ELEKTRON MESSAGE API V3.0

DEVELOPERS GUIDE

C++ EDITION

Document ID: XXXXX
Document Version: 1.0
Date of Issue: 30 April 2015

THOMSON REUTERS

Contents

Tables

FIGURES

No table of figures entries found.

Chapter 1 Introduction

1.1 Product Description

The Elektron Message API, or EMA, is a data neutral, multi-threaded API providing access to OMM / 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 represented by the Elektron Transport API, or ETA (also known as the UPA).

The EMA provides a set of interfaces and features intended to aid in message level application development. These interfaces simplify information setting in and getting from OMM containers and messages. A set of few other interfaces abstracts behavior of a consumer type application.

The EMA enables applications to source market data from and to provide it to different components supporting OMM and RWF (e.g. Elektron, Enterprise Platform, ATS, RDF-D, etc).

Besides the ease of use and intuitiveness of the EMA interfaces, the EMA strives to leave a minimal code footprint in the 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. Provided with the EMA, training applications constitute a self learning environment and demonstrate basic yet still functional examples of the EMA applications.

1.2 Audience

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

1.3 Programming Language

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

1.4 Document Conventions

- Classes, methods, in-line code snippets are shown in `orange, Lucida Console` font
- Parameters, filenames, tools, utilities and directories are shown in **Bold** font
- Document titles and variables are shown in *italics* font
- Longer code samples are shown in **`Lucida Console`** font against an orange background

1.5 Using This Document

The material presented in this guide is divided into the following sections:

CHAPTER	DESCRIPTION
Chapter 1	About this Manual
Chapter 2	EMA Overview and Description

Table 1: Chapter Overview

1.6 Acronyms and Abbreviations

ACRONYM	DEFINITION
ADH	Advanced Data Hub
ADS	Advanced Distribution Server
API	Application Programming Interface
ATS	Advanced Transformation Server
EMA	Elektron Message API
ETA	Elektron Transport API (previously known as UPA)
ETA VA	Elektron Transport API ValueAdded Components
OMM	Open Message Model
RDM	Reuters Data Model
RSSL	Reuters Source Sink Library
RWF	Reuters Wire Format
SDK	Software Development Kit
UPA	Ultra Performance API, previously known as RSSL
UPA VA	Ultra Performance API ValueAdded Components

Table 2: Acronyms and Abbreviations

1.7 References

[1] *API Concept Guide*

[2] *EMA C++ Config Guide*

[3] *EMA C++ Reference Manual*

[4] *ETA C Developers Guide*

[5] *ETA C ValueAdded Components Developers Guide*

[6] *RDM Usage Guide*

1.8 Documentation Feedback

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

- Send us your comments via email at apidocumentation@thomsonreuters.com.
- Mark up the PDF using the Comment feature in Adobe Reader. After adding your comments, you can submit the entire PDF to Thomson Reuters by clicking **Send File** in the **File** menu. Use the apidocumentation@thomsonreuters.com address.

Chapter 2 Product Description and Overview

2.1 Product Overview

The EMA is considered an ease of use interface positioned at the message level of the API stack made available in the Elektron SDK. It is intended to present applications with a simple access to OMM messages and containers while providing all necessary transport level functionalities. In general, EMA applications are concerned with and focused on processing market data items, e.g. opening and receiving item data or providing item data. EMA abstracts and hides all the transport level functionality minimizing application involvement to just optional transport level configuration and server address specification.

EMA provides simple set and get type of functionality to populate and read OMM containers and messages. EMA takes advantage of the fluent interface design which allows users to set disparate values of the same message or container by stringing respective interface methods together one after the other. Fluent interfaces provide means for visual code simplification which help understanding and debugging of applications.

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

2.2 Product Architecture

The EMA incorporates the ETA ValueAdded Reactor component which provides the watchlist and transport level functionality. The EMA wraps up the ETA's VA Reactor component in its own class of `OmmConsumer`. This class provides interfaces to open, modify and close market items or instruments, as well as to submit post messages and generic messages. To complete the set of consumer application functionalities, the `OmmConsumer` class provides the `dispatch()` method. Depending on the application design and configuration, application may need to call this method to dispatch received messages. Configuration of the ETA VA Reactor and `OmmConsumer` is done by the `OmmConsumerConfig` class.

The `OmmConsumerClient` class provides the call back mechanism for EMA to deliver received messages to application. Application needs to implement a class inheriting from the `OmmConsumerClient` class to receive and process the messages. The `OmmConsumerClient` call back methods may be executed on the application or EMA thread of control. By default, the `OmmConsumerClient` call back methods are executed on the EMA thread of control. Using the `OmmConsumerConfig::operationModel()` interface, applications may modify this default behavior. If done so, application needs to call `OmmConsumer::dispatch()` method to dispatch received messages.

The `OmmConsumerErrorClient` class provides an alternate reporting mechanism for error conditions detected by the `OmmConsumer` class. By default, the `OmmConsumer` class throws `OmmException` to report a detected error condition. Passing the `OmmConsumerErrorClient` on the constructor of the `OmmConsumer` class, switches the error reporting from exception throwing into call backs.

In addition to error reporting mechanism, EMA provides the logger mechanism useful while monitoring EMA behavior and or debugging any issues.

The EMA utilizes the ETA decoding and encoding functions for reading and population of OMM containers and messages. Each and every OMM container and message is represented by a respective EMA interface class. These classes provide relevant methods for setting information on these containers and messages as well as getting it from them. All classes representing OMM containers, messages and primitives do inherit from a common parent class of `Data`. This inheritance makes sure that all these classes provide same basic common ease of use functionality applications may expect from them, e.g. `toString()` for simple printing of the contained data.

Depending on the configuration, the EMA may have one or two own threads. One thread, which is always there, is implemented by the ETA VA Reactor. This thread runs the internal ETA VA Reactor's logic. Please see the [5] for details on this thread. If the `OmmConsumerConfig` operation model is set to the `OmmConsumerConfig::ApiDispatchEnum`, the received messages are dispatched by the second internal EMA thread. This second thread does not run, if the `OmmConsumerConfig` operation model is set to the `OmmConsumerConfig::UserDispatch`. In this case, application is responsible for calling the `Ommconsumer::dispatch()` method to dispatch all received messages. In this case it is recommended that application does not put off message dispatching since this would result in the slow consumer behavior.

2.3 Supported Features

FEATURE	DESCRIPTION
Default Admin Domain Requests	EMA uses default login, directory and dictionary request while connecting to server. Login request uses current user's name and defaults all the other login attributes. Directory request calls for all the services and filters. Default RDM dictionaries are requested from the very first service up and accepting requests.
Configurable Admin Domain Requests	EMA provides means for modifying the default admin domain requests.
Batch Request	Application may use a single request message to specify interest in multiple items via the item list
Dynamic View	Application may specify a subset of fields or elements of a particular item
Optimized Pause and Resume	Application may request server to pause and resume item stream
Single Open	EMA supports application selected single open functionality
Connection Redirection	Also known as Load balancing; this feature enables dynamic and balanced provider discovery based on the information received from the provider at login.
RMTES Decoder	EMA provides a built in RMTES decoder. IF desired, application may cache `RmtesBuffer` objects and apply all the received changes to them.
`Data::toString()`	All OMM containers, primitives and messages may simply be printed out to screen in a standardized output format. This is called "stringification".s
`Data::getAsHex()`	Applications may obtain binary representations of all OMM containers, primitives and messages.
Programmatic Config	Enables application to programmatically specify and overwrite EMA configuration
File Config	Enables applications to specify EMA configuration in an EmaConfig.xml file

Table 3: Supported Features

2.4 Learning EMA

Playing and experimenting with the EMA library is the suggested way of learning its usage. To facilitate the learning experience, the EMA package provides a set of the so called training examples. The purpose of these examples is to showcase the usage of the EMA interfaces in the increasing level of complexity and sophistication. The level of example sophistication is reflected in the example series number. The 100 series examples simply open an item and print its received content to the screen using the `Data::toString()` method, which is called the "stringification". The applications in this series present the EMA support for the stringification of messages, containers and primitives. Though useful for learning, debugging and writing of display applications, the stringification is not sufficient for development of more sophisticated applications. The 200 series examples do present extraction of information from OMM containers and messages in native data formats, e.g. `UInt64`, `EmaString`, and `EmaBuffer`. The 300 and 400 series examples depict usage of particular EMA features like posting, generic message, programmatic config and alike.

While coding and debugging applications, developers are encouraged to refer to [3] and or to the development supporting features provided by their IDE of choice, e.g. IntelliSense.

Note: To be most effective on their jobs, the EMA application developers should become familiar with OMM and Market Data distribution systems prior to learning the EMA.

2.5 OMM Containers and Messages

The EMA supports a full set of OMM containers, messages and primitives (e.g. `FieldList`, `Map`, `RefreshMsg`, `Int`). To simplify their usage, EMA adopted the "set / add" type of functionality to populate them and the "get" type of functionality to read them and extract data from them. The set functionality is used for specification of variables occurring once in an OMM container or message. The add functionality is used for population of entries in the OMM containers. The set & add type methods do return a reference to the modified object which enables fluid interface usage.

EMA uses a simple iterative approach to extract entries from the OMM containers, one at a time. Applications iterate over every OMM container type in the same way. While iterating, application may apply a filtering mechanism to just return entries with sought identification. For example, while iterating over a `FieldList`, application may specify field id or field name that it is interested in. If specified, entries with not matching identification will be skipped.

Individual container entries are extracted during the iteration. Depending on the container type, the entry may contain own identity, e.g. field id; action to be applied to the received data, e.g. add action; permission information associated with the received data; and entry's load and its data type. The entry's load may be extracted using the ease of use interfaces returning references to the contained objects whose reference type is based on the load's data type, and an interface returning a reference to the base `Data` class. This latter interface enables more advanced applications to use down-cast operation if such is desired. Please see the 2.5.6 for details on usage of the ease of use interfaces as well as the down-cast operation.

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

- All classes representing primitive / intrinsic data types directly inherit from the `Data` class; e.g. `OmmInt`, `OmmBuffer`, `OmmRmtes`, etc.
- `OmmArray` class inherits directly from the `Data` class. The `OmmArray` is treated as a primitive rather than container since 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, do 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 do inherit from the Msg class which in turn inherits from the ComplexType class; e.g. RefreshMsg, GenericMsg, or PostMsg.

2.5.1 DataType Class

The DataType class provides the set of enumeration values representing each and every supported OMM data type; this includes all OMM containers, messages and primitives. Each class representing OMM data identifies itself with an appropriate DataType enumeration value, e.g. DataType::FieldListEnum, DataType::RefreshMsgEnum. The Data::getDataType() method may be used to learn the data type of a given object.

The list of the enumeration values in the DataType class contains two special enumeration values. They are: DataType::ErrorEnum and DataType::NoDataEnum. These values may only be received during reading or extracting information from OMM containers or messages. If received, the DataType::ErrorEnum indicates that an error condition was detected. Please refer to 2.5.5 for more details. The DataType::NoDataEnum signifies lack of data on the summary of a container, or message payload or attribute.

2.5.2 DataCode Class

The DataCode class provides the set of two enumeration values indicating state of the data. The DataCode::NoCodeEnum indicates that the received data is valid and application may use it. The DataCode::BlankEnum indicates that the data is not present and application needs to blank the respective data fields.

2.5.3 Data Class

The Data class is a parent abstract class from whom all the OMM containers, messages and primitives inherit. This class provides interfaces common across all its children. The common inheritance across all OMM containers, messages and primitives enables down-casting operations. It is worth noticing that even though all primitive data types are represented by classes inheriting from the Data class, none of the ease of use interfaces returns such references. All the primitive data types are always returned by their intrinsic representation.

Note: The Data class and all the classes inheriting from it are built for ease of access to information and are optimized for efficiency. They are designed as temporary and rather short living objects. Therefore they should not be used as storage or caching devices.

Note: Getting information from the “just set” Omm containers or messages is not supported.

2.5.4 Msg Class

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

2.5.5 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 getAsHex() method to obtain binary data associated with the detected error condition. This class

sole purpose is to aid debugging efforts. The following code snippet presents usage of the OmmError class while processing ElementList.

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

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

2.5.6 Working with OMM Containers

EMA supports the following OMM containers: ElementList, FieldList, FilterList, Map, Series and Vector. As appropriate, each of these classes provide the set type interfaces for the container header information, e.g. dictionary id, element list number, and the add type interfaces for addition of entries. Setting of the container header and optional summary must happen prior to adding the very first entry.

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

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

The following code snippet presents population of FieldList class depicting usage of fluid interfaces.

```
try {
  FieldList fieldList;

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

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

The following code snippet presents population of Map class with summary and a single entry containing a FieldList. In this case, FieldList class uses own memory buffer to store its own content while it is populated. This buffer later gets copied to the buffer owned by the Map class.

```
try {
  FieldList fieldList;

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

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

The following code snippet presents population of Map class with a single entry containing a FieldList. In this case, FieldList class uses memory buffer owned by Map class to store its own content while it is populated. Therefore the internal buffer copy incurred in the previous scenario is avoided.

```
try {
  FieldList fieldList;

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

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

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

In the following code snippet application extracts information from FieldList class. The FieldList::forth() method is used to iterate over the FieldList class. In this case information about all entries will be extracted.

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

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

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

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

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

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

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

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

```
    case DataType::IntEnum :
      Int64 value = fieldEntry.getInt();
      break;
    }
  }
}
```

The following code snippet shows extracting information from a FieldList object using the down-cast operation.

2.5.7 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 type interfaces for the message header, permission, key, attribute and payload information.

The following code snippet presents population of `GenericMsg` with payload consisting of `ElementList`.

```
try {
  GenericMsg genMsg;

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

The following example presents extraction of information from `GenericMsg` class.

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

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

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

2.6 OmmConsumer Class

The OmmConsumer class is a 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, modify items, as well as to submit messages to the connected server, both PostMsg and GenericMsg.

2.6.1 Working with OmmConsumer

The following are the steps application makes to connect to a server and open items:

- Optionally specify configuration using the **EmaConfig.xml** file
    - EMA provides default configuration which should be sufficient in simple app cases
- Create OmmConsumerConfig object
- Optionally specify and or modify configuration using methods on the OmmConsumerConfig class
    - If **EmaConfig.xml** file is not used, then at a minimum applications may need to modify the default host address and port
- Implement an application callback client class inheriting from the OmmConsumerClient class
    - Application needs to override default implementation of the call back methods and provide their own business logic. Not all the methods need to be overridden; only the ones that are required for the application business logic
- Optionally implement an application error client class inheriting from the OmmConsumerErrorClient class
    - Application needs to override default error call back methods to be effectively notified about detected error conditions
- Create OmmConsumer object and pass OmmConsumerConfig object and if needed the application error client object to it
- Open items of interest using OmmConsumer::registerClient() method
- Process received messages
- Optionally submit PostMsg, GenericMsg, modify and or close items using respective methods of the OmmConsumer class
- Exit when done.

To allow applications simply to open items right after the OmmConsumer object is created, the EMA performs the following steps during the creation and initialization process of the OmmConsumer object:

- Create internal item watchlist
- Establish connectivity to a configured server / host
- Log into the server and obtain source directory information
- If configured, obtain dictionaries

Destruction of the OmmConsumer object causes log out and disconnect from the connected server. All the items are implicitly closed at that time too.

The following code snippet presents the simplest application depicting some of the above steps.

```
try {
  AppClient client;
```

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

2.6.2 Working with Items

Items or instruments open in EMA are uniquely identified by a numeric value (e.g. UInt64). This value, also known as a handle, is assigned by the EMA and returned by the `OmmConsumer::registerClient()` call. These handles are valid as long as the associated items stay open. Holding onto these handles is important only to the applications that want to modify or close particular items, or use the item streams for submission of `PostMsg` or `GenericMsg` to the connected server. Simple applications that just open and watch several items till they exit do not need to store item handles.

While opening an item, on the call to the `OmmConsumer::registerClient()` method, application may 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 call back method.

The following code snippet shows usage of item handle while modifying item's priority and posting its modified content.

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

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

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

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

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

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

2.7 OmmConsumerClient Class

The OmmConsumerClient class provides a callback mechanism through which applications receive OMM messages on items they subscribed for. The OmmConsumerClient is actually a parent class implementing empty default call back methods. Applications need to implement their own class inheriting from the OmmConsumerClient class and override the methods they are interested in processing. Applications may implemented 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 processing of RefreshMsg, UpdateMsg, StatusMsg, AckMsg and GenericMsg. These messages are processed in their respectively named methods of onRefreshMsg(), onUpdateMsg(), onStatusMsg(), onAckMsg() and onGenericMsg(). Applications only need to override methods for messages they are interested in processing.

The following code snippet presents design of an application client type class depicting details of onRefreshMsg() method and its implementation.

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


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


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


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


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


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

2.8 OmmConsumerConfig Class

The OmmConsumerConfig class is used to customize functionality of the OmmConsumer class. The default behavior of the OmmConsumer is hardcoded in the OmmConsumerConfig class. The customization or rather configuration of the OmmConsumer may be done in the following ways:

- Using **EmaConfig.xml** file,
- Using interface methods on the OmmConsumerConfig class,

- Passing OMM formatted configuration data through the OmmConsumerConfig::config(const Data&) method.

For more details on usage of the OmmConsumerConfig class and configuration parameters please refer to [2].

2.9 OmmConsumerErrorClient Class

The OmmConsumerErrorClient class is an alternate error notification mechanism in the EMA. This is an alternative to the OmmConsumer's default error notification mechanism of OmmException. Both mechanisms deliver the same information and detect the same error conditions. To use the OmmConsumerErrorClient, applications need to implement their own error client class and override default implementation of each method.

The following code snippet presents design of an application error client and depicts simple processing of onInvalidHandle() method.

```
class AppErrorclient : public OmmConsumerErrorClient
{
public :

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

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

  void onMemoryExhaustion( const EmaString& text);

  void onInvalidUsage( const EmaString& text);

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

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

2.10 OmmException Class

If an error condition is detected, EMA may throw an exception. All the exceptions in the EMA inherit from the parent class of OmmException. This class provides functionality and methods common across all OmmException types. The following exception types are supported in EMA:

- OmmInaccessibleLogFileException – thrown if EMA is unable to open a log file for writing
- OmmInvalidConfigurationException – thrown if an unrecoverable configuration error is detected
- OmmInvalidHandleException – thrown if an invalid / unrecognized item handle is passed in on OmmConsumer class methods
- OmmInvalidUsageException – thrown if an invalid interface usage is detected

- `OmmMemoryExhaustionException` – thrown if an out of memory condition is detected
- `OmmOutOfRangeException` – thrown if a passed in parameter is out of valid / supported range
- `OmmSystemException` – thrown if a system exception is detected
- `OmmUnsuppoprtedDomainTypeException` – thrown if domain type specified on a message is not supported

Thomson Reuters recommends that applications do use try / catch blocks especially during their own development and QA cycle to be able to quickly detect and fix any EMA usage or application design errors.

2.11 EMA Logger Usage

The EMA provides a logging mechanism useful for debugging of run time issues. Depending on the configuration, the EMA logs significant events encountered during the run time. The logging output may be directed to a file (a default setting) or to the stdout. Additionally applications may configure the EMA logger mechanism to log every event or just an error event or nothing. Please refer to [2] for more details.