Enterprise Message API C# Edition 3.2.0.L1

ENTERPRISE MESSAGE API DEVELOPERS GUIDE

REFINITIV™

Contents

1   Introduction

1.1   About this Manual

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

This guide documents the functionality and capabilities of the Enterprise Message API C# Edition. The Enterprise Message API can also connect to and leverage many different Refinitiv and customer components. If you want the Enterprise Message 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 document provides detailed yet supplemental information for application developers writing to the Enterprise Message API.

1.3   Programming Languages

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

1.4   Acronyms and Abbreviations

ACRONYM / TERM	MEANING
ADH	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
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.
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol (Secure)

Table 1: Acronyms and Abbreviations

ACRONYM / TERM	MEANING
JWK	JSON Web Key. Defined by RFC 7517, a JWK is a JSON formatted public or private key.
JWKS	JSON Web Key Set, This is a set of JWK, placed in a JSON array.
JWT	JSON Web Token. Defined by RFC 7519, JWT allows users to create a signed claim token that can be used to validate a user.
OMM	Open Message Model
QoS	Quality of Service
RDM	Refinitiv Domain Model
RDP	Refinitiv Data Platform: this platform is used for REST interactions. In the context of Real-Time APIs, an API gets authentication tokens and/or queries Service Discovery to get a list of Refinitiv Real-Time — Optimized endpoints using RDP.
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.
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.

Table 1: Acronyms and Abbreviations

1.5   References

- Enterprise Message API C# Edition *Refinitiv Domain Model Usage Guide*

- *API Concepts Guide*

- *Enterprise Message API C# Edition Configuration Guide*

- Enterprise Message API C# Edition *Developers Guide*

- 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

This document uses the following types of conventions:

- C# classes, 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:

```
consumer = new(new OmmConsumerConfig().OperationModel(OperationModelMode.USER_DISPATCH)
                                        .Host("localhost:14002").UserName("user"));
        consumer.RegisterClient(new RequestMsg().ServiceName("DIRECT_FEED").Name("IBM.N"),
                new AppClient(), 0);
        var endTime = System.DateTime.Now + TimeSpan.FromMilliseconds(60000);
        while(System.DateTime.Now < endTime)
        {
            consumer.Dispatch(10);      // calls to OnRefreshMsg(), OnUpdateMsg(), or
                OnStatusMsg() execute on this thread
        }
```

2   Product Overview

2.1   Enterprise Message API Product Description

The Enterprise Message API is a data-neutral, multi-threaded, ease-of-use API providing access to OMM and RWF data. As part of the Refinitiv Real-Time Software Development Kit, or RTSDK, the Enterprise Message API 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 Enterprise Transport API.

The Enterprise Message API:

- 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. Refinitiv Real-Time, Refinitiv Real-Time Distribution System, Refinitiv Real-Time Advanced Transformation Server, Refinitiv Data Feed Direct, etc).

- Leaves a minimal code footprint in applications written to it. The design of the Enterprise Message API and its interfaces allows application development to focus more on the application business logic than on the usage of the Enterprise Message API.

- Includes training applications that provide basic, yet still functional, examples of Enterprise Message API applications.

- Presents applications with simplified access to OMM messages and containers while providing all necessary transport level functionalities. Generally, Enterprise Message API 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 `accessor` functionality to populate and read OMM containers and messages. Enterprise Message API 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 Enterprise Message API in a few classes whose functionality is implied by their class name.

2.2   Product Documentation and Learning Enterprise Message API

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

NOTE: Enterprise Message API 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 Enterprise Message API 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., int, String, and Buffer).

- The 300- and 400- series examples depict usage of particular Enterprise Message API 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 Enterprise Message API 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 Enterprise Message API features such as user control of the source directory domain, login streaming, connection recovery, programmatic configuration, etc.

2.3   Product Architecture

2.3.1   Enterprise Message API Consumer Architecture

The Enterprise Message API incorporates the ValueAdded Reactor component (called the Transport API VA Reactor) from the Transport API, which provides the watchlist and transport-level functionality. The Enterprise Message API 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 `IOmmConsumerClient` class provides the callback mechanism for Enterprise Message API to send incoming messages to the application. The application needs to implement a class inheriting from the `IOmmConsumerClient` class to receive and process messages. By default, `IOmmConsumerClient` callback methods are executed in Enterprise Message API'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 `IOmmConsumerErrorClient` class provides an alternate reporting mechanism via callbacks. To use the alternate error reporting, pass the `IOmmConsumerErrorClient` on the constructor of the `OmmConsumer` class, which switches the error reporting from exception throwing to callbacks. In addition to its error reporting mechanisms, Enterprise Message API provides a logger mechanism which is useful in monitoring Enterprise Message API behavior and debugging any issues that might arise.

The Enterprise Message API 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 Enterprise Message API to create a second, internal thread to dispatch received messages. To create a second thread, set the `OmmConsumerConfig` operation model to `OmmConsumerConfig.OperationModelMode.API_DISPATCH`. If the `OmmConsumerConfig` operation model is set to the `OmmConsumerConfig.OperationModelMode.USER_DISPATCH`, the Enterprise Message API 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.

2.3.2   Enterprise Message API Provider Architecture

The Enterprise Message API provider incorporates the Value Added (VA) Reactor component from the EnterpriseTransport API, which provides transport-level functionality. The Enterprise Message API 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 Enterprise Message API'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`.

Enterprise Message API sends incoming messages to the application using the `IOmmProviderClient` callback mechanism. To receive and process messages, the application needs to implement a class that inherits from the `IOmmProviderClient` class. By default, `IOmmProviderClient` callback methods are executed in Enterprise Message API's thread of control. However, you can use either the `OmmNiProviderConfig.OperationModel()` or `OmmIProviderConfig.OoperationModel()` 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 `IOmmProviderErrorClient` class provides an alternate reporting mechanism via callbacks. To use the alternate error reporting, pass the `IOmmProviderErrorClient` on the constructor of the `OmmProvider` class, which switches the error reporting from exception throwing to callbacks. In addition to its error-reporting mechanisms, Enterprise Message API provides a logger mechanism which you can use to monitor Enterprise Message API behavior and debug any issues that arise.

An Enterprise Message API 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 Enterprise Message API to create a second internal thread over which to dispatch received messages:

- For non-interactive providers, set the **OmmNiProviderConfig** operation model to **OmmNiProviderConfig.OperationModel.API_DISPATCH**. If the operation model is set to **OmmNiProviderConfig.OperationModel.USER_DISPATCH**, Enterprise Message API will not run a second thread.

- For interactive providers, set the **OmmIProviderConfig** operation model to **OmmIProviderConfig.OperationModel.API_DISPATCH**. If the operation model is set to **OmmIProviderConfig.OperationModel.USER_DISPATCH**, Enterprise Message API 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 Enterprise Message API 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 Enterprise Message API as a non-interactive provider: the application must set **OmmNiProviderConfig.AdminControl.USER_CONTROL** through the **OmmNiProviderConfig.AdminControlDirectory()** method.

- When running Enterprise Message API as an interactive provider: the application must set **OmmIProviderConfig.AdminControl.USER_CONTROL** through the **OmmIProviderConfig.AdminControlDirectory()** method. Additionally, you can configure the ability to disable internal dictionary responses by setting **OmmIProviderConfig.AdminControl.USER_CONTROL** through the **OmmIProviderConfig.AdminControlDictionary()** method.

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

An Enterprise Message API 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 Enterprise Message API as a non-interactive provider: the application must set **OmmNiProviderConfig.AdminControl.API_CONTROL** through the **OmmNiProviderConfig.AdminControlDirectory()** method. An Enterprise Message API non-interactive provider does not support programmatically configuring dictionary information.

- When running Enterprise Message API as an interactive provider: the application must set **OmmIProviderConfig.AdminControl.API_CONTROL** 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.AdminControl.API_CONTROL** through the **OmmIProviderConfig.AdminControlDictionary()** method.

2.3.3   Enterprise Message API Codec Architecture

The Enterprise Message API Codec uses the Enterprise Transport API decoding and encoding functions to read and populate OMM containers and messages. Each OMM container and message is represented by a respective Enterprise Message API 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()**).

3   OMM Containers and Messages

3.1   Overview

Enterprise Message API supports a full set of OMM containers, messages, and primitives (e.g. `FieldList`, `Map`, `RefreshMsg`, `int`). For simplicity, Enterprise Message API 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).
- Iterators are used to iterate over every OMM container.

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 Enterprise Message API 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 `Load` interface to return a reference to the base `Data` class. The `Load` 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, Enterprise Message API 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.DataTypes.FIELD_LIST**, **DataType.DataTypes.REFRESH_MSG**). You can use the **Data.DataType** property 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.DataTypes.ERROR**, which indicates an error condition was detected. For more details, refer to Section 3.2.5.
*   **DataType.DataTypes.NO_DATA**, which signifies a lack of data on the summary of a container, message payload, or attribute.

3.2.2   DataCode ENUM

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

*   The **DataCode.NO_CODE** indicates that the received data is valid and application may use it.
*   The **DataCode.BLANK** 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.

> **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.

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 Enterprise Message API 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 **AsHex()** 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(ElementList elementList)
    {
        foreach (ElementEntry elementEntry in elementList)
        {
            if (Data.DataCode.BLANK == elementEntry.Code)
                Console.WriteLine(" blank");
            else
```

```
        switch (elementEntry.LoadType)
        {
        case DataTypes.REAL:
            Console.WriteLine(elementEntry.OmmRealValue().AsDouble());
            break;
        case DataTypes.ERROR:
            Console.WriteLine(elementEntry.OmmErrorValue().ErrorCode + " (" +
elementEntry.OmmErrorValue().ErrorCodeAsString() + ")");
            break;
        default:
            Console.WriteLine();
            break;
        }
    }
}
```

3.3　　　Working with OMM Containers

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

Each of these classes implements the IEnumerable interface and 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 = new FieldList();

        fieldList.Info(1, 1);
        fieldList.AddUInt(1, 64);
        fieldList.AddReal(6, 11, OmmReal.MagnitudeTypes.EXPONENT_NEG_2);
        fieldList.AddDate(16, 2023, 10, 23);
        fieldList.AddTime(19, 02, 03, 04, 005);
        fieldList.Complete();
    }
    catch (OmmException excp)
    {
        Console.WriteLine(excp);
    }
```

3.3.2 Example: Extracting Information from a `FieldList` Class

In the following example illustrates how to extract information from the **FieldList** class by iterating over the class. The following code extracts information about all entries.

```
void Decode(FieldList fieldList)
    {
        if(fieldList.HasInfo)
          {
            int dictionaryId = fieldList.InfoDictionaryId();
            int fieldListNum = fieldList.InfoFieldListNum();
          }

        foreach (FieldEntry fieldEntry in fieldList)
        {
            if (Data.DataCode.BLANK == fieldEntry.Code)
                Console.WriteLine(" blank");
            else
                switch (fieldEntry.LoadType)
                {
                case DataTypes.ASCII :
                    Console.WriteLine(fieldEntry.OmmAsciiValue());
                    break;
                case DataTypes.ERROR :
                    Console.WriteLine("(" + fieldEntry.OmmErrorValue().ErrorCodeAsString() + ")");
```

```
            break;
        }
    }
}
```

3.3.3   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.

```csharp
void DecodeFieldList(FieldList fieldList)
{
    if (fieldList.HasInfo)
        Console.WriteLine("FieldListNum: " + fieldList.InfoFieldListNum() + " DictionaryId: " +
    fieldList.InfoDictionaryId());

    foreach (FieldEntry fieldEntry in fieldList)
    {
        Console.WriteLine("Load");
        Decode(fieldEntry.Load!);
    }
}

void Decode(Data data)
{
    if (Data.DataCode.BLANK == data.Code)
        Console.WriteLine("Blank data");
    else
        switch (data.DataType)
        {
            case DataType.DataTypes.REFRESH_MSG:
                DecodeRefreshMsg((RefreshMsg)data);
                break;

            case DataType.DataTypes.UPDATE_MSG:
                DecodeUpdateMsg((UpdateMsg)data);
                break;

            case DataType.DataTypes.STATUS_MSG:
                    DecodeStatusMsg((StatusMsg)data);
                break;

            case DataType.DataTypes.FIELD_LIST:
                    DecodeFieldList((FieldList)data);
                break;

            case DataType.DataTypes.MAP:
```

```
                DecodeMap((Map)data);
                break;

        case DataType.DataTypes.NO_DATA:
                Console.WriteLine("NoData");
                break;

        case DataType.DataTypes.TIME:
                Console.WriteLine("OmmTime: " + ((OmmTime)data).ToString());
                break;

        case DataType.DataTypes.DATE:
                Console.WriteLine("OmmDate: " + ((OmmDate)data).ToString());
                break;

        case DataType.DataTypes.REAL:
                Console.WriteLine("OmmReal::asDouble: " + ((OmmReal)data).AsDouble());
                break;

        case DataType.DataTypes.INT:
                Console.WriteLine("OmmInt: " + ((OmmInt)data).Value);
                break;

        case DataType.DataTypes.UINT:
                Console.WriteLine("OmmUInt: " + ((OmmUInt)data).Value);
                break;

        case DataType.DataTypes.ENUM:
                Console.WriteLine("OmmEnum: " + ((OmmEnum)data).Value);
                break;

        case DataType.DataTypes.ASCII:
                Console.WriteLine("OmmAscii: " + ((OmmAscii)data).Value);
                break;

        case DataType.DataTypes.RMTES:
                Console.WriteLine("OmmRmtes: " + ((OmmRmtes)data).Value);
                break;

        case DataType.DataTypes.ERROR:
                Console.WriteLine("Decoding error: " + ((OmmError)data).ErrorCodeAsString());
                break;

        default:
                break;
    }
}
```

3.4   Working with OMM Messages

Enterprise Message API 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**.

```
GenericMsg genMsg = new GenericMsg();

        FieldList nestedFieldList = new FieldList();
        nestedFieldList.AddReal(22, 34, OmmReal.MagnitudeTypes.EXPONENT_POS_1);

        genMsg.DomainType(200).Name("TRI.N").ServiceId(234).Payload(nestedFieldList);
```

3.4.2   Example: Extracting Information from the `GenericMsg` class

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

```
void Decode(GenericMsg genMsg)
   {
      if (genMsg.HasServiceId)
         Console.WriteLine("ServiceId: " + genMsg.ServiceId());

      if (genMsg.HasPartNum)
         Console.WriteLine("PartNum:   " + genMsg.PartNum());

      if (genMsg.HasSeqNum)
         Console.WriteLine("SeqNum:    " + genMsg.SeqNum());

      switch (genMsg.Payload().DataType)
      {
      case DataTypes.FIELD_LIST:
         Decode(genMsg.Payload().FieldList());
         break;
      }
   }
```

4   Consumer Classes

4.1   OmmConsumer Class

The `OmmConsumer` class is the main consumer application interface to the Enterprise Message API. 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 Enterprise Message API provides a default configuration which is usually sufficient in simple application cases.

- Create `OmmConsumerConfig` object (for details, refer to Section 4.3).

- **(Optional)** Change Enterprise Message API 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 `IOmmConsumerClient` 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.

- **(Optional)** Implement an application error client class that inherits from the `IOmmConsumerErrorClient` 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 by calling `OmmConsumer.Uninitialize()`.

4.1.2   Opening Items Immediately After OmmConsumer Object Instantiation

To allow applications to open items immediately after creating the `OmmConsumer` object, the Enterprise Message API 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

Calling `Uninitialize()` on 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.

```
OmmConsumer? consumer = null;
      try
      {
          AppClient appClient = new();
       OmmConsumerConfig config = new OmmConsumerConfig().Host("localhost:14002").UserName("user");
          consumer = new OmmConsumer(config);
          consumer.RegisterClient(new RequestMsg().ServiceName("DIRECT_FEED").Name("IBM.N"),
  appClient);
          Thread.Sleep(60000); // API calls OnRefreshMsg(), OnUpdateMsg() and OnStatusMsg()
      }
      catch (OmmException excp)
      {
          Console.WriteLine(excp.Message);
      }
      finally
      {
          consumer?.Uninitialize();
      }
```

4.1.5   Working with Items

The Enterprise Message API assigns all opened items or instruments a unique numeric identifier (e.g. **long**), 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 Enterprise Message API 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.

```csharp
public void OnRefreshMsg(RefreshMsg refreshMsg, IOmmConsumerEvent consumerEvent)
    {
        Console.WriteLine(refreshMsg);
    }

public static void Main()
    {
        OmmConsumer? consumer = null;
        try
        {
            consumer = new(new OmmConsumerConfig().Host("localhost:14002").UserName("user"));
            consumer.RegisterClient(new RequestMsg().ServiceName("DIRECT_FEED").Name("IBM.N")
                .Qos(RequestMsg.Timeliness.REALTIME, RequestMsg.Rate.TICK_BY_TICK).Priority(2, 1), new
    AppClient());
            Thread.Sleep(60000);// API calls OnRefreshMsg(), OnUpdateMsg() and OnStatusMsg()
        }
        catch (OmmException ommException)
        {
            Console.WriteLine(ommException.Message);
        }
        finally
        {
            consumer?.Uninitialize();
        }
    }
```

4.2   IOmmConsumerClient Class

4.2.1   IOmmConsumerClient Description

The **IOmmConsumerClient** class provides a callback mechanism through which applications receive OMM messages on items for which they subscribe. The **IOmmConsumerClient** is a parent class that implements empty, default callback methods. Applications must implement their own class (inheriting from **IOmmConsumerClient**), 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 **IOmmConsumerClient** 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()**. The **OnAllMsg()** method processes any of these messages. 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.

```
public void OnRefreshMsg(RefreshMsg refreshMsg, IOmmConsumerEvent @ommConsumerEvent)
    {
        Console.WriteLine("Item Name: " + (refreshMsg.HasName ? refreshMsg.Name() : "<not set>"));
        Console.WriteLine("Service Name: " + (refreshMsg.HasServiceName ? refreshMsg.ServiceName() :
    "<not set>"));

        Console.WriteLine("Item State: " + refreshMsg.State());

        if (refreshMsg.Payload().DataType != DataTypes.NO_DATA)
            Decode(refreshMsg.Payload().Data);

    }
```

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

For more details on using the **OmmConsumerConfig** class and associated configuration parameters, refer to the *Enterprise Message API Configuration Guide*.

4.3.2   Encrypted Connections

The Enterprise Message API supports encrypted TCP connections for **Consumer** via a **ChannelType** of **RSSL_ENCRYPTED** (i.e., **ChannelType::RSSL_ENCRYPTED**).

You configure connection types in the Enterprise Message API configuration file. After configuring the file with the appropriate encrypted connection type, to connect the application via a proxy, call the **ProxyHost()** and **ProxyPort()** functions on the **OmmconsumerConfig** class.

Clients can use the `OmmConsumerConfig` class to call `EncryptionProtocol()` to set a cryptographic protocol. Enterprise Message API uses the default value of **TLS** (the operating system chooses the best protocol to use and blocks protocols that are not secure).

An encrypted connection might use proxy devices as it tunnels through the Internet. Client can configure some proxy servers to authenticate client applications before they pass through the proxy. The Enterprise Message API supports Basic authentication schemes. You also use the `OmmConsumerConfig` class to set credential parameters such as **ProxyUsername()**, **ProxyPassword()** for Proxy **Authentication**.

The Enterprise Message API package provides an encrypted connection training example that showcases the usage of the Enterprise Message API's `OmmConsumerConfig` interface configured for encrypted connection (i.e., `ChannelType::RSSL_ENCRYPTED`).

5 Provider Classes

5.1 OmmProvider Class

The **OmmProvider** class is the main provider application interface to the Enterprise Message API. 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 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 Enterprise Message API 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 5.3):

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

3. (**Optional**) Change the Enterprise Message API 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 **IOmmProviderClient** 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 **IOmmProviderErrorClient** 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   Non-Interactive Providers: Post OmmProvider Object Instantiation

After creating an `OmmProvider` object, the Enterprise Message API 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

5.1.3   Interactive Providers: Post OmmProvider Object Instantiation

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

After creating an `OmmProvider` object, the Enterprise Message API 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

5.1.4   Uninitialize the OmmProvider Object

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

For interactive providers, causes Enterprise Message API to close all consumer connections.

5.1.5   Non-Interactive Example: Working with the OmmProvider Class

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

```
try
        {
                OmmNiProviderConfig config = new OmmNiProviderConfig();
                Map map = new Map();
                FieldList summary = new FieldList();
                        FieldList entryLoad = new FieldList();
                long aaoHandle = 5;
                long aggHandle = 6;

                provider = new OmmProvider(config.UserName("user"));
                summary.AddEnumValue(15, 840);
                summary.AddEnumValue(53, 1);
                summary.AddEnumValue(3423, 1);
                summary.AddEnumValue(1709, 2);

                map.SummaryData(summary.Complete());

                entryLoad.AddRealFromDouble(3427, 7.76, OmmReal.MagnitudeTypes.EXPONENT_NEG_2);
                entryLoad.AddRealFromDouble(3429, 9600);
                entryLoad.AddEnumValue(3428, 2);
                entryLoad.AddRmtes(212, new EmaBuffer(Encoding.ASCII.GetBytes("Market Maker")));

                map.AddKeyAscii("100", MapAction.ADD, entryLoad.Complete());

                provider.Submit(new RefreshMsg().DomainType(EmaRdm.MMT_MARKET_BY_ORDER)
                        .ServiceName("TEST_NI_PUB").Name("AAO.V")
                   .State(OmmState.StreamStates.OPEN, OmmState.DataStates.OK, OmmState.StatusCodes.NONE,
                        "UnSolicited Refresh Completed")
                   .Payload(map.Complete()).Complete(true),
                   aaoHandle);

                summary.Clear();

                summary.AddEnumValue(15, 840);
                summary.AddEnumValue(53, 1);
                summary.AddEnumValue(3423, 1);
                summary.AddEnumValue(1709, 2);

                map.Clear();

                map.SummaryData(summary.Complete());

                entryLoad.Clear();
```

```
        entryLoad.AddRealFromDouble(3427, 9.92, OmmReal.MagnitudeTypes.EXPONENT_NEG_2);
        entryLoad.AddRealFromDouble(3429, 1200);
        entryLoad.AddEnumValue(3428, 2);
        entryLoad.AddRmtes(212, new EmaBuffer(Encoding.ASCII.GetBytes("Market Maker")));

        map.AddKeyAscii("222", MapAction.ADD, entryLoad.Complete());

        provider.Submit(new RefreshMsg().DomainType(EmaRdm.MMT_MARKET_BY_ORDER)
            .ServiceName("TEST_NI_PUB").Name("AGG.V")
          .State(OmmState.StreamStates.OPEN, OmmState.DataStates.OK, OmmState.StatusCodes.NONE,
            "UnSolicited Refresh Completed")
          .Payload(map.Complete()).Complete(true),
          aggHandle);

        Thread.Sleep(1000);

        for (int i = 0; i < 60; i++)
        {
            entryLoad.Clear();

            entryLoad.AddRealFromDouble(3427, 7.76 + i * 0.1, OmmReal.MagnitudeTypes
                  .EXPONENT_NEG_2);
            entryLoad.AddRealFromDouble(3429, 9600);
            entryLoad.AddEnumValue(3428, 2);
            entryLoad.AddRmtes(212, new EmaBuffer(Encoding.ASCII.GetBytes("Market Maker")));

            map.Clear();

            map.AddKeyAscii("100", MapAction.UPDATE, entryLoad.Complete());

            provider.Submit(new
UpdateMsg().DomainType(EmaRdm.MMT_MARKET_BY_ORDER).ServiceName("TEST_NI_PUB").Name("AAO.V")
                  .Payload(map.Complete()), aaoHandle);

            entryLoad.Clear();

            entryLoad.AddRealFromDouble(3427, 9.92 + i * 0.1,
OmmReal.MagnitudeTypes.EXPONENT_NEG_2);
            entryLoad.AddRealFromDouble(3429, 1200);
            entryLoad.AddEnumValue(3428, 2);
            entryLoad.AddRmtes(212, new EmaBuffer(Encoding.ASCII.GetBytes("Market Maker")));

            map.Clear();

            map.AddKeyAscii("222", MapAction.UPDATE, entryLoad.Complete());

            provider.Submit(new
UpdateMsg().DomainType(EmaRdm.MMT_MARKET_BY_ORDER).ServiceName("TEST_NI_PUB").Name("AGG.V")
```

```
                     .Payload(map.Complete()), aggHandle);

             Thread.Sleep(1000);
        }
    }
    catch (OmmException excp)
    {
         Console.WriteLine(excp.Message);
    }
    finally
    {
        provider?.Uninitialize();
    }
```

5.1.6   Interactive Provider Example: Working with the OmmProvider Class

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

```
OmmProvider? provider = null;
try
{
        AppClient appClient = new AppClient();
        FieldList fieldList = new FieldList();
        Map map = new Map();

        OmmIProviderConfig config = new OmmIProviderConfig();

        provider = new OmmProvider(config.Port("14002"), appClient);

        while (appClient.ItemHandle == 0)
        {
        Thread.Sleep(1000);
        }

        for (int i = 0; i < 60; i++)
        {
            fieldList.AddRealFromDouble(3427, 7.76 + i * 0.1, OmmReal.MagnitudeTypes.EXPONENT_NEG_2);
            fieldList.AddRealFromDouble(3429, 9600);
            fieldList.AddEnumValue(3428, 2);
            fieldList.AddRmtes(212, new EmaBuffer(Encoding.ASCII.GetBytes("Market Maker")));

            map.AddKeyAscii(appClient.OrderNr, Access.MapAction.ADD, fieldList.Complete());

            provider.Submit(new UpdateMsg().DomainType(EmaRdm.MMT_MARKET_BY_ORDER)
                .Payload(map.Complete()), appClient.ItemHandle);

            map.Clear();
            fieldList.Clear();

            Thread.Sleep(1000);
        }

        Thread.Sleep(60_000);
    }
    catch (OmmException excp)
    {
        Console.WriteLine(excp.Message);
    }
    finally
    {
        provider?.Uninitialize();
    }
```

5.1.7   Interactive Provider Example: Handling a Post Message

The following example illustrates the **OmmProvider** sending an **AckMsg** in response to a **PostMsg**. For further details on post message support by the provider, refer to the *Transport API C# Edition Developers Guide*.

```
public void OnPostMsg(PostMsg postMsg, IOmmProviderEvent providerEvent)
{
        Console.WriteLine("Received PostMsg with id: " + postMsg.PostId());
        Console.WriteLine(postMsg);

        if (postMsg.SolicitAck())
        {
            AckMsg ackMsg = new AckMsg();
        if (postMsg.HasSeqNum)
        {
            ackMsg.SeqNum(postMsg.SeqNum());
        }
        if (postMsg.HasName)
        {
            ackMsg.Name(postMsg.Name());
        }
        if (postMsg.HasServiceId)
        {
            ackMsg.ServiceId(postMsg.ServiceId());
        }
        ackMsg.AckId(postMsg.PostId())
            .DomainType(postMsg.DomainType());
        providerEvent.Provider.Submit(ackMsg, providerEvent.Handle);
    }
}
```

5.1.8   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. For further information on RTT messages and RTT monitoring, refer to the *Transport API C# Edition RDM Usage Guide*.

```
void ProcessRTT(GenericMsg genericMsg, IOmmProviderEvent providerEvent)
{
    if (clientLatencyMap.ContainsKey(providerEvent.Handle))
    {
        if (genericMsg.Payload().DataType == DataType.DataTypes.ELEMENT_LIST
            && genericMsg.DomainType() == EmaRdm.MMT_LOGIN)
        {
            Console.WriteLine("Received login RTT message from Consumer " + providerEvent.Handle);
            ElementList data = genericMsg.Payload().ElementList();

            foreach (var elem in data)
            {
```

```
                      if (elem.Name.Equals(EmaRdm.ENAME_TICKS))
                      {
                          Console.WriteLine("        RTT Tick value is: " + elem.UIntValue());
                          ulong latency = GetNanoseconds() - elem.UIntValue();
                          clientLatencyMap[providerEvent.Handle] = latency;
                          Console.WriteLine("        Last RTT message latency is: " + latency);
                      }
                  }
              }
          }
}
```

5.1.9   Working with Items

The application assigns unique numeric identifiers, called handles (e.g., long) 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.

5.1.10   Packing with Providers

Provider applications can use the **PackedMsg** object to send multiple messages packed together in a single packet. Applications can designate the bounds of the **PackedMsg** by setting its limit for messages packed, the byte limit of data it can send, and then pack messages before sending them together.

The following sections provide packing examples for an Interactive Provider and a Non-interactive Provider.

5.1.10.1   Interactive Example of Packing

The following example illustrates an Interactive Provider application setting up a basic **PackedMsg** object and packing messages together before submitting **PackedMsg**.

```
public void SendPackedMessagesExample(OmmProvider provider, long clientHandle, long itemHandle)
    {
        FieldList fieldList = new FieldList(); // Field list used for message payload

        PackedMsg packedMsg = new PackedMsg(provider);
        packedMsg.InitBuffer(clientHandle); // Initialize buffer using client handle and default
            size of 6000. See reference manual for other uses of InitBuffer().
        UpdateMsg updateMsg = new UpdateMsg();

        for (int i = 0; i < 10; i++) // Send 10 packed messages every second
        {
            for (int j = 0; j < 10; j++) // Pack 10 messages
            {
                fieldList.Clear();
                fieldList.AddReal(22, 3991 + j, OmmReal.MagnitudeTypes.EXPONENT_NEG_2);
                fieldList.AddReal(30, 10 + j, OmmReal.MagnitudeTypes.EXPONENT_0);

                updateMsg.Clear();
                updateMsg.ServiceName("NI_PUB").Name("IBM.N").Payload(fieldList.Complete());
```

```
            try
            {
                packedMsg.AddMsg( msg, itemHandle ); // Add message with its item handle
            }
            catch ( OmmInvalidUsageException excp )
            {
                //The API was unable to add the current message into the packed buffer.
                //If messages have been successfully added to the packed buffer, submit them, get a
                //new packed buffer, and add the current message into that new buffer.
                //and add the current message into that new buffer.
                if ( excp.errorCode() ==  OmmInvalidUsageException.ErrorCode.BUFFER_TOO_SMALL)
                {
                    if (packedMsg.packedMsgCount() > 0) // Packed message has some data
                    {
                        // Submit the messages we've already packed, get a new packed buffer, and
add the current message.
                        provider.submit(packedMsg);// Submit packed message on OmmProvider
                        packedMsg.initBuffer(clientHandle); // Re-initialize buffer for next set of
packed messages.
                        packedMsg.addMsg(msg, itemHandle); // Add missed message with its item
handle
                    }
                    else
                    {
                        //Packed buffer too small to add even first message.
                        //Consider initializing the buffer to a higher value than the default 6000
bytes if needed.
                        //See initBuffer() methods for more details
                    }
                }
                else
                {
                    // Handle other exceptions from addMsg() here
                }
            }

        if (packedMsg.packedMsgCount() > 0)
        {
            provider.submit(packedMsg);// Submit packed message on OmmProvider
            packedMsg.initBuffer(clientHandle); // Re-initialize buffer for next set of packed
messages.
        }
        else
        {
            // Nothing to submit because packed message is empty.
        }
        Thread.sleep(1000);
    }
 }
```

5.1.10.2   Non-interactive Example of Packing

The following example illustrates a Non-interactive Provider application setting up a basic **PackedMsg** object and packing messages together before submitting the **PackedMsg**.

```
public void SendPackedMessagesExample(OmmProvider provider, long itemHandle)
    {
        FieldList fieldList = new FieldList();
        PackedMsg packedMsg = new PackedMsg(provider);
        packedMsg.InitBuffer();
        UpdateMsg updateMsg = new UpdateMsg();

        for (int i = 0; i < 60; i++)
        {
            for (int j = 0; j < 10; j++)
            {
                fieldList.Clear();
                fieldList.AddReal(22, 3991 + j, OmmReal.MagnitudeTypes.EXPONENT_NEG_2);
                fieldList.AddReal(30, 10 + j, OmmReal.MagnitudeTypes.EXPONENT_0);

                updateMsg.Clear();
                updateMsg.ServiceName("NI_PUB").Name("IBM.N").Payload(fieldList.Complete());

                try
                {
                    packedMsg.AddMsg(updateMsg, itemHandle);
                }
                catch (OmmInvalidUsageException excp)
                {
                    //The API was unable to add the current message into the packed buffer.
                    //If messages have been successfully added to the packed buffer, submit them, get
                    //a new packed buffer, and add the current message into that new buffer.
                    if (excp.ErrorCode == OmmInvalidUsageException.ErrorCodes.BUFFER_TOO_SMALL)
                    {
                        if (packedMsg.PackedMsgCount() > 0) // Packed message has some data.
                        {
                            // Submit the messages we've already packed, get a new packed buffer,
                            //and add the current message.
                            provider.Submit(packedMsg); // Submit packed message on OmmProvider
                            packedMsg.InitBuffer(); // Re-initialize buffer for next set of packed
                                    messages.
                            packedMsg.AddMsg(updateMsg, itemHandle); // Add missed message with
                                    its item handle.
                        }
                        else
                        {
                            //Packed buffer too small to add even first message.
                            //Consider initializing the buffer to a higher value than the default 6000
                                    bytes if needed.
```

```
                        //See InitBuffer() methods for more details.
                    }
                }
                else
                {
                    // Handle other exceptions from AddMsg() here.
                }
            }
        }

        if (packedMsg.PackedMsgCount() > 0)
        {
            provider.Submit(packedMsg);
            packedMsg.InitBuffer(); // Re-initialize buffer for next set of packed messages.
        }
        else
        {
            // Nothing to submit because packed message is empty.
        }
        Thread.Sleep(1000);
    }
}
```

5.2   IOmmProviderClient Class

5.2.1   IOmmProviderClient Description

The **IOmmProviderClient** class provides a callback mechanism through which applications receive OMM messages on items for which they subscribe. The **IOmmProviderClient** is a parent class that implements empty, default callback methods. Applications must implement their own class (inheriting from **IOmmProviderClient**), 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 **IOmmProviderClient** 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: IOmmProviderClient

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

```
class AppClient : IOmmProviderClient
{
    bool m_connectionUp;

    public bool IsConnectionUp()
```

1. Interactive Provider Only

```
    {
        return m_connectionUp;
    }

    public void OnRefreshMsg(RefreshMsg refreshMsg, IOmmProviderEvent evt)
    {
        Console.WriteLine("Received Refresh. Item Handle: " + evt.Handle + " Closure: " + evt.Closure);

        Console.WriteLine("Item Name: " + (refreshMsg.HasName ? refreshMsg.Name() : "<not set>"));
        Console.WriteLine("Service Name: " + (refreshMsg.HasServiceName ? refreshMsg.ServiceName() :
            "<not set>"));

        Console.WriteLine("Item State: " + refreshMsg.State());

        if (refreshMsg.State().StreamState == OmmState.StreamStates.OPEN)
        {
            if (refreshMsg.State().DataState == OmmState.DataStates.OK)
                m_connectionUp = true;
            else
                m_connectionUp = false;
        }
        else
            m_connectionUp = false;
    }

    public void OnStatusMsg(StatusMsg statusMsg, IOmmProviderEvent evt)
    {
        Console.WriteLine("Received Status. Item Handle: " + evt.Handle + " Closure: " + evt.Closure);

        Console.WriteLine("Item Name: " + (statusMsg.HasName ? statusMsg.Name() : "<not set>"));
        Console.WriteLine("Service Name: " + (statusMsg.HasServiceName ? statusMsg.ServiceName() :
            "<not set>"));

        if (statusMsg.HasState)
        {
            Console.WriteLine("Item State: " + statusMsg.State());
            if (statusMsg.State().StreamState == OmmState.StreamStates.OPEN)
            {
                if (statusMsg.State().DataState == OmmState.DataStates.OK)
                    m_connectionUp = true;
                else
                {
                    m_connectionUp = false;
                }
            }
            else
                m_connectionUp = false;
        }
    }
}
```

5.2.3   Interactive Provider Example: IOmmProviderClient

```csharp
class AppClient : IOmmProviderClient
{
    public long ItemHandle = 0;
    public string OrderNr = "100";

    public void OnReqMsg(RequestMsg reqMsg, IOmmProviderEvent providerEvent)
    {
        switch (reqMsg.DomainType())
        {
            case EmaRdm.MMT_LOGIN:
                ProcessLoginRequest(reqMsg, providerEvent);
                break;
            case EmaRdm.MMT_MARKET_BY_ORDER:
                ProcessMarketByOrderRequest(reqMsg, providerEvent);
                break;
            default:
                ProcessInvalidItemRequest(reqMsg, providerEvent);
                break;
        }
    }

    void ProcessLoginRequest(RequestMsg reqMsg, IOmmProviderEvent providerEvent)
    {
        providerEvent.Provider.Submit(new RefreshMsg().DomainType(EmaRdm.MMT_LOGIN)
            .Name(reqMsg.Name()).NameType(EmaRdm.USER_NAME)
            .Complete(true).Solicited(true)
            .State(OmmState.StreamStates.OPEN, OmmState.DataStates.OK, OmmState.StatusCodes.NONE,
            "Login accepted"),
            providerEvent.Handle);
    }

    void ProcessMarketByOrderRequest(RequestMsg reqMsg, IOmmProviderEvent providerEvent)
    {
        if (ItemHandle != 0)
        {
            ProcessInvalidItemRequest(reqMsg, providerEvent);
            return;
        }

        FieldList mapSummaryData = new FieldList();
        mapSummaryData.AddEnumValue(15, 840);
        mapSummaryData.AddEnumValue(53, 1);
        mapSummaryData.AddEnumValue(3423, 1);
        mapSummaryData.AddEnumValue(1709, 2);

        FieldList entryData = new FieldList();
        entryData.AddRealFromDouble(3427, 7.76, OmmReal.MagnitudeTypes.EXPONENT_NEG_2);
        entryData.AddRealFromDouble(3429, 9600);
```

```
        entryData.AddEnumValue(3428, 2);
        entryData.AddRmtes(212, new EmaBuffer(Encoding.ASCII.GetBytes("Market Maker")));

        Map map = new Map();
        map.SummaryData(mapSummaryData.Complete());

        map.AddKeyAscii(OrderNr, Access.MapAction.ADD, entryData.Complete());

        providerEvent.Provider.Submit(new RefreshMsg().DomainType(EmaRdm.MMT_MARKET_BY_ORDER)
            .Name(reqMsg.Name()).ServiceName(reqMsg.ServiceName()).Solicited(true)
            .State(OmmState.StreamStates.OPEN, OmmState.DataStates.OK, OmmState.StatusCodes.NONE,
                "Refresh Completed")
            .Payload(map.Complete()).Complete(true),
            providerEvent.Handle);

        ItemHandle = providerEvent.Handle;
    }

    void ProcessInvalidItemRequest(RequestMsg reqMsg, IOmmProviderEvent providerEvent)
    {
        providerEvent.Provider.Submit(new StatusMsg()
            .Name(reqMsg.Name()).ServiceName(reqMsg.ServiceName())
            .State(OmmState.StreamStates.CLOSED, OmmState.DataStates.SUSPECT, OmmState
                .StatusCodes.NOT_FOUND, "Item not found"),
            providerEvent.Handle);
    }
}
```

5.3   OmmNiProviderConfig and OmmIProviderConfig Classes

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

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

6   Consuming Data from the Cloud

6.1   Workflows Overview

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

- Credential Management (for details, refer to Section 6.3)

- Service Discovery (for details, refer to Section 6.5)

- Consuming Market Data (for details, refer to Section 6.6)

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

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 6.4.

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

The Enterprise Transport API will determine which authentication version to use based on the inputs. By default, for cloud connections the Enterprise Message 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*. 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 a Refinitiv Real-Time Advanced Distribution Server in the cloud, you must use a `ChannelType` of `RSSL_ENCRYPTED` (for details on `ChannelType`, refer to the *Enterprise Message API C# Configuration Guide*).

6.3   Credential Management

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

If an `IOmmOAuth2ConsumerClient` is specified when creating the `OmmConsumer` object, the API will callback `IOmmOAuth2ConsumerClient.OnOAuth2CredentialRenewal` whenever credentials are required. This call back must call `OmmConsumer.RenewOAuthCredentials` to provide the updated credentials.

NOTE: `OmmConsumer.RenewOAuthCredentials` can only be called during the callback.

6.4   Version 2 Authentication Using OAuth Client Credentials

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

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

- • When the consumer disconnects and goes into a reconnection state.
- • If the **Channel** stays in reconnection long enough to get close to the expiry time of the Access Token.

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

6.4.1   Configuring and Managing Version 2 Credentials

The client ID and client secret or private JWK must be set on the **OmmConsumer** object as described in Section of the *Enterprise Transport API C# Edition Value Added Developers Guide*. The **IOmmOAuth2ConsumerClient** will handle the credentials the same way as Version 1, with an **IOmmOAuth2ConsumerClient** callback for credentials if the user does not wish for the **IOmmOAuth2ConsumerClient** to store them.

6.4.1.1   JWT Credentials Handling

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

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

6.4.2   Version 2 OAuth Client Credentials Token Lifespan

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

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.

6.5   Service Discovery

After obtaining a token (for details, refer to Section 6.4), the Enterprise Message API can perform a service discovery against the Refinitiv Data Platform to obtain connection details for the Refinitiv Real-Time — Optimized. To discover endpoints, application may rely either on file or programmatic configuration. This is accomplished by making a REST query to the Service Discovery service. EMA API may be configured to perform this query and choose an endpoint (host and port) in a specified region. Or, EMA application may interact with a pre-defined service discovery object (see **ServiceEndpointDiscovery**) to customize choosing endpoint(s).

For service discovery performed by API, see Cons113 example. For service discovery performed in application, see Cons450 example.

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 Message 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 Message API identifies a Refinitiv Real-Time — Optimized endpoint with built-in resiliency whose regional location matches the API's location setting in `ChannelGroup` (for details, refer to Section 3.2.2 "Universal Channel Entry Parameters" of the *Enterprise Message API C# Edition Configuration Guide*). If you do not specify a location, the Enterprise Message 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 Message API chooses to connect to the first endpoint listed.
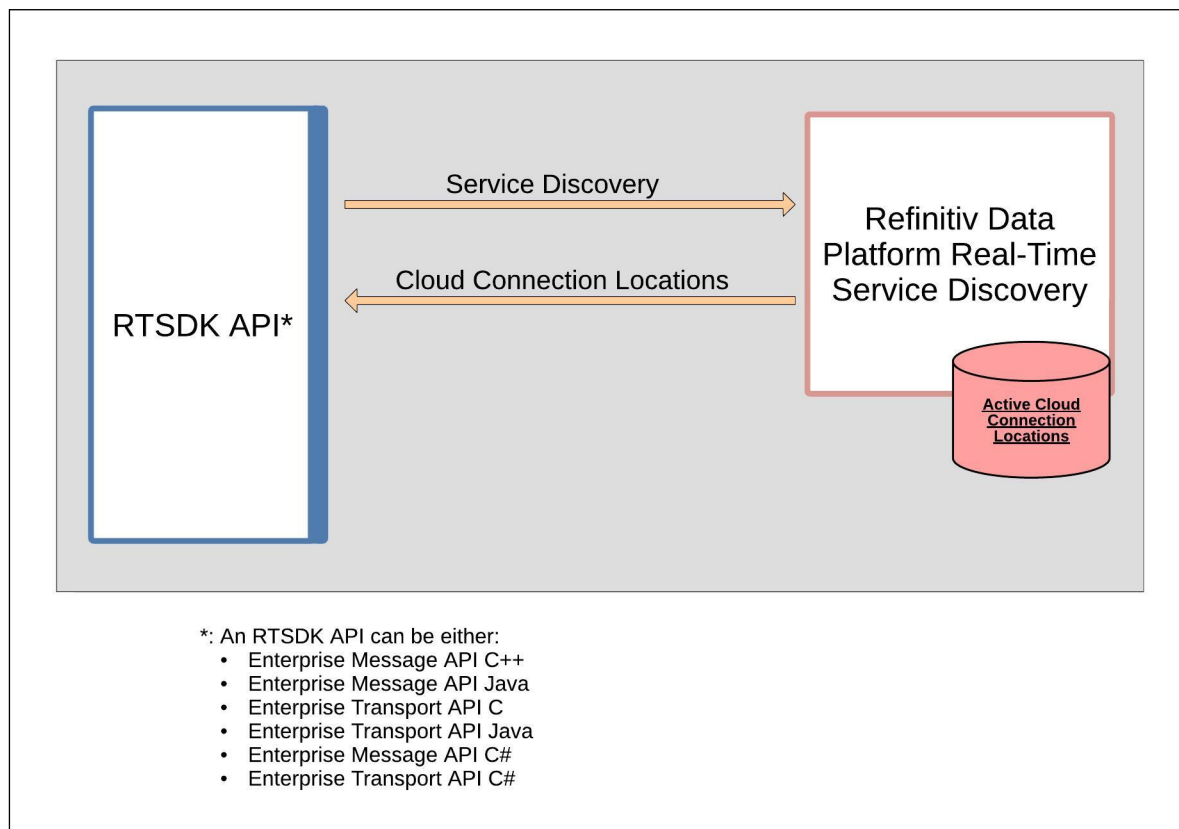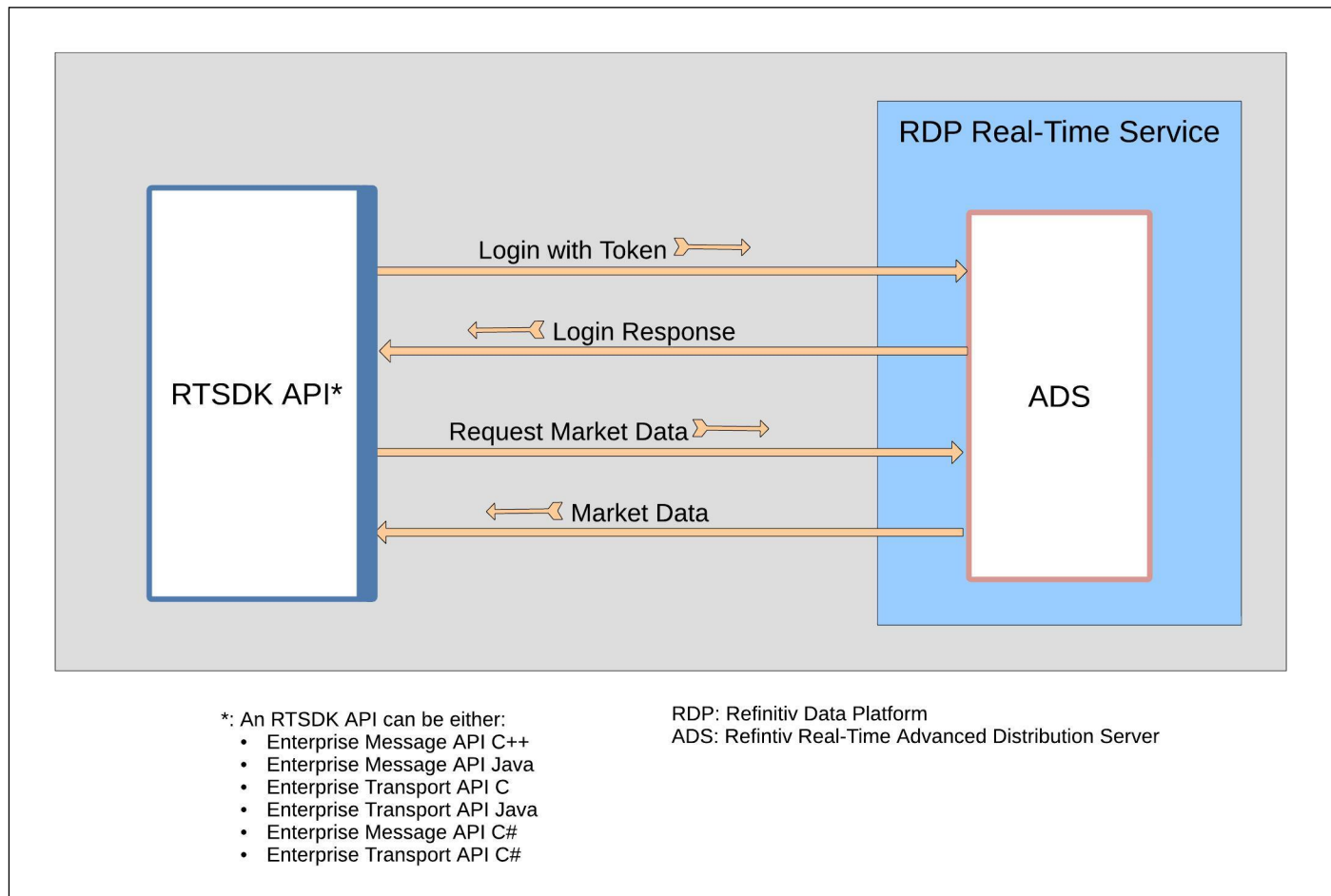

Figure 1. Service Discovery

6.6   Consuming Market Data

After obtaining its login token (for details, refer to Section 6.4) and running a service discovery (for details, refer to Section 6.5), the API can connect to the Refinitiv Real-Time Advanced Distribution Server in the cloud and obtain market data.

*: An RTSDK API can be either:
- Enterprise Message API C++
- Enterprise Message API Java
- Enterprise Transport API C
- Enterprise Transport API Java
- Enterprise Message API C#
- Enterprise Transport API C#

RDP: Refinitiv Data Platform
ADS: Refintiv Real-Time Advanced Distribution Server

6.7   HTTP Error Handling for Reactor Token Reissues

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

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

6.8   Cloud Connection Use Cases

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

- Start to finish session management (for details, refer to Section 6.8.1)

- Explicit service discovery option for applications (for details, refer to Section 6.8.2)

6.8.1   Session Management Use Case

In this use case, the Enterprise Message API manages the entire connection from start to finish. To use session management, you need to configure the API to enable session management. To do so, in the ChannelGroup, set the Channel entry parameter **EnableSessionManagement**).

The API exhibits the following behavior for this use case:

1.   Obtains a token (according to the details in Section 6.4).

2.   Queries service discovery (according to the details in Section 6.5).

3.   Consumes market data (according to the details in Section 6.6).

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

With session management enabled, application may specify a host and port in ChannelGroup parameters. In this case, the Enterprise Message API exhibits the same behavior listed above, but ignores the endpoints it receives from the service discovery and connects to the specified host and port.

6.8.2   Explicit Service Discovery Use Case

Application has the option to do a service discovery, parse the results, and choose an endpoint to pass into API. The API exhibits the following behavior when application does an explicit service discovery:

1.   Obtains a token (according to the details in Section 6.4).

2.   Queries service discovery (according to the details in Section 6.5).

Enterprise Message API'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.

6.9   Logging of Authentication and Service Discovery Interaction

If needed, you can log interactions with the Refinitiv Data Platform. To enable logging, use the parameters **RestEnableLog** and **RestLogFileName** in the EMA configuration file or programmatic configuration in the Consumer Group. If Service Discovery is done from the application, logging may be enabled only via function call configuration. For details on these parameters, refer to the *Enterprise Message API C# Configuration Guide*.

6.9.1   Logged Request Information

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

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

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

6.9.2   Logged Response Information

With logging turned on in the fashion mentioned in , the Enterprise Message 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
```

7   Troubleshooting and Debugging

7.1   EMA Logger Usage

The Enterprise Message API provides a logging mechanism useful for debugging runtime issues. In the default configuration, Enterprise Message API 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 Enterprise Message API logs event (to log every event, only error events, or nothing). For further details on managing and configuring the EMA logging function, refer to the *Enterprise Message API C# Edition Configuration Guide.*

7.2   OMM Error Client Class

7.2.1   IOmmConsumerErrorClient and IOmmProviderErrorClient Descriptions

Enterprise Message API has two Error Client classes: **IOmmConsumerErrorClient** and **IOmmProviderErrorClient**. These two classes are an alternate error notification mechanism in the Enterprise Message API, which you can use instead of the default error notification mechanism (i.e., **OmmException**, for details, refer to Section 7.3). 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: OmmConsumerErrorClient

The following example illustrates an application error client and depicts simple processing of the **OnInvalidHandle()** method.

```csharp
class AppErrorClient : IOmmConsumerErrorClient
{
    public void OnInvalidHandle(long handle, String text)
    {
        Console.WriteLine("onInvalidHandle callback function" + "\nInvalid handle: " + handle + "\nError
text: " + text);
    }

    public void OnInvalidUsage(String text, int errorCode) {
        Console.WriteLine("onInvalidUsage callback function" + "\nError text: " + text +" , Error code:
" + errorCode);
    }
}
```

7.3 OmmException Class

If the Enterprise Message API detects an error condition, the Enterprise Message API might throw an exception. All exceptions in the Enterprise Message API 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 Enterprise Message API usage or application design errors.

The Enterprise Message API supports the following exception types:

- `OmmInvalidConfigurationException`: Thrown when the Enterprise Message API 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 Enterprise Message API detects invalid interface usage.

- `OmmOutOfRangeException`: Thrown when a passed-in parameter lies outside the valid range.

- `OmmUnsupportedDomainTypeException`: Thrown if domain type specified on a message is not supported.

7.4 Creating a DACSLOCK for Publishing Permission Data

Provider applications can create a DACSLocks and publish it to permission data on the Refinitiv Real-Time Distribution System. A DACSLock controls access to data by users. For further details on the DACSLock API, refer to the *Enterprise Message API C# Edition DACSLock Library*.

The following example code illustrates how to create a DACSLock.

```
List<uint> peList = new() { 1001 };
AuthorizationLock authLock = new(serviceID: 261, AuthorizationLock.OperatorEnum.OR, peList);
AuthorizationLockData lockData = new();
AuthorizationLockStatus retStatus = new();
LockResultEnum result = authLock.GetLock(lockData, retStatus);
if (result == LockResultEnum.LOCK_SUCCESS)
{
    Console.WriteLine("Success - GetLock()");
}
else
{
    Console.WriteLine("Failure - GetLock()");
}
```

Document ID: EMACSharp320UM.240
Date of issue: April 2024

REFINITIV™