Elektron Message API Java Edition V3.0.4

ELEKTRON MESSAGE API DEVELOPERS GUIDE

Document Version: 3.0.4
Date of issue: 9 December 2016
Document ID: EMAJ304UM.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 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 is the horizontally scalable service component within Thomson Reuters Enterprise Platform (TREP) providing high availability for publication and contribution messaging, subscription management with optional persistence, conflation and delay capabilities.
ADS	Advanced Distribution Server is the horizontally scalable distribution component within Thomson Reuters Enterprise Platform (TREP) 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
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
RDM	Reuters Domain Model
RMTES	Reuters Multi-Lingual Text Encoding Standard
RSSL	Reuters Source Sink Library
RWF	Reuters Wire Format, a Thomson Reuters proprietary format.
TR-DFD	Thomson Reuters Data Feed Direct
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. *EMA Java Edition Reference Manual*
5. *Transport API Java Edition Value Added Components Developers Guide*
6. *Transport API Java Edition Developers Guide*
7. The Thomson Reuters Professional Developer Community

1.6 Documentation Feedback

While we make every effort to ensure the documentation is accurate and up-to-date, if you notice any errors, or would like to see more details on a particular topic, you have the following options:

- Send us your comments via email at apidocumentation@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

- 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 and provider-type applications.
- Enables applications to source market data from, and provide it to, different components that support OMM and/or RWF (e.g. Elektron, 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

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

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

2.2.1 Consumer Examples

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

- 100-series examples simply open an item and print its received content to the screen (using the `Data::toString()` method). Applications in this series illustrate EMA support for stringification, containers, and primitives. Though useful for learning, debugging, and writing display applications, stringification by itself is not sufficient to develop more sophisticated applications.
- The 200 series examples illustrate how to extract information from OMM containers and messages in native data formats, (e.g., 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.

2.2.2 Provider Examples

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

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

2.3 Supported Features

FEATURE	DESCRIPTION
New in 3.0.4! Interactive Provider	EMA provides interactive provider connectivity (e.g., for ADH or other directly-connected consumers).
New in 3.0.3! Non-interactive Provider	Applications can connect to an ADH TREP component to non-interactively provide item data.
New in 3.0.3! 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 Messages	The EMA consumer uses default login, directory, and dictionary requests when connecting to a provider or ADS: • The Login request uses the current user’s name and defaults all other login attributes. • The Directory request calls for all services and filters. • RDM dictionaries are requested from the first available service that accepts requests. The EMA Non-interactive Provider uses the default login request and configured directory refresh when connecting to ADH: • The login request uses the current user's name and defaults all other login attributes. • The directory refresh message defaults all message attributes as well as status, while its payload is either hardcoded or read from the EMA configuration. The EMA Interactive Provider can use default, preconfigured directory and dictionary refresh messages. • The directory refresh message defaults appropriate message attributes as well as status, while its payload is either hard coded or read from the EMA configuration. • The dictionary refresh message is handled based on either its hard-coded configuration or read from the EMA configuration.
Configurable Admin Domain Messages	EMA provides the means to modify default Admin domain messages.
Batch Request	A consumer application can use a single request message to specify interest in multiple items via the item list.
Optimized Pause and Resume	A consumer application can send a request to the server to pause and resume item stream.
Single Open	The EMA supports application-selected, single-open functionality.
RMTES Decoder	The EMA provides a built-in RMTES decoder. If needed, the application can cache `RmtesBuffer` objects and apply all received changes to them.
`Data::toString()`	Prints all OMM containers, primitives, and messages to screen in a standardized output format (called “stringification”).

Table 2: Supported Features

FEATURE	DESCRIPTION
`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.
Tunnel Stream (also known as a Qualified Stream)	EMA supports private streams, with additional associated behaviors (e.g., end-to-end authentication, guaranteed delivery, and flow control).

Table 2: Supported Features (Continued)

2.4 Product Architecture

2.4.1 EMA Consumer Architecture

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

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

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

The EMA consumer will always have at least one thread, which is implemented by the VA Reactor and runs the internal, VA Reactor logic. For details on this thread, refer to the *Transport API Java 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.OperationModel.API_DISPATCH`**. If the **`OmmConsumerConfig`** operation model is set to the **`OmmConsumerConfig.OperationModel.USER_DISPATCH`**, the EMA will not run a second thread. Without running a second thread, the application is responsible for calling the **`Ommconsumer::dispatch()`** method to dispatch all received messages.

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

2.4.2 EMA Provider Architecture

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

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

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

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

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

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

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

- When running EMA as a non-interactive provider: the application must set `OmmNiProviderConfig.AdminControl.USER_CONTROL` through the `OmmNiProviderConfig::adminControlDirectory()` method.
- When running EMA 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::adminControlDirectory()` method.

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

2.4.3 EMA Codec Architecture

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

2.5 Tunnel Streams

By leveraging the Transport API Value Added Reactor, the EMA allows users to create and use special tunnel streams. A tunnel stream is a private stream that has additional behaviors associated with it, such as end-to-end line of sight for authentication and 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.

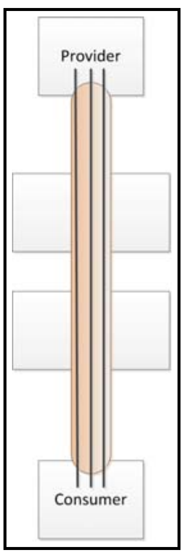

Figure 1. Tunnel Stream

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;
            }
    }
}
```

3.2.6 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 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;
    }
}
```

3.4.3 Example: Working with the TunnelStreamRequest Class

The following example illustrates how to use a **`TunnelStreamRequest`** class in a consumer application to open a tunnel stream.

```
ClassOfService cos = EmaFactory.createClassOfService().
        authentication(EmaFactory.createCosAuthentication().type
(CosAuthentication.CosAuthenticationType.OMM_LOGIN))
        .dataIntegrity(EmaFactory.createCosDataIntegrity().type
(CosDataIntegrity.CosDataIntegrityType.RELIABLE))
        .flowControl(EmaFactory.createCosFlowControl().type
(CosFlowControl.CosFlowControlType.BIDIRECTIONAL).recvWindowSize(1200))
        .guarantee(EmaFactory.createCosGuarantee().type(CosGuarantee.CosGuaranteeType.NONE));

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

Chapter 4 Consumer Classes

4.1 OmmConsumer Class

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

4.1.1 Connecting to a Server and Opening Items

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

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

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

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

  An application needs to override the default implementation of callback methods and provide its own business logic.
- **(Optional)** Implement an application error client class that inherits from the **`OmmConsumerErrorClient`** class (for details, refer to Section 6.2).

  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 Provider Classes

5.1 OmmProvider Class

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

5.1.1 Submitting Items

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

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

To establish a connection and submit items:

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

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

2. Create the appropriate **`Omm`*`ProviderType`*`Config`** object (for details, refer to Section 5.3):

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

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

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

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

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

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

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

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

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

6. Create an **`OmmProvider`** object and pass the **`Omm`*`ProviderType`*`Config`** object (and if needed, also pass in the application error client object).

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

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

5.1.3 Interactive Providers: Post OmmProvider Object Instantiation

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

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

- EMA Accepts the connection request from a consumer
- The application now needs to accept the login
- Submits the source directory information

5.1.4 Uninitialize the OmmProvider Object

For non-interactive providers, calling the `OmmProvider.uninitialize()` method causes the application to log out and disconnect from the connected ADH, at which time all items are closed.

For interactive providers, calling the `OmmProvider.uninitialize()` method causes EMA 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.

```
OmmProvider provider = null;
try
{
    OmmNiProviderConfig config = EmaFactory.createOmmNiProviderConfig();

    provider = EmaFactory.createOmmProvider( config.host( "localhost:14003" ).username(
            "user" ) );

    long itemHandle = 5;
```

```
    FieldList mapSummaryData = EmaFactory.createFieldList();
    mapSummaryData.add( EmaFactory.createFieldEntry().enumValue( 15, 840 ) );
    mapSummaryData.add( EmaFactory.createFieldEntry().enumValue( 53, 1 ) );
    mapSummaryData.add( EmaFactory.createFieldEntry().enumValue( 3423, 1 ) );
    mapSummaryData.add( EmaFactory.createFieldEntry().enumValue( 1709, 2 ) );

    FieldList mapKeyAscii = EmaFactory.createFieldList();
    mapKeyAscii.add( EmaFactory.createFieldEntry().realFromDouble( 3427, 7.76,
            MagnitudeType.EXPONENT_NEG_2 ) );
    mapKeyAscii.add( EmaFactory.createFieldEntry().realFromDouble( 3429, 9600 ) );
    mapKeyAscii.add( EmaFactory.createFieldEntry().enumValue( 3428, 2 ) );
    mapKeyAscii.add( EmaFactory.createFieldEntry().rmtes( 212, ByteBuffer.wrap( "Market
            Maker".getBytes() ) ) );

    Map map = EmaFactory.createMap();
    map.summaryData( mapSummaryData );
    map.add( EmaFactory.createMapEntry().keyAscii( "100", MapEntryActions.ADD, mapKeyAscii )
            );

    provider.submit( EmaFactory.createRefreshMsg().domainType( DomainTypes.MARKET_BY_ORDER
            ).serviceName( "NI_PUB" ).name( "AAO.V" )
        .state( OmmState.StreamState.OPEN, OmmState.DataState.OK, OmmState.StatusCode.NONE,
                "UnSolicited Refresh Completed" )
        .payload( map ).complete( true ), itemHandle );

    Thread.sleep( 1000 );

    for ( int i = 0; i < 60; i++ )
    {
        mapKeyAscii = EmaFactory.createFieldList();
        mapKeyAscii.add( EmaFactory.createFieldEntry().realFromDouble( 3427, 7.76 + i * 0.1,
                MagnitudeType.EXPONENT_NEG_2 ) );
        mapKeyAscii.add( EmaFactory.createFieldEntry().realFromDouble( 3429, 9600 ) );
        mapKeyAscii.add( EmaFactory.createFieldEntry().enumValue( 3428, 2 ) );
        mapKeyAscii.add( EmaFactory.createFieldEntry().rmtes( 212, ByteBuffer.wrap( "Market
                Maker".getBytes() ) ) );

        map = EmaFactory.createMap();
        map.add( EmaFactory.createMapEntry().keyAscii( "100", MapEntryActions.ADD,
                mapKeyAscii ) );

        provider.submit( EmaFactory.createUpdateMsg().serviceName( "NI_PUB" ).name( "AAO.V"
                ).domainType( DomainTypes.MARKET_BY_ORDER ).payload( map ), itemHandle );
        Thread.sleep( 1000 );
    }
}
```

```
catch ( InterruptedException | OmmException excp )
{
    System.out.println( excp.getMessage() );
}
finally
{
    if ( provider != null ) 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 = EmaFactory.createFieldList();
    Map map = EmaFactory.createMap();

    OmmIProviderConfig config = EmaFactory.createOmmIProviderConfig();

    provider = EmaFactory.createOmmProvider(config.port("14002"), appClient);

    while(appClient.itemHandle == 0) Thread.sleep(1000);

    for( int i = 0; i < 60; i++ )
    {
        fieldList.add(EmaFactory.createFieldEntry().realFromDouble(3427, 7.76 + i * 0.1,
                MagnitudeType.EXPONENT_NEG_2));
        fieldList.add(EmaFactory.createFieldEntry().realFromDouble(3429, 9600));
        fieldList.add(EmaFactory.createFieldEntry().enumValue(3428, 2));
        fieldList.add(EmaFactory.createFieldEntry().rmtes(212, ByteBuffer.wrap("Market
                Maker".getBytes())));

        map.add(EmaFactory.createMapEntry().keyAscii(appClient.OrderNr, MapEntry.MapAction.ADD,
                fieldList));

        provider.submit( EmaFactory.createUpdateMsg().domainType(EmaRdm.MMT_MARKET_BY_ORDER).payload(
                map ), appClient.itemHandle );

        map.clear();
        fieldList.clear();

        Thread.sleep(1000);
    }

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

5.1.7 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.

5.2 OmmProviderClient Class

5.2.1 OmmProviderClient Description

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

5.2.2 Non-Interactive Example: OmmProviderClient

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

```
class AppClient implements OmmProviderClient
{
    boolean _connectionUp;

    boolean isConnectionUp()
    {
        return _connectionUp;
    }

    public void onRefreshMsg(RefreshMsg refreshMsg, OmmProviderEvent event)
    {
        System.out.println("Received Refresh. Item Handle: " + event.handle() + " Closure: "
                + event.closure());

        System.out.println("Item Name: " + (refreshMsg.hasName() ? refreshMsg.name() : "<not
                set>"));
        System.out.println("Service Name: " + (refreshMsg.hasServiceName() ?
                refreshMsg.serviceName() : "<not set>"));

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

        if ( refreshMsg.state().streamState() == OmmState.StreamState.OPEN)
        {
            if (refreshMsg.state().dataState() == OmmState.DataState.OK)
                _connectionUp = true;
            else
                _connectionUp = false;
        }
        else
```

1. Interactive Provider Only

```
            _connectionUp = false;
    }

    public void onStatusMsg(StatusMsg statusMsg, OmmProviderEvent event)
    {
        System.out.println("Received Status. Item Handle: " + event.handle() + " Closure: " +
                event.closure());

        System.out.println("Item Name: " + (statusMsg.hasName() ? statusMsg.name() : "<not
                set>"));
        System.out.println("Service Name: " + (statusMsg.hasServiceName() ?
                statusMsg.serviceName() : "<not set>"));

        if (statusMsg.hasState())
        {
            System.out.println("Item State: " +statusMsg.state());
            if ( statusMsg.state().streamState() == OmmState.StreamState.OPEN)
            {
                if (statusMsg.state().dataState() == OmmState.DataState.OK)
                    _connectionUp = true;
                else
                {
                    _connectionUp = false;
                }
            }
            else
                _connectionUp = false;
        }
    }

    public void onGenericMsg(GenericMsg genericMsg, OmmProviderEvent event){}
    public void onAllMsg(Msg msg, OmmProviderEvent event){}
}
```

5.2.3 Interactive Provider Example: OmmProviderClient

```
class AppClient implements OmmProviderClient
{
    public long itemHandle = 0;
    public String OrderNr="100";

    public void onReqMsg(ReqMsg reqMsg, OmmProviderEvent event)
    {
        switch (reqMsg.domainType())
        {
            case EmaRdm.MMT_LOGIN :
```

```
                processLoginRequest(reqMsg, event);
                break;
            case EmaRdm.MMT_MARKET_BY_ORDER :
                processMarketByOrderRequest(reqMsg, event);
                break;
            default :
                processInvalidItemRequest(reqMsg, event);
                break;
        }
    }

    public void onRefreshMsg(RefreshMsg refreshMsg,OmmProviderEvent event){}
    public void onStatusMsg(StatusMsg statusMsg, OmmProviderEvent event){}
    public void onGenericMsg(GenericMsg genericMsg, OmmProviderEvent event){}
    public void onPostMsg(PostMsg postMsg, OmmProviderEvent event){}
    public void onReissue(ReqMsg reqMsg, OmmProviderEvent event){}
    public void onClose(ReqMsg reqMsg, OmmProviderEvent event){}
    public void onAllMsg(Msg msg, OmmProviderEvent event){}

    void processLoginRequest(ReqMsg reqMsg, OmmProviderEvent event)
    {
        event.provider().submit( EmaFactory.createRefreshMsg().domainType(EmaRdm.MMT_LOGIN).
                name(reqMsg.name()).nameType(EmaRdm.USER_NAME).complete(true).solicited(true).
                state(OmmState.StreamState.OPEN, OmmState.DataState.OK, OmmState.StatusCode.NONE,
                "Login accepted"), event.handle() );
    }

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

        FieldList mapSummaryData = EmaFactory.createFieldList();
        mapSummaryData.add(EmaFactory.createFieldEntry().enumValue(15,  840));
        mapSummaryData.add(EmaFactory.createFieldEntry().enumValue(53,  1));
        mapSummaryData.add(EmaFactory.createFieldEntry().enumValue(3423,  1));
        mapSummaryData.add(EmaFactory.createFieldEntry().enumValue(1709,  2));

        FieldList entryData = EmaFactory.createFieldList();
        entryData.add(EmaFactory.createFieldEntry().realFromDouble(3427, 7.76,
                MagnitudeType.EXPONENT_NEG_2));
        entryData.add(EmaFactory.createFieldEntry().realFromDouble(3429, 9600));
        entryData.add(EmaFactory.createFieldEntry().enumValue(3428, 2));
        entryData.add(EmaFactory.createFieldEntry().rmtes(212, ByteBuffer.wrap("Market
                Maker".getBytes())));

        Map map = EmaFactory.createMap();
```

```
        map.summaryData(mapSummaryData);

        map.add(EmaFactory.createMapEntry().keyAscii(OrderNr, MapEntry.MapAction.ADD, entryData));

        event.provider().submit( EmaFactory.createRefreshMsg().
                domainType(EmaRdm.MMT_MARKET_BY_ORDER).name(reqMsg.name()).
                serviceName(reqMsg.serviceName()).solicited(true).
                state(OmmState.StreamState.OPEN, OmmState.DataState.OK, OmmState.StatusCode.NONE,
                "Refresh Completed").payload(map).complete(true), event.handle() );

        itemHandle = event.handle();
    }

    void processInvalidItemRequest(ReqMsg reqMsg, OmmProviderEvent event)
    {
        event.provider().submit( EmaFactory.createStatusMsg().name(reqMsg.name()).
                serviceName(reqMsg.serviceName()).state(OmmState.StreamState.CLOSED,
                OmmState.DataState.SUSPECT,OmmState.StatusCode.NOT_FOUND, "Item not found"),
                event.handle() );

    }
}
```

5.3 OmmNiProviderConfig and OmmIProviderConfig Classes

In the following section, 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**.

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 *EMA Configuration Guide*.

Chapter 6 Troubleshooting and Debugging

6.1 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.

6.2 OMM Error Client Classes

6.2.1 OmmConsumerErrorClient and OmmProviderErrorClient Descriptions

EMA has two Error Client classes: **`OmmConsumerErrorClient`** and **`OmmProviderErrorClient`**. These two classes are an alternate error notification mechanism in the EMA, which you can use instead of the default error notification mechanism (i.e., **`OmmException`**, for details, refer to Section 6.3). 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`**.

6.2.2 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);
    }
}
```

6.3 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`** or **`OmmProvider`** 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.

THOMSON REUTERS

Document ID: EMAJ304UM.160
Date of issue: 9 December 2016