WebSphere MQ Everyplace V2.0.2

Contenido

Difusión de la aplicación

Empaquetado y difusión

MQe es un sistema de mensajería flexible que puede difundirse a varios sistemas operativos y dispositivos.

En este apartado se proporciona información que le servirá de ayuda para la creación, el empaquetado y la difusión de MQe.

Se divide en dos apartados: la base de código en Java y la base de código nativa.

Puesto que MQe puede difundirse a varios dispositivos, sistemas operativos y módulos de ejecución, no es posible detallar cada aplicación. Por lo tanto, en algunos temas, sólo se proporciona una breve descripción e introducción.

Difusión de Java

La base de código en Java de MQe se puede difundir a muchos módulos de ejecución Java. Éstos son:

- J2ME CLDC/MIDP
- J2ME CDC/Foundation
- PersonalJava V1.1
- Java 1.1
- J2SE 1.2 (o posterior)
- Custom Environment jclGateway de la plataforma de desarrollo de software IBM Rational (o posterior)

El modo en que MQe, la aplicación y otras clases se empaquetan y se difunden depende del tipo de módulo de ejecución de Java, del sistema operativo y del tipo de procesador del dispositivo en el que se está difundiendo.

En los temas siguientes de proporciona información de ayuda para empaquetar y difundir aplicaciones de MQe basadas en Java en distintos entornos.

Archivos jar proporcionados

Cuando se difundan aplicaciones de MQe, se recomienda empaquetar el conjunto de clases mínimo que exija la aplicación en archivos jar comprimidos. Esto garantizará que la aplicación necesite los recursos del sistema mínimos. MQe proporciona los ejemplos siguientes sobre cómo se pueden empaquetar las clases de MQe en archivos .jar. Estos ejemplos se encuentran en el directorio <directorio_instalación_MQe>\Java\Jars de la instalación estándar de MQe.

Existen tres tipos de archivos jar: *base*, de *extensión* y *otros*:

- Los archivos jar base permiten crear, administrar y ejecutar un gestor de colas utilizable.
- Los archivos jar de extensión se pueden utilizar junto con los archivos jar de base para proporcionar más posibilidades.
- Los demás archivos jar son los conjuntos de ejemplo y núcleo de las clases que debe utilizar como base para el desarrollo.

Archivos jar base

MQeBase.jar
: Contiene clases que proporcionan una ejecución de gestor de colas básico en modalidad de cliente y servidor en un módulo de ejecución Java J2ME CDC/Foundation o J2SE o posterior.

MQeMidp.jar
Es parecido a MQeBase.jar, pero se utiliza en un módulo de ejecución Java J2ME CLDC/MIDP. Permite que se ejecute un gestor de colas en modalidad de cliente. En este jar se incluyen todas las clases compatibles con MIDP. No se pueden utilizar jar de extensión con este archivo, porque éstos no son compatibles con MIDP.

MQeGateway.jar
Contiene clases que proporcionan una ejecución de gestor de colas básico en modalidad de cliente o servidor, de servidor y de puente en un módulo de ejecución Java J2SE o posterior.

Archivos jar de extensión

MQeJMS.jar
Contiene las clases que amplían un gestor de colas de MQe para proporcionar una interfaz de programación JMS.

MQeRetail.jar
Contiene clases adicionales para utilizarlas en entornos de minoristas. En concreto, estas clases son útiles en un sistema de minorista 4690.

MQeSecurity.jar
Conjunto de clases que se utilizan para proporcionar seguridad basada tanto en colas como en mensajes. Contiene un conjunto de cifradores, compresores y autentificadores.

MQeBindings.jar
Este archivo contiene toda la información específica de los enlaces C. Es obligatorio si es necesario acceder a un gestor de colas en Java desde una aplicación en C (sólo en plataformas Win32).

MQeMigration.jar
Contiene clases que ayudan en la migración de una versión anterior de MQe.

MQeDeprecated.jar
Este archivo contiene todos los archivos de clases que se han dejado de utilizar y que ya no necesita una aplicación de MQe. Estos archivos de clase que ya no se utilizan pueden ayudar a ejecutar aplicaciones escritas con una versión anterior de MQe, sin realizar ningún cambio.

MQeDiagnostics.jar
Este archivo ayuda a diagnosticar problemas con las clases de MQe. Contiene herramientas para buscar la vía de acceso de las clases con el fin de saber cuál es el nivel de cada una de las clases encontradas.

MQeJMX.jar
Contiene las clases necesarias para administrar MQe mediante JMX.

MQeJMSAdmin.jar
Contiene las clases necesarias para administrar MQe mediante JMS.

MQeJMSSQL.jar
Contiene las clases necesarias para administrar MQe mediante SQL.

Otros archivos jar

MQeExamples.jar
Paquete de todos los ejemplos de MQe en un archivo jar. Incluye todos los ejemplos que se proporcionan con MQe, pero excluye las clases que se han dejado de utilizar.

MQeCore.jar
Contiene un conjunto mínimo de clases. Por sí solo, no se puede utilizar pero sí que se puede utilizar como base para crear un sistema de MQe de pequeño tamaño. Puede encontrar más información detallada sobre cómo reducir el espacio en “Optimización del espacio” en la página 3.

MQeBundle.jar
Este archivo jar proporciona el 'paquete' de MQe para infraestructuras de OSGi y es un paquete sólo de código.

MQeClientBundle.jar
Paquetes de OSGi de ejemplo que muestran cómo ejecutar MQe en infraestructuras de OSGi.

MQeServerBundle.jar
Paquetes de OSGi de ejemplo que muestran cómo ejecutar MQe en la infraestructura de OSGi.

Ejemplos relacionados con OSGi

MQeJMSReceiver.jar
Paquetes de OSGi de ejemplo en los que se utiliza la interfaz JMS.

MQeJMSSender.jar
Paquetes de OSGi de ejemplo en los que se utiliza la interfaz JMS.

Ejemplos relacionados de JMS

MQeJMSAdmin.jar
Proporciona clases para crear y editar objetos administrados almacenados en un espacio de nombres de JNDI.

MQeJMSSQL.jar
Proporciona las clases obligatorias si se tienen que utilizar los selectores de SQL de JMS.

MQeTraceDecode.jar
Proporciona las clases para descodificar el archivo de rastreo de MQe.

Optimización del espacio

En muchos casos los archivos jar que se proporcionan se pueden utilizar sin realizar en ellos ningún cambio, aunque existen instancias en las que esto no es aplicable. En concreto, en algunos entornos, en los que el espacio es limitado, el conjunto de clases que se despliegan debe reducirse al tamaño más pequeño posible. Los archivos jar proporcionados tienen una finalidad general y contienen más de lo necesario para un entorno optimizado.

En la tabla siguiente se dividen las clases en grupos asociados con una función o configuración concreta y la tabla le resultará útil para determinar las clases obligatorias para optimizar el espacio de una aplicación. Mediante esta tabla, el conjunto de clases mínimo necesario se puede deducir tomando las clases obligatorias para las categorías necesarias y, a continuación, añadiendo las clases opcionales para esa categoría.

Debido a la amplia variedad de módulos de ejecución Java que hay ahora disponibles, no todas las clases se pueden ejecutar en todas los módulos de ejecución. En la tabla se enumeran todas las clases y, si no se indica lo contrario, cada clase se ejecutará en un módulo de ejecución J2SR. Debido a las diferencias entre un módulo de ejecución J2SE y J2ME, algunas de estas clases no son adecuadas para un módulo de ejecución J2ME. Hay dos columnas marcadas con una X para mostrar una clase que se puede utilizar en módulos de ejecución J2ME MIDP o J2ME CDC/Foundation.

Tabla 1. Optimización de clases

Categoría			**Detalle**		
	Tipo		**Detalles**	**Midp**	**CDC**
		Clases necesarias (com.ibm.mqe)			
Clases obligatorias					

Tabla 1. Optimización de clases (continuación)

		Para todos los gestores de colas	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`		

Tipo de registro		Debe seleccionarse una opción de esta categoría.		
	Registro de archivos	Añadir necesaria: adaptador de almacenamiento	X	X
		`registry.MQeFileSession` `registry.MQeRegistrySession`		
	Registro privado con o sin credenciales	Añadir: registro de archivos		X
		`registry.MQePrivateRegistry` `registry.MQePrivateSession`		
	Registro privado con credenciales	Añadir: registro privado con o sin credenciales		X

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

		Funciones de gestión de minicertificados		X
		`attributes.MQeListCertificates` `registry.MQePrivateRegistryConfigure`		
	Registro público	Aplicable a tipos de seguridad a nivel de mensajes; añadir: registro privado con credenciales		X
		`registry.MQePublicRegistry`		

Tabla 1. Optimización de clases (continuación)

Tipo de gestor de colas		Para todos los tipos, añadir necesarias: administración, adaptadores de almacenamiento, almacén de mensajes, autentificadores, cifradores, compresores, normas, seguridad		
	Gestor de colas autónomo	Sin clases adicionales		
	Gestor de colas de cliente	Añadir necesaria: comunicaciones	X	X

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

	Gestor de colas de servidor	Añadir: gestor de colas de cliente. Añadir necesaria: comunicaciones		X
	Nota: aunque el escucha de MQe no se utiliza en el cliente, tiene que incluirse al realizar una verificación previa de una aplicación J2ME.			
	Gestor de colas de pasarela	Añadir: gestor de colas de servidor. Añadir necesarias: transformadores de comunicaciones		

	`MQeBridgeLoadable`
	`MQeBridgeManager`
	`mqbridge.*`			

Comunicaciones				
	TCP/IP con o sin historial y persistencia			X
	`adapters.MQeTcpipAdapter` `adapters.MQeTcpipLengthAdapter`			
	TCP/IP con historial y persistencia	Añadir: TCP/IP con o sin historial y persistencia		X
	`adapters.MQeTcpipHistoryAdapter` `adapters.MQeTcpipHistoryAdapterElement`			
	HTTP 1.0 no para servidor de autentificación de proxy WES			X
	`adapters.MQeTcpipAdapter` `adapters.MQeTcpipHttpAdapter`			
	HTTP para servidor de autentificación de proxy WES			X
	`adapters.MQeTcpipAdapter` `adapters.MQeWESAuthenticationAdapter`			
	HTTP 1.1/1.0 J2ME	Sólo MIDP	X	
	`adapters.MQeMidpHttpAdapter`			
	UDP			X

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

Tabla 1. Optimización de clases (continuación)

Tipos de colas		Para todos los tipos de colas, añadir necesarias: autentificadores, cifradores, compresores, normas		
	Local	Añadir: adaptador de almacenamiento, almacenamiento de mensajes	X	X

	`MQeAbstractQueueImplementation`
	`MQeEventTrigger`
	`MQeMessageEvent`
	`MQeMessageListenerInterface`
	`MQeQueue`
	`MQeQueueRule (o un sustituto)`			

	Remoto	Añadir: cola local (adaptador de almacenamiento y almacenamiento de mensajes, sólo si es necesario)	X	X
	`MQeRemoteQueue`			
	Servidor local	Añadir: cola remota (sin adaptador de almacenamiento ni almacenamiento de mensajes)	X	X
	`MQeHomeServerQueue`			
	Almacenar y reenviar	Añadir: cola remota	X	X
	`MQeStoreAndForwardQueue`			
	cola puente	Añadir: cola remota		

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

Almacenamiento de mensajes				
	Base		X	X

	`MQeMessageStoreException`
	`MQeAbstractMessageStore`
	`messagestore.MqeIndexEntry`			

	Estándar	Añadir: base	X	X
	`messagestore.MQeMessageStore`			
	Nombre de archivo abreviado. Utilice siempre nombres de archivo 8.3 para los mensajes.	Añadir: estándar		X
	`messagestore.MQeShortFilenameMessageStore`			
	Específico de 4690	Añadir: nombre de archivo abreviado		
	`messagestore.MQe4690ShortFilenameMessageStore`			
Tipo de mensaje				
	Básico		X	X
	El soporte para MQeMsgObject se encuentra en las clases obligatorias			
	WebSphere MQ			
	`mqemqmessage.*`			
Adaptadores de almacenamiento				
	Disco garantizado	Independencia de grabaciones diferidas de SO		X
	`adapters.MQeDiskFieldsAdapter`			

Tabla 1. Optimización de clases (continuación)

	Disco no garantizado	Dependencia de grabaciones diferidas de SO; añadir: disco garantizado		X
	`adapters.MQeReducedDiskFieldsAdapter`			
	No distingue entre mayúsculas y minúsculas	Añadir: disco garantizado		X
	`adapters.MQeCaseInsensitiveAdapter`			
	Correlación entre nombre de archivo largo y abreviado			X
	`adapters.MQeMappingAdapter`			
	Almacenamiento Midp RMS	Sólo MIDP	X	
	`adapters.MQeMidpFieldsAdapter` `com.ibm.mqe.adapters.MQeMidpFieldsAdapter$RMSFile`			
	Memoria	Almacenamiento volátil	X	X
	`adapters.MQeMemoryFieldsAdapter`			
Administración				

Tabla 1. Optimización de clases (continuación)

	Posibilidad de administración básica	Añadir: cola local	X	X

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

	Gestionar gestor de colas	Añadir: posibilidad de administración básica	X	X
		`administration.MQeQueueManagerAdminMsg`		
	Gestionar definiciones de conexión	Añadir: posibilidad de administración básica	X	X
		`administration.MQeConnectionAdminMsg`		
	Gestionar escuchas de comunicaciones	Añadir: posibilidad de administración básica	X	X
		`administration.MQeCommunicationsListenerAdminMsg`		
	Gestionar cola local	Añadir: posibilidad de administración básica	X	X
		`administration.MQeQueueAdminMsg`		
	Gestionar cola de administración	Añadir: gestionar cola local	X	X
		`administration.MQeAdminQueueAdminMsg`		
	Gestionar cola remota	Añadir: gestionar cola local	X	X
		`administration.MQeRemoteQueueAdminMsg`		
	Gestionar cola de servidor local	Añadir: gestionar cola remota	X	X
		`administration.MQeHomeServerQueueAdminMsg`		
	Gestionar cola para almacenar y reenviar	Añadir: gestionar cola remota	X	X
		`administration.MQeStoreAndForwardQueueAdminMsg`		
	Gestionar cola puente	Añadir: gestionar cola remota		X
		`mqbridge.MQeMQBridgeQueueAdminMsg` `mqbridge.MQeCharacteristicLabels`		
	WebSphere MQ	Añadir: colas remotas		

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

Creación y supresión de gestores de colas		`MQeQueueManagerConfigure`	X	X
Autentificadores				
	Minicertificado			X

		`attributes.DHk (el origen se puede generar)`
		`attributes.MQeSharedKey`
		`attributes.MQeRandom`
		`attributes.MQeWTLSCertificate`
		`attributes.MQeWTLSCertAuthenticator`		

Compresores				

Tabla 1. Optimización de clases (continuación)

	GZIP	`attributes.MQeGZIPCompressor`		X
	LZW	`attributes.MQeLZWCompressor` `attributes.MQeLZWDictionaryItem`	X	X
	RLE	`attributes.MQeRleCompressor`	X	X
Cifradores				
	DES triple	`attributes.MQe3DESCryptor`		X
	DES	`attributes.MQe3DESCryptor`		X
	MARS	`attributes.MQeDESCryptor`		X
	RC4	`attributes.MQeRC4Cryptor`		X
	RC6	`attributes.MQeRC6Cryptor`		X
	XOR	`attributes.MQeXorCryptor`	X	X
Servicios de seguridad de aplicaciones				
	Seguridad local	Añadir necesaria: cifradores	X	X
	`attributes.MQeLocalSecure`			
	Seguridad de nivel de mensajes	Añadir necesaria: cifradores		X
	`attributes.MQeMAttribute`			
	Seguridad a nivel de mensajes con firma digital y validación	Añadir: registro público. Añadir necesaria: cifradores		X
	`attributes.MQeMTrustAttribute`			
Rastreo				

Tabla 1. Optimización de clases (continuación)

	Recopilar rastreo binario en J2SE/CDC				X
		`trace.MQeTraceToBinary` `trace.MQeTraceToBinaryFile`			
	Recopilar rastreo binario en almacén Midp RMS y/o enviar al servlet de rastreo de MIDP			X	
		`trace.MQeTraceToBinary` `trace.MQeTraceToBinaryMidp`			
	Representador de rastreo base				X

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

	Descodificar un archivo binario en un formato legible	Añadir: representador de rastreo base			X
		`trace.MQeTraceToReadable` `trace.MQeTraceFromBinaryFile`			
	Rastreo en una corriente de salida legible	Añadir: representador de rastreo base			X
		`trace.MqeTraceToReadable`			
	Recopilación de servlet de rastreo binario de Midp	Añadir: representador de rastreo base			
		`trace.MQeTraceToReadable` `examples.trace.MQeServlet`			
Varios					
	Soporte criptográfico	Uso sólo para aplicación o instalación			X
		`attributes.MQeCL (¿nota al pie?)` `attributes.MQeGenDH (genera una versión de attributes.MQeDHk.java)`			
	MQeServerSupport SupportPac ES06	MQeServerSupport (consulte las instrucciones de instalación de ES06)			
Enlaces		Acceso a las clases Java desde otros lenguajes			
	Lenguaje C	`bindings.*`			
JMS		Soporte para la API de Java Message Service			XX
		`jms.*` `transaction.*`			
Extensiones de MQe definidas por el usuario					
			Autentificadores, adaptadores de comunicaciones, compresores, cifradores, clases de registro, clases de mensajes, clases de normas, control de seguridad, adaptadores de almacenamiento, manejador de rastreo		

Requisitos de JMS

Para poder utilizar la interfaz de programación JMS de MQe son obligatorias las clases de la interfaz JMS.

Suelen estar en jms.jar.

MQe no se entrega con jms.jar: lo deberá descargar para poder utilizar JMS.

En el momento en el que se ha escrito esta documentación, se puede descargar de forma gratuita de http://java.sun.com/products/jms/docs.html. Es necesario el archivo jar de JMS Versión 1.0.2b.

JNDI

Además, si los objetos administrados de JMS se tienen que almacenar y recuperar con JNDI (Java Naming and Directory Interface), las clases javax.naming.* deben estar disponibles.

Si se va a utilizar Java 1 (por ejemplo el JRE de la versión 1.1.8), debe obtenerse el archivo jndi.jar y añadirlo a la variable CLASSPATH.

Si se va a utilizar Java 2 (por ejemplo, un JRE de la versión 1.2 o posterior), el JRE puede contener estas clases.

Puede utilizar MQe sin JNDI, pero ello supondrá una pequeña dependencia del proveedor. Las clases específicas de MQ deben utilizarse para objetos ConnectionFactory y Destination.

Puede descargar los archivos jar de JNDI de http://java.sun.com/products/jndi

Clases de MQe para requisitos Java

Para utilizar el puente de MQ son obligatorias las *clases de MQ para Java* de la versión 5.1 o posterior.

Estas clases se empaquetan con MQ 5.3 y posteriores. Si se utiliza una versión anterior de MQ, se pueden descargar de forma gratuita de Internet como SupportPac MA88.

Para obtener un ejemplo sobre cómo configurar la variable CLASSPATH para incluir los archivos jar de MQ, consulte los archivos de proceso por lotes:

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

Ocasionalmente, los archivos jar cambian entre las versiones de MQ: si se detectan problemas como consecuencia de ello, consulte la documentación sobre las clases de MQ para poder determinar los archivos jar correctos que se deben utilizar.

Utilización del enlazador inteligente de Rational Device Developer

La herramienta de enlazador inteligente que se entrega con la plataforma de desarrollo de software Rational se utiliza en el proceso de creación y empaquetado de una aplicación en un archivo jar o jxe. El enlazador inteligente puede eliminar las clases (y los métodos) que se considere que no son necesarios; esto puede hacer que se eliminen las clases que son necesarias pero que se cargan de forma dinámica. MQe utiliza la carga dinámica, por lo que se debe tener cuidado de evitar esta función o se deben asignar nombres de forma explícita a las clases que deban estar presentes, aunque no se referencien de forma explícita en el código.

Para evitar que las clases que no se utilicen no se eliminen, utilice la opción `-noRemoveUnused`.

De lo contrario, si se establece la opción `-removeUnused`, cualquier clase que se cargue de forma dinámica debe incluirse de forma específica. Una opción que se puede utilizar para conseguir esto es `-includeWholeClass`.

Por ejemplo,

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

incluirá todas las clases en el paquete de adaptadores y

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

sólo incluirá el adaptador de http.

Se pueden especificar varias opciones de inclusión (o exclusión) en el archivo de opciones de enlazador inteligente:

Puede utilizar las directrices siguientes para determinar las clases que se cargan de forma dinámica. La directriz básica es que cualquier clase que se referencia a través de un alias de clase de MQe o cualquier clase que se establezca como parámetro al administrar los recursos de MQe se cargará de forma dinámica. Esto incluye:

- Adaptadores de comunicaciones
- Adaptadores de almacenamiento
- Almacenes de mensajes
- Normas
- Alias
- Cifrador
- Compresores
- Autentificadores
- Colas
- Transportador
- Conexión (consulte el ejemplo siguiente)

Un ejemplo de conjunto de inclusiones necesario para una aplicación de MIDP sencilla es:

```
-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"
```

Características específicas de Midp de J2ME

Al difundir la aplicación en Java para el entorno Midp, es necesario tener en cuenta unos cuantos aspectos.

- Debe utilizar los adaptadores de almacenamiento y de comunicaciones específicos de Midp (consulte "Utilización del enlazador inteligente de Rational Device Developer" en la página 11) y excluir las clases que no sean compatibles con Midp.
- Puede utilizar el archivo MQeMidp.jar preempaquetado o una versión propia reducida, aunque también se debe incluir un archivo JAD (descriptor de aplicación en Java) que detalle los midlets disponibles en la aplicación. Al realizar la difusión al dispositivo, todas las clases deben empaquetarse y verificarse previamente en un archivo jar antes de la difusión. No obstante, al realizar pruebas con un emulador, se pueden utilizar varios jar incluyéndolos en la variable CLASSPATH.
- Debe asegurarse de que todas las clases obligatorias se incluyan en el archivo jar/prc u otro ejecutable. Algunas clases se cargan de forma dinámica y puede que no estén presentes cuando se utilice algún enlazador inteligente. Consulte el apartado "Utilización del enlazador inteligente de Rational Device Developer" en la página 11 para obtener más detalles.

Características específicas de 4690

Tenga en cuenta los siguientes requisitos cuando configure MQe para utilizarlo con 4690.

- Las aplicaciones de terminal tienen una restricción de longitud de vía de acceso máxima de 24 caracteres, pero las aplicaciones de controlador de almacén pueden tener 127 caracteres. Las aplicaciones en Java también tienen una restricción de una longitud de 24 caracteres.
- El sistema de archivos virtual (VFS) no puede tener más de 64.000 archivos. Puesto que se utilizan tamaños de disco en GB, puede que la unidad C: no tenga un límite en el número de archivos, en función del sistema operativo.
- Cuando desee acceder a un archivo, debe especificar su vía de acceso. La vía de acceso consta de nombres de directorio que se separan con un carácter de barra invertida "\" o un carácter de barra inclinada "/".

  Nota: aunque el sistema acepte tanto el carácter "\" como el carácter "/", probablemente es menos confuso utilizar uno u otro.
- Los demás ejemplos de este manual muestran cómo configurar el gestor de colas de forma que los datos que describen sus recursos, certificados y otros datos de configuración se almacenen en archivos con nombres largos. Estos nombres de archivo son para un solo directorio de nivel superior, que también se puede encontrar en el espacio de nombres de unidad del VFS.
- Con el formato 8.3, la longitud de caracteres total del nombre de archivo completo supera los límites permitidos impuestos por un sistema de archivos nativo de 4960. Por lo tanto, en el VFS:
  - La longitud máxima de un nombre de archivo es de 256 caracteres.
  - La longitud máxima de una vía de acceso, incluidos los directorios y archivos, es de 260 caracteres.
  - La profundidad máxima de directorios es de 60 niveles, incluido el directorio raíz.
- Las clases de MQe se pueden almacenar con nombres de formato largo en el VFS. Sin embargo, por motivos de rendimiento y comodidad, puesto que existen muchos archivos de clases, recomendamos que la aplicación y las clases de MQe se empaqueten en archivos .jar y se difundan.
- Según el manual del VFS, el sistema operativo proporciona soporte para los nombres de archivo que superen los ocho caracteres de longitud a través del uso de un sistema de archivos virtual (VFS) de 4690.
- El manual del VFS indica que se debe habilitar la configuración de la unidad de VFS a través de la configuración del sistema. Al habilitar los valores de la unidad VFS, el sistema operativo crea dos unidades lógicas: la C: y la D:. La unidad determina dónde se encuentra el directorio del VFS. No obstante, la información se almacena de hecho en las unidades C: y D:. La información de la unidad M: se almacena en la unidad C: y la información de la unidad N: se almacena en la unidad D:. Después de habilitar VFS, puede utilizar las unidades M: y N: para proporcionar localmente soporte de nombres de archivos largos.
- Se recomienda utilizar `MQeCaseInsensitiveDiskAdapter` en el SO 4690. Esta clase implementa un adaptador de disco que no distingue entre las mayúsculas y minúsculas del nombre del archivo que se utilice durante la búsqueda de coincidencias. Algunas JVM o combinaciones de SO enumeran archivos con combinaciones de mayúsculas y minúsculas distintas a las combinaciones con las que se han creado. Esto significa que el filtro sencillo de la superclase las ignora. De todos modos, esta clase convierte tanto el comparador como el comparando en minúsculas antes de realizar la comparación. Esto garantiza la mejor probabilidad de encontrar una coincidencia válida. Tenga en cuenta que la conversión a minúsculas puede que sea inadecuada en plataformas en las que no se distingue entre las mayúsculas y las minúsculas y donde hay archivos no MQe almacenados que pueden dar lugar a confusiones debido a las mayúsculas y las minúsculas. En resumen, este adaptador está pensado para utilizarlo con el sistema de archivos 4690.

Empaquetado

A continuación se detalla una lista de algunas de las técnicas y herramientas que se pueden utilizar para empaquetar aplicaciones preparadas para la difusión en un dispositivo. Esta lista no es una lista completa y no proporciona información detallada, pero está pensada para proporcionar una introducción a algunas de las maneras como se puede empaquetar una aplicación en Java.

Archivo Jar único

Cree una aplicación autocontenida con MQe incorporado en ella. Esta opción minimiza el espacio y garantiza un mínimo para la variable CLASSPATH.

Varios archivos Jar

Coloque la aplicación en un archivo jar y, a continuación, utilice los archivos jar de MQe proporcionados o construya un archivo jar de MQe independiente. Disponer de MQe en uno o más archivos jar independientes facilita el uso de MQe desde varias aplicaciones independientes.

JNLP

JNLP (Java Network Launching Protocol y API) es un estándar emergente para utilizarlo en el empaquetado y la difusión de aplicaciones en Java. Está diseñado para automatizar la difusión, a través de Internet, para aplicaciones escritas para la plataforma J2SE.

OSGi

OSGi u Open Services Gateway Initiative define una plataforma para el empaquetado y la entrega dinámica de servicios de software en Java a dispositivos en red. Esto se consigue a través de una arquitectura coherente basada en componentes para el desarrollo y la entrega de componentes de software en Java conocidos como paquetes y servicios. Tanto los componentes como las aplicaciones de MQe se tienen que convertir en paquetes y servicios de OSGi para utilizarlos en un entorno OSGi. Los paquetes se entregan desde un servidor de paquetes. Existen varios productos que proporcionan servidores de paquetes junto con el código de cliente para manejar la instalación y el ciclo de vida de los paquetes. Según la implementación, los paquetes se pueden descargar a petición y se pueden actualizar de forma automática cuando hay una nueva versión disponible. La plataforma de desarrollo de software IBM Rational se entrega con SMF (Service Management Framework), que le ayudará a crear y probar paquetes junto con un servidor de paquetes.

Puede obtener más información al respecto en “Open Services Gateway initiative (OSGi)” en la página 16.

Midlet

Una aplicación J2ME MIDP de MQe debe empaquetarse como midlet o conjunto de midlets (.jad y .jar).

JXE

IBM Rational Device Developer dispone de una herramienta de enlazador inteligente que puede generar un empaquetado optimizado de una aplicación que contenga el conjunto mínimo de clases y métodos obligatorios para la plataforma de desarrollo. La salida del enlazador inteligente se almacena en un archivo .JXE, que el módulo de ejecución IBM j9 Java interpreta.

Instalador

Varias herramientas empaquetarán una aplicación preparada para su instalación en una o más plataformas. Un par de ejemplos de estas herramientas son archivos de InstallShield y autoextraíbles.

Aplicar un mecanismo de distribución propio

Por ejemplo, mediante un cargador de clases Java que pueda cargar clases en una red.

Difusión a dispositivos

A continuación se detalla una lista de algunas de las técnicas y herramientas que se pueden utilizar para difundir aplicaciones en dispositivos. La lista no es de ningún modo completa y no proporciona información detallada, pero está pensada para proporcionar una introducción a algunas de las manera como se puede difundir una aplicación en Java.

Herramientas de desarrollo

Muchos IDE (Integrated Development Environments), como la plataforma de desarrollo de software IBM Rational, proporcionan herramientas que permiten la difusión de aplicaciones en un dispositivo y la depuración de la aplicación desde el entorno de desarrollo.

Gestión relacionada con OSGi

OSGi u Open Services Gateway Initiative define una plataforma para el empaquetado y la entrega dinámica de servicios de software en Java a dispositivos en red. Esto se consigue a través de una arquitectura coherente basada en componentes para el desarrollo y la entrega de componentes de software en Java conocidos como paquetes y servicios. Tanto los componentes como las aplicaciones de MQe se tienen que convertir en paquetes y servicios de OSGi para utilizarlos en un entorno OSGi. Los paquetes se entregan desde un servidor de paquetes. Existen varios productos que proporcionan servidores de paquetes junto con el código de cliente para manejar la instalación y el ciclo de vida de los paquetes. Según la implementación, los paquetes se pueden descargar a petición y se pueden actualizar de forma automática cuando hay una nueva versión disponible. La plataforma de desarrollo de software IBM Rational se entrega con SMF (Service Management Framework), que le ayudará a crear y probar paquetes junto con un servidor de paquetes.

Puede obtener más información al respecto en "Open Services Gateway initiative (OSGi)" en la página 16.

JNLP

JNLP, o Java Network Launching Protocol y API, es un estándar emergente para utilizarlo en el empaquetado y la difusión de aplicaciones en Java. Está diseñado para automatizar la difusión, a través de Internet, para aplicaciones escritas para la plataforma J2SE.

Productos de gestión de dispositivos

Existen varios productos en el mercado que se pueden utilizar para la difusión de software a gran escala. Un ejemplo es el Tivoli Configuration Manager de IBM.

Difusión en C

DLL proporcionadas

Para difundir aplicaciones en dispositivos PocketPC 2000, 2002 y 2003, debe copiar las DLL de MQe en el dispositivo. Copie las DLL en el directorio raíz de Windows o en el mismo directorio en el que está instalada la aplicación. En la lista siguiente se muestran las DLL que necesitará para distintas entidades de MQe:

Para la base de gestión de colas local

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

Las siguientes DLL son necesarias, junto con las DLL básicas, en función de cómo desee configurar la aplicación:

Gestión de colas remota

HMQ_HttpAdapter.dll

Nota: Se puede eliminar la biblioteca HMQ_LocalQueue.dll si no desea dar soporte a la gestión de colas local o a las colas de administración.

Soporte a las colas remotas síncronas

HMQ_SyncRemoteQueue.dll

Soporte a las colas remotas asíncronas

HMQ_AsyncRemoteQueue.dll

Soporte a las colas del servidor local

HMQ_HomeServerQueue.dll

Soporte a las colas de administración

HMQ_AdminQueue.dll y HMQ_LocalQueue.dll

Soporte a los compresores de RLE

HMQ_RleCompressor.dll

Soporte a los cifradores RC4

HMQ_RC4Cryptor.dll

Soporte a los ejemplos incluidos

BrokerDLL.dll

Open Services Gateway initiative (OSGi)

Open Services Gateway initiative (OSGi) es una infraestructura de aplicaciones que puede difundir aplicaciones o servicios en Java, que, a su vez, se pueden utilizar, actualizar o eliminar de forma dinámica. Por lo tanto, puede resultar un medio muy útil para proporcionar actualizaciones de servicios y para garantizar que todas las clases obligatorias para una aplicación estén disponibles cuando sea necesario. MQe proporciona un paquete de ejemplo que ofrece la mensajería de MQe dentro de esta infraestructura.

En los temas siguientes se ofrecen más detalles al respecto.

Contenido del paquete de ejemplo de MQe

MQe proporciona un paquete principal para el desarrollo de OSGi y dos paquetes de aplicaciones de ejemplo que proporcionan sugerencias sobre cómo crear un cliente de MQe o una aplicación de servidor dentro de OSGi. Ningún paquete exporta ni importa un servicio: todos se basan en dependencias de paquete. En la tabla siguiente se detallan los paquetes y sus dependencias.

Tabla 2. Paquetes y dependencias

Nombre del paquete	Descripción	Paquetes de exportación	Paquetes de importación
MQeBundle.jar	Paquete que contiene todas las clases de MQe obligatorias, excluidas las funciones del puente de MQ	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	Paquete de ejemplo que contiene una aplicación de servidor de MQe		com.ibm.mqe com.ibm.mqe.adapters com.ibm.mqe.administration com.ibm.mqe.trace org.osgi.framework

Tabla 2. Paquetes y dependencias (continuación)

MQeClientBundle.jar	Paquete de ejemplo que contiene una aplicación de cliente de MQe		com.ibm.mqe com.ibm.mqe.adapters com.ibm.mqe.administration com.ibm.mqe.trace org.osgi.framework

Los dos paquetes de aplicaciones de ejemplo, MQeClientBundle.jar y MQeServerBundle.jar, contienen activadores de paquetes que inician y detienen la aplicación cuando la infraestructura inicia o detiene el paquete. Los paquetes se encuentran en INICIO_MQE/Java/Jars.

Utilización de MQe en OSGi

Cuando se desarrollan paquetes propios, la importación de los paquetes de MQe correctos en el archivo de manifiesto de los paquetes garantiza que el paquete de MQe también se instale en la infraestructura al instalar el paquete del usuario.

Un factor principal en el desarrollo de un paquete es que sólo se puede ejecutar un gestor de colas de MQe en un módulo de ejecución de OSGi. Esto significa que pueden generarse conflictos si se instalan varios paquetes y cada uno exige su propio gestor de colas. Para evitar este problema es obligatorio diseñar con precisión la aplicación de paquetes. No obstante, no debería haber ningún limite en el número de paquetes que pueden utilizar el mismo gestor de colas.

Ejecución de los paquetes de ejemplo

Como ejemplo de cómo utilizar MQe en el entorno de OSGi, proporcionamos dos paquetes de aplicaciones de ejemplo que se han diseñado para que funcionen conjuntamente en un caso de ejemplo sencillo.

El caso de ejemplo consiste en una aplicación de cliente y una aplicación de servidor:

- El servidor simplemente espera a que lleguen los mensajes y muestra los que recibe.
- El cliente sólo envía un mensaje.

En esta situación, es posible que varios clientes envíen mensajes al mismo servidor o se puede detener y reiniciar el mismo cliente para enviar otro mensaje al servidor.

Estos paquetes se explican más detalladamente en los temas siguientes.

Aplicación de servidor (MQeServerBundle.jar)

Cuando se inicia este paquete, se crea y se inicia un gestor de colas de MQe, con un escucha y las colas por omisión en la memoria.

El código de aplicación se ejecuta, a continuación, en una nueva hebra y espera a que lleguen mensajes utilizando un escucha de mensajes; los mensajes que se reciben se muestran en la consola. Esta hebra sigue estando a la escucha hasta que el paquete se detiene, momento en el que ésta se detiene y suprime el gestor de colas de MQe.

Aplicación de cliente (MQeClientBundle.jar)

Al iniciar este paquete, éste comprueba si ya hay un gestor de colas de MQe en ejecución en la JVM y, si lo hay, presupone que se está ejecutando en el mismo módulo de ejecución que el servidor y, por lo tanto, utiliza ese gestor de colas. Si no se detecta ningún gestor de colas, se define uno nuevo que se inicia en la memoria y se configuran una definición de conexión y una definición de cola remota en el servidor.

El código de aplicación de cliente se ejecuta, a continuación, en una hebra nueva que envía un solo mensaje al servidor. No se efectúan comprobaciones para garantizar que el mensaje se recibe.

Cuando se detiene el paquete, si se ha creado un nuevo gestor de colas para el cliente, éste se detiene y se suprime.

El origen de las clases que se incluye en los paquetes se puede consultar en el directorio MQe\Java\examples\osgi. Se proporcionan más detalles sobre estas clases en la consulta de programación en Java.

Algunos puntos que se deben tener en cuenta al ejecutar las aplicaciones:

- Cada aplicación se ha escrito teniendo en cuenta dos partes. La primera es la configuración de la infraestructura de mensajería de MQe subyacente y la segunda es la aplicación principal. Por ello, cada una tiene una clase independiente que proporciona funciones para cada parte.
- Tanto MQeClientBundle.class como MQeServerBundle.class se inician en sus propias hebras mediante el método de inicio del activador de paquetes. De este modo, el método de inicio no se retrasa para finalizar, aunque las tareas de envío y recepción de mensajes pueden tardar algo de tiempo. Esto garantiza una transición sin impacto del estado de los paquetes de resuelto a iniciado.

  Nota: el cliente y el servidor comparten la misma clase MQeAdmin en sus paquetes. Esta clase podría haberse colocado en su propio paquete para evitar la duplicación, pero no lo hemos hecho para simplificar el proceso.
- El servidor debe iniciarse siempre antes que los clientes. Cada servidor debe ejecutarse en su propio módulo de ejecución. Un solo cliente puede compartir el módulo de ejecución del servidor o puede residir en una propio.

Ejecución del ejemplo

Independientemente del modo en que ejecute los ejemplos, tanto el cliente como el servidor exigen el paquete MQeBundle.jar y deben estar presentes en el servidor de paquetes.

Para ejecutar el ejemplo, inicie primero el servidor:

1. Importe el paquete MQeServerBundle.jar en el servidor de paquetes.
2. Inicie un nuevo módulo de ejecución de SMF (Service Management Framework) e instale e inicie en él el paquete MQeServerBundle. Con esta acción, también deben instalarse tres paquetes de requisitos previos.
3. El servidor, a continuación, empieza la escucha y en la consola se deberá visualizar la salida:

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

   Esto significa que el servidor está preparado para recibir mensajes.

A continuación, tiene que ejecutar un cliente para enviar un mensaje. Existen dos métodos para ejecutar el paquete de cliente:

Método 1

En el mismo módulo de ejecución de SMF que el servidor:

1. Importe el paquete MQeClientBundle.jar en el servidor de paquetes.
2. Instale e inicie el paquete MQeClientBundle en el módulo de ejecución.
3. El cliente ahora se inicia y envía un mensaje, que el servidor mostrará en la consola. Puede detener e iniciar el paquete de cliente para enviar otro mensaje.

Método 2

En módulos de ejecución de SMF independientes:

1. Importe el paquete MQeClientBundle.jar en el servidor de paquetes.
2. Inicie un módulo de ejecución de SMF nuevo e instale e inicie en él el paquete MQeClientBundle. Con esta acción, también deben instalarse tres paquetes de requisitos previos.

3. El cliente se inicia y envía un mensaje, que el servidor mostrará en la consola. Puede detener e iniciar el paquete de cliente para enviar otro mensaje.

Por omisión, en el ejemplo se espera que tanto el cliente como el servidor estén ejecutándose en la misma máquina con el receptor a la escucha en el puerto 8085. No obstante, puede cambiar el puerto y la dirección del servidor, es decir, ejecute el servidor en otra máquina. Antes de que se inicie el servidor, indíquele el puerto que se debe ejecutar estableciendo la propiedad del sistema java `examples.osgi.server.port`. Esto se puede establecer en el IDE del módulo de ejecución seleccionando **Mostrar las propiedades del módulo de ejecución** en el menú desplegable.

Para indicar al cliente la dirección y el puerto en el que el servidor está a la escucha, antes de iniciar el cliente, establezca las propiedades del sistema `examples.osgi.server.address` y`examples.osgi.server.port`.

Nota: el servidor ignora la propiedad de dirección, si no está presente. Además, si el cliente ya se ha ejecutado y desea cambiar la dirección y el puerto, el módulo de ejecución tiene que interrumpirse y volverse a iniciar para garantizar que se ha limpiado de la memoria la información de MQeConnectionDefinition antigua.

Cómo proporcionar normas definidas por el usuario y la carga dinámica de clases

El módulo de ejecución de OSGi controla la visibilidad de los paquetes inferiores entre distintos paquetes. Si un paquete no importa de forma explícita otro paquete inferior, no tendrá acceso a las clases de ese paquete inferior cuando se deban cargar de forma dinámica. Esto es especialmente importante en MQe, porque se ha diseñado teniendo en cuenta esta flexibilidad. Sin algunos pequeños cambios en los paquetes, los desarrolladores no pueden utilizar normas o adaptadores propios o de otros proveedores. Existen dos formas de eliminar este problema:

1. OSGi versión incluye una sentencia DynamicImport-Package para el archivo de manifiesto de los paquetes. Se ha incluido en MQeBundle.jar y cuando el paquete de la clase definida por el usuario se exporte de su manifiesto de paquetes, MQe podrá tener acceso a esta clase.

   Nota: estas funciones están disponibles para la versión de SMF 3.1.0 o posterior.
2. Cree un nuevo MQeLoader y añada todas las clases definidas por el usuario antes de que se utilicen, probablemente en el activador de paquetes, por ejemplo:

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

   Nota: tenga cuidado de que el cargador en MQe no se sustituya por otro cargador de otro paquete durante la ejecución de la aplicación.

Índice