

Elektron Message API Java Edition V3.0

ELEKTRON MESSAGE API DEVELOPERS GUIDE



© Thomson Reuters 2016. All rights reserved.

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

This document contains information proprietary to Thomson Reuters and may not be reproduced, disclosed, or used in whole or part without the express written permission of Thomson Reuters.

Any Software, including but not limited to, the code, screen, structure, sequence, and organization thereof, and Documentation are protected by national copyright laws and international treaty provisions. This manual is subject to U.S. and other national export regulations.

Nothing in this document is intended, nor does it, alter the legal obligations, responsibilities or relationship between yourself and Thomson Reuters as set out in the contract existing between us.

Contents

Chapter 1	Guide Introduction	1
1.1	About this Manual	1
1.2	Audience	1
1.3	Programming Languages	1
1.4	Acronyms and Abbreviations	1
1.5	References	2
1.6	Documentation Feedback	2
1.7	Document Conventions	3
Chapter 2	Product Overview	4
2.1	EMA Product Description	4
2.2	Product Documentation and Learning EMA	4
2.3	Supported Features	5
2.4	Product Architecture	6
Chapter 3	OMM Containers and Messages	7
3.1	Overview	7
3.2	Classes	8
3.2.1	<i>DataType Class</i>	<i>8</i>
3.2.2	<i>DataCode Class</i>	<i>8</i>
3.2.3	<i>Data Class</i>	<i>8</i>
3.2.4	<i>Msg Class</i>	<i>8</i>
3.2.5	<i>OmmError Class</i>	<i>9</i>
3.3	Working with OMM Containers	10
3.3.1	<i>Example: Populating a FieldList Class</i>	<i>10</i>
3.3.2	<i>Example: Extracting Information from a FieldList Class</i>	<i>10</i>
3.3.3	<i>Example: Extracting FieldList information using a Downcast operation</i>	<i>11</i>
3.4	Working with OMM Messages	13
3.4.1	<i>Example: Populating the GenericMsg with an ElementList Payload</i>	<i>13</i>
3.4.2	<i>Example: Extracting Information from the GenericMsg class</i>	<i>13</i>
Chapter 4	Consumer Classes	14
4.1	OmmConsumer Class	14
4.1.1	<i>Connecting to a Server and Opening Items</i>	<i>14</i>
4.1.2	<i>Opening Items Immediately After OmmConsumer Object Instantiation</i>	<i>14</i>
4.1.3	<i>Destroying the OmmConsumer Object</i>	<i>15</i>
4.1.4	<i>Example: Working with the OmmConsumer Class</i>	<i>15</i>
4.1.5	<i>Working with Items</i>	<i>15</i>
4.1.6	<i>Example: Working with Items</i>	<i>16</i>
4.2	OmmConsumerClient Class	17
4.2.1	<i>OmmConsumerClient Description</i>	<i>17</i>
4.2.2	<i>Example: OmmConsumerClient</i>	<i>17</i>
4.3	OmmConsumerConfig Class	17
Chapter 5	Troubleshooting and Debugging	18
5.1	EMA Logger Usage	18
5.2	OmmConsumerErrorClient Class	18
5.2.1	<i>OmmConsumerErrorClient Description</i>	<i>18</i>
5.2.2	<i>Example: OmmConsumerErrorClient</i>	<i>18</i>
5.3	OmmException Class	19

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 Java 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 Languages

The Message API is written using the Java 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
EDF	Elektron Data Feeds
EED	Elektron Edge Device
EMA	Elektron Message API, referred to simply as the Message API
ETA	Elektron Transport API, referred to simply as the Transport API
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol (Secure)
OMM	Open Message Model
QoS	Quality of Service
RDF Direct	Reuters Data Feed Direct

Table 1: Acronyms and Abbreviations

ACRONYM	MEANING
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 Java Edition RDM Usage Guide*
2. *API Concepts Guide*
3. *Elektron Message API Java Configuration Guide*
4. *The EMA Java Edition Reference Manual*
5. *Transport API Java Edition Developers Guide*
6. *Transport API Value Added Components Developers Guide* specific to the programming language you use.

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.
- Add your comments to the PDF using Adobe's **Comment** feature. After adding your comments, 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:

- Java 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 = new AppClient();

OmmConsumerConfig config = EmaFactory.createOmmConsumerConfig();

OmmConsumer consumer =
    EmaFactory.createOmmConsumer(config.operationModel(OperationModel.USER_DISPATCH)
        .host("localhost:14002").username("user"));

ReqMsg reqMsg = EmaFactory.createReqMsg();

consumer.registerClient(reqMsg.domainType(EmaRdm.MMT_MARKET_BY_PRICE).serviceName(
    "DIRECT_FEED").name("BBH. ITC"), client);
```

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 **accessor** functionality to populate and read OMM containers and messages. EMA takes advantage of fluent interface design, which users can leverage to set disparate values of the same message or container by stringing respective interface methods together, one after the other. Fluent interfaces provide the means for visual code simplification which helps in understanding and debugging applications.

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

2.2 Product Documentation and Learning EMA

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. EMA is designed to facilitate a hands-on (experiment-based) learning experience (versus a documentation-based methodology). To support a hands-on learning methodology, the EMA package provides a set of training examples which showcase the usage of EMA interfaces in increasing levels of complexity and sophistication. 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., int, String, and Buffer).
- The 300 and 400 series examples depict usage of particular EMA features such as posting, generic message, programmatic configuration, and etc.

While coding and debugging applications, developers are encouraged to refer to the *EMA Java Edition Reference Manual* Java and or to the features provided by their IDE (e.g., Eclipse).

2.3 Supported Features

FEATURE	DESCRIPTION
Default Admin Domain Requests	The EMA uses default login, directory, and dictionary requests when connecting to a server: <ul style="list-style-type: none"> • 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.
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::asHex()	Applications can obtain binary representations of all OMM containers, primitives, and messages.
File Config	An EMA configuration can be specified in an EmaConfig.xml file.

Table 2: Supported Features

2.4 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 Value Added Component Developers Guide* specific to the programming language you use. 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.OperationModel.API_DISPATCH`. The EMA will not run a second thread, if the `OmmConsumerConfig` operation model is set to the `OmmConsumerConfig.OperationModel.USER_DISPATCH`. Without running a second thread, the application is responsible for calling the `OmmConsumer::dispatch()` method to dispatch all received messages.

The `EmaFactory` class is a factory class that is used to create OMM message objects and OMM container objects. `EmaFactory` can also create `OmmConsumer` and `OmmConsumerConfig` objects.



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

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 Java Collections Framework approach is 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 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 `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, 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.DataTypes.FIELD_LIST**, **DataType.DataTypes.REFRESH_MSG**). You can use the **Data::dataType()** 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.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 Class

The **DataCode** class 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 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 **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 )
{
    for (ElementEntry elementEntry : elementList)
    {
        if ( elementEntry.code() != Data.DataCode.BLANK )
            switch (elementEntry.loadType())
            {
                case DataTypes.REAL:
                    System.out.println(elementEntry.real().asDouble());
                    break;
                case DataTypes.ERROR:
                    System.out.println(elementEntry.error().errorCode() + " (" +
                        elementEntry.error().errorCodeAsString() + ")");
                    break;
            }
    }
}
```

3.3 Working with OMM Containers

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

Each of these classes extends the Java collections framework 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.

```
FieldList fieldList = EmaFactory.createFieldList();

fieldList.info( 1, 1 );
fieldList.add( EmaFactory.createFieldEntry().uintValue( 1, 64 ) );
fieldList.add( EmaFactory.createFieldEntry().real( 6, 11, OmmReal.MagnitudeType.EXPONENT_NEG_2 ) );
fieldList.add( EmaFactory.createFieldEntry().date( 16, 1999, 11, 7 ) );
fieldList.add( EmaFactory.createFieldEntry().time( 18, 02, 03, 04, 005 ) );
```

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();
    }

    for( FieldEntry fieldEntry : fieldList )
    {
        if( fieldEntry.code() != Data.DataCode.BLANK )
            switch( fieldEntry.loadType() )
            {
                case DataTypes.ASCII:
                    System.out.println(fieldEntry.ascii());
                    break;
                case DataTypes.ERROR:
                    System.out.println( elementEntry.error().errorCode() + " ( " +
```

```

                elementEntry.error().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.

```

void decodeFieldList( FieldList fl)
{
    if (fl.hasInfo())
        System.out.println("FieldListNum: " + fl.infoFieldListNum() + " DictionaryId: " +
            fl.infoDictionaryId());

    for (FieldEntry fieldEntry : fl)
    {
        System.out.println("Load");
        decode(fieldEntry.load());
    }
}

void decode(Data data)
{
    if (Data.DataCode.BLANK == data.code())
        System.out.println("Blank data");
    else
        switch (data.dataType())
        {
            case DataTypes.REFRESH_MSG :
                decodeRefreshMsg( (RefreshMsg)data );
                break;
            case DataTypes.UPDATE_MSG :
                decodeUpdateMsg( (UpdateMsg)data );
                break;
            case DataTypes.FIELD_LIST :
                decodeFieldList( (FieldList)data );
                break;
            case DataTypes.MAP :
                decodeMap( (Map)data );
                break;
            case DataTypes.NO_DATA :
                System.out.println("NoData");
                break;
            case DataTypes.TIME :
                System.out.println("OmmTime: " + ((OmmTime)data).toString());
                break;
            case DataTypes.DATE :

```

```

        System.out.println("OmmDate: " + ((OmmDate)data).toString());
        break;
    case DataTypes.REAL :
        System.out.println("OmmReal::asDouble: " + ((OmmReal)data).asDouble());
        break;
    case DataTypes.INT :
        System.out.println("OmmInt: " + ((OmmInt)data).intValue());
        break;
    case DataTypes.UINT :
        System.out.println("OmmUInt: " + ((OmmUInt)data).longValue());
        break;
    case DataTypes.ENUM :
        System.out.println("OmmEnum: " + ((OmmEnum)data).enumValue());
        break;
    case DataTypes.ASCII :
        System.out.println("OmmAscii: " + ((OmmAscii)data).ascii());
        break;
    case DataTypes.ERROR :
        System.out.println("Decoding error: " + ((OmmError)data).errorCodeAsString());
        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**.

```
GenericMsg genMsg = EmaFactory.createGenericMsg();

FieldList nestedFieldList = EmaFactory.createFieldList();
nestedFieldList.add(EmaFactory.createFieldEntry().real(22, 34,
    OmmReal.MagnitudeType.EXPONENT_POS_1));

genMsg.domainType( 200 ).name( "TR.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.hasName() )
        System.out.println("Name: " + genMsg.name());

    if ( genMsg.hasExtendedHeader() )
    {
        ByteBuffer header = genMsg.extendedHeader();
    }

    switch ( genMsg.payload().dataType() )
    {
        case DataTypes.FIELD_LIST :
            decode( genMsg.payload().fieldList() );
            break;
    }
}
```


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.
- **(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 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 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

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 client = new AppClient();

    OmmConsumerConfig config = EmaFactory.createOmmConsumerConfig();

    consumer = EmaFactory.createOmmConsumer(
        config.host("localhost:14002").username("user") );
    ReqMsg reqMsg = EmaFactory.createReqMsg();
    consumer.registerClient( reqMsg.serviceName("DIRECT_FEED").name("IBM.N"), client );

    Thread.sleep( 60000 );
}
catch (InterruptedException | OmmException excp)
{
    System.out.println(excp.getMessage());
}
finally
{
    if (consumer != null) consumer.uninitialize();
}
```

4.1.5 Working with Items

The EMA 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 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 onRefreshMsg( RefreshMsg refreshMsg, OmmConsumerEvent event )
{
    System.out.println("Received refresh message for item handle = " + event.handle());
    System.out.println(refreshMsg);
}

public static void main(String[] args)
{
    OmmConsumer consumer = null;
    try
    {
        AppClient client = new AppClient();
        OmmConsumerConfig config = EmaFactory.createOmmConsumerConfig();
        consumer = EmaFactory.createOmmConsumer(
            config.host("localhost:14002").username("user") );

        ReqMsg reqMsg = EmaFactory.createReqMsg();
        long closure = 1;
        long itemHandle = consumer.registerClient( reqMsg.serviceName( "DIRECT_FEED" ).name(
            "IBM.N" ), client, closure );

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

        reqMsg.clear();
        PostMsg postMsg = EmaFactory.createPostMsg();
        FieldList nestedFieldList = EmaFactory.createFieldList();
        nestedFieldList.add( EmaFactory.createFieldEntry().uintValue(1, 100) );
        consumer.submit( postMsg.payload(nestedFieldList), itemHandle );

        Thread.sleep( 60000 );
    }
    catch (InterruptedException | OmmException excp)
    {
        System.out.println(excp.getMessage());
    }
    finally
    {
        if (consumer != null) consumer.uninitialize();
    }
}

```

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()`. 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.

```
class AppClient implements OmmConsumerClient
{
    public void onRefreshMsg( RefreshMsg refreshMsg, OmmConsumerEvent event)
    {
        if ( refreshMsg.hasMsgKey() )
            System.out.println("Item Name: " +refreshMsg.name() +"Service Name: " +
                               refreshMsg.serviceName());

        System.out.println("Item State: " + refreshMsg.state().toString());

        if ( refreshMsg.payload().dataType() != DataTypes.NO_DATA )
            decode( refreshMsg.payload().data() );
    }
}
```

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(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.

The EMA uses the SLF4J logging API, in which you can have the underlying logging backend be the Java standard logger utility package (`java.util.logging`), `log4j`, or other logger adapters which implement the SLF4J logging interface.

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). 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 implements OmmConsumerErrorClient
{
    public void onInvalidHandle( long handle, String text )
    {
        System.out.println("Handle = " + handle + ", text = " + text);
    }

    public void onInvalidUsage( String text )
    {
        System.out.println("Invalid Usage: " + text);
    }
}
```

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:

- `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.
- `OmmOutOfRangeException`: Thrown when a passed-in parameter lies outside the valid range.
- `OmmUnsupportedDomainTypeException`: Thrown if domain type specified on a message is not supported.

© 2016 Thomson Reuters. All rights reserved.

Republication or redistribution of Thomson Reuters content, including by framing or similar means, is prohibited without the prior written consent of Thomson Reuters. 'Thomson Reuters' and the Thomson Reuters logo are registered trademarks and trademarks of Thomson Reuters and its affiliated companies.

Any third party names or marks are the trademarks or registered trademarks of the relevant third party.

Document ID: EMAJ300UM.160

Date of issue: 30 April 2016



THOMSON REUTERS