.
renewalMode	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 6.9.2.5.

Table 86: ReactorOAuthCredentialRenewal Options

6.9.2.5 ReactorOAuthCredentialRenewalOptions.RenewalModes Class

MODE	DESCRIPTION
PASSWORD	Use this renewal mode when normally submitting a password to obtain an access and refresh token.
PASSWORD_CHANGE	Use this renewal mode only when changing the application's password.

Table 87: ReactorOAuthCredentialRenewalMode Enums

6.10 JSON to RWF Protocol Conversion for WebSocket Support

For the consumer to support WebSocket connection requests, you must initialize the consumer and define **`WSocketOpts.protocols`**, based on the types of protocol you want to support, as follows:

- To support WebSocket connections for only the RWF sub-protocol, use **`ConnectOptions.wSocketOpts.protocols("rssl.rwf")`**.
  - This is the only requirement. For an example of setting **`wSocketOpts`**, refer to Section 6.10.1.1.
  - For remaining tasks for managing the Consumer, refer to Chapter 3, Building an OMM Consumer.
- To support WebSocket connection requests for JSON2 and RWF sub-protocols (either alone, or in tandem with RWF), use **`ConnectOptions.wSocketOpts.protocols("rssl.rwf, rssl.json.v2, tr_json2")`**.

For details on **`WSocketOpts`**, refer to the *Enterprise Transport API Java Edition Developer Guide*.

To support WebSocket connections in JSON (either alone, or in tandem with RWF), after initializing the Interactive Provider, you must then perform the following:

- Connect the Consumer over an encrypted WebSocket. For an example, refer to Section 6.10.1.2.
- Define the Reactor JSON Converter event callback by using the **`JsonConverterOptions.jsonConversionEventCallback()`** function. For an example, refer to Section 6.10.1.3.
- Define the **`ServiceName`** to **`ServiceId`** callback (using the **`JsonConverterOptions.serviceNameToIdCallback()`** function). For an example, refer to Section 6.10.1.3.
- Clear and populate a JSON Converter options object using the callbacks from the preceding tasks. For an example, refer to Section 6.10.1.4.
- Initialize the Reactor JSON Converter using the **`Reactor.initJsonConverter()`** function passing in the JSON converter options **`ReactorJsonConverterOptions`**. For an example, refer to Section 6.10.2.4.
- For remaining tasks on managing the Consumer, refer to Chapter 3, Building an OMM Consumer.

6.10.1 Consumer and WebSocket Support

6.10.1.1 Example: Initializing the Consumer to Support WebSocket Connections for RWF Only

```
ConnectOptions cOpt = chnlInfo.connectOptions.connectionList().get(0).connectOptions();
cOpt.connectionType(ConnectionTypes.WEBSOCKET);
cOpt.wSocketOpts().protocols("rssl.rwf");
```

6.10.1.2 Example: Connecting the Consumer over an Encrypted WebSocket

```
ConnectOptions cOpt = chnlInfo.connectOptions.connectionList().get(0).connectOptions();
cOpt.connectionType(ConnectionTypes.ENCRYPTED);
cOpt.encryptionOptions().connectionType(ConnectionTypes.WEBSOCKET);
cOpt.wSocketOpts().protocols("rssl.rwf");
```

6.10.1.3 Example: Defining Reactor JSON Converter Event Callback and ServiceName to ServiceId Callback

```
public class JsonConversionEventCallbackImpl implements ReactorJsonConversionEventCallback {

    @Override
    public int reactorJsonConversionEventCallback(ReactorJsonConversionEvent jsonConversionEvent)
    {
        System.out.println("JSON Conversion error: " + jsonConversionEvent.error().text());

        return ReactorCallbackReturnCodes.SUCCESS;
    }
}

public class ServiceNameIdCallbackImpl implements ReactorServiceNameToIdCallback {

    @Override
    public int reactorServiceNameToIdCallback(ReactorServiceNameToId serviceNameToId,
            ReactorServiceNameToIdEvent serviceNameToIdEvent) {

        ChannelInfo chnlInfo = (ChannelInfo)serviceNameToIdEvent.reactorChannel().userSpecObj();

        /* Checks whether the service name is used by the channel. */
        if(chnlInfo.serviceInfo.checkHasInfo() && serviceNameToId.serviceName()
                .equals(chnlInfo.serviceInfo.info().serviceName().toString()))
        {
            serviceNameToId.serviceId(chnlInfo.serviceInfo.serviceId());
            return CodecReturnCodes.SUCCESS;
        }
        else
        {
            return CodecReturnCodes.FAILURE;
        }
    }
}
```

6.10.1.4 Example: Clear and Populate a JSON Converter Options Structure

```
ReactorJsonConverterOptions jsonConverterOptions = ReactorFactory.createReactorJsonConverterOptions();

jsonConverterOptions.clear();
jsonConverterOptions.dataDictionary(dictionary);
jsonConverterOptions.serviceNameToIdCallback(new ServiceNameIdCallbackImpl());
jsonConverterOptions.jsonConversionEventCallback(new JsonConversionEventCallbackImpl());
```

6.10.1.5 Example: Initialize the Reactor JSON Converter

```
// Initialize the JSON converter
if ( reactor.initJsonConverter(jsonConverterOptions, errorInfo) != ReactorReturnCodes.SUCCESS)
{
    System.out.println("Reactor.initJsonConverter() failed: " + errorInfo.toString());
    System.exit(ReactorReturnCodes.FAILURE);
}
```

6.10.2 Interactive Provider and WebSocket Support

For the Interactive Provider to support WebSocket connection requests, you must initialize the Interactive Provider and define **`WSocketOpts.protocols`**, based on the types of protocol you want to support, as follows:

- To support WebSocket connections for only the RWF sub-protocol, use **`BindOptions.wSocketOpts().protocols("rssl.rwf")`**.
  - This is the only requirement. For an example of setting **`wSocketOpts`**, refer to Section 6.10.2.1.
  - For remaining tasks for managing the Interactive Provider, refer to Chapter 4, Building an OMM Interactive Provider.
- To support WebSocket connection requests for both JSON2 and RWF sub-protocols, use **`BindOptions.wSocketOpts().protocols("rssl.rwf, rssl.json.v2, tr_json2")`**.
- To support WebSocket connection requests only JSON2 sub-protocol, use **`BindOptions.wSocketOpts().protocols("rssl.json.v2, tr_json2")`**.

For details on **`WSocketOpts`**, refer to the *Enterprise Transport API Java Edition Developer Guide*.

To support the JSON protocol (either alone, or in tandem with RWF), after initializing the Interactive Provider, you must then perform the following:

- Define the Reactor JSON Converter event callback by using the **`JsonConverterOptions.jsonConversionEventCallback()`** function. For an example, refer to Section 6.10.2.2.
- Define the **`ServiceName`** to **`ServiceId`** callback (using the **`JsonConverterOptions.serviceNameToIdCallback()`** function). For an example, refer to Section 6.10.2.2.
- Clear and populate a JSON converter options object using the callbacks from the preceding tasks. For an example, refer to Section 6.10.2.3.
- Initialize the Reactor JSON Converter using the **`Reactor.initJsonConverter()`** function passing in the JSON converter options **`ReactorJsonConverterOptions`**. For an example, refer to Section 6.10.2.4.
- For remaining tasks on managing the Interactive Provider, refer to Chapter 4, Building an OMM Interactive Provider.

6.10.2.1 Example: Initializing the Publisher to Support WebSocket Connections for RWF Only

To support WebSocket connections for only the RWF sub-protocol, initialize the **`Server`** (for details, refer to Section 4.3) with **`bindOptions.wSocketOpts().protocols("rssl.rwf")`**.

```
bindOptions.majorVersion(Codec.majorVersion());
bindOptions.minorVersion(Codec.minorVersion());
bindOptions.protocolType(Codec.protocolType());
bindOptions.serviceName(portNo);
bindOptions.wSocketOpts().protocols("rssl.rwf");
```

6.10.2.2 Example: Defining Reactor JSON Converter Event Callback and ServiceName to ServiceId Callback

```
@Override
    public int reactorJsonConversionEventCallback(ReactorJsonConversionEvent jsonConversionEvent)
    {
        System.out.println("JSON Conversion error: " + jsonConversionEvent.error().text());

        return ReactorCallbackReturnCodes.SUCCESS;
    }

public class ServideNameIdCallbackImpl implements ReactorServiceNameToIdCallback {

    @Override

    public int reactorServiceNameToIdCallback(ReactorServiceNameToId serviceNameToId,
            ReactorServiceNameToIdEvent serviceNameToIdEvent)
    {
        if(directoryHandler.serviceName().equals(serviceNameToId.serviceName()))
            {
                serviceNameToId.serviceId(directoryHandler.serviceId());
                return ReactorReturnCodes.SUCCESS;
            }

        return ReactorReturnCodes.FAILURE;
    }
}
```

6.10.2.3 Example: Clear and Populate a JSON Converter Options Structure with Callback Information

```
ReactorJsonConverterOptions jsonConverterOptions = ReactorFactory.createReactorJsonConverterOptions();

jsonConverterOptions.clear();
jsonConverterOptions.dataDictionary(dictionary);
jsonConverterOptions.serviceNameToIdCallback(new ServiceNameIdCallbackImpl());
jsonConverterOptions.jsonConversionEventCallback(new JsonConversionEventCallbackImpl());
jsonConverterOptions.defaultServiceId(1);
```

6.10.2.4 Example: Initialize the Reactor JSON Converter Using the initJsonConverter() Function

```
if ( reactor.initJsonConverter(jsonConverterOptions, errorInfo) != ReactorReturnCodes.SUCCESS)
{
    System.out.println("Reactor.initJsonConverter() failed: " + errorInfo.toString());
    System.exit(ReactorReturnCodes.FAILURE);
}
```

6.10.3 ReactorJsonConverterOptions

The **ReactorJsonConverterOptions** structure includes the following options:

MEMBER	DESCRIPTION
catchUnknownJsonFids	When converting from JSON to RWF, sets the Enterprise Transport API to catch unknown JSON field IDs. **catchUnknownJsonFids** is a **boolean** and defaults to **true**.
catchUnknownJsonKeys	When converting from JSON to RWF, sets the Enterprise Transport API to catch unknown JSON keys. **catchUnknownJsonKeys** is a **boolean** and defaults to **false**.
closeChannelFromFailure	A **boolean** that closes the channel if the Reactor fails to parse a JSON message or if the Reactor receives a JSON error message. **closeChannelFromFailure** defaults to **true**.
defaultServiceId	If both **ServiceName** and **ServiceID** are not set, **defaultServiceId** specifies a default service ID for requests. **defaultServiceId** defaults to **-1** (i.e., not set). **defaultServiceId** accepts a valid range of **0** to **65535**.
jsonExpandedEnumFields	A **boolean** that expands enumerated values in field entries to their display values for the JSON protocol. **jsonExpandedEnumFields** defaults to **false** (do not expand the field).
dataDictionary	Sets the data dictionary (**DataDictionary**) that Enterprise Transport API uses to initialize the RWF/JSON converter. **dataDictionary** defaults to **null**.

Table 88: ReactorJsonConverterOptions Class Members

MEMBER	DESCRIPTION
JsonConversionEventCallback	Specifies the callback function (**ReactorJsonConversionEventCallback**) that receives the **ReactorJsonConversionEvent** if the JSON converter fails to convert a message. **JsonConversionEventCallback** defaults to **null**. • For further details on **ReactorJsonConversionEventCallback**, refer to Section 6.10.4. • For further details on **ReactorJsonConversionEvent**, refer to Section 6.10.7.
serviceNameToIdCallback	Specifies the callback function (**ReactorServiceNameToIdCallback**) that handles conversion of the **ServiceName** to **ServiceId**. **serviceNameToIdCallback** defaults to **null**. For further details on **ReactorServiceNameToIdCallback**, refer to Section 6.10.5.
userSpec	Specifies a user-defined pointer which is retrieved in the callback function. **userSpec** is an object type and defaults to **null**.

Table 88: ReactorJsonConverterOptions Class Members (Continued)

6.10.4 ReactorJsonConversionEventCallback Function

The **ReactorJsonConversionEventCallback** function communicates conversion information when the JSON converter fails to convert JSON to RWF messages by providing an **ReactorJsonConversionEvent** object to the application. For further details on **ReactorJsonConversionEvent**, refer to Section 6.10.7.

6.10.5 ReactorServiceNameToIdCallback Function

ReactorServiceNameToIdCallback calls back to the application to translate a **ServiceName** to **ServiceId** by application by providing an **ReactorServiceNameToIdEvent**. For further details on **ReactorServiceNameToIdEvent**, refer to Section 6.10.6.

- If **ReactorServiceNameToIdCallback** succeeds, it returns **ReactorReturnCodes.SUCCESS**.
- If **ReactorServiceNameToIdCallback** fails, it returns **ReactorReturnCodes.FAILURE**.

PARAMETER	DESCRIPTION
ReactorServiceNameToId	Carries the **serviceName** and **serviceId** fields. The application populates the **serviceId** field by looking up the appropriate **serviceName**.
ReactorServiceNameToIdEvent	**ReactorServiceNameToIdEvent** occurs when the Reactor needs to convert from a **serviceName** to a **serviceId**.

Table 89: ReactorServiceNameToIdCallback Parameters

6.10.6 ReactorServiceNameToIdEvent Class

The **ReactorServiceNameToIdEvent** class includes the following options:

MEMBER	DESCRIPTION
userSpecObj	Specifies a user-defined reference provided when specifying the callback for the **ReactorServiceNameToIdEvent** event. **userSpecObj** is an object type and defaults to **null**.

Table 90: ReactorServiceNameToIdEvent Class Members

6.10.7 ReactorJsonConversionEvent Class

The **ReactorJsonConversionEvent** class includes the following options:

MEMBER	DESCRIPTION
userSpec	Specifies a user-defined reference provided when specifying the callback for the **ReactorServiceNameToIdEvent** event. **userSpec** is an object type and defaults to **null**.
error	Contains any error information (**Error**) associated with a JSON conversion.

Table 91: ReactorJsonConversionEvent Class Members

6.11 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 the following Section 6.11.1 - Section 6.11.3.

6.11.1 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** object defined in Section 6.11.2. This method calls the Enterprise Transport API **Channel.info** method which has its use and return values described in the Enterprise Transport APIJava Edition *Developers Guide*.
ioctl	Allows the application to change various settings associated with the **ReactorChannel**. The available options are defined in Section 6.11.3. This method calls the **Channel.ioctl** method which has its use and return values described in the Enterprise Transport APIJava Edition *Developers Guide*.

Table 92: Reactor Utility Methods

6.11.2 ReactorChannelInfo Class Members

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

CLASS MEMBER	DESCRIPTION
channelInfo	Returns the underlying **Channel** information. This includes **maxFragmentSize**, number of output buffers, compression information, and more. The **ChannelInfo** method object is fully described in the Enterprise Transport APIJava Edition *Developers Guide*.

Table 93: ReactorChannelInfo Class Members

6.11.3 ReactorChannel.ioctl Option Values

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

6.11.4 Debugging for Reactor

The **Reactor** class provides the following methods for controlling debugging process.

METHOD	DESCRIPTION
void enableDebuggingLevel(int level)	Enables the provided debugging level. For the list of available debugging levels, see Section 6.2.1.4.

METHOD	DESCRIPTION
void disableDebuggingLevel(int level)	Disables the provided debugging level.
int debuggingLevels()	Returns the debugging levels currently set for this Reactor instance.
byte[] getDebuggingInfo()	Provides access to the messages debugged up to this point in case the underlying debugger stream is an instance of **`ByteArrayOutputStream`** (which is the case when, e.g., no output stream was provided in the **`ReactorDebuggerOptions`** while creating the **`Reactor`** instance. Returns a byte array that contains debugging messages (up to a certain capacity) currently written from the beginning of debugging or since the last call to this method. In case debugging is not enabled at the point when this method is called, one of the following can be returned: • Null in case the user has provided a debug output stream that is not an instance of ByteArrayOutputStream. • Otherwise, a byte array; if debugging was at some point enabled and this method has not been priorly called, the array will contain the previously logged messages.

Table 94: Reactor Methods for Debugging Handling

7 Consuming Data from the Cloud

7.1 Overview

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

- Credential Management (for details, refer to Section 7.3)
- Service Discovery (for details, refer to Section 7.6)
- Consuming Market Data (for details, refer to Section 7.7)
- Login Reissue (for details, refer to Section 7.4.3)

There are two versions of login credentials for the Refinitiv Data Platform:

- Version 1 Authentication also known as “V1 auth”, “oAuthPasswordGrant” or “V1 Password Credentials”: Uses the oAuth2.0 Password grant or Refresh Token grant. Requires a Machine Account consisting of username and password; also requires a client ID generated by the Refinitiv **AppGenerator**. For details, refer to Section 7.4.
- 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 7.5.

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 Refinitiv Real-Time as it functions in the cloud, refer to the *Refinitiv Real-Time - Optimized: Installation and Configuration for Client Use*.

7.2 Encrypted Connections

When connecting to a Refinitiv Real-Time Advanced Distribution Server in the cloud, you must use an encrypted connection type (for details on connection types, refer to the *ETA Java 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.

When configuring the **`rsslReactorChannel`**, 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 1 Authentication Using oAuth Password and Refresh_Token

7.4.1 Client_ID (AppKey)

To connect to Refinitiv Real-Time - Optimized infrastructure, the Enterprise Transport API requires a **Client_ID**, and optionally can include a client secret. **Client_ID**s are generated using **AppGenerator**, which refers to the **Client_ID** as an AppKey. Each user must obtain their unique **Client_ID** using the machine account email sent by Refinitiv, which includes a link to **AppGenerator**. Keep your **Client_ID** private: do not share **Client_ID**s.

- For further details on generating this ID, refer to the *Refinitiv Real-Time - Optimized: Installation and Configuration for Client Use* document. Each **Client_ID** is unique: do not share it with others.
- For details on how OAuth uses a Client Secret with a Client ID and their relationship, refer to OAuth documentation at: the following URL: https://www.oauth.com/oauth2-servers/client-registration/client-id-secret/.

7.4.2 Obtaining Initial Access and Refresh Tokens

To obtain an access token, the RTSDK API sends its username, **Client_ID** (from **`ConsumerRole`** as described in Section 6.3.2.1), and password (defined in the Login Domain, as described in Section 8.3) in a single message to the Refinitiv Data Platform. You must configure these details (in the **`OmmConsumerConfig`** object) before executing a calls connect (for details on the **`Reactor.connect`** method, refer to Section 6.4.1.1).

In response, the Refinitiv Data Platform sends an access token, its expiration timeout (by default: 300 seconds), and a refresh token for use in the login reissue process (for details on the expiration timeout and login reissue process, refer to Section 7.4.3). The API must obtain an access token before executing a service discovery or obtaining market data.

The following diagram illustrates the process by which the RTSDK API obtains its tokens:

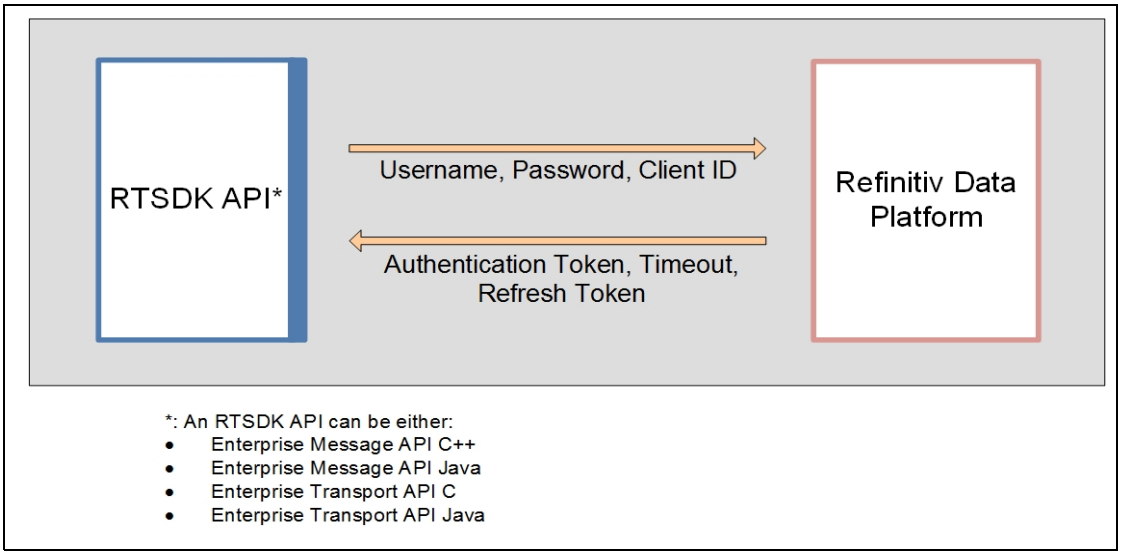


Figure 9. Obtaining an Authentication Token

7.4.3 Refreshing the Access Token and Sending a Login Reissue

In response to the API's token request, the Refinitiv Data Platform sends an access token and a refresh token, both with associated expiration timeouts which set the length of time for which the token is valid. If the Refinitiv Real-Time Advanced Distribution Server does not receive a new access token before the end of the expiration timeout, the Refinitiv Real-Time Advanced Distribution Server sends a login close status message and closes the connection.

To create a seamless experience for API users, the API sends the refresh token to proactively obtain a new access token prior to the published expiration timeout. The Enterprise Transport API calculates the time at which it requests a new access token by multiplying the token's published timeout by 4/5 (i.e., **0.8**). Thus, if the default is 300 seconds, the API requests a new access token after 240 seconds. You can configure this reissue ratio using **`ReactorOptions.tokenReissueRatio`** (for details, refer to Section 6.2.1.2).

In response to receiving a refresh token, the Refinitiv Data Platform sends a new access token with an associated timeout to the API. After receiving the new access token from the Refinitiv Data Platform, the API renews its connection by sending a Login Reissue with the new access token to the Refinitiv Real-Time Advanced Distribution Server. The process of renewing the access token and refreshing the Refinitiv Real-Time Advanced Distribution Server connection via a Login Reissue continues until the refresh token itself expires (which can take several hours or days). When using a **grant_type** of **refresh_token**, if the value for **expires_in** does not match the **expires_in** received from when the API obtained the **refresh_token** (i.e., when **grant_type** was **password**), this is an indication that the **refresh_token** is about to expire. In this case, the API will obtain a new set of both refresh and access tokens as described in Section 7.4.2.

The login reissue process is illustrated in the following diagram:

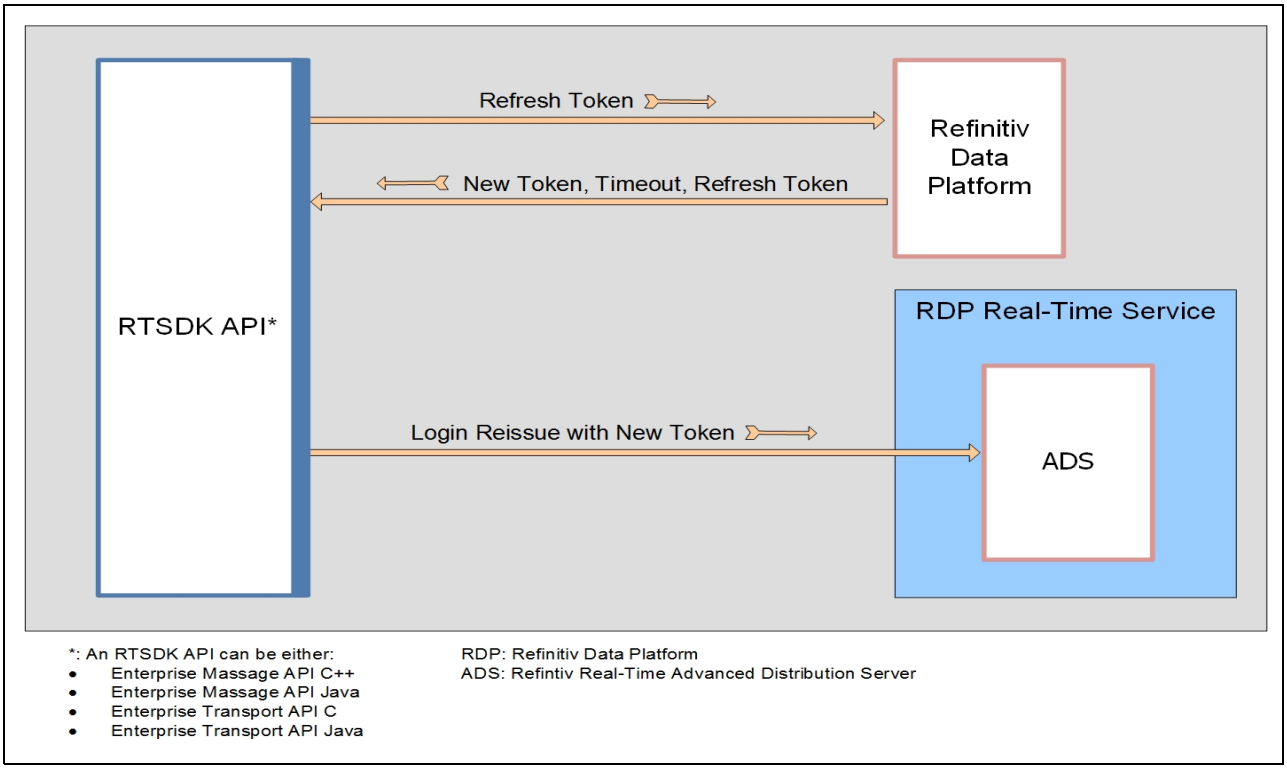


Figure 10. Login Reissue

7.4.4 Session Management per User Credential

Prior to Version 3.6.0, the Enterprise Transport API would manage tokens separately across each channel, even when using the same Username, Client ID, and password credentials. So that each channel had a unique pair of access and refresh tokens. API would manage each channel distinct from the others.

As of Version 3.6.0, the Enterprise Transport API connects to the Refinitiv Data Platform once and reuses the same access and refresh tokens for all channels. The Enterprise Transport API supports up to, but no more than, 5 channels per OAuth credential set.

7.5 Version 2 Authentication Using oAuth Client Credentials

Version 2 oAuth Client Credentials requires a client ID and client secret. Version 2 will only generate an Access Token, not both Access and Refresh Token.

Once connected to the Refinitiv Real-Time Optimized ADS, there is no need to renew the Access Token. The login session to the ADS will remain valid until the consumer disconnects or is disconnected from RTO. The API will only re-request an Access Token in the following scenarios:

- When the consumer disconnects and goes into a reconnection state.
- If the **`reactorChannel`** 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.

7.5.1 Configuring and Managing Version 2 Credentials

The client ID and client secret must be set on the **`RsslReactorOAuthCredential`** as described in Section 6.10.2.1Section 6.9.2.1 of the *Enterprise Transport API Java Edition Value Added Developers Guide*. The **`Reactor`** will handle the credentials the same way as Version 1, with an **`RsslReactorOAuthCredentialEvent`** callback for credentials if the user does not wish for the **`Reactor`** to store them.

7.5.2 Version 2 oAuth Client Credentials Token Lifespan

Unlike Version 1, Version 2 will only 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 Refinitiv Real-Time - Optimized (RTO).

Once connected, the API does not need to periodically renew a token.

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.

7.6 Service Discovery

After obtaining a token (for details, refer to Section 7.4.2), the Enterprise Transport API can perform a service discovery against the Refinitiv Data Platform to obtain connection details for the Refinitiv Real-Time - Optimized. The Enterprise Transport API Java Edition uses the **`Reactor.queryServiceDiscovery`** method (refer to Section 6.2.1 for a description of this reactor method) to submit a service discovery.

In response to a service discovery, the Refinitiv Data Platform returns transport and data format protocols and a list of hosts and associated ports for the requested service(s) (i.e., a Refinitiv Real-Time Advanced Distribution Server running in the cloud or endpoint). Refinitiv 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 Refinitiv Data Platform, the Enterprise Transport API identifies an RTO 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.

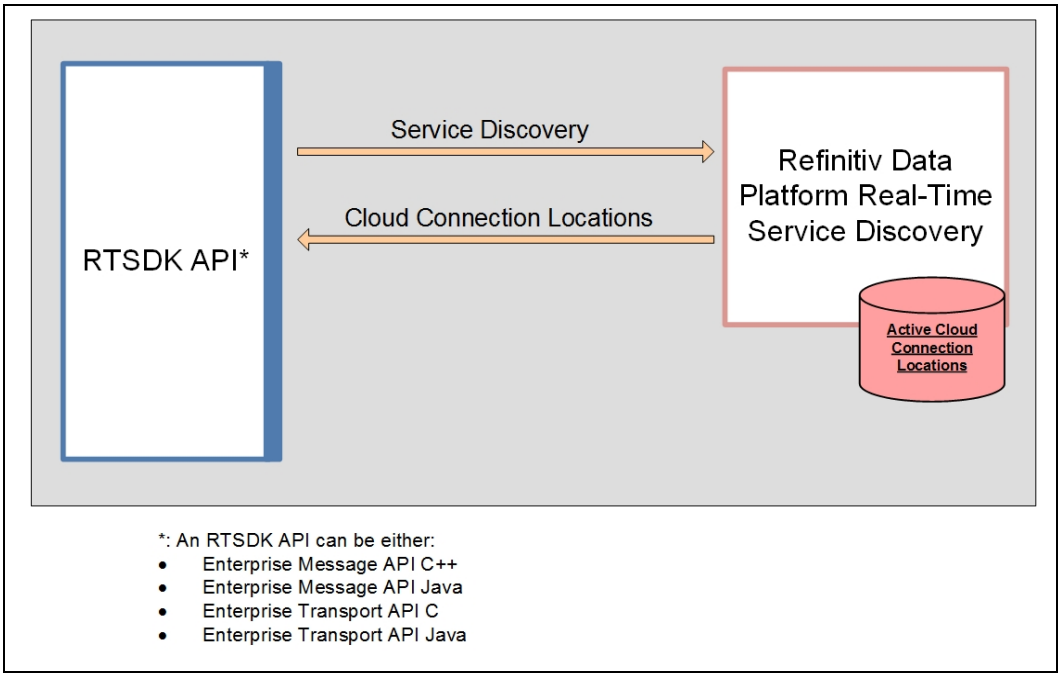
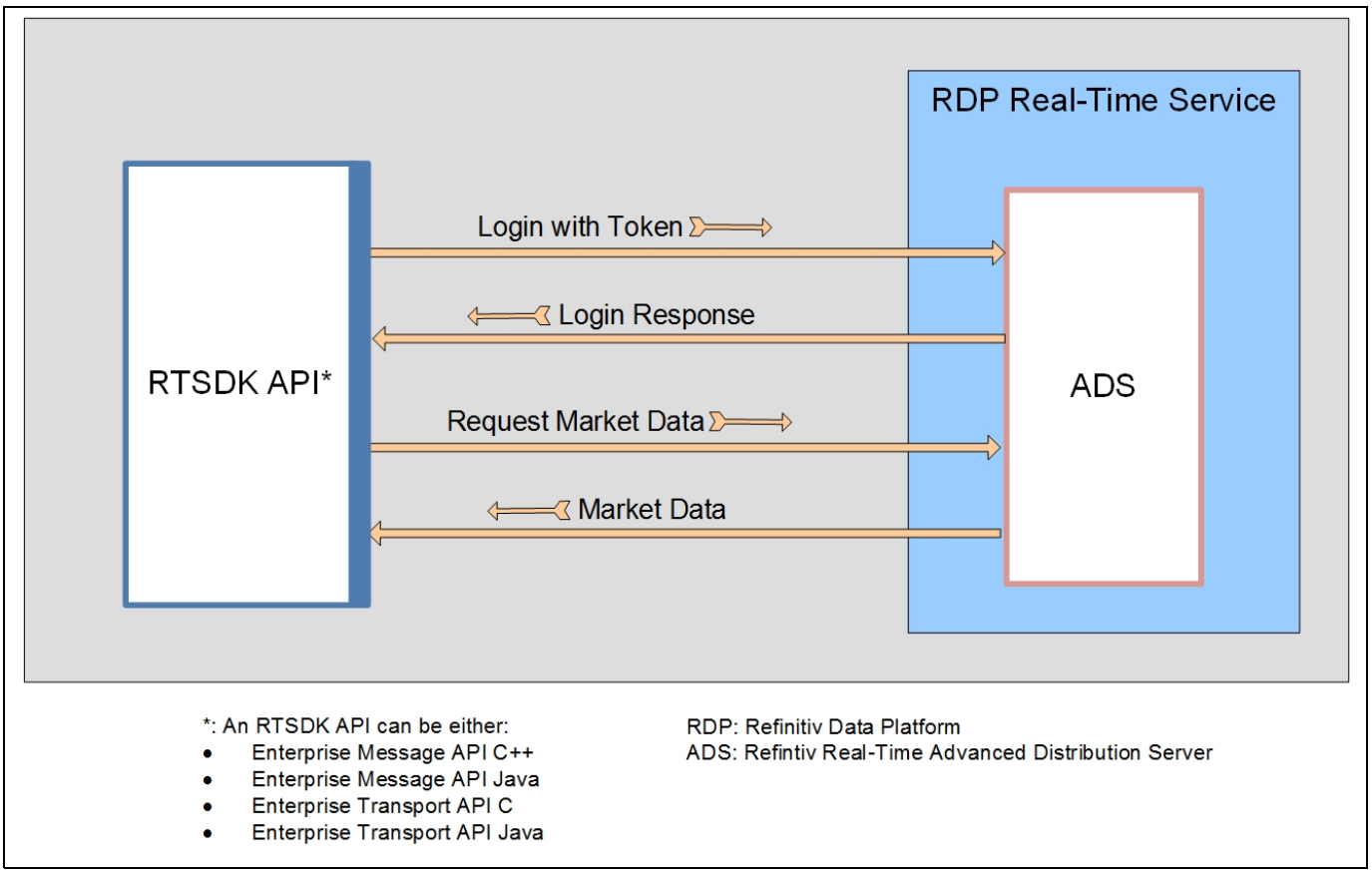


Figure 11. Service Discovery

7.7 Consuming Market Data

After obtaining its login token (for details, refer to Section 7.4.2) and running a service discovery (for details, refer to Section 7.6), the API can connect to the Refinitiv Real-Time Advanced Distribution Server in the cloud and obtain market data. While consuming market data, the API must periodically renew its token via the login reissue workflow (for details, refer to Section 7.4.3).



7.8 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:
  - For Version 1 authentication, retry with username and password for error codes 400 and 401
  - 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

7.9 Cloud Connection Use Cases

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

- Start to finish session management (for details, refer to Section 7.9.1)
- Disabling the watchlist (for details, refer to Section 7.9.2)
- Query service discovery (for details, refer to Section 7.9.3)

7.9.1 Session Management Use Case

In the session management use case, the Enterprise Transport API manages the entire connection from start to finish. To use session management, you need to configure the API to enable the watchlist and session management (i.e., in the **`ReactorConnectInfo`** object, set **enableSessionManagement**).

The API exhibits the following behavior (listed in order) when operating in a session management use case:

- Obtains a token (according to the details in Section 7.4.2)
- Queries service discovery (according to the details in Section 7.6)
- Consumes market data (according to the details in Section 7.7)
- Manages login reissues for Version 1 authentication when needed on a cyclical basis (according to the details in Section 7.4.3)

A special use case exists for connecting to a specific (i.e., non-default) host. As described in Section 7.6, by default the Enterprise Transport API connects to whichever host is setup for failover in the location specified by the API. If you want to connect to a specific, non-default host, you must set this in the ***`UnifiedNetworkInfo`*** options. In this case, the Enterprise Transport API exhibits the same behavior listed above, but ignores the endpoints it receives from the service discovery.

7.9.2 Disabling the Watchlist

When connecting to a Refinitiv 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 7.4.2)
- If needed, queries service discovery (according to the details in Section 7.6)

The 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**).

7.9.3 Query Service Discovery

In the query service discovery use case, the API user wants to connect to the Refinitiv Data Platform only for a service discovery, and does not necessarily want to consume market data. The API exhibits the following behavior (listed in order) when operating in a query service discovery use case:

- Obtains a token (according to the details in Section 7.4.2)
- Queries service discovery (according to the details in Section 7.6)

7.10 Logging of Authentication and Service Discovery Interaction

If needed, you can log the interactions with the Refinitiv Data Platform. To enable logging, you must activate the logging section in the **build.gradle** file in the .../**PackageDirectory/Ema/Examples** directory, as described in the *RTSDK Java Edition Installation Guide*.

You can configure the location of the logging file (**logging.properties**), by editing the **java.util.logging.config.file** parameter.

To turn on logging, add the following text to the **logging.properties** file:

```
com.refinitiv.eta.valueadd.reactor.RestReactor.level=FINEST

java.util.logging.ConsoleHandler.level=FINEST
```

7.10.1 Logged Request Information

With logging turned on in the fashion mentioned in Section 7.10, the Enterprise Transport API writes the following request information in the log:

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

```
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
```

7.10.2 Logged Response Information

With logging turned on in the fashion mentioned in Section 7.10, 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 95). This component also handles all encoding and decoding functionality for these domain models, so the application needs only to manipulate the message's object members to send or receive content. Such functionality significantly reduces the amount of code an application needs to interact with OMM devices (i.e., Refinitiv 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 class 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 **`ElementList`**, the elements are represented with Java JDK collections.

The Enterprise Transport APIJava Edition *Refinitiv 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, Refinitiv recommends that provider applications support the domain's use. The Dictionary domain is considered an administrative domain. Many Refinitiv 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 Refinitiv 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. Refinitiv strongly recommends that consumers request this domain. Source Directory is considered an administrative domain, and many Refinitiv components expect this content and require it to conform to the domain model definition. For further details, refer to Section 8.4.

Table 95: Domains Representations in the Administration Domain Model Value Added Component

8.2 Message Base

MsgBase is the root interface 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** interface 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 96: MsgBase Members

8.2.2 Message Base Method

METHOD	DESCRIPTION
clear()	Clears the object for reuse.

Table 97: MsgBase Method

8.2.3 RDM Message Types

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

DOMAIN TYPE	DOMAIN REPRESENTATIONS MESSAGE TYPE	DOMAIN REPRESENTATION INTERFACE
LOGIN (**`LoginMsg`**) Refer to Section 8.3	LoginMsgType.REQUEST	LoginRequest
	LoginMsgType.REFRESH	LoginRefresh
	LoginMsgType.STATUS	LoginStatus
	LoginMsgType.CLOSE	LoginClose
	LoginMsgType.CONSUMER_CONNECTION_STATUS	LoginConsumerConnectionStatus
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
DICTIONARY (**`DictionaryMsg`**) Refer to Section 8.5	DictionaryMsgType.REQUEST	DictionaryRequest
	DictionaryMsgType.REFRESH	DictionaryRefresh
	DictionaryMsgType.STATUS	DictionaryStatus
	DictionaryMsgType.CLOSE	DictionaryClose

Table 98: Domain Representations Message Types

8.3 RDM Login Domain

The Login domain registers (or authenticates) a user with the system, after which the user can request[1], post[2], or provide[3] OMM content. A Login request may also be used to authenticate a user with the system.

- 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 APIJava Edition *Refinitiv Domain Model Usage Guide*.

8.3.1.1 Login Request Members

MEMBER	DESCRIPTION
attrib	**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 **RDM_LG_RQF_HAS_AUTHN_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 to Section 8.3.1.2.

Table 99: `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.
rdmMsgBase	**Required**. Specifies the login message type (i.e., **LoginMsgType.REQUEST** for the login request).
role	**Optional**. If present, a **`flags`** value of **LoginRequestFlags.HAS_ROLE** should be specified. If absent, a default value of **Login.Role.CONS** is assumed. Indicates the role of the application logging onto the system. • **0**: **Login.Role.CONS**, indicates application is a consumer. • **1**: **Login.Role.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 **USER_NAME** is assumed. Possible values: • **Login.UserIdTypes.NAME = 1** • **Login.UserIdTypes.EMAIL_ADDRESS = 2** • **Login.UserIdTypes.TOKEN = 3** • **Login.UserIdTypes.COOKIE = 4** • **Login.UserIdTypes.AUTHN_TOKEN==5** A type of **Login.UserIdTypes.NAME** typically corresponds to a Data Access Control System user name and can to authenticate and permission a user. **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. **Login.UserIdTypes.AUTHN_TOKEN** is specified when using Refinitiv 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 *Refinitiv Real-Time Distribution System Authentication User Manual*.[a]

Table 99: `LoginRequest` Members (Continued)

a. For further details on Refinitiv Real-Time Distribution System Authentication, refer to the *Refinitiv Real-Time Distribution System Authentication User Manual*, accessible on MyRefinitiv 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 **attrib**.
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 **Login.Role.CONS** is assumed.
LoginRequestFlags.HAS_USERNAME_TYPE	Indicates the presence of **userNameType**. If not present, a **userNameType** of **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 APIJava Edition *Developers Guide*.

Table 100: `LoginRequest` Flags

8.3.1.3 Login Request Utility Method

METHOD NAME	DESCRIPTION
initDefaultRequest	Clears a **LoginRequest** object and populates **userName**, **position**, **applicationId**, and **applicationName** with default values.

Table 101: `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 APIJava Edition *Refinitiv Domain Model Usage Guide*.

8.3.2.1 Login Refresh Members

MEMBER	DESCRIPTION
attrib	**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 a Refinitiv Real-Time Distribution System Authentication environment, where **0** indicates an error-free condition. For further information, refer to the *Refinitiv 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 *Refinitiv 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 *Refinitiv 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 *Refinitiv Real-Time Distribution System Authentication User Manual*.[a]
connectionConfig	**Optional.** Indicates the connection configuration that the consumer uses for its standby servers when setup for Warm Standby. If present, a **`flags`** value of **LoginRefreshFlags.HAS_CONN_CONFIG** should be specified.
features	**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.
rdmMsgBase	**Required**. Specifies the login message type (i.e., **LoginMsgType.REFRESH** for login refresh).

Table 102: `LoginRefresh` Members

MEMBER	DESCRIPTION
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.
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 APIJava 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 **Login.UserIdTypes.NAME** is assumed. Possible values: • **Login.UserIdTypes.NAME = 1** • **Login.UserIdTypes.EMAIL_ADDRESS = 2** • **Login.UserIdTypes.TOKEN = 3** • **Login.UserIdTypes.COOKIE==4** • **Login.UserIdTypes.AUTHN_TOKEN==5** A type of **Login.UserIdTypes.NAME** typically corresponds to a Data Access Control System user name and can be used to authenticate and permission a user. **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. **Login.UserIdTypes.AUTHN_TOKEN** is specified when using Refinitiv 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 *Refinitiv Real-Time Distribution System Authentication User Manual*.[a]

Table 102: `LoginRefresh` Members (Continued)

a. For further details on Refinitiv Real-Time Distribution System Authentication, refer to the *Refinitiv Real-Time Distribution System Authentication User Manual*, accessible on MyRefinitiv 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 **`attrib`**.
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`**.

Table 103: `LoginRefresh` Flags

FLAG ENUMERATION	DESCRIPTION
LoginRefreshFlags.HAS_AUTHN_TT_REISSUE	Indicates the presence of **`authenticationTTReissue`**.
LoginRefreshFlags.HAS_CONN_CONFIG	Indicates the presence of connection configuration information used with warm standby functionality.
LoginRefreshFlags.HAS_FEATURES	Indicates the presence of **`features`**. If absent, a value of **0** is assumed.
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 **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 103: `LoginRefresh` Flags (Continued)

8.3.2.3 Login Support Feature Set Members

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

MEMBER	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.
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.

Table 104: `LoginSupportFeatures` Members

MEMBER	DESCRIPTION
supportProviderDictionaryDownload	**Optional**. Indicates whether the non-interactive provider can request dictionaries from the Refinitiv Real-Time Advanced Data Hub: • **1**: The non-interactive provider can request dictionaries from the Refinitiv Real-Time Advanced Data Hub. • **0**: The non-interactive provider cannot request dictionaries from the Refinitiv Real-Time Advanced Data 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 104: `LoginSupportFeatures` Members

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 Java 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.

Table 105: `LoginSupportFeaturesFlags`

FLAG ENUMERATION	MEANING
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 105: LoginSupportFeaturesFlags (Continued)

8.3.2.5 Login Connection Config Members

MEMBER	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 106: LoginConnectionConfig Members

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 107: LoginConnectionConfig Methods

8.3.2.7 Server Info Members

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 **Login.ServerTypes**.

Table 108: 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 109: `ServerInfo` Flags

8.3.2.9 Server Info Methods

METHOD NAME	DESCRIPTION
clear	Clears a **`ServerInfo`** object. Useful for object reuse.
copy	Performs a deep copy of a **`ServerInfo`** object.

Table 110: `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 APIJava Edition *Refinitiv Domain Models Usage Guide*.

8.3.3.1 Login Status Members

MEMBER	DESCRIPTION
authenticationErrorCode	**Optional**. If present, a **`flags`** value of **LoginStatusFlags.HAS_AUTHN_ERROR_CODE** should be specified. **`authenticationErrorCode`** is specific to deployments using Refinitiv Real-Time Distribution System Authentication, and specifies an error code. A code of **0** indicates no error condition. For further information, refer to the *Refinitiv 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 *Refinitiv 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.
rdmMsgBase	**Required**. Specifies the login message type (i.e., **LoginMsgType.STATUS** for login status).
state	**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 APIJava Edition *Developers Guide*.

Table 111: `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 **Login.UserIdTypes.NAME** is assumed. Possible values: • **Login.UserIdTypes.NAME = 1** • **Login.UserIdTypes.EMAIL_ADDRESS = 2** • **Login.UserIdTypes.TOKEN = 3** • **Login.UserIdTypes.COOKIE = 4** • **Login.UserIdTypes.AUTHN_TOKEN = 5** A type of **Login.UserIdTypes.NAME** typically corresponds to a Data Access Control System user name and can be used to authenticate and permission a user. **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. **Login.UserIdTypes.AUTHN_TOKEN** is specified when using Refinitiv 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 *Refinitiv 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 111: LoginStatus Members (Continued)

a. For further details on Refinitiv Real-Time Distribution System Authentication, refer to the *Refinitiv Real-Time Distribution System Authentication User Manual*, accessible on MyRefinitiv 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 **Login.UserIdTypes.NAME** is assumed.

Table 112: 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.

MEMBER	DESCRIPTION
rdmMsgType	**Required**. Specifies the login message type (for a login close, this will be **LoginMsgType.CLOSE**).

Table 113: `LoginClose` Member

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 APIJava Edition *Refinitiv 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.
rdmMsgBase	**Required**. Indicates the Login Message type (for login connection status, set to **LoginMsgType.CONSUMER_CONNECTION_STATUS**).
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 114: `LoginConsumerConnectionStatus` 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 115: `LoginConsumerConnectionStatus` Flags

8.3.5.3 Login Warm Standby Info Members

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 APIJava Edition *Developers Guide*.
warmStandbyMode	**Required**. Indicates whether a server is active (**Login.ServerTypes.ACTIVE**) or standby (**Login.ServerTypes.SERVER**).

Table 116: `LoginWarmStandbyInfo` Members

8.3.5.4 Login Warm Standby Info Methods

METHOD NAME	DESCRIPTION
clear	Clears a **`LoginWarmStandbyInfo`** object for reuse.
copy	Performs a deep copy of a **`LoginWarmStandbyInfo`** object.

Table 117: `LoginWarmStandbyInfo` 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 Java Edition *RDM Usage Guide*.

8.3.6.1 Login Round Trip Time Members

MEMBER	DESCRIPTION
flags	**Required**. Indicates the presence of optional Round Trip Time members. For details, refer to Section 8.3.6.2.
rdmMsgBase	**Required**. Contains general message information, such as **`streamId`** and **`domainType`**.
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 118: Login Round Trip Time Members

8.3.6.2 Login Round Trip Time Flag Enumeration Values

FLAG ENUMERATION	DESCRIPTION
HAS_TCP_RETRANS	Indicates the presence of the **tcpRetrans** member.
PROVIDER_DRIVEN	Indicates that this interaction was driven by the provider, not an interactive consumer.
ROUND_TRIP_LATENCY	Indicates the presence of the **rtLatency** member.

Table 119: 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 a **LoginRTT** object for reuse.
copy	Performs a deep copy of a **LoginRTT** object.

Table 120: 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 APIJava 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 APIJava 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 Members

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

MEMBER	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 121: LoginAttrib Members

MEMBER	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 Refinitiv Real-Time Advanced Data Hub: • **1**: The interactive provider can request dictionaries from the Refinitiv Real-Time Advanced Data Hub. • **0**: The interactive provider cannot request dictionaries from the Refinitiv Real-Time Advanced Data 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 121: `LoginAttrib` Members (Continued)

8.3.9.2 Login Attrib Methods

METHOD NAME	DESCRIPTION
clear	Clears a **LoginAttrib** object for reuse.
copy	Performs a deep copy of a **LoginAttrib** object.

Table 122: 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 123: LoginAttribFlags

8.3.10 Login Message

LoginMsg is the base interface for all Login messages. It is provided for use with general login-specific functionality. The following table summarizes different login messages.

INTERFACE	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 124: LoginMsg Interfaces

8.3.11 Login Message Utility Method

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

Table 125: 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.
encode	Encodes a Login message. This method takes the **EncodeIterator** as a parameter into which the encoded content is populated. Each login subinterface overrides this method to encode specific login message.

Table 126: Login Encoding and Decoding Methods

8.3.12.2 Encoding a Login Request

```
EncodeIterator encodeIter = CodecFactory.createEncodeIterator();
LoginRequest loginRequest = (LoginRequest)LoginMsgFactory.createMsg();

/* Clear the Login Request object. */
loginRequest.clear();

/* Set login message type - required as object created by LoginMsgFactory.createMsg() is generic Login */
/* object. */
loginRequest.rdmMsgType(LoginMsgType.REQUEST);

/* Set stream id. */
loginRequest.streamId(streamId);

/* Set flags indicating presence of optional members. */
loginRequest.applyHasAttrib();

/* Set UserName. */
loginRequest.userName().data("username");

/* Set ApplicationName */
loginRequest.attrib().applyHasApplicationName();
loginRequest.attrib().applicationName().data("upa");

/* Set ApplicationId */
loginRequest.attrib().applyHasApplicationId();
loginRequest.attrib().applicationId().data("256");

/* Set Position */
loginRequest.attrib().applyHasPosition();
loginRequest.attrib().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 = loginRequest.encode(encodeIter);
```

Code Example 19: Login Request Encoding Example

8.3.12.3 Decoding a Login Request

```
DecodeIterator decodeIter = CodecFactory.createDecodeIterator();
LoginRequest loginRequest = (LoginRequest)LoginMsgFactory.createMsg();
Msg msg = CodecFactory.createMsg();

/* Clear the decode iterator, set its RWF Version, and set it to the encoded buffer. */
decodeIter.clear();
ret = decodeIter.setBufferAndRWFVersion(msgBuf,channelMajorVersion, channelMinorVersion);

/* Decode the message to a Msg object. */
ret = msg.decode(decodeIter);

if (ret == CodecReturnCodes.SUCCESS && msg.domainType() == DomainTypes.LOGIN && msg.msgClass() ==
        MsgClasses.REQUEST)
{
    loginRequest.clear();
    loginRequest.rdmMsgType(LoginMsgType.REQUEST);

    ret = loginRequest.decode(decodeIter, msg);

    if(ret == CodecReturnCodes.SUCCESS)
    {
        /* Print username. */
        printf("Username: " + loginRequest.userName());

        if (loginRequest.checkHasAttrib())
        {
            LoginAttrib attrib = loginRequest.attrib();

            /* Print ApplicationName if present. */
            if(attrib.checkHasApplicationName())
                System.out.println("ApplicationName: " + attrib.applicationName().toString());

            /* Print ApplicationId if present. */
            if(attrib.checkHasApplicationId())
                System.out.println("ApplicationId: " + attrib.applicationId().toString());

            /* Print Position if present. */
            if(attrib.checkHasPosition())
                System.out.println("Position: " + attrib.position().toString());
        }
    }
}
```

Code Example 20: Login Request Decoding Example

8.3.12.4 Encoding a Login Refresh

```
EncodeIterator encodeIter = CodecFactory.createEncodeIterator();
LoginRefresh loginRefresh = (LoginRefresh)LoginMsgFactory.createMsg();

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

/* Set login message type - required as object created by LoginMsgFactory.createMsg() is generic Login */
/* object.*/
encodeIter.rdmMsgType(LoginMsgType.REFRESH);

/* Set stream id. */
loginRefresh.streamId(streamId);

/* Set flags indicating presence of optional members. */
loginRefresh.applyHasAttrib();
loginRefresh.applyHasUserName();

/* Set UserName. */
loginRefresh.userName().data("username");

/* Set ApplicationName */
loginRefresh.attrib().applyHasApplicationName();
loginRefresh.attrib().applicationName().data("upa");

/* Set ApplicationId */
loginRefresh.attrib().applyHasApplicationId();
loginRefresh.attrib().applicationId().data("256");

/* Set Position */
loginRefresh.attrib().applyHasPosition();
loginRefresh.attrib().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 21: Login Refresh Encoding Example

8.3.12.5 Decoding a Login Refresh

```
DecodeIterator decodeIter = CodecFactory.createDecodeIterator();
LoginRefresh loginRefresh = (LoginRefresh)LoginMsgFactory.createMsg();
Msg msg = CodecFactory.createMsg();

/* Clear the decode iterator, set its RWF Version, and set it to the encoded buffer. */
decodeIter.clear();
ret = decodeIter.setBufferAndRWFVersion(msgBuf,channelMajorVersion, channelMinorVersion);

/* Decode the message to a Msg object. */
ret = msg.decode(decodeIter);

if (ret == CodecReturnCodes.SUCCESS && msg.domainType() == DomainTypes.LOGIN && msg.msgClass() ==
        MsgClasses.REFRESH)
{
    loginRefresh.clear();
    loginRefresh.rdmMsgType(LoginMsgType.REFRESH);

    ret = loginRefresh.decode(decodeIter, msg);

    if(ret == CodecReturnCodes.SUCCESS)
    {
        /* Print username. */
        if(loginRefresh.checkHasUserName())
                printf("Username: " + loginRefresh.userName().toString());

        if (loginRefresh.checkHasAttrib())
        {
            LoginAttrib attrib = loginRefresh.attrib();

            /* Print ApplicationName if present. */
            if(attrib.checkHasApplicationName())
                System.out.println("ApplicationName: " + attrib.applicationName().toString());

            /* Print ApplicationId if present. */
            if(attrib.checkHasApplicationId())
                System.out.println("ApplicationId: " + attrib.applicationId().toString());

            /* Print Position if present. */
            if(attrib.checkHasPosition())
                System.out.println("Position: " + attrib.position().toString());
        }
    }
}
```

Code Example 22: 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 a Refinitiv Real-Time Advanced Data 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 APIJava Edition *Refinitiv Domain Model Usage Guide*.

8.4.1.1 Directory Request Members

MEMBER	DESCRIPTION
filter	**Required**. Indicates the service information in which the consumer is interested. The available flags are: • **Directory.ServiceFilterFlags.INFO = 0x01** • **Directory.ServiceFilterFlags.STATE = 0x02** • **Directory.ServiceFilterFlags.GROUP = 0x04** • **Directory.ServiceFilterFlags.LOAD = 0x08** • **Directory.ServiceFilterFlags.DATA = 0x10** • **Directory.ServiceFilterFlags.LINK = 0x20** In most cases, you should set the **Directory.ServiceFilterFlags.INFO**, **Directory.ServiceFilterFlags.STATE**, and **Directory.ServiceFilterFlags.GROUP**.
flags	**Required**. Indicates the presence of optional directory request members. For details, refer to Section 8.4.1.2.
rdmMsgBase	**Required**. Specifies the directory message type (a directory request would be **DirectoryMsgType.REQUEST**).
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 127: `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 128: `DirectoryRequest` Flags

8.4.1.3 Directory Request Methods

METHOD NAME	DESCRIPTION
clear	Clears a **`DirectoryRequest`** object. Useful for object reuse.
copy	Performs a deep copy of a **`DirectoryRequest`** object.

Table 129: `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 APIJava Edition *Refinitiv Domain Model Usage Guide*.

8.4.2.1 Directory Refresh Members

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: • **Directory.ServiceFilterFlags.INFO = 0x01** • **Directory.ServiceFilterFlags.STATE = 0x02** • **Directory.ServiceFilterFlags.GROUP = 0x04** • **Directory.ServiceFilterFlags.LOAD = 0x08** • **Directory.ServiceFilterFlags.DATA = 0x10** • **Directory.ServiceFilterFlags.LINK = 0x20**
flags	**Required**. Indicates the presence of optional directory refresh members. Refer to Section 8.4.2.2.
rdmMsgBase	**Required**. Specifies the type of directory message. For a directory refresh, send **DirectoryMsgType.REFRESH**.
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 list of information about available services.
state	**Required**. Indicates stream and data state information. For further details on **`State`**, refer to the Enterprise Transport APIJava Edition *Developers Guide*.

Table 130: `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 131: `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 APIJava Edition *Refinitiv Domain Model Usage Guide*.

8.4.3.1 Directory Update Members

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: • **Directory.ServiceFilterFlags.INFO = 0x01** • **Directory.ServiceFilterFlags.STATE = 0x02** • **Directory.ServiceFilterFlags.GROUP = 0x04** • **Directory.ServiceFilterFlags.LOAD = 0x08** • **Directory.ServiceFilterFlags.DATA = 0x10** • **Directory.ServiceFilterFlags.LINK = 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.
rdmMsgBase	**Required**. Specifies the message type. For a directory update, send **DirectoryMsgType.UPDATE**.

Table 132: `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 133: `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 APIJava Edition *Refinitiv Domain Model Usage Guide*.

8.4.4.1 Directory Status Members

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: • **Directory.ServiceFilterFlags.INFO = 0x01** • **Directory.ServiceFilterFlags.STATE = 0x02** • **Directory.ServiceFilterFlags.GROUP = 0x04** • **Directory.ServiceFilterFlags.LOAD = 0x08** • **Directory.ServiceFilterFlags.DATA = 0x10** • **Directory.ServiceFilterFlags.LINK = 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 APIJava Edition *Developers Guide*.
rdmMsgBase	**Required**. Specifies the message type. For a directory status, send **DirectoryMsgType.STATUS**.

Table 134: `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 135: `DirectoryStatus` Flags

8.4.4.3 Directory Status Methods

METHOD NAME	DESCRIPTION
clear	Clears an **DirectoryStatus** object. Useful for object reuse.
copy	Performs a deep copy of an **DirectoryStatus** object.

Table 136: 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.

MEMBER	DESCRIPTION
rdmMsgBase	**Required**. Specifies the directory message type. For a Directory Close message, set **rdmMsgBase** to **DirectoryMsgType.CLOSE**.

Table 137: DirectoryClose Member

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 APIJava Edition *Refinitiv Domain Model Usage Guide*.

8.4.6.1 Directory Consumer Status Members

MEMBER	DESCRIPTION
consumerServiceStatusList	**Optional**. Contains a list of **`ConsumerStatusService`** objects.
rdmMsgBase	**Required**. Specifies the directory message type. For a directory consumer status, set **`rdmMsgBase`** to **DirectoryMsgType.CONSUMER_STATUS**.

Table 138: `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 APIJava 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: • **Directory.SourceMirroringMode.ACTIVE_NO_STANDBY = 0**, • **Directory.SourceMirroringMode.ACTIVE_WITH_STANDBY = 1**, • **Directory.SourceMirroringMode.STANDBY = 2**

Table 139: `ConsumerStatusService` Members

8.4.7 Directory Service

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

The members of an **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 APIJava 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 140: 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 141: Service Flags

8.4.7.3 Service Methods

METHOD NAME	DESCRIPTION
clear	Clears a **Service** object. Useful for object reuse.
copy	Performs a deep copy of a **Service** object.

Table 142: Service Utility Methods

8.4.8 Directory Service Info Filter

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

8.4.8.1 Service Info Members

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 APIJava 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 **DomainTypes.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 143: 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 APIJava 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 143: 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 144: ServiceInfo Flags

8.4.8.3 Service Info Methods

METHOD NAME	DESCRIPTION
clear	Clears a **ServiceInfo** object. Useful for object reuse.
copy	Performs a deep copy of a **ServiceInfo** object.

Table 145: ServiceInfo Utility Methods

8.4.9 Directory Service State Filter

A **ServiceState** object 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 APIJava Edition *Refinitiv 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 APIJava 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 146: 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 147: ServiceState Flags

8.4.9.3 Service State Methods

METHOD NAME	DESCRIPTION
clear	Clears a **ServiceState** object. Useful for object reuse.
copy	Performs a deep copy of a **ServiceState** object.

Table 148: ServiceState Utility Methods

8.4.10 Directory Service Group Filter

A **ServiceGroup** object 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 APIJava Edition *Refinitiv Domain Model Usage Guide*.

8.4.10.1 Service Group Members

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 APIJava 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 149: 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 150: ServiceGroup Flags

8.4.10.3 Service Group Methods

METHOD NAME	DESCRIPTION
clear	Clears an **ServiceGroup** object. Useful for object reuse.
copy	Performs a deep copy of a **ServiceGroup** object.

Table 151: ServiceGroup Utility Methods

8.4.11 Directory Service Load Filter

A **ServiceLoad** object 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 APIJava Edition *Refinitiv 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 APIJava 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 APIJava Edition *Refinitiv 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.
openWindow	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 152: 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 153: ServiceLoad Flags

8.4.11.3 Service Load Methods

METHOD NAME	DESCRIPTION
clear	Clears an **ServiceLoad** object. Useful for object reuse.
copy	Performs a deep copy of a **ServiceLoad** object.

Table 154: ServiceLoad Utility Methods

8.4.12 Directory Service Data Filter

An **ServiceData** object 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 APIJava Edition *Refinitiv 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 APIJava 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 APIJava 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: • **Directory.DataTypes.TIME = 1** • **Directory.DataTypes.ALERT = 2** • **Directory.DataTypes.HEADLINE = 3** • **Directory.DataTypes.STATUS = 4** If present, **flags** value of **ServiceDataFlags.HAS_DATA** should be specified.

Table 155: 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 156: ServiceData Flags

8.4.12.3 Service Data Methods

METHOD NAME	DESCRIPTION
clear	Clears a **ServiceData** object. Useful for object reuse.
copy	Performs a deep copy of a **ServiceData** object.

Table 157: ServiceData Methods

8.4.13 Directory Service Link Info Filter

A **ServiceLinkInfo** object 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 APIJava Edition *Refinitiv Domain Model Usage Guide*.

The **ServiceLinkInfo** object contains a list of **ServiceLink** objects 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 APIJava Edition *Developers Guide*.
linkList	**Optional**. Contains a list of **ServiceLink** objects, each representing a source.

Table 158: ServiceLinkInfo Members

8.4.13.2 Service Link Info Methods

METHOD NAME	DESCRIPTION
clear	Clears a **ServiceLinkInfo** object. Useful for object reuse.
copy	Performs a deep copy of a **ServiceLinkInfo** object.

Table 159: ServiceLinkInfo Methods

8.4.14 Directory Service Link

A **ServiceLink** object 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** object. For further details on Service Link filter content, refer to the Enterprise Transport APIJava Edition *Refinitiv Domain Model Usage Guide*.

8.4.14.1 Service Link Members

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 APIJava 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: • **Directory.LinkCodes.NONE = 0** • **Directory.LinkCodes.OK = 1** • **Directory.LinkCodes.RECOVERY_STARTED = 2** • **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: • **Directory.LinkTypes.INTERACTIVE = 1** • **Directory.LinkTypes.BROADCAST = 2** If present, a **flags** value of **ServiceLinkFlags.HAS_TYPE** should be specified.

Table 160: 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 161: ServiceLink Flags

8.4.14.3 Service Link Methods

METHOD NAME	DESCRIPTION
clear	Clears a **ServiceLink** object. Useful for object reuse.
copy	Performs a deep copy of a **ServiceLink** object.

Table 162: ServiceLink Methods

8.4.15 Directory Message

DirectoryMsg is the general purpose base interface for 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 163: DirectoryMsg Interfaces

8.4.16 Directory Message Utility Methods

METHOD NAME	DESCRIPTION
copy	Performs a deep copy of a **DirectoryMsg** object.

Table 164: 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 165: Directory Encoding and Decoding Methods

8.4.17.2 Encoding a Source Directory Request

```
EncodeIterator encodeIter = CodecFactory.createEncodeIterator();
DirectoryRequest directoryRequest = (DirectoryRequest)DirectoryMsgFactory.createMsg();

/* Clear the Directory Request object. */
directoryRequest.clear();

/* Set directory message type - required as object created by DirectoryMsgFactory.createMsg() is generic
        Directory object. */
directoryRequest.rdmMsgType(DirectoryMsgType.REQUEST);

/* Set stream id. */
directoryRequest.streamId(streamId);

/* Set flags indicating presence of optional members. */
directoryRequest.flags(DirectoryRequestFlags.HAS_SERVICE_ID | STREAMING);

/* Set Service ID. */
directoryRequest.serviceId(273);

/* Set service filter. */
directoryRequest.filter(Directory.ServiceFilterFlags.INFO | Directory.ServiceFilterFlags.STATE |
        Directory.ServiceFilterFlags.GROUP);

/* 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 = directoryRequest.encode(encodeIter);
```

Code Example 23: Directory Request Encoding Example

8.4.17.3 Decoding a Source Directory Request

```
DecodeIterator decodeIter = CodecFactory.createDecodeIterator();
DirectoryRequest directoryRequest = (DirectoryRequest)DirectoryMsgFactory.createMsg();
Msg msg = CodecFactory.createMsg();

/* Clear the decode iterator, set its RWF Version, and set it to the encoded buffer. */
decodeIter.clear();

ret = decodeIter.setBufferAndRWFVersion(msgBuf,channelMajorVersion, channelMinorVersion);

/* Decode the message to a Msg object. */
ret = msg.decode(decodeIter);

if (ret == CodecReturnCodes.SUCCESS &&
    msg.domainType() == DomainTypes.SOURCE && msg.msgClass() ==  MsgClasses.REQUEST)
{
    directoryRequest.clear();
    directoryRequest.rdmMsgType(DirectoryMsgType.REQUEST);

    ret = directoryRequest.decode(decodeIter, msg);

    if(ret == CodecReturnCodes.SUCCESS)
    {

        /* Print if Info filter was requested. */
        if ((directoryRequest.filter() & Directory.ServiceFilterFlags.INFO) != 0)
                System.out.println("Info filter requested.");

        /* Print if State filter was requested. */
        if ((directoryRequest.filter() & Directory.ServiceFilterFlags.STATE) != 0)
            System.out.println("State filter requested.");

        /* Print if Group filter was requested. */
        if ((directoryRequest.filter() & Directory.ServiceFilterFlags.GROUP) != 0)
                System.out.println("Group filter requested.");

        /* Print service ID if present. */
        if (directoryRequest.checkHasServiceId())
                System.out.println("Service ID: " + directoryRequest->serviceId);
    }
}
```

Code Example 24: Directory Request Decoding Example

8.4.17.4 Encoding a Source Directory Refresh

```
EncodeIterator encodeIter = CodecFactory.createEncodeIterator();
DirectoryRefresh directoryRefresh = (DirectoryRefresh)DirectoryMsgFactory.createMsg();

/* Clear the Directory Refresh object. */
directoryRefresh.clear();

/* Set directory message type - required as object created by DirectoryMsgFactory.createMsg() is generic
        Directory object. */
directoryRefresh.rdmMsgType(DirectoryMsgType.REFRESH);

/* Set stream id. */
directoryRefresh.streamId(streamId);

/* Set flags for optional members */
directoryRefresh.applySolicited();

/* Set state. */
directoryRefresh.state().streamState(StreamStates.OPEN);
directoryRefresh.state().dataState(DataStates.OK);
directoryRefresh.state().code(StateCodes.NONE);

/* Set filter to say the Info, State, and Group filters are supported. */
directoryRefresh.filter(Directory.ServiceFilterFlags.INFO | Directory.ServiceFilterFlags.STATE |
Directory.ServiceFilterFlags.GROUP);

/* List of services to be used.
 * This example will show encoding of one service. Additional services can be set up using the same method
 * shown below. */
/*** Create Service ***/
Service service = CodecFactory.createService();

/*** Build Service MY_SERVICE. ***/
service.clear();

/* Set flags to indicate Info and State filter are present. */
service.flags(ServiceFlags.HAS_INFO | ServiceFlags.HAS_STATE);

/* Set action to indicate adding a new service. */
service.info().action(MapEntryActions.ADD);

/* Set flags to indicate optional members. */
service.info().flags(ServiceInfoFlags.HAS_VENDOR | ServiceInfoFlags.HAS_DICTS_PROVIDED |
        ServiceInfoFlags. HAS_DICTS_USED |  ServiceInfoFlags.HAS_QOS);

/* Set service name. */
service.info().serviceName().data("MY_SERVICE");

/* Set vendor name. */
```

```
service.info().vendor().data("Refinitiv");

/* Set capabilities list. */
service.info().capabilitiesList().add(DomainTypes.DICTIONARY);
service.info().capabilitiesList().add(DomainTypes.MARKET_PRICE);
service.info().capabilitiesList().add(DomainTypes.MARKET_BY_ORDER);

/* Set dictionaries provided. */
service.info().dictionariesProvidedList().add("RWFFld");
service.info().dictionariesProvidedList().add("RWFEnum");

/* Set dictionaries used. */
service.info().dictionariesUsedList ().add("RWFFld");
service.info().dictionariesUsedList ().add("RWFEnum");

/* Build QoS list. */
Qos qos1 = CodecFactory.createQos();
qos1.timeliness(QosTimeliness.REALTIME);
qos1.rate(QosRates.TICK_BY_TICK);
Qos qos2 = CodecFactory.createQos();
service.info().qosList().add(qos2);
qos2.timeliness(QosTimeliness.REALTIME);
qos2.rate(QosRates.JIT_CONFLATED);

/* Set QoS list. */
service.info().qosList().add(qos1);
service.info().qosList().add(qos2);

/*** Build Service State for MY_SERVICE ***/
service.state().flags(ServiceStateFlags.HAS_ACCEPTING_REQS);
service.state().serviceState(1);
service.state().acceptingRequests(1);

/*** Finish and encode. ***/

/* Set the list of services on the message.*/
directoryRefresh.serviceList().add(service);

/* 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 = directoryRefresh.encode(encodeIter);
```

Code Example 25: Directory Refresh Encoding Example

8.4.17.5 Decoding a Source Directory Refresh

```
DecodeIterator decodeIter = CodecFactory.createDecodeIterator();
DirectoryRefresh directoryRefresh = (DirectoryRefresh)DirectoryMsgFactory.createMsg();
Msg msg = CodecFactory.createMsg();

/* Clear the decode iterator, set its RWF Version, and set it to the encoded buffer. */
decodeIter.clear();

ret = decodeIter.setBufferAndRWFVersion(msgBuf,channelMajorVersion, channelMinorVersion);

/* Decode the message to a Msg object. */
ret = msg.decode(decodeIter);

if (ret == CodecReturnCodes.SUCCESS && msg.domainType() == DomainTypes.SOURCE && msg.msgClass() ==
        MsgClasses.REFRESH)
{
    directoryRefresh.clear();
    directoryRefresh.rdmMsgType(DirectoryMsgType.REFRESH);

    ret = directoryRefresh.decode(decodeIter, msg);

    if(ret == CodecReturnCodes.SUCCESS)
    {
        /* Print serviceId if present. */
        if (directoryRefresh.checkHasServiceId())
            System.out.println("Service ID: " +  directoryRefresh.serviceId());

        /* Print information about each service present in the refresh. */
        for(Service service : directoryRefresh.serviceList())
        {
            /* Print Service Info if present */
            if (service.checkHasInfo())
            {
                 ServiceInfo info = service.info();

                 /* Print service name. */
                 System.out.println("Service Name: " + info.serviceName().toString());

                 /* Print vendor name if present.*/
                 if (info.checkHasVendor())
                     System.out.println("Vendor: " + info.vendor().toString());

                 /* Print supported domains if present.*/
                 for(long capability : info.capabilityList())
                     System.out.println("Capability: " +  DomainTypes.toString(capability));

                 /* Print dictionaries provided if present.*/
                 if (info.checkHasDictionariesProvided())
                 {
```

```
                    for (String dictProv : info.dictionariesProvidedList())
                        System.out.println("Dictionary Provided: " +  dictProv);
                }

                /* Print dictionaries used if present. */
                if (info.checkHasDictionariesUsed())
                {
                    for (String dictUsed : info.dictionariesUsedList())
                        System.out.println("Dictionary Used: " +  dictUsed);
                }

                /* Print qualities of service supported if present. */
                if (info.checkHasQos())
                {
                    for (Qos qos : info.qosList())
                        System.out.println ("QoS: " + qos.toString());
                }
            }

            if (service.checkHasState())
            {
                ServiceState state = service.state();
                System.out.println("Service state: " + state.serviceState());
                if(state.checkHasAcceptingRequests())
                    System.out.println("Accepting Requests: " + state.acceptingRequests());
            }
        }
    }
}
```

Code Example 26: 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 APIJava Edition *Refinitiv 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 APIJava Edition *Refinitiv 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 object follows the behavior and layout that is defined in the Enterprise Transport APIJava Edition *Refinitiv Domain Model Usage Guide*.

8.5.1.1 Dictionary Request Members

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: • **Dictionary.VerbosityValues.INFO = 0x00**: Version information only • **Dictionary.VerbosityValues.MINIMAL = 0x03**: Provides information needed for caching • **Dictionary.VerbosityValues.NORMAL = 0x07**: Provides all information needed for decoding • **Dictionary.VerbosityValues.VERBOSE = 0x0F**: Provides all information (including comments) Providers are not required to support the **MINIMAL** and **VERBOSE** filters.

Table 166: `DictionaryRequest` Members

8.5.1.2 Dictionary Request Flag Enumeration Values

FLAG ENUMERATION	DESCRIPTION
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 APIJava Edition *Refinitiv Domain Model Usage Guide*.

Table 167: `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 object follows the behavior and layout that is defined in the Enterprise Transport APIJava Edition *Refinitiv Domain Model Usage Guide*.

8.5.2.1 Dictionary Refresh Members

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 an **`DataDictionary`** object that contains content to encode. For more information on the **`DataDictionary`** object, refer to the Enterprise Transport APIJava Edition *Refinitiv 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 APIJava Edition *Developers Guide*.
dictionaryType	**Required**. Indicates the type of dictionary being provided. The dictionary encoder and decoder support the following types: • **Dictionary.VerbosityValues.FIELD_DEFINITIONS = 1** • **Dictionary.VerbosityValues.ENUM_TABLES = 2**
verbosity	**Required**. Indicates the amount of information desired from the dictionary. Available enumerations are: • **Dictionary.VerbosityValues.INFO = 0x00**: Provides version information only • **Dictionary.VerbosityValues.MINIMAL = 0x03**: Provides information needed for caching • **Dictionary.VerbosityValues.NORMAL = 0x07**: Provides all information needed for decoding • **Dictionary.VerbosityValues.VERBOSE = 0x0F**: Provides all information (including comments) Providers do not need to support the **MINIMAL** and **VERBOSE** filters.

Table 168: `DictionaryRefresh` Members

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 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 169: `DictionaryRefreshFlags`

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 APIJava Edition *Refinitiv 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 object follows the behavior and layout that is defined in the Enterprise Transport APIJava Edition *Refinitiv 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.
rdmMsgBase	**Required**. Specifies the dictionary message type. For a dictionary status, set to **DictionaryMsgType.STATUS**.
state	**Optional**. Indicates the state of the dictionary stream. For more information on **`State`**, refer to the Enterprise Transport APIJava Edition *Developers Guide*. If present, a **`flags`** value of **DictionaryStatusFlags.HAS_STATE** should be specified.

Table 170: `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 171: `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.

MEMBER	DESCRIPTION
rdmMsgBase	**Required**. Specifies the dictionary message type. For a dictionary close, set to **DictionaryMsgType.CLOSE**.

Table 172: `DictionaryClose` Member

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 173: `DictionaryMsg` Interfaces

8.5.6 Dictionary Message: Utility Methods

FUNCTION NAME	DESCRIPTION
copy	Performs a deep copy of a **`DictionaryMsg`** object.

Table 174: `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 175: Dictionary Encoding and Decoding Methods

8.5.7.2 Encoding a Dictionary Request

```
EncodeIterator encodeIter = CodecFactory.createEncodeIterator();
DictionaryRequest dictionaryRequest = (DictionaryRequest)DictionaryMsgFactory.createMsg();

/* Clear the Dictionary Request object. */
dictionaryRequest.clear();

/* Set dictionary message type - required as object created by DictionaryMsgFactory.createMsg() is
        generic Dictionary object. */
dictionaryRefresh.rdmMsgType(DictionaryMsgType.REQUEST);

/* Set stream id. */
dictionaryRefresh.streamId(streamId);

/* Set streaming flag. */
dictionaryRequest.applyStreaming();

/* Set serviceId. */
dictionaryRequest.serviceId(273);

/* Set verbosity. */
dictionaryRequest.verbosity(Dictionary.VerbosityValues.NORMAL);

/* Set dictionary name. */
dictionaryRequest.dictionaryName().data("RWFFld");

/* 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 = dictionaryRequest.encode(encodeIter);
```

Code Example 27: Dictionary Request Encoding Example

8.5.7.3 Decoding a Dictionary Request

```
DecodeIterator decodeIter = CodecFactory.createDecodeIterator();
DictionaryRequest dictionaryRequest = (DictionaryRequest)DictionaryMsgFactory.createMsg();
Msg msg = CodecFactory.createMsg();

/* Clear the decode iterator, set its RWF Version, and set it to the encoded buffer. */
decodeIter.clear();
ret = decodeIter.setBufferAndRWFVersion(msgBuf,channelMajorVersion, channelMinorVersion);

/* Decode the message to a Msg object. */
ret = msg.decode(decodeIter);

if (ret == CodecReturnCodes.SUCCESS && msg.domainType() == DomainTypes.DICTIONARY && msg.msgClass() ==
        MsgClasses.REQUEST)
{
    dictionaryRequest.clear();

/* Set dictionary message type - required as object created by DictionaryMsgFactory.createMsg() is
        generic Dictionary object. */
    dictionaryRequest.rdmMsgType(DictionaryMsgType.REQUEST);

    ret = dictionaryRequest.decode(decodeIter, msg);

    if(ret == CodecReturnCodes.SUCCESS)
    {
        if(dictionaryRequest.checkStreaming())
            System.out.println("Request is streaming");

        /* Print serviceId. */
        System.out.println ("Service ID: " + dictionaryRequest.serviceId());

        /* Print verbosity. */
        System.out.println ("Verbosity: " + dictionaryRequest.verbosity());

        /* Print dictionary name. */
        System.out.println ("Dictionary Name: " + dictionaryRequest.dictionaryName().toString());
    }
}
```

Code Example 28: Dictionary Request Decoding Example

8.5.7.4 Encoding a Dictionary Refresh

```
EncodeIterator encodeIter = CodecFactory.createEncodeIterator();
DictionaryRefresh dictionaryRefresh = (DictionaryRefresh)DictionaryMsgFactory.createMsg();

/* Clear the Dictionary Refresh object. */
dictionaryRefresh.clear();
dictionaryRefresh.rdmMsgType(DictionaryMsgType.REFRESH);

DataDictionary dataDictionary = CodecFactory.createDataDictionary();
dataDictionary.clear();

ret = dataDictionary.loadFieldDictionary("RDMFieldDictionary", errorText);

/* Clear the Dictionary Refresh object. */
dictionaryRefresh.clear();

/* Set dictionary message type - required as object created by DictionaryMsgFactory.createMsg() is
        generic Dictionary object. */

dictionaryRefresh.rdmMsgType(DictionaryMsgType.REFRESH);

/* Set stream id. */
dictionaryRefresh.streamId(streamId);

/* Set state fields to state object managed by dictionary refresh. */
dictionaryRefresh.state().streamState(StreamStates.OPEN);
dictionaryRefresh.state().dataState(DataStates.OK);
dictionaryRefresh.state().code(StateCodes.NONE);

/* Set flags. */
dictionaryRefresh.applySolicited();

/* Set dictionary name. */
dictionaryRefresh.dictionaryName().data("RWFFld");

/* Set dictionary type. */
dictionaryRefresh.dictionaryType(Dictionary.Types.FIELD_DEFINITIONS);

/* Set the dictionary. */
dictionaryRefresh.dictionary(dataDictionary);

/* Set serviceId. */
dictionaryRefresh.serviceId(273);

/* Set verbosity. */
dictionaryRefresh.verbosity(Dictionary.VerbosityValues.NORMAL);

do
{
```

```
    /* (Represents the application getting a new buffer to encode the message into.) */
    getNextEncodeBuffer(msgBuffer);

    /* 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. This will return CodecReturnCodes.DICT_PART_ENCODED if it only a part
    * was encoded. We must keep encoding the message until CodecReturnCodes.SUCCESS is returned. */
    ret = dictionaryRefresh.encode(encodeIter);
} while (ret == CodecReturnCodes.DICT_PART_ENCODED);
```

Code Example 29: Dictionary Refresh Encoding Example

8.5.7.5 Decoding a Dictionary Refresh

```
DecodeIterator decodeIter = CodecFactory.createDecodeIterator();
DictionaryRefresh dictionaryRefresh = (DictionaryRefresh)DictionaryMsgFactory.createMsg();
Msg msg = CodecFactory.createMsg();

DataDictionary dataDictionary = CodecFactory.createDataDictionary();
dataDictionary.clear();

int dictionaryTypeForThisStreamId = 0;

do
{
    /* (Represents the application getting the next buffer to decode.) */
    getNextDecodeBuffer(msgBuf);

    /* Clear the decode iterator, set its RWF Version, and set it to the encoded buffer. */
    decodeIter.clear();
    ret = decodeIter.setBufferAndRWFVersion(msgBuf,channelMajorVersion, channelMinorVersion);

    /* Decode the message to a Msg object. */
    sret = msg.decode(decodeIter);
    if (ret == CodecReturnCodes.SUCCESS && msg.domainType() == DomainTypes.DICTIONARY && msg.msgClass()
            == MsgClasses.REFRESH)
    {
        dictionaryRefresh.clear();

        /* Set dictionary message type - required as object created by DictionaryMsgFactory.createMsg()
                is generic Dictionary object. */
        dictionaryRefresh.rdmMsgType(DictionaryMsgType.REFRESH);

        ret = dictionaryRefresh.decode(decodeIter, msg);

        if(ret == CodecReturnCodes.SUCCESS)
        {
```

```
            /* Print if request is streaming. */
            if (dictionaryRefresh.checkSolicited())
                System.out.println("Refresh is solicited.");

            /* Print info if present. If the dictionary is split into parts, this is normally only
                present on the first part. */
            if (dictionaryRefresh.checkHasInfo())
            {
                /* Remember the dictionary type for this stream since subsequent parts will not indicate
                        it. */
                dictionaryTypeForThisStreamId = dictionaryRefresh.dictionaryType();

                /* Print version. */
                System.out.println("Version: " + dictionaryRefresh.version.toString());

                /* Print dictionary ID. */
                System.out.println("Dictionary ID: " + dictionaryRefresh.dictionaryId());
            }

            /* Print serviceId. */
            System.out.println("Service ID: " + dictionaryRefresh.serviceId());

            /* Print verbosity. */
            System.out.println("Verbosity: " + dictionaryRefresh.verbosity());

            /* Print dictionary name. */
            System.out.println("Dictionary Name: " + dictionaryRefresh.dictionaryName().toString());

            if (dictionaryTypeForThisStreamId ==  Dictionary.Types.FIELD_DEFINITIONS)
            {
                /* Decode the dictionary content into the DataDictionary object. */
                decodeIter.clear();
                ret = decodeIter.setBufferAndRWFVersion(dictionaryRefresh.dataBody(),
                        channelMajorVersion, channelMinorVersion);

                ret = dataDictionary.decodeFieldDictionary(decodeIter,
                        Dictionary.VerbosityValues.NORMAL, errorText);
            }
        }
    }
} while(!(dictionarRefresh.checkRefreshComplete());
```

Code Example 30: Dictionary Refresh Decoding Example

8.6 RDM Queue Messages

The Queue Messaging domain model is a series of message constructs that you use to interact with a Queue Provider. A Queue Provider can persist content for which users want to have guaranteed delivery and can also help send content to destinations with which users cannot directly communicate.

8.6.1 Queue Data Message Persistence

When opening a queue messaging stream with a queue provider, using a persistence file can guarantee delivery of messages sent by the OMM consumer on that queue stream. The queue file will be named after the name of the queue stream (as specified in the **`QueueRequest`** message that opened the stream). When the consumer submits **`QueueData`** messages, the consumer stores these messages in the persistence file in case the tunnel stream to the queue provider is lost and reconnected. As **`QueueAck`** messages are received from the queue provider, space in the persistence file is freed for additional messages. If at any time the application submits an **`QueueData`** message but the persistence file has no room for it, the application receives the **ReactorReturnCodes.PERSISTENCE_FULL** return code.

The **`ClassOfService.guarantee.persistLocally`** option (set when opening the tunnel stream) specifies whether to create and maintain persistence files. The location for storage of persistent files is specified by the **`ClassOfService.guarantee.persistenceFilePath`** option. For more information on these options, refer to Section 6.8.3.

NOTE: Refinitiv recommends that the **`ClassOfService.guarantee.persistenceFilePath`** be set to a local storage device.

If a particular queue stream is no longer needed, the user may delete the persistence file that carries the associated queue stream's name.

Warning! If you delete a persistence file that stores messages that were not successfully transmitted, the messages will be lost.

8.6.2 Queue Request

The OMM application encodes and sends a ***Queue Request*** message to a Queue Provider to open a user queue. By opening a queue with an **`QueueRequest`**, the user receives any content previously sent to and persisted on a Queue Provider. To send content to another user's queue, a user must first open their own queue.

MEMBER	DESCRIPTION
rdmMsgBase	**Required**. Specifies the message type. For a queue request, use **QueueMsgType.REQUEST**.
sourceName	**Required**. Specifies the name of the queue you want to open.

Table 176: `QueueRequest` Members

8.6.3 Queue Refresh

A Queue Provider encodes and sends a ***Queue Refresh*** message to OMM applications to inform users about queue open requests and give state information pertaining to specific queue refresh request attempts.

MEMBER	DESCRIPTION
rdmMsgBase	**Required**. Sets the message type. For a queue refresh, use **QueueMsgType.REFRESH**.
sourceName	**Required**. Specifies the name of a queue you want to open.
state	**Required**. Indicates the state of the queue. • States of **Open** and **Ok** indicate the queue was successfully opened. • Other state combinations indicate an issue, for which additional code and text provide supplemental information. For more information on **`State`**, refer to the Enterprise Transport APIJava Edition *Developers Guide*.
queueDepth	**Required**. Indicates the number of messages remaining in the queue for this stream.

Table 177: `QueueRefresh` Members

8.6.4 Queue Status

A Queue Provider encodes and sends ***Queue Status*** messages to OMM applications, conveying state information about a user's queue.

MEMBER	DESCRIPTION
flags	**Required**. Indicates the presence of optional queue status members. **`flags`** has only one enumeration: **HAS_STATE**, which indicates the presence of the **`state`** member. If **`flags`** is absent (or has no value), any previously conveyed state continues to apply.
rdmMsgBase	**Required**. Sets the message type. For a queue status, use **QueueMsgType.STATUS**.
state	Indicates the state of the queue: • States of **Open** and **Ok** indicate the queue is in a good state. • Other state combinations indicate an issue, for which additional code and text provide supplemental information. For more information on **`State`**, refer to the Enterprise Transport APIJava Edition *Developers Guide*.

Table 178: `QueueStatus` Members

8.6.5 Queue Close

An OMM application encodes and sends a ***Queue Close*** message to a Queue Provider, closing the user's queue.

MEMBER	DESCRIPTION
rdmMsgBase	**Required**. Sets the message type. For a queue close, use **QueueMsgType.CLOSE**.

Table 179: `QueueClose` Members

8.6.6 Queue Data

Both OMM applications and queue providers can send and receive ***Queue Data*** messages, which exchange data content between queue users and also communicate whether content was undeliverable.

8.6.6.1 Queue Data Members

MEMBER	DESCRIPTION
containerType	**Required**. Indicates the type of contents in this queue data message.
destName	**Required**. Specifies the name of the queue to which content is sent.
encodedDataBody	Optional. • If sending a message, populate **`encodedDataBody`** with pre-encoded content. If sending a message without pre-encoded contents, you can use the encoding methods described in Section 8.6.6.4. • If receiving a message, **`encodedDataBody`** can be used to access payload contents for decoding.
flags	**Required**. Specifies any flags that indicate more information about this message. For further details on available flags, refer to Section 8.6.6.2.
identifier	**Required**. A user-specified unique identifier for the message being sent. **`identifier`** is used when acknowledging this content via a Queue Ack message.
queueDepth	**Required**. Indicates the number of Queue Data or Queue Data Expired messages still inbound on this queue stream, following this message.
rdmMsgBase	**Required**. Sets the message type. For queue data, use **QueueMsgType.DATA**.
sourceName	**Required**. Specifies the name of the queue from which content is sourced, which should match the **`sourceName`** specified in the Queue Request for this substream.
timeout	**Optional**. Specifies the desired timeout for this content (which can be any of the **QueueMsgTimeoutCodes** in Section 8.6.6.3 or a specific time interval in milliseconds). If a timeout value expires during the course of delivery, the content is returned as a **QueueDataExpired** message. If not specified, this defaults to **QueueMsgTimeoutCodes.INFINITE** (i.e., the content never times out).

Table 180: `QueueData` Members

8.6.6.2 Queue Data Flag

`QueueData` messages and **`QueueDataExpired`** messages use the following flag:

FLAG	DESCRIPTION
QueueDataFlags.POSSIBLE_DUPLICATE= 0x1	Indicates that the message was retransmitted and that the application might have already received it.

Table 181: Queue Data Flag

8.6.6.3 Queue Message Timeout Codes

ENUMERATION	DESCRIPTION
INFINITE	This message persists in the system for an infinite amount of time.
IMMEDIATE	This message immediately times out if any portion of its delivery path is unavailable.
PROVIDER_DEFAULT	This message persists in the system for a duration set by the provider.

Table 182: Queue Data Message Timeout Codes

8.6.6.4 Queue Data Encoding

METHOD NAME	DESCRIPTION
encode	When sending no payload or payload content is preencoded and specified on the **`QueueData.encodedDataBody`** buffer, this method encodes the **`QueueData`** message in a single call.
encodeComplete	Completes the content encoding into this **`QueueData`** message.
encodeInit	Begins the process of encoding content into this **`QueueData`** message. This method takes an **`EncodeIterator`** as a parameter, where the **`EncodeIterator`** is associated with the buffer into which content is encoded. When this method returns, users should call additional methods required to encode the content. After all remaining encoding is completed, call the **`encodeComplete`** method.

Table 183: Queue Data Message Encoding Methods

8.6.6.5 Queue Data Message Encoding Code Sample

```
EncodeIterator _msgEncIter = CodecFactory.createEncodeIterator();
QueueData _queueData = QueueMsgFactory.createQueueData();

// initialize the QueueData encoding
_queueData.clear();
_queueData.streamId(QUEUE_MSG_STREAM_ID);
_queueData.identifier(124);
_queueData.sourceName().data("MY_QUEUE");
_queueData.destName().data("DESTINATION_QUEUE");
_queueData.timeout(QueueMsgTimeoutCodes.INFINITE);
_queueData.containerType(DataTypes.FIELD_LIST);

_msgEncIter.clear();
_msgEncIter.setBufferAndRWFVersion(buffer, tunnelStream.classOfService().common().
        protocolMajorVersion(), tunnelStream.classOfService().common().protocolMinorVersion());

// begin encoding content into QueueData message
if ((ret = _queueData.encodeInit(_msgEncIter)) < ReactorReturnCodes.SUCCESS)
{
    System.out.println("QueueData.encodeInit() failed");
    return;
}

// Start Content Encoding - follow standard field list encoding
// as shown in the Transport API Java Developers Guide examples

// when content encoding is done, complete the QueueData encoding
if ((ret = _queueData.encodeComplete(_msgEncIter)) < ReactorReturnCodes.SUCCESS)
{
    System.out.println("QueueData.encodeComplete() failed");
    return;
}
```

Code Example 31: Queue Data Message Encoding Example

8.6.7 QueueDataExpired

If queue data messages sent on a queue stream cannot be successfully delivered, the queue provider sends **QueueDataExpired** messages on the queue stream to OMM consumer applications.

OMM consumer applications do not send this message.

8.6.7.1 QueueDataExpired Members

MEMBER	DESCRIPTION
containerType	**Required**. Indicates the type of contents in the message.
destName	**Required**. **destName** specifies the name of the queue from which content is sourced (i.e., the value of **sourceName** as set in the original **QueueData** message).
encodedDataBody	Optional. Contains the payload contents (if any) of the original Queue Data message.
flags	**Required**. **flags** indicate more information about this message. For details, refer to Section 8.6.6.2.
identifier	**Required**. A user-specified, unique identifier for the message (which is the same as the **identifier** from the original **QueueData** message).
queueDepth	**Required**. Indicates how many Queue Data or Queue Data Expired messages are still inbound on this queue stream (following this message).
rdmMsgBase	**Required**. Specifies the queue message type. For expired queue data, use **QueueMsgType.DATAEXPIRED**.
sourceName	**Required**. **sourceName** specifies the name of the queue to which content was sent (i.e., the value of **destName** as set in the original **QueueData** message).
undeliverableCode	**Required**. Specifies a code explaining why the content was undeliverable. For more information on undeliverable codes and their meanings, refer to Section 8.6.7.2.

Table 184: QueueDataExpired Members

8.6.7.2 Queue Data Message Undeliverable Codes

ENUMERATION	REASON FOR DELIVERY FAILURE
EXPIRED	Indicates that the timeout value specified for this message has expired.
INVALID_SENDER	Indicates that the sender of this message has now become invalid.
INVALID_TARGET	Indicates that the specified destination of this message does not exist.
MAX_MSG_SIZE	Indicates that the message was too large.
NO_PERMISSION	Indicates that the source/sender of this message is not permitted to send or is not permitted to send to the specified destination.
QUEUE_DISABLED	Indicates that the specified destination of this message has a disabled queue.
QUEUE_FULL	Indicates that the specified destination of this message has a full queue and cannot receive any additional content.
TARGET_DELETED	Indicates that the target queue was deleted after sending the message, but before the message was delivered.

Table 185: Queue Data Message Undeliverable Codes

8.6.8 Queue Ack

A Queue Provider encodes and sends a ***Queue Ack*** message to OMM applications, acknowledging that a Queue Data message is persisted on the Queue Provider. After a Queue Provider acknowledges persistence, the application no longer needs to persist the acknowledged content.

MEMBER	DESCRIPTION
destName	**Required**. Specifies the name of the queue from which content is sourced (i.e., the value of **`sourceName`** as set in the original Queue Data message).
identifier	**Required**. The identifier of the message being acknowledged. This should match the **`QueueData.identifier`** for the message being acknowledged.
sourceName	**Required**. Specifies the name of the queue to which content was originally sent (i.e., the value of **`destName`** as set in the original Queue Data message).
rdmMsgBase	**Required**. Sets the message type. For a queue ack, this is set to **QueueMsgType.ACK**.

Table 186: Queue Ack Members

9 Warm Standby Feature

9.1 Overview

The Warm Standby feature is implemented at the Value Add Watchlist layer of Enterprise Transport API and is client-side feature. This feature works by providing the application the capability to failover from an active to one or more standby server(s) in the event that the primary/active fails. Application must configure the active and standby servers to use this API feature. After the connections are established with the provided servers which form a Warm Standby group, the client-side or consumer sends messages to the standby server connections to change their mode to Standby. Requested items are opened on all servers by the consumer but the active server responds with messages such as refresh, updates, status, etc. to the consumer. Standby servers respond with blank/empty refreshes. When primary fails, consumer notifies the next server in standby list that it is now Active. The new active server responds with refresh as needed resumes updates for all open items. This process of cut-over is transparent to the application.

The watchlist feature must be enabled to use the Warm Standby feature. A server qualifies to be a standby only if it advertises support for Warm Standby, supports similar features over login and offers an identical service (supported domains, quality of service, etc.) as the active server.

Warm Standby not only reduces overall recovery time, but also network traffic by not inducing a “packet storm” with a flurry of re-requests to a standby server. Because the standby server is already aware of items an application has subscribed for, during a failover Enterprise Enterprise Transport API does not need to re-subscribe open items between a provider and consumer.

9.2 Warm Standby Modes

The Enterprise Transport API Value Add layer supports two Warm Standby modes:

- Login based Warm Standby
- Service based Warm Standby

The login based Warm Standby uses the connection lost event to switch from a primary server to a standby server from the standby server list. The service based Warm Standby uses the service down event OR connection lost event to switch all subscribe items from a primary service to a standby service.

The service based Warm Standby mode offers better resiliency than the login based mode as it can switch from primary to standby if an upstream service is down but the connection to both servers remains intact. A particular server may be the primary for one service and standby for another service as a result. This ability to failover in the event of service down or channel down events makes the service based Warm Standby the recommended mode.

The following figure illustrates the sequence of events when using the Login Based Warm Standby feature:

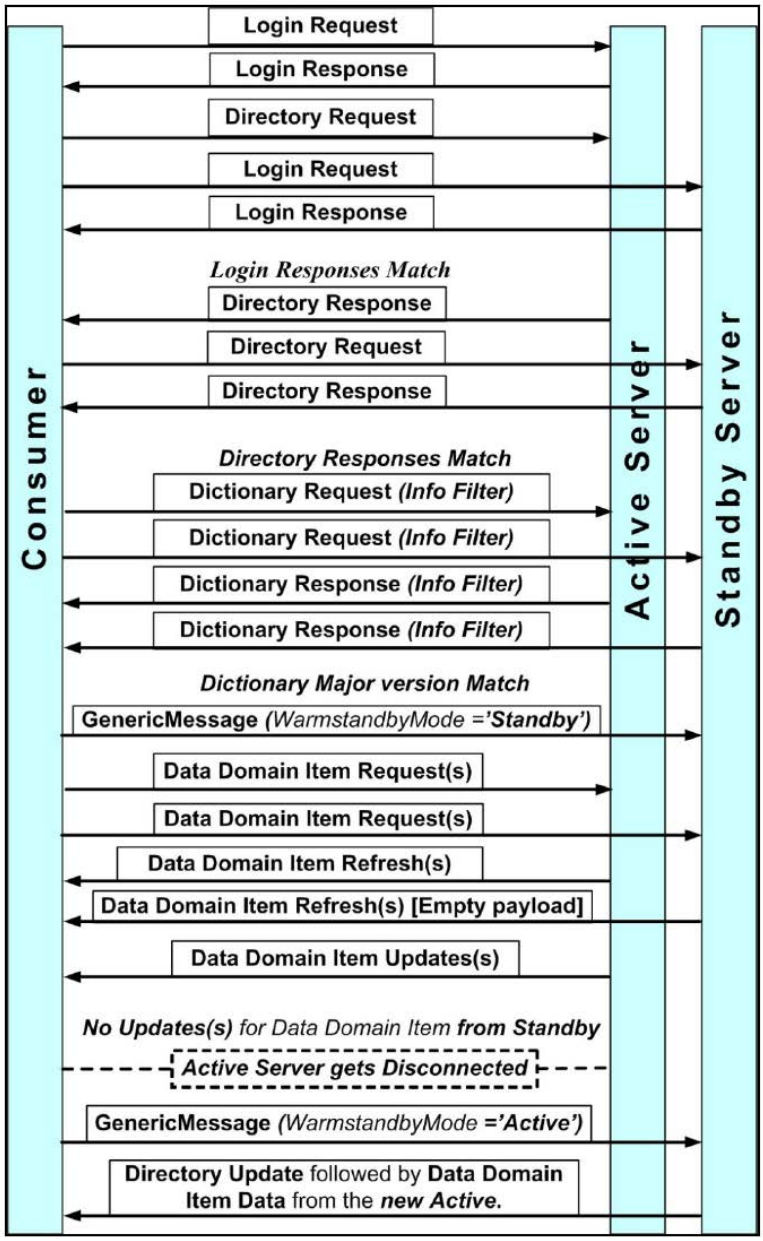


Figure 12. Login Based Warm Standby Order of Events in a Cutover from Active to Standby

The following figure illustrates the sequence of events when using the Service Based Warm Standby feature:

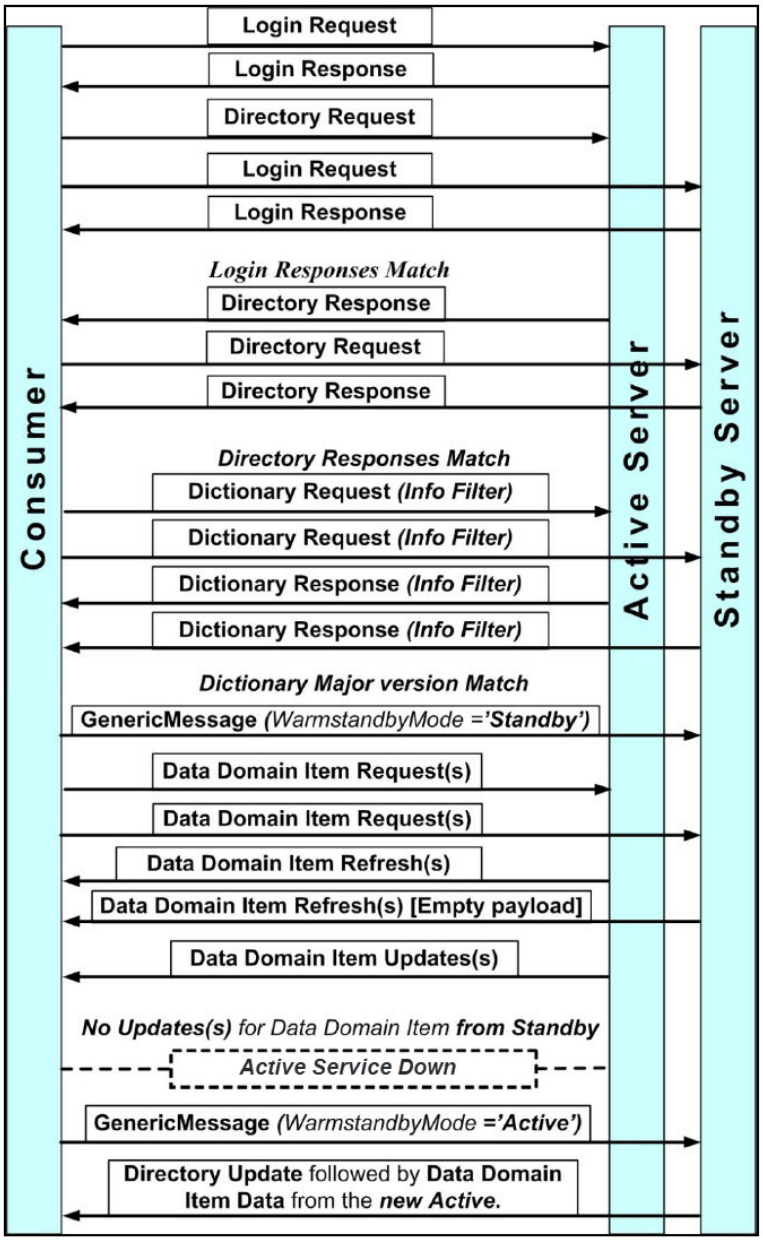


Figure 13. Service Based Warm Standby Order of Events in a Cutover from Active to Standby

9.3 Warm Standby Configuration and Feature Details

Application written to the Enterprise Transport API VA layer with watchlist enabled may be configured to use a particular stand by mode. Application must configure a group of servers, or Warm Standby group in which a starting active server and one or more standby servers are specified. Using this configuration, the Enterprise Transport API library will internally request the same items from both active and standby servers.

For the sample code for specifying a starting active server and a standby server, see Section 9.3.1.

The consumer application receives updates only on the item streams opened on the active server or active service; it does not get updates, status, unsolicited refreshes, and Generic Messages from the standby servers(s) or standby service(s). If the active server or active service is down, Enterprise Transport API notifies one of the standby servers that it is a new active for the requested service. The new active then begins sending data without the consumer application needing to re-request the items. After cut-over, the new active responds by sending

any conflated updates containing all changed data for conflatable domains and sending unsolicited refresh messages followed by updates for any non-conflated domains. This process brings active data streams back to their pre-failover state.

If the failed server or service comes back online, Enterprise Transport API does not switch back to it. It becomes one of the servers in the standby list. Internally, the Enterprise Transport API consumer-side sends a *ConsumerConnectionStatus GenericMsg* on the Login domain for login based and on the Directory domain for service based to indicate to the provider applications whether they should operate as an active server or service based on the Warm Standby mode. Refer to the *RDM Usage Guide* for more details on this message on the Login and Directory domains.

After establishing a connection to the standby server, Enterprise Transport API internally obeys the following rules for Warm Standby:

- The Standby server's login response must match the following login elements and state support for standby and the same standby mode as active server:
  - **ApplicationId**
  - **PositionId**
  - **ProvidePermissionProfile**: If requested by the user, the values received from all servers must match what was requested.
  - **ProvidePermissionExpressions**: If requested by the user, the values received from all servers must match what was requested.
  - **SingleOpen**
  - **AllowSuspectData**
  - **SupportPost**
  - **SupportBatchRequests**
  - **SupportOMMPost**
  - **SupportStandby**
  - **SupportViewRequests**

  If any of these elements does not match, then Enterprise Transport API disconnects the standby server.
- The standby server's Directory response must match all attributes except **Vendor** and **IsSource** of the Directory Info Filter for services common to both active and standby servers. Otherwise Enterprise Transport API disconnects the standby server.
- The standby server's Dictionary response must match the major version. Otherwise Enterprise Transport API disconnects the standby server.
- The provider application can optionally send the **SupportStandbyMode** element in login response to advertise supported Warm Standby mode to consumer applications. Enterprise Transport API disconnects the server if the mode does not match with the configured Warm Standby mode on consumer side.
- Enterprise Transport API routes Generic Messages only on streams in administrative domains. Generic Message on data streams is not routed to the standby server(s).
- Post Message is routed to the active and all standby servers to ensure data on all servers is synchronized.

If the active server or service fails when switching to the new active server or service, Enterprise Transport API does the following:

- For all services that were offered by the old active server but are not offered by the new active server, Enterprise Transport API sends a directory update message stating that the service is down for login based.
- For an item on the failed server but no longer open on the newly active server, Enterprise Transport API sends an internally-generated status message with the *ClosedRecover* stream state to the consumer application.

9.3.1 Configuration Example for a Staring Server and a Standby Server

```
ReactorWarmStandbyGroup reactorWarmStandbyGroup = ReactorFactory.createReactorWarmStandbyGroup();
ReactorWarmStandbyServerInfo standbyServerInfo = ReactorFactory.createReactorWarmStandbyServerInfo();
ReactorConnectOptions reactorConnectOpts = ReactorFactory.createReactorConnectOpts();

reactorWarmStandbyGroup.startingActiveServer().reactorConnectInfo().connectOptions().connectionType(C
   onnectionTypes.SOCKET);
reactorWarmStandbyGroup.startingActiveServer().reactorConnectInfo().connectOptions().unifiedNetworkIn
   fo().address("dataserver1");
reactorWarmStandbyGroup.startingActiveServer().reactorConnectInfo().connectOptions().unifiedNetworkIn
   fo().serviceName("14002");

standbyServerInfo.reactorConnectInfo().connectOptions().connectionType(ConnectionTypes.SOCKET);
standbyServerInfo.reactorConnectInfo().connectOptions().unifiedNetworkInfo().address("dataserver2");
standbyServerInfo.rreactorConnectInfo().connectOptions().unifiedNetworkInfo().serviceName("14002");

reactorWarmStandbyGroup.standbyServerList().add(standbyServerInfo);
reactorWarmStandbyGroup.warmStandbyMode(ReactorWarmStandbyMode.SERVICE_BASED);
reactorConnectOpts.reactorWarmStandbyGroupList.add(reactorWarmStandbyGroup);
```

Example 32: Warm Standby Configuration Example

10 Payload Cache Detailed View

10.1 Concepts

The Value Added Payload Cache component provides a facility for storing OMM containers (the message's data payload). Typical use of a payload cache is to store the current image of OMM data streams, where each entry in the cache corresponds to a single data stream. The initial content of a cache entry is defined by the payload of a refresh message. The current (or last) value of the entry is defined by the cumulative application of all refresh and update messages applied to the cache entry container. Values are stored in and retrieved from the cache as encoded OMM containers.

A cache is defined as a collection of OMM data containers. An application may create multiple cache collections, or instances, depending on how it wants to organize the data. The only restriction on cache organization is that all entries in a cache must use the same RDM Field Dictionary to define the set of field definitions it will use. At minimum, a separate cache would be required for each field dictionary in use by the application. However, because cache instances can also share the same field dictionary, partitioning is not restricted to dictionary usage. Some examples of how cache instances can be organized in an application include: all item streams on a connection; all items belonging to a particular service; all items across the entire application.

The application is responsible for organizing cache instances, managing the lifecycle of all entries in each cache, and applying and retrieving data from the cache. Figure 14 shows an example consumer type application which has created two cache instances to store data from two services on an OMM provider.

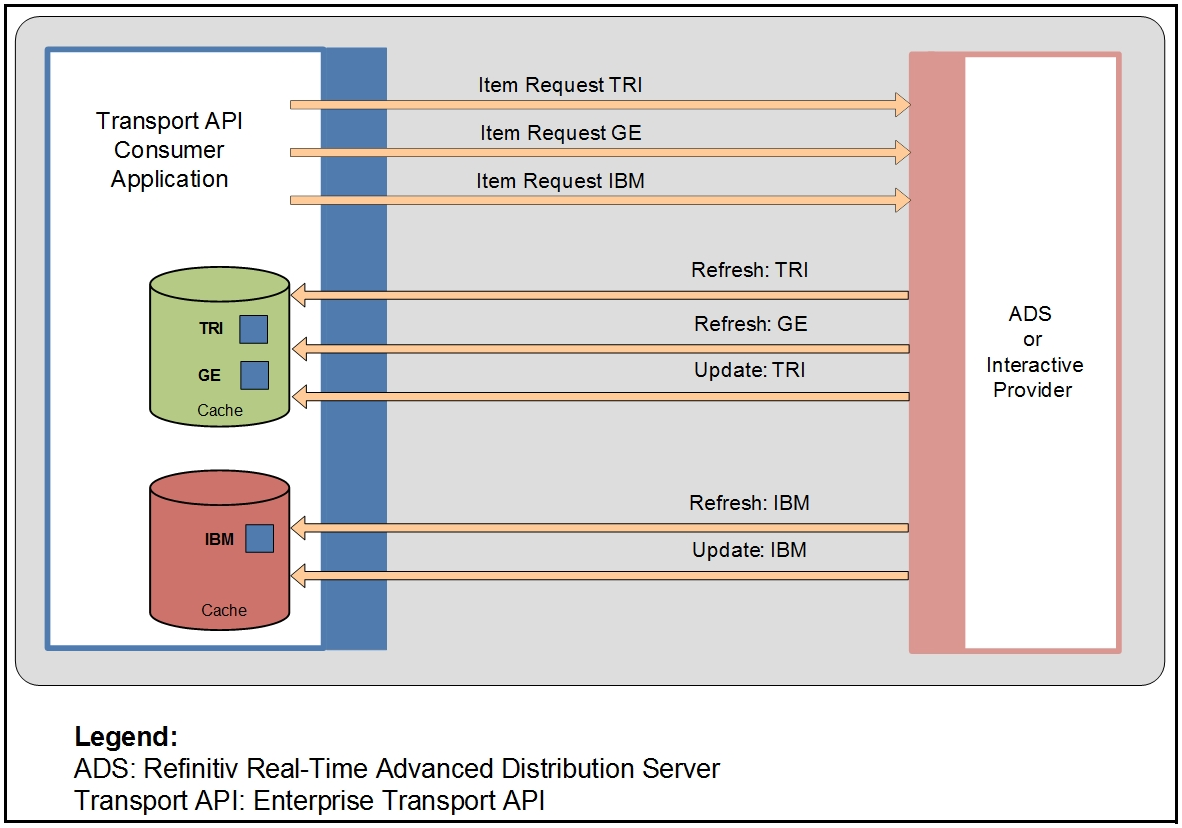


Figure 14. Consumer Application using Cache to Store Payload Data for Item Streams

10.2 Payload Cache

This section describes how the payload cache is managed (initialization and uninitialization), and how instances of cache (collections of payload entries) are created and destroyed.

10.2.1 Payload Cache Management

After the first Value Added Payload Cache instance is created, all global static resources used by the cache are initialized. When the application destroys the last cache instance, the cache releases all the resources it used.

10.2.2 Cache Error Handling

Some of the methods on the payload cache interface use the **CacheError** object to return error information. This object will be populated with additional information if an error occurs during the method call. The application should check the return value from methods. The application can optionally provide the **CacheError** object to obtain additional information.

10.2.2.1 Cache Error Class Members

The **CacheError** has the following class members:

CLASS MEMBER	DESCRIPTION
errorId	Specifies an error ID. The range of values is defined by the set of Enterprise Transport API Codec return codes (from the **CodecReturnCodes** enumeration).
text	This **String** will contain text with additional information when a method call returns a failed result.

Table 187: CacheError Class Members

10.2.2.2 Clearing a Cache Error

The following method clears the **CacheError**.

METHOD NAME	DESCRIPTION
clear	Clears the **CacheError** object. Use this method prior to passing the object to a cache interface method.

Table 188: CacheError Utility Method

10.2.3 Payload Cache Instances

A payload cache instance is a collection of payload data containers. An empty cache instance must be created before any data can be stored in the cache. When a cache or its entries are no longer needed, it can be destroyed. For methods used to create and destroy a cache, refer to Section 10.2.3.1.

10.2.3.1 Managing the Payload Cache Instance

METHOD NAME	DESCRIPTION
CacheFactory.createPayloadCache	Creates a payload cache instance. Options are passed in via the **`PayloadCacheConfigOptions`** whose member is defined in Section 10.2.3.2.
destroy	Destroys a payload cache instance. Any entries remaining in the cache are also destroyed at this time.
destroyAll	Destroys all payload cache instances. All entries remaining in the application are also destroyed at this time.

Table 189: Methods for Managing Cache Instances

10.2.3.2 Payload Cache Config Options

MEMBER	DESCRIPTION
maxItems	Sets the maximum number of entries allowed in the cache. When the maximum number of items is reached, the cache refuses new entries until existing entries are removed. The **`CacheFactory.createPayloadEntry`** method will return a null **`PayloadEntry`** when the maximum number of items is reached. When set to zero, the cache allows an unlimited number of items. Refer to Section 10.3.1.

Table 190: `PayloadCacheConfigOptions` Members

10.2.4 Managing RDM Field Dictionaries for Payload Cache

Each cache instance requires an RDM Field Dictionary, to define the set of fields that may be encoded in the OMM containers stored in the cache.

A cache is associated with a field dictionary through a setting process, which requires a **`DataDictionary`** object loaded with the field dictionary. The dictionary object can be loaded from a file (using the **`loadFieldDictionary`** method) or from an encoded dictionary message from a provider (using the **`decodeFieldDictionary`** method). The cache does not use the enumerated dictionary content, so loading the enumeration dictionary is not required. For more information on using **`DataDictionary`**, refer to the *Enterprise Transport API Reference Manual*.

After the **`DataDictionary`** loads, it is set to a cache instance using a key (an arbitrary string identifier assigned by the application to name the dictionary). The key allows multiple cache instances to share the same dictionary. After the first setting of a dictionary, it can be set to additional cache instances by simply providing the same key on additional settings. For a list of methods used in setting a dictionary to a cache, refer to Section 10.2.4.1.

The cache builds its own field definition database from the **`DataDictionary`** definitions. After setting, the application does not need to retain the dictionary object, because the cache does not refer to the **`DataDictionary`** used during the setting. In typical usage, the application will likely retain the dictionary for use with other encoding and decoding operations.

NOTE: A cache can be set to a dictionary only once during its lifetime. While a cache cannot be switched to a new dictionary, the dictionary in use can be extended with new definitions. Refer to Section 10.2.4.3.

10.2.4.1 Setting Functions

METHOD NAME	DESCRIPTION
setDictionary	This method sets a **DataDictionary** to a cache instance. Use this method the first time a dictionary is set to a cache. The application must provide a key parameter to this method to name the dictionary for future reference. This key is used in future setting operations when the application wants to share a dictionary between cache instances or to extend the definitions in the dictionary. The first time a particular key is used with this method will be the initial setting of that dictionary to a cache. The second time the same key is used in this method; it will reload the field definitions from the given **DataDictionary** object, enabling the dictionary to be extended. Refer to Section 10.2.4.3.
setSharedDictionaryKey	Use this method when sharing a dictionary among multiple caches. This method sets a cache (identified by the dictionary key name) to a previously set dictionary (identified by the dictionary key name). To share a dictionary, the dictionary must have previously had an initial setting to another cache using the **setDictionary** method. This method does not require the **DataDictionary** object, since that was already loaded during the initial setting with this dictionary key.

Table 191: Methods for Setting Dictionary to Cache

10.2.4.2 Setting Example

In the following example, two cache instances are created and set to a single, shared field dictionary.

```
PayloadCacheConfigOptions cacheConfig = CacheFactory.createPayloadCacheConfig();
cacheConfig.maxItems(0); /* unlimited */

/* For simplicity in this code fragment, CHK is assumed to be a macro for error handling (performing
        cleanup and returning from method). */

/* create cache instances */
CacheError cacheError = CacheFactory.createCacheError();
PayloadCache cacheInstance1 = CacheFactory.createPayloadCache(cacheConfig, cacheError);
if (cacheInstance1 == null)
{
    System.out.println("CacheFactory.createPayloadCache failure: " + cacheError.text());
    CHK(cacheError.errorId());
}

PayloadCache cacheInstance2 = CacheFactory.createPayloadCache(cacheConfig, cacheError);
if (cacheInstance2 == null)
{
    System.out.println("CacheFactory.createPayloadCache failure: " + cacheError.text());
    CHK(cacheError.errorId());
}

/* Load an RDM Field Dictionary object from file: set to each cache. */
DataDictionary dataDictionary = CodecFactory.createDataDictionary();
```

```
com.refinitiv.eta.transport.Error error = TransportFactory.createError();
int ret = dataDictionary.loadFieldDictionary("RDMFieldDictionary", error); CHK(ret);

String dictionaryKey = "SharedKey1";

/* Initial setting of the dictionary to the first cache */
ret = cacheInstance1.setDictionary(dataDictionary, dictionaryKey, cacheError); CHK(ret);
/* Shared setting of the same dictionary to the second cache */
ret = cacheInstance2.setSharedDictionaryKey(dictionaryKey, cacheError); CHK(ret);
/* The dataDictionary can be destroyed after setting, but is typically retained by the application for
        encoding and decoding. */

/* Two cache instances are now ready for applying and retrieving data */

/* Cleanup */
cacheInstance1.destroy(); /* destroys all entries and the cache instance */
cacheInstance2.destroy();
```

Code Example 33: Creating Cache and Setting to Dictionary

10.2.4.3 Extending the Cache Field Dictionary

While a cache can only be set to a single dictionary during its lifetime, the set of field definitions defined by the dictionary can be extended. This is accomplished by reloading the cache field definition database with another call to the **`setDictionary`** method. When extending the field dictionary, the **`DataDictionary`** must contain the original field definitions and any new definitions the application wishes to use. Changes or deletions to the original field definitions are not supported; only additions are allowed. Using the same **`PayloadCache`** instance and dictionary key that were previously set, call the **`setDictionary`** method again with extended dictionary object.

NOTE: When extending a field dictionary that is shared, all caches sharing that same dictionary key will see the extension with only a single call to **`setDictionary`**. There is no need to set the shared dictionary key again to each cache after a dictionary is extended.

10.2.5 Payload Cache Utilities

Use the following methods for managing cache instances. These utilities provide a count of the cache entries and a list of handles to each cache entry.

METHOD NAME	DESCRIPTION
entryCount	Returns the number of item payload entries in this cache instance
entryList	Populates an array list for this cache instance. Because each cache entry is likely associated with an entry in the application's item list, an application would typically manage the entire set of entry instances. This utility provides access to the entire entry instance list if needed.
clear	Destroys all entries in the cache instance. The empty cache can be reused and remains bound to it's data dictionary.

Table 192: `PayloadCache` Utility Methods

10.3 Payload Cache Entries

A payload cache entry stores a single OMM container (whose containers are defined by **DataTypes**). While a cache entry can store any arbitrary OMM data, the primary use case is to maintain the last known value of an item data stream by applying the sequence of refresh and update messages in the stream to the cache entry. Initial data applied to a container must be a refresh message payload, which will define the container type to be stored (e.g. Map). As refresh and update messages from the item stream are applied to the cache entry, the cache decodes the OMM data and sets the current value by following the OMM rules for the container (e.g., adding, deleting, or updating map entries in a Map, or updating fields in a field list). The last value of the data stream can be retrieved from cache at any time as an encoded OMM container.

10.3.1 Managing Payload Cache Entries

Payload cache entries are created within a cache instance. Use the **CacheFactory.createPayloadEntry** method to create a cache entry instance. You cannot move entries between different cache instances, due to their dependency on the field dictionary set to the cache where they are created.

Cache entries only store the payload container of an item. Maintain other item data (e.g. message key attributes, domain, state) as needed in an item list managed by the application, which will identify the source or sink associated with the cache entry data. This item list will likely include the **PayloadEntry** instance if the payload of the item is cached.

For a list of basic utilities provided by the payload cache to manage the collection of entries in the cache, refer to Section 10.2.5.

Use the following methods to manage cache entries:

METHOD NAME	DESCRIPTION
CacheFactory.createPayloadEntry	This method returns a newly created entry in the cache defined by the given **PayloadCache** instance. This method will return a null instance if it cannot create the entry (e.g., if the maximum number of entries as defined in **PayloadCacheConfigOptions** would be exceeded).
destroy	This method destroys the cache entry defined **PayloadEntry** instance and removes it from its cache.
clear	This method deletes any data in the cache entry instance and returns the entry to its initial state. The entry itself remains in the cache and can be re-used.

Table 193: Payload Cache Entry Management Methods

10.3.2 Applying Data

Data is applied to a cache entry from the payload of an OMM message by using the **apply** method. The decoded **Msg** and an **DecodeIterator** are passed to the apply method. The iterator (positioned at the start of the encoded payload data **Msg.encodedDataBody**) will be used to decode the OMM data so that the cache entry data can be set or updated.

Some caching behaviors are controlled by flags in the **Msg**. When a **RefreshMsg** is applied to the cache entry, the following **RefreshMsgFlags** take effect:

- **CLEAR_CACHE**: Cache entry data will be cleared prior to applying this message.
- **DO_NOT_CACHE**: The payload will not be applied to the cache entry.

When an **UpdateMsg** is applied to cache, the following **UpdateMsgFlags** take effect:

- **DO_NOT_CACHE**: The payload data will not be applied to the cache entry.
- **DO_NOT_RIPPLE**: When applying the data, entry rippling is not performed.

The following example demonstrates how to create a payload entry in a cache instance and apply the payload of a **Msg** to the cache entry.

```
/* For simplicity in this code fragment, CHK is assumed to be a macro for error handling (performing
        cleanup and returning from method). */

CacheError cacheError = CacheFactory.createCacheError();
PayloadEntry entryInstance = CacheFactory.createPayloadEntry(cacheInstance, cacheError);
if (entryInstance == null)
{
    System.out.println("Error " + cacheError.errorId() + " creating cache entry: " + cacheError.text());
    CHK(cacheError.errorId());
}

/* Apply buffer containing an encoded Msg to cache entry */
int applyBufferToCache(Channel channel, TransportBuffer buffer, PayloadEntry entryInstance)
{
    /* Perform message decoding. */
    DecodeIterator dIter = CodecFactory.createDecodeIterator();
    dIter.clear();
    dIter.setBufferAndRWFVersion(buffer, channel.majorVersion(), channel.minorVersion());
    Msg msg = CodecFactory.createMsg();
    int ret = msg.decode(dIter);
    if (ret < CodecReturnCodes.SUCCESS)
    {
        System.out.println("Failure decoding message from buffer");
        CHK(ret);
    }

    /* Apply the decoded Msg to cache, with iterator positioned at the start of the payload */
    CacheError cacheError = CacheFactory.createCacheError();
    ret = entryInstance.apply(dIter, msg, cacheError);
    if (ret < CodecReturnCodes.SUCCESS)
    {
        System.out.println("Error " + cacheError.errorId() + " applying data to cache entry: " +
                cacheError.text());
        CHK(ret);
    }
```

```
    CHK(ret);
}
```

Code Example 34: Applying Data to a Payload Cache Entry

10.3.3 Retrieving Data

Data is retrieved from a cache entry as an encoded OMM container by using the **retrieve** method. The application provides the data buffer (via an **EncodeIterator**) where the container will be encoded. The retrieve method supports both encoding scenarios. When using **Msg.encodedDataBody**, the encoded content retrieved from the cache entry can be set on the **Msg** data body. If using **encodeInit** and **encodeComplete** encoding, the cache retrieve method can encode the message payload prior to **encodeComplete**.

There are two options for using the **retrieve** method. For single-part retrieval, the buffer provided by the application must be large enough to hold the entire encoded container. For multi-part retrieval, the application makes a series of calls to **retrieve** to get the OMM container in fragments (e.g., a sequence of maps are retrieved which together contain the entire set of map entries for the container). In this usage, the optional **PayloadCursor** instance is required to maintain the state of the multi-part retrieval. Container types **FieldList** and **ElementList** cannot be fragmented, so the buffer size must be large enough to retrieve the entire container.

The following methods describe data-related operations on a cache entry.

METHOD NAME	DESCRIPTION
dataType	Returns the **DataType** stored in the cache entry instance. When initially created (or after the entry is cleared), the data type will be **UNKNOWN**. The data type is defined by the container type of the first refresh message applied to the entry.
apply	Applies the OMM data in the payload of the **Msg** to the cache entry instance. The first message applied must be a refresh message (class **REFRESH**).
retrieve	Retrieves data from the cache entry by encoding the OMM container into the buffer provided with the **EncodeIterator** given by the application. The buffer can be **Buffer** or **TransportBuffer**. For single-part retrieval, the **PayloadCursor** parameter is optional. For details on multi-part retrieval, refer to Section 10.3.3.1.

Table 194: Methods for Applying and Retrieving Cache Entry Data

10.3.3.1 Multi-Part Retrieval

For data types that support fragmentation, the container can be retrieved in multiple parts by calling **retrieve** until the complete container is returned. To support multi-part retrieval, the optional **PayloadCursor** parameter is required when calling **retrieve**. The cursor is used to maintain the position where the next retrieval will resume. The application must check the state of the cursor after each call to **retrieve** to determine when the retrieval is complete. The following methods are needed when using the payload cursor.

METHOD NAME	DESCRIPTION
CacheFactory.createPayloadCursor	Creates a cursor for optional use in the **retrieve** method (required for multi-part retrieval). Returns the **PayloadCursor** instance.
destroy	Destroys the cursor instance.

Table 195: Methods for Using the Payload Cursor

METHOD NAME	DESCRIPTION
clear	Clears the state of the cursor instance. Whenever retrieving data from a cache entry, the cursor must be cleared prior to the first call to **retrieve**. Clearing the cursor also allows it to be reused with a retrieval on a different container.
isComplete	Returns the completion state of a retrieval where the **PayloadCursor** instance was used. The state must be checked after each call to **retrieve** to determine whether additional data needs to be encoded for the cache entry container. When the cursor state is complete, the entire container of the cache entry has been retrieved.

Table 195: Methods for Using the Payload Cursor (Continued)

10.3.3.2 Buffer Management

In multi-part usage, the size of the buffer used in the calls to **retrieve** will affect how many fragments are required to retrieve the entire image of the cache entry. The retrieve method will continue to encode OMM entries from the cache container until it runs out of room in the buffer to encode the next entry. To progress during a multi-part retrieval, the buffer size must be at least large enough to encode a single OMM entry from the payload container. For example, if retrieving a map in multiple parts, the buffer must be large enough to encode at least one **MapEntry** on each retrieval.

There are three general outcomes when using the **retrieve** method:

- Full cache container is encoded into the buffer. This can occur with or without the use of the optional **PayloadCursor** instance. If used in this scenario, the cursor state would indicate the retrieval is complete.
- Partial container encoded into the buffer. This is only possible when using the **PayloadCursor** instance for container types that support fragmentation. The application must check the cursor to test whether this is the final part.
- No data encoded into container due to insufficient buffer size. This can occur with or without the use of the optional **PayloadCursor** instance. The application may retrieve again with a larger buffer.

10.3.3.3 Example: Cache Retrieval with Multi-Part Support

The following example illustrates data retrieval from a cache entry, which supports multi-part encoding of a container.

```
/*Code fragment showing use of retrieve for multi-part retrieval.*/
com.refinitiv.eta.transport.Error error = TransportFactory.createError();
TransportBuffer buffer = channel.getBuffer(DEFAULT_BUFFER_SIZE, false, error);

int ret;
CacheError cacheError = CacheFactory.createCacheError();
PayloadCursor cursorInstance = CacheFactory.createPayloadCursor();
cursorInstance.clear();
EncodeIterator eIter = CodecFactory.createEncodeIterator();
while (!cursorInstance.isComplete())
{
    eIter.clear();
    eIter.setBufferAndRWFVersion(buffer, channel.majorVersion(), channel.minorVersion());

    /* entryInstance created outside the scope of this code fragment */
    ret = entryInstance.retrieve(eIter, cursorInstance, cacheError);
    if (ret == CodecReturnCodes.SUCCESS)
    /* buffer is big enough to hold whole container data. Application can used encoded data, e.g. set the
            payload on Msg.encodedDataBody and encode a message to be transmitted. */
    else if (ret == CodecReturnCodes.BUFFER_TOO_SMALL)
        /* Increase buffer size and reallocate buffer. */
    else
        /* Handle terminal error condition. See cacheError.text() for additional information. */
}

cursorInstance.destroy();
```

Code Example 35: Cache Retrieval with Multi-Part Support

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: ETAJ367UMVAC.220
Date of issue: October 2022