WebSphere MQ Everyplace V2.0.2

Contents

Deploying your application

Packaging and deployment

MQe is a flexible messaging system that can be deployed to a wide variety of operating systems and devices.

This section provides information to assist in the build, packaging and deployment of MQe.

It is split into two sections covering the Java™ code base and the native code base.

Because MQe can be deployed on a variety of devices, operating systems, and runtimes, it is not possible to detail each application. Hence in some topics only a brief outline and introduction is provided.

Java deployment

The MQe Java code base can be deployed onto a large variety of Java runtimes. These include:

- J2ME CLDC/MIDP
- J2ME CDC/Foundation
- PersonalJava V1.1
- Java 1.1
- J2SE 1.2 (or later)
- IBM® Rational® Software Development Platform Custom Environment jclGateway (or better)

The way that MQe, the application and other classes are packaged and deployed is dependant on the type of Java runtime, the operating system and processor type of the device that is being deployed to.

The following topics provide information to assist in packaging and deploying Java based MQe applications to different environments.

Supplied jar files

When deploying MQe applications, you are recommended to pack the minimum set of classes required by the application into compressed jar files. This ensures that the application requires the minimum system resources. MQe provides the following examples of how the MQe classes can be packaged into .jar files. These examples are in the<MQeInstallDir>\Java\Jars directory of a standard MQe installation.

There are three types of jar file; *base*, *extension*, and *other*:

- The base jar files allow a usable queue manager to be created, administered and run
- The extension jar files can be used in addition to the base jar files to provide additional capability
- The other jar files include example, and core, sets of classes for you to use as a base for your development

Base jar files

MQeBase.jar
: Contains classes that provide for a basic queue manager running in client and server mode on a J2ME CDC/Foundation or J2SE or better Java runtime.

MQeMidp.jar
: Similar to MQeBase.jar but for use on a J2ME CLDC/MIDP Java runtime. Allows a queue manager to run in client mode. All MIDP compliant classes are included in this jar. No extension jars can be used with this one, as they are not MIDP compliant.

MQeGateway.jar
Contains classes that provide for a basic queue manager running in client, server and bridge mode on a J2SE or better Java runtime.

Extension jar files

MQeJMS.jar
Contains the classes that extend an MQe queue manager to provide a JMS programming interface.

MQeRetail.jar
Contains extra classes for use in retail environments. In particular, these classes are useful on a 4690 retail system.

MQeSecurity.jar
A set of classes that are used to provide both queue and message based security. It contains a set of cryptors, compressors and authenticators.

MQeBindings.jar
This file contains all C bindings specific information. It is required if access to a Java queue manager from a C application is needed (only on Win32 platforms).

MQeMigration.jar
Contains classes that assist in migrating from an earlier version of MQe.

MQeDeprecated.jar
This contains all of the deprecated class files that are no longer needed by an MQe application. These deprecated class files help you run applications written using a previous version of MQe, without making any changes.

MQeDiagnostics.jar
This file helps to diagnose problems with MQe classes. It contains tooling to search the class path to find out the level of each class found.

MQeJMX.jar
Contains the classes needed to administer MQe using JMX.

MQeJMSAdmin.jar
Contains the classes needed to administer MQe using JMS.

MQeJMSSQL.jar
Contains the classes needed to administer MQe using SQL.

Other jar files

MQeExamples.jar
A packaging of all the MQe examples into one jar file. This includes all of the examples supplied with MQe, but excludes the deprecated classes.

MQeCore.jar
This contains a minimal set of classes. On its own it is not usable but it can be used as a base for building a small footprint MQe system. More details on reducing footprint can be found in “Optimizing footprint” on page 3.

MQeBundle.jar
This jar provides the MQe 'bundle' for OSGi frameworks and is a code-only bundle.

MQeClientBundle.jar
Example OSGi bundles showing how to run MQe within the OSGi framework.s

MQeServerBundle.jar
Example OSGi bundles showing how to run MQe within the OSGi framework.

OSGi-related examples

MQeJMSReceiver.jar
Example OSGi bundles using the JMS interface.

MQeJMSSender.jar
Example OSGi bundles using the JMS interface.

JMS-related examples

MQeJMSAdmin.jar
Provides classes to create and edit administered objects stored in a JNDI namespace.

MQeJMSSQL.jar
Provides the classes required if JMS SQL selectors are to be used.

MQeTraceDecode.jar
Provides classes to decode MQe trace file.

Optimizing footprint

In many cases the supplied jar files can be used without change, however there are instances where this is not the case. In particular, on some environments where footprint is limited, the set of classes that are deployed must be reduced to the smallest possible size. The supplied jar files are general purpose and contain more than is necessary for an optimized environment.

The table below separates the classes into groups associated with a particular function or configuration and will help determine which classes will be required to optimize an applications footprint. Using this table the minimum required set of classes can be deduced by taking the mandatory classes for the required categories and then adding in required optional classes for that category.

Due to the wide ranging set of Java runtimes that are now available, not all classes can run on all runtimes. The table lists all classes, and unless otherwise stated, each class will run on a J2SE runtime. Because of the differences between a J2SE and a J2ME runtime, some of the classes are not appropriate for a J2ME runtime. There are two columns marked with an X to show a class that can be used on J2ME MIDP or J2ME CDC/Foundation runtimes.

Table 1. Class optimization

Category	**Detail**		
Type	**Details**	**Midp**	**CDC**
Classes required (com.ibm.mqe)			
Mandatory classes			

Table 1. Class optimization (continued)

		For all queue managers	X	X

		`MQe`
		`MQeAdapter`
		`MQeAttribute`
		`MQeAttributeDefaultRule`
		`MQeAttributeRule`
		`MQeAuthenticator`
		`MQeCompressor`
		`MQeCryptor`
		`MQeEnumeration`
		`MQeException`
		`MQeExceptionCodes`
		`MQeField`
		`MQeFields`
		`MQeKey`
		`MQeLoaderMQeProperties`
		`MQePropertyProvider`
		`MQeQueueControlBlock`
		`MQeQueueProxy`
		`MQeQueueManager`
		`MQeQueueManagerRule`
		`MQeResourceControlBlock`
		`MQeRule`
		`MQeRunnable`
		`MQeRunnableInstance`
		`MQeThread`
		`MQeThreadPool$1`
		`MQeThreadPool$PooledThread`
		`MQeThreadPool$Target`
		`MQeThreadPool`
		`MQeTrace`
		`MQeTraceHandler`
		`MQeTraceInterface`
		`registry.MQeRegistry`		

Registry type		One option in this category must be selected		
	File registry	Add required: Storage adapter	X	X
		`registry.MQeFileSession` `registry.MQeRegistrySession`		
	Private registry w/o credentials	Add: File registry		X
		`registry.MQePrivateRegistry` `registry.MQePrivateSession`		
	Private registry with credentials	Add: Private registry w/o credentials		X

		`attributes.MQeMiniCertRequest`
		`attributes.MQeSharedKey`
		`attributes.MQeWTLSCertificate`		

			Mini-certificate management functions		X
		`attributes.MQeListCertificates` `registry.MQePrivateRegistryConfigure`		
	Public registry	Applicable to types of message-level security, Add: Private registry with credentials		X
		`registry.MQePublicRegistry`		
Queue manager type		For all types add required: Administration Storage adapters Message store Authenticators Cryptors Compressors Rules Security		

Table 1. Class optimization (continued)

	Standalone qMgr.	No additional classes		
	Client qMgr.	Add required: Communications	X	X

		`MQeTransporter`
		`adapters.MQeCommunicationsAdapter`
		`communications.MQeChannel`
		`communications.MQeChannelCommandInterface`
		`communications.MQeChannelControlBlock`
		`communications.MQeCommunicationsException`
		`communications.MQeCommunicationsManager`
		`communications.MQeConnectionDefinition`
		`communications.MQeListener`
		`communications.MQeListenerSlave`		

	Server qMgr.	Add: Client qMgr. Add required: Communications		X
		Note: whilst MQeListener is not used in the Client, they need to be included when preverifying a J2ME application		
	Gateway qMgr.	Add: Server qMgr. Add required Communications Transformers		

		`MQeBridgeLoadable`
		`MQeBridgeManager`
		`mqbridge.*`		

Communications				
	TCP/IP w/o history & persistence			X
		`adapters.MQeTcpipAdapter` `adapters.MQeTcpipLengthAdapter`		
	TCP/IP with history & persistence	Add: TCP/IP w/o history and persistence		X
		`adapters.MQeTcpipHistoryAdapter` `adapters.MQeTcpipHistoryAdapterElement`		
	HTTP 1.0 Not to WES Proxy Authentication server			X
		`adapters.MQeTcpipAdapter` `adapters.MQeTcpipHttpAdapter`		
	HTTP To WES Proxy Authentication server			X
		`adapters.MQeTcpipAdapter` `adapters.MQeWESAuthenticationAdapter`		
	HTTP 1.1/1.0 J2ME	MIDP only	X	
		`adapters.MQeMidpHttpAdapter`		
	UDP			X

		`adapters.MQeUdpipBasicAdapter$Initiator`
		`adapters.MQeUdpipBasicAdapter$InternalAdapter`
		`adapters.MQeUdpipBasicAdapter$Responder`
		`adapters.MQeUdpipBasicAdapter$Writer`
		`adapters.MQeUdpipBasicAdapter`		

Queue Types		For all queue types add required: Authenticators Cryptors Compressors Rules		

Table 1. Class optimization (continued)

	Local	Add: Storage adapter Message storage	X	X

		`MQeAbstractQueueImplementation`
		`MQeEventTrigger`
		`MQeMessageEvent`
		`MQeMessageListenerInterface`
		`MQeQueue`
		`MQeQueueRule (or replacement)`		

	Remote	Add: Local queue (storage adapter & msg. storage only if needed)	X	X
		`MQeRemoteQueue`		
	Home server	Add: Remote queue (no storage adapter or msg. storage)	X	X
		`MQeHomeServerQueue`		
	Store and forward	Add: Remote queue	X	X
		`MQeStoreAndForwardQueue`		
	Bridge queue	Add: Remote queue		

		`mqbridge.MQeMQBridgeAdminMsg`
		`mqbridge.MQeBridgeServices`
		`mqbridge.MQeMQBridgeQueue`
		`mqbridge.MQeMQQMgrName`
		`mqbridge.MQeMQQName`		

Message storage				
	Base		X	X

		`MQeMessageStoreException`
		`MQeAbstractMessageStore`
		`messagestore.MqeIndexEntry`		

	Standard	Add: Base	X	X
		`messagestore.MQeMessageStore`		
	Short filename. Always use 8.3 file name for messages.	Add: Standard		X
		`messagestore.MQeShortFilenameMessageStore`		
	4690 specific	Add: Short filename		
		`messagestore.MQe4690ShortFilenameMessageStore`		
Message type				
	Basic		X	X
		Support for MQeMsgObject is in Mandatory classes		
	WebSphere® MQ			
		`mqemqmessage.*`		
Storage adapters				
	Assured disk	Independence from OS lazy writes		X
		`adapters.MQeDiskFieldsAdapter`		
	Non-assured disk	Dependence on OS lazy writes Add: Assured disk		X
		`adapters.MQeReducedDiskFieldsAdapter`		
	Case-Insensitive	Add: Assured disk		X

Table 1. Class optimization (continued)

		`adapters.MQeCaseInsensitiveAdapter`		
	Long to Short Filename Mapping			X
		`adapters.MQeMappingAdapter`		
	Midp RMS Storage	MIDP Only	X	
		`adapters.MQeMidpFieldsAdapter` `com.ibm.mqe.adapters.MQeMidpFieldsAdapter$RMSFile`		
	Memory	Volatile storage	X	X
		`adapters.MQeMemoryFieldsAdapter`		
Administration				
	Basic administration capability	Add: Local queue	X	X

		`MQeAdminMsg`
		`MQeAdminQueue`
		`MQeAdminQueue$1`
		`MQeAdminQueue$Timer`		

	Manage queue manager	Add: Basic administration capability	X	X
		`administration.MQeQueueManagerAdminMsg`		
	Manage connection definitions	Add: Basic administration capability	X	X
		`administration.MQeConnectionAdminMsg`		
	Manage communications listeners	Add: Basic administration capability	X	X
		`administration.MQeCommunicationsListenerAdminMsg`		
	Manage local queue	Add: Basic administration capability	X	X
		`administration.MQeQueueAdminMsg`		
	Manage administration queue	Add: Manage local queue	X	X
		`administration.MQeAdminQueueAdminMsg`		
	Manage remote queue	Add: Manage local queue	X	X
		`administration.MQeRemoteQueueAdminMsg`		
	Manage home server queue	Add: Manage remote queue	X	X
		`administration.MQeHomeServerQueueAdminMsg`		
	Manage store and forward queue	Add: Manage remote queue	X	X
		`administration.MQeStoreAndForwardQueueAdminMsg`		
	Manage bridge queue	Add: Manage remote queue		X
		`mqbridge.MQeMQBridgeQueueAdminMsg` `mqbridge.MQeCharacteristicLabels`		
	WebSphere MQ	Add: Remote queues		

		`mqbridge.*AdminMsg`
		`mqbridge.MqeCharacteristicLabels`
		`mqbridge.MqeRunState`
		`mqbridge.MqeBridgeServices`
		`mqbridge.MQeBridgeExceptionCodes`		

Table 1. Class optimization (continued)

Queue manager creation and deletion		`MQeQueueManagerConfigure`	X	X
Authenticators				
	mini-certificate			X

	`attributes.DHk (source may be generated)`
	`attributes.MQeSharedKey`
	`attributes.MQeRandom`
	`attributes.MQeWTLSCertificate`
	`attributes.MQeWTLSCertAuthenticator`			

Compressors				
	GZIP	`attributes.MQeGZIPCompressor`		X
	LZW	`attributes.MQeLZWCompressor` `attributes.MQeLZWDictionaryItem`	X	X
	RLE	`attributes.MQeRleCompressor`	X	X
Cryptors				
	triple DES	`attributes.MQe3DESCryptor`		X
	DES	`attributes.MQe3DESCryptor`		X
	MARS	`attributes.MQeDESCryptor`		X
	RC4	`attributes.MQeRC4Cryptor`		X
	RC6	`attributes.MQeRC6Cryptor`		X
	XOR	`attributes.MQeXorCryptor`	X	X
Application security services				
	Local security	Add required: Cryptors	X	X
	`attributes.MQeLocalSecure`			
	Message-level security	Add required: Cryptors		X
	`attributes.MQeMAttribute`			
	Message-level security with digital signature & validation	Add: Public registry. Add required: Cryptors		X
	`attributes.MQeMTrustAttribute`			
Trace				

Table 1. Class optimization (continued)

	Collect binary trace in J2SE/CDC			X
		`trace.MQeTraceToBinary` `trace.MQeTraceToBinaryFile`		
	Collect binary trace to Midp RMS Store And or send to MIDP Trace servlet		X	
		`trace.MQeTraceToBinary` `trace.MQeTraceToBinaryMidp`		
	Base trace renderer			X

		`trace.MQeTracePoint`
		`trace.MQeTracePointGroup`
		`trace.MQeTraceRenderer`		

	Decode a binary file to readable form	Add: Base trace renderer		X
		`trace.MQeTraceToReadable` `trace.MQeTraceFromBinaryFile`		
	Trace to a readable output stream	Add: Base trace renderer		X
		`trace.MqeTraceToReadable`		
	Servlet collection of Midp binary trace	Add Base trace renderer		
		`trace.MQeTraceToReadable` `examples.trace.MQeServlet`		
Miscellaneous				
	Cryptographic support	Application or installation use only		X
		`attributes.MQeCL (footnote?)` `attributes.MQeGenDH (generates a version of attributes.MQeDHk.java)`		
	MQeServerSupport SupportPac™ ES06	MQeServerSupport (See ES06 installation instructions)		
Bindings		Access to Java classes from other languages		
	C language	`bindings.*`		
JMS		Support for the Java Message Service API		XX
		`jms.*` `transaction.*`		
User-defined MQe extensions				
			Authenticators Communications adapters Compressors Cryptors Logging classes Message classes Rule classes Security control Storage adapters Trace handler	

JMS requirements

In order to use the MQe JMS programming interface, the JMS interface classes are required.

These are contained typically in jms.jar.

MQe does not ship with jms.jar, and this must be downloaded before JMS can be used.

At the time of writing, this can be freely downloaded from http://java.sun.com/products/jms/docs.html. The JMS Version 1.0.2b jar file is required.

JNDI

In addition, if JMS administered objects are to be stored and retrieved using the Java Naming and Directory Interface (JNDI), the javax.naming.* classes must be available.

If Java 1 is being used, for example, a 1.1.8 JRE, jndi.jar must be obtained and added to the classpath.

If Java 2 is being used, for example a 1.2 or later JRE, the JRE might contain these classes.

You can use MQe without JNDI, but at the cost of a small degree of provider dependence. MQe-specific classes must be used for the ConnectionFactory and Destination objects.

You can download JNDI jar files from http://java.sun.com/products/jndi

MQe classes for Java requirements

To use the MQ bridge the *MQ Classes for Java* are required, version 5.1 or later.

These are packaged with MQ 5.3 and above. If using an earlier version of MQ then they are available for free download from the Web as SupportPac MA88.

For an example of how to setup the classpath to include MQ jar files, see batch files:

- <MqeInstallDir>\Java\Demo\Windows\javaenv.bat
- <MqeInstallDir>\Java\Demo\UNIX\javaenv

Occasionally, the jar files change between versions of MQ - if problems are encountered as a consequence of this, consult the documentation for MQ classes in order to determine the correct jar files to use.

Using Rational Device Developer smart linker

The smart linker tool that ships with Rational Software Development Platform is used in the process of building and packaging an application into a jar or jxe file. The smart linker can remove classes (and methods) that are deemed not to be required; this can cause the removal of classes that are needed but dynamically loaded. MQe makes use of dynamic loading so care should be taken to either avoid this feature or to explicitly name classes that must be present, even though not explicitly referenced in the code.

To prevent unused classes being removed use the `-noRemoveUnused` option.

Otherwise, if the `-removeUnused` option is set then any class that is dynamically loaded must be specifically included. One option that can be used to achieve this is `-includeWholeClass`.

For example

```
-includeWholeClass "com.ibm.mqe.adapters.*"
```

will include all classes in the adapters package, and

```
-includeWholeClass "com.ibm.mqe.adapters.MQeTcpipHttpAdapter"
```

will include only the http adapter.

Multiple include (or exclude) options can be specified in the smart linker options file.

The following guidelines can be used to determine which classes are dynamically loaded. The basic guideline is any class that is referenced though an MQe class alias or any class that is set as a parameter when administering MQe resources will be dynamically loaded. This includes:

- Communications adapters
- Storage adapters
- Message stores
- Rules
- Aliases
- Cryptors
- Compressors
- Authenticators
- Queues
- Transporter
- Connection (refer to the following example)

An example of a set of includes needed for a simple MIDP application is:

```
-includeWholeClass "com.ibm.mqe.MQeQueue"
-includeWholeClass "com.ibm.mqe.MQeRemoteQueue"
-includeWholeClass "com.ibm.mqe.MQeHomeServerQueue"
-includeWholeClass "com.ibm.mqe.MQeTransporter"
-includeWholeClass "com.ibm.mqe.communications.MQeConnectionDefinition"
-includeWholeClass "com.ibm.mqe.adapters.MQeMidpFieldsAdapter"
-includeWholeClass "com.ibm.mqe.adapters.MQeMidpHttpAdapter"
-includeWholeClass "com.ibm.mqe.messagestore.MQeMessageStore"
-includeWholeClass "com.ibm.mqe.registry.MQeFileSession"
```

J2ME Midp specifics

When deploying the Java application for the Midp environment, a few additional comments are worth mentioning.

- You must use the Midp-specific Storage and Communication adapters (see "Using Rational Device Developer smart linker" on page 10) and exclude any classes that are not Midp compliant.
- You can either use the prepackaged MQeMidp.jar file or your own reduced version, however a JAD file (Java application descriptor) must also be included detailing the Midlets available within the application. When deploying to the device all classes should be packaged and preverified in one jar before deploying. However, while testing using an emulator several jars can be used by including them in the classpath.
- Care must be taken to ensure that all the required classes are included in either the jar/prc file or other executable. Some classes are dynamically loaded and may be missed when using any Smart-Linker. See "Using Rational Device Developer smart linker" on page 10 for more details.

4690 specifics

Take the following requirements into account when configuring MQe for use with 4690.

- Terminal Applications are restricted to 24 character maximum path length, but Store Controller Applications can have 127 characters. Java Applications are also restricted to the 24 length.
- The virtual file system (VFS) cannot hold greater than 64,000 files. With GB disk sizes being used, the C: drive may not have a limit on the number of files, depending on your operating system.
- When you want to access a file, you must specify the path that leads to it. The path consists of directory names that are separated by a backslash character "\" or a forward slash "/".

  Note: Although your system accepts both the "\" and the "/" character, it is probably less confusing to use one or the other.

- Examples elsewhere in this manual demonstrate how to configure your queue manager such that the data describing its resources, certificates, and other configuration data is stored in files with long filenames. These filenames are for a single top-level directory, which can also be located on the VFS drive namespace.
- Using the 8.3 format, the total character length of the fully-qualified filename exceeds the allowable limits imposed by the 4960 native file system. Therefore, in VFS :
  - The maximum length of a filename is 256 characters.
  - The maximum path length, including directories and files, is 260 characters.
  - The maximum directory depth is 60 levels including the root directory.
- MQe classes can be stored in long format names in VFS. However, for performance and convenience, as there are lots of class files, we would recommend that the application and MQe classes are packaged into a .jar files and deployed.
- According to the VFS manual ″The operating system provides support for file names greater than eight characters in length through the use of a 4690 Virtual File System (VFS)″.
- The VFS manual states: ″The VFS drive setting must be enabled through system configuration. On enabling VFS drive settings, the operating system creates two logical drives. C: and D:. The drive determines where the VFS directory is located. However, the information is actually stored on drives C: and D:. Drive M: information is stored on drive C:, and drive N: information is stored on drive D:. Once you have enabled VFS, you can use drives M: and N: to provide long file name support locally.″
- It is recommended that you use the `MQeCaseInsensitiveDiskAdapter` on the 4690 OS. This class implements a disk adapter that is insensitive to the case (upper or lower) of the filename used during matching. Some JVM or OS combinations list files with different case to that in which they were created. This means that the simple filtering in the superclass ignores them. However this class converts both the comparator and the comparand to lowercase before performing the comparison. This ensures the best chance of finding a valid match. Note that the conversion to lower case may be inappropriate on platforms where the case is honoured, and where there are non-MQe files stored that could be confused by case. In summary, this adapter is suited for use with the 4690 filesystem.

Packaging

Following is a list of some of the techniques and tools that can be used to package applications ready for deployment to a device. The list is not a full list and does not go into any detail but is intended to provide an introduction to some of the ways a Java application can be packaged.

Single Jar file

Build a self-contained application with MQe embedded in it. This option minimizes the footprint and ensures that the classpath is kept to a minimum.

Multiple Jar files

Put the application into one jar file, and then also use either the supplied MQe jar files or construct a separate MQe jar file. Keeping MQe in one or more separate jars makes it easy to use MQe from multiple independent applications.

JNLP

JNLP (Java Network Launching Protocol and API) is an emerging standard for use in packaging and deploying Java applications. It is designed to automate the deployment, via the web, for applications written for the J2SE platform.

OSGi

OSGi or Open Services Gateway Initiative defines define a platform for the packaging of and dynamic delivery of Java software services to networked devices. This is achieved via a consistent, component-based, architecture for the development and delivery of Java software components known as bundles and services. Both MQe components and applications can be turned into OSGi bundles and services for use in an OSGi environment. The bundles are delivered from a bundle server There are several products that provide bundle servers together

with the client code to handle the installation and life cycle of bundles. Depending on implementation the bundles can be downloaded on demand, and updated automatically when a new version is available. IBM Rational Software Development Platform ships with SMF (Service Management Framework), which assists in the creation and testing of bundles together with a bundle server.

See more at "Open Services Gateway initiative (OSGi)" on page 14.

Midlet

An MQe J2ME MIDP application must be packaged as a midlet or midlet suite (.jad and .jar).

JXE

IBM Rational Device Developer has a SmartLinker tool that can produce an optimized packaging of an application that contains the minimum set of required classes and methods for the deployment platform. The output from the smartlinker is stored in a .JXE file which is understood by the IBM j9 Java runtime.

Installer

There are several tools that will package an application ready for installation on one or more platforms. A couple of examples of these are InstallShield and self extracting zip files.

Roll your own distribution mechanism

For instance using a Java class loader that can load classes over a network.

Deployment to devices

Following is a list of some of the techniques and tools that can be used to deploy applications to devices. The list is by no means complete and does not go into any detail but is intended to provide an introduction to some of the ways a Java application can be deployed.

Development tools

Many IDEs (Integrated Development Environments) such as IBM Rational Software Development Platform provide tools that allow deployment of applications onto a device and debugging of the application from the development environment.

OSGi related management

OSGi or Open Services Gateway Initiative defines define a platform for the packaging of and dynamic delivery of Java software services to networked devices. This is achieved via a consistent, component-based, architecture for the development and delivery of Java software components known as bundles and services. Both MQe components and applications can be turned into OSGi bundles and services for use in an OSGi environment. The bundles are delivered from a bundle server. There are several products that provide bundle servers together with the client code to handle the installation and life cycle of bundles. Depending on implementation the bundles can be downloaded on demand, and updated automatically when a new version is available. IBM Rational Software Development Platform ships with SMF (Service Management Framework), which assists in the creation and testing of bundles together with a bundle server.

See more at "Open Services Gateway initiative (OSGi)" on page 14.

JNLP

JNLP or Java Network Launching Protocol and API, is an emerging standard, for use in packaging and deploying Java applications. It is designed to automate the deployment, via the web, for applications written to the J2SE platform.

Device management products

There are several products on the market that can be used for large-scale deployment of software. One example is Tivoli® Configuration Manager from IBM.

C deployment

Supplied DLLs

To deploy applications on the PocketPC 2000, 2002 and 2003 devices, you must copy the MQe DLLs to the device. Copy the DLLs to the Windows® directory, the root directory, or the same directory that holds the application. The following list shows which DLLs you need for different MQe entities:

For the local queuing base

- HMQ_Core.dll
- HMQ_DiskAdapter.dll
- HMQ_HAL.dll
- HMQ_nativeAPI.dll
- HMQ_nativeOSA.dll
- HMQ_RegistryFileSession.dll
- HMQ_LocalQueue.dll

Along with the base DLLs you require the following DLLs depending on how you wish to configure your application:

Remote queuing

HMQ_HttpAdapter.dll

Note: You can remove HMQ_LocalQueue.dll, if you do not want to support administration queues or local queueing.

Synchronous remote queue support

HMQ_SyncRemoteQueue.dll

Asynchronous remote queue support

HMQ_AsyncRemoteQueue.dll

Home server queue support

HMQ_HomeServerQueue.dll

Administration queue support

HMQ_AdminQueue.dll and HMQ_LocalQueue.dll

RLE compressor support

HMQ_RleCompressor.dll

RC4 crytpor support

HMQ_RC4Cryptor.dll

Support for included examples

BrokerDLL.dll

Open Services Gateway initiative (OSGi)

Open Services Gateway initiative (OSGi) is an application framework capable of deploying java applications or services, which can be dynamically employed, updated, or removed. Therefore, it can be a very useful means for providing service updates and ensuring that all the required classes for an application are made available as and when required. MQe provides an example bundle that provides MQe messaging within this framework.

The following topics explain more.

MQe example bundle contents

MQe provides one main bundle for OSGi development and two example application bundles that provide hints on how to create an MQe client or server application within OSGi. No bundle exports or imports a service; they all rely on package dependency. The following table details the bundles and their dependencies.

Table 2. Bundles and dependencies

Bundle name	Description	Export packages	Import packages
MQeBundle.jar	Bundle containing all the required MQe classes excluding mqbridge functionality	com.ibm.mqe com.ibm.mqe.adapters com.ibm.mqe.administration com.ibm.mqe.attributes com.ibm.mqe.communications com.ibm.mqe.messagestore com.ibm.mqe.mqemqmessage com.ibm.mqe.registry com.ibm.mqe.trace	
MQeServerBundle.jar	Example bundle containing an MQe Server application		com.ibm.mqe com.ibm.mqe.adapters com.ibm.mqe.administration com.ibm.mqe.trace org.osgi.framework
MQeClientBundle.jar	Example bundle containing an MQe Client application		com.ibm.mqe com.ibm.mqe.adapters com.ibm.mqe.administration com.ibm.mqe.trace org.osgi.framework

Both example application bundles, MQeClientBundle.jar and MQeServerBundle.jar contain bundle activators which start and stop the application when the framework starts or stops the bundle. The bundles are in MQE_HOME/Java/Jars.

Using MQe within OSGi

When developing your own bundles, importing the correct MQe packages into your bundles manifest file ensures that the MQe bundle is also installed into the framework when your bundle is installed.

One major factor in developing a bundle is that only one MQe queue manager can be run within an OSGi runtime. This means that there may be conflicts if several bundles are installed and each requires its own queue manager. Careful design of the bundle application is required to eliminate this problem. However, there should be no limit on the number of bundles that can use the same queue manager.

Running the example bundles

As an example of how to use MQe within the OSGi environment, we provide two example application bundles that are designed to work together in a simple scenario.

The scenario consists of a client application and a server application:

- The Server simply sits and waits for messages and prints out any that it receives,
- The Client just sends one message.

Within this scenario it is possible to have multiple Clients sending to the same Server or the same Client can be stopped and restarted to send another message to the Server.

These bundles are explained in more detail in the following topics.

Server application (MQeServerBundle.jar)

When this bundle is started, an MQeQueueManager is created and started, with a listener and default queues in memory.

The Application code is then run in a new thread and waits for incoming messages using a message listener; any received messages are displayed in the console. This thread continues to listen until the bundle is stopped, at which time it stops and then deletes the MQeQueueManager.

Client application (MQeClientBundle.jar)

When this bundle is started it checks to see if an MQeQueueManager is already running in the JVM, and if so, it assumes it is running in the same runtime as the server, and so uses that queue manager. If no queue manager is detected then a new one is defined and started in memory and a connection definition and remote queue definition are setup to the server.

Client application code is then run in a new thread which sends a single message to the server. No checks are made to ensure the message is received.

When the bundle is stopped, if a new QueueManager was created for the Client, it is stopped and deleted.

The source for the classes included in the bundles can be seen in the MQe\Java\examples\osgi directory. More details are given in the Java Programming Reference for these classes.

Some points to note when running the applications:

- Each application was written with two parts in mind. The first is setup of the underlying MQe messaging infrastructure, and the second is the main application. This is why each one has a separate class providing function for each part.
- The MQeClientBundle.class and MQeServerBundle.class are both started in their own threads by the bundle activator start method. This way the start method is not delayed in completing as the tasks of sending and receiving messages can take some time. This ensures a smooth transition of the bundles state from resolved to started.

  Note: The Client and Server share the same MQeAdmin class in their bundles. This class could have been placed in its own bundle to avoid the duplication but for simplicities sake we have not done this.
- The Server must always be started before any Clients. Each Server must run in its own runtime. A single client can share the server's runtime or can reside in its own.

Running the example

Whichever way you run the examples, the MQeBundle.jar bundle is required by both the client and server and must be present on the Bundle Server.

To run the example, first start the Server:

1. Import the MQeServerBundle.jar bundle onto the Bundle Server.
2. Start a new SMF (Service Management Framework) runtime, and install and start the MQeServerBundle bundle on it. This should also install the three prerequisite bundles.
3. The server then starts listening, you should see output on the console including:

   ```
   'MQeServerBundle - registering message listener'
   ```

   This means the server is ready for messages.

Next you need to run a client to send a message. There are two methods for runnning the client bundle:

Method 1

In the same SMF runtime as the server:

1. Import the MQeClientBundle.jar bundle onto the Bundle Server.
2. Install and start the MQeClientBundle bundle on the runtime.
3. The client now starts and sends a message, which the server will print on the console. You can stop and start the client bundle to send another message.

Method 2

In separate SMF runtimes:

1. Import the MQeClientBundle.jar bundle onto the Bundle Server.
2. Start a new SMF runtime, and install and start the MQeClientBundle bundle on it. This should also install the three prerequisite bundles.
3. The client starts and sends a message, which the server will print on the console. You can stop and start the client bundle to send another message.

By default the example expects both client and server to be on the same machine running with the receiver listening on port 8085. However, you can change the port and address of the server, that is run the server on a separate machine. Before the server is started, tell it which port to run on by setting the java system property, `examples.osgi.server.port`. This can be set in the Runtime IDE by selecting **Show runtime properties** from the drop down menu.

To tell the client the address and port that the server is listening on, before starting the client set the system properties `examples.osgi.server.address` and `examples.osgi.server.port`.

Note: The server ignores the address property if it is not present. Also, if the client has already been run and you want to change the address and port, the runtime needs to be terminated and restarted to ensure that old MQeConnectionDefinition information is wiped from memory.

Providing user-defined rules and dynamic class loading

The OSGi runtime controls package visibility across bundles. If a bundle does not explicitly import a package, then it will not have access to classes within that package when it comes to dynamically loading them. This is especially important to MQe, because it has been designed with this flexibility in mind. Without some small changes to the bundles, developers cannot use 3rd party or their own Rules or Adapters. There are two ways to remove this problem:

1. OSGi version 3 includes a DynamicImport-Package statement for the bundles manifest file. This has been included in the MQeBundle.jar and when the user-defined class's package is exported from its bundles manifest, MQe will be able to have access to this class.

   Note: This functionality is available to SMF version 3.1.0 or higher.

2. Create a new MQeLoader and add all the user-defined classes before they are used, most likely within the bundles activator, for example:

   ```
   String MyRule = "UserQMRule";
   MQeLoader loader = new MQeLoader();
   loader.addClass(MyRule, Class.forName(MyRule));
   MQe.setLoader(loader);
   ```

   Note: Take care that the loader within MQe is not replaced with another loader from another bundle during the application runtime.

Index

A

D