

**COMPORSYS Connector for IBM MQSeries /
WebSphere MQ**

Version 1.5

Installation and Reference Guide

Contents

Preface..... 5

Installation 6

 Requirements 6

 Contents of MQ_CONNECTOR.ZIP 7

 J2EE 1.3 compliant Application Server..... 8

 Installation steps..... 8

Deployment and Configuration..... 10

 J2EE 1.3 compliant Application Server..... 10

 Configuring the connector for Point-to-Point model 10

 Configuring the connector for Publish-Subscribe model 10

 Configuration Parameters 11

 ConnectionFactory 11

 Connection 13

 Configuring the connector for other JMS providers 15

 Configuring for Point-2-Point..... 15

 Configuring for Publish-Subscribe 15

 Configuring Transaction support..... 16

COMPORSYS Code Generation Tool..... 17

 Processing COBOL Copybooks 17

 Naming Conventions 18

 Data type mapping 18

 Generating access classes 18

 Point-to-Point 19

 Publish-Subscribe 20

 Remarks 20

 Design of access classes..... 21

 Programming Pattern..... 22

 How to transmit a text..... 22

 How to transmit a record class 22

 How to transmit a byte array 23

Programming with the UCI..... 24

Contents

Setting up the connector for UCI	24
Establishing a connection	24
Managed Environment.....	24
Unmanaged Environment	25
Transaction Management	26
Managed Environment.....	27
Unmanaged Environment	28
Accessing the UCI.....	29
Point-to-Point Communication	29
Publish-Subscribe Communication	29
Properties of the Request class.....	30
How to use InteractionMode.....	32
Setting JMS properties	32
Programming with the CCI	33
Setting up the connector for CCI	33
Establishing a connection	33
Managed Environment.....	33
Unmanaged Environment	33
Transaction Management	34
Managed Environment.....	34
Unmanaged Environment	35
Point-to-Point Communication	35
Publish-Subscribe Communication	36
Properties of the InteractionSpec bean	37
How to use InteractionVerbs	39
Setting JMS properties	39
Retrieving JMS properties	40
Accessing the underlying JMSMessage	40
Support for MQ/IMS Bridge	41
Further information	43

List Of Tables

Table 1: Directory Structure 7

Table 2: Required Java Archives..... 8

Table 3: Transaction support matrix 16

Table 4: ImportCobol Parameters..... 17

Table 5: Data type mapping between COBOL and Java 18

Table 6: GenMQP2P Parameters 19

Table 7: GenMQPubSub Parameters..... 20

Table 8: Influence of AutoCommit on Transaction Management 27

Table 9: Properties of the Request class 31

Table 10: Properties of the InteractionSpec bean..... 38

Preface

The COMPORSYS Connector for IBM MQSeries enables fast and easy integration of all transactions and programs that can be accessed through IBM MQSeries / WebSphere MQ. It can be used in a Client/Server Environment, for example in a Java application (*Unmanaged Environment*), as well as in an Application Server of the Java 2 Enterprise Edition (J2EE)[™] (*Managed Environment*).

The COMPORSYS Connector includes

- a resource adapter for IBM MQSeries / WebSphere MQ according to the J2EE[™] Connector Architecture Specification 1.0.
- the code generator COMPORSYS Code Generation Tool
 - to create Java records from COBOL Copybooks for data exchange with IBM MQSeries on mainframes. These *records* allow an easy access to mainframe programs and carry-through the necessary mapping of data types.
 - to create access records for the connector, often making your own programming superfluous.

The COMPORSYS Connector supports the standard Common Client Interface (CCI) of the Java Connector Architecture as well as COMPORSYS User Call Interface (UCI), that allows an effortless access to all features of the Connector:

- Point-to-Point

Use the Connector to for synchronous queue-based communication.

- Publish/Subscribe

Use the Connector for asynchronous topic-based communication.

- Events

Due to the lack of support in JCA 1.0 you will need the INSEVO Business Integration Framework (<http://www.insevo.com>) to enable asynchronous communication.

Installation

Requirements

- IBM MQSeries installed and configured.
(<http://www.ibm.com/software/ts/mqseries>)
- IBM MQSeries Extension MA88 installed.
(<http://www.ibm.com/software/ts/mqseries/txappc/ma88.html>)
- Application Server with support for Java Connector Architecture.
We recommend BEA WebLogic Server
(<http://www.bea.com>)
- ANT from Apache to configure the resource adapter and to build the examples.
(<http://jakarta.apache.org/ant/index.html>)

Contents of MQ_CONNECTOR.ZIP

The MQ_CONNECTOR.ZIP contains all files needed for the COMPORSYS Connector for IBM MQSeries.

Included are:

- Resource adapter MQ_CONNECTOR.RAR
- Required libraries
- Documentation and examples

mq_connector\	COMPORSYS Connector root directory
mq_connector\deploy	Resource Adapter MQ_CONNECTOR.RAR
mq_connector\lib	All required java archives
mq_connector\META-INF	Deployment Descriptors
mq_connector\doc	Documentation
mq_connector\templates	Templates for various configuration files
mq_connector\examples	Application sample files

Table 1: Directory Structure

J2EE 1.3 compliant Application Server

The COMPORSYS Connector for IBM MQSeries contains a JCA compliant resource adapter. Therefore it can be easily deployed into the BEA WebLogic Server 6.1 or later and other Application Servers supporting the Java Connector Architecture.

Installation steps

Follow these steps to install the COMPORSYS Connector for IBM MQSeries.

1. Copy the COMPORSYS Connector installation files

Unzip the provided MQ_CONNECTOR.ZIP into the root directory of the BEA WebLogic Server (i.e. c:\bea\weblogic).

2. Editing the Server CLASSPATH

Add the MQ_CONNECTOR.JAR to the server CLASSPATH.

MQ_CONNECTOR.JAR	COMPORSYS Connector for IBM MQSeries
CONNECTOR_FOUNDATION.JAR	COMPORSYS Connector Foundation Classes
com.ibm.mq.jar	IBM MQSeries Extension MA88 for JMS, located in %MQCLIENT%\java\lib
com.ibm.mqjms.jar	IBM MQSeries Extension MA88 for JMS, located in %MQCLIENT%\java\lib
%MQCLIENT%\java\lib	Resources for IBM MQSeries Extension

Table 2: Required Java Archives

3. Installing the License file

Applies only if you are using Insevo Business Integration Framework. Copy your license file into a directory named "license" (i.e. ./mq_connector/license) and define a JVM system property "insevo.adapter.root" that points to the parent directory. (i.e. -Dinsevo.adapter.root=./mq_connector)

Example:

```
startWebLogicServer.cmd:
...
set CONNECTOR=.\mq_connector\lib\mq_connector.jar;
    .\mq_connector\lib\connector_foundation.jar;
    .\mqclient\java\lib\com.ibm.mq.jar;
    .\mqclient\java\lib\com.ibm.mqjms.jar;
    .\mqclient\java\lib

set CLASSPATH=.;.\lib\weblogic_sp.jar;.\lib\weblogic.jar;%CONNECTOR%

REM now start the server

"%JAVA_HOME%\bin\java" ... -classpath %CLASSPATH%
                        -Dinsevo.adapter.root=./mq_connector ...
```

Deployment and Configuration

J2EE 1.3 compliant Application Server

To deploy the MQSeries Connector either copy the MQ_CONNECTOR.RAR from MQ_CONNECTOR/DEPLOY into the APPILCATIONS directory of your server or use the management console.

The configuration parameters are stored in the deployment descriptor "ra.xml" into the MQ_CONNECTOR.RAR. To rebuild the archive use command "ANT pack" in the root directory of the connector installation.

To configure the connector use the tools that comes with your application server. Please refer to the server documentation for further information.

Configuring the connector for Point-to-Point model

1. Change the `<managedconnectionfactory-class>` property in "ra.xml" to `de.comporsys.connector.mq.queue.ManagedConnectionFactoryImpl`
2. Rebuild the RAR with "ANT pack"
3. Redeploy the MQ_CONNECTOR.RAR

Configuring the connector for Publish-Subscribe model

1. Change the `<managedconnectionfactory-class>` property in "ra.xml" to `de.comporsys.connector.mq.topic.ManagedConnectionFactoryImpl`
2. Rebuild the RAR with "ANT pack"
3. Redeploy the MQ_CONNECTOR.RAR

Configuration Parameters

The COMPORSYS Connector uses parameters to adapt to features of the target environment.

Structure: Deployment Descriptor Property (Type)

ConnectionFactory

The configuration of the ConnectionFactory of the COMPORSYS Connectors is done by means of the following parameters.

ServerName (String)

Name of the IBM MQSeries Manager Host.

It can be either a DNS name or a IP address provided.

PortNumber (Integer)

TCP/IP port on which MQSeries Manager is listening.

ConnectionURL (String)

Not used. Use ServerName and PortNumber to identify your MQSeries Manager Host.

CCISupport (Boolean)

Determines which interface the COMPORSYS Connector should offer.

Valid values are: `true`, `false`

The default value is: `false`

Remark

The interface of the ConnectionFactory is either type

`javax.resource.cci.ConnectionFactory`

or `de.comporsys.connector.jms.ConnectionFactory`

XASupport (Boolean)

Enables the `XATransaction` mode.

If this property is set to `true`, the property `BindMode` must be set to `BIND` and the `<transaction-support>` must be `XATransaction`.

Valid values are: `true`, `false`

The default value is: `false`

BindMode (String)

Set the transport type to use.

Valid values are: `BIND`, `TCP`

Default value is: `TCP`

ClientId (String)

Identity of the client to be used.

CCSID (String)

Numeric value of the CCSID.

SendExit (String)

Name of a `SendExit` class to be used.

SendExitInit (String)

Initialisation data for the `SendExit` class.

ReceiveExit (String)

Name of a `ReceiveExit` class to be used.

ReceiveExitInit (String)

Initialisation data for the `ReceiveExit` class.

SecurityExit (String)

Name of a SecurityExit class to be used.

SecurityExitInit (String)

Initialisation data for the SecurityExit class.

Connection

The following parameters define default values for establishing connections to IBM MQSeries.

QueueManagerName (String)

Name of the MQSeries Manager.

ChannelName (String)

Name of the channel to use to connect to the manager.

UserName (String)

The username to connect to the manager.

Remark

This default value is only used if there is no other value given when establishing a connection.

The default value is empty.

Password (String)

The user password to connect to the manager.

Remark

This default value is only used if there is no other value given when establishing a connection.

The default value is empty.

AutoCommitMode (Boolean)

Defines the transaction handling of the COMPORSYS Connector.

With `autoCommitMode = false` the a MQSeries transacted session will be used. Refer to the MQSeries documentation for further information.

Valid values are: `true, false`

Default value is: `false`

LogLevel (Integer)

The level of messages the COMPORSYS Connector is recording.

Remark

The value should only be changed for problem-shooting purposes as the result is a reduction of performance.

Valid values are:

0: LOG_LEVEL_DEBUG

1: LOG_LEVEL_TRACE

2: LOG_LEVEL_WARNING,

3: LOG_LEVEL_ERROR

Default value is: 3 (LOG_LEVEL_ERROR)

Configuring the connector for other JMS providers

Though the connector is optimized for the use with IBM MQSeries it can be connected with other JMS providers like Sonicsoftware's SonicMQ as well.

The JMS Administered Objects of the JMS Provider like Queue, QueueConnectionFactory, Topic and TopicConnectionFactory must be bound to the default naming service, that means

`(new InitialContext()).lookup("name")` must find the desired object.

Configuring for Point-2-Point

1. Change the `<managedconnectionfactory-class>` property in "ra.xml" to `de.comporsys.connector.jms.queue.ManagedConnectionFactoryImpl`
2. Add a `<config-property>` to your "ra.xml" with name `QueueConnectionFactoryName` of type `java.lang.String` and as value the JNDI name of the QueueConnectionFactory to be used by the connector.
3. Remove all MQSeries specific configuration properties from the "ra.xml"
4. Rebuild the RAR with "ANT pack"
5. Redeploy the MQ_CONNECTOR.RAR

Configuring for Publish-Subscribe

1. Change the `<managedconnectionfactory-class>` property in "ra.xml" to `de.comporsys.connector.jms.topic.ManagedConnectionFactoryImpl`
2. Add a `<config-property>` to your "ra.xml" with name `TopicConnectionFactoryName` of type `java.lang.String` and as value the JNDI name of the TopicConnectionFactory to be used by the connector.
3. Remove all MQSeries specific configuration properties from the "ra.xml"
4. Rebuild the RAR with "ANT pack"
5. Redeploy the MQ_CONNECTOR.RAR

Configuring Transaction support

The value of element <transaction-support> in "ra.xml" must match some specific configuration property values. Configure your adapter according to following table.

Transaction Support	Config property	Value
NoTransaction	AutoCommitMode	true
LocalTransaction	AutoCommitMode	false
XATransaction	AutoCommitMode	false
	XASupport	true
	BindMode	BIND (TCP as well, only if you have IBM Extended Transactional Client installed)

Table 3: Transaction support matrix

COMPORSYS Code Generation Tool

Use COMPORSYS Code Generation Tool to:

- create "Java records" from COBOL Copybooks needed for data exchange with mainframes. A record converts data types between COBOL and Java.
- create "Access Classes" to simplify programming.

Processing COBOL Copybooks

Using the description of the message format modeled after a COBOL Copybook the COMPORSYS Connector allows the creation of corresponding Java classes (*records*).

A record represents the message, that the COMPORSYS Connector exchanges with IBM MQSeries. A record allows a standardized access to the individual components of the message and converts them into corresponding COBOL data types.

Command:

```
java de.comporsys.codegen.ImportCobol
    <cobol-source> <output-root> <package-name> <codepage>
```

Optional:

```
<codepage>
```

<cobol-source>	Name of the COBOL Copybook
<output-root>	Output directory of the generated class
<package-name>	Package-Name of the generated class
<codepage>	The code page of the target system (Default: cp273: EDCDIC (German))

Table 4: ImportCobol Parameters

To display the complete list of all parameters and options type:

```
java de.comporsys.codegen.ImportCobol
```

Naming Conventions

1. The Copybook must start with a structured type.
2. The name of the first structured type defines the name of the class generated.
3. Every hyphen "-" in a name will be substituted by an underline "_".
4. Lowercase and uppercase letters are pertinent.

Data type mapping

The following rules for data type mapping are valid for all data types with the Usage-Clause `DISPLAY`, `COMP` and `COMP-3`.

PIC A	java.lang.String
PIC A(n)	java.lang.String
PIC X	java.lang.String
PIC X(n)	java.lang.String
PIC .X.	java.lang.String
PIC 9(n=0..4)	int
PIC 9(n=5..9)	int
PIC 9(n=10..18)	long
PIC S9(n=0..4)	int
PIC S9(n=5..9)	int
PIC S9(n=10..18)	long
PIC 9(n)V(m)	java.math.BigDecimal
PIC S9(n)V(m)	java.math.BigDecimal

Table 5: Data type mapping between COBOL and Java

Generating access classes

The COMPORSYS Connector includes a tool to generate access classes for MQSeries communication according to point-to-point and publish-subscribe model.

An access class

- encapsulate the call to the MQSeries adapter using CCI or UCI,
- contains methods to create records classes,
- can be extended through inheritance.

Point-to-Point

```
java de.comporsys.codegen.GenMQP2P
    <send-queue> <receive-queue> <class-name>
    <input-record-class> <output-record-class>
    <output-root> <package-name>
```

<send-queue>	Name of the send queue
<receive-queue>	Name of the receive queue that may contain the reply message
<class-name>	Name of the generated access class
<input-record-class>	Name of the input record class
<output-record-class>	Name of the output record class
<output-root>	Output directory of the generated class
<package-name>	Package-name of the generated class

Table 6: GenMQP2P Parameters

Publish-Subscribe

```
java de.comporsys.codegen.GenMQPubSub
    <topic> <receive-queue> <class-name>
    <input-record-class> <output-record-class>
    <output-root> <package-name>
```

<topic>	Name of the topic under that the message is published.
<receive-queue>	Name of the receive queue that may contain the reply message
<class-name>	Name of the generated access class
<input-record-class>	Name of the input record class
<output-record-class>	Name of the output record class
<output-root>	Output directory of the generated class
<package-name>	Package-name of the generated class

Table 7: GenMQPubSub Parameters

Remarks

1. Lowercase and uppercase letters are pertinent.
2. The actual existence of the record classes specified are **not** verified at the time it is created.

The generated class contains a commented method `main(String[] args)` that can be used for testing purposes.

Design of access classes

```
public class <CLASS-NAME> {  
  
    public <CLASS-NAME>( ConnectionFactory cf )  
  
        creates an access object, that uses the ConnectionFactory. The  
        ConnectionFactory of both the CCI and UCI can be used.  
  
    public <INPUT-RECORD> createInputRecord()  
  
        creates an instance of the input record.  
        Overwrite if needed.  
  
    public <OUTPUT-RECORD> createOutputRecord()  
  
        creates an instance of the output record.  
        Overwrite if needed.  
  
    public InteractionSpec createInteractionSpec()  
  
        creates and sets up an instance of the interaction spec. (CCI only)  
        Overwrite if needed.  
  
    public Request createRequest( Statement s )  
  
        creates and sets up an instance of the request. (UCI only)  
        Overwrite if needed.  
  
    public <OUTPUT-RECORD> execute( <INPUT-RECORD> input )  
  
        executes the request with the specified input record and returns the  
        output record.  
        Do not overwrite.  
  
}
```

Programming Pattern

Using access classes simplifies the necessary steps to call a program via MQSeries:

1. Find the ConnectionFactory using JNDI
2. Get an instance of the access class
3. Get an instance of the input record
4. Set the values of the input record
5. Call the execute() method of the access object
6. Retrieve the result in the output record

How to transmit a text

1. Run the appropriate access class generator with the `de.comporsys.connector.datatype.DefaultTextRecord` class or a subclass of it.

Example:

```
java de.comporsys.codegen.GenMQP2P SEND.QUEUE RECEIVE.QUEUE
MQAccess de.comporsys.connector.datatype.DefaultTextRecord
de.comporsys.connector.datatype.DefaultTextRecord c:\work p2p
or
```

```
java de.comporsys.codegen.GenPubSub MYTOPIC RECEIVE.QUEUE
MQAccess de.comporsys.connector.datatype.DefaultTextRecord
de.comporsys.connector.datatype.DefaultTextRecord c:\work pubsub
```

2. Execute the access object or extend the class to set further properties.

How to transmit a record class

1. Run the CODEGEN ImportCobol tool to generate a record class from a COBOL file.

If you don't want to use the generator you can use any other record class that implements the `javax.resource.cci.Streamable` interface.

2. Run the appropriate access class generator with your record classes.

Example:

```
java de.comporsys.codegen.GenMQP2P SEND.QUEUE RECEIVE.QUEUE
MQAccess record.MyInRecord record.MyOutRecord c:\work p2p
or
```

```
java de.comporsys.codegen.GenPubSub MYTOPIC RECEIVE.QUEUE
MQAccess record.MyInRecord record.MyOutRecord c:\work pubsub
```

3. Execute the access object or extend the class to set further properties.

How to transmit a byte array

1. Run the appropriate access class generator with the

`de.comporsys.connector.datatype.BytesRecord` class or a subclass of it.

Example:

```
java de.comporsys.codegen.GenMQP2P SEND.QUEUE RECEIVE.QUEUE
MQAccess de.comporsys.connector.datatype.BytesRecord
de.comporsys.connector.datatype.BytesRecord c:\work p2p
or
```

```
java de.comporsys.codegen.GenPubSub MYTOPIC RECEIVE.QUEUE
MQAccess de.comporsys.connector.datatype.BytesRecord
de.comporsys.connector.datatype.BytesRecord c:\work pubsub
```

2. Execute the access object or extend the class to set further properties.

Programming with the UCI

Even if using generated access classes is recommended, it may be necessary to program the COMPORSYS Connector directly.

Although the COMPORSYS Connector supports the standard Common Client Interface (CCI) you may want to use the User Call Interface (UCI) for special purposes.

Setting up the connector for UCI

Set the **CCISupport** property of the `ManagedConnectionFactoryImpl` to **false** to enable the UCI.

Establishing a connection

The connection to MQSeries is created by the *ConnectionFactory* object.

The creation of a *ConnectionFactory* object is depending on whether the COMPORSYS Connector is being used in a *Managed Environment* or *Unmanaged Environment*.

Managed Environment

The application server is responsible for the creation of the *ConnectionFactory* object in accordance with properties for configuration and publishing in the Naming Service.

The client gets the *ConnectionFactory* object using the Java Naming and Directory Interface (JNDI) for immediate use.

```
import javax.naming.*;
import de.comporsys.connector.jms.* ;
.
.
.
InitialContext context = new InitialContext();
ConnectionFactory cf = (ConnectionFactory) context.lookup("MQ");
Connection c = cf.getConnection();
```

Unmanaged Environment

Using the COMPORSYS Connector outside of an Application Server the client must create, configure and manage the *ConnectionFactory* object.

The client uses `createConnectionFactory()` of the *ManagedConnectionFactory* to create a new *ConnectionFactory* object.

```
import de.comporsys.connector.mq.queue.* ;
import de.comporsys.connector.jms.* ;
.
.
.
ManagedConnectionFactoryImpl mcf = new
    ManagedConnectionFactoryImpl();
mcf.setServerName("myhost");
mcf.setPortNumber("1414");
mcf.setQueueManagerName("QM_DEMO");
ConnectionFactory cf = (ConnectionFactory)
    mcf.createConnectionFactory();
Connection con = cf.getConnection();
```

Transaction Management

The COMPORSYS Connector integrates into the Java Transaction Service (JTS) of the Application Server and propagates the transactional context to MQSeries.

If deployed as a JCA compliant resource adapter the transaction management is configured by tag `<transaction-support>` in "ra.xml".

The COMPORSYS Connector supports `NoTransaction`, `LocalTransaction` and `XATransaction`.

Note: To enable the `XATransaction` mode the property `XASupport` must be set to `true` and the `BindMode` must be `BIND`.

In an Unmanaged Environment outside of an Application Server the COMPORSYS Connector uses own methods to set up transaction management.

The type of transaction management depends on the `AutoCommitMode` property of the `Connection` object. For default all connections are created with the `AutoCommitMode` property value of the `ConnectionFactory` object. The property can also be set for every single connection.

- `public void setAutoCommitMode(boolean autocommit)`

The actual value can be retrieved by

- `public boolean getAutoCommitMode()`

Property	Influence on Transaction Management
autoCommit=true	<ul style="list-style-type: none"> ▪ Every call is executed in it's own transaction. ▪ No integration into the JTS takes place in the Managed Environment. ▪ <transaction-support> must be set to <code>NoTransaction</code>
autoCommit=false	<ul style="list-style-type: none"> ▪ All calls using the same connection are executed in a common session. ▪ In a Managed Environment: <ul style="list-style-type: none"> - The transaction context and the result of the JTS transaction (COMMIT, ROLLBACK) are propagated to MQSeries. - <transaction-support> must be either <code>LocalTransaction</code> OR <code>XATransaction</code> ▪ In an Unmanaged Environment: <ul style="list-style-type: none"> - The transaction management must be done by the Client.

Table 8: Influence of AutoCommit on Transaction Management

Remark

- For the duration of transactions resources are locked, especially database locks hold to rollback changes. Transactions should be as short as possible so as not to be depended on the duration of user input.

Managed Environment

The COMPORSYS Connector integrates into the JTS and assures that all calls within a JTS transaction operate through the same session in MQSeries.

Transaction management is applied by the Java 2 Enterprise Edition (J2EE):

- Container Managed Transaction with Enterprise JavaBeans
- explicit transaction management through the interface `javax.transaction.UserTransaction`

The COMPORSYS Connector propagates the actual transaction context with the executed request to MQSeries and reports the result to MQSeries.

Unmanaged Environment

The Java transaction service is not available and the transaction management is done by the client. The connection offers two methods:

- `public void commit() throws javax.resource.ResourceException`
- `public void rollback() throws javax.resource.ResourceException`

The method `commit` acknowledges all changes of all requests that were executed using the connection.

The method `rollback` cancels all changes of all requests that were executed using the connection.

Remark

Transactions that are not finished, either because of a programming error or a program abort, will be cancelled at the end of the process by the COMPORSYS Connector using `ROLLBACK`.

Accessing the UCI

The client uses the created connection to get the *Statement* object needed to access the UCI. All functions of the UCI can be called by using the *Statement* object.

```
import de.comporsys.connector.jms.* ;  
.  
.  
.  
Statement s = con.createStatement();
```

Point-to-Point Communication

The *Request* object is set with the name of the MQSeries queues and the message to be transferred. If you are using the connector with another JMS provider, type the JNDI names of the JMS Administered Objects.

The parameterized *Request* object is executed using the *Statement* object.

```
Request request = s.createRequest();  
request.setSendQueueName( "DEMO.SEND.QUEUE" );  
request.setReceiveQueueName( "DEMO.RECEIVE.QUEUE" );  
// set message as a string  
request.setMessage( "MyMessage" );  
// or as a record class  
request.setMessage( myrecord );  
Reply r = s.execute ( request );
```

Publish-Subscribe Communication

The *Request* object is set with the name of the MQSeries topic and the message to be transferred. The parameterised *Request* object is executed using the *Statement* object.

```
Request request = s.createRequest();
request.setTopicName( "DEMO.TOPIC" );
request.setReceiveQueueName( "DEMO.RECEIVE.QUEUE" );
// set message as a string
request.setMessage( "MyMessage" );
// or as a record class
request.setMessage( myrecord );
Reply r = s.execute( request );
```

Properties of the Request class

The Request class offers a set of properties. They allow you to configure the communication for your special needs.

Property	Meaning
SendQueueName	Provide the name of the queue to send the request message to.
ReceiveQueueName	Provide the name of the queue to receive the reply message from.
TopicName	Provide the name of the topic to publish the message under. (Publish-Subscribe only)
TargetClient	Specifies the target client. If the target client is a legacy application, this property must be set to TARGETCLIENT_NONJMS. One of constant value: Request.TARGETCLIENT_NONJMS Request.TARGETCLIENT_JMS
InteractionMode	Specifies the communication model. One of the constant value: javax.resource.cci.InteractionSpec.SYNC_SEND_RECEIVE

	javax.resource.cci.InteractionSpec.SYNC_SEND javax.resource.cci.InteractionSpec.SYNC_RECEIVE
GenerateCorrelationID	If set to true, the bean creates a unique correlation id and expects the same id as correlation id of the reply message. Otherwise the message id of the request is expected is correlation id of the reply.
Timeout	Sets the milliseconds to wait for the reply message.
Expiration	Sets the expiry of the message in milliseconds.
DeliveryMode	Sets the delivery mode of this message One of constant value: javax.jms.DeliveryMode.NON_PERSISTENT javax.jms.DeliveryMode.PERSISTENT
Priority	Sets the priority of this message.
NoLocal	if set, inhibits the delivery of messages published by this connection (Publish-Subscribe only)
Selector	Sets an optional JMS selector string to receive a message. (requires InteractionVerb.SYNC_RECEIVE)

Table 9: Properties of the Request class

How to use InteractionMode

The COMPORSYS Connector supports three InteractionModes.

javax.resource.cci.InteractionSpec.SYNC_SEND_RECEIVE

The request sends the message and waits the given timeout for a reply message. The given GenerateCorrelationID property describes the expected CorrelationID of the reply message.

javax.resource.cci.InteractionSpec.SYNC_SEND

The request sends the message and returns immediately. The expected CorrelationID can be retrieved from the request with `getReplyHandle`.

javax.resource.cci.InteractionSpec.SYNC_RECEIVE

The request receives a message from a queue.

To receive a specific reply message, set the earlier stored reply handle. If no reply handle is provided, then the first available message will be received. The request waits the given timeout for the message.

Setting JMS properties

You can set any JMS standard or JMS provider-specific property by adding the desired property to the request.

This example shows how to turn on a report option for IBM MQSeries.

```
Request request = s.createRequest();
...
request.addProperty( "JMS_IBM_Report_COA",
    new Integer( com.ibm.mq.MQC.MQRO_COA ) ) ; ;
...
Reply r = s.execute( request );
```

Programming with the CCI

Setting up the connector for CCI

Set the **CCISupport** property of the `ManagedConnectionFactoryImpl` to **true** to enable the CCI.

Establishing a connection

The connection to MQSeries is created by the *ConnectionFactory* object.

The creation of a *ConnectionFactory* object is depending on whether the COMPORSYS Connector is being used in a *Managed Environment* or *Unmanaged Environment*.

Managed Environment

The application server is responsible for the creation of the *ConnectionFactory* object in accordance with properties for configuration and publishing in the Naming Service.

The client gets the *ConnectionFactory* object using the Java Naming and Directory Interface (JNDI) for immediate use.

```
import javax.naming.*;
import javax.resource.cci.* ;
.
.
.
InitialContext context = new InitialContext();
ConnectionFactory cf = (ConnectionFactory) context.lookup("MQ");
Connection c = cf.getConnection();
```

Unmanaged Environment

Using the COMPORSYS Connector outside of an Application Server the client must create, configure and manage the *ConnectionFactory* object.

The client uses `createConnectionFactory()` of the *ManagedConnectionFactory* to create a new *ConnectionFactory* object.

```
import de.comporsys.connector.mq.queue.* ;
import javax.resource.cci.* ;
.
.
.
ManagedConnectionFactoryImpl mcf = new
    ManagedConnectionFactoryImpl();
mcf.setServerName("myhost");
mcf.setPortNumber("1414");
mcf.setQueueManagerName("QM_DEMO");
ConnectionFactory cf = (ConnectionFactory)
    mcf.createConnectionFactory();
Connection con = cf.getConnection();
```

Transaction Management

The COMPORSYS Connector integrates into the Java Transaction Service (JTS) of the Application Server and propagates the transactional context to MQSeries.

If deployed as a JCA compliant resource adapter the transaction management is configured by tag `<transaction-support>` in "ra.xml".

The COMPORSYS Connector supports `NoTransaction`, `LocalTransaction` and `XATransaction`.

Note: To enable the `XATransaction` mode the property `XASupport` must be set to `true` and the `BindMode` must be `BIND`. If you have installed the IBM Extended Transactional Client, the you can set `BindMode` to `TCP` as well.

Managed Environment

The COMPORSYS Connector integrates into the JTS and assures that all calls within a JTS transaction operate through the same session in MQSeries.

Transaction management is applied by the Java 2 Enterprise Edition (J2EE):

- Container Managed Transaction with Enterprise JavaBeans
- explicit transaction management through the interface
`javax.transaction.UserTransaction`

The COMPORSYS Connector propagates the actual transaction context with the executed request to MQSeries and reports the result to MQSeries.

Unmanaged Environment

The Java transaction service is not available and the client does the transaction management. The connection offers a method to access the

`javax.resource.cci.LocalTransaction` to control the transaction.

```
...
Connection con = cf.getConnection();
LocalTransaction tx = con.getLocalTransaction() ;
tx.begin() ;
try {
    // do some interactions
    tx.commit() ;
} catch ( Exception e ) {
    tx.rollback() ;
}
```

Point-to-Point Communication

The *InteractionSpec* object is set with the name of the MQSeries queues and the message to be transferred. If you are using the connector with another JMS provider, type the JNDI names of the JMS Administered Objects.

The parameterised *InteractionSpec* object is executed using the *Interaction* object.

```
import de.comporsys.connector.jms.cci.InteractionSpecImpl ;
import de.comporsys.connector.datatype.DefaultTextRecord ;
import javax.resource.cci.* ;
.
.
.
Connection con = cf.getConnection();
// create and setup the specification
InteractionSpecImpl spec = new InteractionSpecImpl() ;
spec.setSendQueueName( "DEMO.SEND.QUEUE" ) ;
spec.setReceiveQueueName( "DEMO.RECEIVE.QUEUE" ) ;
// set up the input message as a record
DefaultTextRecord in = new DefaultTextRecord() ;
in.setText( "MyMessage" );
// create the output record to hold the reply
DefaultTextRecord out = new DefaultTextRecord() ;
// create and execute the interaction
Interaction ix = con.createInteraction() ;
ix.execute ( spec, in, out );
// extract the message from the output record
System.out.println( "received: " + out.getText() ) ;
```

Publish-Subscribe Communication

The *InteractionSpec* object is set with the name of the MQSeries topic and the message to be transferred. The parameterised *InteractionSpec* object is executed using the *Interaction* object.

```
import de.comporsys.connector.jms.cci.InteractionSpecImpl ;
import de.comporsys.connector.datatype.DefaultTextRecord ;
import javax.resource.cci.* ;
.
.
.
Connection con = cf.getConnection();
// create and setup the specification
InteractionSpecImpl spec = new InteractionSpecImpl() ;
spec.setTopicName( "DEMO.TOPIC" ) ;
spec.setReceiveQueueName( "DEMO.RECEIVE.QUEUE" ) ;
// set up the input message as a record
DefaultTextRecord in = new DefaultTextRecord() ;
in.setText( "MyMessage" );
// create the output record to hold the reply
DefaultTextRecord out = new DefaultTextRecord() ;
// create and execute the interaction
Interaction ix = con.createInteraction() ;
ix.execute ( spec, in, out );
// extract the message from the output record
System.out.println( "received: " + out.getText() ) ;
```

Properties of the InteractionSpec bean

The InteractionSpec bean offers a set of properties. They allow you to configure the communication for your special needs.

Property	Meaning
SendQueueName	Provide the name of the queue to send the request message to.
ReceiveQueueName	Provide the name of the queue to receive the reply message from.
TopicName	Provide the name of the topic to publish the message under. (Publish-Subscribe only)

InteractionVerb	<p>Specifies the communication model.</p> <p>One of the constant value:</p> <p>javax.resource.cci.InteractionSpec.SYNC_SEND_RECEIVE javax.resource.cci.InteractionSpec.SYNC_SEND javax.resource.cci.InteractionSpec.SYNC_RECEIVE</p>
GenerateCorrelationID	<p>If set to true, the bean creates a unique correlation id and expects the same id as correlation id of the reply message. Otherwise the message id of the request is expected is correlation id of the reply.</p>
ExecutionTimeout	<p>Sets the milliseconds to wait for the reply message.</p>
Expiration	<p>Sets the expiry of the message in milliseconds.</p>
DeliveryMode	<p>Sets the delivery mode of this message</p> <p>One of constant value:</p> <p>javax.jms.DeliveryMode.NON_PERSISTENT javax.jms.DeliveryMode.PERSISTENT</p>
Priority	<p>Sets the priority of this message.</p>
NoLocal	<p>if set, inhibits the delivery of messages published by this connection (Publish-Subscribe only)</p>
Selector	<p>Sets an optional JMS selector string to receive a message. (requires InteractionVerb.SYNC_RECEIVE)</p>

Table 10: Properties of the InteractionSpec bean

How to use InteractionVerbs

The COMPORSYS Connector supports all three specified InteractionVerbs.

javax.resource.cci.InteractionSpec.SYNC_SEND_RECEIVE

The interaction sends the request message and waits the given timeout for a reply message. The given GenerateCorrelationID property describes the expected CorrelationID of the reply message.

javax.resource.cci.InteractionSpec.SYNC_SEND

The interaction sends the request message and returns immediately. The expected CorrelationID is stored in the interaction instance.

javax.resource.cci.InteractionSpec.SYNC_RECEIVE

The interaction receives a message from a queue.

It uses an earlier stored CorrelationID to receive the corresponding reply. If the interaction does not have an outstanding reply, then the first available message will be received. It waits the given timeout for the message.

Setting JMS properties

You can set any JMS standard or JMS provider-specific property by adding the desired property to the request. Use the java wrapper classes for setting primitive data types.

This example shows how to turn on a report option for IBM MQSeries.

```
InteractionSpecImpl spec = new InteractionSpecImpl() ;
...
spec.addProperty( "JMS_IBM_Report_COA",
    new Integer( com.ibm.mq.MQC.MQRO_COA ) ) ; ;
...
ix.execute( spec, in, out );
```

This example shows how to send a message as the last message of a group.

```
InteractionSpecImpl spec = new InteractionSpecImpl() ;
...
in.addProperty( "JMSXGroupID", "ID:1234567890abcdef" ) ;
in.addProperty( "JMSXGroupSeq", new Integer( 1 ) ) ;
in.addProperty( "JMS_IBM_Last_Msg_In_Group",
    new Boolean( true ) ) ;
...
ix.execute( spec, in, out ) ;
```

Retrieving JMS properties

All JMS standard and vendor-specific properties are stored as properties on the output record. The values can be either easily accessed by using the corresponding constants on the `InteractionSpecImpl` class or by using the name of the `JMSProperty`.

This example shows how to retrieve the `JMSMessageId` of the received message.

```
InteractionSpecImpl spec = new InteractionSpecImpl() ;
...
ix.execute( spec, in, out ) ;
String corrId = out.getProperty( spec.PROP_CORRELATION_ID ) ;
//same as: String corrId = out.getProperty( "JMSCorrelationID" ) ;
...
```

Accessing the underlying JMSMessage

For some reasons it may be useful to access the underlying `JMSMessage` directly. Therefore the last message sent or received by the adapter is stored as property on the output record.

This example shows how to retrieve the message of the received message.

```
InteractionSpecImpl spec = new InteractionSpecImpl() ;
spec.setInteractionVerb( spec.SYNC_SEND_RECEIVE ) ;
...
ix.execute( spec, in, out );
Message msg = (javax.jms.Message)out.getProperty(spec.PROP_MESSAGE);
...
```

Support for MQ/IMS Bridge

The COMPORSYS Connector supports the MQ/IMS Bridge to access IMS transactions directly.

Use the interaction specification

`de.comporsys.connector.mq.cci.IMSBridgeInteractionSpecImpl` to send or receive and message to or from the IMS Bridge.

The connector supports both MQ/IMS bridge message types:

- Messages containing IMS transaction data and an MQIIH structure
- Messages containing IMS transaction data but no MQIIH structure

The interaction specification class provides getter and setters for all fields of the MQIIH header. To use the MQIIH header set the property `MQIIH` to `true` and provide the proper values.

If the property `MQIIH` is set to `false` the MQ/IMS Bridge creates a default header and removes it automatically.

This example shows how to send and receive of message with MQIIH header over the IMS Bridge. The MQIIH structure and the received bytes are accessible as properties on the output record.

```
import de.comporsys.connector.mq.cci.IMSBridgeInteractionSpecImpl ;
import de.comporsys.connector.mq.imsbridge.MQIIH ;
...
BytesRecord in = new BytesRecord() ;
input.setBytes( "LLZZTRANCODEDATA".getBytes( codepage ) ) ;
BytesRecord out = new BytesRecord() ;

IMSBridgeInteractionSpecImpl spec =
    new IMSBridgeInteractionSpecImpl() ;

spec.setMQIIH( true ) ;
spec.setInteractionVerb( spec.SYNC_SEND_RECEIVE ) ;
// set more MQIIH properties an InteractionSpec
...
ix.execute( spec, in, out ) ;
...
// access extended values as record properties
BytesRecord fullRec = (BytesRecord) out.getProperty(
    spec.PROP_FULL_REPLY_RECORD ) ;
MQIIH mqiih = (MQIIH) out.getProperty( spec.PROP_MQIIH ) ;
```

Use the COMPORSYS CodeGen tool to generate a record class based on a Cobol copybook description.

Notice: The record class must contain the LLZZ fields.

Further information

- JAVADOC of the COMPORSYS Connector for IBM MQSeries
[%MQSeries_CONNECTOR%/doc/javadoc/index.html](#)
- J2EE™ Java Connector Architecture
<http://www.javasoft.com/j2ee/connector>
- IBM MQSeries
<http://www.ibm.com>
- BEA WebLogic Server
<http://edocs.bea.com>
- Support for installation and deployment of the COMPORSYS Connector
<mailto://support@comporsys.de>