Fiorano
Enabling change at the speed of thought

www.fiorano.com

FioranoMQ

C++ RTL Native Guide

AMERICA'S
Fiorano Software, Inc.
230 S. California Avenue,
Suite 103, Palo Alto,
CA 94306 USA
Tel: +1 650 326 1136
Fax: +1 646 607 5875
Toll-Free: +1 800 663 3621
Email: info@fiorano.com

EMEA
Fiorano Software Ltd.
3000 Hillswood Drive Hillswood
Business Park Chertsey Surrey
KT16 0RS UK
Tel: +44 (0) 1932 895005
Fax: +44 (0) 1932 325413
Email: info_uk@fiorano.com

APAC
Fiorano Software Pte. Ltd.
Level 42, Suntec Tower Three 8
Temasek Boulevard 038988
Singapore
Tel: +65 68292234
Fax: +65 68292235
Email: info_asiapac@fiorano.com

Fiorano

Entire contents © Fiorano Software and Affiliates. All rights reserved. Reproduction of this document in any form without prior written permission is forbidden. The information contained herein has been obtained from sources believed to be reliable. Fiorano disclaims all warranties as to the accuracy, completeness or adequacy of such information. Fiorano shall have no liability for errors, omissions or inadequacies in the information contained herein or for interpretations thereof. The opinions expressed herein are subject to change without prior notice.

Copyright (c) 1999-2007, Fiorano Software Private Limited. and Affiliates

Copyright (c) 2008-2013, Fiorano Software Pte Ltd. and Affiliates

All rights reserved.

This software is the confidential and proprietary information of Fiorano Software ("Confidential Information"). You shall not disclose such ("Confidential Information") and shall use it only in accordance with the terms of the license agreement enclosed with this product or entered into with Fiorano.

Content

Chapter 1: Introduction 12

Messaging Domains 12
- Connection Factory 12
- Connection 13
- Destination 13
- Session 13
- JMS Message 13
- Message Producer 13
- Message Consumer 14

Chapter 2: Datatypes and Constants 15

Basic Data Types and their Sizes 15
C++RTL Constants 15
Naming convention 16

Chapter 3: Writing Applications in C++ 17

Namespaces in C++ 18
Error Handling 19
Message listeners in C++ 21
Connection Start and Stop 23
Connection Close 23
Exception listeners in C++ 24
Advisory Message listeners in C++ 26
Lookup of Administered Objects 28
Object Deletion 29
Getting Message Properties 30

Chapter 4: FioranoMQ C++ RTL – Classes 32

CLookupHelper 32
- Inheritance Hierarchy 32
- Subclasses 32
- Constructors 32
- Methods 32

CInitialContext 33
- Inheritance Hierarchy 33
- Subclasses 33

Constructors 33
Methods 33
CAdvisoryMessage 34
Inheritance Hierarchy 34
Constructors 34
Subclasses 35
Methods 35
CAdvisoryMsgListener 36
Inheritance Hierarchy 36
Subclasses 37
Methods 37
CFioranoException 37
Inheritance Hierarchy 37
Subclasses 37
Constructors 37
Methods 38
CJMSException 39
Inheritance Hierarchy 39
Subclasses 39
Constructors 39
Methods 40
CExceptionListener 41
Inheritance Hierarchy 41
Subclasses 41
Constructors 41
Methods 41
CMessageListener 41
Inheritance Hierarchy 41
Subclasses 41
Constructors 41
Methods 42
CAdminConnectionFactory 42
Inheritance Hierarchy 42
Subclasses 42
Constructors 42
Methods 42
CConnectionFactory 43
Inheritance Hierarchy 43
Subclasses 43
Methods 44
CFioranoConnectionFactory 44
Inheritance Hierarchy 44
Subclasses 44
Constructors 45
Methods 45
CQueueConnectionFactory 46

Inheritance Hierarchy 46
Subclasses 46
Constructors 46
Methods 47
Inherited Methods 48
CTopicConnectionFactory 48
Inheritance Hierarchy 48
Subclasses 48
Constructors 48
Methods 49
Inherited Methods 50
CAdminConnection 50
Inheritance Hierarchy 50
Subclasses 50
Constructors 50
Methods 50
CConnection 51
Inheritance Hierarchy 51
Subclasses 51
Methods 51
CFioranoConnection 54
Inheritance Hierarchy 54
Subclasses 54
Constructors 54
Methods 54
CQueueConnection 56
Inheritance Hierarchy 56
Subclasses 56
Constructors 56
Methods 57
CTopicConnection 58
Inheritance Hierarchy 58
Subclasses 58
Constructors 58
Methods 58
CJMSSession 59
Inheritance Hierarchy 59
Subclasses 59
Methods 59
CFioranoSession 65
CQueueSession 66
CTopicSession 67
CMessageProducer 68
Inheritance Hierarchy 69
Subclasses 69
Methods 69

CFioranoMessageProducer 73
Inheritance Hierarchy 73
Subclasses 73
Constructors 73
CQueueSender 74
Inheritance Hierarchy 74
Subclasses 74
Constructors 74
Methods 74
CTopicPublisher 75
Inheritance Hierarchy 75
Subclasses 75
Constructors 75
Methods 75
CQueueRequestor 76
Inheritance Hierarchy 76
Subclasses 76
Constructor 76
Methods 76
CTopicRequestor 77
Inheritance Hierarchy 77
Subclasses 77
Constructors 77
Methods 77
CMessageConsumer 78
Inheritance Hierarchy 78
Subclasses 78
Methods 79
CFioranoMessageConsumer 81
Inheritance Hierarchy 81
Subclasses 81
Constructors 81
Methods 81
CQueueReceiver 82
Inheritance Hierarchy 82
Subclasses 82
Constructors 82
Methods 82
CTopicSubscriber 83
Inheritance Hierarchy 83
Subclasses 83
Constructors 83
Methods 83
CTopicMetaData 84
Inheritance Hierarchy 84
Subclasses 84

Constructors 84
Methods 84
CQueueMetaData 85
Inheritance Hierarchy 85
Subclasses 86
Constructors 86
Methods 86
CDestination 87
Inheritance Hierarchy 87
Subclasses 87
Constructors 87
Methods 87
CQueue 88
Inheritance Hierarchy 88
Subclasses 88
Constructors 88
Methods 89
CTopic 89
Inheritance Hierarchy 89
Subclasses 90
Constructors 90
Methods 90
CTemporaryQueue 91
Inheritance Hierarchy 91
Subclasses 91
Constructors 91
Methods 91
CTemporaryTopic 92
Inheritance Hierarchy 92
Subclasses 92
Constructors 92
Methods 92
CProperty 93
Inheritance Hierarchy 93
Subclasses 93
Constructors 93
Methods 93
CMessage 94
Inheritance Hierarchy 94
Subclasses 94
Constructors 94
Methods 96
CTextMessage 110
Inheritance Hierarchy 110
Subclasses 110
Constructors 110

Methods 110
CByteMessage 111
Inheritance Hierarchy 111
Subclasses 111
Constructors 111
Methods 111
CMapMessage 118
Inheritance Hierarchy 118
Subclasses 118
Constructors 118
Methods 118
CStreamMessage 125
Inheritance Hierarchy 125
Subclasses 125
Constructors 125
Methods 125
CMQAdminService 131
Inheritance Hierarchy 131
Subclasses 131
Constructors 131
Methods 131
CHashTable 139
Inheritance Hierarchy 139
Subclasses 139
Constructors 139
Methods 139
CHashTableEnumerator 140
Inheritance Hierarchy 140
Subclasses 140
Constructors 140
Methods 141
CLogHandler 141
Inheritance Hierarchy 141
Subclasses 141
Constructors 141
Methods 142
CCSPManager 143
Inheritance Hierarchy 143
Subclasses 143
Constructors 143
Constructors 143
Methods 143
CCSPBrowser 143
Inheritance Hierarchy 143
Subclasses 144
Constructors 144

Methods 144
CCSPEnumeration 147
Inheritance Hierarchy 147
Subclasses 147
Constructors 147
Methods 147

Chapter 5: Large Message Support 148

CFioranoConnection 148
CRecoverableMessagesEnum 148
Inheritance Hierarchy 148
Subclasses 148
Constructors 148
Methods 148
CLargeMessage 149
Inheritance Hierarchy 149
Subclasses 149
Constructors 149
Methods 149
CLMStatusListener 152
Inheritance Hierarchy 152
Subclasses 153
Constructors 153
Methods 153
CLMTransferStatus 153
Inheritance Hierarchy 153
Subclasses 153
Constructors 153
Methods 153

Chapter 6: Message Compression 156

Message Compression Characteristics 156

Chapter 7: Using Sample Programs 158

Organization of Samples Provided 158
Compiling and Running the Samples 158
Operating Environments 158
Limitations of C++ RTL 159

Chapter 8: Native C++ Runtime Examples........................ 160

Platforms Supported..161
Building and Running C++ Applications 162
PTP Samples 162
Admin 162
Basic..163
Browser 163
Csp Browser 163
DeadMessageQueue..164
HTTP..164
LMS 165
Message Compression 165
PerDestination 165
PerMessage..166
MsgSel 166
Mtptp...167
NonJndi...167
Reqrep 167
Basic 167
TimedOut..168
RevalidateConnections 168
ServerlessMode...169
SSL 169
Transaction..169
PubSub Samples 170
Admin 170
Basic..170
CspBrowser 170
Dursub 171
HierarchicalTopics...171
HTTP..172
LMS 172
Message Compression 173
PerDestination 173
PerMessage..173
Msgsel 174
Mtpubsub 174
NonJndi...174
Reqrep 175
Basic 175
TimedOut..175
RevalidateConnections 175
ServerlessMode...176
SSL 176
Transaction..176

Unified Samples 177
NonJndi 177
SendReceive 177

Chapter 1: Introduction

The Native C++Runtime Library (C++RTL) allows C++ based application to interact with FioranoMQ. A C++ based client can thus seamlessly communicate with Java based Fiorano Clients.

This version of C++RTL is designed to run on both native and .NET platforms. Both the versions support secure and non-secure TCP and HTTP connections on Win32 platform for Point-to-Point and Publish/Subscribe communication models.

The C++ Runtime is designed to provide maximum conformance with the JMS specifications. All public APIs have similar signature as the corresponding java APIs specified by JMS. The classes have similar naming convention.

The Native C++Runtime Library (C++RTL) allows C++ based application to interact with FioranoMQ. A C++ based client can thus seamlessly communicate with Java based Fiorano Clients.

Messaging Domains

There are two kinds of messaging domains:

1. Point-to-Point (PTP)
2. Publish-Subscribe(Pub/Sub)

In PTP domains, messages are sent to a particular destination where they are queued. A client application delivers messages from this queue to the destination specified by the provider. Though there can be several messages in the queue, each message is intended for only one destination/receiver.

The Pub/Sub domain, on the other hand, allows a message to be distributed to more than one subscriber via the provider.

Both these domains can be deployed by FioranoMQ. In addition, FioranoMQ can handle the unified domains introduced by JMS 1.1.

The JMS common interface provides a domain-independent view of the PTP and Pub/Sub messaging domains. Following are the list of JMS concepts with brief definition:

Connection Factory

As per JMS specifications, an application uses a `Connection Factory` to fetch the details of a connection instance to connect to the Server. The connection factory instance encapsulates all the parameters (like URL, protocol, and so on) required to connect to the Server. These parameters are configured to use the default socket acceptor settings and must be modified if the Server uses a socket acceptor with a non-default configuration. The Server creates the default connection to **factories** when it is launched for the first time. These connection factories are automatically created based on the configuration of the socket acceptor being used.

Connection

A Connection object is a client's active connection to its JMS provider. A connection could represent an open TCP/IP socket between a client and a service provider domain. Connections support concurrent use. You use a connection to create one or more sessions. Connections can be created specific to PTP and PubSub messaging styles. For example, CQueueConnection class can be used for Queue connection type and CTopicConnection class for Topic connection.

Destination

A destination is the object which a client uses to specify the target of messages it produces and the source of messages it consumes. In the PTP messaging domain, destinations are called Queues, and in the pub/sub messaging domain, destinations are called Topics.

Session

A session is the single-thread context for producing and consuming messages. It can create and serve multiple Producers and Consumers.

A session can be either transacted or non-transacted. Each session supports a single series of transactions and treats them as a unit. Messages produced and consumed within a transaction become the content of that particular transaction. A commit method indicates that message processing can occur. A rollback method disables the processing of messages. In both cases, a transaction is considered to have been completed. A non-transacted session receives message in a mode specified by JMS 1.1: This mode could be one of the following modes: AUTO_ACKNOWLEDGE, CLIENT_ACKNOWLEDGE, and DUPS_OK_ACKNOWLEDGE. The DUPS_OK_ACKNOWLEDGE is used in applications where messages delivery can be duplicated.

JMS Message

The ultimate purpose of a JMS application is to produce and to consume messages that can then be used by other software applications. JMS messages have a basic format that is simple but highly flexible, allowing you to create messages that match formats used by non-JMS applications on heterogeneous platforms. A JMS message has three parts:

1. A header
2. Properties (optional)
3. A body (optional)

Message Producer

A message producer is an object created by a session and is used for sending messages to a destination. The PTP form of a message producer implements the CQueueSender interface. The pub/sub form implements the CTopicPublisher interface. You can create an unidentified producer by specifying null as the argument to createSender or createPublisher. With an unidentified producer, you can wait to specify which destination to send the message to, until you send or publish a message.

Message Consumer

A message consumer is an object created by a session and is used for receiving messages sent to a destination. A message consumer allows a JMS client to register interest in a destination with a JMS provider. The JMS provider manages the delivery of messages from a destination to the registered consumers of the destination. The PTP form of message consumer implements the CQueueReceiver class. The pub/sub form implements the CTopicSubscriber class.

The following diagram illustrates the flow of a JMS client.

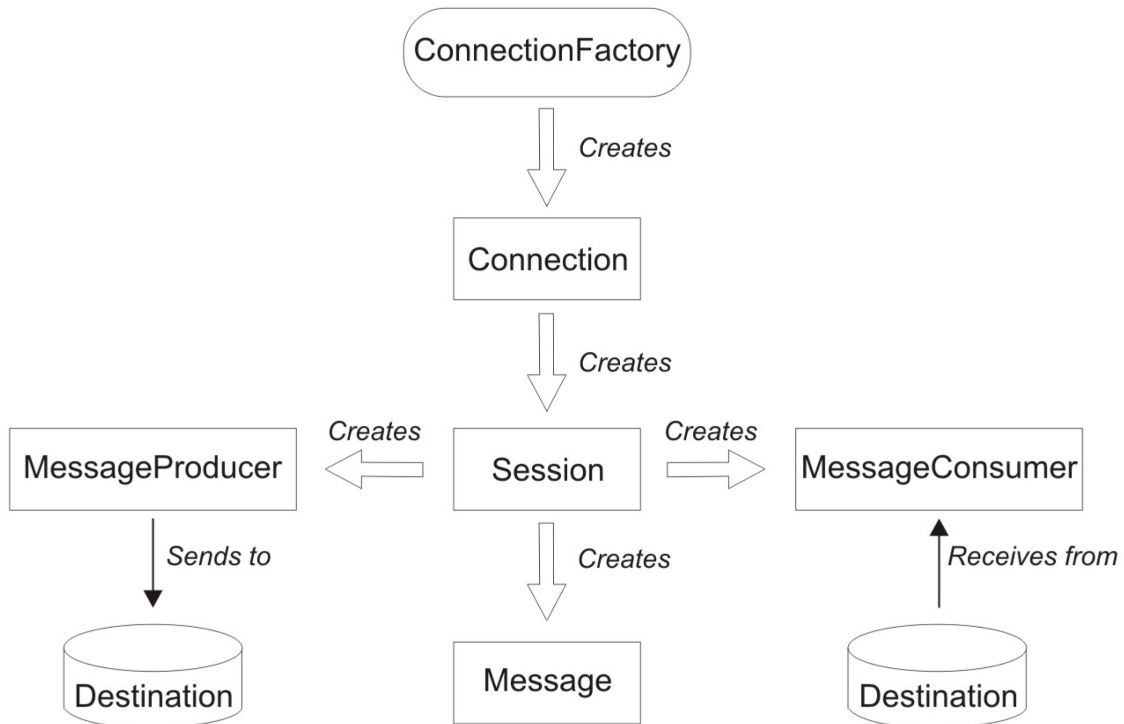


Figure: JMS client flow

Chapter 2: Datatypes and Constants

This chapter contains an overview of the C++RTL specific data types and constants. A brief explanation of each data type along with sizes is provided.

Basic Data Types and their Sizes

This section lists the basic data types used in C++RTL APIs, indicating the size of each.

- `mqbyte` - Data defined to occupy 8 bits (unsigned).
- `mqchar` - Data defined to occupy 8 bits.
- `mqshort` - Data defined to occupy 16 bits.
- `mqint` - Data defined to occupy 32 bits.
- `mqlong` - Data defined to occupy 64 bits.
- `mqfloat` - Data defined to occupy 32 bits.
- `mqdouble` - Data defined to occupy 64 bits.

C++RTL Constants

All required public constants are categorized and declared in `common_def.h` according to JMS specifications.

Messaging related constants are required while creating sessions and sending messages to a queue or topic on the FioranoMQ Server.

Following are the possible values of the acknowledgement mode that can be used while creating a TopicSession or a QueueSession:

- #define AUTO_ACKNOWLEDGE 1
- #define CLIENT_ACKNOWLEDGE 2
- #define DUPS_OK_ACKNOWLEDGE 3

Following are the permissible values for the message delivery mode that can be used in publish or send call. The persistence of a message is decided using these constants:

- #define NON_PERSISTENT 1
- #define PERSISTENT 2

Following are the string representations of the various message types that can be sent by a C client to the FioranoMQ Server. A call of Msg_getJMSType returns one of these values:

- #define JMSMESSAGE “Message"
- #define BYTESMESSAGE "BytesMessage"
- #define STREAMMESSAGE "StreamMessage"
- #define MAPMESSAGE "MapMessage"
- #define TEXTMESSAGE "TextMessage"

Following are some of the permissible values for message priority that can be used in publish or send call:

- #define MinPriority 0
- #define MaxPriority 9
- #define LowPriority 0
- #define NormPriority 4
- #define HighPriority 9

Naming convention

The C++RTL adheres to the following naming convention for you to easily identify the classes, constants and member functions with the corresponding definitions in JMS specifications.

Type	**Naming Convention**	**Example**
Class	C<JMSClass name>	`CTopicPublisher(JMS:TopicPub-lisher)`
Function	As defined in JMS Specifications	`publish`

Chapter 3: Writing Applications in C++

This chapter provides brief information on getting started with writing C++ applications to connect with FioranoMQ Server. Following sections detail on the C++ RTL namespace, error handling, handling exception listeners, message listeners, and advisory message listeners.

Destinations

A `destination` is the object a client uses to specify the target of messages it produces and the source of messages it consumes. In the PTP messaging domain, destinations are called Queues. In the pub/sub messaging domain, destinations are called Topics.

Temporary Destinations

An application typically uses a `Temporary Destination` to receive replies to request messages. To specify the destination where a reply to a request message is to be sent, an application calls the `Set JMSReplyTo` method of the Message object representing the request message. The destination specified on the call can be a temporary destination.

To create a temporary queue, a C application calls the `TS_createTemporaryQueue(QueueSession)` function and `TS_createTemporaryTopic(TopicSession)` for creating a Topic, with session as the parameter. In a C++ application, a temporary queue is created by calling `createTemporaryQueue` method of Session object and `createTemporaryTopic` for a Topic destination.

Message Producers

A `message producer` is an object created by a session and is used for sending messages to a destination. The PTP form of a message producer uses the `CQueueSender` class. The pub/sub form uses the `CTopicPublisher` class.

You can create an unidentified producer by specifying NULL as the argument to `createSender` or `createPublisher`. With an unidentified producer, you can wait to specify which destination to send the message to until you send or publish a message. If you created an unidentified producer, use the overloaded send or publish method that specifies the destination as the first parameter.

Message Consumers

A `message consumer` is an object created by a session and is used for receiving messages sent to a destination. A message consumer allows a JMS client to register interest in a destination with a JMS provider. The JMS provider manages the delivery of messages from a destination to the registered consumers of the destination. The PTP form of message consumer uses the `CQueueReceiver` class. The pub/sub form uses the `CTopicSubscriber` class.

When using either the `CQueueReceiver` or the `CTopicSubscriber`, you can use the receive method to consume a message synchronously. Use this method at any time after you call the start method.

For consuming the messages asynchronously, message listeners are provided.

Namespaces in C++

All the C++ classes are declared in a namespace called *cppnativertl*. A C++ application can therefore adopt one of the following approaches when referring to the names of *cppnativertl* classes:

The application can qualify the names of *cppnativertl* classes with the name of the namespace, *cppnativertl*, as show in the following C++ code fragment:

```
#include <cppHeaders.h>
#include <iostream>
using namespace std;
int main(int argc,char* argv[])
{
    cppnativertl :: CHashTable *m_env;
    cppnativertl :: CInitialContext *m_ic;
    // Queue Connection Factory
    cppnativertl :: CQueueConnectionFactory *m_qcf;
    // Queue Connection
    cppnativertl :: CQueueConnection *m_qc;

    // Set up the environment variables
    m_env = new CHashTable();
    m_env->Put(SECURITY_PRINCIPAL, "anonymous");
    m_env->Put(SECURITY_CREDENTIALS,"anonymous");
    m_env->Put(CONNECT_URL,"http://localhost:1856");
    m_env->Put(BACKUP_URLS,"http://localhost:1956");
    m_env->Put(PROVIDER_URL,"http://localhost:1856");

    m_ic = new CInitialContext(m_env);

    m_qcf = dynamic_cast<CQueueConnectionFactory *>(m_ic->Lookup("primaryQCF"));
    if(m_qcf == 0)
    {
        throw new CJMSException("Connection factory lookup failed.Error in Type casting.");
    }

    m_qc = m_qcf->createQueueConnection("anonymous", "anonymous");

    //Other code here
    //Deletion of objects
    return(0);
}
```

The application can use a `using` directive to make the names of `cppnativertl` classes available without having to qualify them. For example:

```
#include <cppHeaders.h>
#include <iostream>
using namespace std;
// Using cpp native rtl namespace
using namespace cppnativertl;
int main(int argc,char* argv[])
{
    CHashTable *m_env;
```

```
    CInitialContext *m_ic;
    // Queue Connection Factory
    CQueueConnectionFactory *m_qcf;
    // Queue Connection
    CQueueConnection *m_qc;

    // Set up the environment variables
    m_env = new CHashTable();

    m_env->Put(SECURITY_PRINCIPAL, "anonymous");
    m_env->Put(SECURITY_CREDENTIALS,"anonymous");
    m_env->Put(CONNECT_URL,"http://localhost:1856");
    m_env->Put(BACKUP_URLS,"http://localhost:1956");
    m_env->Put(PROVIDER_URL,"http://localhost:1856");

    m_ic = new CInitialContext(m_env);

    m_qcf = dynamic_cast<CQueueConnectionFactory *>(m_ic->Lookup("primaryQCF"));
    if(m_qcf == 0)
    {
        throw new CJMSException("Connection factory lookup failed.Error in Type casting.");
    }

    m_qc = m_qcf->createQueueConnection("anonymous", "anonymous");
    //Other code here
    //Deletion of objects
    return(0);
}
```

Error Handling

The C++ RTL uses exceptions to provide error handling. A C++ RTL exception is an object of one of the following types:

- CFioranoException
- CJMSException

The `CFioranoException` class is a super class of `CJMSException` which has a utility function `checkForException()` that throws `CJMSException` if an exception occurs. As a result, an application can include all the Fiorano C++ RTL methods in a `try-catch` block and to catch all types of exception. In event of an error in the C++RTL layer, `CJMSException` with a specific error code and description is thrown.

The Exception can be caught at the application level and the associated message can be read using the public API `getMessage()`. The exception handling is also exhaustive; it provides the complete function stack trace at the moment of the error.

The exception stack is maintained in the Thread-local storage of C RTL, so that the exception that occurred in one thread doesn't interfere with the flow of other threads. The stack trace can be printed on the console using the API call `printStackTrace()`.

The following code fragment illustrates this technique:

```
#include <cppHeaders.h>
#include <iostream>
using namespace std;
// Using cpp native rtl namespace
using namespace cppnativertl;

int main(int argc,char* argv[])
{
    CHashTable *m_env;
    CInitialContext *m_ic;
    // Queue Connection Factory
    CQueueConnectionFactory *m_qcf;
    // Queue Connection
    CQueueConnection *m_qc;
    CQueue *m_queue;

    // Set up the environment variables
    try
    {
       m_env = new CHashTable();

       m_env->Put(SECURITY_PRINCIPAL, "anonymous");
       m_env->Put(SECURITY_CREDENTIALS,"anonymous");
       m_env->Put(CONNECT_URL,"http://localhost:1856");
       m_env->Put(BACKUP_URLS,"http://localhost:1956");
       m_env->Put(PROVIDER_URL,"http://localhost:1856");

       m_ic = new CInitialContext(m_env);

       m_qcf = dynamic_cast<CQueueConnectionFactory *> (m_ic->Lookup("primaryQCF"));
       if(m_qcf == 0)
       {
    throw new CJMSException("Connection Factory lookup  failed.Error in Type
casting.");
       }

       m_qc = m_qcf->createQueueConnection("anonymous", "anonymous");

       //Additional code here
       //Deletion of objects
    }
    catch(CJMSException *e)
    {
        cout << e->getMessage();
        e->printStackTrace();
        delete e;
    }
return(0);
}
```

The application must release the CJMSException object using the delete operator. The C++ RTL can create an exception for each error it detects during a call and link the exceptions to form a chain. After an application has caught the first exception, it can call the getLinkedException() method to get a pointer to the next exception in the chain. The application can continue to call the getLinkedException() method on each exception in the chain until a null pointer is returned, indicating that there are no more exceptions in the chain. Because the getLinkedException() method returns a pointer to a linked exception, the application must release the object using the delete operator.

Message listeners in C++

A C++ application uses a message listener to receive messages asynchronously. To receive messages asynchronously, the C++ application must define a message listener class that is based on the abstract class CMessageListener. The message listener class defined in the application must provide an implementation of the onMessage() method. The application can then instantiate the class to create a message listener and register the message listener with one or more message consumers by calling the setMessageListener() method for each message consumer. Subsequently, when a message arrives for a message consumer, the onMessage() method is invoked to deliver the message.

To stop the asynchronous delivery of messages to a message consumer, the application can call the setMessageListener() method again by passing a null pointer as a parameter instead of a pointer to a message listener. Unless the registration of a message listener is cancelled in this method, the message listener must exist for as long as the message consumer exists.

A new message listener can be registered with a message consumer without cancelling the registration of an existing message listener. If the onMessage() method of an existing message listener is running when a new message listener is registered, the active method completes normally and any subsequent messages are processed by calls to the onMessage() method of the new message listener. If a transaction is in progress when a message listener is changed, the transaction is completed by calls to the onMessage() method of the new message listener.

The following code fragment provides an example of a message listener class implementation with an onMessage() method:

```
#include <cppHeaders.h>
#include <iostream>
using namespace std;

// Using cpp native rtl namespace
using namespace cppnativertl;
// Message Listener implementation
class CMyMessageListener: public CMessageListener
{
public:
       void onMessage(CMessage *msg)
       {
            if(msg != NULL)
            {
                  CTextMessage *cmsg = (CTextMessage *)msg;
                  cout << "Received Message :: " << cmsg->getText() <<endl;
                  delete msg;
            }
       }
```

```
};
```

As the C ++RTL delivers a pointer to a message when it calls the onMessage() method, the application must release the message using the delete operator.

The following code fragment shows how an application can use this message listener class to implement the asynchronous delivery of messages to a message consumer:

```
#include <cppHeaders.h>
#include <iostream>
using namespace std;

// Using cpp native rtl namespace
using namespace cppnativertl;

int main(int argc,char* argv[])
{
    CHashTable *m_env;
    CInitialContext *m_ic;
    // Queue Connection Factory
    CQueueConnectionFactory *m_qcf;
    // Queue Connection
    CQueueConnection *m_qc;
    CQueue *m_queue;
    CMessageListener *m_msgl;

    try
    {

        // Set up the environment variables
        m_env = new CHashTable();

        m_env->Put(SECURITY_PRINCIPAL, "anonymous");
        m_env->Put(SECURITY_CREDENTIALS,"anonymous");
        m_env->Put(CONNECT_URL,"http://localhost:1856");
        m_env->Put(BACKUP_URLS,"http://localhost:1956");
        m_env->Put(PROVIDER_URL,"http://localhost:1856");

        m_ic = new CInitialContext(m_env);

        m_qcf = dynamic_cast<CQueueConnectionFactory *>(m_ic->Lookup("primaryQCF"));
        if(m_qcf == 0)
        {
            throw new CJMSException("Connection Factory lookup  failed.Error in Type
casting.");
        }

        m_qc = m_qcf->createQueueConnection("anonymous", "anonymous");

        m_queue = dynamic_cast<CQueue *>(m_ic->Lookup("primaryQueue"));
        if(m_queue == 0)
        {
            throw new CJMSException("Destination lookup failed.Error in Type casting.");
        }
```

```
        m_qc = m_qcf->createQueueConnection("anonymous", "anonymous");
m_qs = m_qc->createQueueSession(FALSE, AUTO_ACKNOWLEDGE);
m_receiver = m_qs->createReceiver(m_queue);
m_msgl = new CMyMessageListener();
m_receiver->setMessageListener(m_msgl);
m_qc->start();
//Other code here
//Deleting the objects
delete m_env;
if(m_receiver != NULL)
m_receiver->close();
delete m_receiver;
if(m_qs != NULL)
m_qs->close();
delete m_qs;
if(m_qc != NULL)
m_qc->close();
delete m_qc;
delete m_qcf;
delete m_queue;
delete m_ic;
delete m_msgl;
}
catch(CJMSException *e)
{
cout << e->getMessage();
e->printStackTrace();
delete e;
}
return(0);
}
```

Connection Start and Stop

When an application creates a connection, the connection is in stop mode. When the connection is in stop mode, the application can initialize sessions and it can send messages but cannot receive them, either synchronously or asynchronously.

An application can start a connection by calling the Start Connection method. When the connection is in start mode, the application can send and receive messages. The application can stop and restart the connection by calling the `stop()` and `start()` methods.

Connection Close

In the above code fragment, the statement `m_qc->close()`closes the connection. When an application closes a connection, it closes all the sessions associated with the connection and deletes certain objects associated with these sessions. It also rolls back any transactions currently in progress within the sessions. It ends the communications connection with the messaging server. It releases the memory and other internal resources used by the connection in the underlying CRTL.

Exception listeners in C++

Using an exception listener is similar in principle to using a message listener. A C++ application must define an exception listener class that is based on the abstract class C*ExceptionListener*. The exception listener class defined in the application must provide an implementation of the `onException()` method. The application can then instantiate the class to create an exception listener, and register the exception listener with a connection by calling the `setExceptionListener()` method. Subsequently, if C++RTL detects a problem with the connection, the `onException()` method is invoked to pass an exception to the application.

To stop the asynchronous reporting of problems with a connection, the application can call the `setExceptionListener()` method by passing a null pointer as the parameter instead of a pointer to an exception listener. Unless the registration of an exception listener is cancelled in this method, the exception listener must exist for as long as the connection exists.

As the C++RTL passes a pointer to an exception when it calls the `onException()` method, the application must release the exception by using the C++ delete operator.

The following code fragment provides an example of a exception listener class implementation with an `onException()` method:

```
#include <cppHeaders.h>
#include <iostream>
using namespace std;

// Using cpp native rtl namespace
using namespace cppnativertl;

class CMyExceptionListener: public CExceptionListener
{
public:
void onException (CFioranoException* pException)
{
    try
    {
        if(pException->getMessage() != NULL)
            cout << "Exception is ::"<<pException->getMessage()<<endl;
        delete pException;
    }
    catch (CJMSException *je)
    {
        je->printStackTrace();
        je->clearException();
        delete je;
    }
 }
};
int main(int argc,char* argv[])
{
CHashTable *m_env;
CInitialContext *m_ic;
// Queue Connection Factory
CQueueConnectionFactory *m_qcf;
// Queue Connection
```

```
CQueueConnection *m_qc;
CQueue *m_queue;
CExceptionListener *m_exp;
// Set up the environment variables
m_env = new CHashTable();
m_env->Put(SECURITY_PRINCIPAL, "anonymous");
m_env->Put(SECURITY_CREDENTIALS,"anonymous");
m_env->Put(CONNECT_URL,"http://localhost:1856");
m_env->Put(BACKUP_URLS,"http://localhost:1956");
m_env->Put(PROVIDER_URL,"http://localhost:1856");
try
{
m_ic = new CInitialContext(m_env);
m_qcf = dynamic_cast<CQueueConnectionFactory *>(m_ic->Lookup("primaryQCF"));
if(m_qcf == 0)
{
throw new CJMSException("Connection Factory lookup failed.Error in Type
casting.");
}
m_qc = m_qcf->createQueueConnection("anonymous", "anonymous");
m_queue = dynamic_cast<CQueue *>(m_ic->Lookup("primaryQueue"));
if(m_queue == 0)
{
throw new CJMSException("Destination lookup failed.Error in Type casting.");
}
m_qc = m_qcf->createQueueConnection("anonymous", "anonymous");
m_exp = new CMyExceptionListener();
m_qc->setExceptionListener(m_exp);
//Other code here
//Deleting the objects
delete m_env;

if(m_qc != NULL)
m_qc->close();
delete m_qc;
delete m_exp;
delete m_qcf;
delete m_queue;
delete m_ic;

}
catch(CJMSException *e)
{
cout << e->getMessage();
e->printStackTrace();
delete e;
}
return(0);
}
```

Advisory Message listeners in C++

Advisory message listener is used to receive asynchronously delivered advisory messages when the reconnection thread is active. Using an advisory message listener is similar in principle to using a message listener.

A C++ application must define an advisory message listener class that is based on the abstract class CAdvisoryMessage. The advisory message listener class defined in the application must provide an implementation of the `onAdvisoryMessage()` method. The application can then instantiate the class to create an advisory message listener, and register the advisory message listener with a connection by calling the `setAdvisoryMessageListener()` method. Subsequently, if C++RTL receives asynchronous delivered advisory messages when the reconnection thread is active, the `onAdvisoryMessage()` method is invoked to pass an advisory message to the application.

To stop the asynchronous reporting problems with a connection, the application can call the `setAdvisoryMessageListener()` method again, by passing a null pointer as the parameter instead of a pointer to an advisory message listener. Unless the registration of an advisory message listener is cancelled in this method, the advisory message listener must exist as long as the connection exists.

As the C++RTL passes a pointer to an advisory message when it calls the `onAdvisoryMessage()` method, the application must release the advisory message by using the C++ delete operator.

The following code fragment provides an example of a exception listener class implementation with an `onAdvisoryMessage()` method:

```
#include <cppHeaders.h>
#include <iostream>
using namespace std;

// Using cpp native rtl namespace
using namespace cppnativertl;
class CMyAdvisoryMessageListener: public CAdvisoryMsgListener
{
public:
    void onAdvisoryMessage(CAdvisoryMessage *msg)
    {
        try{

                cout << "Advisory Message is :: " << msg->getAdvisoryMsgString() <<endl;
                cout << "State of the server is :: " << msg->getAMState() << endl;
                delete msg;
        }
        catch (CJMSException *je)
        {
                je->printStackTrace();
                je->clearException();
                delete je;
        }
    }
};

int main(int argc,char* argv[])
{
    CHashTable *m_env;
```

```
    CInitialContext *m_ic;
    // Queue Connection Factory
    CQueueConnectionFactory *m_qcf;
    // Queue Connection
    CQueueConnection *m_qc;
    CQueue *m_queue;
    CAdvisoryMsgListener *m_adv;

    try
    {
        // Set up the environment variables
        m_env = new CHashTable();
        m_env->Put(SECURITY_PRINCIPAL, "anonymous");
        m_env->Put(SECURITY_CREDENTIALS,"anonymous");
        m_env->Put(CONNECT_URL,"http://localhost:1856");
        m_env->Put(BACKUP_URLS,"http://localhost:1956");
        m_env->Put(PROVIDER_URL,"http://localhost:1856");

        m_ic = new CInitialContext(m_env);

        m_qcf = dynamic_cast<CQueueConnectionFactory *>(m_ic->Lookup("primaryQCF"));
        if(m_qcf == 0)
        {
            throw new CJMSException("Connection Factory lookup  failed.Error in Type
casting.");
        }

        m_qc = m_qcf->createQueueConnection("anonymous", "anonymous");

        m_queue = dynamic_cast<CQueue *>(m_ic->Lookup("primaryQueue"));
        if(m_queue == 0)
{
throw new CJMSException("Destination lookup failed.Error in Type casting.");
}
m_qc = m_qcf->createQueueConnection("anonymous", "anonymous");
m_adv = new CMyAdvisoryMessageListener();
m_qc->setAdvisoryMessageListener(adv);
//Other code here
//Deleting the objects
delete m_env;
if(m_qc != NULL)
m_qc->close();
delete m_qc;
delete m_adv;
delete m_qcf;
delete m_queue;
delete m_ic;

}
catch(CJMSException *e)
{
cout << e->getMessage();
e->printStackTrace();
delete e;
```

```
}
return(0);
}
```

Lookup of Administered Objects

FioranoMQ supports lookup of administered objects using the JNDI interface. In case of the CPPRTL, this support has been provided using the CInitialContext class that works mostly on the lines of the InitialContext object of JNDI. Client applications can create an object of CInitialContext and lookup different administered objects from the FioranoMQ Server.

CInitialContext class is the starting context for performing naming operations. The CInitialContext is used to lookup administered objects from the FioranoMQ JNDI store. The client application must use Lookup method of CInitialcontext class to retrieve the named object. The lookup method returns the named object as CLookupHelper which can be dynamically casted at runtime to the required admin object.

The following code fragment provides an example of a CInitialContext class and looking up administered objects.

```
#include <cppHeaders.h>
#include <iostream>
using namespace std;

// Using cpp native rtl namespace
using namespace cppnativertl;

int main(int argc,char* argv[])
{
    CHashTable *m_env;
    CInitialContext *m_ic;
    // Queue Connection Factory
    CQueueConnectionFactory *m_qcf;
    // Queue Connection
    CQueue *m_queue;

    try
    {
        // Set up the environment variables
        m_env = new CHashTable();
        m_env->Put(SECURITY_PRINCIPAL, "anonymous");
        m_env->Put(SECURITY_CREDENTIALS,"anonymous");
        m_env->Put(CONNECT_URL,"http://localhost:1856");
        m_env->Put(BACKUP_URLS,"http://localhost:1956");
        m_env->Put(PROVIDER_URL,"http://localhost:1856");

        m_ic = new CInitialContext(m_env);

        m_qcf = dynamic_cast<CQueueConnectionFactory *>(m_ic->Lookup("primaryQCF"));
        if(m_qcf == 0)
        {
```

```
            throw new CJMSException("Connection Factory lookup failed.Error in Type
casting.");
        }

        m_queue = dynamic_cast<CQueue *>(m_ic->Lookup("primaryQueue"));
        if(m_queue == 0)
        {
            throw new CJMSException("Destination lookup failed.Error in Type casting.");
        }

        //Other code here
    }
    catch(CJMSException *e)
    {
        cout << e->getMessage();
        e->printStackTrace();
        delete e;
    }
return(0);
}
```

Object Deletion

When an application deletes a cppnativertl object that it has created, cppnativertl releases the internal resources that have been allocated to that object.

When an application creates a cppnativertl object, cppnativertl allocates memory and other internal resources to the object. cppnativertl retains these internal resources until the application explicitly deletes the object by calling the object's close or delete method. The application should close the MessageConsumer, MessageProducer, QueueBrowser, Requestor, Session, and Connection objects before deleting it.

The application should close/delete MessageConsumer, MessageProducer, QueueBrowser, and Requestor objects before closing/deleting Session object and then connection object should be closed/deleted. MessageListener, ExceptionListener, AdvisoryMessageListener, and administered objects such as Connectionfactory, Destination (Queue, Topic) should be deleted only after closing/deleting the connection object. The cppnativertl releases the internal resources of the associated object that has been deleted.

The following objects should be deleted in application side:

- MessageConsumer, MessageProducer, QueueBrowser and Requestor
- Session
- Connection
- MessageListener, ExceptionListener and AdvisoryMessageListener
- Administered objects such as Connectionfactory, Destination (Queue, Topic)

Getting Message Properties

The CProperty class wraps the message property value, which includes the value type, size and the value itself.

The CMessage class has a member function `getPropertyNames` which returns an CEnumeration object.The CEnumeration object contains all the property names present in the received message.The property names can be retrieved from the CEnumeration object using `nextElement` method.The `nextElement` method returns void* and it should be type casted to const char*. By using `getProperty` method of CMessage class which will return CProperty object, all the property values can be retreived.

The following code segments gives an example of using CProperty class.

```
class CMyMessageListener: public CMessageListener
{

public:
    void onMessage(CMessage *msg)
    {
        CEnumeration *propertyNames = NULL;
        CTextMessage *cmsg = (CTextMessage *)msg;
        propertyNames = cmsg->getPropertyNames();

        CProperty *value;

        while(propertyNames->hasMoreElements())
        {
        std::string name;
        PropertyIndex type;

        name = (const char*)propertyNames->nextElement();
        cout << "property name " << name.c_str() << endl;

        value = cmsg->getProperty(name.c_str());

        type = value->getPropertyType();

            switch(type)
            {
                case StringIndex:
                {
                    mqcstring val = cmsg->getStringProperty(name.c_str());
                    break;
                }

                case 0:
                {
                    mqbyte b = cmsg->getByteProperty(name.c_str());
                    break;
                }

                case ByteArrayIndex:
                    cout << "ByteArray" << endl;
            }
```

```
            //Other datatype checks here
        }

        cout << "Received Message :: " << cmsg->getText() <<endl;
        delete msg;
    }
};
```

Chapter 4: FioranoMQ C++ RTL – Classes

CLookupHelper

The CLookupHelper class is an abstract base class. The CLookupHelper helper provides methods to lookup connection factories and destinations.

Inheritance Hierarchy

None

Subclasses

- CTopicConnectionFactory
- CQueueConnectionFactory
- CFioranoConnectionFactory
- CAdminConnectionFactory
- CGenericAdminObject
- CQueue
- CTopic

Constructors

`CLookupHelper()`

Parameters:

Default constructor

Methods

`virtual AdminObjectType getLookupObjectType() FMQCONST throw (CJMSException *) = 0`

Returns the type of the looked up administered object.

Parameters: None

Returns: Returns enum AdminObjectType, the looked up Object type.

Exceptions: CJMSException

CInitialContext

The CInitialContext class is used to bind to a local or a remote FioranoMQ Server in order to perform `lookup()` operations on administered objects such as Topics, Queues and Connection Factories.

Inheritance Hierarchy

None

Subclasses

None

Constructors

```
CInitialContext() throw (CJMSException *);
```

Initial context with default parameters

Parameters: None

```
CInitialContext(CHashTable *env) throw (CJMSException*);
```

Creates an initial context using the supplied environment variables.

Parameters: env - CHashtable to map key value pairs.

Example:

```
m_env->Put(SECURITY_PRINCIPAL, m_usrName);
m_env->Put(SECURITY_CREDENTIALS,m_usrPasswd);
m_env->Put(CONNECT_URL,m_providerURL);
m_env->Put(BACKUP_URLS,m_backupURL);
m_env->Put(PROVIDER_URL,m_providerURL);
```

Methods

```
CLookupHelper* Lookup(mqcstring adminObjectName) FMQCONST throw (CJMSException *);
```

Retrieves the named object of type CLookupHelper class. The CLookupHelper class can be dynamically casted to one of:

- CTopicConnectionFactory
- CQueueConnectionFactory
- CFioranoConnectionFactory
- CAdminConnectionFactory
- CGenericAdminObject
- CQueue

- CTopic

One can use the CLookupHelper's `getLookupObjectType()` to get the type of the retrieved administered object.

Parameters:

- adminObjectName: Name of the administered object to be looked up.

Returns: CLookupHelper

Exceptions: CJMSException.

```
CLookupHelper* LookupTCF(mqcstring adminObjectName) FMQCONST throw (CJMSException *);
```

Retrieves the named administered object from FioranoMQ's JNDI store. This method is used in the scenario of serverless mode.

Parameters:

- adminObjectName – Name of the administered object to be looked up.

Returns: CLookupHelper

Exceptions: CJMSException.

```
CLookupHelper* LookupQCF(mqcstring adminObjectName) FMQCONST throw (CJMSException *);
```

Retrieves the named administered object from FioranoMQ's JNDI store. This method is used in the scenario of serverless mode.

Parameters:

- adminObjectName – Name of the administered object to be looked up.

Returns: CLookupHelper

Exceptions: CJMSException.

CAdvisoryMessage

The CAdvisoryMessage class defines methods that return the connection state of the application with the server.

Inheritance Hierarchy

None

Constructors

```
CAdvisoryMessage(struct _MQAdvisoryMessage *msg);
```

CAdvisoryMessage constructor.

Parameters: The MQAdvisoryMessage structure as defined in C runtime library.

Subclasses

None

Methods

```
mqcstring_unicode getAdvisoryMsgString() throw (CJMSException*);
```

Gets the advisory message describing the connection status.

Parameters: None

Returns: The Advisory message string (const char*).

Exceptions: CJMSException.

```
mqint getAMState()throw (CJMSException *);
```

Gets the state of the connection with the server.

Parameters: None

Returns: The state of the connection with the server.

Exceptions: CJMSException.

```
mqboolean isActive() throw (CJMSException *);
```

Returns TRUE (1) if the connection with the server is alive.

Parameters: None

Returns:

- 1 – If the connection state is active
- 0 – Inactive.

Exceptions: CJMSException.

```
mqboolean isRevalidating() throw (CJMSException *);
```

Returns TRUE (1) if the connection is trying to revalidate with the server.

Parameters: None

Returns:

- TRUE – If the connection is trying to revalidate
- FALSE – If the connection is not trying to revalidate

Exceptions: CJMSException.

```
mqboolean isDisconnected() throw (CJMSException *);
```

Returns TRUE if the connection with the server is down.

Parameters: None

Returns:

- TRUE – If the connection is down
- FALSE – If the connection is active

Exceptions: CJMSException.

```
mqboolean isTransferring()throw (CJMSException *);
```

True if connection is transferring messages from CSP (Client Side Persistence)

Parameters: None

Returns:

- 1 – TRUE
- 0 – FALSE

Exceptions: CJMSException.

```
mqboolean isTransferComplete()throw (CJMSException *);
```

True if connection has completed transferring CSP messages.

Parameters: None

Returns:

- 1 – TRUE
- 0 – FALSE

Exceptions: CJMSException.

CAdvisoryMsgListener

An application uses CAdvisoryMsgListener to receive Advisory message on the connection state.

Inheritance Hierarchy

None

Subclasses

None

Methods

```
virtual void onAdvisoryMessage(CAdvisoryMessage *msg) = 0;
```

Application user must implement the 'onAdvisoryMessage' to receive Advisory messages. See more information on Advisory Message Listeners, section Advisory Message listeners in C++.

Parameters:

- msg – CadvisoryMessage

Returns: void

Exceptions: None

CFioranoException

This is the base class for all exceptions defined in the Fiorano framework.

Inheritance Hierarchy

None

Subclasses

CJMSException

Constructors

```
CFioranoException(mqcstring str);
```

Constructs a new `CFioranoException` object with the provided error code string.

Parameters:

- str
- String representing the errorcode

```
CFioranoException(mqcstring errCode, mqcstring errDesc);
```

Constructs a new `CFioranoException` object with the provided error code string and error description.

Parameters:

- errCode
- String representing the error code

- errDesc
- String representing the error description

Methods

`static void checkForException() throw (CJMSException *)`

Utility function that throws `CJMSException` if an exception had occurred.

Parameters: None

Returns: void

Exceptions: CJMSException

`static void checkForException(mqcstring errCode)`

Overloaded utility function that throws `CJMSException` if an exception had occurred with the provided errCode.

Parameters:

- errCode
- String represting the errorcode

Returns: void

`static void checkForException(mqcstring errCode, mqcstring errDesc)`

Overloaded utility function that throws `CJMSException` if an exception had occurred with the provided errCode and errDesc.

Parameters:

- errCode: The error code that identifies the error
- errDesc: Description of the error

Returns: void

`const DN_STRING getLastErrorCode() FMQCONST`

Returns the last error code as string

Parameters: None

Returns: DN_STRING(mqstring)

`const DN_STRING getMessage() FMQCONST`

The associated error message can be read.

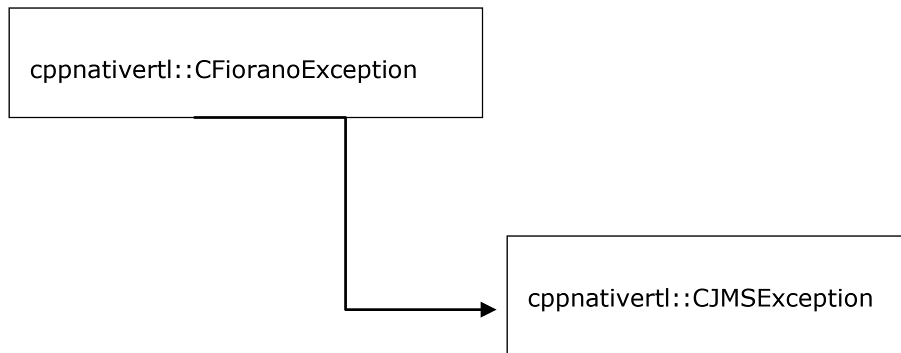
Parameters: None

Returns: DN_STRING(mqstring)

CJMSException

Constructs a `CJMSException` with the specified reason and with the error code.

Inheritance Hierarchy



Subclasses

None

Constructors

`CJMSException(mqcstring errCode)`

Constructs a new `CJMSException` object with the provided error code string.

Parameters:

- errCode: The error code that identifies the error

`CJMSException(mqcstring errCode, mqcstring errDesc)`

Constructs a new `CJMSException` object with the provided error code string and error description.

Parameters:

- errCode: The error code that identifies the error
- errDesc: Description of the error

`CJMSException(mqcstring errCode, CJMSException *linkedException)`

Constructs a new `CJMSException` object with the provided error code string and linked exception.

Parameters:

- errCode: The error code that identifies the error

- linkedException: The pointer to the `CJMSException` which acts as linked exception

`CJMSException(mqcstring errCode, mqcstring errDesc, CJMSException *linkedException)`

Constructs a new `CJMSException` object with the provided error code string and linked exception and error description.

Parameters:

- errCode: The error code that identifies the error
- errDesc: Description of the error
- linkedException: The pointer to the `CJMSException` which acts as linked exception.

Methods

`void printStackTrace()`

Prints the function stack trace on the console.

Parameters: None

Returns: void

`const DN_STRING getStackTrace()`

Gets the stack trace of the function.

Parameters: None

Returns: Returns the stack trace as DN_STRING (mqstring).

`const DN_STRING getErrorCode()`

Gets the vendor specific error code.

Parameters: None

Returns: The vendor specific error code.

`CJMSException *getLinkedException()`

Gets the exception linked to this one.

Parameters: None

Returns: The linked exception

CExceptionListener

If FioranoMQ cpp detects a serious problem with a CConnection object, it informs the CConnection object's CExceptionListener, if one has been registered. It does this by calling the listener's `onException` method, passing it a `CJMSException` argument describing the problem.

Inheritance Hierarchy

None

Subclasses

None

Constructors

Default

Methods

```
virtual void onException(CFioranoException *msg) = 0
```

Notifies the user of JMS exception. The user is expected to override this function with the required functionality.

Parameters

- Msg: Message handle

Returns: Void

CMessageListener

A CMessageListener object is used to receive asynchronously delivered messages. Passes a message to the listener.

Inheritance Hierarchy

None

Subclasses

None

Constructors

Default

Methods

```
virtual void onMessage(CMessage *msg) = 0
```

Notifies the user with message received. The user is expected to override this function with the required functionality.

Parameters

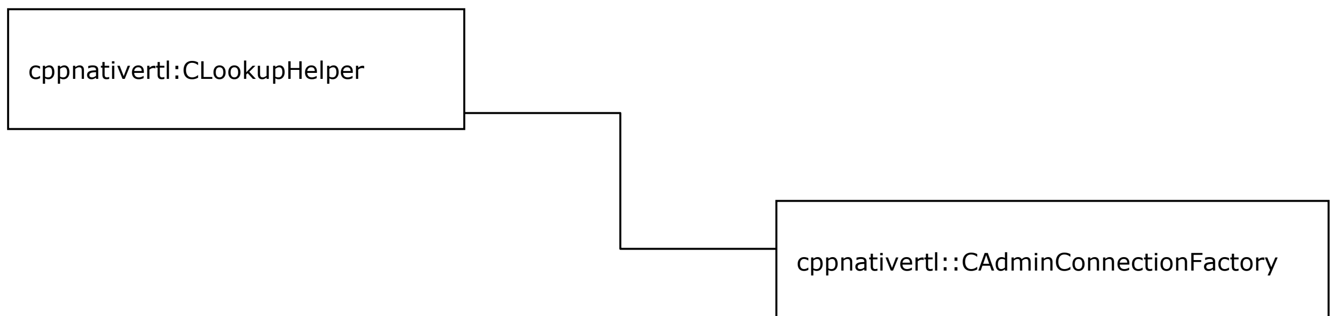
- Msg: The message passed to the listener

Returns: Void

CAdminConnectionFactory

The Admin connection Factory used for creating Admin connections to the FioranoMQ Server.

Inheritance Hierarchy



Subclasses

None

Constructors

```
CAdminConnectionFactory (struct _AdminConnectionFactory *pacf);
```

Parameters:

- pacf – pointer to the AdminConnectionFactory C structure.

Methods

```
CAdminConnection *createAdminConnection(mqcstring username, mqcstring password) throw
(CJMSException *);
```

Creates an Admin connection with the FioranoMQ Server using the specified user credentials. The username provided must belong to the Administrators group of FioranoMQ Security realm.

Parameters:

- username: user who belongs Administrators group of Fiorano Security realm

- password: password

Returns: CAdminConnection

Exceptions: CJMSException

```
CAdminConnection *createAdminConnectionDefParams()throw (CJMSException *);
```

Creates an Admin connection with the FioranoMQ Server using default credentials.

Parameters: None

Returns: CAdminConnection

Exceptions: CJMSException

```
AdminObjectType getLookupObjectType() FMQCONST throw (CJMSException *);
```

This function will returns the type of looked up Object. The return value AdminObjectType is enum of:

- OBJID_QCF, (QueueConnectionFactory)
- OBJID_TCF, (TopicConnectionFactory)
- OBJID_QUEUE, (Queue destination)
- OBJID_TOPIC, (Topic destination)
- OBJID_GAO, (GenericAdminObject)
- OBJID_ACF, (AdminConnectionFactory)
- OBJID_UCF, (UnifiedConnectionFactory)

Parameters: None

Returns: AdminObjectType

Exceptions: CJMSException

CConnectionFactory

A `ConnectionFactory` object encapsulates a set of connection configuration parameters that has been defined by an administrator. A client uses it to create a connection with a JMS provider. The CConnectionFactory is an abstract base class.

Inheritance Hierarchy

None

Subclasses

- CFioranoConnectionFactory
- CQueueConnectionFactory

- CTopicConnectionFactory

Methods

```
virtual CConnection *createConnection() throw (CJMSException *) = 0;
```

Creates a connection with the default user identity. The connection is created in stopped mode. No messages will be delivered until the `Connection.start()` method is explicitly called.

Parameters: None

Returns: CConnection object

Exceptions: CJMSException

```
virtual CConnection *createConnection(mqcstring username, mqcstring password)throw
(CJMSException *) = 0;
```

Creates a connection with the specified user identity. The connection is created in stopped mode. No messages will be delivered until the Connection`.start` method is explicitly called. If NULL values are passed as parameters, then it considers the default identity.

Parameters:

- username – The caller's user name
- password – The caller's password

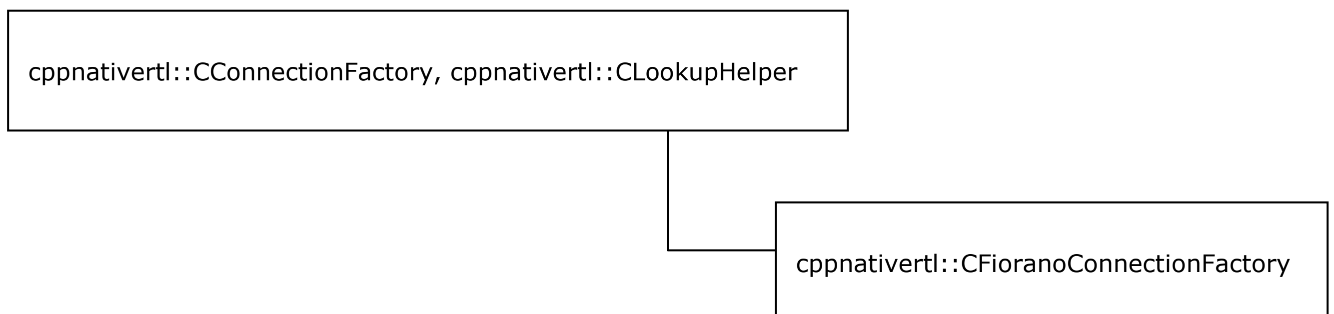
Returns: CConnection object

Exceptions: CJMSException

CFioranoConnectionFactory

A client uses a `CFioranoConnectionFactory` object to create unified Connection objects to use with PTP and PubSub messaging models.

Inheritance Hierarchy



Subclasses

None

Constructors

```
CFioranoConnectionFactory(struct _ConnectionFactory *pcf);
```

Parameters:

- `pcf` – pointer to the ConnectionFactory C structure.

```
CFioranoConnectionFactory();
```

Parameters: None

```
CFioranoConnectionFactory(CHashTable *env, mqstring connectionFactoryName)throw
(CJMSException *);
```

Creates a ConnectionFactory with the specified name and specified environment variables without using JNDI lookup.

Parameters:

- `env` - CHashTable contains the specified environment variables.
- `connectionFactoryName` - Name of the ConnectionFactory

Exceptions: CJMSException

Methods

```
CConnection *createConnection() throw (CJMSException *);
```

Creates a unified connection with the default user identity. The connection is created in stopped mode. No messages will be delivered until the Connection.start method is explicitly called.

Parameters: None

Returns: CConnection object

Exceptions: CJMSException

```
CConnection *createConnection(mqcstring username, mqcstring password) throw(CJMSException *);
```

Creates a unified connection with the default user identity. The connection is created in stopped mode. No messages will be delivered until the `Connection.start()` method is explicitly called.

Parameters:

- username – The caller's user name
- password – The caller's password

Returns: CConnection object

Exceptions: CJMSException

```
AdminObjectType getLookupObjectType() FMQCONST throw (CJMSException *);
```

This function returns the type of looked up Object. The return value AdminObjectType is enum of:

- OBJID_QCF, (QueueConnectionFactory)
- OBJID_TCF, (TopicConnectionFactory)
- OBJID_QUEUE, (Queue destination)
- OBJID_TOPIC, (Topic destination)
- OBJID_GAO, (GenericAdminObject)
- OBJID_ACF, (AdminConnectionFactory)
- OBJID_UCF, (UnifiedConnectionFactory)

Parameters: None

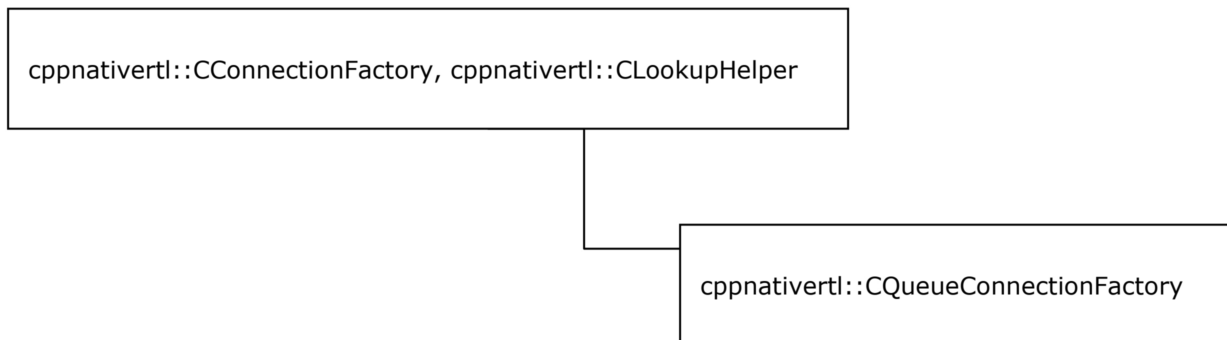
Returns: AdminObjectType

Exceptions: CJMSException

CQueueConnectionFactory

A client uses a `CQueueConnectionFactory` object to create `CQueueConnection` objects with a point-to-point JMS provider.

Inheritance Hierarchy



Subclasses

None

Constructors

```
CQueueConnectionFactory(struct _QueueConnectionFactory *pqcf);
```

Parameters:

- Pqcf – pointer to the QueueConnectionFactory C structure.

```
CQueueConnectionFactory(CHashTable* env,mqstring qcfName)throw (CJMSException *);
```

Creates a new QueueConnectionFactory with the specified name and with the
specified properties on the hashtable, without using JNDI lookup.

Parameters:

- env - CHashTable contains the specified environment variables.
- qcfName - Name of the QueueConnectionFactory

Exceptions: CJMSException

Methods

```
CQueueConnection *createQueueConnection() throw (CJMSException *);
```

Creates a queue connection with the default user identity. The connection is created in stopped mode. No messages will be delivered until the Connection.start method is explicitly called.

Parameters: None

Returns: CQueueConnection object

Exceptions: CJMSException

```
CQueueConnection *createQueueConnection(mqcstring username, mqcstring password) throw
(CJMSException *);
```

Creates a queue connection with the default user identity. The connection is created in stopped mode. No messages will be delivered until the `Connection.start` method is explicitly called.

Parameters:

- username – The caller's user name
- password – The caller's password

Returns: CQueueConnection object

Exceptions: CJMSException

```
AdminObjectType getLookupObjectType() FMQCONST throw (CJMSException *);
```

This function will returns the type of looked up Object. The return value AdminObjectType is enum of:

- OBJID_QCF, (QueueConnectionFactory)
- OBJID_TCF, (TopicConnectionFactory)
- OBJID_QUEUE, (Queue destination)
- OBJID_TOPIC, (Topic destination)
- OBJID_GAO, (GenericAdminObject)
- OBJID_ACF, (AdminConnectionFactory)

- OBJID_UCF, (UnifiedConnectionFactory)

Parameters: None

Returns: AdminObjectType

Exceptions: CJMSException

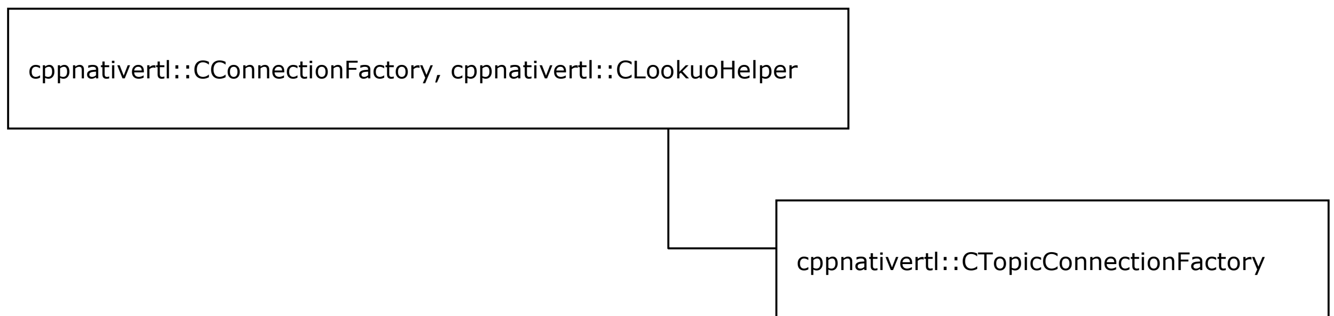
Inherited Methods

CreateConnection, getLookupObjectType

CTopicConnectionFactory

A client uses a `CTopicConnectionFactory` object to create `CTopicConnection` objects with a PubSub JMS provider.

Inheritance Hierarchy



Subclasses

None

Constructors

```
CTopicConnectionFactory(struct _TopicConnectionFactory *ptcf);
```

Parameters:

- ptcf – pointer to the TopicConnectionFactory C structure.

```
CTopicConnectionFactory(CHashTable *env,mqstring tcfName)throw (CJMSException *);
```

Creates a new TopicConnectionFactory with the specified name and with the specified properties on the hashtable, without using JNDI lookup.

Parameters:

- env - CHashTable contains the specified environment variables.
- tcfName - Name of the TopicConnectionFactory

Exceptions: CJMSException

Methods

```
CTopicConnection *createTopicConnection()  throw (CJMSException *);
```

Creates a topic connection with the default user identity. The connection is created in stopped mode. No messages will be delivered until the `Connection.start()` or simple `start()` method is explicitly called.

Parameters: None

Returns: CTopicConnection object

Exceptions: CJMSException

```
CTopicConnection *createTopicConnection(mqcstring username, mqcstring password) throw
(CJMSException *);
```

Creates a topic connection with the default user identity. The connection is created in stopped mode. No messages will be delivered until the `Connection.start` method is explicitly called.

Parameters:

- username – The caller’s user name
- password – The caller’s password

Returns: CTopicConnection object

Exceptions: CJMSException

```
AdminObjectType getLookupObjectType() FMQCONST throw (CJMSException *);
```

This function will returns the type of looked up Object. The return value AdminObjectType is enum of:

- OBJID_QCF, (QueueConnectionFactory)
- OBJID_TCF, (TopicConnectionFactory)
- OBJID_QUEUE, (Queue destination)
- OBJID_TOPIC, (Topic destination)
- OBJID_GAO, (GenericAdminObject)
- OBJID_ACF, (AdminConnectionFactory)
- OBJID_UCF, (UnifiedConnectionFactory)

Parameters: None

Returns: AdminObjectType

Exceptions: CJMSException

Inherited Methods

`CreateConnection, getLookupObjectType`

CAdminConnection

AdminConnectionFactories are used to create AdminConnections with the FioranoMQ Server. The AdminConnection is created with the server running on the ConnectURL specified in the AdminConnectionFactory and if the same is unavailable then the RTL tries to make a connection with a BackupURL, if any. AdminConnections can be created using default user identity ("admin","passwd" in case of FioranoMQ) or by specifying a username and password.

Inheritance Hierarchy

None

Subclasses

None

Constructors

`CAdminConnection()`

The default constructor.

Parameters: None

`CAdminConnection(struct _AdminConnection* pAdminConn)`

Parameters:

- pAdminConn – AdminConnection structure defined in C runtime.

Methods

`CMQAdminService* getMQAdminService() FMQCONST throw (CJMSException*)`

A MQAdminService object provides methods for creating and deleting Queues, Topics,QueueConnectionFactory and TopicConnectionFactory objects. Various get/set methods specify the object properties to and from the server.

Returns: A newly created admin connection.

Parameters: None

Exceptions: CJMSException

```
void close() throw (CJMSException*)
```

Closes the connection.

Returns: Void

Parameters: None

Exceptions: CJMSException

```
void setAdvisoryMessageListener(const CAdvisoryMsgListener *advMsgListener) throw
(CJMSException *);
```

Sets the Advisory Message listener to the connection. The user application should implement the 'onAdvisoryMessage' method to receive any state change event that happens on this connection. For more information refer to section on Advisory Message listeners in C++.

Parameters:

- advMsgListener – the CAdvisoryMsgListener

Returns: Void

Exceptions: CJMSException

CConnection

A `Connection` object is a client's active connection to its JMS provider.

Inheritance Hierarchy

None

Subclasses

- CFioranoConnection
- CQueueConnection
- CTopicConnection

Methods

```
virtual void close() throw (CJMSException *) = 0;
```

Closes the connection. If an application tries to close a connection that is already closed, the call is ignored. Closing a connection causes all temporary destinations to be deleted.

Parameters: None

Returns: Void

Exceptions: CJMSException.

```
virtual CJMSSession *createSession(const mqboolean transacted, const mqint acknowledgeMode)
throw (CJMSException *) = 0;
```

Creates a CJMSSession object.

Parameters:

- transacted – Indicates whether the session is transacted.
- acknowledgeMode: indicates whether the consumer or the client will acknowledge any messages it receives; ignored if the session is transacted. Legal values are `AUTO_ACKNOWLEDGE`, `CLIENT_ACKNOWLEDGE`, and `DUPS_OK_ACKNOWLEDGE`.

Returns: CJMSSession object

Exceptions: CJMSException.

```
virtual mqcstring getClientID() FMQCONST throw (CJMSException *) = 0;
```

Gets the client identifier for this connection. This value is specific to the JMS provider. It is either preconfigured by an administrator in a `ConnectionFactory` object or assigned dynamically by the application by calling the `setClientID` method.

Parameters: None

Returns: The client id of this connection.

Exceptions: CJMSException.

```
virtual CExceptionListener *getExceptionListener() FMQCONST throw (CJMSException *) = 0;
```

Get a pointer to the exception listener that is registered with the connection.

Parameters: None

Returns: Pointer to the exception listener.

Exceptions: CJMSException.

```
virtual void setExceptionListener(const CExceptionListener * exceptionListener) throw
(CJMSException *) = 0;
```

Sets an exception listener for this connection. If a JMS provider detects a serious problem with a connection, it informs the connection's ExceptionListener, if one has been registered. It does this by calling the listener's onException method, passing it a JMSException object describing the problem.

An exception listener allows a client to be notified of a problem asynchronously. Some connections only consume messages, so they would have no other way to learn their connection has failed.

Parameters:

- exceptionListener: A pointer to the exception listener. If an exception listener is already registered with the connection, you can cancel the registration by specifying a null pointer instead.

Returns: Void

Exceptions: CJMSException

```
virtual void setClientID(mqcstring clientID) throw (CJMSException *) = 0;
```

Set a client identifier for the connection. If an application calls this method to set a client identifier for a connection, the application must do so immediately after creating the connection, and before performing any other operation on the connection.

Parameters:

- clientID: The unique client identifier.

Returns: Void

Exceptions: CJMSException

```
virtual void start() FMQCONST throw (CJMSException *) = 0;
```

Starts (or restarts) a connection's delivery of incoming messages. A call to `start` on a connection that has already been started is ignored.

Parameters: None

Returns: Void

Exceptions: CJMSException

```
virtual void stop() FMQCONST throw (CJMSException *) = 0;
```

Temporarily stops a connection's delivery of incoming messages. Delivery can be restarted using the connection's start method. When the connection is stopped, delivery to all the connection's message consumers is inhibited: synchronous receives block, and messages are not delivered to message listeners. Stopping a connection has no effect on its ability to send messages. A call to `stop` on a connection that has already been stopped is ignored.

Parameters: None

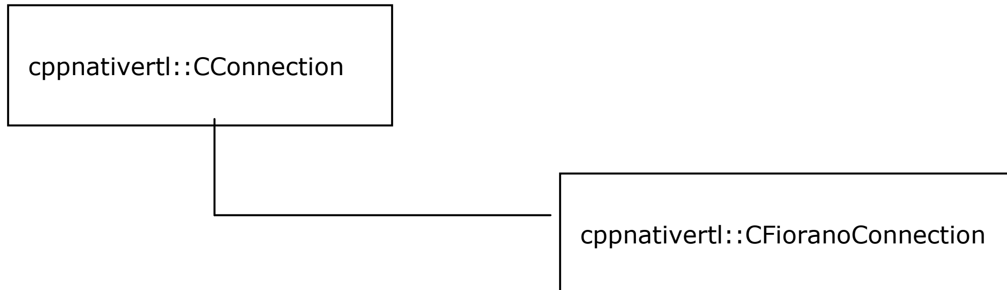
Returns: Void

Exceptions: CJMSException

CFioranoConnection

CFioranoConnection represents the Unified messaging model, where it can be used for both Point-to-Point and Pub/Sub messaging domains. Each CFioranoConnection creates a CQueueConnection and CTopicConnection based on the destination type used for sending/Receiving messages.

Inheritance Hierarchy



Subclasses

None

Constructors

- CFioranoConnection() throw (CJMSException *);
- Creates defualt Fiorano Unified Connection.

  Parameters: None

  Exceptions: CJMSException

```
CFioranoConnection(struct _FioranoConnection* pfc)throw (CJMSException *);
```

Creates CFioranoConnection object.

Parameters:

- Pfc – pointer to FioranoConnection structure.

Exceptions: CJMSException

Methods

```
mqcstring getUnifiedConnectionID() FMQCONST throw (CJMSException *);
```

Gets the CFioranoConnection’s unique connection ID.

Parameters: None

Returns: The connection id as const char*.

Exceptions: CJMSException

```
void setUnifiedConnectionID(mqcstring ucId) throw (CJMSException *);
```

Sets connection id for Unified connection. The unified connection id is different from Queue Connection id or topic connection id. Each Queue/Topic connection ids in a unified connection will have a common unified connection id.

Parameters:

- ucId – connection id as const char*.

Returns: void

Exceptions: CJMSException

```
CJMSSession *createQueueSession(const mqboolean isTransacted, const mqint ackMode)throw
(CJMSException *);
```

Creates Queue Session.

Parameters:

- isTransacted – If the session has to be transacted or not.
- ackMode: indicates whether the consumer or the client will acknowledge any messages it receives; ignored if the session is transacted. Legal values are `AUTO_ACKNOWLEDGE`, `CLIENT_ACKNOWLEDGE`, and `DUPS_OK_ACKNOWLEDGE`.

Returns: CJMSSession

Exceptions: CJMSException

```
CJMSSession *createTopicSession(const mqboolean isTransacted, const mqint ackMode) throw
(CJMSException *);
```

Creates Topic session.

Parameters:

- isTransacted – If the session has to be transacted or not.
- ackMode: indicates whether the consumer or the client will acknowledge any messages it receives; ignored if the session is transacted. Legal values are `AUTO_ACKNOWLEDGE`, `CLIENT_ACKNOWLEDGE`, and `DUPS_OK_ACKNOWLEDGE`.

Returns: CJMSSession

Exceptions: CJMSException

```
void setAdvisoryMessageListener(const CAdvisoryMsgListener *advMsgListener) throw
(CJMSException *);
```

Sets the Advisory Message listener to the unified connection. The user application should implement the “onAdvisoryMessage” method to receive any state change event that happens on this connection. For more information refer to section on Advisory Message listeners in C++.

Parameters: advMsgListener: the CAdvisoryMsgListener

Returns: Void

Exceptions: CJMSException

```
void setExceptionListener(const CExceptionListener *exceptionListener)throw (CJMSException
*);
```

Sets an exception listener for this unified connection. If a JMS provider detects a serious problem with a connection, it informs the connection's ExceptionListener, if one has been registered. It does this by calling the listener's onException method, passing it a JMSException object describing the problem. An exception listener allows a client to be notified of a problem asynchronously. Some connections only consume messages, so they would have no other way to learn their connection has failed.

Parameters:

exceptionListener: A pointer to the exception listener. If an exception listener is already registered with the connection, you can cancel the registration by specifying a null pointer instead.

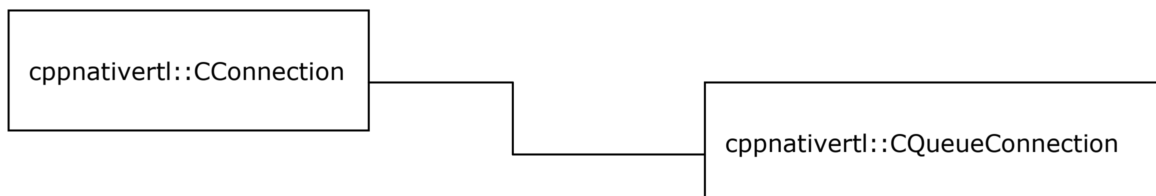
Returns: Void

Exceptions: CJMSException

CQueueConnection

A CQueueConnection object is an active connection to a point-to-point JMS provider. A client uses a CQueueConnection object to create one or more CQueueSession objects for producing and consuming messages.

Inheritance Hierarchy



Subclasses

None

Constructors

```
CQueueConnection(struct _QueueConnection *pqc) throw (CJMSException *);
```

Creates CQueueConnection Object.

Parameters:

- Pqc – pointer to _QueueConnection C structure.

Exceptions: CJMSException

Methods

```
CQueueSession *createQueueSession(mqboolean transacted, mqint acknowledgeMode) throw
(CJMSException *);
```

```
CJMSSession *createSession(const mqboolean transacted, const mqint acknowledgeMode)throw
(CJMSException *);
```

Creates a Queue Session Object.

Parameters:

- transacted – If the session has to be transacted or not.
- acknowledgeMode: indicates whether the consumer or the client will acknowledge any messages it receives; ignored if the session is transacted. Legal values are AUTO_ACKNOWLEDGE, CLIENT_ACKNOWLEDGE, and DUPS_OK_ACKNOWLEDGE.

Returns: CQueueSession

Exceptions: CJMSException

```
void setAdvisoryMessageListener(const CAdvisoryMsgListener *advMsgListener) throw
(CJMSException *);
```

Sets the Advisory Message listener to the connection. The user application should implement the ‘onAdvisoryMessage’ method to receive any state change event that happens on this connection. For more information refer to section on Advisory Message listeners in C++.

Parameters:

- advMsgListener: the CAdvisoryMsgListener

Returns: Void

Exceptions: CJMSException

```
mqboolean revalidate() throw (CJMSException *);
```

Revalidate and reconnect this connection with the MQ server. This method should be used only if the ‘Auto-Revalidation’ feature is disabled.

Parameters: None

Returns:

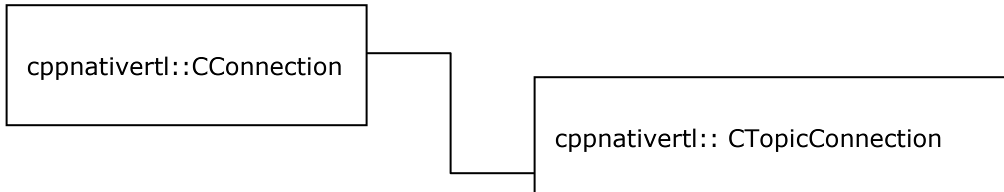
- 1 – If the revalidation is successful
- 0 – If the revalidation fails.

Exceptions: CJMSException

CTopicConnection

A CTopicConnection object is an active connection to a publish/subscribe JMS provider. A client uses a CTopicConnection object to create one or more CTopicSession objects for producing and consuming messages.

Inheritance Hierarchy



Subclasses

None

Constructors

```
CTopicConnection(struct _TopicConnection *ptc) throw (CJMSException *);
```

Creates CTopicConnection Object.

Parameters:

- ptc – pointer to _TopicConnection C structure.

Exceptions: CJMSException

Methods

```
CTopicSession *createTopicSession(mqboolean transacted,  mqint acknowledgeMode) throw
(CJMSException *);
```

```
CJMSSession *createSession(const mqboolean transacted, const mqint acknowledgeMode)throw
(CJMSException *);
```

Creates a Topic Session Object.

Parameters:

- transacted – If the session has to be transacted or not.
- acknowledgeMode: indicates whether the consumer or the client will acknowledge any messages it receives; ignored if the session is transacted. Legal values are `AUTO_ACKNOWLEDGE`, `CLIENT_ACKNOWLEDGE`, and `DUPS_OK_ACKNOWLEDGE`.

Returns: CTopicSession

Exceptions: CJMSException

```
void setAdvisoryMessageListener(const CAdvisoryMsgListener *advMsgListener)    throw
(CJMSException *);
```

Sets the Advisory Message listener to the connection. The user application should implement the 'onAdvisoryMessage' method to receive any state change event that happens on this connection. For more information refer to section on Advisory Message listeners in C++.

Parameters:

- advMsgListener – the CAdvisoryMsgListener

Returns: Void

Exceptions: CJMSException

```
mqboolean revalidate() throw (CJMSException *);
```

Revalidate and reconnect this connection with the MQ server. This method should be used only if the 'Auto-Revalidation' feature is disabled.

Parameters: None

Returns:

- 1 – If the revalidation is successful
- 0 – If the revalidation fails.

Exceptions: CJMSException

CJMSSession

CJMSSession is an abstract base class. A CJMSSession object is a single-threaded context for producing and consuming messages. It is considered a lightweight JMS object.

Inheritance Hierarchy

None

Subclasses

- CTopicSession
- CQueueSession
- CFioranoSession

Methods

```
virtual void close()throw (CJMSException *) = 0;
```

Closes the session. If the session is transacted, any transaction in progress is rolled back. This call will block until a receive call or message listener in progress has completed. A blocked message consumer's receive call returns NULL when this session is closed.

Parameters: None

Returns: Void

Exceptions: CJMSException

```
virtual void commit() throw (CJMSException *) = 0;
```

Commit all messages in the current transaction only if the session is set as transacted.

Parameters: None

Returns: Void

Exceptions: CJMSException

```
virtual CQueueBrowser *createBrowser(const CQueue *q) FMQCONST throw (CJMSException *) = 0;
```

Creates a `CQueueBrowser` object to look at the messages on the specified queue.

Parameters: CQueue object to access.

Returns: The CQueueBrowser object.

Exceptions: CJMSException

```
virtual CQueueBrowser *createBrowser(const CQueue *queue, mqcstring messageSelector) FMQCONST
throw (CJMSException *) = 0;
```

Creates a QueueBrowser object to look at the messages on the specified queue using a message selector.

Parameters:

- Queue: CQueue object to access.
- messageSelector: only messages with properties matching the message selector expression are delivered. A value of null or an empty string indicates that there is no message selector for the message consumer.

Returns: The CQueueBrowser object.

Exceptions: CJMSException

```
virtual CTopicSubscriber *createDurableSubscriber(const CTopic *topic,  mqcstring name)
FMQCONST throw (CJMSException *) = 0;
```

Creates a durable subscriber to the specified topic. If a client needs to receive all the messages published on a topic, including the ones published while the subscriber is inactive, it uses a durable TopicSubscriber. The JMS provider retains a record of this durable subscription and ensures that all messages from the topic's publishers are retained until they are acknowledged by this durable subscriber or they have expired.

Sessions with durable subscribers must always provide the same client identifier. Additionally, each client must specify a name that uniquely identifies (within client identifier) each durable subscription it creates. Only one session at a time can have a TopicSubscriber for a particular durable subscription.

A client can change an existing durable subscription by creating a durable TopicSubscriber with the same name, a new topic and/or message selector. Changing a durable subscriber is equivalent to unsubscribing (deleting) the old one and creating a new one.

Parameters:

- topic: the non-temporary Topic to subscribe to
- name: the name used to identify this subscription

Returns: The TopicSubscriber Object.

Exceptions: CJMSException.

```
virtual CTopicSubscriber *createDurableSubscriber(const CTopic *topic, mqcstring name,
mqcstring messageSelector, mqboolean noLocal) FMQCONST throw (CJMSException *) = 0;
```

Creates a durable subscriber to the specified topic, using a message selector and specifying whether messages published by its own connection should be delivered to it.

Parameters:

- topic: the non-temporary Topic to subscribe to
- name: the name used to identify this subscription
- messageSelector: only messages with properties matching the message selector expression are delivered. A value of null or an empty string indicates that there is no message selector for the message consumer.
- noLocal: if set, inhibits the delivery of messages published by its own connection

Returns: The TopicSubscriber Object.

Exceptions: CJMSException.

```
virtual CMapMessage *createMapMessage() FMQCONST throw (CJMSException *) = 0;
```

Creates a Map Message.

Parameters: None

Returns: The CMapMessage Object.

Exceptions: CJMSException.

```
virtual CMessage *createMessage()FMQCONST throw (CJMSException *) = 0;
```

Creates a message that has no body.

Parameters: None

Returns: The CMessage Object.

Exceptions: CJMSException.

```
virtual CObjectMessage *createObjectMessage() FMQCONST throw (CJMSException *) = 0;
```

Creates an Object Message.

Parameters: None

Returns: The CObjectMessage Object.

Exceptions: CJMSException.

```
virtual CQueue *createQueue(mqcstring queueName) FMQCONST throw (CJMSException *) = 0;
```

Creates a Queue destination in the messaging server with the specified queueName as mqctsring.

Parameters:

- queueName – name of the Queue (Destination) on the MQ Server.

Returns: The CQueue Object.

Exceptions: CJMSException.

```
virtual CStreamMessage *createStreamMessage() FMQCONST throw (CJMSException *) = 0;
```

Creates a CStreamMessage.

Parameters: None

Returns: The CStreamMessage Object.

Exceptions: CJMSException.

```
virtual CTemporaryQueue *createTemporaryQueue() FMQCONST throw (CJMSException *) = 0;
```

Creates a `TemporaryQueue` object. Its lifetime will be that of the `Connection` unless it is deleted earlier.

Parameters: None

Returns: The CTemporaryQueue Object.

Exceptions: CJMSException.

```
virtual CTemporaryTopic *createTemporaryTopic() FMQCONST throw (CJMSException *) = 0;
```

Creates a `TemporaryTopic` object. Its lifetime will be that of the `Connection` unless it is deleted earlier.

Parameters: None

Returns: The CTemporaryTopic Object.

Exceptions: CJMSException.

```
virtual CTextMessage *createTextMessage() FMQCONST throw (CJMSException *) = 0;
```

Creates a TextMessage with empty body.

Parameters: None

Returns: The CTextMessage Object.

Exceptions: CJMSException.

```
virtual CTextMessage *createTextMessage(mqcstring text) FMQCONST throw (CJMSException *) = 0;
```

Creates a TextMessage initialised with text provided.

Parameters:

- text: message in the form of const char*

Returns: The CTextMessage Object.

Exceptions: CJMSException.

```
virtual CTopic *createTopic(mqcstring topicName) FMQCONST throw (CJMSException *) = 0;
```

Creates a Topic destination in the messaging server.

Parameters:

- topicName: name of the Topic in the form of const char*

Returns: The CTopic Object.

Exceptions: CJMSException.

```
virtual CMessageListener *getMessageListener() FMQCONST  throw (CJMSException *) = 0;
```

Returns the session's distinguished message listener.

Parameters: None

Returns: The CMessageListener Object.

Exceptions: CJMSException.

```
virtual mqboolean getTransacted() FMQCONST throw (CJMSException *) = 0;
```

Indicates whether the session is transacted or not.

Parameters: None

Returns:

- 1 – Transacted
- 0: Non-Transacted.

Exceptions: CJMSException.

```
virtual void recover()throw (CJMSException *) = 0;
```

Stops message delivery in this session and restarts message delivery with the oldest unacknowledged message.

All consumers deliver messages in a serial order. Acknowledging a received message automatically acknowledges all messages that have been delivered to the client. This session must be a Non-Transacted one.

Restarting a session causes it to take the following actions:

- Stop message delivery
- Mark all messages that might have been delivered but not acknowledged as "redelivered"
- Restart the delivery sequence including all unacknowledged messages that had been previously delivered. Redelivered messages do not have to be delivered in exactly their original delivery order.

Parameters: None

Returns: Void

Exceptions: CJMSException.

```
virtual void rollback() throw (CJMSException *) = 0;
```

Rolls back any messages done in this transaction. The Session must be Transacted.

Parameters: None

Returns: Void

Exceptions: CJMSException.

```
virtual void setMessageListener(const CMessageListener *messageListener)throw (CJMSException
*) = 0;
```

Sets the session's distinguished message listener.

Parameters: messageListener

Returns: Void

Exceptions: CJMSException.

```
virtual void unsubscribe(mqcstring name) throw (CJMSException *) = 0;
```

Unsubscribes a durable subscription that has been created by a client. This method deletes the state being maintained on behalf of the subscriber by its provider. It is erroneous for a client to delete a durable subscription while there is an active `MessageConsumer` or `TopicSubscriber` for the subscription, or while a consumed message is part of a pending transaction or has not been acknowledged in the session.

Note: It is recommended to unsubscribe the subscription after closing the Subscriber.

Parameters:

- name – Subscription name used to identify the durable subscriber.

Returns: Void

Exceptions: CJMSException.

CFioranoSession

The CFioranoSession class provides a unified Session object and provides methods to create producers and consumers.

```
CFioranoMessageConsumer *createConsumer(const CDestination *dest) FMQCONST throw
(CJMSException *);
```

Creates a CFioranoMessageConsumer for the specified destination. Since `Queue` and `Topic` both inherit from `Destination`, they can be used in the destination parameter to create a `CFioranoMessageConsumer`.

Parameters:

- dest – destination to consume messages

Returns: CFioranoMessageConsumer

Exceptions: CJMSException.

```
CFioranoMessageConsumer *createConsumer(const CDestination *dest, mqcstring messageSelector,
mqboolean noLocal) FMQCONST  throw (CJMSException *);
```

Creates MessageConsumer for the specified destination, using a message selector. This method can specify whether messages published by its own connection should be delivered to it, if the destination is a topic.

Since Queue and Topic both inherit from Destination, they can be used in the destination parameter to create a MessageConsumer. A client uses a MessageConsumer object to receive messages that have been published to a destination. In some cases, a connection may both publish and subscribe to a topic. The consumer NoLocal attribute allows a consumer to inhibit the delivery of messages published by its own connection. The default value for this attribute is False. The noLocal value must be supported by destinations that are topics.

Parameters:

- dest – destination to consume messages
- messageSelector: only messages with properties matching the message selector expression are delivered. A value of null or an empty string indicates that there is no message selector for the message consumer.
- NoLocal: if true, and the destination is a topic, inhibits the delivery of messages published by its own connection. The behavior for NoLocal is not specified if the destination is a queue.

Returns: CFioranoMessageConsumer

Exceptions: CJMSException.

```
CFioranoMessageProducer *createProducer(const CDestination *dest) FMQCONST throw
(CJMSException *);
```

Creates a MessageProducer to send messages to the specified destination. A client uses a CFioranoMessageProducer object to send messages to a destination. Since Queue and Topic both inherit from Destination, they can be used in the destination parameter to create a CFioranoMessageProducer object.

Parameters:

- dest – destination to publish messages

Returns: CFioranoMessageProducer

Exceptions: CJMSException.

CQueueSession

The CQueueSession class provides methods for creating CQueueReceiver, CQueueSender, CQueueBrowser, and CTemporaryQueue objects.

```
CQueueReceiver *createReceiver(const CQueue *queue) FMQCONST throw (CJMSException *);
```

Creates a `CQueueReceiver` object to receive messages from the specified queue.

Parameters:

- queue – Queue destination to access.

Returns: CQueueReceiver object

Exceptions: CJMSException.

`CQueueReceiver *createReceiver(const CQueue *queue, mqcstring messageSelector) FMQCONST throw (CJMSException *);`

Creates a `CQueueReceiver` object to receive messages from the specified queue using a message selector.

Parameters:

- queue – Queue destination to access.
- messageSelector: only messages with properties matching the message selector expression are delivered. A value of null or an empty string indicates that there is no message selector for the message consumer.

Returns: CQueueReceiver object

Exceptions: CJMSException.

`CQueueSender *createSender(const CQueue *queue) FMQCONST throw (CJMSException *);`

Creates a `CQueueSender` object to send messages to the specified queue.

Parameters:

- queue – Queue destination to access, or null if this is an unidentified producer

Returns: CQueueSender object

Exceptions: CJMSException.

CTopicSession

The CTopicSession class provides methods for creating CTopicPublisher, CTopicSubscriber, and CTemporaryTopic objects. It also provides a method for deleting its client's durable subscribers.

`CTopicPublisher *createPublisher(const CTopic *topic) FMQCONST throw (CJMSException *);`

Creates a publisher for the specified topic. A client uses a TopicPublisher object to publish messages on a topic. Each time a `client` creates a TopicPublisher on a topic, it defines a new sequence of messages that have no ordering relationship with the messages it has previously sent.

Parameters:

- topic – Topic destination to publish, or null if it is an unidentified producer

Returns: CTopicPublisher

Exceptions: CJMSException.

`CTopicSubscriber *createSubscriber(const CTopic *topic) FMQCONST throw (CJMSException *);`

Creates a nondurable subscriber to the `specified` topic. A client uses a `TopicSubscriber` object to receive messages that have been published to a topic. Regular `TopicSubscriber` objects are not durable. They receive only messages that are published while they are active.

In some cases, a connection may both publish and subscribe to a topic. The subscriber NoLocal attribute allows a subscriber to inhibit the delivery of messages published by its own connection. The default value for this attribute is false.

Parameters:

- topic – Topic destination to subscribe to

Returns: CTopicSubscriber

Exceptions: CJMSException.

```
CTopicSubscriber *createSubscriber(const CTopic *topic, mqcstring messageSelector,mqboolean
noLocal) FMQCONST throw (CJMSException *);
```

Creates a nondurable subscriber to the specified topic, using a message selector or specifying whether messages published by its own connection should be delivered to it. A client uses a TopicSubscriber object to receive messages that have been published to a topic.

Regular TopicSubscriber objects are not durable. They receive only messages that are published while they are active.

Messages filtered out by a subscriber's message selector will never be delivered to the subscriber. From the subscriber's perspective, they do not exist.

In some cases, a connection may both publish and subscribe to a topic. The subscriber NoLocal attribute allows a subscriber to inhibit the delivery of messages published by its own connection. The default value for this attribute is false.

Parameters:

- topic – Topic destination to subscribe to
- messageSelector: only messages with properties matching the message selector expression are delivered. A value of null or an empty string indicates that there is no message selector for the message consumer.
- noLocal: if set, inhibits the delivery of messages published by its own connection

Returns: CTopicSubscriber

Exceptions: CJMSException.

CMessageProducer

A client uses a CMessageProducer object to send messages to a destination. A CMessageProducer object is created by passing a Destination object to a message-producer creation method supplied by a session.

A client also has the option of creating a message producer without supplying a destination. In this case, a destination must be provided with every send operation. A typical use for this kind of message producer is to send replies to requests using the request's JMSReplyTo destination.

A client can specify a default delivery mode, priority, and time to live for messages sent by a message producer. It can also specify the delivery mode, priority, and time to live for an individual message.

A client can specify a time-to-live value in milliseconds for each message it sends.

Inheritance Hierarchy

None

Subclasses

CMessageProducer is an abstract base class with the following derived classes.

- CQueueSender
- CTopicPublisher
- CFioranoMessageProducer

Methods

```
virtual void close()throw (CJMSException *)=0;
```

Closes the message producer.

Parameters: None

Returns: void

Exceptions: CJMSException

```
virtual mqint getDeliveryMode() FMQCONST throw (CJMSException *)  = 0;
```

Gets the producer's default delivery mode.

Parameters: None

Returns: Delivery mode on this producer. It can be one of NON_PERSISTENT (1) or PERSISTENT (2)

Exceptions: CJMSException

```
virtual CDestination *getDestination() FMQCONST throw (CJMSException *)  = 0;
```

Parameters: None

Returns: This producer’s destination.

Exceptions: CJMSException

```
virtual mqboolean getDisableMesageTimestamp() FMQCONST throw (CJMSException *)  = 0;
```

Gets an indication of whether message timestamps are disabled.

Parameters: None

Returns: An indication of whether message timestamps are disabled

Exceptions: CJMSException

```
virtual mqboolean getDisableMessageID() FMQCONST throw (CJMSException *)  = 0;
```

Gets an indication of whether message IDs are disabled.

Parameters: None

Returns: An indication of whether message IDs are disabled. By default this is disabled for FioranoMQ.

Exceptions: CJMSException

```
virtual mqint getPriority() FMQCONST throw (CJMSException *)  = 0;
```

Gets the producer's default priority.

Parameters: None

Returns: The message priority for this message producer

Exceptions: CJMSException

```
virtual mqlong getTimeToLive() FMQCONST throw (CJMSException *)  = 0;
```

Gets the default length of time in milliseconds from its dispatch time that a produced message should be retained by the message system.

Parameters: None

Returns: The message time to live in milliseconds; zero is unlimited

Exceptions: CJMSException

```
virtual void setDeliveryMode(mqint  deliveryMode) throw (CJMSException *)  = 0;
```

Sets the producer's default delivery mode. Delivery mode is set to PERSISTENT by default.

Parameters:

- deliveryMode: the message delivery mode for this message producer; legal values are NON_PERSISTENT(1) and PERSISTENT(2)

Returns: Void

Exceptions: CJMSException

```
virtual void setDisableMessageID(mqboolean value) throw (CJMSException *)  = 0;
```

Sets whether message IDs are disabled. Disabled for FioranoMQ by default.

Parameters:

- `value`: indicates if message IDs are disabled

Returns: Void

Exceptions: CJMSException

```
virtual void setDisableMessageTimestamp(mqboolean value) throw (CJMSException *)  = 0;
```

Sets whether message timestamps are disabled.

Parameters:

- `value`: indicates if message timestamps are disabled

Returns: Void

Exceptions: CJMSException

```
virtual void setPriority(mqint defaultPriority)   throw (CJMSException *)  = 0;
```

Sets the producer's default priority. Priority is set to 4 by default.

Parameters:

- `defaultPriority`: the message priority for this message producer; must be a value between 0 and 9

Returns: Void

Exceptions: CJMSException

```
virtual void setTimeToLive(mqlong timeToLive) throw (CJMSException *)  = 0;
```

Sets the default length of time in milliseconds from its dispatch time that a produced message should be retained by the message system. Time to live is set to zero by default.

Parameters:

- `timeToLive`: the message time to live in milliseconds; zero is unlimited

Returns: Void

Exceptions: CJMSException

```
virtual void send(const CDestination *dest, CMessage *msg) FMQCONST throw (CJMSException *)
= 0;
```

Sends a message to a destination for an unidentified message producer. Uses the CMessageProducer's default delivery mode, priority, and time to live.

Typically, a message producer is assigned a destination at creation time; however, the JMS API also supports unidentified message producers, which require that the destination be supplied every time a message is sent.

Parameters:

- dest: the destination to send this message
- msg: the message to send

Returns: void

Exceptions: CJMSException

```
virtual void send(const CDestination *dest, CMessage *msg, mqint deliveryMode, mqint
priority, mqint timeToLive) FMQCONST throw (CJMSException *)  = 0;
```

Sends a message to a destination for an unidentified message producer, specifying delivery mode, priority and time to live.

Parameters:

- dest: the destination to send this message
- ms: the message to send
- deliveryMode: the delivery mode to use
- priority: the priority for this message
- timeToLive: the message's lifetime (in milliseconds)

Returns: void

Exceptions: CJMSException

```
virtual void send(CMessage *msg) FMQCONST throw (CJMSException *)  = 0;
```

Sends a message using the MessageProducer's default delivery mode, priority, and time to live.

Parameters:

- msg: the message to send

Returns: void

Exceptions: CJMSException

```
virtual void send(CMessage *msg,mqint deliveryMode, mqint priority, mqlong timeToLive)
FMQCONST throw (CJMSException *)  = 0;
```

Sends a message to the destination, specifying delivery mode, priority, and time to live.

Parameters:

- msg: the message to send

- deliveryMode: the delivery mode to use
- priority: the priority for this message
- timeToLive: the message's lifetime (in milliseconds)

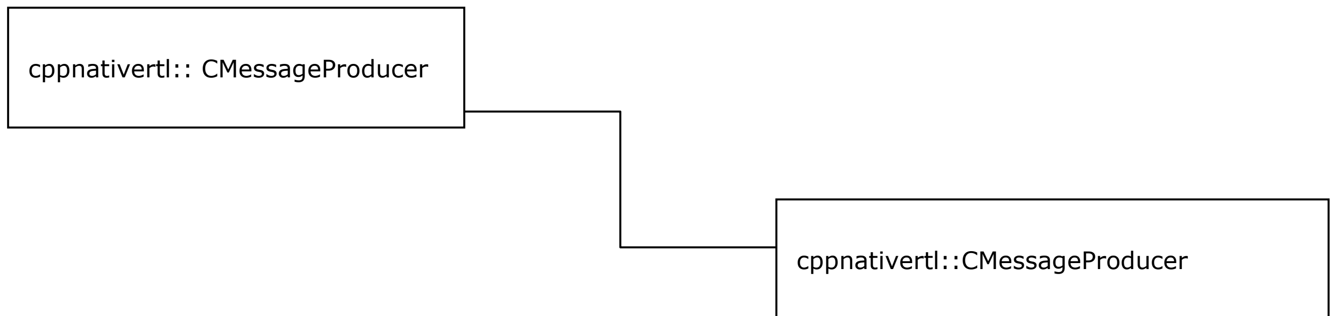
Returns: void

Exceptions: CJMSException

CFioranoMessageProducer

The CFioranoMessageProducer class provides a unified MessageProducer object which can be used as QueueSender or TopicPublisher.

Inheritance Hierarchy



Subclasses

None

Constructors

```
CFioranoMessageProducer();
```

Default constructor.

Parameters: None

```
CFioranoMessageProducer(struct _FioranoMessageProducer* producer);
```

Parameters:

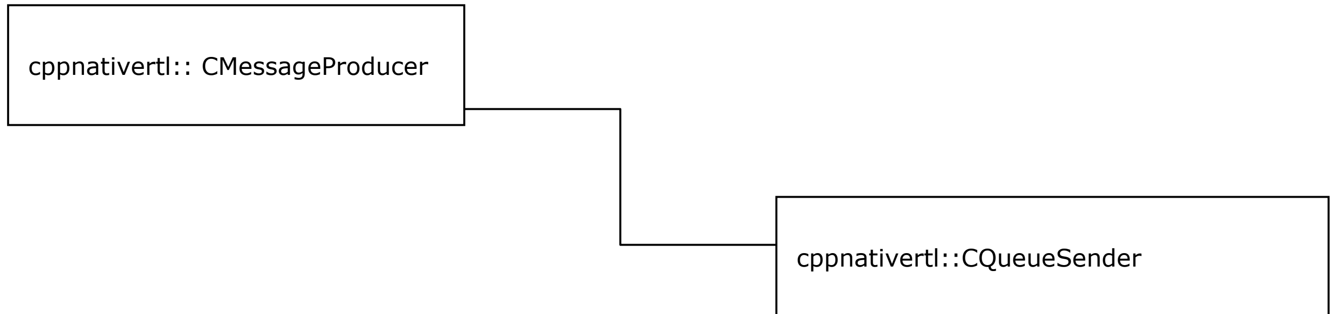
- producer – FioranoMessageProducer structure defined in C runtime.

(CFioranoMessageProducer defines all the methods in CMessageProducer class.)

CQueueSender

A client uses a CQueueSender object to send messages to a queue. Normally, the Queue is specified when a CQueueSender is created. An exception will be thrown if an attempt is made to use the send methods for an unidentified CQueueSender.

Inheritance Hierarchy



Subclasses

None

Constructors

```
CQueueSender(struct _QueueSender *queueSender);
```

Parameters:

- queueSender – QueueSender structure defined in C runtime.

Methods

(CQueueSender defines all the methods in CMessageProducer class, with the following additional APIs)

```
CQueue *getQueue() FMQCONST throw (CJMSException *);
```

Parameters: None

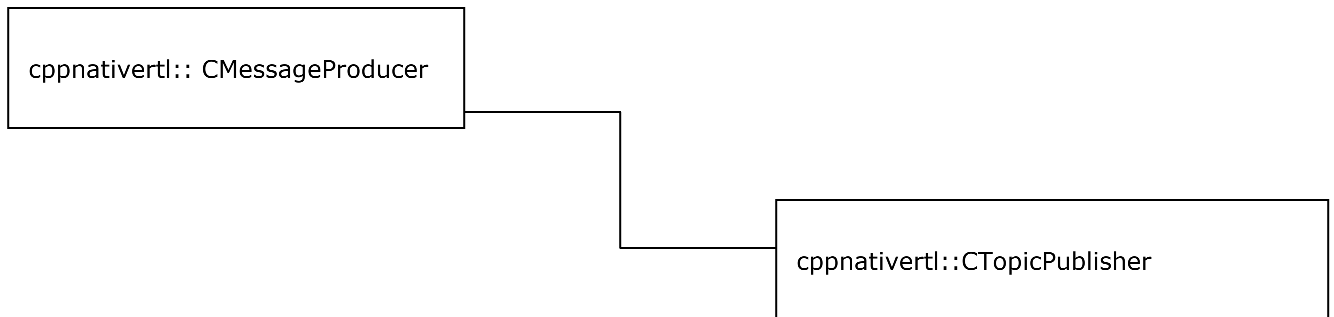
Returns: CQueue

Exceptions: CJMSException

CTopicPublisher

A client uses a CTopicPublisher object to publish messages on a topic. A CTopicPublisher object is the publish-subscribe form of a message producer. Normally, the Topic is specified when a TopicPublisher is created. An exception will be thrown if an attempt is made to use the publish methods for an unidentified CTopicPublisher.

Inheritance Hierarchy



Subclasses

None

Constructors

```
CTopicPublisher(struct _TopicPublisher *tpub);
```

Parameters:

- tpub – TopicPublisher structure defined in C runtime.

Methods

(CTopicPublisher defines all the methods in CMessageProducer class, with the following additional APIs)

```
CTopic *getTopic() FMQCONST throw (CJMSException *);
```

Parameters: None

Returns: CTopic

Exceptions: CJMSException

CQueueRequestor

The `CQueueRequestor` helper class simplifies making service requests.

The `CQueueRequestor` constructor is given a non-transacted `CQueueSession` and a destination Queue. It creates a `TemporaryQueue` for the responses and provides a request method that sends the request message and waits for its reply.

Inheritance Hierarchy

None

Subclasses

None

Constructor

```
CQueueRequestor(CQueueSession *qs,CQueue *queue) throw (CJMSException *);
```

Constructor for the CQueueRequestor class. This implementation assumes the session parameter to be non-transacted, with a delivery mode of either `AUTO_ACKNOWLEDGE` or `DUPS_OK_ACKNOWLEDGE`.

Parameters:

- qs – The CQueueSession the queue belongs to
- queue – The Queue to perform request/reply call on.

Exceptions: CJMSException.

Methods

```
void close() throw (CJMSException *);
```

Closes the CQueueRequestor. Note that the CQueueSession is not closed in this call.

Parameters: None

Returns: Void

Exceptions: CJMSException.

```
CMessage *request(CMessage *msg) FMQCONST throw (CJMSException *);
```

Sends a request and waits for a reply. The temporary queue is used for the `JMSReplyTo` destination, and only one reply per request is expected.

Parameters:

- msg – The message to send

Returns: CMessage – the reply message

Exceptions: CJMSException.

```
CMessage *request(CMessage *msg, const mqlong timeout) FMQCONST throw (CJMSException *);
```

Send a request and wait for a reply within the specified timeout. The temporary topic is used for replyTo; the first reply is returned and the subsequent replies are discarded.

Parameters:

- msg – The message to send
- timeout: the time for which the requestor will wait for a reply

Returns:

- CMessage – the reply message

Exceptions: CJMSException.

CTopicRequestor

The `CTopicRequestor` helper class simplifies making service requests. The `CTopicRequestor` constructor is given a non-transacted `TopicSession` and a destination `Topic`. It creates a `TemporaryTopic` for the responses and provides a `request` method that sends the request message and waits for its reply.

Inheritance Hierarchy

None

Subclasses

None

Constructors

```
CTopicRequestor(CTopicSession *ts,CTopic *topic) throw (CJMSException *);
```

Constructor for the TopicRequestor class. This implementation assumes the session parameter to be non-transacted, with a delivery mode of either AUTO_ACKNOWLEDGE or DUPS_OK_ACKNOWLEDGE.

Parameters:

- Ts: The CTopicSession the topic belongs to
- Topic: The Topic destination to perform request/reply call on.

Exceptions: CJMSException.

Methods

```
void close() throw (CJMSException *);
```

Closes the CTopicRequestor. Note that the CTopicSession is not closed in this call.

Parameters: None

Returns: Void

Exceptions: CJMSException.

```
CMessage *request(CMessage *msg) FMQCONST  throw (CJMSException *);
```

Sends a request and waits for a reply. The temporary queue is used for the `JMSReplyTo` destination, and only one reply per request is expected.

Parameters:

- msg: The message to send

Returns:

- CMessage: the reply message

Exceptions: CJMSException.

```
CMessage *request(CMessage *msg, const mqlong timeout) FMQCONST  throw (CJMSException *);
```

Send a request and wait for a reply within the specified timeout. The temporary topic is used for replyTo; the first reply is returned and the subsequent replies are discarded.

Parameters:

- Msg: The message to send
- Timeout: the time for which the requestor will wait for a reply

Returns:

- CMessage: the reply message

Exceptions: CJMSException.

CMessageConsumer

A client uses a `CMessageConsumer` object to receive messages from a destination. A `CMessageConsumer` object is created by passing a `Destination` object to a message-consumer creation method supplied by a session.

Inheritance Hierarchy

None

Subclasses

- CTopicSubscriber
- CQueueReceiver

- CFioranoMessageConsumer

Methods

```
virtual void close() throw (CJMSException *) = 0;
```

Closes the message consumer. This call blocks until a receive or message listener in progress has completed. A blocked message consumer receive call returns null when this message consumer is closed.

> **Parameters:** None
>
> **Returns:** None
>
> **Exceptions:** CJMSException

```
virtual CMessageListener *getMessageListener() FMQCONST throw (CJMSException *) = 0;
```

Gets the message consumer's `MessageListener`.

> **Parameters:** None
>
> **Returns:** The listener for the consumer
>
> **Exceptions:** CJMSException

```
virtual mqcstring getMessageSelector() FMQCONST throw (CJMSException *) = 0;
```

Gets this message consumer's message selector expression.

> **Parameters:** None
>
> **Returns:** Message consumer's message selector, or null if no message selector exists for the message consumer (that is, if the message selector was not set or was set to null or the empty string)
>
> **Exceptions:** CJMSException

```
virtual void setMessageListener(const CMessageListener  *messageListener) throw
(CJMSException *) = 0;
```

Sets the message consumer's MessageListener. Setting the message listener to null is the equivalent of unsetting the message listener for the message consumer.

The effect of calling MessageConsumer.`setMessageListener` while messages are being consumed by an existing listener or the consumer is being used to consume messages synchronously is undefined.

> **Parameters:**

  - `messageListener`: the listener to which the messages are to be delivered

Returns: Void

Exceptions: CJMSException

```
virtual CMessage *receive() FMQCONST throw (CJMSException *) = 0;
```

Receives the next message produced for this message consumer. This call blocks indefinitely until a message is produced or until this message consumer is closed.

Parameters: None

Returns: The next message produced for this message consumer, or null if this message consumer is concurrently closed

Exceptions: CJMSException

```
virtual CMessage *receive(mqlong timeout) FMQCONST throw (CJMSException *) = 0;
```

Receives the next message that arrives within the specified timeout interval. This call blocks until a message arrives, the timeout expires, or this message consumer is closed. A timeout of zero never expires, and the call blocks indefinitely.

Parameters:

- `timeout`: the timeout value (in milliseconds)

Returns: The next message produced for this message consumer, or null if the timeout expires or this message consumer is concurrently closed

Exceptions: CJMSException

```
virtual CMessage *receiveNoWait() FMQCONST throw (CJMSException *) = 0;
```

Receives the next message if one is immediately available.

Parameters: None

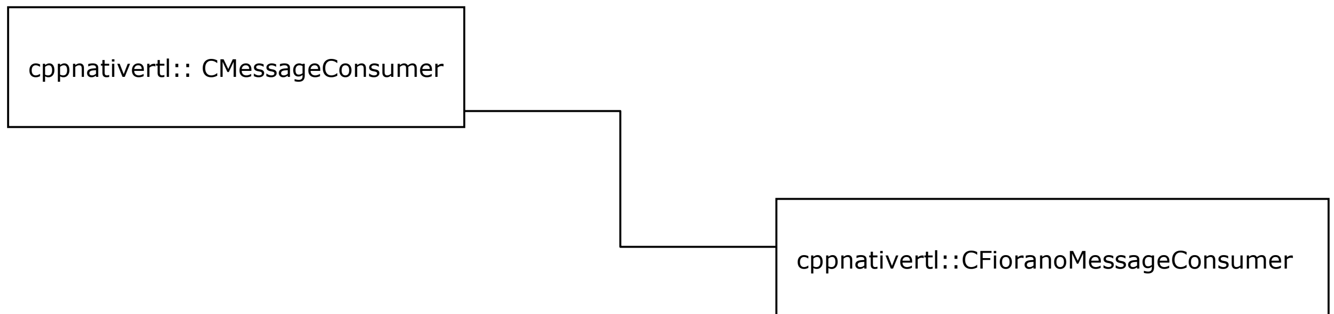
Returns: The next message produced for this message consumer, or null if one is not available.

Exceptions: CJMSException

CFioranoMessageConsumer

A client uses a `CFioranoMessageConsumer` (unified message consumer) object to receive messages that have been published to a topic/queue. A `CFioranoMessageConsumer` object is the publish/subscribe form of a message consumer.

Inheritance Hierarchy



Subclasses

None

Constructors

```
CFioranoMessageConsumer(struct _FioranoMessageConsumer *consumer);
```

Creates a CFioranoMessageConsumer Object.

Parameters: FioranoMessageConsumer structure defined in C runtime library.

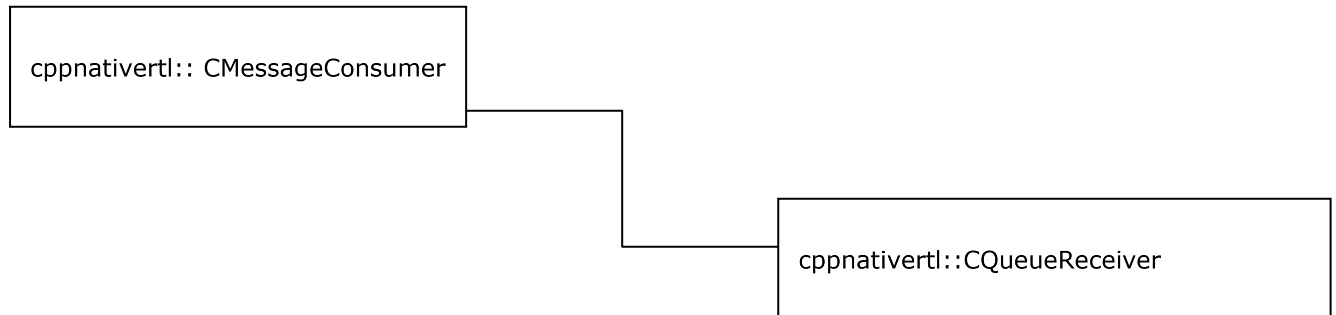
Methods

(CFioranoMessageConsumer defines all the methods in CMessageConsumer class.)

CQueueReceiver

A client uses a `QueueReceiver` object to receive messages that have been delivered to a queue.

Inheritance Hierarchy



Subclasses

None

Constructors

```
CQueueReceiver(struct _QueueReceiver *queueRcvr);
```

Creates a CQueueReceiver object.

Parameters: QueueReceiver structure defined in C runtime library.

Methods

(CQueueReceiver defines all the methods in CMessageConsumer class, with the following additional APIs)

```
CQueue *getQueue() throw (CJMSException *);
```

Gets the `Queue` associated with this queue receiver.

Parameters: None

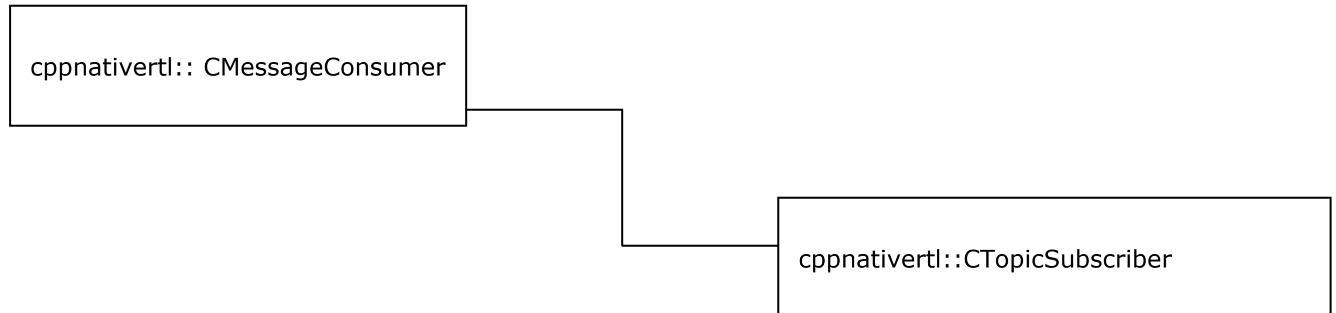
Returns: The CQueue object to which this receiver is associated with.

Exceptions: CJMSException

CTopicSubscriber

A client uses a `CTopicSubscriber` object to receive messages that have been published to a topic. A `CTopicSubscriber` object is the publish/subscribe form of a message consumer.

Inheritance Hierarchy



Subclasses

None

Constructors

```
CTopicSubscriber(struct _TopicSubscriber *tsub);
```

Creates a CTopicSubscriber Object.

Parameters: TopicSubscriber structure defined in C runtime library.

Methods

(CTopicSubscriber defines all the methods in CMessageConsumer class, with the following additional APIs)

```
mqboolean getNoLocal() FMQCONST throw (CJMSException *);
```

Gets the `NoLocal` attribute for this subscriber. The default value for this attribute is false.

Parameters: None

Returns: true if locally published messages are being inhibited

Exceptions: CJMSException

```
CTopic *getTopic() throw (CJMSException *);
```

Gets the `Topic` associated with this subscriber.

Parameters: None

Returns: The CTopic object to which this subscriber is associated with.

Exceptions: CJMSException

CTopicMetaData

CTopicMetaData class represents the metadata information for a Topic destination as it is stored in the JNDI store. CTopicMetaData is used for creating Topics.

Inheritance Hierarchy

None

Subclasses

None

Constructors

```
CTopicMetaData()throw (CJMSException *);
```

Creates the CTopicMetaData object.

Parameters: None

```
mqboolean enableCompression(const mqint compressLevel, const mqint compressStrategy) throw
(CJMSException *);
```

Enables Message Compression for a given message with specified compression level and compression strategy.

Parameters:

- m_nCompressionStrategy - Compression Strategy to be set

Returns: mqboolean

Exceptions: CJMSException

Methods

```
TopicMetaData getTopicMetaData() FMQCONST;
```

Parameters: None

Returns: C runtime TopicMetaData structure.

Exceptions: None

```
mqboolean setName(mqcstring metaDataName) throw (CJMSException *);
```

Sets name of the topic in the Topic's metadata.

Parameters:

- metaDataName – Topic's name

Returns:

Exceptions: CJMSException

```
mqcstring getName() FMQCONST throw (CJMSException *);
```

Parameters: None

Returns: Topic's name.

Exceptions: CJMSException

```
void setDescription(mqcstring metaDataDescription)throw (CJMSException *);
```

Sets description name for the topic.

Parameters:

- metaDataDescription – Description for the Topic.

Returns: None

Exceptions: CJMSException

```
mqcstring getDescription() FMQCONST throw (CJMSException *);
```

Parameters: None

Returns: Description name for the Topic.

Exceptions: CJMSException

CQueueMetaData

CQueueMetaData class represents the metadata information for a Queue destination as it is stored in the JNDI store. CQueueMetaData is used for creating Queues.

Inheritance Hierarchy

None

Subclasses

None

Constructors

```
CQueueMetaData()throw (CJMSException *);
```

Creates the CQueueMetaData object.

Parameters: None

```
mqboolean enableCompression(const mqint compressLevel, const mqint compressStrategy) throw
(CJMSException *);
```

Purpose: Enables Message Compression for a given message with specified compression level and compression strategy.

Parameters:

- m_nCompressionStrategy - Compression Strategy to be set

Returns: mqboolean

Exceptions: CJMSException

Methods

```
QueueMetaData getQueueMetaData() FMQCONST;
```

Parameters: None

Returns: C runtime QueueMetaData structure.

Exceptions: None

```
mqboolean setName(mqcstring metaDataName) throw (CJMSException *);
```

Sets name of the queue in the Queue's metadata.

Parameters:

- metaDataName – Queue�much's name

Returns:

Exceptions: CJMSException

```
mqcstring getName() FMQCONST throw (CJMSException *);
```

Parameters: None

Returns: Queue's name.

Exceptions: CJMSException

```
void setDescription(mqcstring metaDataDescription)throw (CJMSException *);
```

Sets description name for the queue.

Parameters:

- metaDataDescription – Description for the Queue.

Returns: None

Exceptions: CJMSException

```
mqcstring getDescription() FMQCONST throw (CJMSException *);
```

Parameters: None

Returns: Description name for the Queue.

Exceptions: CJMSException

CDestination

A CDestination object encapsulates a provider-specific address.

Inheritance Hierarchy

None

Subclasses

Base class for:

- CTopic
- CQueue

Constructors

```
CDestination(struct _Destination *dest)
```

Parameters:

- dest – Destination structure defined in C runtime.

Methods

```
mqboolean isQueue() FMQCONST throw (CJMSException *)
```

Checks whether the destination object is a queue or not.

Parameters: None

Returns: mqboolean value for success or failure from the server.

Exceptions: CJMSException

```
mqboolean isTopic() FMQCONST throw (CJMSException *)
```

Checks whether the destination object is a topic or not.

Parameters: None

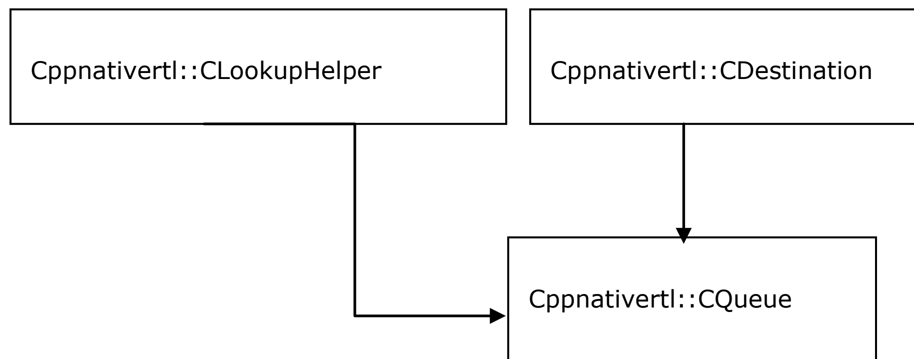
Returns: mqboolean value for success or failure from the server.

Exceptions: CJMSException

CQueue

A CQueue object encapsulates a provider-specific queue name. It is the way a client specifies the identity of a queue to JMS API methods. For those methods that use a CDestination as a parameter, a CQueue object is used as an argument.

Inheritance Hierarchy



Subclasses

Base class for:

- CTemporaryQueue

Constructors

```
CQueue(struct _Destination *dest);
```

Parameters:

- dest – Destination structure defined in C runtime.

```
CQueue(mqstring queueName) throw (CJMSException *);
```

Creates a new Queue with the specified name without using JNDI lookup.

Parameters:

- queueName - Name of the Queue to be created

Exceptions: CJMSException

Methods

```
mqcstring getQueueName() FMQCONST  throw (CJMSException *)
```

Gets the name of this queue. Clients that depend upon the name are not portable.

Parameters: None

Returns: The queue name

Exceptions: CJMSException

```
mqcstring toString() FMQCONST  throw (CJMSException *)
```

Returns a mqcstring representation of this object.

Parameters: None

Returns: The provider-specific identity values for this queue.

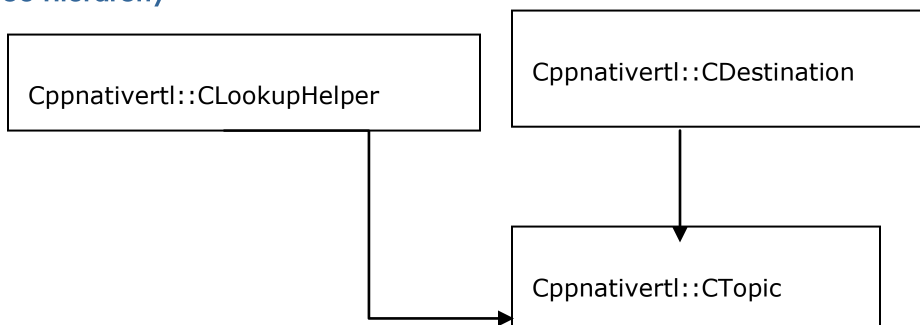
Exceptions: CJMSException

(CQueue defines `getLookupObjectType()` method of CLookupHelper base class)

CTopic

A CTopic object encapsulates a provider-specific topic name. It is the way a client specifies the identity of a topic to JMS API methods. For those methods that use a CDestination as a parameter, a CTopic object may used as an argument.

Inheritance Hierarchy



Subclasses

Base class for:

- CTemporaryTopic

Constructors

`CTopic(struct _Destination *dest);`

Parameters:

- dest – Destination structure defined in C runtime.

`CTopic(mqstring topicName) throw (CJMSException *);`

Creates a new Topic with the specified name without using JNDI lookup.

Parameters:

- topicName - Name of the Topic to be created

Exceptions: CJMSException

Methods

`mqcstring getTopicName() FMQCONST   throw (CJMSException *)`

Gets the name of this topic. Clients that depend upon the name are not portable.

Parameters: None

Returns: The topic name

Exceptions: CJMSException

`mqcstring toString() FMQCONST throw (CJMSException *)`

Returns a mqcstring representation of this object.

Parameters: None

Returns: The provider-specific identity values for this topic.

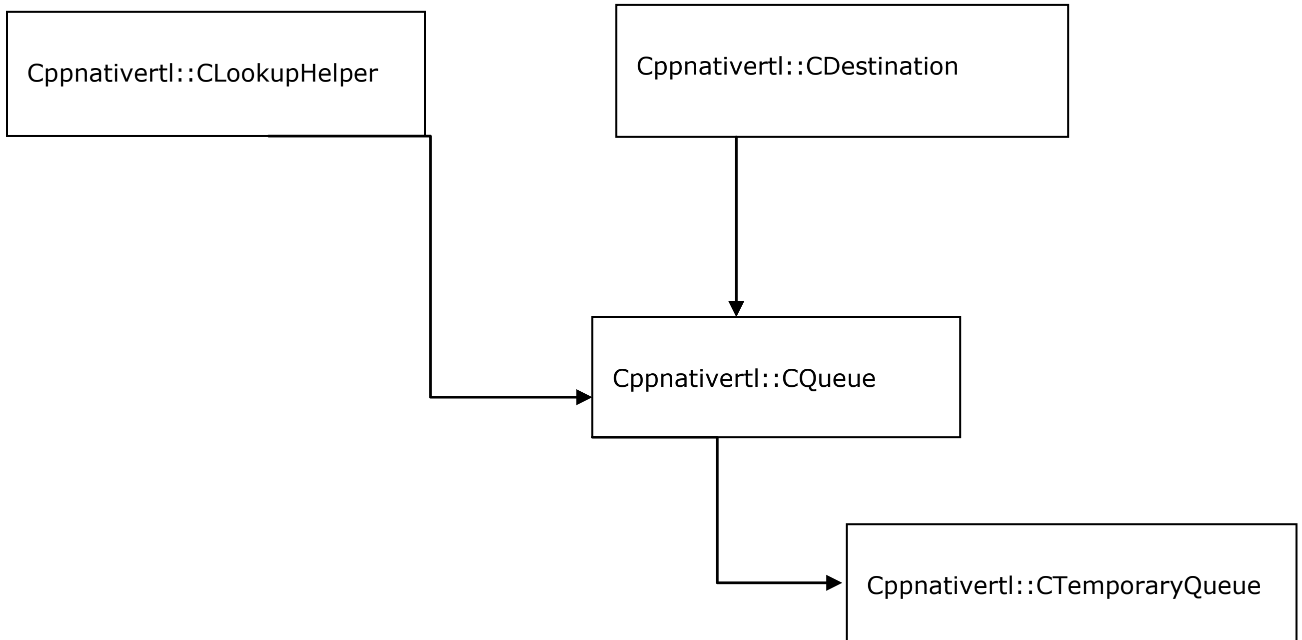
Exceptions: CJMSException

(CTopic defines `getLookupObjectType()` method of CLookupHelper base class)

CTemporaryQueue

A CTemporaryQueue object is a unique CQueue object created for the duration of a CConnection. It is a system-defined queue that can be consumed only by the CConnection that created it.

Inheritance Hierarchy



Subclasses

None

Constructors

```
CTemporaryQueue(struct _TemporaryQueue *temporaryQueue)
```

Parameters:

- temporaryQueue – TemporaryQueue structure defined in C runtime.

Methods

```
void remove() throw (CJMSException *);
```

Deletes this temporary queue. If there are existing receivers still using it, a `CJMSException` will be thrown.

Parameters: None

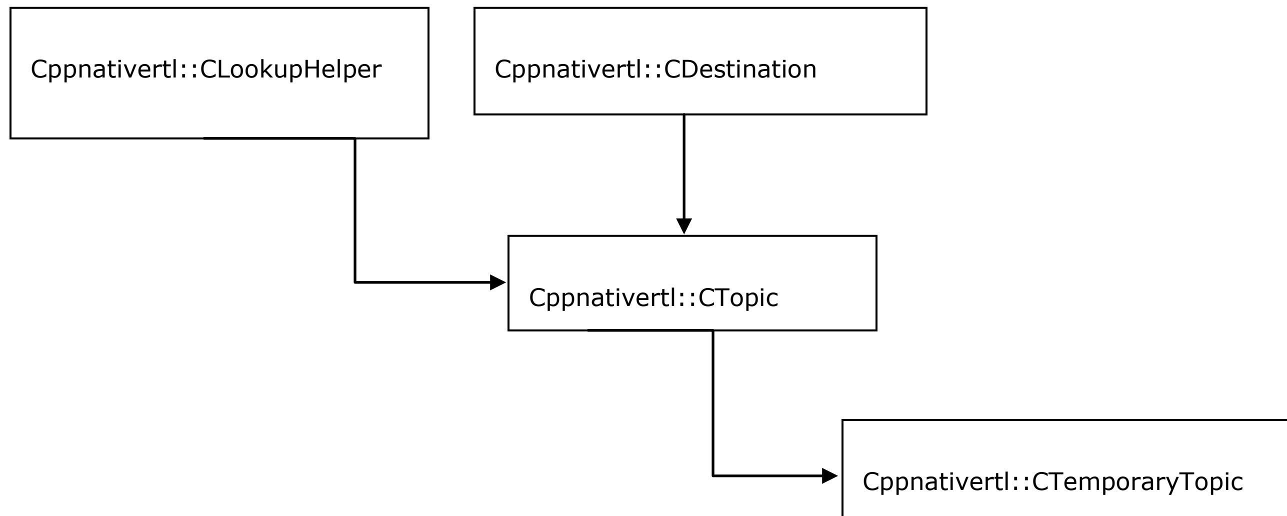
Returns: void

Exceptions: CJMSException

CTemporaryTopic

A CTemporaryTopic object is a unique CTopic object created for the duration of a CConnection. It is a system-defined topic that can be consumed only by the CConnection that created it.

Inheritance Hierarchy



Subclasses

None

Constructors

```
CTemporaryTopic(struct _TemporaryTopic *temporaryTopic)
```

Parameters:

- temporaryTopic – TemporaryTopic structure defined in C runtime.

Methods

```
void remove() throw (CJMSException *);
```

Deletes this temporary topic. If there are existing subscribers still using it, a `CJMSException` will be thrown.

Parameters: None

Returns: void

Exceptions: CJMSException

CProperty

The CProperty class wraps the 'value' part of a message property, which includes the value type, size, and the value itself.

Inheritance Hierarchy

None

Subclasses

None

Constructors

`CProperty(struct _Property* m_property)`

Parameters:

- m_property – Property structure defined in C runtime.

Methods

`PropertyIndex getPropertyType() FMQCONST throw (CJMSException*)`

Gets the type of the property.

Parameters: None

Returns: It returns an enum PropertyIndex. PropertyIndex is an enum of Byte, Short, Int, Float, Double, Long, ByteArray, String, Bool, Char, NullObj, Invalid, AnySerializable indices.PropertyIndex enum is defined in basic_datatypes.h file in $FMQ_DIR/clients/c/native/include directory.

Exceptions: `CJMSException`

`mqbyteArray getPropertyValue() FMQCONST throw (CJMSException*)`

Gets the value of the property. Despite the type of property being set, this function returns the value as a mqbyteArray (ie. char*) type.

Parameters: None

Returns: mqbyteArray

Exceptions: `CJMSException`

`mqint getPropertySize() FMQCONST throw (CJMSException*)`

Gets the size of the value as number of bytes.

Parameters: None

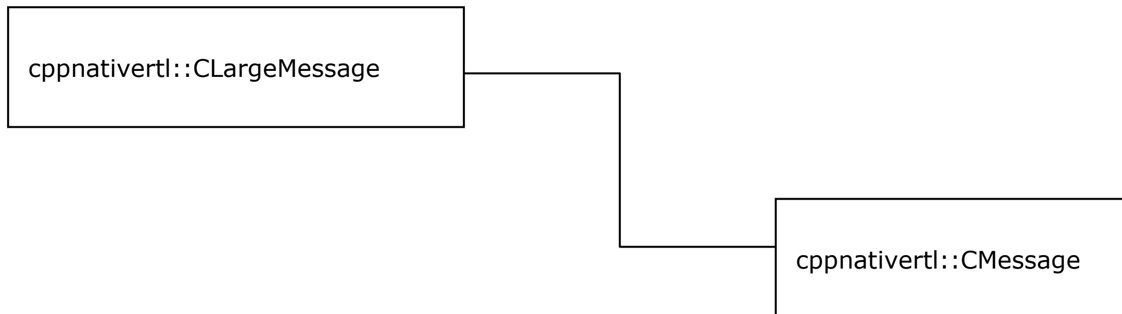
Returns: mqint

Exceptions: `CJMSException`

CMessage

The CMessage interface is the base class of all JMS messages. It defines the message header and the acknowledge method used for all messages.

Inheritance Hierarchy



Subclasses

Base class for:

- CTextMessage
- CBytesMessage
- CMapMessage
- CStreamMessage
- CObjectMessage

Constructors

```
CMessage(struct _Message *msg);
```

Parameters:

- msg – Message structure defined in C runtime.

```
mqcstring getActualDestination() FMQCONST throw (CJMSException *);
```

Purpose: Gets the actual destination name of the message.This method is used to return destination name for which the message was intended to before its arrival on SYSTEM_DMQ due to TTL expiration.

Parameters: None

Returns: mqcstring

Exceptions: CJMSException

```
mqint getCompressionLevel (mqint *val) FMQCONST throw (CJMSException *);
```

Purpose: Returns the Compression Level set on a given message.

Parameters: Pointer to mqint, into which the mqint property is to be read

Returns: mqint

Exceptions: CJMSException

```
mqint getCompressionStrategy (mqint *val) FMQCONST throw (CJMSException *);
```

Purpose: Returns the Compression Strategy set on a given message.

Parameters: Pointer to mqint, into which the mqint property is to be read

Returns: mqint

Exceptions: CJMSException

```
mqfloat getCompressionRatio() FMQCONST  throw (CJMSException *);
```

Purpose: Returns the Compression Ratio.

Parameters: None

Returns: mqfloat

Exceptions: CJMSException

```
mqboolean setCompressionLevel(const mqint m_nCompressionLevel)   throw (CJMSException *);
```

Purpose: Sets the specified Compression Level on a given message.

Parameters:

- m_nCompressionLevel - Compression level to be set

Returns: mqboolean

Exceptions: CJMSException

```
mqboolean setCompressionStrategy(const mqint m_nCompressionStrategy) throw (CJMSException *);
```

Purpose: Sets the specified Compression Strategy on a given message.

Parameters:

- m_nCompressionLevel - Compression level to be set
- m_nCompressionStrategy - Compression Strategy to be set

Returns: mqboolean

Exceptions: CJMSException

```
mqboolean enableCompression() throw (CJMSException *);
```

Purpose: Enables Message Compression for a given message

Parameters: None

Returns: mqboolean

Exceptions: CJMSException

```
mqboolean enableCompression_params(const mqint m_nCompressionLevel,const mqint
m_nCompressionStrategy) throw (CJMSException *);
```

Purpose: Enables Message Compression for a given message with specified compression level and compression strategy.

Parameters:

- m_nCompressionStrategy - Compression Strategy to be set

Returns: mqboolean

Exceptions: CJMSException

Methods

```
void acknowledge()
```

Acknowledges all consumed messages of the session of this consumed message. All consumed JMS messages support the acknowledge method for use when a client has specified that its JMS session's consumed messages are to be explicitly acknowledged. By invoking acknowledge on a consumed message, a client acknowledges all messages consumed by the session that the message was delivered.

Calls to acknowledge are ignored for both transacted sessions and sessions specified to use implicit acknowledgement modes. A client may individually acknowledge each message as it is consumed, or it may choose to acknowledge messages as an application-defined group (which is done by calling acknowledge on the last received message of the group, thereby acknowledging all messages consumed by the session.) Messages that have been received but not acknowledged may be redelivered.

Parameters: None

Returns: Void

Exceptions: `CJMSException`

```
void clearBody() throw (CJMSException *)
```

Clears out the message body. Clearing a message's body does not clear its header values or property entries. If this message body was read-only, calling this method leaves the message body in the same state as an empty body in a newly created message.

Parameters: None

Returns: Void

Exceptions: `CJMSException`

```
void clearProperties() throw (CJMSException *)
```

Clears a message's properties. The message's header fields and body are not cleared.

Parameters: None

Returns: Void

Exceptions: CJMSException

```
CProperty getProperty(mqcstring propName) FMQCONST throw (CJMSException *);
```

Gets the Property object from the message with the specified propName.

Note : This property object should be freed by the client application as it is not freed by the RTL.

Parameters:

- propName – Name of the property to be returned.

Returns: CProperty

Exceptions: CJMSException

```
mqboolean getBooleanProperty(mqcstring name) FMQCONST throw (CJMSException *)
```

Returns the value of the boolean property with the specified name.

Parameters:

- name: The name of the boolean property

Returns: The boolean property value for the specified name

Exceptions: CJMSException

```
mqbyte getByteProperty(mqcstring name) FMQCONST throw (CJMSException *)
```

Returns the value of the byte property with the specified name.

Parameters:

- name: The name of the byte property

Returns: The byte property value for the specified name

Exceptions: CJMSException

```
mqdouble getDoubleProperty(mqcstring name) FMQCONST throw (CJMSException *)
```

Returns the value of the double property with the specified name.

Parameters:

- name: The name of the double property

Returns: The double property value for the specified name. If there is no property by this name, a null value is returned

Exceptions: CJMSException

```
mqfloat getFloatProperty(mqcstring name)FMQCONST throw (CJMSException *)
```

Returns the value of the float property with the specified name.

Parameters:

- name: The name of the float property

Returns: The float property value for the specified name

Exceptions: CJMSException

```
mqint getIntProperty(mqcstring name) FMQCONST throw (CJMSException *)
```

Returns the value of the int property with the specified name.

Parameters:

- name: The name of the int property

Returns: The int property value for the specified name

Exceptions: CJMSException

```
mqcstring getJMSCorrelationID() FMQCONST throw (CJMSException *)
```

Gets the correlation ID for the message. This method is used to return correlation ID values that are either provider-specific message IDs or application-specific String values.

Parameters: None

Returns: The correlation ID of a message as a String

Exceptions: CJMSException

```
mqcstring getJMSCorrelationIDAsBytes() FMQCONST throw (CJMSException *);
```

Gets the correlation ID as an array of bytes for the message. The use of a byte[] value for JMSCorrelationID is non-portable.

Parameters: None

Returns: The correlation ID of a message as an array of bytes

Exceptions: CJMSException

```
mqint getJMSDeliveryMode() FMQCONST throw (CJMSException *);
```

Gets the DeliveryMode value specified for this message.

Parameters: None

Returns: The delivery mode for this message

Exceptions: CJMSException

```
CDestination *getJMSDestination() FMQCONST throw (CJMSException *)
```

Gets the Destination object for this message. The JMSDestination header field contains the destination to which the message is being sent. When a message is sent, this field is ignored. After completion of the send or publish method, the field holds the destination specified by the method. When a message is received, its JMSDestination value must be equivalent to the value assigned when it was sent.

Parameters: None

Returns: The destination of this message

Exceptions: CJMSException

```
mqlong getJMSExpiration() FMQCONST  throw (CJMSException *)
```

Gets the message's expiration value. When a message is sent, the JMSExpiration header field is left unassigned. After completion of the send or publish method, it holds the expiration time of the message. This is the sum of the time-to-live value specified by the client and the GMT at the time of the send or publish. If the time-to-live is specified as zero, JMSExpiration is set to zero to indicate that the message does not expire. When a message's expiration time is reached, a provider should discard it.The JMS API does not define any form of notification of message expiration. Clients should not receive messages that have expired; however, the JMS API does not guarantee that this will not happen.

Parameters: None

Returns: The time the message expires, which is the sum of the time-to-live value specified by the client and the GMT at the time of the send

Exceptions: CJMSException

```
mqcstring getJMSMessageID() FMQCONST throw (CJMSException *)
```

Gets the message ID. The JMSMessageID header field contains a value that uniquely identifies each message sent by a provider. When a message is sent, JMSMessageID can be ignored. When the send or publish method returns, it contains a provider-assigned value. A JMSMessageID is a String value that should function as a unique key for identifying messages in a historical repository. The exact scope of uniqueness is provider-defined. It should at least cover all messages for a specific installation of a provider, where an installation is some connected set of message routers. All JMSMessageID values must start with the prefix 'ID:'. Uniqueness of message ID values across different providers is not required.

Since message IDs take some effort to create and increase a message's size, some JMS providers may be able to optimize message overhead if they are given a hint that the messageID is not used by an application. By calling the MessageProducer. setDisableMessageID method, a JMS client enables this potential optimization for all messages sent by that message producer. If the JMS provider accepts this hint, these messages must have the message ID set to null. If the provider ignores the hint, the messageID must be set to its normal unique value.

Parameters: None

Returns: The message ID

Exceptions: CJMSException

If the JMS provider fails to get the message ID due to some internal error.

```
mqint getJMSPriority() FMQCONST throw (CJMSException *)
```

Gets the message priority level. The JMS API defines ten levels of priority value, with 0 as the lowest priority and 9 as the highest. In addition, clients should consider priorities 0-4 as gradations of normal priority and priorities 5-9 as gradations of expedited priority. The JMS API does not require that a provider strictly implement priority ordering of messages; however, it should do its best to deliver expedited messages ahead of normal messages.

Parameters: None

Returns: The default message priority

Exceptions: CJMSException

If the JMS provider fails to get the message priority due to some internal error.

```
mqboolean getJMSRedelivered() FMQCONST throw (CJMSException *)
```

Gets an indication of whether this message is being redelivered. If a client receives a message with the JMSRedelivered field set. It is likely, but not guaranteed, that this message was delivered earlier but that its receipt was not acknowledged at that time.

Parameters: None

Returns: TRUE if this message is being redelivered

Exceptions: CJMSException

If the JMS provider fails to get the redelivered state due to some internal error.

`CDestination *getJMSReplyTo() FMQCONST throw (CJMSException *)`

Gets the Destination object to which a reply to this message should be sent.

Parameters: None

Returns: Destination to which to send a response to this message

Exceptions: CJMSException

If the JMS provider fails to get the JMSReplyTo destination due to some internal error.

`mqlong getJMSTimestamp() FMQCONST throw (CJMSException *)`

Gets the message timestamp. The JMSTimestamp header field contains the time when the message was handed off to a provider to be sent. It is not the time when the message was actually transmitted, because the actual transmission may occur later due to transactions or other client-side queuing of messages. When a message is sent, JMSTimestamp is ignored.

Parameters: None

Returns: The timestamp of the message

Exceptions: CJMSException

If the JMS provider fails to get the timestamp due to some internal error.

`mqcstring getJMSType() FMQCONST throw (CJMSException *)`

Gets the message type identifier supplied by the client when the message was sent.

Parameters: None

Returns: The message type

Exceptions: CJMSException

If the JMS provider fails to get the message type due to some internal error.

`mqlong getLongProperty(mqcstring propName) FMQCONST throw (CJMSException *)`

Returns the value of the long property with the specified name.

Parameters:

- propName: The name of the long property

Returns: The long property value for the specified name

Exceptions: CJMSException

If the JMS provider fails to get the property value due to some internal error.

`mqobject getObjectProperty(mqcstring propName) FMQCONST throw (CJMSException *)`

Returns the value of the object property with the specified name. This method can be used to return, in objectified format, an object that has been stored as a property in the message with the equivalent setObjectProperty method call, or its equivalent primitive settypeProperty method.

Parameters:

- PropName: The name of the object property

Returns: The object property value with the specified name.

Exceptions: CJMSException

If the JMS provider fails to get the property value due to some internal error.

`CEnumeration *getPropertyNames() FMQCONST throw (CJMSException *)`

Returns an enumeration of the property names from the message object

Parameters: None

Returns: An enumeration of the property names.

> **Note**: The CEnumeration object contains all the property names present in the received message.The property names can be retrieved from the CEnumeration object using nextElement method.The nextElement method returns void* and it should be type casted to const char *.

Exceptions: CJMSException

If the JMS provider fails to get the property value due to some internal error.

`mqshort getShortProperty(mqcstring propName) FMQCONST throw (CJMSException *)`

Returns the value of the short property with the specified name.

Parameters:

- PropName: The name of the short property

Returns: The short property value for the specified name

Exceptions: CJMSException

If the JMS provider fails to get the property value due to some internal error.

`mqcstring getStringProperty(mqcstring propName) FMQCONST throw (CJMSException *)`

Returns the value of the String property with the specified name.

Parameters:

- PropName: The name of the String property

Returns: The String property value for the specified name. If there is no property by this name, a null value is returned

Exceptions: CJMSException

If the JMS provider fails to get the property value due to some internal error.

`mqcstring_unicode getStringProperty_unicode(mqcstring_unicode propName) FMQCONST throw (CJMSException *)`

Returns the unicode string property with the specified name.

Parameters:

- PropName: The name of the String property in Unicode.

Returns: The unicode string property value for the specified name. If there is no property by this name, a null value is returned

Exceptions: CJMSException

If the JMS provider fails to get the property value due to some internal error.

`mqint getMessageType() FMQCONST throw (CJMSException *)`

Returns the message type property value as mqint.

Parameters: None

Returns: The mqint property value for the message type.

Exceptions: CJMSException

If the JMS provider fails to get the property value due to some internal error.

`mqboolean propertyExists(mqcstring propName) FMQCONST throw (CJMSException *)`

Indicates whether a property value exists.

Parameters:

- PropName: The name of the property to test

Returns: TRUE if the property exists

Exceptions: CJMSException

If the JMS provider fails to determine if the property exists due to some internal error.

`void setBooleanProperty(mqcstring propName, const mqboolean value) throw (CJMSException *)`

Sets a boolean property value with the specified name into the message.

Parameters:

- name: The name of the boolean property
- value: the boolean property value to set

Returns: Void

Exceptions: CJMSException

If the JMS provider fails to set the property due to some internal error.

```
void setByteProperty(mqcstring propName,const  mqbyte value) throw (CJMSException *)
```

Sets a byte property value with the specified name into the message.

Parameters:

- name: The name of the byte property value - the byte property value to set

Returns: Void

Exceptions: CJMSException

```
void setDoubleProperty(mqcstring propName,const  mqdouble value) throw (CJMSException *)
```

Sets a double property value with the specified name into the message.

Parameters:

- name: The name of the double property
- value: The double property value to set

Returns: Void

Exceptions: CJMSException

If the JMS provider fails to set the property due to some internal error.

```
void setFloatProperty(mqcstring propName,const  mqfloat value) throw (CJMSException *);
```

Sets a float property value with the specified name into the message.

Parameters:

- name: The name of the float property
- value: The float property value to set

Returns: Void

Exceptions: CJMSException

If the JMS provider fails to set the property due to some internal error.

```
void setIntProperty(mqcstring propName,const  mqint value) throw (CJMSException *)
```

Sets an int property value with the specified name into the message.

Parameters:

- propName: The name of the int property
- Value: The int property value to set

Returns: Void

Exceptions: CJMSException

If the JMS provider fails to set the property due to some internal error.

`void setJMSCorrelationID(mqcstring corrID) throw (CJMSException *)`

Sets the correlation ID for the message. A client can use the JMSCorrelationID header field to link one message with another. A typical use is to link a response message with its request message.

Parameters:

- correlationID: The message ID of a message being referred to

Returns: Void

Exceptions: CJMSException

If the JMS provider fails to set the correlation ID due to some internal error.

`void setJMSDeliveryMode(const mqint deliveryMode) throw (CJMSException *)`

Sets the DeliveryMode value for this message. JMS providers set this field when a message is sent. This method can be used to change the value for a message that has been received.

Parameters:

- deliveryMode: The delivery mode for this message

Returns: Void

Exceptions: CJMSException

If the JMS provider fails to set the delivery mode due to some internal error.

`void setJMSDestination(const CDestination *dest) throw (CJMSException *)`

Sets the Destination object for this message. JMS providers set this field when a message is sent. This method can be used to change the value for a message that has been received.

Parameters:

- Destination: The destination for this message

Returns: Void

Exceptions: CJMSException

If the JMS provider fails to set the destination due to some internal error.

`void setJMSExpiration(const mqlong expiration) throw (CJMSException *)`

Sets the message's expiration value. JMS providers set this field when a message is sent. This method can be used to change the value for a message that has been received.

Parameters:

- Expiration: The message's expiration time

Returns: Void

Exceptions: CJMSException

If the JMS provider fails to set the message expiration due to some internal error.

```
void setJMSMessageID(mqcstring msgID) throw (CJMSException *)
```

Sets the message ID. JMS providers set this field when a message is sent. This method can be used to change the value for a message that has been received.

Parameters:

- msgID: The ID of the message

Returns: Void

Exceptions: CJMSException

If the JMS provider fails to set the message ID due to some internal error.

```
void setJMSPriority(const mqint priority)  throw (CJMSException *)
```

Sets the priority level for this message. JMS providers set this field whena message is sent. This method can be used to change the value for a message that has been received.

Returns: Void

Parameters:

- Priority: The priority of this message

Exceptions: CJMSException

If the JMS provider fails to set the message priority due to some internal error.

```
void setJMSRedelivered(const mqboolean redelivered) throw (CJMSException *);
```

Specifies whether this message is being redelivered. This field is set at the time the message is delivered. This method can be used to change the value for a message that has been received.

Parameters:

- Redelivered: An indication of whether this message is being redelivered

Returns: Void

Exceptions: CJMSException

If the JMS provider fails to set the redelivered state due to some internal error.

```
void setJMSReplyTo(const CDestination *dest) throw (CJMSException *)
```

Sets the Destination object to which a reply to this message should be sent. The JMSReplyTo header field contains the destination where a reply to the current message should be sent. If it is null, no reply is expected. The destination may be either a Queue object or a Topic object. Messages sent with a null JMSReplyTo value may be a notification of some event, or they may just be some data the sender thinks is of interest. Messages with a JMSReplyTo value typically expect a response. A response is optional; it is up to the client to decide. These messages are called requests. A message sent in response to a request is called a reply. In some cases a client may wish to match a request it sent earlier with a reply it has just received. The client can use the JMSCorrelationID header field for this purpose.

Parameters:

- Dest: Destination to which to send a response to this message

Returns: Void

Exceptions: CJMSException

If the JMS provider fails to set the JMSReplyTo destinationdue to some internal error.

```
void setJMSTimestamp(const mqlong timestamp) throw (CJMSException *)
```

Sets the message timestamp. JMS providers set this field when a message is sent. This method can be used to change the value for a message that has been received.

Parameters:

- Timestamp: The timestamp for this message

Returns: Void

Exceptions: CJMSException

If the JMS provider fails to set the timestamp due to some internal error.

```
void setJMSType(mqcstring type) throw (CJMSException *)
```

Sets the message type. Some JMS providers use a message repository that contains the definitions of messages sent by applications. The JMSType header field may reference a message's definition in the provider's repository. The JMS API does not define a standard message definition repository, nor does it define a naming policy for the definitions it contains.

Parameters:

- Type: The message type

Returns: Void

Exceptions: CJMSException

If the JMS provider fails to set the message type due to some internal error.

```
void setLongProperty(mqcstring propName,const mqlong value) throw (CJMSException *);
```

Sets a long property value with the specified name into the message.

Parameters:

- Name: The name of the long property
- Value: The long property value to set.

Returns: Void

Exceptions: CJMSException

If the JMS provider fails to set the property due to some internal error.

```
void setObjectProperty(mqcstring propName, mqobject value,const mqint size) throw
(CJMSException *);
```

Sets a object property value with the specified name into the message. This method works only for the objectified primitive object types (Integer, Double, Long...) and String objects.

Parameters:

- Name: The name of the object property
- Value: The object property value to set.

Returns: Void

Exceptions: CJMSException

If the JMS provider fails to set the property due to some internal error.

```
void setShortProperty(mqcstring propName,const mqshort value) throw (CJMSException *);
```

Sets a short property value with the specified name into the message.

Parameters:

- PropName: The name of the short property
- Value: The short property value to set.

Returns: Void

Exceptions: CJMSException

If the JMS provider fails to set the property due to some internal error.

```
void setStringProperty(mqcstring propName, mqcstring value) throw (CJMSException *);
```

Sets a String property value with the specified name into the message.

Parameters:

- PropName: The name of the String property
- Value: The String property value to set.

Returns: Void

Exceptions: CJMSException

If the JMS provider fails to set the property due to some internal error.

`void setDiscardable(mqboolean isDiscardable) throw (CJMSException *)`

Gets the isDiscardable value specified for this message.

Parameters:

- isDiscardable: boolean value TRUE or FALSE

Returns: Void

Exceptions: CJMSException

If the JMS provider fails to set the property due to some internal error.

`mqboolean isDiscardable() FMQCONST  throw (CJMSException *)`

Gets the isDiscardable value specified for this message.

Parameters: None

Returns:

- mqboolean: the Discardable value for this message

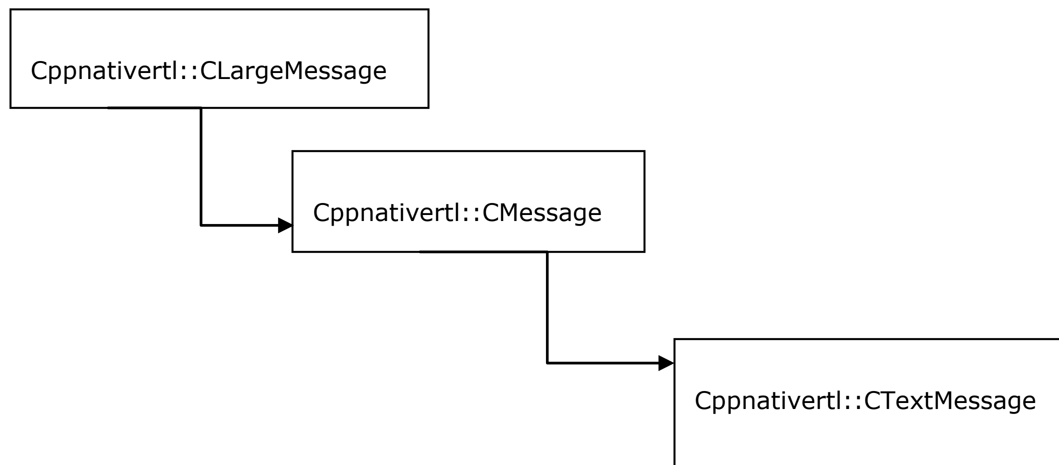
Exceptions: CJMSException

If the JMS provider fails to set the property due to some internal error.

CTextMessage

A CTextMessage object is used to send a message containing a mqcstring. It inherits from the CMessage class and adds a text message body.

Inheritance Hierarchy



Subclasses

None

Constructors

```
CTextMessage(struct _Message *msg);
```

Parameters:

- msg: Message structure defined in C runtime.

Methods

```
mqcstring_unicode getText() FMQCONST
```

Gets the mqcstring_unicode containing this message's data. The default value is null.

Parameters: None

Returns: The String containing the message's data

Exceptions: CJMSException

If the JMS provider fails to get the text due to some internal error.

```
void setText(mqcstring_unicode)
```

Sets the mqcstring_unicode containing this message's data.

Parameters:

- mqcstring_unicode - The String containing the message's data

Returns: Void

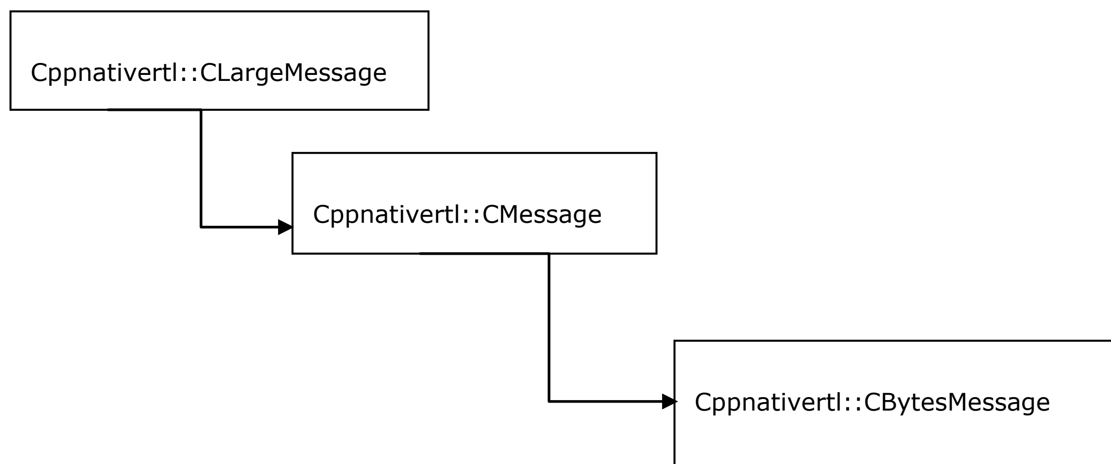
Exceptions: CJMSException

If the JMS provider fails to set the text due to some internal error.

CByteMessage

A CBytesMessage object is used to send a message containing a stream of uninterpreted bytes. It inherits from the CMessage class and adds a bytes message body. The receiver of the message supplies the interpretation of the bytes.

Inheritance Hierarchy



Subclasses

None

Constructors

```
CBytesMessage(struct _Message *msg)
```

Parameters:

- msg: Message structure defined in C runtime.

Methods

```
mqlong getBodyLength() FMQCONST throw (CJMSException  *)
```

Gets the number of bytes of the message body when the message is in read-only mode. The value returned can be used to allocate a byte array. The value returned is the entire length of the message body, regardless of where the pointer for reading the message is currently located.

Parameters: None

Returns: Number of bytes in the message.

Exceptions: CJMSException

```
mqboolean readBoolean() FMQCONST   throw (CJMSException  *)
```

Reads a boolean from the bytes message stream.

Parameters: None

Returns: The boolean value read

Exceptions: CJMSException

If the JMS provider fails to read the message due to some internal error.

```
mqbyte readByte() FMQCONST  throw (CJMSException  *)
```

Reads a signed 8-bit value from the bytes message stream.

Parameters: None

Returns: The next byte from the bytes message stream as a signed 8-bit byte

Exceptions: CJMSException

If the JMS provider fails to read the message due to some internal error.

```
mqint readBytes(mqbyteArray value, int length) FMQCONST  throw (CJMSException  *)
```

Reads a byte array from the bytes message stream. If the length of array value is less than the number of bytes remaining to be read from the stream, the array should be filled. A subsequent call reads the next increment, and so on. If the number of bytes remaining in the stream is less than the length of array value, the bytes should be read into the array. The return value of the total number of bytes read will be less than the length of the array, indicating that there are no more bytes left to be read from the stream.

The next read of the stream returns -1.

Parameters:

- value - The buffer into which the data is read.
- length - The length of the buffer.

Returns: The total number of bytes read into the buffer or -1 if there is no more data because the end of the stream has been reached.

Exceptions: CJMSException

If the JMS provider fails to read the message due to some internal error.

`mqchar readChar() FMQCONST throw (CJMSException *)`

Reads a Unicode character value from the bytes message stream.

Parameters: None

Returns: The next two bytes from the bytes message stream as a Unicode character

Exceptions: CJMSException

If the JMS provider fails to read the message due to some internal error.

`mqdouble readDouble() FMQCONST throw (CJMSException *)`

Reads a double from the bytes message stream.

Parameters: None

Returns: The next eight bytes from the bytes message stream, interpreted as a double.

Exceptions: CJMSException

`mqfloat readFloat() FMQCONST throw (CJMSException *)`

Reads a float from the bytes message stream.

Parameters: None

Returns: The next four bytes from the bytes message stream, interpreted as a float

Exceptions: CJMSException

If the JMS provider fails to read the message due to some internal error.

`mqint readInt() FMQCONST throw (CJMSException *)`

Reads a signed 32-bit integer from the bytes message stream.

Parameters: None

Returns: The next four bytes from the bytes message stream, interpreted as an int

Exceptions: CJMSException

If the JMS provider fails to read the message due to some internal error.

`mqlong readLong() FMQCONST throw (CJMSException *)`

Reads a signed 64-bit integer from the bytes message stream.

Parameters: None

Returns: The next eight bytes from the bytes message stream, interpreted as a long.

Exceptions: CJMSException

If the JMS provider fails to read the message due to some internal error.

`mqshort readShort() FMQCONST throw (CJMSException *)`

Reads a signed 16-bit number from the bytes message stream.

Parameters: None

Returns: The next two bytes from the bytes message stream, interpreted as a signed 16-bit number

Exceptions: CJMSException

If the JMS provider fails to read the message due to some internal error.

`mqint readUnsignedByte() FMQCONST throw (CJMSException *)`

Reads an unsigned 8-bit number from the bytes message stream.

Parameters: None

Returns: The next byte from the bytes message stream, interpreted as an unsigned 8-bit number

Exceptions: CJMSException

If the JMS provider fails to read the message due to some internal error.

`mqint readUnsignedShort() FMQCONST throw (CJMSException *)`

Reads an unsigned 16-bit number from the bytes message stream.

Parameters: None

Returns: The next two bytes from the bytes message stream, interpreted as an unsigned 16-bit integer.

Exceptions: CJMSException

If the JMS provider fails to read the message due to some internal error.

`mqcstring_unicode readUTF() FMQCONST throw (CJMSException *)`

Reads a mqcstring that has been encoded using a modified UTF-8 format from the bytes message stream.

Parameters: None

Returns: A Unicode mqcstring from the bytes message stream

Exceptions: CJMSException

If the JMS provider fails to read the message due to some internal error.

`void reset() throw (CJMSException *)`

Puts the message body in read-only mode and repositions the stream of bytes to the beginning.

Parameters: None

Returns: None

Exceptions: CJMSException

If the JMS provider fails to reset the message due to some internal error.

```
void writeBoolean(mqboolean value) throw (CJMSException  *)
```

Writes a boolean to the bytes message stream as a 1-byte value. The value true is written as the value (byte)1; the value false is written as the value (byte)0.

Parameters:

- Value: The boolean value to be written

Returns: Void

Exceptions: CJMSException

If the JMS provider fails to write the message due to some internal error.

```
void writeByte(mqbyte value) throw (CJMSException  *)
```

Writes a byte to the bytes message stream as a 1-byte value.

Parameters:

- Value- The byte value to be written

Returns: Void

Exceptions: CJMSException

If the JMS provider fails to write the message due to some internal error.

```
void writeByte(mqbyte value) throw (CJMSException  *)
```

Writes a portion of a byte array to the bytes message stream.

Parameters:

- Value - The byte array value to be written
- Offset - The initial offset within the byte array
- Length - The number of bytes to use

Returns: Void

Exceptions: CJMSException

If the JMS provider fails to write the message due to some internal error.

```
void writeChar(mqchar value) throw (CJMSException  *)
```

Writes a char to the bytes message stream as a 2-byte value, high byte first.

Parameters:

- Value - The char value to be written

Returns: Void

Exceptions: CJMSException

If the JMS provider fails to write the message due to some internal error.

`void writeDouble(mqdouble value) throw (CJMSException *)`

Writes a double from the bytes message stream.

Parameter:

- Value - mqdouble value to be written to the bytes message

Returns: Void

Exceptions: CJMSException

If the JMS provider fails to read the message due to some internal error.

`void writeFloat(mqfloat value) throw (CJMSException *)`

Converts the float argument to an int using the floatToIntBits method in class Float, and then writes that int value to the bytes message stream as a 4-byte quantity, high byte first.

Parameters:

- Value - The float value to be written

Returns: Void

Exceptions: CJMSException

If the JMS provider fails to write the message due to some internal error.

`void writeInt(mqint value) throw (CJMSException *)`

Writes an int to the bytes message stream as four bytes, high byte first.

Parameters:

- Value- The int to be written

Returns: Void

Exceptions: CJMSException

If the JMS provider fails to write the message due to some internal error.

`void writeLong(mqlong value) throw (CJMSException *)`

Writes a long to the bytes message stream as eight bytes, high byte first.

Parameters:

- Value - The long to be written

Returns: Void

Exceptions: CJMSException

If the JMS provider fails to write the message due to some internal error.

```
void writeShort(mqshort value)  throw (CJMSException  *)
```

Writes a short to the bytes message stream as two bytes, high byte first.

Parameters:

- Value: The short to be written

Returns: Void

Exceptions: CJMSException

If the JMS provider fails to write the message due to some internal error.

```
void writeUTF(mqcstring_unicode value) throw (CJMSException  *)
```

Writes a mqcstring to the bytes message stream using UTF-8 encoding in a machine-independent manner.

Parameters:

- Value - The String value to be written

Returns: Void

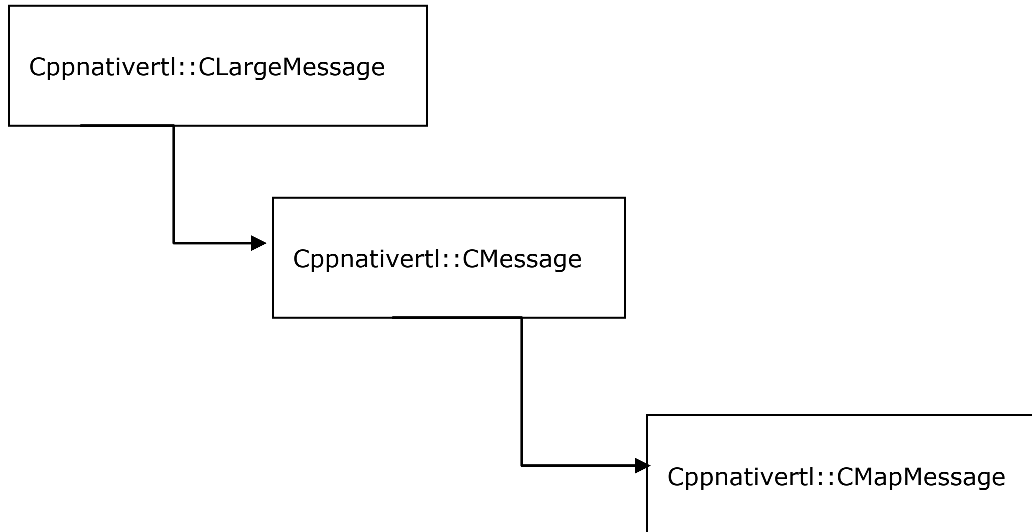
Exceptions: CJMSException

If the JMS provider fails to write the message due to some internal error.

CMapMessage

The CMapMessage object is used to send a set of name-value pairs. The names must have a value that is not null, and not an empty mqcstring. The entries can be accessed sequentially or randomly by name.CMapMessage inherits from the CMessage class and adds a message body that contains a Map.

Inheritance Hierarchy



Subclasses

None

Constructors

```
CMapMessage(struct _Message *msg)
```

Parameters:

- Msg: Message structure defined in C runtime.

Methods

```
mqboolean getBoolean(mqcstring name) FMQCONST throw (CJMSException *)
```

Returns the boolean value with the specified name.

Parameters:

- Name: The name of the Boolean

Returns: The boolean value with the specified name

Exceptions: CJMSException

If the JMS provider fails to read the message due to some internal error.

`mqbyte getByte(mqcstring name) FMQCONST throw (CJMSException *)`

Returns the byte value with the specified name.

Parameters:

- Name: The name of the byte

Returns: The byte value with the specified name.

Exceptions: CJMSException

If the JMS provider fails to read the message due to some internal error.

`mqchar getChar(mqcstring name) FMQCONST throw (CJMSException *)`

Returns the Unicode character value with the specified name.

Parameters

- Name: The name of the Unicode character

Returns: The Unicode character value with the specified name

Exceptions: CJMSException

If the JMS provider fails to read the message due to some internal error.

`mqdouble getDouble(mqcstring name) FMQCONST throw (CJMSException *)`

Returns the double value with the specified name.

Parameters

- Name: The name of the double

Returns: The double value with the specified name

Exceptions: CJMSException

If the JMS provider fails to read the message due to some internal error.

`mqfloat getFloat(mqcstring name)FMQCONST   throw (CJMSException *)`

Returns the float value with the specified name.

Parameters

- Name: The name of the float

Returns: The float value with the specified name

Exceptions: CJMSException

If the JMS provider fails to read the message due to some internal error.

`mqint getInt(mqcstring name) FMQCONST      throw (CJMSException *)`

Returns the int value with the specified name.

Parameters

- Name: The name of the int

Returns: The int value with the specified name

Exceptions: CJMSException

If the JMS provider fails to read the message due to some internal error.

```
mqlong getLong(mqcstring name) FMQCONST   throw (CJMSException *)
```

Returns the long value with the specified name.

Parameters

- Name: The name of the long

Returns: The long value with the specified name

Exceptions: CJMSException

If the JMS provider fails to read the message due to some internal error.

```
mqcstring* getMapNames() FMQCONST   throw (CJMSException *)
```

Returns a pointer to mqcstrings of all the names in the MapMessage object.

Parameters: None

Returns: An a pointer to all the names (mqcstring) in this MapMessage

Exceptions: CJMSException

If the JMS provider fails to read the message due to some internal error.

```
CHashTableEnumerator *getMapNamesHTEnum() FMQCONST throw (CJMSException *)
```

Returns an Enumeration of all the names in the MapMessage object.

Parameters: None

Returns: An enumeration of all the names in this MapMessage

Exceptions: CJMSException

If the JMS provider fails to read the message due to some internal error.

```
mqshort getShort(mqcstring name) FMQCONST  throw (CJMSException *)
```

Returns the short value with the specified name.

Parameters

- Name: The name of the short

Returns: The short value with the specified name

Exceptions: CJMSException

If the JMS provider fails to read the message due to some internal error.

`mqcstring getString(mqcstring name) FMQCONST throw (CJMSException *)`

Returns the String value with the specified name.

Parameters

- Name: The name of the String

Returns: The String value with the specified name; if there is no item by this name, a null value is returned.

Exceptions: CJMSException

If the JMS provider fails to read the message due to some internal error.

`mqboolean itemExists(mqcstring name) FMQCONST     throw (CJMSException *)`

Indicates whether an item exists in this MapMessage object.

Parameters

- Name: The name of the item to test

Returns: TRUE if the item exists

Exceptions: CJMSException

If the JMS provider fails to determine if the item exists due to some internal error.

`void setBoolean(mqcstring name, const mqboolean value)    throw (CJMSException *)`

Sets a boolean value with the specified name into the Map.

Parameters

- Name: The name of the boolean
- Value: The boolean value to set in the Map

Returns: void

Exceptions: CJMSException

If the JMS provider fails to write the message due to some internal error.

`void setByte(mqcstring name, const mqbyte value)  throw (CJMSException *)`

Sets a byte value with the specified name into the Map.

Parameters

- Name: The name of the byte
- Value: The byte value to set in the Map

Return: Void

Exceptions: CJMSException

If the JMS provider fails to write the message due to some internal error.

```
void setBytes(mqcstring name, mqcstring value,const  mqint length) throw (CJMSException *)
```

Sets a byte array value with the specified name and length into the Map.

Parameters

- Name: The name of the byte array
- Value: The byte array value to set in the Map
- Length: Number of bytes to be set

Return: Void

Exceptions: CJMSException

If the JMS provider fails to write the message due to some internal error.

```
void setBytes(mqcstring name, mqbyteArray value) throw (CJMSException *);
```

Sets a byte array value with the specified name into the Map.

Parameters

- Name: The name of the byte array
- Value: The byte array value to set in the Map; the array is copied so that the value for name will not be altered by future modifications

Returns: Void

Exceptions: CJMSException

If the JMS provider fails to write the message due to some internal error.

```
void setBytes(mqcstring name,mqcstring value, const mqint offset, const mqint length) throw
(CJMSException *)
```

Sets a portion of the byte array value with the specified name into the Map.

Parameters

- Name: The name of the byte array
- Value: The byte array value to set in the Map
- offset : The initial offset within the byte array
- length: The number of bytes to use

Returns: Void

Exceptions: CJMSException

If the JMS provider fails to write the message due to some internal error.

`void setChar(mqcstring name,const mqchar value) throw (CJMSException *)`

Sets a Unicode character value with the specified name into the Map.

Parameters

- name: The name of the Unicode character
- Value: The Unicode character value to set in the Map

Returns: Void

Exceptions: MSException

If the JMS provider fails to write the message due to some internal error.

`void setDouble(mqcstring name, const mqdouble value) throw (CJMSException *)`

Sets a double value with the specified name into the Map.

Parameters

- Name: The name of the double
- Value: The double value to set in the Map

Returns: Void

Exceptions: CJMSException

If the JMS provider fails to write the message due to some internal error.

`void setFloat(mqcstring name, const mqfloat value)throw (CJMSException *)`

Sets a float value with the specified name into the Map.

Parameters

- Name: The name of the float
- Value: The float value to set in the Map

Returns: Void

Exceptions: CJMSException

If the JMS provider fails to write the message due to some internal error.

`void setInt(mqcstring name, const mqint value) throw (CJMSException *)`

Sets an int value with the specified name into the Map.

Parameters

- Name: The name of the int
- Value: The int value to set in the Map

Returns: Void

Exceptions: CJMSException

If the JMS provider fails to write the message due to some internal error.

```
void setLong(mqcstring name, const mqlong value)  throw (CJMSException *)
```

Sets a long value with the specified name into the Map.

Parameters

- Name: The name of the long
- Value: The long value to set in the Map

Returns: Void

Exceptions: CJMSException

If the JMS provider fails to write b the message due to some internal error.

```
void setShort(mqcstring name, const mqshort value) throw (CJMSException *)
```

Sets a short value with the specified name into the Map.

Parameters

- Name: The name of the short
- Value: The short value to set in the Map

Returns: Void

Exceptions: CJMSException

If the JMS provider fails to write the message due to some internal error.

```
void setString(mqcstring name, mqcstring value) throw (CJMSException *)
```

Sets a String value with the specified name into the Map.

Parameters

- Name: The name of the String
- Value: The String value to set in the Map

Returns: Void

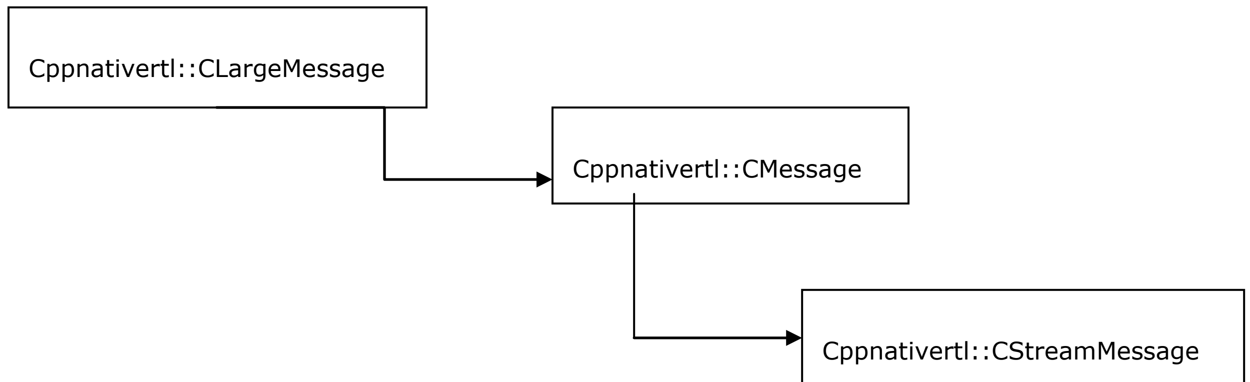
Exceptions: CJMSException

If the JMS provider fails to write the message due to some internal error.

CStreamMessage

A CStreamMessage object is used to send a stream of primitive types in C++ programming language. It is filled and read sequentially. It inherits from the CMessage class and adds a stream message body.

Inheritance Hierarchy



Subclasses

None

Constructors

```
CStreamMessage(struct _Message *msg);
```

Parameters:

- Msg: Message structure defined in C runtime.

Methods

```
mqboolean readBoolean() FMQCONST   throw (CJMSException *)
```

Reads a boolean from the stream message.

Parameters: None

Returns: The Boolean value read

Exceptions: CJMSException

If the JMS provider fails to read the message due to some internal error.

```
mqbyte readByte() FMQCONST  throw (CJMSException *)
```

Reads a byte value from the stream message.

Parameters: None

Returns: The next byte from the stream message as a 8-bit byte

Exceptions: CJMSException

If the JMS provider fails to read the message due to some internal error.

`mqint readBytes(mqbyteArray value, mqint length) FMQCONST throw (CJMSException *)`

Reads a byte array field from the stream message into the specified value.

Parameters

- Value: The buffer into which the data is read
- Length: Length of the byte array

Returns: The total number of bytes read into the buffer, or -1 if there is no more data because the end of the byte field has been reached.

Exceptions: CJMSException

If the JMS provider fails to read the message due to some internal error.

`mqchar readChar() FMQCONST   throw (CJMSException *)`

Reads a Unicode character value from the stream message.

Parameters: None

Returns: A Unicode character from the stream message

Exceptions: CJMSException

If the JMS provider fails to read the message due to some internal error.

`mqdouble readDouble() FMQCONST   throw (CJMSException *)`

Reads a double from the stream message.

Parameters: None

Returns: A double value from the stream message

Exceptions: CJMSException

If the JMS provider fails to read the message due to some internal error.

`mqfloat readFloat() FMQCONST throw (CJMSException *)`

Reads a float from the stream message.

Parameters: None

Returns: A float value from the stream message.

Exceptions: CJMSException

If the JMS provider fails to read the message due to some internal error.

`mqint readInt() FMQCONST throw (CJMSException *)`

Reads a 32-bit integer from the stream message.

Parameters: None

Returns: A 32-bit integer value from the stream message, interpreted as an integer

Exceptions: CJMSException

If the JMS provider fails to read the message due to some internal error.

`mqlong readLong() FMQCONST  throw (CJMSException *)`

Reads a 64-bit integer from the stream message.

Parameters: None

Returns: A 64-bit integer value from the stream message, interpreted as a long.

Exceptions: CJMSException

If the JMS provider fails to read the message due to some internal error.

`mqshort readShort() FMQCONST throw (CJMSException *)`

Reads a 16-bit integer from the stream message.

Parameters: None

Returns: A 16-bit integer from the stream message.

Exceptions: CJMSException

If the JMS provider fails to read the message due to some internal error.

`mqcstring readString() FMQCONST throw (CJMSException *);`

Reads a String from the stream message.

Parameters: None

Returns: A mqcstring from the stream message

Note: This string should be manually freed by the user, as it is not freed by deleting the Message Object.

Exceptions: CJMSException

If the JMS provider fails to read the message due to some internal error.

`void reset() throw (CJMSException *)`

Puts the message body in read-only mode and repositions the stream to the beginning.

Parameters: None

Returns: Void

Exceptions: CJMSException

If the JMS provider fails to reset the message due to some internal error.

```
void writeBoolean(mqboolean value)  throw (CJMSException *)
```

Writes a Boolean to the stream message.

Parameters

- Value: The Boolean value to be written.

Returns: Void

Exceptions: CJMSException

If the JMS provider fails to write the message due to some internal error.

```
void writeByte(mqbyte value) throw (CJMSException *)
```

Writes a byte to the stream message.

Parameters

- Value: The byte value to be written.

Returns: Void

Exceptions: CJMSException

If the JMS provider fails to write the message due to some internal error.

```
void writeBytes(mqcstring value, mqint length) throw (CJMSException *)
```

Writes a byte array field to the stream message. The byte array value is written to the message as a byte array field. Consecutively written byte array fields are treated as two distinct fields when the fields are read.

Parameters

- Value: The byte array value to be written
- Length: length of the bytes to be written

Returns: Void

Exceptions: CJMSException

If the JMS provider fails to write the message due to some internal error.

```
void writeBytes(mqcstring value,mqint offset,mqint length) throw (CJMSException *)
```

Writes a portion of a byte array as a byte array field to the stream message. The portion of the byte array value is written to the message as a byte array field. Consecutively written byte array fields are treated as two distinct fields when the fields are read.

Parameters

- Value: The byte array value to be written
- Offset: The initial offset within the byte array
- Length: The number of bytes to be written

Returns: Void

Exceptions: CJMSException

If the JMS provider fails to write the message due to some internal error.

```
void writeChar(mqchar value) throw (CJMSException *)
```

Writes a char to the stream message.

Parameters

- Value: The char value to be written

Returns: Void

Exceptions: CJMSException

If the JMS provider fails to write the message due to some internal error.

```
void writeDouble(mqdouble value)throw (CJMSException *)
```

Writes a double to the stream message.

Parameters

- Value: The double value to be written

Returns: Void

Exceptions: CJMSException

if the JMS provider fails to write the message due to some internal error.

```
void writeFloat(mqfloat value)  throw (CJMSException *)
```

Writes a float to the stream message.

Parameters

- Value: The float value to be written

Returns: Void

Exceptions: CJMSException

If the JMS provider fails to write the message due to some internal error.

`void writeInt(mqint value) throw (CJMSException *)`

Writes an int to the stream message.

Parameters

- Value: The int value to be written

Returns: Void

Exceptions: CJMSException

If the JMS provider fails to write the message due to some internal error.

`void writeLong(mqlong value) throw (CJMSException *)`

Writes a long to the stream message.

Parameters

- Value: The long value to be written

Exceptions: CJMSException

If the JMS provider fails to write the message due to some internal error.

`void writeShort(mqshort value)  throw (CJMSException *)`

Writes a short to the stream message.

Parameters

- Value: The short value to be written

Exceptions: CJMSException

If the JMS provider fails to write the message due to some internal error.

`void writeString(mqcstring value) throw (CJMSException *)`

Writes a String to the stream message.

Parameters

- Value: The String value to be written

Exceptions: CJMSException

If the JMS provider fails to write the message due to some internal error.

CMQAdminService

The CMQAdminService class provides methods for all Admin requests to create Administered Objects.

Inheritance Hierarchy

None

Subclasses

None

Constructors

`CMQAdminService(struct _FioranoMQAdminService* pmqas)`

Parameters:

- pmqas – FioranoMQAdminService structure defined in C runtime.

Methods

`bool disconnectClient(mqcstring clientID) FMQCONST throw (CJMSException*);`

Parameters: None

Returns: Void

Exceptions: CJMSException

`mqint getNumberOfActiveClientConnections() FMQCONST throw (CJMSException*);`

Gets the number of active client connections to the MQ Server.

Parameters: None

Returns: mqint

Exceptions: CJMSException

```
CEnumeration* getDurableSubscribersForTopic(mqcstring topicName) FMQCONST throw
(CJMSException*)
```

Returns a pointer to enumeration of all the subscriber names on the specified topic.

Parameters: topicName - The topic name.

Returns: Pointer to CEnumeration object of all the subscriber names on the specified topic.

Note: The CEnumeration object contains all the durable subscriber names for the topic.The names can be retrieved from the CEnumeration object using nextElement method.The nextElement method returns void* and it should be type casted to const char *.

Exceptions: CJMSException

```
CEnumeration* getClientIDs() FMQCONST throw (CJMSException*)
```

Returns a pointer to enumeration of all client ids.

Parameters: None

Returns: Pointer to CEnumeration object of all the client ids.

Note: The CEnumeration object contains all the client ID's for the topic.The ID's can be retrieved from the CEnumeration object using nextElement method.The nextElement method returns void* and it should be type casted to const char *.

Exceptions: CJMSException

```
CEnumeration* getPTPClientIDs() FMQCONST throw (CJMSException*)
```

Returns a pointer to enumeration of all client ids for PTP (Point to Point JMS model).

Parameters: None

Returns: Pointer to CEnumeration object of all the client ids on PTP.

Note: The CEnumeration object contains all the PTP client ID's for the topic.The ID's can be retrieved from the CEnumeration object using nextElement method.The nextElement method returns void* and it should be type casted to const char *.

Exceptions: CJMSException

```
CEnumeration* getPubSubClientIDs() FMQCONST throw (CJMSException*)
```

Returns a pointer to enumeration of all client ids for PubSub (Publish / Subscribe JMS model).

Parameters: None

Returns: Pointer to CEnumeration object of all the client ids on PubSub.

Exceptions: CJMSException

Note: The CEnumeration object contains all the PUBSUB client ID's for the topic.The ID's can be retrieved from the CEnumeration object using nextElement method.The nextElement method returns void* and it should be type casted to const char *.

```
CEnumeration* getSubscriberIDs(mqcstring clientID) FMQCONST throw (CJMSException*)
```

Returns a pointer to enumeration of all subscriber ids on the specified client id.

Parameters: clientID -String representing the client id.

Returns: Pointer to CEnumeration object of all the Subscriber ids on the specified client id.

Note: The CEnumeration object contains all the Subscriber ID's for the topic.The ID's can be retrieved from the CEnumeration object using nextElement method.The nextElement method returns void* and it should be type casted to const char *.

Exceptions: CJMSException

`mqcstring getSubscriptionTopicName(mqcstring clientID, mqcstring subscriberID) FMQCONST throw (CJMSException*)`

Returns a mqcstring subscription topic name for the specified clientID and subscriberID combo.

Parameters:

- clientID - String representing the client id
- subscriberID - String representing the subscriber id.

Returns: Mqcstring- subscription topic name for the specified clientID and subscriberID combo.

Exceptions: CJMSException

`mqlong getNumberOfDeliverableMessages1(mqcstring queueName) FMQCONST throw (CJMSException*)`

Returns a mqlong object for the number of deliverable messages on the specified queue.

Parameters:

- queueName - String representing the queue name.

Returns: mqlong object for the number of deliverable messages on the specified queue.

Exceptions: CJMSException

`mqlong getNumberOfDeliverableMessages2(mqcstring clientID, mqcstring subscriberID) FMQCONST throw (CJMSException*)`

Returns a mqlong object for the number of deliverable messages on the specified clientID and subscriberID combo.

Parameters:

- clientID - String representing the client id.
- subscriberID - String representing the subscriber id.

Returns: mqlong object for the number of deliverable messages on the specified clientID and subscriberID combo.

Exceptions: CJMSException

`mqlong getNumberOfUndeletedMessages(mqcstring queueName) FMQCONST throw (CJMSException*)`

Returns a mqlong object for the number of undeleted messages on the server for the specified queue name.

Parameters:

- queueName - String representing the queue Name.

Returns: Returns a mqlong object for the number of undeleted messages on the server for the specified queueName.

Exceptions: CJMSException

`mqboolean unsubscribe(mqcstring clientID, mqcstring subscriberID) FMQCONST throw (CJMSException*)`

Unsubcribes a durable subscriber for the specified clientID and subscriberID combo.

Parameters:

- clientID - String representing the client id.
- subscriberID - String representing the subscriber id.

Returns: mqboolean when unsubscribing a durable subscriber for the specified clientID and subscriberID combination.

Exceptions: CJMSException

`mqboolean purgeSubscriptionMessages(mqcstring clientID, mqcstring subscriberID) FMQCONST throw (CJMSException*)`

Purge all messages for a durable subscriber for the specified clientID and subscriberIDcombination.

Parameters:

- clientID - String representing the client id.
- subscriberID - String representing the subscriber id.

Returns: mqboolean when purging all messages for a durable subscriber for the specified clientID and subscriberID combo.

Exceptions: CJMSException

`mqboolean createTopic(const CTopicMetaData* topicMetaData) FMQCONST throw (CJMSException*)`

Creates a topic with the specified topic meta-data on the MQ Server.

Parameters:

- topicMetaData – pointer to CTopicMetaData object.

Returns: mqboolean value for success or failure from the server.

Exceptions: CJMSException

`mqboolean createQueue(const CQueueMetaData* queueMetaData) FMQCONST throw (CJMSException*)`

Creates a queue with the specified queue meta data on the MQ Server.

Parameters:

- queueMetaData – pointer to CQueueMetaData object.

Returns: mqboolean value for success or failure from the server.

Exceptions: CJMSException

`mqboolean deleteTopic(mqcstring topicName) FMQCONST throw (CJMSException*)`

Delete a topic with the specified topic name.

Parameters:

- topicName - String representing the topic name.

Returns: mqboolean value for success or failure from the server.

Exceptions: CJMSException

`mqboolean deleteQueue(mqcstring queueName) FMQCONST throw (CJMSException*)`

Delete a queue with the specified queue name.

Parameters:

- queueName - String representing the queue name.

Returns: mqboolean value for success or failure from the server.

Exceptions: CJMSException

`mqboolean deleteTopicConnectionFactory(mqcstring tcfName) FMQCONST throw (CJMSException*)`

Delete a topicConnectionFactory with the specified topicConnectionFactory name.

Parameters:

- tcfName - String representing the topic connection factory name.

Returns: mqboolean value for success or failure from the server.

Exceptions: CJMSException

`mqboolean deleteQueueConnectionFactory(mqcstring qcfName) FMQCONST throw (CJMSException*)`

Delete a queueConnectionFactory with the specified queueConnectionFactory name.

Parameters:

- qcfName - String representing the queue connection factory name.

Returns: mqboolean value for success or failure from the server.

Exceptions: CJMSException

`mqboolean deleteAdminConnectionFactory(mqcstring acfName) FMQCONST throw (CJMSException*)`

Delete a adminConnectionFactory with the specified adminConnectionFactory name.

Parameters:

- acfName -String representing the admin connection factory name.

Returns: mqboolean value for success or failure from the server.

Exceptions: CJMSException

```
mqboolean createTopicConnectionFactory( const CTopicConnectionFactoryMetaData *tcfMetaData)
FMQCONST throw (CJMSException*)
```

Creates a topicConnectionFactory with the specified topicConnectionFactory meta data on the MQ Server.

Parameters:

- tcfMetaData – pointer to CTopicConnectionFactoryMetaData object.

Returns: mqboolean value for success or failure from the server.

Exceptions: CJMSException

```
mqboolean createQueueConnectionFactory( const CQueueConnectionFactoryMetaData *qcfMetaData)
FMQCONST throw (CJMSException*)
```

Creates a queueConnectionFactory with the specified queueConnectionFactory meta data on the MQ Server.

Parameters:

- qcfMetaData – pointer to CQueueConnectionFactoryMetaData object.

Returns: mqboolean value for success or failure from the server.

Exceptions: CJMSException

```
mqboolean createAdminConnectionFactory( const CAdminConnectionFactoryMetaData *acfMetaData)
FMQCONST throw (CJMSException*)
```

Creates a adminConnectionFactory with the specified adminConnectionFactory meta data on the MQ Server.

Parameters:

- acfMetaData – pointer to CAdminConnectionFactoryMetaData object.

Returns: mqboolean value for success or failure from the server.

Exceptions: CJMSException

```
CEnumeration* getCurrentUsers() FMQCONST throw (CJMSException*)
```

Returns a pointer to enumeration of all the current users on the MQ Server.

Parameters: None

Returns: Pointer to CEnumeration object of all the current users on the MQ Server.

Note: The CEnumeration object contains all the current users.The users can be retrieved from the CEnumeration object using nextElement method.The nextElement method returns void* and it should be type casted to const char *.

Exceptions: CJMSException

```
mqboolean restartServer() FMQCONST throw (CJMSException*)
```

Restarts the MQ Server.

Parameters: None

Returns: mqboolean value for success or failure from the server.

Exceptions: CJMSException

```
mqboolean shutdownServer() FMQCONST throw (CJMSException*)
```

Shuts down the MQ Server.

Parameters: None

Returns: mqboolean value for success or failure from the server.

Exceptions: CJMSException

```
mqboolean shutDownActiveHAServer() FMQCONST throw (CJMSException*)
```

Shuts down the Active HA MQ Server.

Parameters: None

Returns: mqboolean value for success or failure from the server.

Exceptions: CJMSException

```
mqboolean shutDownPassiveHAServer() FMQCONST throw (CJMSException*)
```

Shuts down the Passive HA MQ Server.

Parameters: None

Returns: mqboolean value for success or failure from the server.

Exceptions: CJMSException

```
mqboolean purgeQueueMessages2(mqcstring queueName, const mqboolean forcefully) FMQCONST
    throw (CJMSException*)
```

Purges all the messages on the specified queue forcefully.

Parameters:

- queueName - String representing the queue name.
- forcefully - boolean representing whether the messages in the queue have to be purged forcefully, irrespective of active consumers.

Returns: mqboolean value for success or failure from the server.

Exceptions: CJMSException

```
mqboolean purgeQueueMessages1(mqcstring queueName) FMQCONST throw (CJMSException*)
```

Purges all the messages on the specified queue,if no active consumer is present.

Parameters:

- queueName - String representing the queue name.

Returns: mqboolean value for success or failure from the server.

Exceptions: CJMSException

```
mqboolean showStatusOfAllQueues() FMQCONST throw (CJMSException*)
```

Shows the status of all the queues on the server.

Parameters: None

Returns: mqboolean value for success or failure from the server.

Exceptions: CJMSException

```
mqint deleteMessagesOnServer(mqcstring queueName, const mqlong startIndex, const mqlong
endIndex, const mqint priority) FMQCONST throw (CJMSException*)
```

Delete messages on the specified queue with start index upto the end index and with specified priority.

Parameters:

- queueName - String representing the queue name.
- startIndex - mqlong start index.
- endIndex - mqlong end index.
- Priority - Mqint message priority.

Returns: Mqint representing the number of messages deleted on the server.

Exceptions: CJMSException

```
mqboolean loadAdminObjects() FMQCONST throw (CJMSException*)
```

Load admin objects on the server.

Parameters: None

Returns: mqboolean value for success or failure from the server.

Exceptions: CJMSException

CHashTable

This class implements a hashtable, which maps keys to values. Any non-null mqobject can be used as a value and any non-null mqcstring can be used as a key.

Inheritance Hierarchy

None

Subclasses

None

Constructors

`CHashTable() throw (CJMSException *);`

Constructs a new, empty hashtable.

Parameters: None

Methods

`mqobject Put(mqcstring key, mqobject value) throw (CJMSException *)`

Maps the specified key to the specified value in this hashtable. Neither the key nor the value can be null.

Parameters:

- Key –The string which acts as a key.
- Value - the value mapped to the key which should be casted to mqobject

Returns: mqobject

Exceptions: `CJMSException`

`mqobject Get(mqcstring key) FMQCONST throw (CJMSException *)`

Returns the value to which the specified key is mapped in this hashtable.

Parameters:

- Key – The string which acts as a key.

Returns: mqobject

Exceptions: `CJMSException`

`mqobject RemoveElement(mqcstring key) throw (CJMSException *)`

Removes the key (and its corresponding value) from this hashtable. This method does nothing if the key is not in the hashtable.

Parameters:

- Key – The string which acts as a key.

Returns: mqobject

Exceptions: JMSException

```
mqboolean ContainsValue(mqobject value) throw (CJMSException *)
```

Checks if the given value exists in this hashtable.

Parameters:

- Value - the value mapped to the key which should be casted to mqobject

Returns:

- Mqboolean – True/False

Exceptions: `CJMSException`

```
mqboolean ContainsKey(mqcstring key) throw (CJMSException *)
```

Checks if the given key exists in this hashtable.

Parameters:

- Key – The string which acts as a key.

Returns:

- Mqboolean - True/False

Exceptions: `CJMSException`

CHashTableEnumerator

This is the enumeration class that gets the contents of the CHashTable as an enumerated object.

Inheritance Hierarchy

None

Subclasses

None

Constructors

```
CHashTableEnumerator(struct _HashtableEnumerator *htenum); throw (CJMSException *);
```

Parameters:

- htenum – HashtableEnumerator structure defined in C runtime.

Methods

`mqboolean hasMoreElements() FMQCONST`

Tests if this enumeration contains more elements.

Parameters: None

Returns: Returns true if enumeration contains more elements otherwise it returns false

`mqcstring nextKeyElement() FMQCONST`

Returns the nextKey value of this enumeration if this enumeration object has at least one more element to provide.

Parameters: None

Returns:

- mqcstring - Returns the next Key value

`mqobject nextValueElement() FMQCONST`

Returns the next value element of this enumeration if this enumeration object has at least one more element to provide

Parameters: None

Returns:

- mqobject - Returns the next value Element

CLogHandler

This class is used for logging to a file. It is a singleton class and only one instance of Logger exists for the whole application.

Inheritance Hierarchy

None

Subclasses

None

Constructors

`CLogHandler() throw (CJMSException*);`

Default constructor

Methods

`static mqboolean setLoggerName(mqstring name) throw (CJMSException*)`

Sets file name [in which data is logged] to name. The default file name is cclient.

For example, CLogHandler::setLoggerName("Publisher");

Parameters: Logger name

Returns: Boolean returning true or false

Exceptions: `CJMSException`

`static CLogHandler* getLogHandler() throw (CJMSException*)`

This function gets the reference to the LogHandler, using which data can be logged from application.

Parameters: None

Returns: Reference to CLogHandler

Exceptions: `CJMSException`

`mqboolean setTraceLevel(mqstring level)`

The trace level can also be set using this function. The legal values for the "level" are:

1. ERROR
2. INFO
3. DEBUG

Parameters

- level - Trace level in mqstring

Returns: Boolean returning true or false

`mqboolean logData(mqstring info, ...) throw (CJMSException*)`

It logs the given data to the log file. This function is variable argument function. The usage of this function is similar to printf function.

Example

```
CLogHandler *logger = CLogHandler::getLogHandler();
logger->logData("Logging from file - %s, func -%s", "FileName", "funcName");
```

In log file it is printed as follows:

```
Fri Dec 19 14:41:25 2008 :APPL: Logging from file - FileName, func –funcName
```

Parameters: User defined

Returns: Boolean returning true or false

Exceptions: `CJMSException`

CCSPManager

This class is used to create CSPBrowser, which is used for browsing messages stored in CSP cache.

Inheritance Hierarchy

None

Subclasses

None

Constructors

`CCSPManager(mqcstring cspPath)`

Constructors are used for creating CCSPManager class.

Parameters: The path to the CSP has to be given as parameter to the constructor.

Methods

`CCSPBrowser* createCSPBrowser() throw (CJMSException *)`

Creates CCSPBrowser for the location specified for the manager.

Parameters: None

Returns: Returns CCSPBrowser pointer.

Exceptions: `CJMSException`

CCSPBrowser

CSP Browser is used to browse the messages stored in Client-Side Persistent Cache. While browsing, messages are read from CSP and are not deleted. So, next time when the Server is re-connected, messages are sent to the Server. The classes used for CSP Browsing are CCSPBrowser and CCSPEnumeration.

Inheritance Hierarchy

None

Subclasses

None

Constructors

`CCSPBrowser(struct _CSPBrowser *m_cspBrowser)`

Parameters:

- m_cspBrowser – CSPBrowser structure defined in C runtime.

Methods

`CEnumeration* getAllConnections() FMQCONST throw (CJMSException *)`

This function returns the enumeration of all the Connection IDs whose messages are stored in CSP

Parameters: None

Returns: returns the enumeration of all the Connection IDs

Note: The CEnumeration object contains all the Connection IDs whose messages are stored in CSP. The names can be retrieved from the CEnumeration object using nextElement method. The nextElement method returns void* and it should be type casted to const char *.

Exceptions: `CJMSException`

`CEnumeration* getTopicsForConnection(mqcstring connectionID) FMQCONST throw (CJMSException *)`

This function returns the Enumeration of all the topic names for the specified connection with clientID as connectionID.

Parameters

- connectionID - clientID as string

Returns: returns the enumeration of all the topic names

Note: The CEnumeration object contains all the topic names for the specified connection.The names can be retrieved from the CEnumeration object using nextElement method.The nextElement method returns void* and it should be type casted to const char *.

Exceptions: `CJMSException`

`CEnumeration* getQueuesForConnection(mqcstring connectionID) FMQCONST throw (CJMSException *)`

This function returns the Enumeration of all the queue names for the specified connection with clientID as connectionID.

Parameters

- connectionID - clientID as string

Returns: returns the enumeration of all the queue names

Note: The CEnumeration object contains all the queue names for the specified connection. The names can be retrieved from the CEnumeration object using nextElement method. The nextElement method returns void* and it should be type casted to const char *.

Exceptions: CJMSException

```
CCSPEnumeration* browseMessagesOnQueue(mqcstring queueName, mqboolean checkTransacted)
FMQCONST throw (CJMSException *)
```

This function returns the enumeration of messages stored in the queue specified by queueName. It searches for messages in queue in all the existing connections.

Parameters

- queueName - Name of the queue to browse as mqcstring
- checkTransacted - If checkTransacted is TRUE, transacted messages are also browsed.

Returns: returns the enumeration of all the messages in queue

Exceptions: CJMSException

```
CCSPEnumeration* browseMessagesOnQueue(mqcstring connectionID, mqcstring queueName, mqboolean
checkTransacted) FMQCONST throw (CJMSException *)
```

This function returns the enumeration of messages stored in the queue specified by queueName. It searches for messages in queue for the connection specified by connectionID. If **checkTransacted** is TRUE, transacted messages are also browsed.

Parameters

- connectionID - clientID as string
- queueName - Name of the queue to browse as mqcstring
- checkTransacted - If checkTransacted is TRUE, transacted messages are also browsed.

Returns: returns the enumeration of all the messages in queue

Exceptions: CJMSException

```
CCSPEnumeration* browseMessagesOnTopic(mqcstring topicName,mqboolean checkTransacted)
FMQCONST throw (CJMSException *)
```

This function returns the enumeration of messages stored in the topic specified by topicName. It searches for messages in topic in all the existing connections.

Parameters

- topicName - Name of the topic to browse as mqcstring
- checkTransacted - If checkTransacted is TRUE, transacted messages are also browsed.

Returns: returns the enumeration of all the messages in topic

Exceptions: `CJMSException`

`CCSPEnumeration* browseMessagesOnTopic(mqcstring connectionID, mqcstring topicName, mqboolean checkTransacted) FMQCONST throw (CJMSException *)`

This function returns the enumeration of messages stored in the topic specified by topicName. It searches for messages in topic for the connection specified by connectionID.

Parameters

- connectionID - clientID as string
- topicName - Name of the topic to browse as mqcstring
- checkTransacted - If checkTransacted is TRUE, transacted messages are also browsed.

Returns: returns the enumeration of all the messages in topic

Exceptions: `CJMSException`

`mqlong numberOfMessagesinQueue(mqcstring connID, mqcstring queueName,mqboolean checkTransacted) FMQCONST throw (CJMSException *)`

This function returns the number of messages stored in the queue specified by queueName for a given connection with clientID connID.

Parameters

- connID - clientID as string
- queueName - Name of the queue to browse as mqcstring
- checkTransacted - If checkTransacted is TRUE, transacted messages are also counted.

Returns: returns the messages count in mqlong

Exceptions: `CJMSException`

`mqlong numberOfMessagesinTopic(mqcstring connID, mqcstring topicName, mqboolean checkTransacted) FMQCONST throw (CJMSException *)`

This function returns the number of messages stored in the topic specified by topicName for a given connection with clientID connID.

Parameters

- connID - clientID as string
- topicName - Name of the topic to browse as mqcstring
- checkTransacted - If checkTransacted is TRUE, transacted messages are also counted.

Returns: returns the messages count in mqlong

Exceptions: `CJMSException`

CCSPEnumeration

This is the enumeration class that gets the list of messages from CSP Cache.

Inheritance Hierarchy

None

Subclasses

None

Constructors

`CCSPEnumeration(CSPEnumeration cspe);`

Parameters:

- cspe – CSPEnumeration structure defined in C runtime.

Methods

`mqboolean hasMoreElements() FMQCONST  throw (CJMSException *)`

This function checks whether more elements are available in the enumeration.

Parameters: None

Returns: returns true/false

Exceptions: `CJMSException`

`CMessage* nextElement() FMQCONST  throw (CJMSException *)`

This function returns the next element of this enumeration if this enumeration object has at least one more element to provide.

Parameters: None

Returns: returns the next CMessage

Exceptions: `CJMSException`

Chapter 5: Large Message Support

With Large Message Support (LMS) in FioranoMQ, clients can transfer large files in the form of large messages with theoretically no limit on the message size. Large messages can be attached with any JMS message and the client can be sure of a reliable and secure transfer of the message through FioranoMQ Server.

The following classes and functions are used in LMS.

CFioranoConnection

These two functions are used for resuming any pending large messages.

`CRecoverableMessagesEnum *getUnfinishedMessagesToSend() FMQCONST`

This function Returns the enumeration of Messages whose transfers are pending as they were not fully sent.

`CRecoverableMessagesEnum *getUnfinishedMessagesToReceive() FMQCONST;`

This function Returns the enumeration of Messages which were not fully received and need to be resumed.

CRecoverableMessagesEnum

This class is the enumeration of Unfinished Messages.

Inheritance Hierarchy

None

Subclasses

None

Constructors

`CRecoverableMessagesEnum(RMEnum enumr);`

Parameters:

- enumr - _RecoverableMessagesEnum structure defined in C runtime.

Methods

`bool hasMoreElements() FMQCONST`

Returns true if there are more elements in the enumeration, false otherwise

Parameters: None

Returns: Returns true if there are more elements in the enumeration, false otherwise

`CMessage *nextElement() FMQCONST`

Return the next element (CMessage Object) of the enumeration.

Parameters: None

Returns: Returns CMessage Object

CLargeMessage

Inheritance Hierarchy

None

Subclasses

CMessage

Constructors

Default

Methods

`virtual void setLMStatusListener(CLMStatusListener *listener, const mqint updateFrequency)`

Sets the status listener for the message. This function is used to know the status of message transfer asynchronously.

Parameters

- listener
- CLMStatusListener object
- updateFrequency

Returns: void

`virtual CLMStatusListener *getLMStatusListener() FMQCONST`

Gets the status listener for the message.

Parameters: None

Returns: CLMStatusListener object

```
virtual void SaveTo(mqcstring fileName, bool isBlocking)
```

Saves the contents of the message in the file specified.

Parameters

- fileName - file name to save.
- isBlocking - boolean for blocking or non blocking

Returns: void

```
virtual void resumeSaveTo(bool isBlocking)
```

Resumes saving the contents of the message in the file specified.

Parameters:

- isBlocking: boolean for blocking or non blocking

Returns: void

```
virtual void resumeSend()
```

Resumes an incomplete transfer. This function is used to resume a message transfer which could not be completed earlier either due to some internal error or due to some problem at the client side.

Parameters: None

Returns: void

```
virtual void cancelAllTransfers()
```

Cancels all message transfers which are currently transferring this message. A cancelled transfer also removes the resume information of the transfer. Hence a transfer once cancelled cannot be resumed.

Parameters: None

Returns: void

```
virtual void cancelTransfer(mqint consumerID)
```

Cancels the transfer specified by the consumerID. A cancelled transfer also removes the resume information of the transfer. Hence a transfer once cancelled cannot be resumed.

Parameters

- consumerID - Id of the consumer for which the transfer has to be stopped.

Returns: void

```
virtual void suspendAllTransfers ()
```

Suspends all the message transfers which are transferring this large message temporarily. Suspending a transfer only stops the thread which is doing the message transfer and does not delete resume related information. Hence, a suspended transfer can be resumed using resume functions.

Parameters: None

Returns: void

`virtual void suspendTransfer(mqint consumerID)`

Suspends the transfer specified by the consumerID temporarily. Suspending a transfer only stops the thread which is doing the message transfer and does not delete resume related information. Hence, a stopped transfer can be resumed using resume functions

Parameters

- consumerID - Id of the consumer for which the transfer has to be suspended.

Returns: void

`virtual void setFragmentSize(const mqint size)`

Sets the fragment size for the message.

Parameters

- size – size of the fragment in int

Returns: mqlong

`virtual int getFragmentSize() FMQCONST`

Gets the fragment size of the message.

Parameters: None

Returns:

- Int – returns the size

`virtual void  setWindowSize(const mqint size)`

Sets the frequency after which acknowledgement will be sent

Parameters

- size –size of the window in int

Returns: void

`virtual int getWindowSize() FMQCONST`

Gets the window size of the message.

Parameters: None

Returns

- size – size of the window in int

```
virtual void setRequestTimeoutInterval(const mqlong timeout)
```

Sets the time until which the sender will wait for message transfer to start

Parameters

- timeout - time in mqlong (in milliseconds)

Returns: void

```
virtual mqlong getRequestTimeoutInterval() FMQCONST
```

Gets the time until which the sender will wait for message transfer to start

Parameters: None

Returns: mqlong

```
virtual void setResponseTimeoutInterval(const mqlong responseInterval)
```

Sets the time until which the sender/receiver will wait for message from the other end.

Parameters

- responseInterval - responseInterval time as mqlong (in milliseconds)

Returns: void

```
virtual mqlong getResponseTimeoutInterval() FMQCONST
```

Gets the time until which the sender/receiver will wait for messages from the other end.

Parameters: None

Returns: mqlong

CLMStatusListener

This class listens to the status of the Large Message received or sent. This has one virtual function which has to over ridden in Status Listener implementation. Refer to the lms samples for the sample implementation of this class.

Inheritance Hierarchy

None

Subclasses

None

Constructors

Default

Methods

`virtual void onLMStatus(CLMTransferStatus *status, bool exception) = 0`

Notifies the user with large message status. The user is expected to override this function with the required functionality.

Parameters

- status - CLMTransferStatus object which holds the status.

Exception: Boolean indicating whether exception occured or not.

Returns: Void

CLMTransferStatus

This class provides methods that provide information about the status of the Message transfer such as number of bytes transferred, number of bytes left to be transferred, etc.

Inheritance Hierarchy

None

Subclasses

None

Constructors

`CLMTransferStatus(LMTransferStatus status)`

Parameters:

- enumr - LMTransferStatus structure defined in C runtime.

Methods

`mqlong getBytesTransferred() FMQCONST`

Returns the number of bytes transferred.

Parameters: None

Returns: number of bytes transferred

`mqlong getBytesToTransfer() FMQCONST`

Returns the number of bytes to be transferred.

Parameters: None

Returns: number of bytes to be transferred.

`mqlong getLastFragmentID() FMQCONST`

Returns the ID of the last fragment.

Parameters: None

Returns: ID as mqlong

`float getPercentageProgress() FMQCONST`

The percentage of the progress of message transfer.

Parameters: None

Returns: Returns the percentage of the progress message transfer in float

`mqbyte getStatus()`

Returns the status of the Message transfer.

- LM_TRANSFER_NOT_INIT or 1

  Indicates that the transfer has not yet started
- LM_TRANSFER_IN_PROGRESS or 2

  Indicates that transfer is currently in progress
- LM_TRANSFER_DONE or 3

  Indicates that the transfer is complete for one consumer
- LM_TRANSFER_ERR or 4

  Indicates that an error occurred during the transfer
- LM_ALL_TRANSFER_DONE or 5

  Indicates that the transfer is complete for all consumers

Parameters: None

Returns: Returns the status of the Message transfer.

`bool isTransferComplete() FMQCONST`

Returns true if transfer completes, else returns false.

Parameters: None

Returns: true if transfer completes, else returns false.

`bool isTransferCompleteForAll() FMQCONST`

Returns true if all the transfers are completed, else returns false.

Parameters: None

Returns: true if all the transfers are completed, else returns false

`CLargeMessage *getLargeMessage() FMQCONST`

Returns the large message for which this status is created.

Parameters: None

Returns: The large message

`int getConsumerID() FMQCONST`

Returns the consumerID of the connection that is being used.

Parameters: None

Returns: Returns the consumerID

Chapter 6: Message Compression

Message compression is a functionality that allows messages sent through FioranoMQ to be compressed when sending and decompressed, to their original size, prior to delivery to consumers.

Compression has the advantage of improving performance. Less bandwidth is used during message transfer. Memory and storage requirements on the server are reduced as well. This function is important for performance-sensitive applications operating over WAN links. Many data compression implementations have been developed in the past, of which the Zlib implementation is, by far, the most significant one. The Fiorano compression implementation is based on Zlib Compressed Data Format Specification Version.

This specification defines a lossless compression data format. The advantages of this compression implementation, as per specification, are:

- It is independent of CPU type, operating system, file system and character set.
- Can be produced or consumed by an arbitrarily long sequentially presented input data stream, using a bounded amount of intermediate storage.
- Can be implemented readily in a manner not covered by patents.
- Can use a number of different compression methods.

In FioranoMQ C RTL, the Zlib implementation is provided using the zlib general purpose compression library. The zlib compression library provides in-memory compression and decompression functions. This implementation provides 'deflate' and 'inflate' mechanisms using different compression levels and different compression strategies.

- Compression level is the amount of compression required.
- Compression strategy is the actual compression method used.

The default strategy uses a combination of the LZ77 algorithm and Huffman coding.

Message Compression Characteristics

FioranoMQ provides message compression on a ‘per message’ as well as on ‘per destination’ basis. In ‘per message’ compression, clients can enable or disable compression for each message. In ‘per destination’ compression, all messages sent to a particular destination (topic or queue) are compressed. Client applications can choose compression levels and strategies from Zlib specifications.

The available options are:

- Z_NO_COMPRESSION
- Z_BEST_SPEED (fastest compression)
- Z_BEST_COMPRESSION
- Z_DEFAULT_COMPRESSION

There are ten possible compression levels (0-9) available, where Z_BEST_SPEED is defined as 1 and Z_BEST_COMPRESSION is defined as 9.

The possible values for the compression strategy are:

- Z_FILTERED: Compression strategy best used for data consisting primarily of small values with random distribution. It enforces more Huffman coding and less string matching.
- Z_HUFFMAN_ONLY: Enforces more Huffman coding.
- Z_RLE: Limits match distances to one (run-length encoding). Z_RLE is designed to be almost as fast as Z_HUFFMAN_ONLY, but give better compression for PNG image data.
- Z_FIXED: Prevents the use of dynamic Huffman codes, allowing for a simpler decoder for special applications.
- Z_DEFAULT_STRATEGY: This uses a combination of the LZ77 method and Huffman coding.

Compression support provided helps a client application to decide on the optimum compression level and strategy by providing APIs to check compression ratios of messages sent and/or received. Compression involves compressing only the payload of the message and not its JMS header.

Refer the samples in PTP/Message Compression & PubSub/Message Compression on enabling compression on PerMessage or PerDestination basis in the application code.

Chapter 7: Using Sample Programs

This chapter explains the various steps involved in running the sample programs which are shipped as part of the installer.

Organization of Samples Provided

The sample programs illustrating the use of C++RTL for PubSub and PTP Operations are organized into the following categories.

- **Pubsub:** This directory contains sample programs, which illustrate basic JMS Publish/Subscribe functionality, using the C++RTL.
- **Ptp:** This directory contains sample programs which illustrate JMS send-receive mechanism using the C++RTL.
- **Unified:** This directory contains sample programs which illustrate the JMS Send-Receive functionality using unified APIs.

Compiling and Running the Samples

To run the samples using FioranoMQ, perform the following steps:

Compile each of the source files, using the script file, cppclientbuild.bat, in the fmq\clients\cpp\native\scripts folder. Environment settings like path, compiler settings can be modified in the cppclientbuild.bat (.sh script for UNIX platform) file.

For information on compiling and running these samples please refer to the readme file in the *cpp\native\samples* directory of FioranoMQ installation.

Operating Environments

The JMS C/C++ RTL supports client applications for each of the following operating systems. *Table 1 lists the compiler for each client platform.*

Table 1. Message Service Client for C/C++ platforms and compilers

Operating System	**Compiler**
Microsoft Windows XP/Server 2000/Server 2003 (Intel 32 bit)	Microsoft Visual C++ .NET 2003 (Visual C++ 7.1), Microsoft Visual C++ 2005, Microsoft Visual C++ 2008, Microsoft Visual C++ 2010
Fedora/Cent OS/Open Suse Linux (Intel 32 bit)	gcc 4.1.2
Fedora/Cent OS/Open Suse Linux (x86_64)	gcc 4.1.2
Sun Solaris 5.10 (Intel 32 bit)	gcc 3.4.3
Sun Solaris 5.10 (SPARC)	gcc 3.4.3

Limitations of C++ RTL

Exception handling is limited to `CJMSException` and any variation is implemented by varying the error Code and error description parameters.

C++RTL provides a limited set of APIs for the end user, which just allows messaging using Pub/Sub or Sender/Receiver.

Chapter 8: Native C++ Runtime Examples

This chapter describes usage of C++ Runtime for connecting to FioranoMQ Server. The various sample programs illustrates the use of simple Point-To-Point and Publish-Subscribe operations.

PTP

This directory contains samples which demonstrate the following functionality over PTP using C++ Runtime.

- Admin
- Basic Send Receive
- Browser
- HTTP
- Message Selector
- Multi-thread PTP
- RequestReply
- Basic
- TimedRequestReply
- SSL
- Transaction
- cspBrowser
- lms
- nonJndi
- revalidateConnections
- serverlessMode
- DeadMessageQueue
- Message Compression - PerDestination
- Message Compression - PerMessage

PubSub

This directory contains samples which demonstrate the following functionality over PubSub using C++ Runtime.

- Admin
- Basic Pub Sub
- Durable Subscriber
- HTTP

- Message Selector
- Multi-thread PubSub
- RequestReply
- Basic
- TimedRequestReply
- SSL
- Transaction
- cspBrowser
- lms
- nonJndi
- revalidateConnections
- serverlessMode
- DeadMessageQueue
- Message Compression - PerDestination
- Message Compression - PerMessage

Unified

This directory contains samples which demonstrate the following functionality over Unified APIs using C++ Runtime.

- NonJndi
- Sendreceive

These samples are available in %FMQ_DIR%\clients\cpp\native\samples directory. The %FMQ_DIR%\clients\cpp\native\script directory contains a script called build_samples.bat (.sh for UNIX platform) which compiles the C++ programs.

Platforms Supported

Currently C++ libraries are supported on the following OS platforms: Windows, Linux (32 and 64 bit), Solaris x86, Solaris Sparc, and HP-UX.

Building and Running C++ Applications

FioranoMQ C++ library package comes with a comprehensive list of sample applications covering important features of PTP and PubSub messaging domains. The Non-java client library packages should be downloaded along with the FioranoMQ installer and extracted to the 'fmq' directory of Fiorano_Home.

For getting started with a basic send/receive sample, while compiling please ensure to link the required libraries and header files as mentioned in the script files under the scripts directory (located *at $FMQ_DIR/clients/cpp/native/scripts for compiling C++ samples*). In order to compile all the samples at one shot, please use the `build_samples.bat(.sh for unix platform)` or use the `cppclientbuild.bat(.sh)` for compiling a particular sample.

Both static and dynamic libraries are provided for all supported platforms. To run any C++ samples, set the LD_LIBRARY_PATH environment variable in ~/.bashrc profile to both C and C++ library directories for UNIX platforms,

Example: LD_LIBRARY_PATH = $FMQ_DIR/clients/c/native/lib:$FMQ_DIR/clients/cpp/native/lib

For Windows platform, set the PATH environment variable to the location of c/native/lib and cpp/native/lib folders.

Example: PATH=%FMQ_DIR%\clients\c\native\lib;%FMQ_DIR%\clients\cpp\native\lib

Note: For Message Compression support in Windows zlibwapi.lib/.dll provides dynamic linking with fmq cpprtl. In Linux & Solaris platforms, the same is provided by libz.a static library and libz.so shared object.The above libraries are present in the %FMQ_DIR%\clients\c\native\lib directory.

PTP Samples

Admin

This directory contains one sample program which illustrates basic JMS Administration API functionality using the FioranoMQ C++ Runtime Library.

- **AdminTest.cpp** - Creates an Admin Connection with the MQServer and gets an MQAdminService object to create and delete Queues and QueueConnectionFacotries and retrieves information of users connected from the server.

To run these samples using FioranoMQ, perform the following steps:

1. Compile each of the source files. The %FMQ_DIR%\clients\cpp\native\scripts directory contains a script called cppclientbuild.bat which compiles the C++ program.
2. Run the AdminTest by executing the *AdminTest.exe* executable file.

Basic

This directory contains two sample programs which illustrate JMS Send-Receive mechanism using the FMQ C++ Runtime Library.

- **Sender.cpp** - Reads strings from standard input and sends the text messages on the queue "PrimaryQueue".
- **Receiver.cpp** - Implements a synchronous blocking receiver, which listens on the queue "PrimaryQueue", and prints the text of the received text messages on the console.

To run these samples using FioranoMQ, perform the following steps:

1. Compile each of the source files. The %FMQ_DIR%\clients\cpp\native\scripts directory contains a script called cppclientbuild.bat which compiles the C++ program.
2. Run the Sender by executing the Sender.exe executable file.
3. Run the receiver by executing the *Receiver.exe* file.

Browser

This directory contains two sample programs which illustrate basic JMS Browser functionality using the FioranoMQ C++ Runtime Library.

- **QSender.cpp** - Reads strings from standard input and sends them on the queue "PrimaryQueue".
- **Browser.cpp** - Implements a browser, which is used to browse the messages on the queue "PrimaryQueue", and prints out the received messages.

To run these samples using FioranoMQ, perform the following steps:

1. Compile each of the source files. For convenience, compiled versions of the sources are included in this directory. The %FMQ_DIR%\clients\cpp\native\scripts directory contains a script called cppclientbuild.bat which compiles the C++ program.
2. Run the Sender by executing the QSender.exe executable file. Send some messages before running the browser application
3. Run the browser by executing the *Browser.exe* file.

Csp Browser

This directory contains two sample programs which illustrate basic JMS CspBrowser functionality using the FioranoMQ CPP Runtime Library.

1. Sender.cpp

   Reads strings from standard input and sends them on the queue "PrimaryQueue".

2. Browser.cpp

   Implements a browser, which is used to browse the messages on the cspcache, and prints out the received messages.

   To run these samples using FioranoMQ, perform the following steps:

   a. Compile each of the source files.

The %FMQ_DIR%\clients\cpp\native\scripts directory contains a script called cclientbuild.bat which compiles the CPP program.

b. Run the Sender by executing the Sender.exe executable file. Send some messages before running the browser application.

c. Run the asynchronous receiver by executing the Browser.exe file.

DeadMessageQueue

This directory contains three sample programs which illustrate functionality of Dead Message Queue using the FioranoMQ C Runtime Library.

Note: Before running the given sample, enable the following configuration through WMT:

1. To use WMT, open a web browser and type http://localhost:1780
2. Go to JMX -> ConfigureFMQServer -> fiorano->mq->ptp->queuingSubSystem->EnableDMQOnAllQueues to true
    a. Sender.cpp - Reads strings from standard input and sends them on the queue "PrimaryQueue".Some messages expire after the TTL and are sent to "SYSTEM_DEADMESSAGES_QUEUE".
    b. Receiver.cpp - Implements an asynchronous listener, which listens on the queue "PrimaryQueue", and prints out the received messages.
    c. ReceiverDMQ.cpp - Implements an asynchronous listener, which listens on the queue “SYSTEM_DEADMESSAGES_QUEUE”, and prints out the received messages and the actual destination of the messages.

To run this sample using FioranoMQ, perform the following steps:

1. Compile the source file. The %FMQ_DIR%\clients\cpp\native\scripts directory contains a script called cppclientbuild.bat which compiles the C++ program.
2. Run the Sender by executing the Sender.exe executable file.
3. Run the Receiver by executing the Receiver.exe executable file.
4. Run the ReceiverDMQ by executing the ReceiverDMQ.exe executable file.

HTTP

This directory contains two sample programs which illustrate the use of HTTP protocol for basic JMS PTP functionality using the FioranoMQ C++ Runtime Library.

- **QReceiver.cpp** - Receives messages asynchronously on primaryQueue. This program implements a synchronous listener to listen for messages published on the queue “PrimaryQueue”.
- **QSender.cpp** - Implements a client application publishing user specified data on primaryQueue. This program reads strings from standard input and publishes them on the Queue “PrimaryQueue”.

To run this sample using FioranoMQ, perform the following steps:

1. Compile the source file. The %FMQ_DIR%\clients\cpp\native\scripts directory contains a script called cppclientbuild.bat which compiles the C++ program.
2. Run the HttpReceiver by executing the QReceiver.exe executable file.
3. Run the HttpSender by executing the *QSender.exe* executable file.

LMS

This directory contains sample programs which illustrate the use of Large Message Support - LMS for basic JMS ptp functionality using the FioranoMQ CPP Runtime Library.

1. LmSender.cpp

   Accept filename to be send from standard input and send them on queue, "PrimaryQueue".
2. LmReceiver.cpp

   Implements an asynchronous listener, which listens on the queue "PrimaryQueue", and create a received message file (default output fileName, "received.zip").

   To run this sample using FioranoMQ, perform the following steps:

   a. Compile the source file.

      The %FMQ_DIR%\clients\CPP\native\scripts directory contains a script called cclientbuild.bat which compiles the CPP program.

   b. Run the LmReceiver by executing the LmReceiver.exe executable file.

   c. Run the LmSender by executing the LmSender.exe executable file.

Message Compression

This directory contains two folders PerDestination and PerMessage

PerDestination

This directory contains three sample programs which illustrate the message compress/uncompress abstraction supplied by the JMS API using a C++ Sender application and Receiver application.

1. Sender.cpp - Sends a file as a compressed message to the server on the "CompressedQueue".
2. Receiver.cpp - Implements an asynchronous listener, which listens on the queue CompressedQueue". Receives the message sent by the sender in the uncompressed original form and writes the data in the message to a file.
3. CreateQueue.cpp - Creates a queue called "CompressedQueue" that has compression enabled on it which means that all messages sent on this queue will be compressed by default.

To run this sample using FioranoMQ, perform the following steps:

1. Compile the source file. The %FMQ_DIR%\clients\cpp\native\scripts directory contains a script called cppclientbuild.bat which compiles the C++ program.

2. Run the CreateQueue by executing the CreateQueue.exe executable file.
3. Run the Sender by executing the Sender.exe executable file.
4. Run the Receiver by executing the Receiver.exe executable file.

PerMessage

This directory contains two sample programs which illustrate the message compress/uncompress abstraction supplied by the JMS API using a C++ Sender application and Receiver application

1. Sender.cpp - Sends a file as a compressed message to the server on the "PrimaryQueue". Compression is enabled for every message and the default compression level and strategy are used.
2. Receiver.cpp - Implements an asynchronous listener, which listens on the queue "PrimaryQueue". Receives the message sent by the sender in the uncompressed original form and writes the data in the message to a file.

To run this sample using FioranoMQ, perform the following steps:

1. Compile the source file. The %FMQ_DIR%\clients\cpp\native\scripts directory contains a script called cppclientbuild.bat which compiles the C++ program.
2. Run the Sender by executing the Sender.exe executable file.
3. Run the Receiver by executing the Receiver.exe executable file.

MsgSel

This directory contains two sample programs which illustrate the use of message selectors using the FioranoMQ C++ Runtime Library.

- **QSelSender.cpp** - Selector sends messages with the string property "name" and an int property "value", set differently for 3 consecutive messages.
- **SelReceive.cpp** - Implements a synchronous listener, which listens on the queue "primaryqueue" for the messages which match the criteria specified in the message selector, and prints out the received messages.

To run these samples using FioranoMQ, perform the following steps:

1. Compile each of the source files. The %FMQ_DIR%\clients\cpp\native\scripts directory contains a script called cppclientbuild.bat which compiles the C++ program.
2. Run the Sender by executing the QSelSender.exe executable file.
3. Run the synchronous receiver by executing the *QSelReceive.exe* file.

Mtptp

This directory contains one sample programs which illustrate basic JMS Sender/Receiver functionality using the FioranoMQ C++ Runtime Library multithreading support.

- **mtPtp.cpp** - The multithreaded version of basic PTP. Single Sender is created, sends 10 text messages on PrimaryQueueand a single receiver blocking receive with timewait of 1 second reads the messages. Each executes on a separate thread. On receipt of 10 messages, the receiver notifies the main thread to end. Sender and Receiver threads are joined to the main thread.

To run these samples using FioranoMQ, perform the following steps:

1. Compile each of the source files. The %FMQ_DIR%\clients\cpp\native\scripts directory contains a script called cppclientbuild.bat which compiles the C++ program.
2. Run the mtPtp by executing the *mtPtp.exe* executable file.

NonJndi

This directory contains samples to create Queue connection Factory and Queue without using JNDI:

1. Sender.cpp

   Implements a client application publishing user specified data on "SampleQueue" created without using JNDI in persistent mode. It reads strings from standard input and publishes them on the queue, "SampleQueue"
2. Receiver.cpp

   Implements an asynchronous listener, which listens on the queue "SampleQueue", and prints out the received messages.

   To run these samples using FioranoMQ, perform the following steps:

   a. Compile each of the source files.

      The %FMQ_DIR%\clients\cpp\native\scripts directory contains a script called cppclientbuild.bat which compiles the C++ program.

   b. Run the Sender by executing the Sender.exe executable file.

   c. Run the asynchronous receiver by executing the Receiver.exe file.

Reqrep

This directory contains two folders Basic and timeout

Basic

This directory contains two samples which illustrate JMS Request-Reply mechanism over Queues.

- **QueueRequestor.cpp** - Reads strings from standard input and sends the text messages on the queue "PrimaryQueue".
- **QueueReplier.cpp** - Implements an asynchronous listener, which listens on the queue "PrimaryQueue", and replies to the received message. The reply is sent on TemporaryQueue.

To run these samples using FioranoMQ, perform the following steps:

1. Compile each of the source files. The %FMQ_DIR%\clients\cpp\native\scripts directory contains a script called cppclientbuild.bat which compiles the C++ program.
2. Run the Replier by executing the QueueReplier.exe executable file.
3. Run the requestor by executing the *QueueRequestor.exe* file.

TimedOut

This directory contains two sample programs which illustrate Timed Request-Reply mechanism over Queues using the FioranoMQ C++ Runtime Library.

- **TimedQueueRequestor.cpp** - Reads strings from standard input and sends the text messages on the queue "PrimaryQueue". The Requestor waits for a specified time for the reply. If the reply is not received within the stipulated time requestor times out.
- **TimedQueueReplier.cpp** - Implements an asynchronous listener, which listens on the queue "PrimaryQueue", and replies on a TemporaryQueue.

To run these samples using FioranoMQ, perform the following:

1. Compile each of the source files. The %FMQ_DIR%\clients\cpp\native\scripts directory contains a script called cppclientbuild.bat which compiles the C++ program.
2. Run the replier by executing the QueueReplier.exe executable file.
3. Run the timed requestor by executing the *TimedQueueRequestor.exe* file.

RevalidateConnections

This directory contains two sample programs which illustrate connection revalidation functionality using the FioranoMQ CPP Runtime Library.

1. Sender.cpp

   Reads strings from standard input and use it to send a request message on the topic "PrimaryQueue". Also, revalidate the connection with the server.

2. Receiver.cpp

   Implements an asynchronous listener, which listens on the queue "PrimaryQueue", and prints out the received messages. Also, revalidate the connection with the server.

   To run these samples using FioranoMQ, perform the following steps:

   a. Compile each of the source files.

      The %FMQ_DIR%\clients\cpp\native\scripts directory contains a script called cppclientbuild.bat which compiles the C++ program.

   b. Run the Sender by executing the Sender.exe executable file.

   c. Run the Receiver by executing the Receiver.exe file.

ServerlessMode

This directory contains two sample programs which illustrate server less mode implementation of JMS Sender/Receiver functionality using the FioranoMQ C++ Runtime Library.

1. Sender.cpp

   Reads strings from standard input and sends them on the topic "PrimaryQueue".

2. Receiver.cpp

   Implements an asynchronous listener, which listens on the topic "PrimaryQueue", and prints out the received messages.

   To run these samples using FioranoMQ, perform the following steps:

   a. Compile each of the source files.

      The %FMQ_DIR%\clients\cpp\native\scripts directory contains a script called cppclientbuild.bat which compiles the C++ program.

   b. (b) Run the Sender by executing the Sender.exe executable file.

   c. (c) Run the asynchronous Receiver by executing the Receiver.exe file.

SSL

This directory contains two sample programs which illustrate the basic JMS Send/Receive functionality over Secure Socket Layer using the FioranoMQ C++ Runtime Library.

- **Sender.cpp** - Reads strings from standard input and sends them on the queue "PrimaryQueue".
- **Receiver.cpp** - Implements a synchronous listener, which listens on the queue "PrimaryQueue", and prints out the received messages.

To run these samples using FioranoMQ, perform the following steps:

1. Compile each of the source files. The %FMQ_DIR%\clients\cpp\native\scripts directory contains a script called cppclientbuild.bat which compiles the C++ program.
2. Run the Sender by executing the Sender.exe executable file.
3. Run the synchronous receiver by executing the *Receiver.exe* file.

Transaction

This directory contains a sample programs which illustrate JMS Transaction functionality using the FioranoMQ C++ Runtime Library.

- **QTransaction.cpp** - Implements the sender and receiver, and uses the commit/rollback functionality to demonstrate JMS Transactions

To run these samples using FioranoMQ, perform the following steps:

1. Compile each of the source files. The %FMQ_DIR%\clients\cpp\native\scripts directory contains a script called cppclientbuild.bat which compiles the C++ program.
2. Run the sample by executing the QTransaction.exe executable file. For proper results from the sample, ensure that there are no messages in the primaryQueue.

PubSub Samples

Admin

This directory contains one sample program which illustrates basic JMS Administration API functionality using the FioranoMQ C++ Runtime Library.

- **AdminTest.cpp** - Creates an Admin Connection with the MQServer and gets an MQAdminService object to create and delete Topics and TopicConnectionFacotries and retrieves information of users connected from the server.

To run these samples using FioranoMQ, perform the following steps:

1. Compile each of the source files. The %FMQ_DIR%\clients\cpp\native\scripts directory contains a script called cppclientbuild.bat which compiles the C++ program.
2. Run the AdminTest by executing the *AdminTest.exe* executable file.

Basic

This directory contains two sample programs which illustrate basic JMS Publisher/Subscriber functionality using the FioranoMQ C++ Runtime Library.

- **Publisher.cpp** - Reads strings from standard input and sends them on the topic "PrimaryTopic".
- **Subscriber.cpp** - Implements a synchronous listener, which listens on the topic "PrimaryTopic", and prints out the received messages.

To run these samples using FioranoMQ, perform the following steps:

1. Compile each of the source files. The %FMQ_DIR%\clients\cpp\native\scripts directory contains a script called cppclientbuild.bat which compiles the C++ program.
2. Run the Publisher by executing the Publisher.exe executable file.
3. Run the synchronous subscriber by executing the *Subscriber.exe* file.

CspBrowser

This directory contains two sample programs which illustrate basic JMS CspBrowser functionality using the FioranoMQ CPP Runtime Library.

1. Publisher.cpp: Reads strings from standard input and sends them on the queue "PrimaryTopic".
2. Browser.cpp: Implements a browser, which is used to browse the messages on the cspcache, and prints out the received messages.

To run these samples using FioranoMQ, perform the following steps:

1. Compile each of the source files. The %FMQ_DIR%\clients\cpp\native\scripts directory contains a script called cclientbuild.bat which compiles the CPP program.
2. Run the Sender by executing the Publisher.exe executable file. Send some messages before running the browser application

3. Run the asynchrounous receiver by executing the Browser.exe file.

Dursub

This directory contains two sample programs which illustrate basic JMS DurableSubscriber functionality using the FioranoMQ C++ Runtime Library.

- **DurPublisher.cpp -** Reads strings from standard input and publishes PERSISTENT messages on the topic "PrimaryTopic".
- **DurSubscriber.cpp -** Implements a durable subscriber using the client ID "DS_Client_1" and durable subscriber name "Sample_Durable_Subscriber", listening on the topic "PrimaryTopic".

To run these samples using FioranoMQ, perform the following steps:

1. Compile each of the source files. The %FMQ_DIR%\clients\cpp\native\scripts directory contains a script called cppclientbuild.bat which compiles the C++ program.
2. Start the DurableSubscriber program first, so that the subscriber can register with the FioranoMQ Server.
3. Next, start the Publisher_d program. When the program comes up, type in a few strings, pressing the Enter key after each string. The string is published and is received by the Durable Subscriber started in step (2) above.
4. Now, shut down the Durable Subscriber, but keep typing in messages into the Publisher program. These messages are automatically stored by the FioranoMQ Server, since a Durable Subscriber was previously registered on the topic to which the messages are being published.
5. After a while, restart the DurableSubscriber program. On restart, you would find that all messages that were published during the time that the durable subscriber was down are now made available to the subscriber.
6. Repeat steps (4) and (5) over. Each time, you would find that all messages published during the time that the Subscriber is down are immediately made available to the Subscriber when it restarts.

HierarchicalTopics

This directory contains two sample programs which illustrate basic JMS Publisher/Subscriber functionality using the FioranoMQ CPP Runtime Library.

1. Publisher.cpp: Reads strings from standard input and sends them on the hierarchical topics.
2. Subscriber.cpp: Receives messages from hierarchical topics.
3. createHierarchicalTopics.cpp: Creates required hierarchical topics that are used in Publisher and Subscriber.

To run these samples using FioranoMQ, perform the following steps:

Compile each of the source files.

1. The %FMQ_DIR%\clients\cpp\native\scripts directory contains a script called cppclientbuild.bat which compiles the CPP program.

2. Run createHierarchicalTopics that creates topics
3. Run the Publisher by executing the Publisher.exe executable file.
4. Run the asynchrounous subscriber by executing the Subscriber.exe file.

HTTP

This directory contains four sample programs which illustrate the use of HTTP protocol for basic JMS PubSub functionality using the FioranoMQ C++ Runtime Library.

- **Subscriber.cpp** - Receives messages synchronously published on "PrimaryTopic". This program implements an synchronous listener to listen for messages published on “PrimaryTopic".
- **Publisher.cpp** - Implements a client application publishing user specified data on "PrimaryTopic". This program reads strings from standard input and publishes them on “PrimaryTopic”.

To run this sample using FioranoMQ, perform the following steps:

1. Compile the source file. The %FMQ_DIR%\clients\cpp\native\scripts directory contains a script called cppclientbuild.bat which compiles the C++ program.
2. Run the HttpSubscriber by executing the Subscriber.exe executable file.
3. Run the HttpPublisher by executing the *Publisher.exe* executable file.

LMS

This directory contains sample programs which illustrate the use of Large Message Support - LMS for basic JMS ptp functionality using the FioranoMQ CPP Runtime Library.

1. LmPublisher.cpp: Accept filename to be send from standard input and send them on queue, "PrimaryTopic".
2. LmSubcriber.cpp: Implements an asynchronous listener, which listens on the queue "PrimaryTopic", and creates a received message file (default output fileName, "received.zip").

To run this sample using FioranoMQ, perform the following steps:

1. Compile the source file. The %FMQ_DIR%\clients\CPP\native\scripts directory contains a script called cclientbuild.bat which compiles the CPP program.
2. Run the Receiver by executing the LmSubscriber.exe executable file.
3. Run the Sender by executing the LmPublisher.exe executable file.

Message Compression

This directory contains two folders PerDestination and PerMessage.

PerDestination

This directory contains three sample programs which illustrate the message compress/uncompress abstraction supplied by the JMS API using a C++ Publisher application and Subscriber application.

1. TPublisher.cpp - Sends a file as a compressed message to the server on the "CompressedTopic".
2. TSubscriber.cpp - Implements an asynchronous listener, which listens on the topic “CompressedTopic". Receives the message sent by the producer in the uncompressed original form and writes the data in the message to a file.
3. CreateTopic.cpp - Creates a topic called "CompressedTopic" that has compression enabled on it which means that all messages sent on this topic will be compressed by default.

To run this sample using FioranoMQ, perform the following steps:

1. Compile the source file. The %FMQ_DIR%\clients\cpp\native\scripts directory contains a script called cppclientbuild.bat which compiles the C++ program.
2. Run the CreateTopic by executing the CreateTopic.exe executable file.
3. Run the TPublisher by executing the TPublisher.exe executable file.
4. Run the TSubscriber by executing the TSubscriber.exe executable file.

PerMessage

This directory contains two sample programs which illustrate the message compress/uncompress abstraction supplied by the JMS API using a C++ Publisher application and Subscriber application

1. TPublisher.cpp - Sends a file as a compressed message to the server on the "PrimaryTopic". Compression is enabled for every message and the default compression level and strategy are used.
2. TSubscriber.cpp - Implements an asynchronous listener, which listens on the topic "PrimaryTopic". Receives the message sent in the uncompressed original form and writes the data in the message to a file.

To run this sample using FioranoMQ, perform the following steps:

1. Compile the source file. The %FMQ_DIR%\clients\cpp\native\scripts directory contains a script called cppclientbuild.bat which compiles the C++ program.
2. Run the TPublisher by executing the TPublisher.exe executable file.
3. Run the TSubscriber by executing the TSubscriber.exe executable file.

Msgsel

This directory contains two sample programs which illustrate the use of message selectors using the FioranoMQ C++ Runtime Library.

- **SelSend.cpp -** Selector sends messages with the string property "name" and an int property "value", set differently for 3 consecutive messages.
- **SelRecv.cpp** - Implements a synchronous listener, which listens on the topic "PrimaryTopic" for the messages which match the criteria specified in the message selector, and prints out the received messages.

To run these samples using FioranoMQ, perform the following steps:

1. Compile each of the source files. The %FMQ_DIR%\clients\cpp\native\scripts directory contains a script called cppclientbuild.bat which compiles the C++ program.
2. Run the Sender by executing the SelSend.exe executable file.
3. Run the synchronous receiver by executing the *SelRecv.exe* file.

Mtpubsub

This directory contains one sample programs which illustrate basic JMS Publish/Subscribe functionality using the FioranoMQ C++ Runtime Library multithreading support.

- **mtPubSub.cpp** - The multithreaded version of basic PubSub. Single Publisher is created, publishes 10 text messages on ‘PrimaryTopic’and a single subscriber blocking receive with timewait of 1 second reads the messages. Each executes on a separate thread. On receipt of 10 messages, the subscriber notifies the main thread to end. Publisher and Subscriber threads are joined to the main thread.

To run these samples using FioranoMQ, perform the following steps:

1. Compile each of the source files. The %FMQ_DIR%\clients\cpp\native\scripts directory contains a script called cppclientbuild.bat which compiles the C++ program.
2. Run the mtPubSub by executing the *mtPubSub.exe* executable file.

NonJndi

This directory contains samples to create Topic connection Factory and Topic without using JNDI:

1. Publisher.cpp: Reads strings from standard input and publishes them on the topic "SampleTopic" created without using JNDI.
2. Subscriber.cpp: Implements an asynchronous listener, which listens on the topic "SampleTopic", and prints out the received messages.

To run these samples using FioranoMQ, perform the following step:

1. Compile each of the source files. The %FMQ_DIR%\clients\cpp\native\scripts directory contains a script called cppclientbuild.bat which compiles the C++ program.

2. Run the Publisher by executing the Publisher.exe executable file.
3. Run the asynchrounous subscriber by executing the Subscriber.exe file.

Reqrep

This directory contains two folders basic and timeout.

Basic

This directory contains two sample programs which illustrate JMS Request-Reply mechanism over Topics using the FioranoMQ C++ Runtime Library.

- **TopicRequestor.cpp -** Reads strings from standard input and sends the text messages on the topic "PrimaryTopic".
- **TopicReplier.cpp** - Implements an asynchronous listener, which listens on the topic "PrimaryTopic", and replies to the received. The reply is sent on TemporaryTopic.

To run these samples using FioranoMQ, perform the following:

1. Compile each of the source files. The %FMQ_DIR%\clients\cpp\native\scripts directory contains a script called cppclientbuild.bat which compiles the C++ program.
2. Run the Replier by executing the TopicReplier.exe executable file.
3. Run the requestor by executing the *TopicRequestor.exe* executable file.

TimedOut

This directory contains two sample programs which illustrate Timed Request-Reply mechanism over Topics using the FioranoMQ C++ Runtime Library.

- **TimedTopicRequestor.cpp -** Reads strings from standard input and sends the text messages on the topic "PrimaryTopic". The Requestor waits for a specified time for the reply. If the reply is not received within the stipulated time requestor times out.
- **TopicReplier.cpp** - Implements an asynchronous listener, which listens on the topic "PrimaryTopic", and replies on a TemporaryTopic.

To run these samples using FioranoMQ, perform the following:

1. Compile each of the source files. The %FMQ_DIR%\clients\cpp\native\scripts directory contains a script called cppclientbuild.bat which compiles the C++ program.
2. Run the replier by executing the TopicReplier.exe executable file.
3. Run the timed requestor by executing the *TimedTopicRequestor.exe* file.

RevalidateConnections

This directory contains two sample programs which illustrate connection revalidation functionality using the FioranoMQ CPP Runtime Library.

1. Publisher.cpp: Reads strings from standard input and use it to send a request message on the topic "PrimaryTopic". Also, revalidate the connection with the server.

2. Subscriber.cpp: Implements an asynchronous listener, which listens on the queue "PrimaryTopic", and prints out the received messages. Also, revalidate the connection with the server.

To run these samples using FioranoMQ, perform the following steps:

1. Compile each of the source files. The %FMQ_DIR%\clients\cpp\native\scripts directory contains a script called cppclientbuild.bat which compiles the C++ program.
2. Run the Publisher by executing the Publisher.exe executable file.
3. Run the Subscriber by executing the Subscriber.exe file.

ServerlessMode

This directory contains two sample programs which illustrate server less mode implementation of JMS Publisher/Subscriber functionality using the FioranoMQ C++ Runtime Library.

1. Publisher.cpp: Reads strings from standard input and sends them on the topic "PrimaryTopic".
2. Subscriber.cpp: Implements an asynchronous listener, which listens on the topic "PrimaryTopic", and prints out the received messages.

To run these samples using FioranoMQ, perform the following steps:

1. Compile each of the source files. The %FMQ_DIR%\clients\cpp\native\scripts directory contains a script called cppclientbuild.bat which compiles the C++ program.
2. Run the Publisher by executing the Publisher.exe executable file.
3. Run the asynchrounous subscriber by executing the Subscriber.exe file.

SSL

This directory contains two sample programs which illustrate basic JMS Publisher/Subscriber functionality using the FioranoMQ C++ Runtime Library.

- **Publisher.cpp** - Reads strings from standard input and sends them on the topic "PrimaryTopic".
- **Subscriber.cpp** - Implements a synchronous listener, which listens on the topic "PrimaryTopic", and prints out the received messages.

To run these samples using FioranoMQ, perform the following steps:

1. Compile each of the source files. The %FMQ_DIR%\clients\cpp\native\scripts directory contains a script called cppclientbuild.bat which compiles the C++ program.
2. Run the Publisher by executing the Publisher.exe executable file.
3. Run the asynchronous subscriber by executing the *Subscriber.exe* file.

Transaction

This directory contains a sample programs which illustrate JMS Transaction functionality using the FioranoMQ C++ Runtime Library.

- **Transaction.cpp** - Implements the sender and receiver, and uses the commit/rollback functionality to demonstrate JMS Transactions

To run these samples using FioranoMQ, perform the following steps:

1. Compile each of the source files. The %FMQ_DIR%\clients\cpp\native\scripts directory contains a script called cppclientbuild.bat which compiles the C++ program.
2. Run the sample by executing the *Transaction.exe* executable file. For proper results from the sample, ensure that there are no messages in the primaryTopic

Unified Samples

NonJndi

This directory contains samples to create Unified connection Factory, Topic, and Queue without using JNDI:

1. UnifiedProducer.cpp: Reads strings from standard input and sends them on to the queue "SampleQueue" and to the topic "SampleTopic".
2. UnifiedConsumer.cpp: Implements an asynchronous listener, which listens on the queue "SampleQueue" and on the topic "SampleTopic" and prints out the received messages.

To run these samples using FioranoMQ, perform the following steps:

1. Compile each of the source files. The %FMQ_DIR%\clients\c\native\scripts directory contains a script called cclientbuild.bat which compiles the C++ program.
2. Run the Sender by executing the UnifiedProducer.exe executable file.
3. Run the asynchronous receiver by executing the UnifiedConsumer.exe file.

SendReceive

This directory contains two sample programs which illustrate basic JMS Send/Receive functionality using the FioranoMQ C++ Runtime Library.

1. UnifiedProducer.cpp: Reads strings from standard input and sends them on to the queue "PrimaryQueue" and to the topic "PrimaryTopic".
2. UnifiedConsumer.cpp: Implements an asynchronous listener, which listens on the queue "PrimaryQueue" and on the topic "PrimaryTopic" and prints out the received messages.

To run these samples using FioranoMQ, perform the following steps:

1. Compile each of the source files. The %FMQ_DIR%\clients\c\native\scripts directory contains a script called cclientbuild.bat which compiles the C++ program.
2. Run the Sender by executing the UnifiedProducer.exe executable file.
3. Run the asynchrounous receiver by executing the UnifiedConsumer.exe file.