Elektron Message API C++ Edition 3.5.0.G1

ELEKTRON MESSAGE API DEVELOPERS GUIDE

Document Version: 3.5.0.1
Date of issue: 30 June 2020
Document ID: EMAC350UM.201

REFINITIV™

Refinitiv, by publishing this document, does not guarantee that any information contained herein is and will remain accurate or that use of the information will ensure correct and faultless operation of the relevant service or equipment. Refinitiv, its agents, and its employees, shall not be held liable to or through any user for any loss or damage whatsoever resulting from reliance on the information contained herein.

Contents

1 Guide Introduction

1.1 About this Manual

This document is authored by Elektron Message API architects and programmers. Several of its authors have designed, developed, and maintained the Elektron Message API product and other Refinitiv products which leverage it.

This guide documents the functionality and capabilities of the Elektron Message API C++ Edition. The Elektron Message API can also connect to and leverage many different Refinitiv and customer components. If you want the Elektron Message API to interact with other components, consult that specific component's documentation to determine the best way to configure and interact with these other devices.

1.2 Audience

This document is intended to provide detailed yet supplemental information for application developers writing to the Message API.

1.3 Programming Language

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

1.4 Acronyms and Abbreviations

ACRONYM	MEANING
ADH	Advanced Data Hub is the horizontally scalable service component within the Refinitiv Data Management System (RDMS) providing high availability for publication and contribution messaging, subscription management with optional persistence, conflation and delay capabilities.
ADS	Advanced Distribution Server is the horizontally scalable distribution component within the Refinitiv Data Management System (RDMS) 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
EDP	Elektron Data Platform
EED	Elektron Edge Device
EMA	Elektron Message API, referred to simply as the Message API. EMA is part of the Elektron SDK.
ETA	Elektron Transport API, referred to simply as the Transport API. Formerly referred to as UPA. ETA is a low-level API, currently used by the Refinitiv Data Management System (and its dependent APIs) for optimized distribution of OMM/RWF data. ETA is part of the Elektron SDK.
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol (Secure)
JSON	JavaScript Object Notation

Table 1: Acronyms and Abbreviations

ACRONYM	MEANING
OMM	Open Message Model
QoS	Quality of Service
RDM	Reuters Domain Model
RDMS	Refinitiv Data Management Solutions (formerly called the Thomson Reuters Enterprise Platform, or TREP); includes the RDMS infrastructure (i.e., ADS, ADH) and Refinitiv Data Platform APIs.
Reactor	The Reactor is a low-level, open-source, easy-to-use layer above ETA. It offers heartbeat management, connection and item recovery, and many other features to help simplify application code for users.
RMTES	Reuters Multi-Lingual Text Encoding Standard
RSSL	Reuters Source Sink Library
RTT	Round Trip Time, this definition is used for round trip latency monitoring feature.
RWF	Reuters Wire Format, a Refinitiv proprietary format.
R-DFD	Reuters Data Feed Direct
UML	Unified Modeling Language
UTF-8	8-bit Unicode Transformation Format

Table 1: Acronyms and Abbreviations

1.5 References

1. Elektron Message API C++ Edition *RDM Usage Guide*
2. *API Concepts Guide*
3. *Elektron Message API C++ Configuration Guide*
4. *EMA C++ Edition Reference Manual*
5. Transport API C Edition *Value Added Components Developers Guide*
6. *Transport API C Edition Developers Guide*
7. 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 apidocumentation@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 apidocumentation@refinitiv.com address.

1.7 Document Conventions

This document uses the following types of conventions:

- C++ classes, methods, in-line code snippets, and types are shown in **`orange, 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:

```
AppClient client;
    OmmConsumer consumer( OmmConsumerConfig().operationModel(
    OmmConsumerConfig::UserDispatchEnum ).host( "localhost:14002" ).username( "user" ) );
    consumer.registerClient( ReqMsg().domainType( MMT_MARKET_BY_PRICE ).serviceName(
    "DIRECT_FEED" ).name( "BBH.ITC" ).privateStream( true ), client );
    unsigned long long startTime = getCurrentTime();
```

2 Product Overview

2.1 EMA Product Description

The Elektron Message API is a data-neutral, multi-threaded, ease-of-use API providing access to OMM and RWF data. As part of the Elektron Software Development Kit, or Elektron SDK, the EMA allows applications to consume and provide OMM data at the message level of the API stack. The message level is set on top of the transport level which is handled by the Elektron Transport API (also known as the UPA).

The Elektron Message API (EMA):

- Provides a set of easy-to-use and intuitive interfaces and features intended to aid in message-level application development. These interfaces simplify the setting of information in and getting information from OMM containers and messages. Other interfaces abstract the behavior of consumer-type and provider-type applications.
- Enables applications to source market data from, and provide it to, different components that support OMM and/or RWF (e.g. Elektron, RDMS, ATS, RDF-D, etc).
- Leaves a minimal code footprint in applications written to it. The design of the EMA and its interfaces allows application development to focus more on the application business logic than on the usage of the EMA.
- Includes training applications that provide basic, yet still functional, examples of EMA applications.
- Presents applications with simplified access to OMM messages and containers while providing all necessary transport level functionalities. Generally, EMA applications are meant to process market data items (e.g. open and receive item data or provide item data).
- Abstracts and hides all the transport level functionality minimizing application involvement to just optional transport level configuration and server address specification.
- Provides simple **`set`**- and **`get`**-type functionality to populate and read OMM containers and messages. EMA takes advantage of fluent interface design, which users can leverage to set disparate values of the same message or container by stringing respective interface methods together, one after the other. Fluent interfaces provide the means for visual code simplification which helps in understanding and debugging applications.

Transport level functionality is abstracted, specialized, and encapsulated by the EMA in a few classes whose functionality is implied by their class name.

2.2 Product Documentation and Learning EMA

When learning the EMA, Refinitiv recommends you set up a sandbox environment where developers can experiment with various iterations of EMA applications. EMA is designed to facilitate a hands-on (experiment-based) learning experience (versus a documentation-based methodology). To support a hands-on learning methodology, the EMA package provides a set of training examples which showcase the usage of EMA interfaces in increasing levels of complexity and sophistication. While coding and debugging applications, developers are encouraged to refer to the *EMA C++ Edition Reference ManualC++* and/or to the features provided by their IDE (e.g., IntelliSense).

NOTE: EMA application developers should already be familiar with OMM and Market Data distribution systems.

2.2.1 Consumer Examples

The complexity of a consumer example is reflected in its series number as follows:

- 100-series examples simply open an item and print its received content to the screen (using the **`Data::toString()`** method). Applications in this series illustrate EMA support for stringification, containers, and primitives. Though useful for learning, debugging, and writing display applications, stringification by itself is not sufficient to develop more sophisticated applications.
- The 200-series examples illustrate how to extract information from OMM containers and messages in native data formats, (e.g., UInt64, EmaString, and EmaBuffer).
- The 300- and 400- series examples depict usage of particular EMA features such as posting, generic message, programmatic configuration, and etc.

2.2.2 Provider Examples

The complexity of an example is reflected in its series number. Each provider type (i.e., non-interactive versus interactive) has its own directory structure in the product package:

- 100-series examples simply create streaming items and submit their refreshes and updates. Applications in this series use the hardcoded EMA configuration.
- The 200-series examples showcase the submission of multiple, streaming items from different market domains. Applications in this series use the **EmaConfig.xml** file to modify its configuration.
- The 300- and 400- series examples depict usage of particular EMA features such as user control of the source directory domain, login streaming, connection recovery, programmatic configuration, and etc.

2.3 Supported Features

FEATURE	DESCRIPTION
New in 3.5 Round Trip Time (RTT) monitoring support and configuration option added	The provider can initiate RTT monitoring by sending RTT requests to the consumer over the Generic Login stream. In EMA, the consumer's automatic support of this feature can be configured either via the configuration file or programmatically.
New in 3.5 Port information added for Consumer and NIProvider	**`ChannelInformation.port()`** returns the port to which the consumer or NIProvider is connected.
New in 3.5 Support for the WebSocket protocol over ETA	EMA C++ Edition supports the Web Socket transport protocol over the Elektron Transport API.
New in 3.3.1 Ability to manage Client_ID and passwords for cloud connections Ability to clone and copy messages Centralized management of multiple OAuth tokens	• EMA no longer stores an application's passwords for the EDP Gateway. The application must manage its own passwords. • EMA supports multi-channel OAuth credential management. • EMA requires the use of a generated **Client_ID** from the AppGenerator for use when obtaining access and refresh tokens for cloud connections. • EMA messages can now be cloned and copied to decode the payload outside of message callbacks
New in 3.3.0 Encryption and Cloud Connectivity	You can encrypt content sent over EMA and configure Session Management behaviors when connecting to cloud infrastructure.

Table 2: Supported Features

FEATURE	DESCRIPTION
New in 3.1.2 Support for Programmatic Configuration of *Provider*	You can use EMA to programmatically configure parameters for interactive and non-interactive providers in the EMA configuration file. For details, refer to the *Elektron Message API C++ Configuration Guide*.
New in 3.1.2 Support for HTTP and ENCRYPTED Connection Types	EMA provides support for `ChannelType::RSSL_HTTP` and `ChannelType::RSSL_ENCRYPTED` connection types for EMA consumers. For further details, refer to Section 4.3.
New in 3.1 Domain Representations for Admin Domain Login Messages	EMA provides domain representations for Admin Domain Login Messages. Domain Representations are easy-to-use objects which can setup and return encoded OMM Messages without extensive effort. For further details, refer to *EMA's RDM Usage Guide*.
New in 3.1 RDMS Authentication	EMA provides support for the RDMS Authentication feature for consumers and non-interactive providers. For more information, refer to *EMA's RDM Usage Guide* and to the *RDMS Authentication user manual*[a].
New in 3.1 Enhanced Login Stream Handling	EMA applications can register for Login events when they create OmmConsumer or non-interactive OmmProvider objects.
New in 3.0.6 Interactive Provider	EMA provides interactive provider connectivity (e.g., for ADH or other directly-connected consumers).
New in 3.0.3 Non-interactive Provider	Applications can connect to an ADH RDMS component to non-interactively provide item data.
Connection Failover	You can specify a list of failover servers via the **ChannelSet** configuration. If a connection attempt fails, EMA attempts to connect to the next channel in the **ChannelSet** list. Both EMA consumers and non-interactive providers support the connection failover feature.
ADS Multicast	Applications can connect to the ADS multicast component by specifying the connection type **RSSL_RELIABLE_MCAST**.
Default Admin Domain Messages	The EMA consumer uses default login, directory, and dictionary requests when connecting to a provider or ADS: • The Login request uses the current user's name and defaults all other login attributes. • The Directory request calls for all services and filters. • RDM dictionaries are requested from the first available service that accepts requests. The EMA non-interactive provider uses the default login request and configured directory refresh when connecting to ADH: • The login request uses the current user's name and defaults all other login attributes. • The directory refresh message defaults all message attributes as well as status, while its payload is either hardcoded or read from the EMA configuration. The EMA interactive provider can use default, preconfigured directory and dictionary refresh messages. • The directory refresh message defaults appropriate message attributes as well as status, while its payload is either hard coded or read from the EMA configuration. • The dictionary refresh message is handled based on either its hard-coded configuration or read from the EMA configuration.
Configurable Admin Domain Messages	EMA provides the means to modify default Admin domain messages.
Batch Request	A consumer application can use a single request message to specify interest in multiple items via the item list.
Dynamic View	A consumer application can specify a subset of fields or elements for a particular item.

Table 2: Supported Features (Continued)

FEATURE	DESCRIPTION
Optimized Pause and Resume	A consumer application can send a request to the server to pause and resume item stream.
Single Open	The EMA supports application-selected, single-open functionality.
RMTES Decoder	The EMA provides a built-in RMTES decoder. If needed, the application can cache **RmtesBuffer** objects and apply all received changes to them.
Data::toString()	Prints all OMM containers, primitives, and messages to screen in a standardized output format (called “stringification”).
Data::getAsHex()	Applications can obtain binary representations of all OMM containers, primitives, and messages.
Programmatic Configuration	The application can programmatically specify and overwrite its EMA configuration for the consumer for the consumer, NiProvider, and IProvider.
File Configuration	An EMA configuration can be specified in an **EmaConfig.xml** file.
Tunnel Stream (also known as a Qualified Stream)	EMA supports private streams, with additional associated behaviors (e.g., end-to-end authentication, reliable delivery, and flow control).
File Logger	EMA allows the application to turn on / off EMA logging, to specify the desired severity level of error reporting, and to specify whether to send logger messages to stdout or a file.
Connected Component Information	Whenever EMA connects to a component, EMA sends its component version information, and if the connection is successful, EMA logs the component’s version.

Table 2: Supported Features (Continued)

a. The *RDMS Authentication User Manual* can be obtained on MyRefinitiv under the DACS product content set.

2.4 Product Architecture

2.4.1 EMA Consumer Architecture

The EMA incorporates the ValueAdded Reactor component (called the Transport API VA Reactor) from the Transport API, which provides the watchlist and transport-level functionality. The EMA wraps up the reactor component in its own class of `OmmConsumer`. `OmmConsumer` provides interfaces to open, modify, and close market items or instruments, as well as submit Post and Generic messages. To complete the set of consumer application functionalities, the `OmmConsumer` class provides the `dispatch()` method. Depending on its design and configuration, an application might need to call this method to dispatch received messages. The `OmmConsumerConfig` class configures the reactor and `OmmConsumer`.

The `OmmConsumerClient` class provides the callback mechanism for EMA to send incoming messages to the application. The application needs to implement a class inheriting from the `OmmConsumerClient` class to receive and process messages. By default, `OmmConsumerClient` callback methods are executed in EMA's thread of control. However, you can use the `OmmConsumerConfig::operationModel()` interface to execute callback methods on the application thread. If you choose to execute callback methods in this manner, the application must also call the `OmmConsumer::dispatch()` method to dispatch received messages.

While the `OmmConsumer` class throws an `OmmException` to report error conditions, the `OmmConsumerErrorClient` class provides an alternate reporting mechanism via callbacks. To use the alternate error reporting, pass the `OmmConsumerErrorClient` on the constructor of the `OmmConsumer` class, which switches the error reporting from exception throwing to callbacks. In addition to its error reporting mechanisms, EMA provides a logger mechanism which is useful in monitoring EMA behavior and debugging any issues that might arise.

The EMA consumer will always have at least one thread, which is implemented by the VA Reactor and runs the internal, VA Reactor logic. For details on this thread, refer to the *Transport API C Edition Value Added Component Developers Guide*. Additionally, you can configure the EMA to create a second, internal thread to dispatch received messages. To create a second thread, set the `OmmConsumerConfig` operation model to `OmmConsumerConfig::ApiDispatchEnum`. If the `OmmConsumerConfig` operation model is set to the `OmmConsumerConfig::UserDispatch`, the EMA will not run a second thread. Without running a second thread, the application is responsible for calling the `Ommconsumer::dispatch()` method to dispatch all received messages.

⚠ **WARNING!** If the application delays in dispatching messages, it can result in slow consumer behavior.

2.4.2 EMA Provider Architecture

The EMA provider incorporates the Value Added (VA) Reactor component from the Transport API, which provides transport-level functionality. The EMA wraps the reactor component in its own class of `OmmProvider`. `OmmProvider` provides interfaces to submit item messages as well as handling login, directory, and dictionary domains (depending on EMA's specific provider role). To complete the set of provider functionalities, the `OmmProvider` class provides the `dispatch()` method. Depending on its design and configuration, an application might need to call this method to dispatch received messages. The provider configuration class (i.e., `OmmNiProviderConfig` or `OmmIProviderConfig`) class configures both the reactor and `OmmProvider`.

EMA sends incoming messages to the application using the `OmmProviderClient` callback mechanism. To receive and process messages, the application needs to implement a class that inherits from the `OmmProviderClient` class. By default, `OmmProviderClient` callback methods are executed in EMA's thread of control. However, you can use either the `OmmNiProviderConfig::operationModel()` or `OmmIProviderConfig::operationModel()` interface to execute callback methods on the application's thread, in which case the application must also call the `OmmProvider::dispatch()` method to dispatch received messages.

While the `OmmProvider` class throws an `OmmException` to report error conditions, the `OmmProviderErrorClient` class provides an alternate reporting mechanism via callbacks. To use the alternate error reporting, pass the `OmmProviderErrorClient` on the constructor of the `OmmProvider` class, which switches the error reporting from exception throwing to callbacks. In addition to its error-reporting mechanisms, EMA provides a logger mechanism which you can use to monitor EMA behavior and debug any issues that arise.

An EMA provider must always have at least one thread, which is implemented by the VA Reactor and runs the internal, VA Reactor logic. For details on this thread, refer to the *Transport API C Edition Value Added Component Developers Guide*. Additionally, you can configure EMA to create a second internal thread over which to dispatch received messages:

- For non-interactive providers, set the `OmmNiProviderConfig` operation model to `OmmNiProviderConfig::ApiDispatchEnum`. If the operation model is set to `OmmNiProviderConfig::UserDispatchEnum`, EMA will not run a second thread.
- For interactive providers, set the `OmmIProviderConfig` operation model to `OmmIProviderConfig::ApiDispatchEnum`. If the operation model is set to `OmmIProviderConfig::UserDispatchEnum`, EMA will not run a second thread.

Without running a second thread, the application is responsible for calling the **`OmmProvider::dispatch()`** method to dispatch all received messages.

The EMA provider includes an internal, hard-coded, and configurable initial source directory refresh message. The application can either use the internal hard-coded source directory, configure its own internal one via the **EmaConfig.xml** file, or programmatically create one and/or disable the internal one. To disable the internal source directory message:

- When running EMA as a non-interactive provider: the application must set **`OmmNiProviderConfig::UserControlEnum`** through the **`OmmNiProviderConfig::adminControlDirectory()`** method.
- When running EMA as an interactive provider: the application must set **`OmmIProviderConfig::UserControlEnum`** through the **`OmmIProviderConfig::adminControlDirectory()`** method. Additionally, you can configure the ability to disable internal dictionary responses by setting **`OmmIProviderConfig::UserControlEnum`** through the **`OmmIProviderConfig::adminControlDictionary()`** method.

NOTE: If the user control is enabled, the application is responsible for sending the response messages.

An EMA provider also supports the programmatic configuration of a source directory refresh of dictionary information, which overrides any configuration in **EmaConfig.xml**. To programmatically configure a source directory refresh:

- When running EMA as a non-interactive provider: the application must set **`OmmNiProviderConfig::ApiControlEnum`** through the **`OmmNiProviderConfig::adminControlDirectory()`** method. An EMA non-interactive provider does not support programmatically configuring dictionary information.
- When running EMA as an interactive provider: the application must set **`OmmIProviderConfig::ApiControlEnum`** through the **`OmmIProviderConfig::adminControlDirectory()`** method. Additionally, you can programmatically configure dictionary information, which overrides any dictionary information defined from **EmaConfig.xml**. To programmatically configure dictionary information, set **`OmmIProviderConfig::ApiControlEnum`** through the **`OmmIProviderConfig::adminControlDictionary()`** method.

2.4.3 EMA Codec Architecture

The EMA Codec uses the Elektron Transport API decoding and encoding functions to read and populate OMM containers and messages. Each OMM container and message is represented by a respective EMA interface class, which provides relevant methods for setting information on, and accessing information from, these containers and messages. All classes representing OMM containers, messages, and primitives inherit from the common parent class of **`Data`**. Through such inheritance, classes provide the same basic, common, and easy to use functionality that applications might expect from them (e.g., printing contained data using **`toString()`**).

2.5 Tunnel Streams

By leveraging the Transport API Value Added Reactor, the EMA allows users to create and use special tunnel streams. A tunnel stream is a private stream that has additional behaviors associated with it, such as end-to-end line of sight for authentication and reliable delivery. Because tunnel streams are founded on the private streams concept, these are established between consumer and provider endpoints and then pass through intermediate components, such as RDMS or EED.

The user creating the tunnel stream sets any additional behaviors to enforce, which EMA sends to the provider application end point. The provider end point acknowledges creation of the stream as well as the behaviors that it will also enforce on the stream. Once this is accomplished, the negotiated behaviors will be enforced on the content exchanged via the tunnel stream.

The tunnel stream allows for multiple substreams to exist, where substreams flow and coexist within the confines of a specific tunnel stream. In the following diagram, imagine the tunnel stream as the orange cylinder that connects the Consumer application and the Provider application. Notice that this passes directly through any intermediate components. The tunnel stream has end-to-end line of sight so the Provider and Consumer are effectively talking to each other directly, although they are traversing multiple devices in the system. Each of the black lines flowing through the cylinder represent a different substream, where each substream is its own independent stream of information. Each of these could be for different market content, for example one could be a Time Series request while another could be a request for Market Price content.

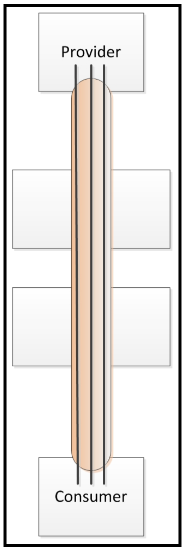


Figure 1. Tunnel Stream

3 OMM Containers and Messages

3.1 Overview

The EMA supports a full set of OMM containers, messages, and primitives (e.g. **`FieldList`**, **`Map`**, **`RefreshMsg`**, **`Int`**). For simplicity, EMA uses:

- The “set / add” type of functionality to populate OMM containers, messages, and primitives
  - Set functionality is used to specify variables that occur once in an OMM container or message.
  - Add functionality is used to populate entries in OMM containers.
  - Set and add type methods return a reference to the modified object (for fluid interface usage).
- The “get” type of functionality to read and extract data from OMM containers, messages, and primitives.EMA uses a simple iterative approach to extract entries from OMM containers, one at a time. Applications iterate over every OMM container type in the same way.

While iterating, an application can apply a filtering mechanism. For example, while iterating over a **`FieldList`**, the application can specify a field ID or field name in which it is interested; the EMA skips entries without matching identification. Individual container entries are extracted during iteration. Depending on the container type, the entry may contain:

- Its own identity (e.g., field id)
- An action to be applied to the received data (e.g., add action)
- Permission information associated with the received data
- An entry’s load and its **`data`** type.

The EMA has two different ways of extracting an entry’s load:

- Use ease-of-use interfaces to return references to contained objects (with reference type being based on the load’s data type)
- Use the **`getLoad`** interface to return a reference to the base **`Data`** class. The **`getLoad`** interface enables more advanced applications to use the down-cast operation (if desired).

For details on ease of use interfaces and the down-cast operation, refer to Section 3.3.

To provide compile time-type safety on the set-type interfaces, EMA provides the following, deeper inheritance structure:

- All classes representing primitive / intrinsic data types inherit from the **`Data`** class (e.g. **`OmmInt`**, **`OmmBuffer`**, **`OmmRmtes`**, etc.).
- **`OmmArray`** class inherits from the **`Data`** class. The **`OmmArray`** is treated as a primitive instead of a container, because it represents a set of primitives.
- **`OmmError`** class inherits from the **`Data`** class. **`OmmError`** class is not an OMM data type.
- All classes representing OMM containers (except **`OmmArray`**) inherit from the **`ComplexType`** class, which in turn inherits from the **`Data`** class (e.g., **`OmmXml`**, **`OmmOpaque`**, **`Map`**, **`Series`**, or **`Vector`**).
- All classes representing OMM messages inherit from the **`Msg`** class, which in turn inherits from the **`ComplexType`** class (e.g., **`RefreshMsg`**, **`GenericMsg`**, or **`PostMsg`**).

3.2 Classes

3.2.1 DataType Class

The **DataType** class provides the set of enumeration values that represent each and every supported OMM data type, including all OMM containers, messages, and primitives. Each class representing OMM data identifies itself with an appropriate **DataType** enumeration value (e.g., **DataType::FieldListEnum**, **DataType::RefreshMsgEnum**). You can use the **Data::getDataType()** method to learn the data type of a given object.

The **DataType** class list of enumeration values contains two special enumeration values, which can only be received when reading or extracting information from OMM containers or messages:

- **DataType::ErrorEnum**, which indicates an error condition was detected. For more details, refer to Section 3.2.5.
- **DataType::NoDataEnum**, which signifies a lack of data on the summary of a container, message payload, or attribute.

3.2.2 DataCode Class

The **DataCode** class provides two enumeration values that indicate the data's state:

- The **DataCode::NoCodeEnum** indicates that the received data is valid and application may use it.
- The **DataCode::BlankEnum** indicates that the data is not present and application needs to blank the respective data fields.

3.2.3 Data Class

The **Data** class is a parent abstract class from which all OMM containers, messages, and primitives inherit. **Data** provides interfaces common across all its children, which in turn enables down-casting operations. The **Data** class and all classes that inherit from it are optimized for efficiency and built so that data can be easily accessed. Though all primitive data types are represented by classes that inherit from the **Data** class, the ease-of-use interfaces do not return such references: all primitive data types are returned by their intrinsic representation.

⚠ **WARNING!** The **Data** class and all classes that inherit from it are designed as temporary and short-lived objects. For this reason, do not use them as storage or caching devices.

The EMA does not support immediately retrieving data from freshly created OMM containers or messages. The following code snippet demonstrates this restriction:

```
FieldList fieldList;

fieldList.addAscii( 1, "ascii" ).addInt( 10, 20 ).complete();

while ( fieldList.forth() )
{
    const FieldEntry& fieldEntry = fieldList.getEntry();

    ...
}
```

3.2.4 Msg Class

The **Msg** class is a parent class for all the message classes. It defines all the interfaces that are common across all message classes.

3.2.5 OmmError Class

The **OmmError** class is a special purpose class. It is a read only class implemented in the EMA to notify applications about errors detected while processing received data. This class enables applications to learn what error condition was detected. Additionally it provides the **getAsHex()** method to obtain binary data associated with the detected error condition. The sole purpose of this class is to aid in debugging efforts.

The following code snippet presents usage of the **OmmError** class while processing **ElementList**.

```
void decode( const ElementList& elementList )
{
    while ( elementList.forth() )
    {
        const ElementEntry& elementEntry = elementList.getEntry();

        if ( elementEntry.getCode() == Data::BlankEnum )
            continue;
        else
            switch ( elementEntry.getLoadType() )
            {
            case DataType::RealEnum:
                cout << elementEntry.getReal().getAsDouble() << endl;
                break;
            case DataType::ErrorEnum:
                cout << elementEntry.getError().getErrorCode() << "( " <<
                        elementEntry.getError().getErrorCodeAsString() << " )" << endl;
                break;
        }
    }
}
```

3.2.6 TunnelStreamRequest and ClassOfService Classes

The **TunnelStreamRequest** class specifies request information for use in establishing a tunnel stream. A tunnel stream is a private stream that provides additional functionalities such as user authentication, end-to-end flow control and reliable delivery. You can configure these features on a per-tunnel stream basis. The **ClassOfService** class specifies these features and some other related parameters. The identity of the tunnel stream is specified on the **TunnelStreamRequest** class.

3.3 Working with OMM Containers

EMA supports the following OMM containers: **ElementList**, **FieldList**, **FilterList**, **Map**, **Series**, and **Vector**.

Each of these classes provides set type interfaces for container header information (e.g., dictionary id, element list number, and the add-type interfaces for adding entries). You must set the container header and optional summary before adding the first entry.

Though it is treated as an OMM primitive, the **OmmArray** acts like a container and therefore provides add-type interfaces for adding primitive entries.

NOTE: OMM Container classes do perform some validation of their usage. If a usage error is detected, an appropriate **OmmException** will be thrown.

3.3.1 Example: Populating a FieldList Class

The following example illustrates how to populate a **FieldList** class with fluid interfaces.

```
try {
    FieldList fieldList;

    fieldList.info( 1, 1 )
        .addUInt( 1, 64 )
        .addReal( 6, 11, OmmReal::ExponentNeg2Enum )
        .addDate( 16, 1999, 11, 7 )
        .addTime( 18, 02, 03, 04, 005 )
        .complete();
} catch ( const OmmException & excp  ) {
    cout << excp << endl;
}
```

3.3.2 Example: Populating a **Map** Class Relying on the **FieldList** Memory Buffer

The following code snippet illustrates how to populate a **Map** class with summary data and a single entry containing a **FieldList**. In this example, the **FieldList** class uses its own memory buffer to store content while it is populated. This buffer later gets copied to the buffer owned by the **Map** class. This container population model applies to all OMM containers that might contain other containers, primitives, or messages.

```
try {
   FieldList fieldList;

   fieldList.addUInt( 1, 64 )
       .addReal( 6, 11, OmmReal::ExponentNeg2Enum )
       .addDate( 16, 1999, 11, 7 )
       .addTime( 18, 02, 03, 04, 005 )
       .complete();

   Map map;
   map .summary( fieldList ).addKeyAscii( "entry_1", MapEntry::AddEnum, fieldList
          ).complete();
} catch ( const OmmException& excp  ) {
   cout << excp << endl;
}
```

3.3.3 Example: Populating a Map Class Relying on the Map Class Buffer

The following example illustrates how to populate a **Map** class with a single entry containing a **FieldList**. In this case, the **FieldList** class uses the memory buffer owned by the **Map** class to store its own content while it is populated, therefore avoiding the internal buffer copy described in Section 3.3.2. This container population model applies to iterable containers only (e.g., OmmArray, ElementList, FieldList, FilterList, Map, Series, and Vector).

```
try {
    FieldList fieldList;

    Map map;
    fieldList.addUInt( 1, 64 )
        .addReal( 6, 11, OmmReal::ExponentNeg2Enum )
        .addDate( 16, 1999, 11, 7 )
        .addTime( 18, 02, 03, 04, 005 )
        .complete();

    map.addKeyAscii( "entry_1", MapEntry::AddEnum, fieldList );

        map.complete();
} catch ( const OmmException& excp  ) {
    cout << excp << endl;
}
```

3.3.4 Example: Extracting Information from a `FieldList` Class

In the following example illustrates how to use the **`FieldList::forth()`** method to extract information from the **`FieldList`** class by iterating over the class. The following code extracts information about all entries.

```
void decode( const FieldList& fieldList )
{
    if ( fieldList.hasInfo() )
    {
        Int16 dictionaryId = fieldList.getInfoDictionaryId();
        Int16 fieldListNum = fieldList.getInfoFieldListNum();
    }

    while ( fieldList.forth() )
    {
        const FieldEntry& fieldEntry = fieldList.getEntry();

        if ( fieldEntry.getCode() == Data::BlankEnum )
            continue;

        switch ( fieldEntry.getLoadType() )
        {
        case DataType::AsciiEnum :
            const EmaString& value = fieldEntry.getAscii();
            break;
        case DataType::IntEnum :
            Int64 value = fieldEntry.getInt();
            break;
        }
    }
}
```

3.3.5 Example: Application Filtering on the `FieldList` Class

In the following code snippet application filters or extracts select information from FieldList class. The FieldList::forth(Int16) method is used to iterate over the FieldList class. In this case only entries with field id of 22 will be extracted; all the other ones will be skipped.

```
void decode( const FieldList& fieldList )
{
    while ( fieldList.forth( 22 ) )
    {
        const FieldEntry& fieldEntry = fieldList.getEntry();

        if ( fieldEntry.getCode() == Data::BlankEnum )
            continue;

        switch ( fieldEntry.getLoadType() )
        {
        case DataType::AsciiEnum :
            const EmaString& value = fieldEntry.getAscii();
            break;
        case DataType::IntEnum :
            Int64 value = fieldEntry.getInt();
            break;
        }
    }
}
```

3.3.6 Example: Extracting FieldList information using a Downcast Operation

The following example illustrates how to extract information from a **`FieldList`** object using the down-cast operation.

```
void AppClient::decodeFieldList( const FieldList& fl )
{
    if ( fl.hasInfo() )
        cout << "FieldListNum: " << fl.getInfoFieldListNum() << " DictionaryId: " << fl
fl.getInfoDictionaryId() << endl;

    while ( fl.forth() )
    }
        cout << "Load" << endl;
        decode( fl.getEntry().getLoad() );
    }
}

void AppClient::decode( const Data& data )
{
    if ( data.getCode() == Data::BlankEnum )
        cout << "Blank data" << endl;
    else
        switch ( data.getDataType() )
```

```
      {
      case DataType::RefreshMsgEnum :
         decodeRefreshMsg( static_cast<const RefreshMsg&>( data ) );
         break;
      case DataType::UpdateMsgEnum :
         decodeUpdateMsg( static_cast<const UpdateMsg&>( data ) );
         break;
      case DataType::FieldListEnum :
         decodeFieldList( static_cast<const FieldList&>( data ) );
         break;
      case DataType::MapEnum :
         decodeMap( static_cast<const Map&>( data ) );
         break;
      case DataType::NoDataEnum :
         cout << "NoData" << endl;
         break;
      case DataType::TimeEnum :
         cout << "OmmTime: " << static_cast<const OmmTime&>( data ).toString() << endl;
         break;
      case DataType::DateEnum :
         cout << "OmmDate: " << static_cast<const OmmDate&>( data ).toString() << endl;
         break;
      case DataType::RealEnum :
         cout << "OmmReal::getAsDouble: " << static_cast<const OmmReal&>( data
                ).getAsDouble() << endl;
         break;
      case DataType::IntEnum :
         cout << "OmmInt: " << static_cast<const OmmInt&>( data ).getInt() << endl;
         break;
      case DataType::UIntEnum :
         cout << "OmmUInt: " << static_cast<const OmmUInt&>( data ).getUInt() << endl;
         break;
      case DataType::EnumEnum :
         cout << "OmmEnum: " << static_cast<const OmmEnum&>( data ).getEnum() << endl;
         break;
      case DataType::AsciiEnum :
         cout << "OmmAscii: " << static_cast<const OmmAscii&>( data ).toString() << endl;
         break;
      case DataType::ErrorEnum :
         cout << "Decoding error: " << static_cast<const OmmError&>( data
                ).getErrorCodeAsString() << endl;
         break;
      default :
         break;
      }
}
```

3.4 Working with OMM Messages

EMA supports the following OMM messages: **RefreshMsg**, **UpdateMsg**, **StatusMsg**, **AckMsg**, **PostMsg** and **GenericMsg**. As appropriate, each of these classes provide set and get type interfaces for the message header, permission, key, attribute, and payload information.

3.4.1 Example: Populating the GenericMsg with an ElementList Payload

The following example illustrates how to populate a **GenericMsg** with a payload consisting of an **ElementList**.

```
try {
    GenericMsg genMsg;

    genMsg.domainType( 200 ).name( "TR.N" ).serviceId( 234 ).payload( ElementList().addAscii(
            "entry_1", "value_1" ).complete() );
} catch ( const OmmException& excp  ) {
    cout << excp << endl;
}
```

3.4.2 Example: Extracting Information from the GenericMsg Class

The following example illustrates how to extract information from the **GenericMsg** class.

```
void decode( const GenericMsg& genMsg )
{
    if ( genMsg.hasName() )
        cout << endl << "Name: " << genMsg.getName();

    if ( genMsg.hasHeader() )
        const EmaBuffer& header = genMsg.getHeader();

    switch ( genMsg.getPayload().getDataType() )
    {
    case DataType::FieldListEnum :
        decode( genMsg.getPayload().getFieldList() );
        break
    }
}
```

3.4.3 Example: Working with the `TunnelStreamRequest` Class

The following code snippet demonstrates using the **`TunnelStreamRequest`** class in the consumer application to open a tunnel stream.

```
CosAuthentication cosAuthentication;
cosAuthentication.type( CosAuthentication::OmmLoginEnum );

CosDataIntegrity cosDataIntegrity;
cosDataIntegrity.type( CosDataIntegrity::ReliableEnum );

CosFlowControl cosFlowControl;
cosFlowControl.type( CosFlowControl::BidirectionalEnum ).recvWindowSize( 1200
       ).sendWindowSize( 1200 );

ClassOfService cos;
cos.authentication( cosAuthentication ).dataIntegrity( cosDataIntegrity ).flowControl(
       cosFlowControl );

TunnelStreamRequest tsr;
tsr.classOfService( cos ).domainType( MMT_SYSTEM ).name( "TUNNEL" ).serviceName( "DIRECT_FEED" );
```

4 Consumer Classes

4.1 OmmConsumer Class

The `OmmConsumer` class is the main consumer application interface to the EMA. This class encapsulates watchlist functionality and transport level connectivity. It provides all the interfaces a consumer-type application needs to open, close, and modify items, as well as submit messages to the connected server (both `PostMsg` and `GenericMsg`). The `OmmConsumer` class provides configurable admin domain message processing (i.e., login, directory, and dictionary requests).

4.1.1 Connecting to a Server and Opening Items

Applications observe the following steps to connect to a server and open items:

- **(Optional)** Specify a configuration using the **EmaConfig.xml** file.

  This step is optional because the EMA provides a default configuration which is usually sufficient in simple application cases.
- Create `OmmConsumerConfig` object (for details, refer to Section 4.3).
- **(Optional)** Change EMA configuration using methods on the `OmmConsumerConfig` class.

  If an **EmaConfig.xml** file is not used, then at a minimum, applications might need to modify the default host address and port.
- Implement an application callback client class that inherits from the `OmmConsumerClient` class (for details, refer to Section 4.2).

  An application needs to override the default implementation of callback methods and provide its own business logic. Not all methods need to be overridden; only methods required for the application's business logic.
- **(Optional)** Implement an application error client class that inherits from the `OmmConsumerErrorClient` class (for details, refer to Section 7.2).

  The application needs to override default error call back methods to be effectively notified about error conditions.
- Create an `OmmConsumer` object and pass the `OmmConsumerConfig` object (and if needed, also pass in the application error client object), and optionally register for Login events by passing in an application callback client class.
- Open items of interest using the `OmmConsumer::registerClient()` method.
- Process received messages.
- **(Optional)** Submit `PostMsg` and `GenericMsg` messages and modify / close items using appropriate `OmmConsumer` class methods.
- Exit.

4.1.2 Opening Items Immediately After OmmConsumer Object Instantiation

To allow applications to open items immediately after creating the **OmmConsumer** object, the EMA performs the following steps when creating and initializing the **OmmConsumer** object:

- Create an internal item watchlist.
- Establish connectivity to a configured server / host.
- Log into the server and obtain source directory information.
- Obtain dictionaries (if configured to do so).

4.1.3 Destroying the OmmConsumer Object

Destroying an **OmmConsumer** object causes the application to log out and disconnect from the connected server, at which time all items are closed.

4.1.4 Example: Working with the OmmConsumer Class

The following example illustrates the simplest application managing the OmmConsumer Class.

```
try {
   AppClient client;
   OmmConsumer consumer( OmmConsumerConfig().host( "localhost:14002" ).username( "user" ) );
   consumer.registerClient( ReqMsg().serviceName( "DIRECT_FEED" ).name( "IBM.N" ), client );
   sleep( 60000 );
} catch ( const OmmException& excp ) {
   cout << excp << endl;
}
```

4.1.5 Working with Items

The EMA assigns all opened items or instruments a unique numeric identifier (e.g. **UInt64**), called a handle, which is returned by the **OmmConsumer::registerClient()** call. A handle is valid as long as its associated item stays open. Holding onto these handles is important only to applications that want to modify or close particular items, or use the items' streams for sending **PostMsg** or **GenericMsg** messages to the connected server. Applications that just open and watch several items until they exit do not need to store item handles.

While opening an item, on the call to the **OmmConsumer::registerClient()** method, an application can pass an item closure or an application-assigned numeric value. The EMA will maintain the association of the item to its closure as long as the item stays open.

Respective closures and handles are returned to the application in an **OmmConsumerEvent** object on each item callback method.

4.1.6 Example: Working with Items

The following example illustrates using the item handle while modifying an item’s priority and posting modified content.

```
void AppClient::onRefreshMsg( const RefreshMsg& refreshMsg, const OmmConsumerEvent& event )
{
   cout << “Received refresh message for item handle = “ << event.getHandle() << endl;
   cout << refreshMsg << endl;
}

try {
   AppClient client;
   OmmConsumer consumer( OmmConsumerConfig().host( "localhost:14002" ).username( "user" ) );

   Int64 closure = 1;
   UInt64 itemHandle = consumer.registerClient( ReqMsg().serviceName( "DIRECT_FEED" ).name(
            "IBM.N" ), client, (void*)closure );

   consumer.reissue( ReqMsg().serviceName( "DIRECT_FEED" ).name( "IBM.N" ).priority( 2, 2 ),
            itemHandle );

   consumer.submit( PostMsg().payload( FieldList().addInt( 1, 100 ).complete() ), itemHandle
            );

   sleep( 60000 );
} catch ( const OmmException& excp ) {
   cout << excp << endl;
}
```

4.1.7 Working with Tunnel Streams

EMA assigns all tunnel streams a unique numeric identifier (e.g., UInt64), called a parent handle, which is returned by the call: **`OmmConsumer::registerClient(TunnelStreamRequest,…)`**. A parent handle is valid only as long as its associated tunnel stream is open. You can use parent handles to open substreams (as illustrated in Section 4.1.8).

When opening a tunnel stream, on the call to the **`OmmConsumer::registerClient(TunnelStreamRequest,…)`** method, an application can pass a tunnel stream closure or an application-assigned numeric value. The EMA will maintain the association of the tunnel stream to its closure as long as the tunnel stream stays open. Respective closures and parent handles are returned to the application in an **`OmmConsumerEvent`** object on each tunnel stream callback method.

For more details on a **`TunnelStreamRequest`** and how to create it, refer to Section 3.2.6 and Section 3.4.3.

4.1.8 Example: Working with Tunnel Streams

The following example illustrates the use of a parent handle (as returned by **OmmConsumer::registerClient(TunnelStreamRequest,…)**) to open a substream from the **OmmConsumerClient::onStatusMsg()** callback.

```
void onStatusMsg(const StatusMsg& statusMsg, const OmmConsumerEvent& event)
{
    if (event.getHandle() == _tunnelStreamHandle &&
        statusMsg.hasState() &&
        statusMsg.getState().getStreamState() == OmmState::OpenEnum )
    {
        // open substream with parent handle returned when opening tunnel stream below
        _pOmmConsumer->registerClient( ReqMsg().name( "TUNNEL_IBM" ).serviceId( 1 ), *this,
                (void*)1, _tunnelStreamHandle );
    }
}

int main()
{
    try {
        AppClient client;
        OmmConsumer consumer( OmmConsumerConfig().username( "user" ) );
        client.setOmmConsumer( consumer );
        CosAuthentication cosAuthentication;
        cosAuthentication.type( CosAuthentication::OmmLoginEnum );
        CosDataIntegrity cosDataIntegrity;
        cosDataIntegrity.type( CosDataIntegrity::ReliableEnum );
        CosFlowControl cosFlowControl;
        cosFlowControl.type( CosFlowControl::BidirectionalEnum ).recvWindowSize( 1200
                ).sendWindowSize( 1200 );
        ClassOfService cos;
        cos.authentication( cosAuthentication ).dataIntegrity( cosDataIntegrity
                ).flowControl( cosFlowControl );
        TunnelStreamRequest tsr;
        tsr.classOfService( cos ).domainType( MMT_SYSTEM ).name( "TUNNEL" ).serviceName(
                "DIRECT_FEED" );
        /* open tunnel stream and save tunnel stream parent handle to be used for opening
        substreams in onStatusMsg() callback above */
        _tunnelStreamHandle = consumer.registerClient( tsr, client );

        sleep( 60000 ); // API calls onRefreshMsg(), onUpdateMsg(), or onStatusMsg()
    } catch ( const OmmException& excp ) {
        cout << excp << endl;
    }
}
```

4.2 OmmConsumerClient Class

4.2.1 OmmConsumerClient Description

The **OmmConsumerClient** class provides a callback mechanism through which applications receive OMM messages on items for which they subscribe. The **OmmConsumerClient** is a parent class that implements empty, default callback methods. Applications must implement their own class (inheriting from **OmmConsumerClient**), and override the methods they are interested in processing. Applications can implement many specialized client-type classes; each according to their business needs and design. Instances of client-type classes are associated with individual items while applications register item interests.

The **OmmConsumerClient** class provides default implementation for the processing of **RefreshMsg**, **UpdateMsg**, **StatusMsg**, **AckMsg** and **GenericMsg** messages. These messages are processed by their respectively named methods: **onRefreshMsg()**, **onUpdateMsg()**, **onStatusMsg()**, **onAckMsg()**, and **onGenericMsg()**. Applications only need to override methods for messages they want to process.

4.2.2 Example: OmmConsumerClient

The following example illustrates an application client-type class, depicting **onRefreshMsg()** method implementation.

```
class AppClient : public thomsonreuters::ema::access::OmmConsumerClient
{
protected :

   void onRefreshMsg( const thomsonreuters::ema::access::RefreshMsg&, const
            thomsonreuters::ema::access::OmmConsumerEvent& );

   void onUpdateMsg( const thomsonreuters::ema::access::UpdateMsg&, const
            thomsonreuters::ema::access::OmmConsumerEvent& );

   void onStatusMsg( const thomsonreuters::ema::access::StatusMsg&, const
            thomsonreuters::ema::access::OmmConsumerEvent& );
};
void AppClient::onRefreshMsg( const RefreshMsg& refreshMsg, const OmmConsumerEvent& )
{
   if ( refreshMsg.hasMsgKey() )
      cout << endl << "Item Name: " << refreshMsg.getName() << endl << "Service Name: " <<
            refreshMsg.getServiceName();

   cout << endl << "Item State: " << refreshMsg.getState().toString() << endl;

   if ( DataType::NoDataEnum != refreshMsg.getPayload().getDataType() )
      decode( refreshMsg.getPayload().getData() );
}
```

4.3 OmmConsumerConfig Class

4.3.1 OmmConsumerConfig Description

You can use the `OmmConsumerConfig` class to customize the functionality of the `OmmConsumer` class. The default behavior of `OmmConsumer` is hard coded in the `OmmConsumerConfig` class. You can configure `OmmConsumer` in any of the following ways:

- Using the **EmaConfig.xml** file
- Using interface methods on the `OmmConsumerConfig` class
- Passing OMM-formatted configuration data through the `OmmConsumerConfig::config( const Data& )` method.

For more details on using the `OmmConsumerConfig` class and associated configuration parameters, refer to the *EMA Configuration Guide*.

4.3.2 Unencrypted Connections

EMA supports unencrypted connections via a **ChannelType** of **RSSL_SOCKET** (on Linux or Windows), **RSSL_WEBSOCKET**, and **RSSL_HTTP** (on Windows only). You set **ChannelType** inside of a **ChannelGroup**. For detailed information on **ChannelGroup** and its **ChannelType**s, refer to the *EMA C++ Configuration Guide*.

4.3.3 Encrypted Connections

EMA supports encrypted TCP connections for both **Consumer** and **NiProvider** via a `ChannelType` of `RSSL_ENCRYPTED` (i.e., `ChannelType::RSSL_ENCRYPTED`).

4.3.3.1 Implementing Protocols and Encryption Behavior

EMA's implementation of TLS protocol and encryption depends on a number of factors including:

- The operating system you use (which determines the types of protocols EMA can use):
  - On Linux, EMA uses only OpenSSL.
  - On Windows, EMA can use either WinINet or OpenSSL.
- The type of protocol you use (as specified by `EncryptedProtocolType`):
  - WinINet (specified by `EncryptedProtocolType::RSSL_HTTP`), or
  - OpenSSL (specified by `EncryptedProtocolType::RSSL_SOCKET` or `EncryptedProtocolType::RSSL_WEBSOCKET`).

EMA supports both OpenSSl 1.0 and OpenSSL 1.1. By default, EMA first attempts to load OpenSSL1.1, if it cannot, then EMA loads OpenSSL 1.0. For details on the specific libraries loaded by EMA, refer to Section 4.3.3.2.

For OpenSSL connections, you can set the specific TLS encryption protocol you want to use in the `SecurityProtocol` flag (for details on setting `SecurityProtocol` flags, refer to the *EMA C++ Configuration Guide*). Though currently, only TLS 1.2 (0x4) is accepted.

4.3.3.2 OpenSSL Libraries

The libraries that EMA uses to implement OpenSSL encryption depends on the machine's operating system and version of OpenSSL in use:

- On Linux:
  - If using OpenSSL1.1, EMA uses **libssl.so.1.1** and **libcrypto.so.1.1**.
  - If using OpenSSL1.0, EMA uses **libssl.so.10** and **libcrypto.so.10**.
- On Windows:
  - If using OpenSSL1.1, EMA uses **libssl-1_1-x64.dll** and **libcrypto-1_1-x64.dll**.
  - If using OpenSSL1.0, EMA uses **ssleay32.dll** and **libeay32.dll**.

If you want EMA to load a specific version, you can specify **libssl** and **libcrypto** libraries using **`libsslName`** and **`libcryptoName`** (for details on setting these channel parameters, refer to the *EMA C++ Configuration Guide*).

NOTE: The ESDK package does not include OpenSSL libraries. You can obtain compiled OpenSSL libraries from the appropriate OS vendor.

4.3.3.3 Certificate Authority

If you use an OpenSSL Certificate Authority store, you can specify the authority store's location using **`openSSLCAStore`**. For details on this parameter and EMA's default behavior, refer to the parameter's description in the *EMA C++ Configuration Guide*.

4.3.4 HTTP Proxy Connections

EMA supports HTTP proxy tunneling for **`ChannelType::RSSL_SOCKET`**, **`ChannelType::RSSL_HTTP`**, and all **`ChannelType::RSSL_ENCRYPTED`** connection types.

On Windows, WinINet provides legacy HTTP connection type functionality, and you must configure the proxy through the Internet Explorer configuration. You can override WinINet's proxy configuration by using **`tunnelingProxyHostName()`** and **`tunnellingProxyPort()`**.

For **`RSSL_SOCKET`** connection types (standard or encrypted), **libcurl** manages the proxy connection. As with OpenSSL, you can specify a particular **libcurl** library using **`libcurlName`**. By default:

- On Linux, EMA loads **libcurl.so**
- On Windows, EMA loads **libcurl.dll**

For **libcurl** connections, you can provide additional proxy authentication credentials with the following functions:

- **`proxyUserName()`**: set the proxy user name.
- **`proxyPasswd()`**: set the password for proxy authentication.
- **`proxyDomain()`**: set the domain for proxy authentication.

5 Provider Classes

5.1 OmmProvider Class

The `OmmProvider` class is the main provider application interface to the EMA. This class encapsulates transport-level connectivity. It provides all the interfaces a provider-type application needs to submit item messages (i.e., refresh, update, status, generic) as well as handle the login, directory, and dictionary domains (depending upon whether or not the application is an interactive provider). The `OmmProvider` class provides configurable admin domain message processing (i.e., login, directory, and dictionary).

5.1.1 Connecting to ADH and Submitting Items

In the following process, the value for ***`ProviderType`*** is dependent on the type of provider with which you are dealing:

- For non-interactive providers, ***`ProviderType`*** is `NiProvider`.
- For interactive providers, ***`ProviderType`*** is `IProvider`.

▶ **To establish a connection and submit items:**

1. (**Optional**) Specify a configuration using the **EmaConfig.xml** file.

   Specifying a configuration in **EmaConfig.xml** is optional because the EMA provides a default configuration which is usually sufficient in simple application cases.

2. Create the appropriate `Omm`***`ProviderType`***`Config` object (for details, refer to Section 5.4):

   - For a non-interactive provider, create an `OmmNiProviderConfig` object.
   - For an interactive provider, create an `OmmIProviderConfig` object

3. (**Optional**) Change the EMA configuration using methods on the `Omm`***`ProviderType`***`Config` class.

   If **EmaConfig.xml** file is not used, then at a minimum:

   - Non-interactive provider applications might need to modify both the default host address and port.
   - Interactive provider applications might need to modify the default port.

4. (**Conditional**) Implement an application callback client class that inherits from the `OmmProviderClient` class (for details, refer to Section 5.2).

   An application might need to override the default callback implementation and provide its own business logic. Not all methods need to be overridden: only those that require the application's business logic.

   - For non-interactive providers, this step is optional because the application may choose not to open login or dictionary items. In such cases, the provider application will not receive return messages.
   - For interactive providers, this step is required, because at a minimum, the application needs to handle all inbound login domain and item request messages.

5. (**Optional**) Implement an application error client class that inherits from the `OmmProviderErrorClient` class (for details, refer to Section 5.2).

   To be effectively notified about error conditions, the application needs to override any default, error callback methods.

6. Create an `OmmProvider` object and pass the `Omm`***`ProviderType`***`Config` object (and if needed, also pass in the application error client object), and optionally in `NiProvider` only, register for Login events by passing in an application callback client class.

7. (**Optional**) For non-interactive providers, open login and dictionary items using the `OmmProvider::registerClient()` method.

8. Process received messages.

9. Create, populate, and submit item messages (refresh, update, status).

   - For non-interactive providers, the application needs to associate each item with a handle that uniquely identifies the item.
   - For interactive providers, the application needs to use the handle from the `OMMProviderEvent`.

10. (**Optional**) Submit `GenericMsg` messages using the appropriate `OmmProvider` class methods.
11. Exit.

5.1.2 Interactive Providers: Post OmmProvider Object Instantiation

Before an interactive provider can start submitting items, the application must first accept a login request. Though EMA accepts connections, it is the responsibility of the application to send the login response. Subsequently, the consumer will request the source directory, and EMA will respond by submitting the source directory.

After creating an `OmmProvider` object, the EMA observes the following process when creating and initializing the `OmmProvider` object so that applications can begin submitting items:

- Accept the connection request from a consumer
- Accept the login
- Submit the source directory information

5.1.3 Non-Interactive Providers: Post OmmProvider Object Instantiation

After creating an `OmmProvider` object, the EMA performs the following steps when creating and initializing the `OmmProvider` object so that applications can begin submitting items:

- Establish connectivity to a configured server / host
- Log into ADH and submit source directory information

5.1.4 Non-Interactive Providers: Encrypted Connections and HTTP Proxy Tunneling

Non-interactive providers support both encrypted and HTTP proxy tunneling connections. Configuration details are identical to that of the Consumer when setting up these types of connections.

- For details on using an encrypted connection, refer to Section 4.3.3.
- For details on using an HTTP proxy tunneling connection, refer to Section 4.3.4.

5.1.5 Destroying the OmmProvider Object

For non-interactive providers, destroying an `OmmProvider` object causes the application to log out and disconnect from the connected ADH, at which time all items are closed.

For interactive providers, destroying an `OmmProvider` object causes EMA to close all consumer connections.

5.1.6 Non-Interactive Example: Working with the OmmProvider Class

The following example illustrates the simplest application managing the **`OmmProvider`** class.

```
try
    {
        OmmProvider provider( OmmNiProviderConfig().host( "localhost:14003").username
                ( "user" ) );
        UInt64 itemHandle = 5;

        provider.submit( RefreshMsg().serviceName( "NI_PUB" ).name( "IBM.N" )
            .state( OmmState::OpenEnum, OmmState::OkEnum, OmmState::NoneEnum, "UnSolicited
                    Refresh Completed" )
            .payload( FieldList()
                .addReal( 22, 3990, OmmReal::ExponentNeg2Enum )
                .addReal( 25, 3994, OmmReal::ExponentNeg2Enum )
                .addReal( 30, 9, OmmReal::Exponent0Enum )
                .addReal( 31, 19, OmmReal::Exponent0Enum )
                .complete() )
            .complete(), itemHandle );

        sleep( 1000 );

        for ( Int32 i = 0; i < 60; i++ )
        {
            provider.submit( UpdateMsg().serviceName( "NI_PUB" ).name( "IBM.N" )
                .payload( FieldList()
                    .addReal( 22, 3391 + i, OmmReal::ExponentNeg2Enum )
                    .addReal( 30, 10 + i, OmmReal::Exponent0Enum )
                    .complete() ), itemHandle );
            sleep( 1000 );
        }
    }
    catch ( const OmmException& excp )
    {
        cout << excp << endl;
    }
    return 0;
}
```

5.1.7 Interactive Provider Example: Working with the OmmProvider Class

The following example illustrates the simplest interactive application managing the **OmmProvider** class.

```
try
    {
        AppClient appClient;

        OmmProvider provider( OmmIProviderConfig().port( "14002" ), appClient );

        while ( itemHandle == 0 ) sleep(1000);

        for ( Int32 i = 0; i < 60; i++ )
        {
            provider.submit( UpdateMsg().domainType( MMT_MARKET_BY_ORDER ).payload( Map()
                .addKeyAscii( OrderNr, MapEntry::UpdateEnum, FieldList()
                    .addRealFromDouble( 3427, 7.76 + i * 0.1, OmmReal::ExponentNeg2Enum )
                    .addRealFromDouble( 3429, 9600 )
                    .addEnum( 3428, 2 )
                    .addRmtes( 212, EmaBuffer( "Market Maker", 12 ) )
                    .complete() )
                .complete() ), itemHandle );

            sleep( 1000 );
        }
    }
    catch ( const OmmException& excp )
    {
        cout << excp << endl;
    }

    return 0;
```

5.1.8 Interactive Provider Example: Handling Post Message

The following example illustrates how to have **OmmProvider** send an **AckMsg** in response to a **PostMsg**. For more information on support of post messages by a provider, refer to the *Transport API C Edition Developers Guide*.

```
void AppClient::onPostMsg( const PostMsg& postMsg, const OmmProviderEvent& event )
{
    if (postMsg.getSolicitAck())
    {
        AckMsg ackMsg;
        ackMsg.domainType(postMsg.getDomainType());
        ackMsg.ackId(postMsg.getPostId());
        if (postMsg.hasSeqNum())
        {
            ackMsg.seqNum(postMsg.getSeqNum());
        }
        event.getProvider().submit(ackMsg, event.getHandle());
    }
}
```

5.1.9 Interactive Provider Example: Handling RTT Responses from Consumer

The following example implements a provider's callback for Generic messages. The example illustrates how the provider can identify and process consumer responses to RTT requests.

```
void AppClient::onGenericMsg(const GenericMsg& genericMsg, const OmmProviderEvent& event)
{
    if (genericMsg.getDomainType() == MMT_LOGIN && event.getHandle() == loginHandle &&
            genericMsg.getPayload().getDataType() == DataType::ElementListEnum)
    {
        cout << "Received login RTT message from Consumer " << event.getHandle() << endl;
        TimeValue currTicks = GetTime::getTicks();
        const ElementList& elementList = genericMsg.getPayload().getElementList();
        while ( elementList.forth() )
        {
            const ElementEntry& elementEntry = elementList.getEntry();
            if ( elementEntry.getName() == ENAME_RTT_TICKS && elementEntry.getLoadType() ==
                    DataType::UIntEnum ) // "Ticks"
            {
                cout << "\tRTT Tick value is " << elementEntry.getUInt() << "us." << endl;
                lastLatency = (UInt64)(((double)currTicks - (double)elementEntry.getUInt()) /
                        GetTime::ticksPerMicro());
                cout << "\tLast RTT message latency is " << lastLatency << "us." << endl;
            }
            else if ( elementEntry.getName() == ENAME_RTT_TCP_RETRANS && elementEntry.getLoadType() ==
                    DataType::UIntEnum ) // "TcpRetrans"
            {
                cout << "\tConsumer side TCP retransmissions: " << elementEntry.getUInt() << endl;
            }
        }
    }
}
```

5.1.10 Working with Items

The application assigns unique numeric identifiers, called handles (e.g., UInt64) to all open items it is providing. Application must pass this identifier along with an item message on the call to **`submit()`**. The handles are used to manage item stream ids. To reassign a handle to a different item, application must first close the item previously associated with the given handle.

5.2 OmmProviderClient Class

5.2.1 OmmProviderClient Description

The **OmmProviderClient** class provides a callback mechanism through which applications receive OMM messages on items for which they subscribe. The **OmmProviderClient** is a parent class that implements empty, default callback methods. Applications must implement their own class (inheriting from **OmmProviderClient**), and override the methods they are interested in processing. Applications can implement many specialized client-type classes; each according to their business needs and design. Instances of client-type classes are associated with individual items while applications register item interests. The **OmmProviderClient** class provides default implementation for the processing of **RefreshMsg**, **StatusMsg**, and **GenericMsg** messages. These messages are processed by their respectively named methods: **onRefreshMsg()**, **onStatusMsg()**, **onGenericMsg()**, **onRequest()**[1], **onReIssue()**[1], **onClose()**[1], and **onPost()**[1]. Applications only need to override methods for messages they want to process.

5.2.2 Non-Interactive Example: OmmProviderClient

The following example illustrates an application client-type class, depicting **onRefreshMsg()** method implementation.

```
class AppClient : public thomsonreuters::ema::access::OmmProviderClient
{
protected :
    void onRefreshMsg( const thomsonreuters::ema::access::RefreshMsg&, const
            thomsonreuters::ema::access::OmmProviderEvent& );
    void onStatusMsg( const thomsonreuters::ema::access::StatusMsg&, const
            thomsonreuters::ema::access::OmmProviderEvent& );
    bool  _bConnectionUp;
};

void AppClient::onRefreshMsg( const RefreshMsg& refreshMsg, const OmmProviderEvent&
        ommEvent )
{
    cout << endl << "Handle: " << ommEvent.getHandle() << " Closure: " <<
            ommEvent.getClosure() << endl;
    cout << refreshMsg << endl;

    if ( refreshMsg.getState().getStreamState() == OmmState::OpenEnum )
    {
        if ( refreshMsg.getState().getDataState() == OmmState::OkEnum )
                _bConnectionUp = true;
        else
                _bConnectionUp = false;
    }
    else
        _bConnectionUp = false;
}
```

[1] Interactive Provider Only

5.2.3 Interactive Example: OmmProviderClient

The following example illustrates an application client-type class, depicting **`onRefreshMsg()`** method implementation.

```
void AppClient::processLoginRequest( const ReqMsg& reqMsg, const OmmProviderEvent& event )
{
    event.getProvider().submit(RefreshMsg().domainType(MMT_LOGIN).name(reqMsg.getName()).
            nameType(USER_NAME).complete().solicited( true ).
            state( OmmState::OpenEnum, OmmState::OkEnum, OmmState::NoneEnum,
            "Login accepted" ),event.getHandle() );
}

void AppClient::processMarketByOrderRequest( const ReqMsg& reqMsg, const OmmProviderEvent&
        event )
{
    if ( itemHandle != 0 )
    {
        processInvalidItemRequest(reqMsg, event);
        return;
    }

    event.getProvider().submit(RefreshMsg().domainType(MMT_MARKET_BY_ORDER).
            name(reqMsg.getName()).serviceName(reqMsg.getServiceName()).solicited(true)


            .summaryData( FieldList().addEnum( 15, 840 ).addEnum( 53, 1 ).addEnum( 3423, 1 ).
                    addEnum( 1709, 2 ).complete() )
            .addKeyAscii( OrderNr, MapEntry::AddEnum, FieldList()
                .addRealFromDouble( 3427, 7.76, OmmReal::ExponentNeg2Enum )
                .addRealFromDouble( 3429, 9600 )
                .addEnum( 3428, 2 )
                .addRmtes( 212, EmaBuffer( "Market Maker", 12 ) )
                .complete() )
            .complete() )
        .complete(), event.getHandle() );

    itemHandle = event.getHandle();
}

void AppClient::processInvalidItemRequest( const ReqMsg& reqMsg, const OmmProviderEvent&
        event )
{
    event.getProvider().submit( StatusMsg().name( reqMsg.getName() ).serviceName(
            reqMsg.getServiceName() )
        .domainType( reqMsg.getDomainType() )
        .state( OmmState::ClosedEnum, OmmState::SuspectEnum, OmmState::NotFoundEnum,
                "Item not found" ),
        event.getHandle() );
}

void AppClient::onReqMsg( const ReqMsg& reqMsg, const OmmProviderEvent& event )
{
    switch ( reqMsg.getDomainType() )
    {
    case MMT_LOGIN:
        processLoginRequest( reqMsg, event );
        break;
    case MMT_MARKET_BY_ORDER:
        processMarketByOrderRequest( reqMsg, event );
        break;
    default:
        processInvalidItemRequest( reqMsg, event );
        break;
    }
}
```

5.3 OMMIProviderConfig

You can use the **OmmNiProviderConfig** class to customize the functionality of the **OmmProvider** class. The default behavior of **OmmProvider** is hard coded in the **OmmNiProviderConfig** class. You can configure **OmmProvider** in any of the following ways:

- Using the **EmaConfig.xml** file
- Using interface methods on the **OmmNiProviderConfig** class
- Passing OMM-formatted configuration data through the **OmmNiProviderConfig::config(const Data&)** method.

For more details on using the **OmmNiProviderConfig** class and associated configuration parameters, refer to the *EMA Configuration Guide*.

5.4 OmmNiProviderConfig Class

In the following, the value for ***ProviderType*** is dependent on the type of provider with which you are dealing, thus:

- For non-interactive providers, ***ProviderType*** is **NiProvider**.
- For interactive providers, ***ProviderType*** is **IProvider**.

You can use the **Omm*ProviderType*Config** class to customize the functionality of the **OmmProvider** class. The default behavior of **OmmProvider** is hard coded in the **Omm*ProviderType*Config** class. You can configure **OmmProvider** in any of the following ways:

- Using the **EmaConfig.xml** file
- Using interface methods on the **Omm*ProviderType*Config** class
- Passing OMM-formatted configuration data through the **Omm*ProviderType*Config::config(const Data&)** method.

For more details on using the **Omm*ProviderType*Config** class and associated configuration parameters, refer to the *EMA Configuration Guide*.

6 Consuming Data from the Cloud

6.1 Overview

You can use the Elektron Message API to consume data from a cloud-based ADS server. The API interacts with cloud-based servers using the following work flows:

- Authentication Token Management (for details, refer to Section 6.3)
- Service Discovery (for details, refer to Section 6.4)
- Consuming Market Data (for details, refer to Section 6.5)
- Login Reissue (for details, refer to Section 6.3.3)

By default, for cloud connections the Elektron Message API connects to a server in the **us-east** cloud location.

For further details on Elektron as it functions within the cloud, refer to the *Elektron Real Time in Cloud: Installation and Configuration for Client Use*. For details on the parameters you use to configure cloud connections, refer to the *EMA C++ Edition Configuration Guide*.

6.2 Encrypted Connections

When connecting to an ADS in the cloud, you must use a **`ChannelType`** of **`RSSL_ENCRYPTED`** (for details on **`ChannelType`**, refer to the *EMA C++ Configuration Guide*).

Encrypted connections to the cloud must use an OpenSSL-based connection type (on both Windows and Linux). WinINet is not supported for cloud connectivity.

6.3 Authentication Token Management

6.3.1 Client_ID (AppKey) and Client Secret

To connect to Elektron infrastructure in the cloud (i.e., for ERT in the Cloud), the Elektron Message API requires a **Client_ID**, and optionally can include a client secret. **Client_IDs** 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_IDs**.

- For further details on generating this ID, refer to the *Elektron Real Time in Cloud: Installation and Configuration for Client Use* document. Each **Client_ID** is unique: do not share it with others.
- For further details on supporting client secret submissions, refer to the *EMA C++ Edition Reference Manual*.
- 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/.

6.3.2 Obtaining Initial Access and Refresh Tokens

To obtain an access token, the ESDK API sends its username, **Client_ID**, and password in a single message to the EDP Gateway.

TIP: You can also specify `tokenScope` and `clientSecret` in the OMMConsumerConfig.

In response, the EDP 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 6.3.3). 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 ESDK API obtains its tokens:

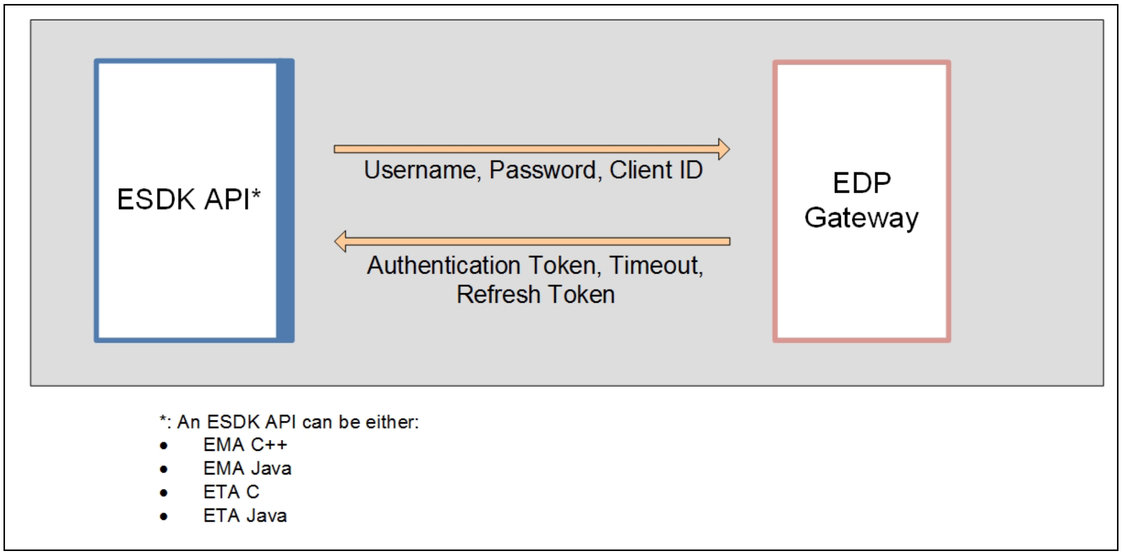

ESDK API*
Username, Password, Client ID
Authentication Token, Timeout, Refresh Token
EDP Gateway
*: An ESDK API can be either:
EMA C++
EMA Java
ETA C
ETA Java

Figure 2. Obtaining an Authentication Token

6.3.3 Refreshing the Access Token and Sending a Login Reissue

In response to the API's token request, the EDP 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 ADS does not receive a new access token before the end of the expiration timeout, the ADS sends a login close status message and closes the connection.

NOTE: The life cycle of `OmmConsumer` in the Elektron Message API depends on the state of the login stream because the Elektron Message API closes the underlying channel whenever the API receives a close status message from ADS. To recover from this scenario, the application must create another `OmmConsumer` and resubscribe to all applicable items.

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 Elektron Message 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**).

In response to receiving a refresh token, the EDP Gateway sends a new access token with an associated timeout to the API. After receiving the new access token from the EDP Gateway, the API renews its connection by sending a Login Reissue with the new access token to the ADS. The process of renewing the access token and refreshing the ADS 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 6.3.2.

The login reissue process is illustrated in the following diagram:

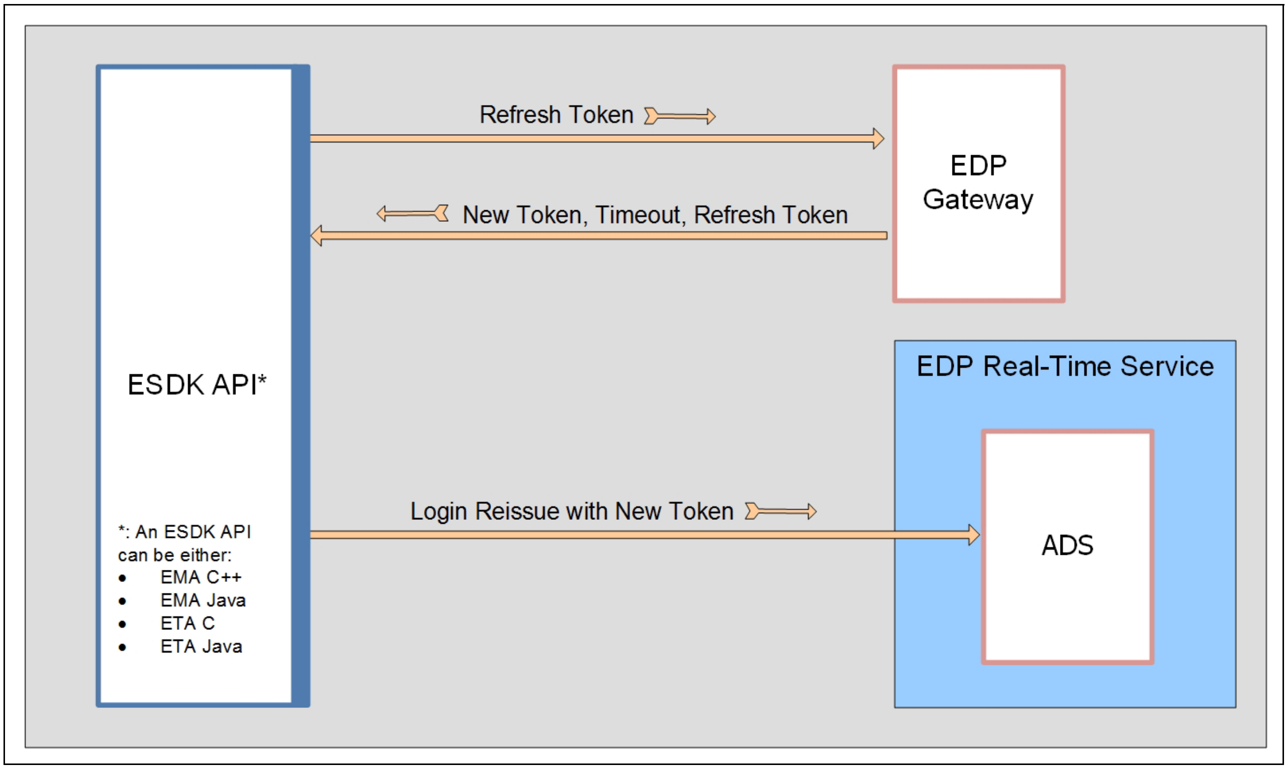


Figure 3. Login Reissue

6.3.4 Session Management per User Credential

Prior to Version 3.3.1, the Elektron Message 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.

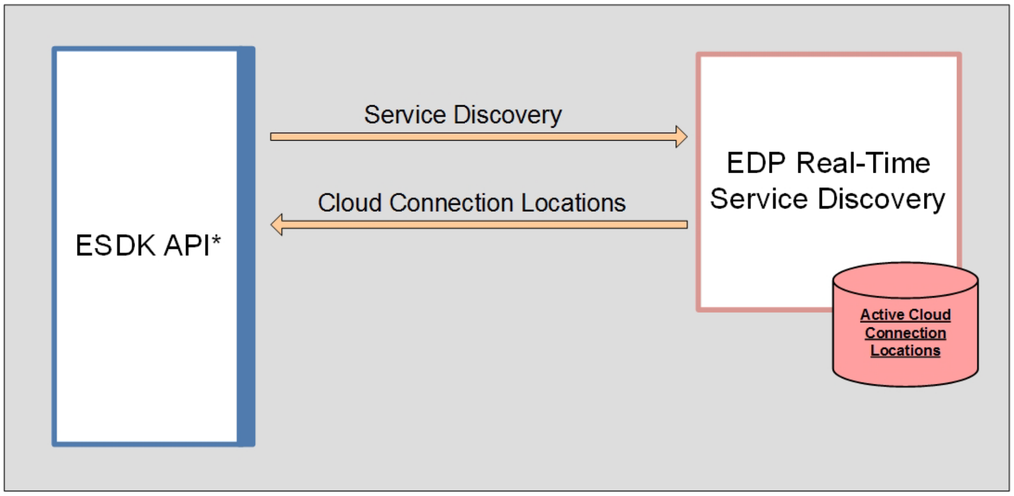
Now, in 3.3.1, the Elektron Message API connects to the EDP Gateway once and reuses the same access and refresh tokens for all channels. The Elektron Message API supports up to, but no more than, 5 channels per OAuth credential set.

6.4 Service Discovery

After obtaining a token (for details, refer to Section 6.3.2), the Elektron Message API can perform a service discovery against the EDP Gateway (whose URL is set in `OmmConsumerConfig`) to obtain connection details for the ADS in the cloud.

In response to a service discovery, the EDP returns transport and data format protocols and a list of hosts and associated ports for the requested service(s) (i.e., an ADS running in the cloud). Refinitiv provides multiple cloud locations based on region, which is significant in how an Elektron Message API chooses the IP address and port to use when connecting to the cloud.

From the list sent by the EDP Gateway, the Elektron Message API identifies an ADS (i.e., an endpoint) set up for failover and whose regional location matches the API's location setting in `ChannelGroup` (for details, refer to Section 3.3.2). If you do not specify a location, the Elektron Message API defaults to the **us-east** cloud location. An endpoint setup for failover lists multiple locations in its location field (e.g., `location: [us-east-1a, us-east-1b]`). If multiple endpoints are set up for failover, the Elektron Message API chooses to connect to the first endpoint listed.


ESDK API*
Service Discovery
Cloud Connection Locations
EDP Real-Time Service Discovery
Active Cloud Connection Locations

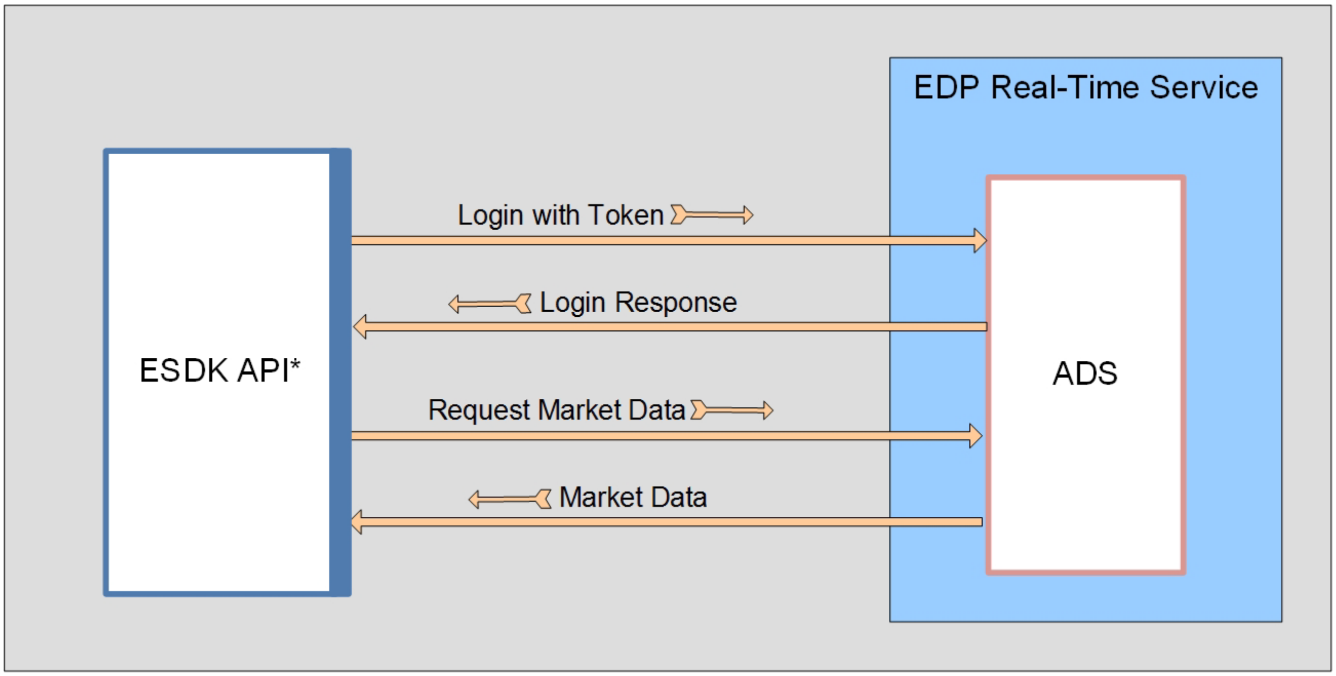
*: An ESDK API can be either:

- EMA C++
- EMA Java
- ETA C
- ETA Java

Figure 4. Service Discovery

6.5 Consuming Market Data

After obtaining its login token (for details, refer to Section 6.3.2) and running a service discovery (for details, refer to Section 6.4), the API can connect to the ADS 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 6.3.3).



*: An ESDK API can be either:

- EMA C++
- EMA Java
- ETA C
- ETA Java

6.6 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 6.6.1)
- Query service discovery (for details, refer to Section 6.6.2)

6.6.1 Session Management Use Case

In the session management use case, the Elektron Message API manages the entire connection from start to finish. To use session management, you need to configure the API to enable session management (i.e., in the **ChannelGroup**, set the **Channel** entry parameter **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 6.3.2)
- Queries service discovery (according to the details in Section 6.4)
- Consumes market data (according to the details in Section 6.5)
- Manages login reissues when needed on a cyclical basis (according to the details in Section 6.3.3)

EMA's **Consumer** example (*113__MarketPrice__SessionManagement example*) provides sample source to illustrate session management.

A special use case exists for connecting to a specific (i.e., non-default) host. As described in Section 6.4, by default the Elektron Message 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 **ChannelGroup** parameters: **Host** and **Port**. In this case, the Elektron Message API exhibits the same behavior listed above, but ignores the endpoints it receives from the service discovery.

6.6.2 Query Service Discovery

In the query service discovery use case, the API user wants to connect to the EDP Gateway 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 6.3.2)
- Queries service discovery (according to the details in Section 6.4)

EMA's **Consumer** example (*450__MarketPrice__QueryServiceDiscovery*) provides sample source that discovers an endpoint using the service discovery feature and establishes an encrypted connection to consume data.

7 Troubleshooting and Debugging

7.1 EMA Logger Usage

The EMA provides a logging mechanism useful for debugging runtime issues. In the default configuration, EMA is set to log significant events encountered during runtime and direct logging output to a file. If needed, you can turn off logging, or direct its output to **`stdout`**. Additionally, applications can configure the logging level at which the EMA logs event (to log every event, only error events, or nothing). For further details on managing and configuring the EMS logging function, refer to the *EMA Configuration Guide*.

7.2 Omm Error Client Classes

7.2.1 Error Client Description

EMA has two Error Client classes: **`OmmConsumerErrorClient`** and **`OmmProviderErrorClient`**. These two classes are an alternate error notification mechanism in the EMA, which you can use instead of the default error notification mechanism (i.e., **`OmmException`**, for details, refer to Section 7.3). Both mechanisms deliver the same information and detect the same error conditions. To use Error Client, applications need to implement their own error client class, override the default implementation of each method, and pass this Error Client class on the constructor to **`OmmConsumer`** and **`OmmProvider`**.

7.2.2 Example: Error Client

The following example illustrates an application error client and depicts simple processing of the **`onInvalidHandle()`** method. In the following example, ***`ClassName`*** is either **`OmmConsumerErrorClient`** (for EMA consumer applications) or **`OmmProviderErrorClient`** (for EMA provider applications).

```
class AppErrorClient : public ClassName
{
public :

   void onInvalidHandle( UInt64 handle, const EmaString& text );

   void onInaccessibleLogFile( const EmaString& filename, const EmaString& text );

   void onMemoryExhaustion( const EmaString& text);

   void onInvalidUsage( const EmaString& text, Int32 errorCode );

   void onSystemError( Int64 code, void* ptr, const EmaString& text );

   void onJsonConverter( const EmaString& text, Int32 errorCode, const ConsumerSessionInfo&
            sessionInfo );
};

void AppErrorclient::onInvalidHandle( UInt64 handle, const EmaString& text )
{
   cout << "Handle = " << handle << endl << ", text = " << text <<endl;
}
```

7.3 OmmException Class

If the EMA detects an error condition, the EMA might throw an exception. All exceptions in the EMA inherit from the parent class **`OmmException`**, which provides functionality and methods common across all **`OmmException`** types.

TIP: Refinitiv recommends you use **`try`** and **`catch`** blocks during application development and QA to quickly detect and fix any EMA usage or application design errors.

The EMA supports the following exception types:

- **`OmmInaccessibleLogFileException`**: Thrown when the EMA cannot open a log file for writing.
- **`OmmInvalidConfigurationException`**: Thrown when the EMA detects an unrecoverable configuration error.
- **`OmmInvalidHandleException`**: Thrown when an invalid / unrecognized item handle is passed in on **`OmmConsumer`** or **`OmmProvider`** class methods.
- **`OmmInvalidUsageException`**: Thrown when the EMA detects invalid interface usage.
- **`OmmJsonConverterException`**: Thrown when the EMA fails to perform a RWF/JSON conversion.
- **`OmmMemoryExhaustionException`**: Thrown when the EMA detects an out-of-memory condition.
- **`OmmOutOfRangeException`**: Thrown when a passed-in parameter lies outside the valid range.
- **`OmmSystemException`**: Thrown when the EMA detects a system exception.
- **`OmmUnsupportedDomainTypeException`**: Thrown if domain type specified on a message is not supported.

Document ID: EMAC350UM.201
Date of issue: 30 June 2020

REFINITIV™