MQSeries Everyplace

XML Conversion Utility

Version 1.0

Take Note!

Before using this report be sure to read the general information under "Notices".

First Edition, August 2001

This edition applies to Version 1.0 of MQSeries Everyplace XML Conversion Utility and to all subsequent releases and modifications unless otherwise indicated in new editions.

Contents

Preface

MQSeries Everyplace is a messaging product that extends the reach of the MQSeries family to a wider range of distributed and mobile environments. Written in pure Java it runs on many platforms and uses a message queuing paradigm to provide secure, robust, and assured once-only in-order delivery of messages between applications.

MQSeries Everyplace seamlessly interfaces with other members of the MQSeries family of products, and is often used to extend the reach of corporate applications and data onto wireless and small footprint devices.

For more information on the MQSeries Everyplace base product, please see http://www-4.ibm.com/software/ts/mqseries/everyplace/, or try the free download of the full product available at http://software.boulder.ibm.com/dl/mqsem/mqsem-p.

MQSeries Everyplace transports messages between applications and queues. The messages are built using a data container, called an MQeFields object, in which the application data is placed.

This support pack allows you to transform any MQeFields object to an XML string and hence to use all the XML-enabled software products (including MQSeries Integrator).

MQSeries Everyplace can be used either on its own, or in conjunction with an MQSeries application. Presently, as MQSeries Everyplace messages are passed to MQSeries, the messages are transformed between formats using a 'bridge'. The default behavior of the bridge, for a basic MQSeries Everyplace message is to embed a binary form of the message into an MQSeries message body (special forms of the MQSeries Everyplace message and customizations of the bridge can provide more readily usable content).

If the binary representation of an MQSeries Everyplace message is placed in the MQSeries message, within MQSeries itself, you cannot easily access the various fields, and you cannot meaningfully process the content of an MQSeries Everyplace message using MQSI. More specifically, you cannot easily parse messages using MQSI, extracting pieces of the message for separate processing. Using the conversion tool provided by this SupportPac, the 'bridge' can be configured to produce MQSeries message with an XML data payload. This format is suitable for MQSI to fragment and process as if the XML content were originating from any other MQSeries XML application.

This SupportPac also enables you to represent MQeFields objects in a human readable language. This can be very useful for debugging purposes, for example, when an MQSeries Everyplace programmer wishes to view the contents of a message, or output the contents of a message to trace output, a file, or the screen.

This documentation explains all the conventions used to convert an MQeFields or MQeMsgObject object to an XML string and back again. The class that performs the transformation is also described.

Some examples of the use of the conversion tool are included:

- The use of the XML adapter as a disk adapter that allows you to configure a queue such that when a message is put on that queue, it is rendered to disk as an XML string, rather than the binary format normally used.

- An example “transformer” used by the bridging facility in a gateway MQSeries Everyplace queue manager. This code converts any MQeMsgObject sent over the MQSeries Everyplace network into an MQSeries message such that the body of that message contains an XML string representing the message content.

- A command-line tool that can render an existing unencrypted or compressed dumped MQeFields object, into an XML string representation for viewing. This application can be useful when observing your messages on the disk.

Summary of changes

Date	Changes
November 2001	Release 1.0

Chapter 1. Getting started

This chapter describes the software you need to use MQSeries Everyplace XML and how to install the SupportPac.

Prerequisites

This section describes the software you require to use this SupportPac:

- MQSeries Everyplace version 1.2.3 or above is required in order to use the SupportPac (we recommend you download the very latest version of MQSeries Everyplace from the MQSeries Everyplace download site)
- The examples within this SupportPac (.bat files) work only on Windows platforms, but the Java parts of the examples, and the XML conversion utility itself are platform-independent.
- A Java virtual machine (JVM) of level 1.2 or above is required to use the XML conversion utility, and a Java development toolkit (JDK) is needed to modify or extend the examples.

Installation

This XML utility is provided in a .zip file. Download the .zip file and unpack it to a directory of your choosing.

The XML utility contains the following files:

Filename	Content
\ea01.pdf	Documentation in adobe acrobat (PDF) format
\license2.txt	The license covering the use of his utility.
\com.ibm.mqe.xml.jar.	Java archive containing the XML conversion utility classes
\examples.xml.jar	Java archive containing example class files.
\examples\	Tree of directories containing example source code.

Edit your classpath to include the path to the XML utility classes on your machine.
On windows this can be done using the command-line syntax:

Set CLASSPATH=<xml_supportpac_directory>\com.ibm.mqe.xml.jar;
<xml_supportpac_directory>\examples.xml.jar;%CLASSPATH%

Where <*xml_supportpac_directory*> is the folder in which you installed the xml SupportPac.

The examples directory contains the Java source code for the example programs.

Chapter 2. The XML representation of MQeFields objects

Many objects in MQSeries Everyplace are subclasses of the MQeFields class.

An MQeFields object can be thought of as an unsorted tree of items, each node in the tree is a 'field' Fields can themselves be MQeFields objects. Hence, a nested multi-level container structure can be built using MQeFields objects.

The MQSeries Everyplace programming interface exposes the MQeFields class, but does not expose any sub-divisions of the MQeFields class in their own right. Although it is not a programming construct, throughout this document we refer to an *MQeField* or *field* to mean a logical node in the MQeFields tree of items.

XML is a popular format for representing tree structures due to its flexibility, the powerful technologies related to it (XSLT, DTD for example), and the increasing number of software products, such as MQSI, that use XML to represent data, and can also parse and manipulate XML content.

By using XML to represent the tree structure of an MQeFields object, MQSeries Everyplace users can exchange data with XML-enabled software products.

The node representation of an MQeFields object

There are three underlying constructs that are used to represent an MQeFields object when it is expressed as an XML string. These are:

- Basic elements
- Array elements
- Embedded MQeFields elements

Basic fields

Each item in an MQeFields object has three characteristics:

- *name:* A string of characters, in the allowable range as specified in the MQSeries Everyplace Programming Guide.

- *type:* The type of the information represented by this MQeFields item. Supported basic types are: ASCII, Boolean, byte, double, float, int, long, short, Unicode. Further types of array and fields are discussed later in this section.

- *value:* The value of a field.

Examples of the XML representation of some primitive-typed fields (unicode string, int, Boolean, long and ASCII) are shown below:

<my_unicode_field type="unicode">This is a Unicode string</my_unicode_field>

<my_integer_field type="int">1234</my_integer_field>

<my_boolean_field type="boolean">true</my_boolean_field>

<my_long_field type="long">12344</my_long_field>

<my_ascii_field type="ascii">This is an ASCII string</my_ascii_field>

Note that the name of the XML tag corresponds to the name of the field in the MQeFields object.

Array fields

The array construct is found in many computer languages. In MQSeries everyplace, an array can be put inside an MQeFields object in one of two ways, either as a 'static array' or as a 'dynamic array' of MQeFields elements.

Each method of array representation provides different capabilities to the MQSeries Everyplace application programmer, and each leads to different XML representations.

Static array representation

In a static array MQeFields element all the data in the array is put into a single MQeFields element, using the putArrayOf<type> style of MQSeries Everyplace call. For example, when using the MQeFields.putArrayOfInt() or MQeFields.putArrayOfLong() methods. A single XML tag is used to express the whole array, and the text between the begin-tag and end-tag bracketing contains a textual representation of all the array data. All elements of the array are marked as being the same type.

The XML representation of a static array conforms to the following syntax:

<its_name type="its_type">[array_length] {<value>;<value>; … <value> }</its_name>

Note that the values enclosed within the '{' '}' bracket are separated by ';' characters.

As an example, a *static array* containing three integer numbers has an XML representation of:

<my_integer_static_array type="int">[10]{9;8;7}</my_integer_static_array>

Dynamic array representation

A dynamic array in an MQeFields object is represented using several individual MQeFields elements, one for each element of the array, all on the same level of the Fields hierarchy.

The array is put into the MQeFields object using the *put<type>Array()* style of MQSeries Everyplace call. For example, when using the *MQeFields.putIntArray()* or *MQeFields.putLongArray()* methods.

When a dynamic array is expressed as an XML string, each element of the array is represent separately within it's own XML tag. All such elements are themselves enclosed by a bracketing XML tag to indicate that the fields are elements of a dynamic array.

The XML representation of a dynamic array follows the form:

<array_name type="dynamicArray">

*<array_name:**index** type="**element_type**" > **value** </array_name:**index**>*

*<array_name:**index+1** type="**element_type**" > **value** </array_name:**index+1**>*

…etc…

*<array_name:**index+n** type="**element_type**" > **value** </array_name:**index+n**>*

</array_name>

Where the array-name corresponds to the name of the dynamic array as it exists in the MQeFields object, the index is an integer, starting at 0, and incremented for each successive element in the dynamic array.

Note that the *type* attribute for each element of the array can be different, although normally all elements of the array would have the same type. For example, a *dynamic array* containing three integer numbers as the elements has an XML representation of:

<my_array type="dynamicArray">

<my_array:0 type="int">9</my_array:0>

<my_array:1 type="int">8</my_array:1>

<my_array:2 type="int">7</my_array:2>

</my_array>

Notes:

- There are no such methods as *putArrayOfAscii* and *putArrayOfUnicode*.
- The methods *putArrayOfByte(String, byte[][])* and *putByteArray(String, byte[])* do not take the same arguments
- You can only put a null array using the putArrayOf*** methods, if you try to use the put***Array methods with a null array, an exception is thrown.

Embedded fields

An MQeFields object can be placed, or embedded into another MQeFields object using the *MQeFields.putFields()* method. This has the effect of adding a nested level of fields within the existing MQeFields structure, forming a deeper hierarchy of field nodes. The MQeFields being embedded has a name similar to an integer field name. If an MQeFields object 'A' is being embedded within the MQeFields object 'B', then the XML representation of this follows the form:

<A type="fields">

… the XML representation of the items contained in A…

</A>

For example, if an MQeFields object 'A' contains a selection of basic fields, in XML this would look like the following:

```
<my_unicode_field type="unicode">This is a Unicode
string</my_unicode_field>

<my_integer_field type="int">1234</my_integer_field>

<my_boolean_field type="boolean">true</my_boolean_field>

<my_long_field type="long">12344</my_long_field>

<my_ascii_field type="ascii">This is an ASCII string</my_ascii_field>
```

Then suppose this MQeFields object is embedded inside another (called 'B'), using the *B.putFields("myEmbeddedMQeFields" , A);,* statement, the resultant XML representation of B would be:

```
<myEmbeddedMQeFields type="fields">

      <my_unicode_field type="unicode">This is a Unicode
String</my_unicode_field>

      <my_integer_field type="int">1234</my_integer_field>

      <my_boolean_field type="boolean">true</my_boolean_field>

      <my_long_field type="long">12344</my_long_field>

      <my_ascii_field type="ascii">This is an ASCII
string</my_ascii_field>

</myEmbeddedMQeFields>
```

The XML representation of other message fields

Three more pieces of information are required to represent an MQeFields object or any instance of an object that derives from the MQeFields class in XML:

- We must be able to set the document root of the XML document (the name of the tag, which is parent to all the other nodes).
- We must store the name of the class of the object we represent, in the XML document, so that it can be restored later if necessary.
- We must have a method of displaying the fields that are generated by MQSeries Everyplace and whose names are not explicit.

The document root

The top-level MQeFields object does not have a name, so we have to choose the name of the document root. By default, the name is "MQeFields" but you can change this value if you wish. The Java symbolic name for this is MQeXMLConverter.Document_Root. The following conventions are used:

- If the MQeFields object contains a Unicode field named 'XML_Document_Root', the value of this field is used as the name of the XML document root.
- If the MQeFields does not contain such a field, the default value is used.

Note: If you want to integrate MQSeries Everyplace into an existing XML application, you will probably not be given the choice of the name of the XML document root, as this will already have been decided by the application.

The class

Java is an object-oriented language, hence we must store the class of the object we are representing to XML. When we restore this object, we must create an instance of the same class to be able to apply the same methods to the restored object and to the original instance.

By convention, an attribute of the XML document root named "class" stores the class of the object represented. For example:

```
<MQeFields class="com.ibm.mqe.MQeMsgObject">
```

If the attribute is missing when we try to restore the object, a default class is used and, as with the document root, you can set the default class to be used if you wish.

The unreadable fields

To reduce the size of the objects it handles or sends through any network, MQSeries Everyplace uses very short names to store message specific information. For instance, if you try to represent an MQeMsgObject object without using any further convention, you will get an XML document like the one below:

<TEST_UNICODE TYPE="UNICODE">THIS IS A UNICODE STRING</TEST_UNICODE>
<TEST_INT TYPE="INT">123</TEST_INT>
<TEST_BYTE_ARRAY TYPE="BYTE">[10] {0;1;2;3;4;5;6;7;8;9}</TEST_BYTE_ARRAY>
<TEST_BOOLEAN TYPE="BOOLEAN">TRUE</TEST_BOOLEAN>
<°TA TYPE="LONG">989588710031</°TA>
<´ TYPE="LONG">989588709968</´>
<² TYPE="ASCII">MQEBRIDGE</²>

The highlighted fields are generated by MQSeries Everyplace and they store information such as the originating queue manager name or the time when the message was created. As you can see, in some cases, the name is not readable and the value of the field can be obscure as well, for instance, when it represents time with a long value.

To provide a more readable format for these fields, the names of the fields are changed while generating the XML representation, and an attribute named “comment” is added to the node whose value is the content of the field in a more readable format. The following table shows the mapping of the original fields to the XML generated fields.

MQeFields Name	Name displayed	Comment
MQe.Msg_OriginQMgr	OriginQMgr	
MQe.Msg_MsgID	MsgID	
MQe.Msg_CorrelID	MsgCorrelID	
MQe.Msg_LockID	MsgLockID	
MQe.Msg_Priority	Priority	
MQe.Msg_ReplyToQ	ReplyToQ	
MQe.Msg_ReplyToQMgr	ReplyToQMgr	
MQe.Msg_Resend	Resend	

MQeFields Name	Name displayed	Comment
MQe.Msg_Style	MsgStyle	Request, Reply or Datagram
MQe.Msg_Time	MsgTime	LongtoString()
MQe.Msg_WrapMsg	WrapMsg	
MQe.Msg_ExpireTime	ExpireTime	Long.toString() or Int.toString()
MQeIndexEntryConstants.INDEX_CONFIRMID	INDEX_CONFIRMID	
MQeIndexEntryConstants.INDEX_PREVIOUS_LOCKTYPE	INDEX_PREVIOUS_LOCKTYPE	various internal message states
MQeIndexEntryConstants.INDEX_TIME_ADDED	INDEX_TIME_ADDED	LongtoString()
MQeIndexEntryConstants.INDEX_LOCKTYPE	INDEX_LOCKTYPE	various internal message states

Using the previous conventions, a sample MQeMsgObject is represented as follows:

```
<MQeFields class="com.ibm.mqe.MQeMsgObject">
  <OriginQMgr type="ascii">MQeBridge</OriginQMgr>
  <MsgTime type="long" comment="11 MAY 2001
14:42:36:109">989588556109</MsgTime>
  <INDEX_TIME_ADDED type="long" comment="11 MAY 2001 14:42:36:265">
       989588556265
  </INDEX_TIME_ADDED>
  <Test_Unicode type="unicode">This is a Unicode String</Test_Unicode>
<Test_Int type="int">123</Test_Int>
  <Test_Byte_Array type="byte">[10] {0;1;2;3;4;5;6;7;8;9}</Test_Byte_Array>
  <Test_Boolean type="boolean">true</Test_Boolean>
</MQeFields>
```

You can represent any object which subclasses the MQeFields class in a human readable XML format, and restore an exact copy of this object from the XML document.

Restrictions

Field Names

MQSeries Everyplace allows a subset of the ASCII code page for field names but the XML parser used in the transformation is not as permissive. Hence, you cannot use the transformation on fields whose name contains any of the following:

‘<’, ’ ‘(space), ‘>’ or ‘/’.

If you try to convert an MQeFields object that contains such names, an exception is not thrown since the parser is not involved in the MQeFields to XML transformation but you will not be able to restore your object from the XML string.

Security

The adapter that uses the XML transformation cannot be used with field and message level security but it can be used with limited queue level security. Remote queue managers have to use security attributes to send messages to the secure queue manager. Most adapters apply security when the messages are moved to disk, but with the XML adapter the attributes are not used when the message is moved to disk, or other storage.

For example, configuring a queue with a queue security characteristic using the example NT authenticator, DES encryption and GZIP compression, and then configuring the XML disk adapter underneath the queue have contradictory aims. The encryption aims to make the information less accessible to readers, while the XML adapter aims to make the information more accessible to readers. In such cases, the data is NOT encrypted when stored on the disk, though the security attributes are used as normal when the messages are transmitted between queue managers.

A corollary is that using the XML adapter in the wrong place could breach your queue-based security model.

Chapter 3. com.ibm.mqe.xml.MQeXMLConverter

This section describes the class that provides the basic function of converting from an MQeFields object into an XML string, and vice-versa.

Constructor

public ***MQeXMLConverter(Errorhandler error_handler)*** :

Parameter:

Errorhandler error_handler:

This parameter allows you to handle the error messages generated by any method of this instance as you wish. A sample class named “com.ibm.mqe.xml.DefaultErrorhandler” is shipped as an example. It outputs the various error and warning messages using MQSeries Everyplace Trace.

Members

public String ***Default_Class = "com.ibm.mqe.MQeFields"*** :

You can set this member to change the class used to create the object from an XML string when no attribute named “class” is set in the document root (see the “class” section in the previous chapter for more information).

public String Document_Root = "MQeFields"

Stores the default document root *to* be used if the MQeFields object does not contains a Unicode field named XML_Document_Root (see the “document root” section in the previous chapter for more information).

Methods

The XML-MQeFields transformation

public MQeFields toMQeFields(String xml) throws Exception :

Generates an MQeFields object from the XML representation that is passed as a parameter.

Effectively the opposite to the *toString()* method.

Parameters:

String xml:

A string that is an XML representation of an MQeFields object or of any instance of a subclass of MQeFields. For example, this string could be the output of the *toString()* method on this class.

Returns:

The object that is represented by the XML string that was passed as parameter. This object is either an MQeFields or an instance of a subclass of the MQeFields class.

Exceptions:

An exception is thrown if the string is not a valid XML representation of an MQeFields object or of an instance of a subclass of the MQeFields class.

In order to make this method more tolerant, two conventions are used:

- If the document root does not contain an attribute named class, the *Default_Class* is used.
- If a tag does not have an attribute named type, it is considered to be an embedded field tag if it has any children. If it does not have any children, it is considered to be a Unicode field.

If any of the above conventions are used, a warning message is generated, but the process is not stopped.

With these conventions, any XML document that is structured as a MQeFields object can be understood.

Related Methods:

toMQeFields, *toString*, *toXMLByteArray*

public MQeFields toMQeFields(byte[] XML) throws Exception :

This method is quite similar to the previous one. It constructs a string using the byte array provided as parameter and calls the previous method to output an MQeFields object.

Parameters:

byte[] XML:

A byte array representation of an XML string.

This is converted internally to a string using the platform's default character encoding, as documented in the *java.lang.String(byte[])* constructor, before the string is converted into an MQeFields object.

Returns:

The object that is represented by the byte array that was passed as parameter. This object is either an MQeFields object, or an instance of a subclass of the MQeFields class.

Exceptions:

An exception is thrown if the string constructed from the byte array is not a valid XML representation of an MQeFields object or of an instance of a subclass of the MQeFields class.

Related methods:

toString, *toMQeFields*

The MQeFields-XML transformation

public String toString(MQeFields MQeMsg) throws Exception :

This method takes any MQeFields object as a parameter and outputs the XML representation of the MQeFields object as a string. This operation is the opposite of the *toMQeFields* method on this class.

Parameters:

MQeFields MQeMsg:

Any instance of an MQeFields class or of a subclass of the MQeFields class.

Returns:

A string that is an XML representation of the object that was passed as a parameter. All the conventions described previously are applied. The name of the class is stored in the “class” attribute of the document root. The fields generated by MQSeries Everyplace are transformed into a readable format and, if the object does not contains an ASCII field named “XML_Document_Root”, the Document_Root field of this class is chosen as the name of the document root.

Exceptions:

No exceptions are thrown as long as you pass an instance of MQeFields or of its subclass that does not overwrite the basic methods of the MQeFields class.

public byte[] toXMLbyteArray(MQeFields mqemsg) throws Exception :

This method is very similar to the previous one. It calls the above method and dumps the result as a byte array.

Parameters :

MQeFields mqemsg:

Any instance of MQeFields or of a subclass of the MQeFields class.

Returns:

A byte array that corresponds to the string containing the XML representation of the object that was passed as a parameter.

Exceptions:

As for the previous method, no exception is thrown as long as you pass an instance of MQeFields or of its subclass that does not overwrite the basic methods of the MQeFields class.

Related Methods:

toMQeFields

The MQSeries Everyplace specific fields

public String toParameters(MQeFields MQeMsg) throws Exception :

This method is used to make the unreadable MQSeries Everyplace specific fields understandable. For more information concerning the transformations performed, see the "unreadable field" section in the previous chapter.

Parameters :

MQeFields MQeMsg:

Any instance of MQeFields or of a subclass of the MQeFields class.

Returns :

A string that is an XML representation of the fields generated by MQSeries Everyplace inside the object that was passed as parameter. The other fields (generated by the user) are not transformed and do not appear in the returned string.

Exceptions :

An exception is thrown if a field reserved to MQSeries Everyplace, for instance the queue manager name field, was overwritten by the user with another field of the wrong type.

public String longtoString(long Time) :

This methods transforms a long, representing a date, to a string with the following format:

day month year(4 digits) hours:minutes:seconds:milliseconds(3 digits)

It is used when displaying the comment of field such as MQe.Msg_ExpireTime or MQe.Msg_Time

Parameters :

long Time :

A long value representing a date in milliseconds

Returns :

A string representation of this date, for example:

```
11 MAY 2001 14:42:36:109.
```

Chapter 4. Example applications

Disk adapter example

By default, MQeFields objects are stored on the disk in a binary format and their content is unreadable. This is because the com.ibm.mqe.adapters.MQeDiskFieldsAdapter is used to store MQeFields objects to disk.

If you choose to configure the XML transformation instead of using the default, then the MQeFields objects will be written to disk using a readable XML format.

The XML adapter can be useful when you are debugging an application since you can view the file on the disk to see if any field is missing or of the wrong type.

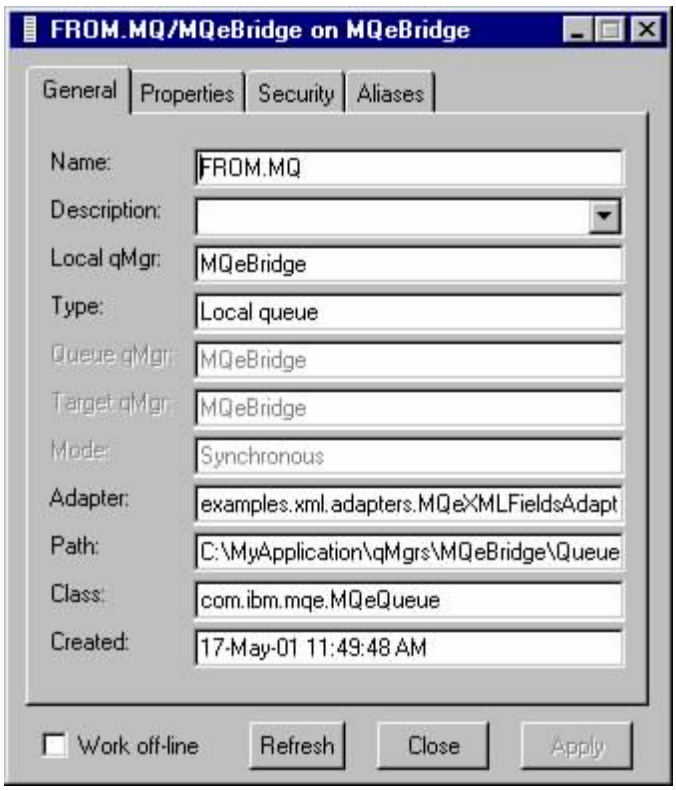
As an alternative to this, you could use the examples.xml.viewer.View example program described below.

Please note the security restrictions documented on 8.

You should not use this adapter when you deploy your application because it is much slower than the default.

Note: You cannot use the Notepad Editor to view the XML files as it does not recognize the new line character used in the transformation. You can use Wordpad, and many other editors.

If you want to store MQeFields objects using the XML transformation, set the adapter for any given queue to use the XML adapter (examples.xml.MQeXMLFieldsAdapter) instead of the default as it is shown in figure below.



From MQSeries Everyplace version 1.2.3 onwards, it is also possible to configure MQSeries Everyplace so that the entire configuration registry is written to disk using the same XML adapter.

This effect can be achieved by adding the following field to the ‘Registry’ section of the queue manager start-up parameters.

```
(ascii)Adapter=examples.xml.adapters.MQeXMLFieldsAdapter
```

For example, if your queue manager obtains its start-up configuration parameters from an .ini file in the same way the product examples do, then the following stanzas would provide a readable registry:

```
[Registry]
(ascii)LocalRegType=FileRegistry
(ascii)DirName=.\ExampleQM\Registry\
(ascii)Adapter=RegistryAdapter
```

The MQeDiskFields viewer

The default adapter used to store MQSeries Everyplace messages onto hard disk is the MQeDiskFieldsAdapter. This creates binary (dumped) rendering of an MQeFields object on the disk.

If the queue used to put the message does not use any security attributes (cryptor or compressor) then the MQSeries Everyplace messages can be converted to a readable format using this simple command-line tool.

Syntax

java examples.xml.viewer.View <filename> [-unwrapcompressor (lzw|null)]

Parameters

filename

The file and directory that holds the message.
This can be a relative or absolute path of the full name of the file.

-unwrapcompressor

An optional flag.
When used, it indicates what the viewer should do when it encounters an MQeMsgObject embedded inside the MQeFields structure, with the field name of MQe. Msg_WrapMsg.
If used, it should be followed by a string indicating which decompression algorithm should be used to de-compress the embedded MQeMsgObject.
If the parameter is not specific then the embedded MQeMsgObject is considered to be a byte array, and rendered to XML as such.

Example

If you view the directory structure of a queue manager, and navigate such that the current directory of a command-line prompt is the same directory in which several *.MQeMsg files are stored, then the following command in Windows would generate XML representations of all the messages in the directory…

```
C:\MQe\ClientQM\Queues\ClientQM\SYSTEM.DEFAULT.LOCAL.QUEUE>FOR %I IN (*.MQeMsg) DO
java examples.xml.viewer.View . %I

C:\MQe\ClientQM\Queues\ClientQM\SYSTEM.DEFAULT.LOCAL.QUEUE>java
examples.xml.viewer.View . 4000000E8DD9CD76B.MQeMsg
<MQeFields class="com.ibm.mqe.MQeMsgObject">
    <OriginQMgr type="ascii">ClientQM</OriginQMgr>
    <MsgTime type="long" comment="10 SEPTEMBER 2001
20:34:16:121">1000150456121</MsgTime>
    <INDEX_TIME_ADDED type="long" comment="10 SEPTEMBER 2001
20:34:16:151">1000150456151</INDEX_TIME_ADDED>
    <Test_Int type="int">123</Test_Int>
    <Test_Unicode type="unicode">This is a Unicode String</Test_Unicode>
    <Test_Boolean type="boolean">true</Test_Boolean>
    <Test_Byte_Array type="byte">[10] {0;1;2;3;4;5;6;7;8;9}</Test_Byte_Array>
</MQeFields>

C:\MQe\ClientQM\Queues\ClientQM\SYSTEM.DEFAULT.LOCAL.QUEUE>java
examples.xml.viewer.View . 4000000E8DD9D09A7.MQeMsg
<MQeFields class="com.ibm.mqe.MQeMsgObject">
    <OriginQMgr type="ascii">ClientQM</OriginQMgr>
    <MsgTime type="long" comment="10 SEPTEMBER 2001
20:34:29:30">1000150469030</MsgTime>
    <INDEX_TIME_ADDED type="long" comment="10 SEPTEMBER 2001
20:34:29:30">1000150469030</INDEX_TIME_ADDED>
    <Test_Int type="int">123</Test_Int>
    <Test_Unicode type="unicode">This is a Unicode String</Test_Unicode>
    <Test_Boolean type="boolean">true</Test_Boolean>
```

```
    <Test_Byte_Array type="byte">[10] {0;1;2;3;4;5;6;7;8;9}</Test_Byte_Array>
</MQeFields>
```

The same viewer can be used to view the MQSeries Everyplace registry files, as configured objects in MQSeries Everyplace also dump an MQeFields format of their configuration information to disk using the DiskFIeldsAdapter.

For example, here is the output when I view the registry entry for the SYSTEM.DEFAULT.LOCAL.QUEUE definition of the ClientQM queue manager on the hard disk:

```
C:\MQe\ClientQM\Registry\ClientQM\Queue>java examples.xml.viewer.View .
ClientQM+SYSTEM.DEFAULT.LOCAL.QUEUE.MQeReg
<MQeFields class="com.ibm.mqe.MQeFields">
    <QMS type="int">-1</QMS>
    <QID type="int">0</QID>
    <QR type="ascii"></QR>
    <QQMN type="ascii">ClientQM</QQMN>
    <QFD type="ascii">MsgLog:C:\MQeDev\ClientQM\Queues\</QFD>
    <QCD type="long">999182727459</QCD>
    <QP type="byte">[1] {4}</QP>
    <QQOS type="fields">
        <Puts type="long">0</Puts>
        <Expired type="long">0</Expired>
        <FailedGets type="long">0</FailedGets>
        <Deletes type="long">0</Deletes>
        <Gets type="long">0</Gets>
        <ConfirmPuts type="long">0</ConfirmPuts>
        <Browses type="long">0</Browses>
        <ConfirmGets type="long">0</ConfirmGets>
    </QQOS>
    <QN type="ascii">SYSTEM.DEFAULT.LOCAL.QUEUE</QN>
    <QM type="byte">0</QM>
    <QMSGSTORE type="ascii">com.ibm.mqe.messagestore.MQeMessageStore</QMSGSTORE>
    <QANL type="dynamicArray">
    </QANL>
    <MQe_Class type="ascii">7:</MQe_Class>
    <QMQS type="int">-1</QMQS>
    <QE type="long">0</QE>
    <QD type="unicode">Default Queue for ClientQM</QD>
</MQeFields>
```

Note: This tool provides a (sometimes useful) view into messages stored using the dumped MQeFields binary format, but the field names, and formats used should not necessarily be relied upon. Future releases of MQSeries Everyplace may well change the amount, type, labels and values of information within these registry files.

Tip:

This tool can be very useful when getting a snapshot of the entire queue manager registry and queue message stores when no encryption or compression is being used.
Under windows, you can use the command
FOR /R <my_queue_manager_directory> %i DO (*) java examples.xml.viewer.View %i

The XML bridge transformer

The aim of this bridge transformer is to provide the ability for any MQSeries Everyplace application to send XML-encoded string messages to MQSeries or MQSI applications, and receive their XML-encoded string reply messages.

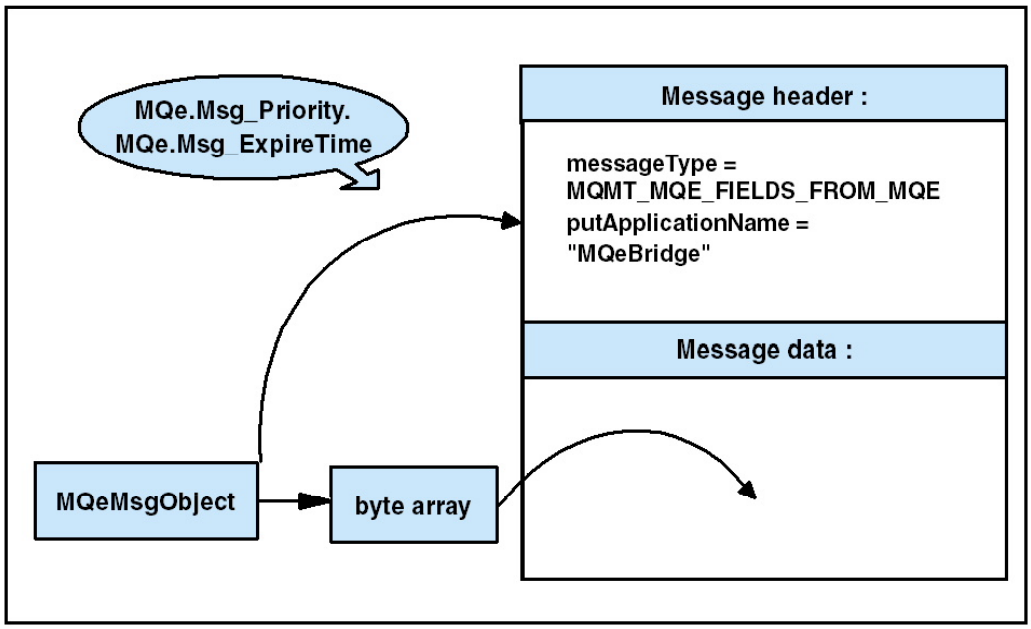
With this feature you can use all the functionality of MQSI relative to XML, for example you can:

- Use MQSI to parse the information within the XML-encoded string message, and only send the relevant part of the message to any other MQSeries queue or any MQSeries Everyplace queue. For example, strip out irrelevant message content.
- Insert some data into a DB2 database using the database nodes; enriching the message content with new data.
- Insert your MQSeries Everyplace application inside your existing MQSeries-MQSI workflow, so that services provided by MQSeries Everyplace applications can be used remotely from within an MQSI message flow application.
- Combinations of all the above.

The MQSeries Everyplace to MQSeries transformation

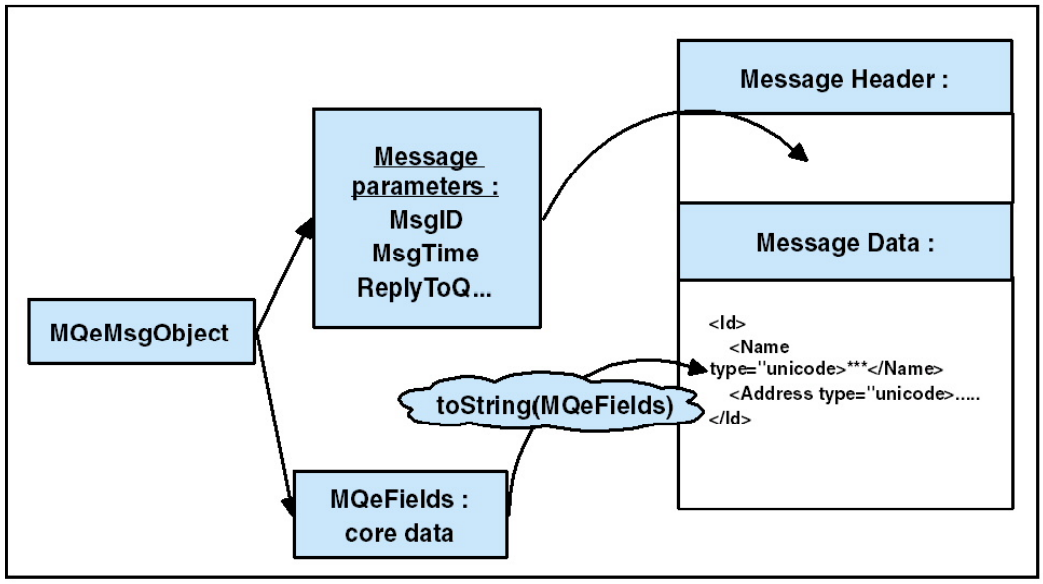
This section describes the default MQSeries Everyplace to MQSeries message transform, and the XML transform provided by this SupportPac.

The default MQSeries Everyplace to MQSeries transformation



As you can see, the MQeMsgObject is dumped to a binary format and is embedded into an MQSeries message as the message payload. The MQSeries message header is populated with information extracted from individual fields within the MQeMsgObject, such as the MQe.Msg_Priority and the MQe.Msg_ExpireTime fields. Using the default MQSeries-bridge transformer (com.ibm.mqe.mqbridge.MQeBaseTransformer) you can send an MQeMsgObject across the MQSeries network without any loss of information (the whole MQeMsgObject is stored in the message data), but you cannot process this message since MQSI can't handle this format.

The XML transformation



Instead of embedding the whole MQeMsgObject in a binary format inside a default MQSeries message, the XML bridge puts all the information relative to the message in the MQSeries header and only transforms the core date into XML. The XML data is put into the MQSeries message data.

The transformation separates the fields into two categories; those fields that contain core data from the fields relating to the behavior of the message. The priority and the message type are examples of the latter.

The fields relating to message behavior are used to populate the MQSeries message header, and the fields relating to the core data are supplied to the XML conversion class for translation into an XML string, which then forms the payload to the MQSeries message.

The mapping of each MQSeries Everyplace field to the MQSeries message header field is summarized in the following table:

MQSeries message header	Value
Format	MQC.MQFMT_STRING
putApplicationName	"MQe"
putApplicationType	MQC.MQAT_JAVA
priority	(int) MQe.Msg_Priority
messageType	MQe.Msg_Style_Datagram -> MQC.MQMT_DATAGRAM MQe.Msg_Style_Reply -> MQC.MQMT_REPLY MQe.Msg_Style_Request -> MQC.MQMT_REQUEST
replyToQueueName	MQe.Msg_ReplyToQ
replyToQueueManagerName	MQe.Msg_ReplyToQMgr
expiry	MQe.Msg_ExpireTime ***Note***: If the expiry time is of type 'long' then the message will expire "at" a certain time. If the expiry time is of type 'int' then the message will expire "after" that number of milliseconds have elapsed since the message was created.
putDateTime	MQe.Msg_Time
messageId	MQe.Msg_MsgID
correlationId	MQe.Msg_CorrelID

When millisecond values are converted into a readable minute-hour-day-month-year format, the default time zone in the default locale is used.

Be aware that for some parameters, further transformations are performed. For example, the putDateTime parameter in the MQSeries message header is a GregorianCalendar but the MQe.Msg_ExpireTime field in the MQeMsgObject is a long.

MQSeries to MQSeries Everyplace transformation

The same mechanism is used to transform an MQMessage to an MQeMsgObject object.

You should be aware that there might be a loss of information during the MQMessage to MQeMsgObject transformation because some features of MQSeries are not supported by MQSeries everyplace. Examples are message group, character set, and persistence. If the header of the MQMessage contains these parameters, they will not be available in the resulting MQeMsgObject object. The same is true for the REPORT message style, that is support by MQSeries Everyplace.

This should not affect your application since if you want an MQSeries message to cross your MQSeries Everyplace network; the best solution is to use the default bridge which does not alter the messages.

MQeMQMsgObject considerations

The com.ibm.mqe.mqbridge.MQeBaseTransformer is the default transformer for the MQSeries Everyplace - MQSeries bridge. When this transformer is sent a message of the MQeMQMsgObject class, the fields within this class are used to directly construct an MQSeries header and payload explicitly using the contents of the message. (See the MQSeries Everyplace Programming Guide for more details.)

Unlike the MQeBaseTransformer, the MQeXMLConverter class does NOT treat such MQeMQMsgObject messages as a special case. Such messages are treated in the same manner as any MQeMsgObject, or subclass of MQeMsgObject in that the MQeFields contained within the message are themselves converted to be part of the XML payload of the MQSeries message.

Consequently, configuring the XML transformer in place of the MQeBaseTransformer needs to be done with care, otherwise applications that use this feature of the MQeBaseTransformer will fail.

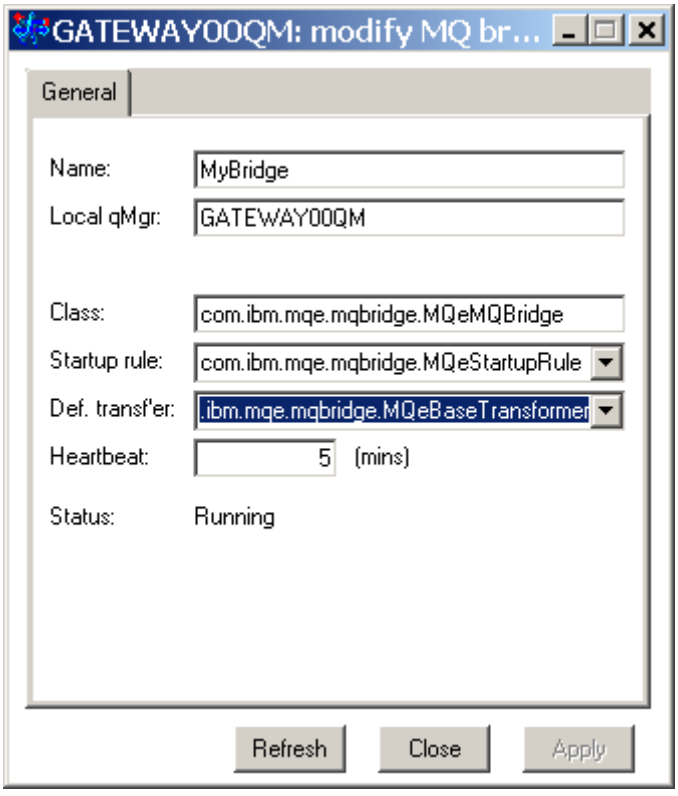
Configuring the transformer in your Gateway MQSeries Everyplace queue manager

The transformer is available in the examples.xml.bridge.MQeXMLTransformer.class.

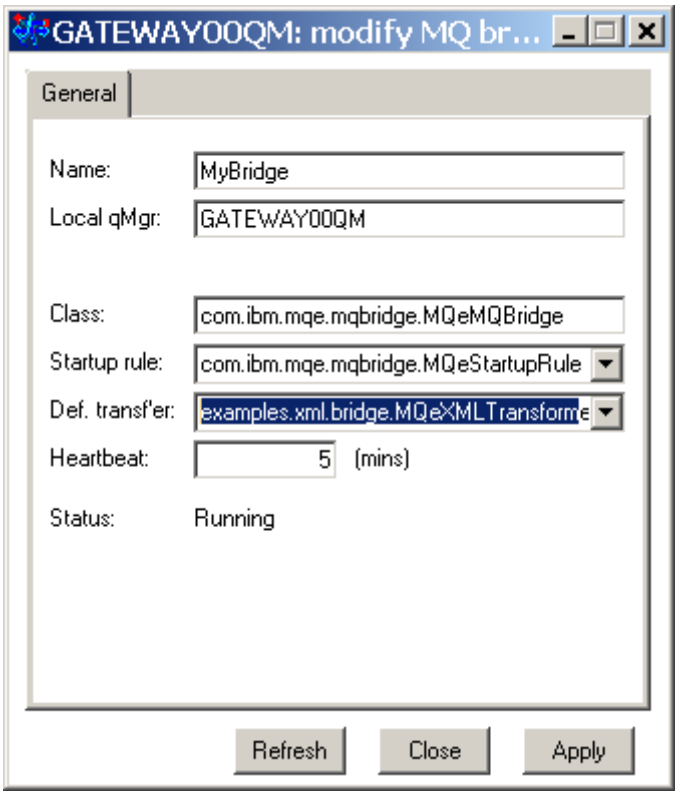
The source for this class can be found in the examples\xml\bridge directory.

To configure your MQSeries Everyplace gateway queue manager, follow your normal method of configuring your MQSeries Everyplace queue manager, but instead of specifying the default com.ibm.mqe.mqbridge.MQeBaseTransformer in the configuration of your bridge object, specify examples.xml.bridge.MQeXMLTransformer instead.

If using the MQe_Explorer.exe to configure your bridge, you might have a set of properties for the bridge object as:



If this is the case, simply change the “Default transformer field” in the panel to be:



For convenience, you will find the class named in the list by default.

If you specify the transformer explicitly on each bridge queue, or MQSeries Everyplace transmission queue listener bridge object, then you can use the same technique to substitute the name of the XML transformer instead of the default.

Bibliography

- *MQSeries Everyplace for Multi-platforms Programming Guide*, SC34-5845
- *MQSeries Everyplace for Multi-platforms Programming* Reference, SC34-5846
- *MQSeries Everyplace for Multi-platforms Introduction*, GC34-5843
- *Business Integration Solutions With MQSeries Integrator,* SC24-6154
- *MQSI Introduction and Planning, GC34-5599*
- *MQSeries for Windows NT Quick Beginnings, GC34-5389*
- *MQSeries System Administration, SC33-1873*

Notices

The following paragraph does not apply in any country where such provisions are inconsistent with local law.

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore this statement may not apply to you.

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates.

Any reference to an IBM licensed program or other IBM product in this publication is not intended to state or imply that only IBM's program or other product may be used. Any functionally equivalent program that does not infringe any of the intellectual property rights may be used instead of the IBM product.

Evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, is the user's responsibility.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, 500 Columbus Avenue, Thornwood, New York 10594, USA.

The information contained in this document has not be submitted to any formal IBM test and is distributed AS-IS. The use of the information or the implementation of any of these techniques is a customer responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment. While each item has been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. Customers attempting to adapt these techniques to their own environments do so at their own risk.

Trademarks and service marks

The following terms, used in this publication, are trademarks of the IBM Corporation in the United States or other countries or both:

IBM, MQSeries

Microsoft, Windows, and Windows NT, are trademarks of Microsoft Corporation in the United States and/or other countries.