MQSeries Integrator

Working with MQSeries Everyplace

Version 1.0

ID03-00

MQSeries Integrator

Working with MQSeries Everyplace

Version 1.0

ID03-00

Note!
Before using this information and the product it supports, be sure to read the general information under "Appendix D. Notices" on page 31.

First edition (June 2001)

This edition applies to Version 1.0 of SupportPac™ ID03 "Working with MQSeries Everyplace" for IBM® MQSeries Integrator Version 2 and to all subsequent releases and modifications until otherwise indicated in new editions.

A form for reader's comments is provided at the back of this publication. If the form has been removed, address your comments to:

User Technologies (MP095)
IBM United Kingdom Laboratories
Hursley Park
Hursley
Hampshire, SO21 2JN
England

When you send information to IBM, you grant IBM a non-exclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you. You may continue to use the information that you supply.

Contents

About this SupportPac

MQSeries® Integrator is designed to act as a message 'broker', using the underlying functionality of MQSeries messaging. In previous versions of MQSeries Integrator, all messaging was performed by 'base' MQSeries. From Version 2.0.2, MQSeries Everyplace messages can also be brokered. To enable this extra ability, new **MQeInput** and **MQeOutput** nodes are provided in MQSeries Integrator. Also, the **Publication** node incorporates the function of the **MQeOutput** node, thus allowing messages to be published to MQSeries Everyplace.

This SupportPac is provided to help you take advantage of the MQSeries Integrator connectivity with 'pervasive' devices provided by MQSeries Everyplace. It includes sample files to enable you to run a simple example which illustrates the communication between MQSeries Everyplace and MQSeries Integrator.

This document is in two distinct parts:

"Chapter 1. Working with MQSeries Everyplace" on page 1.
Configuring MQSeries Everyplace to MQSeries Integrator communication. This chaper provides an explanation, in general terms, of how MQSeries Everyplace and MQSeries Integrator work together.

"Chapter 2. MQSeries Everyplace scenarios" on page 9.
Using the sample code to configure MQSeries Everyplace to MQSeries Integrator communication. This chaper provides the information you need to run the examples provided in this SupportPac.

There is further reference material provided in a number of appendixes which explains how the supplied sample code operates.

Who this document is for

This document is for users of MQSeries Integrator who want to be able to use the connectivity with MQSeries Everyplace introduced in Version 2.0.2.

What you need to know to understand this document

You must be familiar with the concepts of MQSeries Integrator and with message flow design using the Control Center, but not necessarily implementing message flows with MQSeries Everyplace connectivity. You should refer to the MQSeries Integrator library if you need further information about the product. (See "Where to find more information" on page vi.)

Assumptions

If you download and use the files in this SupportPac, the following are assumed:

- You have read and you agree to the conditions documented in file `licence.txt` included in this SupportPac.
- You have installed MQSeries Integrator Version 2.0.2 or later (the scenarios will not work with earlier versions or releases). Your installation options must include the broker, the Configuration Manager, and the Control Center. Other components are optional for this purpose.

- You have created and started a broker on a supported runtime operating system (AIX, HP-UX, Sun Solaris, Windows® NT). The example here specifies queue manager `MQSI_SAMPLE_QM`. You can use a different queue manager if you choose. If you do so, you must modify the name used in the examples.
- You have created and started the Configuration Manager on a Windows NT® system. The name of the queue manager is not assumed. However, if your broker and your Configuration Manager do not share the same queue manager, you must set up MQSeries communications between the two queue managers.
- You must have access to a Java™ compiler (Version 1.1.8 or higher), not just a JRE, if you want to modify and re-compile the examples included in this SupportPac.
- You do not need to have installed MQSeries Everyplace. The parts of MQSeries Everyplace, including sample code, that are needed by MQSeries Integrator and this SupportPac are installed as part of the MQSeries Integrator broker installation.

Contents

This SupportPac is supplied in a zip file that contains all the files that you need to use the scenarios described here:

- This document (`id0300.pdf`).
- Licence agreement (`licence.txt`).
- Version description (`level.txt`).
- Files for the examples:
  - Configuration files (see "Appendix A. MQSeries Everyplace configuration files" on page 13 for an explanation of these two files):
    - `ClientQM.ini` - for creating an MQSeries Everyplace client queue manager.
    - `ServerQM.ini` - for creating an MQSeries Everyplace server queue manager within MQSeries Integrator.
  - Java source (`.java`) and compiled (`.class`) files to create connections between the queue managers and queues to route messages:
    - `SetupMQeExample1` - for creating a route between the client queue manager and the server queue manager, so that messages sent from the client can be routed to the server.
    - `SetupMQeExample2` - for creating a queue called Inbox on the server queue manager that may be used for returning messages from MQSeries Integrator.
  - Java source (`.java`) and compiled (`.class`) files for sending messages to the broker:
    - `SendMessages` - for point-to-point messages.
    - `SendPubSubMessages` - for publish/subscribe messages.

Where to find more information

For further details of how to program with MQSeries Everyplace, you should refer to the MQSeries Everyplace library.

The *"MQSeries Everyplace for Windows Version 1.1 Whitepaper"* provides a useful overview of MQSeries Everyplace.

For information specific to the use of MQSeries Everyplace nodes in MQSeries Integrator, you should refer to:

- Appendix C. *"MQSeries Everyplace Nodes"* in the *MQSeries Integrator Programming Guide* and
- Chapter 5. *"Working with message flows"* in the *MQSeries Integrator Using the Control Center* book.

The Whitepaper mentioned above and all MQSeries family books are available on-line at: `http://www.software.ibm.com/ts/MQSeries/library/`.

Chapter 1. Working with MQSeries Everyplace

MQSeries Integrator is designed to act as a message 'broker' in that it provides, in real-time, for messages to be routed and for the content of messages to be transformed and formatted, all based on rules which you can define as part of 'message flows'.

In order to transport messages which are manipulated in this way, MQSeries Integrator takes advantage of the connectivity provided by MQSeries messaging. MQSeries messaging is available in several flavors. MQSeries itself provides this messaging function on distributed and host platforms, while MQSeries Everyplace is designed primarily for messaging to, from and between pervasive devices, typically small, handheld devices, such as mobile phones and PDAs. SCADA, similarly, provides a messaging facility for pervasive devices, but it uses a very lightweight protocol tailored specifically for specialized applications on small footprint devices; typically in the area of remote data acquisition and process control.

Until version 2.0.2, all messaging in MQSeries Integrator was performed by MQSeries. From Version 2.0.2, input and output nodes are provided to allow messages to be sourced from, and dispatched to MQSeries Everyplace (and also SCADA).

MQSeries Everyplace applications work in rather different ways to 'normal' MQSeries Integrator applications, and so you will find that there are different concepts and procedures involved in setting up and configuring an MQSeries Integrator system to operate with MQSeries Everyplace. Before going on to consider how MQSeries Everyplace interacts with MQSeries Integrator, the following sections look at MQSeries Everyplace itself and how it relates to normal MQSeries messaging.

This brief digression is deliberately not intended to be a comprehensive introduction to MQSeries Everyplace. For that, you should refer to the MQSeries Everyplace library. The *"MQSeries Everyplace for Windows Version 1.1 Whitepaper"* also provides a useful overview of MQSeries Everyplace.

MQSeries Everyplace overview

With MQSeries, you will be familiar with the concept that a client provides assured messaging for local applications. The client can only access queues on an attached server, which it does via a synchronous client channel connection. The server, which can support the attachment of multiple clients, uses message channels to provide asynchronous delivery to remote queues.

MQSeries Everyplace uses what it calls 'devices' and 'gateways'. These are sometimes equated to MQSeries clients and servers, but in reality the analogy is not exact. The following list describes the terminology associated with the principal components of MQSeries Everyplace so that you can see the differences and similarities to MQSeries.

MQSeries Everyplace devices
: An MQSeries Everyplace device provides assured messaging for applications through dynamic channels (see below). It allows both

synchronous local and remote queue access and asynchronous delivery to remote queues. It therefore has the function typically associated with a server application, although it is restricted to handling only one incoming request at a time.

MQSeries Everyplace device code typically runs on a pervasive MQSeries Everyplace device and is started and stopped on demand by applications running intermittently. However, there is nothing to stop you installing a client on any appropriate machine and running it there.

This code is supplied as part of the MQSeries Integrator installation and it runs on the same machine as an MQSeries Integrator installation (and the location of the appropriate jar files are defined in the `CLASSPATH` environment variable - see "Chapter 2. MQSeries Everyplace scenarios" on page 9 for details), it is not necessary to install the MQSeries Everyplace device code separately.

MQSeries Everyplace gateways

MQSeries Everyplace gateways have the same functionality as clients, but also have a channel manager (which supports logical concurrent communication) configured so they can also handle multiple incoming requests at the same time. Gateways also support the attachment of MQSeries servers through MQSeries client channels.

Within MQSeries Integrator, the **MQeInput** node provides access to the MQSeries Everyplace gateway function, as described in "MQSeries Everyplace and MQSeries Integrator" on page 3.

MQSeries Everyplace channels

Devices and gateways use dynamic channels (so called to distinguish them from the MQSeries client and messaging channels) to communicate. Dynamic channels are a logical connection for sending and receiving data; they are bi-directional, and support both synchronous and asynchronous messaging.

Dynamic channels are established by an MQSeries Everyplace queue manager as required so, although you should be aware of their existence, they are not 'visible' to the user and you not need to do anything to enable their operation.

MQSeries Everyplace adapters

Because MQSeries Everyplace is regularly used on different pervasive devices, it is capable of using a variety of communication protocols. These are each implemented as an adapter so that additional protocols can easily be handled and only those actually required need be installed.

MQSeries Everyplace queue managers

MQSeries Everyplace queue managers are similar to their MQSeries counterparts in that they control various types of MQSeries Everyplace queues and channels. However, their architecture is object oriented and they run inside an instance of a JVM (each queue manager requires a separate JVM instance). Communications can be synchronous or asynchronous.

An MQSeries Everyplace queue manager can run:

- ***on an MQSeries Everyplace device*** — handling single incoming requests.
- ***on an MQSeries Everyplace gateway*** — handling many incoming requests simultaneously.
- ***as a servlet*** — with attributes similar to those of a queue manager running on an MQSeries Everyplace gateway. As you would expect, only

http adapters can be used. There is no channel listener (used by typical gateways to listen for incoming connection requests); this function is handled by the web server.

MQSeries Everyplace queues

MQSeries Everyplace queue managers control various types of queues. There are three which are particularly significant here:

- ***Local queues.*** This type of queue is local to, and is owned by, a specific queue manager.
- ***Remote queues.*** This type of queue does not reside locally. There is a local queue definition that identifies the real queue and the queue manager that owns it.
- ***MQSeries-bridge queues.*** These provide a path from MQSeries Everyplace to MQSeries. A bridge queue is a remote MQSeries queue definition on an MQSeries Everyplace gateway (see below).

MQSeries Everyplace messages

Unlike MQSeries messages (which are defined as byte arrays with a message header and a message body), MQSeries Everyplace messages are all passed as Java objects, derived from the base class `MQeFields`.

You should refer to "MQSeries Everyplace messages in MQSeries Integrator" on page 7 for details of the derived classes relevant to using MQSeries Everyplace with MQSeries Integrator.

Bridge to MQSeries

An MQSeries Everyplace gateway (but not an MQSeries Everyplace device) can act as an interface to an MQSeries server. It does this through an 'MQSeries-bridge queue' which uses the MQSeries Java client to interface to one or more MQSeries queue managers, thereby allowing messages to flow between MQSeries Everyplace and MQSeries.

The bridge queue is a remote queue definition on the gateway refering to a queue (the 'target queue') residing on an MQSeries queue manager. In other words, the queue holding the messages resides on the MQSeries queue manager, not on the local MQSeries Everyplace queue manager.

Before being transferred from the MQSeries Everyplace gateway to the MQSeries server, messages are passed through a 'transformer' which creates an MQSeries message from the object oriented MQSeries Everyplace message.

The details of how to configure a bridge between MQSeries Everyplace and MQSeries are not dealt with here. Although it is helpful to understand – at least in outline – how a bridge queue operates (as MQSeries Everyplace also uses bridge queues when communicating with MQSeries Integrator), configuration is different when communicating with MQSeries Integrator.

MQSeries Everyplace and MQSeries Integrator

Communication between MQSeries Everyplace and MQSeries Integrator is achieved through the MQSeries Integrator **MQeInput** and **MQeOutput** nodes. Using these nodes, you can write point-to-point applications where an MQSeries Everyplace input message is transmitted to an **MQeOutput** or an **MQOutput** node or publish/subscribe applications where the message is transmitted to a **Publication** node, as shown in Figure 1 on page 4.

MQSeries Everyplace and MQSeries Integrator

You should note that if you want to use MQSeries Everyplace with MQSeries Integrator, all message flows using an **MQeInput** node should be within the same MQSeries Integrator execution group as only a single execution group can be used in this context.

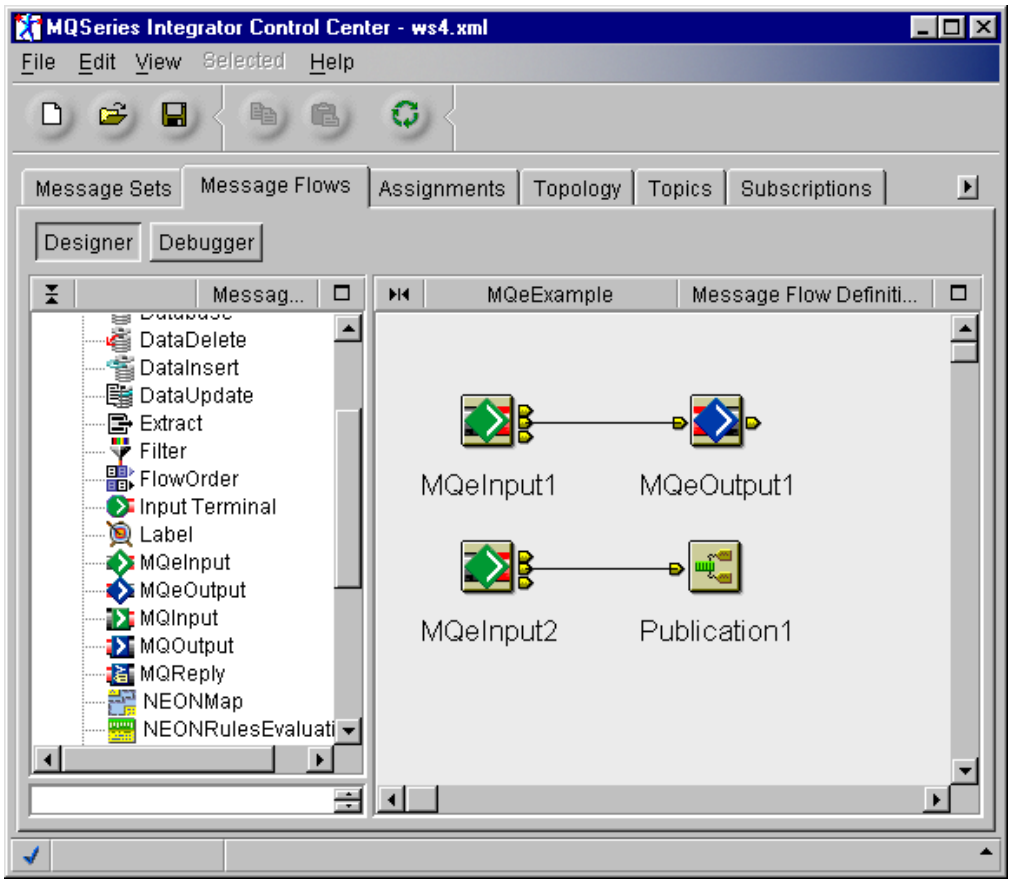

Figure 1. Examples of the use of MQSeries Everyplace nodes in MQSeries Integrator.

The general architecture involved is illustrated in Figure 2 on page 5. In order to pass messages between MQSeries Everyplace and MQSeries Integrator, an MQSeries Everyplace gateway — as part of an **MQeInput** node — is inserted into the MQSeries Integrator JVM (currently Version 1.1.8) running in a particular MQSeries Integrator broker. This can then communicate with other MQSeries Everyplace gateways, devices and servlets through a normal MQSeries Everyplace dynamic channel using appropriate adaptors.

Within this embedded gateway, an MQSeries-bridge queue is created. This, as just described, is a remote MQSeries Everyplace queue, in reality, a local queue definition of an MQSeries queue and the queue manager that owns it. In the case of an MQSeries-bridge queue used by MQSeries Integrator, the latter is the queue manager hosting the MQSeries Integrator broker.

Input from MQSeries Everyplace

In the example shown in Figure 2 on page 5, the MQSeries Everyplace 'client' attached to MQSeries Integrator is an MQSeries Everyplace device with an MQSeries Everyplace queue manager called `ClientQM1`. The broker, hosted by MQSeries queue manager `MQSI_SAMPLE_QM`, has a message flow deployed with an

MQeInput node. This has an embedded MQSeries Everyplace gateway (with its own MQSeries Everyplace queue manager, `ServerQM1` listening on an appropriate port) which treats `MQSI_SAMPLE_QM` as a remote MQSeries Everyplace queue manager. You should note that only one MQSeries Everyplace queue manager can be supported in a single instance of a JVM. So, if you have more than one **MQeInput** node in the same execution group, they must all use the same MQSeries Everyplace queue manager.

A message from the MQSeries Everyplace 'client' destined for MQSeries Integrator must be directed to the queue belonging to the MQSeries queue manager - `MQSI_SAMPLE_QM` - hosting the MQSeries Integrator broker (not the the MQSeries Everyplace queue manager - `ServerQM1` - running within MQSeries Integrator). Then, when the gateway receives a message destined for `MQSI_SAMPLE_QM` rather than itself, the message is put on the MQSeries-bridge queue.

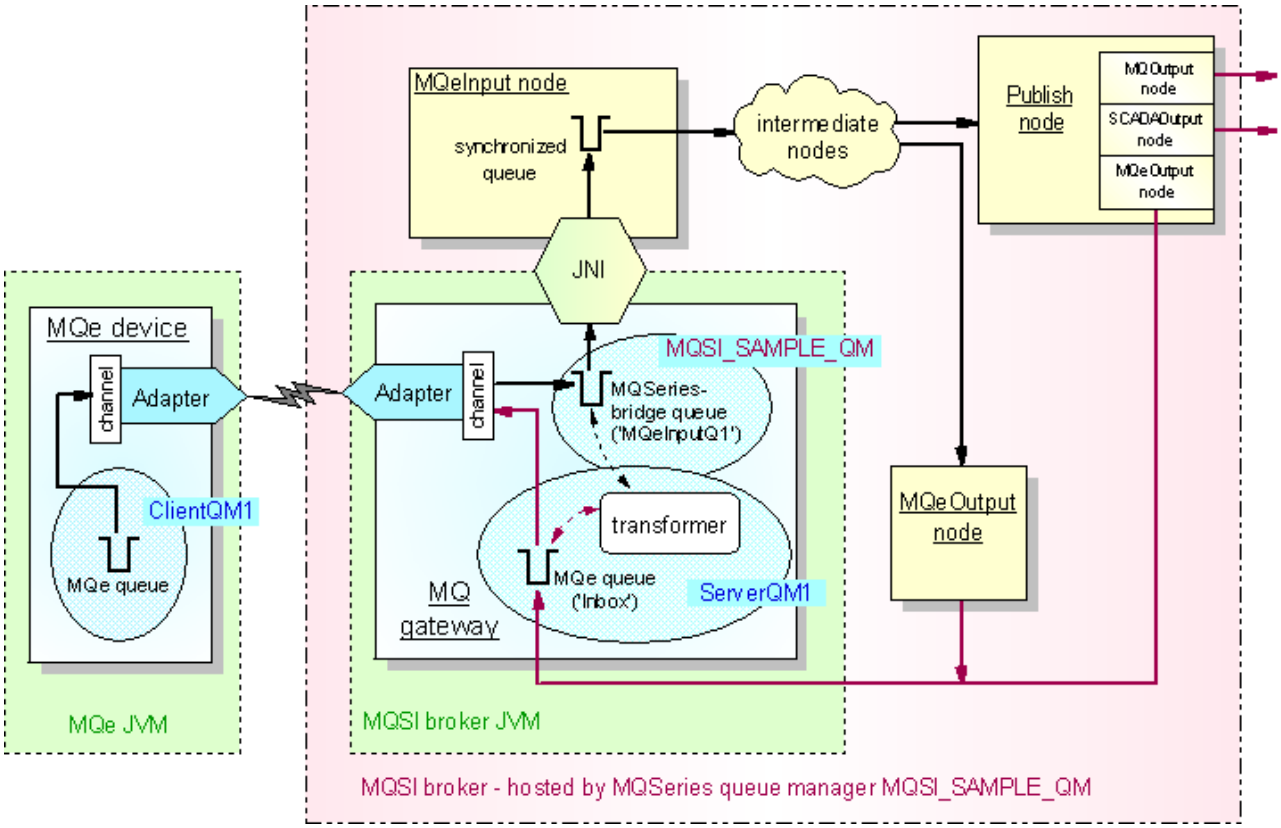

Figure 2. Diagram showing the relationship between MQSeries Everyplace and MQSeries Integrator

At this point, the message is an MQSeries Everyplace object, so it it passed to a transformer which creates an MQSeries form of the message and passes it back to the MQSeries- bridge queue. The message, now an MQSeries message, is passed over a JNI (Java Native Interface) connection and held on a synchronized MQSeries queue belonging to `MQSI_SAMPLE_QM`. From this queue, it is taken into the MQSeries Integrator message flow.

Within MQSeries Integrator, the message can be dealt with in different ways depending on the message class used for the message, as explained later ("MQSeries Everyplace messages in MQSeries Integrator" on page 7). You should

note that there is no parser in MQSeries Integrator directly capable of parsing messages derived from MQSeries Everyplace.

Also note that MQSeries Integrator only supports **MQeInput** or **MQeOutput** nodes in a single execution group. You should therefore ensure that all flows that communicate with MQSeries Everyplace are within the same execution group.

The MQeInput node

You should set the properties of an **MQeInput** node using the node's properties dialog by filling in at least the mandatory fields. You will find that you can accept the defaults provided for most other fields. The on-line help provides a useful reference to the node's properties.

Among the properties you must specify on the General tab are:

Queue Name
: You must specify the name of the MQSeries Everyplace bridge queue from which this input node retrieves messages. If the queue does not exist, it is created for you when the message flow is deployed to the broker.

Queue Manager Name
: This specifies the name of the MQSeries Everyplace queue manager. This is not related in any way to the queue manager of the broker to which you deploy the message flow containing this node. As only one MQSeries Everyplace queue manager can be supported, this property must be set to the same value in every **MQeInput** node.

If you select the *Use Config File* check box specified on the ″General″ tab of the **MQeInput** node's properties dialog, you can specify a file as the source of **MQeInput** node properties. An example of this file is shown in "Appendix A. MQSeries Everyplace configuration files" on page 13

If you select this check box, the node properties entered on the properties dialog are ignored except for the following:

- The *Queue Name* (that is, the name of the MQSeries-bridge queue) and *Config Filename* on this tab.
- All properties on the ″Default″ tab.

In the examples in this SupportPac, a configuration file is *not* used.

As explained in "The **MQeOutput** node" on page 7, you must configure an **MQeInput** node in your message flow, even if you intend only to send output to MQSeries Everyplace.

Output to MQSeries Everyplace

MQSeries Everyplace output can be sent either by using an **MQeOutput** node or by using a **Publication** node. The **Publication** node incorporates the function associated with an **MQeOutput** node, enabling messages to be retrieved by subscribers.

The **MQeOutput** node directs the message to a queue controlled by the MQSeries Everyplace queue manager hosting the MQSeries Everyplace gateway within MQSeries Integrator. This queue uses the transformer, mentioned above, to change the message from a MQSeries format to MQSeries Everyplace; typically creating an object from the `MQeMsgObject` class.

The MQeOutput node

You should set the properties of an **MQeOutput** node using the node's properties dialog. The on-line help provides a useful reference to the node's properties.

You must have an associated **MQeInput** node, even if you are only writing messages *to* MQSeries Everyplace and *not* receiving them from MQSeries Everyplace. There is information (such as the queue manager name and the listening port) that MQSeries Everyplace clients need to connect to MQSeries Integrator that is specified only in the **MQeInput** node.

The Publication node

You should set the properties of a **Publication** node using the node's properties dialog. The on-line help provides a useful reference to the node's properties.

In an MQSeries publish/subsribe application, you would typically send subscription messages to `SYSTEM.BROKER.CONTROL.QUEUE`. When using publish/subscribe with MQSeries Everyplace, you should not use this queue; all such messages are dealt with by the MQSeries-bridge queue.

MQSeries Everyplace messages in MQSeries Integrator

As noted above, MQSeries Everyplace messages are entirely object oriented and are all derived from the base Java class `MQeFields`. However, this class is not used directly, and when MQSeries Everyplace communicates with an MQSeries Integrator network, there are two classes that are used to create MQSeries Everyplace message objects:

- `MQeMsgObject`
- `MQeMbMsgObject`

Objects derived from both of these classes can be placed on the MQSeries-bridge queue named on the **MQeInput** node. Each message class produces different behaviors and each has advantages in different circumstances. The `SendMessage` sample provided in the SupportPac passes a message of each type into the **MQeInput** node and you can see from the sample how each message is constructed.

MQeMsgObject

`MQeMsgObject` does not put any restrictions on the fields it can contain, and so only predefined fields are transferred to the MQMD (the MQSeries message descriptor) when the message is passed to an MQSeries network — the remaining fields are put, 'unparsed', in the message body. The payload of a message derived from `MQeMsgObject` cannot be parsed, but this type of message does enable you to use special MQSeries Everyplace fields, such as `pic`. This allows the message to be reconstructed if it is sent back to MQSeries Everyplace by one of the nodes within the message flow (primarily the **MQeOutput** node). However, the payload part of the message is less parsable within MQSeries Integrator because a parser is currently not supported to read `MQeMsgObject`.

MQeMbMsgObject

A message constructed from `MQeMbMsgObject` has only those fields that are compatible with the broker passed into the message flow; unrecognized fields are ignored. Therefore, if this message is routed back to an MQSeries Everyplace queue, these fields will not be present. Although it enables you to parse the payload, and therefore manipulate or operate on parts of that data (for example, store it in a database), you cannot use certain special MQSeries Everyplace fields, such as `pic`.

Security

If MQSeries Everyplace security is configured then the MQSeries Everyplace authentication (described in the MQSeries Everyplace manuals) is used until the message reaches the special bridge queue.

Problem determination

When using MQSeries Everyplace with MQSeries Integrator, you should use all the normal facilities available to you with MQSeries Integrator, that is:

- trace.
- the application event log.
- the MQSeries Integrator Control Center log.

The use of these is covered in the MQSeries Integrator documentation.

In addition to the MQSeries Integrator trace, you can also choose to enable MQSeries Everyplace trace. To do this, in the **MQeInput** node, on the ″General″ tab of the properties dialog, change the default setting of the ″Trace″ property from none to one of:

- standard
- debug
- full

You should then set the ″Trace Filename″ property to show the path and file where trace is to be written. The directory structure in which the file is specified must already exist, but you do not need to have created the file in advance.

Information about MQSeries Everyplace processing within a message flow is sent to the MQSeries Integrator trace in the event of an error (for example, failure to create a queue) but, if MQSeries Everyplace trace is active, more information is contained in the MQSeries Everyplace-specific trace file.

Chapter 2. MQSeries Everyplace scenarios

This example provides samples that give you a quick start at connecting MQSeries Everyplace into an MQSeries Integrator broker. They enable you to:

- Create two MQSeries Everyplace queue managers: a server called `ServerQM1` and a client called `ClientQM1`.
- Create connections between them and create the queues necessary for routing messages.
- Transmit sample point-to-point or publish/subscribe messages through the message broker.

You can modify any of the samples provided. If you do so, you must recompile the Java source code.

The following steps explain how to run the example. The directory structures described assume that you are using Windows NT. On other platforms, you should change these paths accordingly.

Step 1. **Set up the `CLASSPATH`** for every machine from which you will run programs that communicate with MQSeries Everyplace within MQSeries Integrator and with MQSeries Everyplace itself (standalone).

You need to add:

```
<mqsi_root>\classes\mqimqe.jar;
<mqsi_root>\classes\mqedevice.jar;
<mqsi_root>\classes\mqeexamples.jar;
<mqsi_root>\classes\mqegateway.jar;
<mqsi_root>\classes\mqemqbridge.jar;
```

On Windows NT, there is a limit of 255 characters in a `CLASSPATH` setting. To avoid encountering this restriction, you could assign these to a new environment variable, for example `MQEPATH`, then add `%MQEPATH%;` to your `CLASSPATH`.

Step 2. **Create a directory structure** somewhere convenient on your machine for the sample Java package: `/com/ibm/broker/mqimqe/example`.

Step 3. **Store the files** supplied with this SupportPac into that directory.

Step 4. **Create MQSeries Everyplace queue managers on the MQSeries Everyplace client.**

From a command prompt, noting that the file name is case sensitive, type:

```
java examples.install.CreateQueueManager
```

In the Configuration File box, browse for and select the `ClientQM1.ini` file, supplied in this SupportPac. Set the directory for the queues to `x:\ClientQM1` (where `x` is a drive letter of your choice). Click 'OK'.

An MQSeries Everyplace client queue manager is created with a name `ClientQM1`, with a registry located at `x:\ClientQM1\Registry`.

Step 5. **Create the MQSeries Everyplace server queue manager within MQSeries Integrator.**

From a command prompt, noting that the file name is case sensitive, type:

```
java examples.install.CreateQueueManager
```

In the Configuration File box, browse for and select the `ServerQM.ini` file, supplied in this SupportPac. Set the directory for the queues to `x:\ServerQM1`. Click 'OK'.

An MQSeries Everyplace server queue manager is created with a name `ServerQM1`, with a registry located at `x:\ServerQM1\Registry`.

If you browse the sample `ServerQM.ini` file, you might notice that the listener is set to listen on port 8081. This setting is ignored on the creation of the queue manager. Later in the configuration, if you choose to have the **MQeInput** node take its settings from a configuration file, the port number will be picked up and used.

If you get an error on creating the server queue manager with text:

```
com.ibm.broker.mqimqe.examples.rule.AttributeRule
```

check that you have correctly referenced `<mqsi_root>\classes\mqimqe.jar` in the `CLASSPATH`.

Step 6. **Configure message flows.**

You can either add MQSeries Everyplace nodes to existing message flows or create new flows. Open the Control Center Message Flows (Designer) view. Drag and drop the nodes into the message flow. For point-to-point (non publish/subscribe) messaging, you need an **MQeInput** node and an **MQeOutput** node. For publish/subscribe messaging, you need an **MQeInput** node and a **Publication** node.

Configuring the MQeInput node

For this example, you need to set only the "Queue Name" (on the "General" tab). Each **MQeInput** node needs to have a different queue name. In Step 9 below, the example uses the queue name, `MQeInputQ1`.

You may also want to make other configuration changes (for example, setting the level of trace) but this is not necessary to be able to run these scripts.

Configuring the MQeOutput node

Check that the "Destination Mode" (on the "Advanced" tab) is "Destination List". Leave all other fields blank.

Configuring the Publication node

No action required.

Step 7. **Deploy the message flow.**

Check in the message flow, assign it to a broker, and deploy complete assignments data. Check that the deploy is successful in the Log view of the Control Center. (You should see BIP404I and BIP2056I).

- The client queue manager is started when the client starts.
- The server queue manager is only started when an **MQeInput** node is running and is deployed in a flow where a MQSeries Integrator broker is running.
- Deploying a point-to-point message flow (**MQeInput** —> **MQeOutput**) also starts the MQSeries Everyplace server queue manager.

Step 8. **Set up the MQSeries Everyplace queue managers.**

Run the `SetupMQeExample1` and `SetupMQeExample2` programs to set up the MQSeries Everyplace queue managers.

- SetupMQeExample1 creates a route so that the client queue manager can contact the server queue manager.
- SetupMQeExample2 creates a queue on the server queue manager called Inbox that can be used in the examples for returning messages.

These programs take the following parameters. SetupMQeExample1 requires all five parameters, in the sequence shown. SetupMQeExample2 requires the first two only.

a. MQSeries Everyplace queue manager name (ServerQM1 in this example).
b. Path to the MQSeries Everyplace client configuration file (ClientQM.ini in this example).
c. IP address of the MQSeries Everyplace server(1.23.45.678 in this example).
d. Port on which the MQSeries Everyplace server is listening. The port number must match the "Port" specified on the "Listener" tab of the **MQeInput** node properties dialog. By default, the port number in the **MQeInput** node properties is 8081.
e. Name of the MQSeries queue manager hosting the MQSeries Integrator broker (for example MQSI_SAMPLE_QM).

From a command prompt, type on one line (for example):

```
java com.ibm.broker.mqimqe.example.SetupMQeExample1  ServerQM1
       x:\com\ibm\broker\mqimqe\example\ClientQM.ini
       1.23.45.678  8081  MQSI_SAMPLE_QM
```

Step 9. **Send messages through the message flows.**

- For point-to-point messages, use SendMessages. This sends a message using both the MQeMsgObj class and the MQeMbMsgObj class.
- For publish/subscribe messages, use SendPubSubMessages. This subscribes, publishes, reads the message, and then unsubscribes.

Both these programs take the following parameters, all of which must be present:

a. MQSeries Everyplace queue manager name (ServerQM1 in this example).
b. Path to the MQSeries Everyplace client configuration file (ClientQM.ini).
c. Name of the MQSeries queue manager hosting the MQSeries Integrator broker (for example MQSI_SAMPLE_QM).
d. Name of the MQSeries-bridge queue receiving the input (the queue you named on the **MQInput** node "Queue Name" property; for example MQeInputQ1).
e. Name of the MQSeries Everyplace queue where you want to receive messages back from the broker. The sample code creates a queue called Inbox but, if you already have an MQSeries Everyplace queue defined, you can specify that queue.

To send point-to-point messages, type - on one line - at a command prompt (for example):

```
java com.ibm.broker.mqimqe.example.SendMessages  ServerQM1
       x:\com\ibm\broker\mqimqe\example\ClientQM.ini
       MQSI_SAMPLE_QM  MQeInputQ1  MQeOut
```

Or, to send publish/subscribe messages, type - on one line - at a command prompt (for example):

```
java com.ibm.broker.mqimqe.example.SendPubSubMessages  ServerQM1
       x:\com\ibm\broker\mqimqe\example\ClientQM.ini
       MQSI_SAMPLE_QM  MQeInputQ1  MQeOut
```

In this latter case, if messages are sent and received successfully, you should see output similar to:

```
..Started queue manager: ClientQM1
Subscribing to the topics:
climate
humidity
temperature
..Put message to QM/queue: MQSI_SAMPLE_QM/MQeInputQ2
Publishing message to topic 'climate'
..Put message to QM/queue: MQSI_SAMPLE_QM/MQeInputQ2
Reading message from queue
Topic:  climate
Message: sunny
un-subscribing from the topics:
climate
humidity
temperature
..Put message to QM/queue: MQSI_SAMPLE_QM/MQeInputQ2
```

(where MQeInputQ2 is the name of the **MQeInput** node used).

If, after the sample attempts to put a message to the broker queue manager and the MQSeries-bridge queue, you see the following MQSeries Everyplace exception:

```
java.net.ConnectException: Connection refused
```

you need to check the following:

a. The broker is started (mqsistart <brokername>).
b. The port number that you specified when you ran SetupMQeExample1 matches the port number specified on the ″Listener″ tab of the **MQeInput** node.
c. There is a listener running on that port number (type netstat at a command prompt to see).
d. You have specified an MQSeries-bridge queue name. (The deploy will apparently be succesful even if you have not set this parameter.)

Appendix A. MQSeries Everyplace configuration files

Following, are two listings of example MQSeries Everyplace configuration files:

- **ServerQM.ini** is for an MQSeries Everyplace server running inside the message broker.
- **ClientQM.ini** is for a simple MQSeries Everyplace client.

ServerQM.ini

```
*
* MQeConfig.ini
*   An example ini file for an MQe server running inside the message broker.
*
[Alias]
*
*    Event log class
*
(ascii)EventLog=examples.log.LogToDiskFile
*
*    Network adapter class
*
* Network=com.ibm.mqe.adapters.MQeTcpipHttpAdapter
*
*    Queue Manager class
*
(ascii)QueueManager=com.ibm.mqe.MQeQueueManager
*
*    Trace handler (if any)
*
(ascii)Trace=com.ibm.broker.mqimqe.wrapper.trace.MQeTrace
*
*    Message Log file interface
*
(ascii)MsgLog=com.ibm.mqe.adapters.MQeDiskFieldsAdapter
*
*    Class name for File registry
*
(ascii)FileRegistry=com.ibm.mqe.registry.MQeFileSession
*
*    Class name for Private registry
*
(ascii)PrivateRegistry=com.ibm.mqe.registry.MQePrivateSession
*
*    Default Channel class
*
(ascii)DefaultChannel=com.ibm.mqe.MQeChannel
*
*    Default Transporter class
*
(ascii)DefaultTransporter=com.ibm.mqe.MQeTransporter
*
*    Channel Attribute Rules
*
(ascii)ChannelAttrRules=com.ibm.broker.mqimqe.examples.rules.AttributeRule
*
*    Name of Base Key
*
(ascii)AttributeKey_1=com.ibm.mqe.MQeKey
*
*    Name of Shared Key
*
```

```
(ascii)AttributeKey_2=com.ibm.mqe.attributes.MQeSharedKey
*
(ascii)NetworkHttp=com.ibm.mqe.adapters.MQeTcpipHttpAdapter
*-------------------------------------------------------------------------*
* Activate listeners
[Listeners]
(ascii)01=ListenerHttp
* TcpipHttp (Web server) listener definition
[ListenerHttp]
(ascii)Accept=NetworkHttp:
(ascii)Adapter=AdapterHttp
(ascii)Class=com.ibm.mqe.MQeChannelListener
(ascii)Manager=com.ibm.mqe.MQeChannelManager
(ascii)MaxChannels=0
(ascii)TimeInterval=300
* Configuration of the HTTP adapter
[MapAdapter]
(ascii)AdapterHttp=FileDescriptor=NetworkHttp::8081;Parameter=;Option=<LISTEN>
*-------------------------------------------------------------------------*
[QueueManager]
*
*   Name for this Queue Manager
*
(ascii)Name=ServerQM1
*-------------------------------------------------------------------------*
[Registry]
*
*   Type of registry for config data
*
(ascii)LocalRegType=FileRegistry
*
*   Location of the registry
*
(ascii)DirName=\ServerQM1\Registry\
```

The above sample configuration file contains a special `[Listeners]` section that is not present in a 'normal' MQSeries Everyplace configuration file. Any configuration file used on an **MQeInput** node must be based on this sample and not an existing file. Without this new section,

- the listener will not operate if you have ticked the box in the **MQeInput** node properties to use a configuration file, and so you will not be able to receive any incoming connections.
- you cannot use MQe_Explorer (available as SupportPac ES02: "MQSeries Everyplace MQe_Explorer") once an **MQeInput** node has used it.

Furthermore, the `[MapAdapter]` section, normally optional, is mandatory when writing a configuration file for the MQSeries Everyplace server within the MQSeries Integrator message broker. This is because the stand-alone version of MQSeries Everyplace typically only supports one port number to listen on. But within the message broker, it is possible to have a different port number for each **MQInput** node, and so multiple listeners may be needed in a single MQSeries Everyplace instance. The example demonstrates how these sections are configured:

- The `[Listeners]` section contains a list of definitions for listeners. The example only contains one definition, called `ListenerHttp`.
- Each definition must have an associated Section, in this case called `[ListenerHttp]`. This contains specific information about the listener session. It also includes the `Adapter` field, which correlates to a field within the `[MapAdapter]` section.
- The `[MapAdapter]` section contains specific information concerning the port such as the port number to use and the adapter type.

ClientQM.ini

```
*
* ExamplesMQeClient.ini
*   An example ini file for a simple MQe client
*
[Alias]
*
*    Event log class
*
(ascii)EventLog=examples.log.LogToDiskFile
*
*    Network adapter class
*
(ascii)Network=com.ibm.mqe.adapters.MQeTcpipHttpAdapter
*
*    Queue Manager class
*
(ascii)QueueManager=com.ibm.mqe.MQeQueueManager
*
*    Trace handler (if any)
*
(ascii)Trace=examples.trace.MQeTrace
*
*    Message Log file interface
*
(ascii)MsgLog=com.ibm.mqe.adapters.MQeDiskFieldsAdapter
*
*    Class name for File registry
*
(ascii)FileRegistry=com.ibm.mqe.registry.MQeFileSession
*
*    Class name for Private registry
*
(ascii)PrivateRegistry=com.ibm.mqe.Registry.MQePrivateSession
*
*    Default Channel class
*
(ascii)DefaultChannel=com.ibm.mqe.MQeChannel
*
*    Default Transporter class
*
(ascii)DefaultTransporter=com.ibm.mqe.MQeTransporter
*
*    Channel Attribute Rules
*
(ascii)ChannelAttrRules=examples.rules.AttributeRule
*
*    Name of Base Key
*
(ascii)AttributeKey_1=com.ibm.mqe.MQeKey
*
*    Name of Shared Key
*
(ascii)AttributeKey_2=com.ibm.mqe.Attributes.MQeSharedKey
*-------------------------------------------------------------------------*
*
*  Registry ( configuration data store )
*
[Registry]
*
*   Type of registry for config data
*
(ascii)LocalRegType=FileRegistry
*
*   Location of the registry
*
```

```
(ascii)DirName=\ClientQM1\Registry\
*------------------------------------------------------------------------*
*
*  Queue manager details
*
[QueueManager]
*
*   Name for this Queue Manager
*
(ascii)Name=ClientQM1
```

Appendix B. MQSeries Everyplace queue manager setup files

Following, are two listings of the files to create connections between the queue managers and queues to route messages:

- **SetupMQeExample1.java** creates a route so that the client queue manager knows how to contact the server queue manager.
- **SetupMQeExample2.java** creates a queue on the server queue manager called Inbox that can be used in the examples (send (PubSub) messages) for returning messages.

SetupMQeExample1.java

```
package com.ibm.broker.mqimqe.example;

/*--------------------------------------------------------------------------------*/
/* Licensed Materials - Property of IBM                                           */
/*                                                                                */
/*                              MQSeries Everyplace                               */
/*                              ===================                               */
/*                                                                                */
/*                                                                                */
/* Copyright IBM Corp 2001.  All rights reserved.                                 */
/*                                                                                */
/* US Government Users Restricted Rights - Use, Duplication or disclosure        */
/* restricted by GSA ADP Schedule Contract with IBM Corp.                         */
/*--------------------------------------------------------------------------------*/
import com.ibm.mqe.*;
import com.ibm.mqe.administration.*;
import examples.application.*;
import java.io.File;
import examples.administration.simple.*;

public class SetupMQeExample1 extends ExampleAdminBase
{
      String routeDest1       =   "ServerQM1";
      String routeAddress1    =   "Network://127.0.0.1:8081";
      String routeDest2;

      String routeCommand1    =   null;
      String routeOptions1    =   null;
      static String iniFileName;
public SetupMQeExample1() throws Exception {
   super();
}
public SetupMQeExample1(String args[]) throws Exception {
   routeDest1 = args[0]; // MQe server
   routeAddress1 = "Network://" + args[2] + ":" + args[3]; // route address
   routeCommand1 = null;
   routeOptions1 = null;
   routeDest2 = args[4]; // MQSeries (not MQe) queue manager name
}
public SetupMQeExample1(String QMgrName) throws Exception {
   super(QMgrName);
}
protected void addRoutes() throws Exception {
   System.out.println("..Setup an admin message to add some routes ");

   // Create an empty queue manager admin message and parameters field
   MQeConnectionAdminMsg msg = new MQeConnectionAdminMsg(routeDest1);
```

```
   // Prime message with who to reply to and a unique identifier
   MQeFields msgTest = primeAdminMsg(msg);

   // Set the admin action to create a new queue
   // The connection is setup to use a default channel. This is an alias
   // which must have be setup on the queue manager for the connection to be
   // valid.
   msg.create(routeAddress1, routeCommand1, routeOptions1,
              "DefaultChannel", "Example route 1");

   //  Put the admin message to the admin queue
   System.out.println("..Put admin message to QM/queue: " + qMgrName + "/AdminQ");
   myQM.putMessage(qMgrName, "AdminQ", msg, null, 0);

   // Wait a while for the response message
   MQeAdminMsg respMsg = waitForReply(msgTest);
   checkReply(respMsg);

   // Create a named connection to the message broker, but specifying
   // the address of the MQe server QM. Requests will be sent to the
   // MQe Server QM, which will in turn put them to the message brokers
   // BridgeQueue.
   MQeConnectionAdminMsg msg2 = new MQeConnectionAdminMsg(routeDest2);

   // Prime message with who to reply to and a unique identifier
   MQeFields msgTest2 = primeAdminMsg(msg2);

   // Set the admin action to create a new queue
   // The connection is setup to use a default channel. This is an alias
   // which must have be setup on the queue manager for the connection to be
   // valid.
   msg2.create(routeAddress1, routeCommand1, routeOptions1,
               "DefaultChannel", "Example route 2");

   //  Put the admin message to the admin queue
   System.out.println("..Put admin message to QM/queue: " + qMgrName + "/AdminQ");
   myQM.putMessage(qMgrName, "AdminQ", msg2, null, 0);

   // Wait a while for the response message
   MQeAdminMsg respMsg2 = waitForReply(msgTest2);
   checkReply(respMsg2);
   System.out.println("..Added route to the message broker");
   System.out.println("..Add Routes Successful");
}
/**
 * Creates the StoreAndForwardQueue that will hold messages from the broker
 */
protected void createSandFQueue() throws Exception {
   String newForwardQueue = "ForwardQueue";
   System.out.println("..Setup an admin message to create queue " + newForwardQueue);
   // Now create store and forward queue

   MQeStoreAndForwardQueueAdminMsg msg =
      new MQeStoreAndForwardQueueAdminMsg(qMgrName, newForwardQueue);

   MQeFields parms = new MQeFields();

   // Prime message with who to reply to and a unique identifier
   MQeFields msgTest = primeAdminMsg(msg);

   // Set the admin action to create a new queue
   msg.create(parms);

   //  Put the admin message to the admin queue
   System.out.println("..Put admin message to QM/queue: " + qMgrName + "/AdminQ");
   System.out.println("going to put msg now....");
   myQM.putMessage(qMgrName, "AdminQ", msg, null, 0);
```

```
   // Wait a while for the response message
   MQeAdminMsg respMsg = waitForReply(msgTest);

   // Check that a good reply was received
   checkReply(respMsg);
   System.out.println("StoreAndForward Queue created successfully");
}
/**
 * All work is performed in this method
 */
public void doIt() throws Exception {
   try {
      System.out.println("Adding route to: " + routeDest2);
      // Add route to remote QM
      addRoutes();
      // Create a store and forward queue
      createSandFQueue();
   } catch (Exception e) {
      e.printStackTrace();
      System.out.println(example + " Failed! " + e);
   } finally {
      // Clean up
      close();
   }
}
public static void main(String args[]) throws Exception {
   // Check that we have the correct number of parameters
   if (args.length == 5) {
      // Create an instance of the class and call the doIt() method
      iniFileName = args[1]; // Client config file name
      new SetupMQeExample1(args).doIt();
   } else {
      System.out.println("The required parameters are:");
      System.out.println("1] MQe queue manager name");
      System.out.println("2] Client configuration .ini file");
      System.out.println("3] IP address of server");
      System.out.println("4] Port number used by MQe server");
      System.out.println("5] MQSeries queue manager name used by the broker:");
   }
}
/**
 * Over-ride default parameter settings.
 */
public void setParms() {
   startupIni = iniFileName;
}
}
```

SetupMQeExample2.java

```
package com.ibm.broker.mqimqe.example;

/*--------------------------------------------------------------------------*/
/* Licensed Materials - Property of IBM                                     */
/*                                                                          */
/*                              MQSeries Everyplace                         */
/*                              ===================                         */
/*                                                                          */
/*                                                                          */
/* Copyright IBM Corp 2001.  All rights reserved.                           */
/*                                                                          */
/* US Government Users Restricted Rights - Use, Duplication or disclosure   */
/* restricted by GSA ADP Schedule Contract with IBM Corp.                   */
/*--------------------------------------------------------------------------*/
import com.ibm.mqe.*;
import com.ibm.mqe.administration.*;
```

MQSeries Everyplace queue manager setup files

```
import examples.application.*;
import examples.administration.simple.*;
import java.io.File;

public class SetupMQeExample2 extends ExampleAdminBase {
   String newQueue = "Inbox";
   static String iniFileName;
public SetupMQeExample2(String QMgrName) throws Exception {
   super(QMgrName);
}
/**
 * create a "real" queue to place messages from the message broker
 */
protected void createMsgQueue() throws Exception {
   System.out.println("..Setup an admin message to create queue " + newQueue);

   // Create an empty queue admin message and parameters field
   MQeQueueAdminMsg msg2 = new MQeQueueAdminMsg(qMgrName, newQueue);
   MQeFields parms = new MQeFields();

   // Prime message with who to reply to and a unique identifier
   MQeFields msgTest = primeAdminMsg(msg2);

   // Set the admin action to create a new queue
   msg2.create(parms);

   // Put the admin message to the admin queue
   System.out.println("..Put admin message to QM/queue: " + qMgrName + "/AdminQ");
   myQM.putMessage(qMgrName, "AdminQ", msg2, null, 0);

   // Wait a while for the response message
   MQeAdminMsg respMsg = waitForReply(msgTest);

   // Check that a good reply was receive
   checkReply(respMsg);
   System.out.println("..Queue creation Successful");
}
/**
 * All work is performed in this method
 */
public void doIt() throws Exception {
   try {
      // create a queue called Inbox
      createMsgQueue();
   } catch (Exception e) {
      e.printStackTrace();
      System.out.println(example + " Failed! " + e);
   } finally {
      // Clean up
      close();
   }
}
public static void main(String args[]) throws Exception {
   // Check that we have the correct number of parameters
   if (args.length == 2) {
      // Create an instance of the class and call the doIt() method
      iniFileName = args[1]; // Client config file name
      new SetupMQeExample2(args[0]).doIt();
   } else {
      System.out.println("The required parameters are:");
      System.out.println("1] MQe queue manager name");
      System.out.println("2] Client configuration .ini file");
   }
}
/**
 * Over-ride default parameter settings.
 */
```

```
public void setParms() {
   startupIni = iniFileName;
}
}
```

MQSeries Everyplace queue manager setup files

Appendix C. Files for sending MQSeries Everyplace messages

Following, are two listings of example MQSeries Everyplace configuration files:

- **SendMessages.java** is for point-to-point messages.
- **SendPubSubMessages.java** is for publish/subscribe messages.

Some of the atributes which you must set for the MQSeries-bridge queue behave differently when used in MQSeries Integrator. Note the following:

- **MQeMbConstants.TYPE_OF_MSG** is a mandatory field. The MQSeries-bridge only supports one type of message. The MQSeries Everyplace node supports four types of messages, these are non publication, publication, subscription and unsubscription messages.
- **MsgID** and **CorrelID** are mandatory fields and must be the full length. This differs from the MQSeries-bridge where field lengths may be shorter than those required.
- **MQe.Msg_ExpireTime** is a manditory field. To specify an infinate amount of time, the value -1 may be used.
- **MQeMbConstants.MQ_DestQueueMgr** and **MQeMbConstants.MQ_DestQueueName** are new fields which are used to specify a returning queue manager and queue name for messages returning to MQSeries Everyplace after being processed by the message broker.
- **MQeMbConstants.MQ_AccountingToken** is a mandatory field.

Note: In the following listings, a ¬ symbol is used to indicate a continuation line where the code is too long to fit on the page.

SendMessages.java

```
package com.ibm.broker.mqimqe.example;

/*--------------------------------------------------------------------------*/
/* Licensed Materials - Property of IBM                                     */
/*                                                                          */
/*                              MQSeries Everyplace                         */
/*                              ===================                         */
/*                                                                          */
/*                                                                          */
/* Copyright IBM Corp 2001.  All rights reserved.                           */
/*                                                                          */
/* US Government Users Restricted Rights - Use, Duplication or disclosure   */
/* restricted by GSA ADP Schedule Contract with IBM Corp.                   */
/*--------------------------------------------------------------------------*/
import com.ibm.mqe.*;
import com.ibm.mqe.administration.*;
import examples.application.*;
import examples.administration.simple.*;
import java.io.File;
import com.ibm.broker.mqimqe.wrapper.*;
import java.io.*;

public class SendMessages extends ExampleAdminBase
{
   static String destQueueManager;
   static String destBridgeQueue;
```

```
   static String ServerMQe;
   static String InboxQueue;
   static String iniFileName;
public SendMessages(String QMgrName) throws Exception {
   super(QMgrName);
}
/**
 * All work is performed in this method
 * This time, sending messages as MQe-encoded format
 */
public void doIt() throws Exception {
   // Ensure that there are currently no messages on the queue
   flushQ();

   System.out.println("\n\nSending non publish subscribe message to the broker");
   System.out.println("This is constructed using an MQeMsgObject object so all" +
                      " fields will be passed into the message broker");
   // Send MQe message into the message broker
   nonPubSub_MQe();
   // Read the message returned from the message broker
   readqueueMQe();
   System.out.println("\n\nSending non publish subscribe message to the broker");
   System.out.println("This is constructed using an MQeMbMsgObject so only the" +
                      " fields compatible within the message broker will be sent");
   nonPubSub_Mb();
   // Read the message returned from the message broker
   readqueueMb();

   // Cleanup
   close();
}
public void flushQ() throws Exception {
   MQeMsgObject msgObj = null;
   do {
      try {
         msgObj = myQM.getMessage(ServerMQe, InboxQueue, null, null, 0);
      } catch (Exception e) {
         msgObj = null;
      }
   } while (msgObj != null);
}
public static void main(String args[]) throws Exception {
   // Check that we have the correct number of parameters
   if (args.length == 5) {
      // Set the parameters
      ServerMQe = args[0];
      iniFileName = args[1];
      destQueueManager = args[2];
      destBridgeQueue = args[3];
      InboxQueue = args[4];

      // Create an instance of the class and call the doIt() method
      new SendMessages(ServerMQe).doIt();
   } else {
      System.out.println("The required parameters are:");
      System.out.println("1] MQe queue manager name");
      System.out.println("2] Client configuration .ini file");
      System.out.println("3] MQSeries queue manager name used by the broker");
      System.out.println("4] Bridge queue name");
      System.out.println("5] Queue name to place the returning messages" +
                         " from the message broker");
   }
}
public void nonPubSub_Mb() throws Exception {
   try {
      //System.out.println("Non pub sub - Mb message object");
      MQeMbMsgObject msg = new MQeMbMsgObject();
```

```
      msg.putInt(MQeMbConstants.TYPE_OF_MSG, MQeMbConstants.TYPE_MQE);
      msg.setCorrelationId(
         MQe.hexToByte("0100000000000000000000000000000000000000000000000000"));
      msg.setDestQueueMgr(ServerMQe);
      msg.setDestQueueName(InboxQueue);
      msg.setReplyToQueueManagerName(myQM.getName());
      msg.setReplyToQueueName(MQe.System_Default_Queue_Name);
      msg.setMessageId(
         MQe.hexToByte("414D51204172676F4D51652020202020E636EB3813211200"));
      msg.setExpiry(-1);
      msg.setGroupId(
         MQe.hexToByte("0000000000000000000000000000000000000000000000000001"));
      msg.setAccountingToken(
         MQe.hexToByte("00000000000000000000000000000000000000000000000000000000¬
                                                          0000000000001"));
      msg.setPersistence(0);
      msg.setMessageType(8);

      // Note that this data is passed as is into the message broker but
      // only one field is supported.  This differs from the MQeMsgObject
      // which is not easily parsable within the message broker although
      // passes all avliable fields
      byte[] theData = asciiToByte("This is the payload");
      msg.setData(theData);

      myQM.putMessage(destQueueManager, destBridgeQueue, msg, null, 0);
   } catch (Exception e) {
      e.printStackTrace();
      System.out.println(example + " Failed! " + e);
   }
}
public void nonPubSub_MQe() throws Exception {
   try {
      //System.out.println("Non pub sub - MQe message object");
      MQeMsgObject msg = new MQeMsgObject();
      msg.putInt(MQeMbConstants.TYPE_OF_MSG, MQeMbConstants.TYPE_MQE);
      msg.putArrayOfByte(MQe.Msg_CorrelID,
         MQe.hexToByte("0100000000000000000000000000000000000000000000000000"));
      msg.putAscii(MQeMbConstants.MQ_DestQueueMgr, ServerMQe);
      msg.putAscii(MQeMbConstants.MQ_DestQueueName, InboxQueue);
      msg.putAscii(MQe.Msg_ReplyToQMgr, myQM.getName());
      msg.putAscii(MQe.Msg_ReplyToQ, MQe.System_Default_Queue_Name);
      msg.putArrayOfByte(MQe.Msg_MsgID,
         MQe.hexToByte("414D51204172676F4D51652020202020E636EB3813211200"));
      msg.putArrayOfByte(MQeMbConstants.MQ_GroupId,
         MQe.hexToByte("0000000000000000000000000000000000000000000000000001"));
      msg.putArrayOfByte(MQeMbConstants.MQ_AccountingToken,
         MQe.hexToByte("01000000000000000000000000000000000000000000000000¬
                                                  0000000000000000"));
      msg.putInt(MQe.Msg_ExpireTime, -1);
      msg.putInt(MQeMbConstants.MQ_Persistence, 0);
      msg.putInt(MQeMbConstants.MQ_MessageType, 8);

      // Note that this data is not easily parsable within the message
      // broker although all fields provided are passed.  This is unlike
      // the MQeMbMsgObject which only supports one payload field although
      // is more parsable within the message broker.

      // Data
      byte[] theData = asciiToByte("This is the payload");
      msg.putArrayOfByte(MQeMbConstants.MQ_Data, theData);

      // Own Data
      String ownData = "My own tag with some data";
      msg.putAscii("myOwnTag", ownData);

      // Put the message to the bridge queue
```

```
            myQM.putMessage(destQueueManager, destBridgeQueue, msg, null, 0);
         } catch (Exception e) {
            e.printStackTrace();
            System.out.println(example + " Failed! " + e);
         }
      }
      public void readqueueMb() throws Exception {
         try {
            int limit = 10;      // The number of seconds to wait for a message
            System.out.println("Reading message from queue");

            MQeMsgObject msgObj = null;
            while ((msgObj == null) && (limit-- > 0)) {
               try {
                  msgObj = myQM.getMessage(ServerMQe, InboxQueue, null, null, 0);
               } catch (Exception e) {
                  msgObj = null;
                  Thread.sleep(1000);
               }
            }
            byte data[] = msgObj.getArrayOfByte(MQeMbConstants.MQ_Data);
            System.out.println("Message:         " + byteToAscii(data));
         } catch (Exception e) {
            System.out.println("The message retrieved did not contain all of" +
                               " the expected fields");
         }
      }
      public void readqueueMQe() throws Exception {
         try {
            int limit = 10;      // The number of seconds to wait for a message
            System.out.println("Reading message from queue");

            MQeMsgObject msgObj = null;
            while ((msgObj == null) && (limit-- > 0)) {
               try {
                  msgObj = myQM.getMessage(ServerMQe, InboxQueue, null, null, 0);
               } catch (Exception e) {
                  msgObj = null;
                  Thread.sleep(1000);
               }
            }

            // Extract the data
            byte data[] = msgObj.getArrayOfByte(MQeMbConstants.MQ_Data);
            String ownData = msgObj.getAscii("myOwnTag");

            // Display the data
            System.out.println("Message:         " + byteToAscii(data));
            System.out.println("Own message tag: " + ownData);
         } catch (Exception e) {
            System.out.println("The message retrieved did not contain all of" +
                               " the expected fields");
         }
      }
      /**
       * Over-ride default parameter settings.
       */
      public void setParms() {
         startupIni = iniFileName;
      }
      }
```

SendPubSubMessages.java

```
package com.ibm.broker.mqimqe.example;

/*--------------------------------------------------------------------------------*/
/* Licensed Materials - Property of IBM                                           */
/*                                                                                */
/*                              MQSeries Everyplace                               */
/*                              ===================                               */
/*                                                                                */
/*                                                                                */
/* Copyright IBM Corp 2001.  All rights reserved.                                 */
/*                                                                                */
/* US Government Users Restricted Rights - Use, Duplication or disclosure        */
/* restricted by GSA ADP Schedule Contract with IBM Corp.                         */
/*--------------------------------------------------------------------------------*/
import com.ibm.mqe.*;
import com.ibm.mqe.administration.*;
import examples.application.*;
import examples.administration.simple.*;
import java.io.File;
import com.ibm.broker.mqimqe.wrapper.*;

public class SendPubSubMessages extends ExampleAdminBase
{
   static String destQueueManager;
   static String destBridgeQueue;
   static String ServerMQe;
   static String InboxQueue;
   static String iniFileName;
public SendPubSubMessages(String QMgrName) throws Exception {
   super(QMgrName);
}
public void doIt() throws Exception {
   // Ensure that there are currently no messages on the queue
   flushQ();
   // Send a subscription message
   Subscribe();
   // Sens a publication message
   Publish();
   // Read any messages published for the given subscription
   readqueue();
   // Send an unsubscription message
   UnSubscribe();
   // Cleanup
   close();
}
public void flushQ() throws Exception {
   MQeMsgObject msgObj = null;
   do {
      try {
         msgObj = myQM.getMessage(ServerMQe, InboxQueue, null, null, 0);
      } catch (Exception e) {
         msgObj = null;
      }
   } while (msgObj != null);
}
public static void main(String args[]) throws Exception {
   // Check that we have the correct number of parameters
   if (args.length == 5) {
      // Set the parameters
      ServerMQe = args[0];
      iniFileName = args[1];
      destQueueManager = args[2];
      destBridgeQueue = args[3];
      InboxQueue = args[4];
```

```
            // Create an instance of the class and call the doIt() method
            new SendPubSubMessages(ServerMQe).doIt();
         } else {
            System.out.println("The required parameters are:");
            System.out.println("1] MQe queue manager name");
            System.out.println("2] Client configuration .ini file");
            System.out.println("3] MQSeries queue manager name used by the broker");
            System.out.println("4] Bridge queue name");
            System.out.println("5] Queue name to place the returning messages from" +
                               " the message broker");
         }
      }
      public void Publish() throws Exception {
         try {
            String topic = "weather/climate";
            byte[] message = asciiToByte("cold");
            System.out.println("Publishing message to topic '" + topic + "'");

            // Create a message object
            MQeMsgObject msg = new MQeMsgObject();
            // Set the message type to publish
            msg.putInt(MQeMbConstants.TYPE_OF_MSG, MQeMbConstants.TYPE_PUB);
            // Set the topic for the publication message
            msg.putAscii(MQeMbConstants.TOPIC, topic);
            // Insert the payload
            msg.putArrayOfByte(MQeMbConstants.MESSAGE, message);
            // Set the persistence of the message (zero is non-persistent)
            msg.putInt(MQeMbConstants.MQ_Persistence, 0);
            // Specify that the message is not persistant
            msg.putBoolean(MQeMbConstants.RETAINED, false);
            // Put the message to the MQe bridge queue
            myQM.putMessage(destQueueManager, destBridgeQueue, msg, null, 0);
         } catch (Exception e) {
            e.printStackTrace();
            System.out.println(example + " Failed! " + e);
         }
      }
      public void readqueue() throws Exception {
         try {
            int limit = 10;      // The number of seconds to wait for a message
            System.out.println("Reading message from queue");

            MQeMsgObject msgObj = null;
            while ((msgObj == null) && (limit-- > 0)) {
               try {
                  msgObj = myQM.getMessage(ServerMQe, InboxQueue, null, null, 0);
               } catch (Exception e) {
                  msgObj = null;
                  Thread.sleep(1000);
               }
            }
            // Extract the relevant data from the object and display it
            String topic = msgObj.getAscii(MQeMbConstants.TOPIC);
            byte data[] = msgObj.getArrayOfByte(MQeMbConstants.MESSAGE);
            // Display the data
            System.out.println("Topic:  " + topic);
            System.out.println("Message: " + byteToAscii(data));
         } catch (Exception e) {
            System.out.println("No message found");
         }
      }
      /**
       * Over-ride default parameter settings.
       */
      public void setParms() {
         startupIni = iniFileName;
      }
```

```
public void Subscribe() throws Exception {
   try {
      String[] topics = {"weather/climate", "weather/humidity", "weather/temperature"};
      System.out.println("Subscribing to the topics:");
      for (int numTopics = 0; numTopics < topics.length; numTopics++) {
         System.out.println(topics[numTopics]);
      }
      // Create the message object
      MQeMsgObject msg = new MQeMsgObject();
      // Set the message type to a subscription message
      msg.putInt(MQeMbConstants.TYPE_OF_MSG, MQeMbLaunch.TYPE_SUB);
      // Set the three topics
      msg.putAsciiArray(MQeMbConstants.TOPIC, topics);
      // Set the queue and queue manager to place any messages for this subscription
      msg.putAscii(MQeMbConstants.MQ_DestQueueName, "Inbox");
      msg.putAscii(MQeMbConstants.MQ_DestQueueMgr, "ServerQM1");
      // Put the subscription message to the bridge queue within MQe
      myQM.putMessage(destQueueManager, destBridgeQueue, msg, null, 0);
   } catch (Exception e) {
      e.printStackTrace();
      System.out.println(example + " Failed! " + e);
   }
}
public void UnSubscribe() throws Exception {
   try {
      String[] topics = {"weather/climate", "weather/humidity", "weather/temperature"};
      System.out.println("un-subscribing from the topics:");
      for (int numTopics = 0; numTopics < topics.length; numTopics++) {
         System.out.println(topics[numTopics]);
      }
      // Create the message object
      MQeMsgObject msg = new MQeMsgObject();
      // Set the message type to an unsubscription message
      msg.putInt(MQeMbConstants.TYPE_OF_MSG, MQeMbLaunch.TYPE_UNSUB);
      // Set the topics that we want to unsubscribe from
      msg.putAsciiArray(MQeMbConstants.TOPIC, topics);
      // Specify the queue and queue manager that we specified to place messages to
      msg.putAscii(MQeMbConstants.MQ_DestQueueName, "Inbox");
      msg.putAscii(MQeMbConstants.MQ_DestQueueMgr, "ServerQM1");
      // Put the unsubscription message to the bridge queue within MQe
      myQM.putMessage(destQueueManager, destBridgeQueue, msg, null, 0);
   } catch (Exception e) {
      e.printStackTrace();
      System.out.println(example + " Failed! " + e);
   }
}
}
```

Files for sending MQSeries Everyplace messages

Appendix D. Notices

This information was developed for products and services offered in the United States. IBM may not offer the products, services, or features discussed in this information in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this information. The furnishing of this information does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other 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 NON-INFRINGEMENT, 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.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the information. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this information at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Notices

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM United Kingdom Laboratories,
Mail Point 151,
Hursley Park,
Winchester,
Hampshire,
England
SO21 2JN.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Programming License Agreement, or any equivalent agreement between us.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both:

IBM	MQSeries	MQSeries Integrator
MQSeries Everyplace	SupportPac	

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

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

Other company, product, and service names may be trademarks or service marks of others.

Sending your comments to IBM

If you especially like or dislike anything about this book, please use one of the methods listed below to send your comments to IBM.

Feel free to comment on what you regard as specific errors or omissions, and on the accuracy, organization, subject matter, or completeness of this book.

Please limit your comments to the information in this book and the way in which the information is presented.

To make comments about the functions of IBM products or systems, talk to your IBM representative or to your IBM authorized remarketer.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate, without incurring any obligation to you.

You can send your comments to IBM in any of the following ways:

- By mail, to this address:

  User Technologies Department (MP095)
  IBM United Kingdom Laboratories
  Hursley Park
  WINCHESTER,
  Hampshire
  SO21 2JN
  United Kingdom
- By fax:
  - From outside the U.K., after your international access code use 44–1962–842327
  - From within the U.K., use 01962–842327
- Electronically, use the appropriate network ID:
  - IBM Mail Exchange: GBIBM2Q9 at IBMMAIL
  - IBMLink™: HURSLEY(IDRCF)
  - Internet: idrcf@hursley.ibm.com

Whichever method you use, ensure that you include:

- The publication title and order number
- The topic to which your comment applies
- Your name and address/telephone number/fax number/network ID.

Printed in U.S.A.