Elektron Message API C++ Edition V3.0.1

ELEKTRON MESSAGE API DEVELOPERS GUIDE

Document Version: 3.0.1
Date of issue: 05 February 2016
Document ID: EMAC301UM.160

THOMSON REUTERS

Contents

Chapter 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 Thomson Reuters 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 Thomson Reuters 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
ADS	Advanced Distribution Server
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
EED	Elektron Edge Device
EMA	Elektron Message API, referred to simply as the Message API
EOA	Elektron Object API, referred to simply as the Object API.
ETA	Elektron Transport API, referred to simply as the Transport API
EWA	Elektron Web API
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol (Secure)
OMM	Open Message Model
QoS	Quality of Service

Table 1: Acronyms and Abbreviations

ACRONYM	MEANING
EDF	Elektron Data Feeds
EDF Direct	Elektron Data Feed Direct
RDM	Reuters Domain Model
RMTES	Reuters Multi-Lingual Text Encoding Standard
RSSL	Reuters Source Sink Library
RWF	Reuters Wire Format
TREP	Thomson Reuters Enterprise Platform
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. The *EMA C++ Edition Reference Manual*
5. *Transport API C Edition Developers Guide*
6. *Transport API C Edition Value Added Components Developers Guide*

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@thomsonreuters.com.
- Mark up the PDF using the **Comment** feature in Adobe Reader. After adding your comments, you can submit the entire PDF to Thomson Reuters by clicking **Send File** in the **File** menu. Use the apidocumentation@thomsonreuters.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 an orange 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();
```

Chapter 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 applications.
- Enables applications to source market data from, and provide it to, different components that support OMM and/or RWF (e.g. Elektron, Enterprise Platform, 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 Supported Features

FEATURE	DESCRIPTION
New in 3.0.1! ADS Multicast	Applications can connect to the ADS multicast component by specifying the connection type **RSSL_RELIABLE_MCAST**.
New in 3.0.1! Connection Failover	You can specify a list of failover servers via the **ChannelSet** configuration. If a consumer's connection attempt fails, EMA attempts to connect to the next channel in the **ChannelSet** list.
Default Admin Domain Requests	The EMA uses default login, directory, and dictionary requests when connecting to a server: • The Login request uses the current user's name and defaults all other login attributes. • The Directory request calls for all services and filters. • Default RDM dictionaries are requested from the first service that starts up and accepts requests.
Configurable Admin Domain Requests	The EMA provides the means to modify default Admin domain requests.
Batch Request	An application can use a single request message to specify interest in multiple items via the item list.
Dynamic View	An application can specify a subset of fields or elements for a particular item.
Optimized Pause and Resume	An 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 Config	The application can programmatically specify and overwrite EMA configuration.
File Config	An EMA configuration can be specified in an **EmaConfig.xml** file.
Tunnel Stream	EMA supports private streams, with additional associated behaviors (e.g., end-to-end authentication, guaranteed 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

2.3 Product 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. 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 uses Elektron Transport API decoding and encoding functions for reading and populating OMM containers and messages. Each OMM container and message is represented by a respective EMA interface class, which provide 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()**).

The EMA 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**. The EMA will not run a second thread, if the **OmmConsumerConfig** operation model is set to the **OmmConsumerConfig::UserDispatch**. 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 the dispatch of messages, it can result in slow consumer behavior.

2.4 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 guaranteed 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 TREP 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.

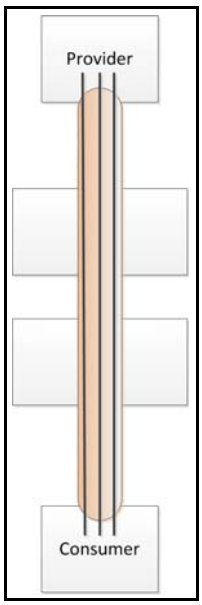

Figure 1. Tunnel Stream

2.5 Learning EMA

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

When learning the EMA, Thomson Reuters recommends you set up a sandbox environment where developers can experiment with various iterations of EMA applications. To facilitate an experiment-based learning experience, the EMA package provides a set of training examples which showcase the usage of EMA interfaces in increasing levels of complexity and sophistication. The complexity of an example is reflected in its series number:

- 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 like posting, generic message, programmatic configuration, and etc.

While coding and debugging applications, developers are encouraged to refer to the and or to the features provided by their IDE (e.g., IntelliSense).

Chapter 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, guaranteed delivery, and persistency. 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;
   map.addKeyAscii( "entry_1", MapEntry::AddEnum, fieldList );

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

      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 );

CosGuarantee cosGuarantee;
cosGuarantee.type( CosGuarantee::NoneEnum );

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

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

Chapter 4 Consumer Classes

4.1 OmmConsumer Class

The **OmmConsumer** class is the main 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**).

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 5.2).

  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).
- 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;
}
```

4.2 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

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*.

Chapter 5 Troubleshooting and Debugging

5.1 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*.

5.2 OmmConsumerErrorClient Class

5.2.1 OmmConsumerErrorClient Description

The **OmmConsumerErrorClient** class is an alternate error notification mechanism in the EMA, which you can use instead of **OmmConsumer**'s default error notification mechanism (i.e., **OmmException**, for details, refer to Section 5.3). Both mechanisms deliver the same information and detect the same error conditions. To use **OmmConsumerErrorClient**, applications need to implement their own error client class and override the default implementation of each method.

5.2.2 Example: OmmConsumerErrorClient

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

```
class AppErrorclient : public OmmConsumerErrorClient
{
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);

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

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

5.3 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: Thomson Reuters 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`** class methods.
- **`OmmInvalidUsageException`**: Thrown when the EMA detects invalid interface usage.
- **`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.
- **`OmmUnsuppoprtedDomainTypeException`**: Thrown if domain type specified on a message is not supported.

Document ID: EMAC301UM.160
Date of issue: 05 February 2016