

---

**COMPORSYS Connector for IBM CICS/TXSeries**

**Version 1.5**

# **Installation and Reference Guide**

---

**Contents**

**Preface..... 5**

**Installation ..... 7**

    Requirements ..... 7

    Contents of CICS\_CONNECTOR.ZIP..... 8

    J2EE 1.3 compliant Application Server..... 8

        Installation steps..... 8

**Deployment and Configuration..... 10**

    J2EE 1.3 compliant Application Server..... 10

    Configuration Parameters ..... 10

        ConnectionFactory ..... 10

        Connection ..... 13

        Connection Pooling ..... 15

**COMPORSYS CodeGen ..... 17**

    Installation..... 17

    External Call Interface ..... 17

    Processing COBOL copybooks..... 18

        Naming Conventions ..... 18

        Supported COBOL keywords..... 19

        Data type mapping ..... 19

        Using HIGH-VALUE and LOW-VALUE..... 20

    Generating access classes for CICS programs..... 20

    Design of access classes..... 22

    Programming Pattern..... 23

**Programming with the UCI..... 24**

    Establishing a connection ..... 24

        Managed Environment..... 24

        Unmanaged Environment ..... 25

    Transaction Management ..... 26

        Managed Environment..... 28

        Unmanaged Environment ..... 28

    External Call Interface ..... 29

COMPORSYS Connector for IBM CICS/TXSeries ..... 2

Contents

- Accessing the ECI..... 29
- Synchronous program call ..... 29
- Asynchronous program calls..... 30
- Properties of an ECI request..... 31
- External Presentation Interface ..... 33
- Accessing the EPI ..... 33
- Accessing a Transaction using EPI..... 33
- Programming with Macros..... 35
- External Security Interface ..... 37
- Accessing the ESI ..... 37
- User authentication..... 38
- Changing the user password ..... 38
- Query user account information..... 39
- Programming with the CCI ..... 40**
- Setting up the connector for CCI ..... 40
- Establishing a connection ..... 40
- Managed Environment..... 40
- Unmanaged Environment ..... 41
- Transaction Management ..... 42
- Managed Environment..... 42
- Unmanaged Environment ..... 43
- Call a program using CCI..... 44
- Properties of the InteractionSpec bean ..... 44
- How to use InteractionVerbs ..... 45
- Further information..... 46**

**List Of Tables**

Table 1: Directory Structure ..... 8

Table 2: Required Java Archives..... 9

Table 3: ImportCobol Parameters..... 18

Table 4: ImportCobol Options ..... 18

Table 5: Data type mapping between COBOL and Java ..... 19

Table 6: GenECI Parameters..... 21

Table 7: Influence of AutoCommit on Transaction Management ..... 27

Table 8: Properties of an ECI request..... 32

Table 9: Properties of an EPI request..... 34

Table 10: Properties of the InteractionSpec bean..... 45

## Preface

The COMPORSYS Connector for IBM CICS/TXSeries enables fast and easy integration of CICS/TXSeries transactions and programs in Java applications. 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)<sup>™</sup> (*Managed Environment*).

The COMPORSYS Connector includes

- a resource adapter for IBM CICS/TXSeries according to the J2EE<sup>™</sup> Connector Architecture Specification 1.0.
- the code generator COMPORSYS CodeGen
  - to create Java records from COBOL Copybooks for data exchange with IBM CICS. These *records* allow an easy access to the Communication Area and carry-through the necessary mapping of data types.
  - to create access records for CICS programs, 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.

Three types of call interfaces for CICS programs and transactions are supported:

- The External Call Interface (ECI) makes feasible the synchronous as well as asynchronous execution of CICS programs within an transaction (Distributed Program Link). The data exchange takes place in a Communication Area (COMMAREA).
- The External Presentation Interface (EPI) creates a virtual terminal, whose display is accessed using an API to start transactions and get results (Screen Scraping).
- The External Security Interface (ESI) provides methods for the authentication of users and password management.

## Installation

### Requirements

- IBM CICS Transaction Gateway 3.1.x installed and configured.  
Please contact your local IBM representative to get a copy of the Gateway
- BEA WebLogic Server 8.1 or higher installed. We recommend BEA WebLogic Server or a different Application Server with support for Java Connector Architecture.  
(<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 CICS\_CONNECTOR.ZIP

The CICS\_CONNECTOR.ZIP contains all files needed for the COMPORSYS Connector for IBM CICS/TXSeries except the classes of the IBM CICS Transaction Gateway. These must be downloaded separately.

Included are:

- Resource adapter CICS\_CONNECTOR.RAR
- Development tool CODEGEN.
- Required libraries
- Documentation and examples

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

**Table 1: Directory Structure**

## J2EE 1.3 compliant Application Server

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

### Installation steps

Follow these steps to install the COMPORSYS Connector for IBM CICS/TXSeries.

#### 1. Copy the COMPORSYS Connector installation files

Unzip the provided CICS\_CONNECTOR.ZIP into the root directory of the BEA WebLogic Server (p.e. c:\bea\weblogic81).

## 2. Editing the Server CLASSPATH

Add the following Java Archives to the server CLASSPATH:

CICS_CONNECTOR.JAR	COMPORSYS Connector for IBM CICS/TXSeries
CONNECTOR_FOUNDATION.JAR	COMPORSYS Connector Foundation Classes
CTGCLIENT.JAR	IBM CICS Transaction Gateway Client (contained in %CTG%\classes)

**Table 2: Required Java Archives**

### Example:

```
startWebLogicServer.cmd:
...
set CTG=c:\ibm\ctg311\classes\ctgclient.jar
set CONNECTOR=.\cics_connector\lib\cics_connector.jar;
    .\cics_connector\lib\connector_foundation.jar;%CTG%

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

## Deployment and Configuration

### J2EE 1.3 compliant Application Server

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

To avoid class-loading issues it is strongly recommended to embed the CICS\_CONNECTOR.RAR in your application's EAR file.

The configuration parameters are stored in the deployment descriptor "ra.xml" into the CICS\_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.

### Configuration Parameters

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

**Structure:** Deployment Descriptor Property (Type) / startup argument

#### ConnectionFactory

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

#### ServerName (String) / serverName

Name of the IBM CICS Server as given in the IBM CICS Transaction Gateway.

**Remark**

Value of the field *Server Name* in the section *Client/Server Connection* of the IBM CICS Transaction Gateway Configuration Tool.

**ConnectionURL (String) / connectionURL**

The Uniform Resource Locator, that describes the connection to the IBM CICS Transaction Gateway. Syntax: protocol://hostname:port

You may add multiple URLs separated by “,” or “;” to enable the connector’s failover functionality. In case a gateway goes down, the connector will automatically reconnect to the next available gateway on the list.

**Remark**

Please refer to the IBM CICS Transaction Gateway documentation for further information.

**CCISupport (Boolean) / cciSupport**

Determines which interface the COMPORSYS Connector should offer.

Valid values are: `true`, `false`

The default value is: `false`

**Remark for J2EE 1.3 compliant Application Server**

The interface of the `ConnectionFactory` is either type

`javax.resource.cci.ConnectionFactory`

or `de.comporsys.connector.cics.ConnectionFactory`

**ESISupport (Boolean) / esiSupport**

Indicates, if the target system supports the External Security Interface for authentication.

Valid values are: `true`, `false`

**Remark**

The ESI can be used only for CICS Transaction Server employing SNA LU6.2.

The default value is: `false`

### **TimeoutSecs (Integer) / timeout**

shows the maximal duration of an operation in seconds.

#### **Remark**

This should not be too small as in case of a timeout the client will not be able to realize the outcome of the transaction in CICS.

The default value is: 30

### **Compression (Integer) / compression**

defines the type of compression required for the communication between the IBM CICS Transaction Gateway and the COMPORSYS Connector.

0 (COMPRESSION\_NONE):

No compression is used.

1 (COMPRESSION\_ZIP):

The ZIP algorithm is used for compression.

2 (COMPRESSION\_TRAILING):

An algorithm is used that compresses similar symbols at the end of the COMMAREA.

#### **Remark**

The compression should only be used when the network between the IBM CICS Transaction Gateway and the COMPORSYS Connector does not have enough capacity.

As the compression is done by "Exit Classes", it will not be possible to simultaneous use compression and `clientExitClass/serverExitClass`.

The default value is: 0 (COMPRESSION\_NONE)

### **ServerExitClass (String) / serverExitClass**

The entire classname (including packagename) implementing the interface.

`com.ibm.ctg.security.ServerSecurity`

**Remark**

You will find more information on "Exit Classes" in the documentation of the IBM CICS Transaction Gateway.

**ClientExitClass (String) / clientExitClass**

The entire classname (including packagename) implementing the interface.

```
com.ibm.ctg.security.Client.Security
```

**Remark**

Further information on using "Exit Classes" can be found in the IBM CICS Transaction Gateway documentation.

**not applicable / prefix**

Defines the context under which the ConnectionFactory of the COMPORSYS Connector is registered in the naming service.

**Remark**

The ConnectionFactory of the COMPORSYS Connector is registered under the name of the StartupClass in the root context of the naming service. Use this parameter to specify a "context.subcontext" for a name such as "context.subcontext.Name".

**Connection**

The following parameters define default values for establishing connections to IBM CICS/TXSeries.

**UserName (String) / userName**

The username under which calls to IBM CICS/TXSeries are executed.

**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) / password**

The user password under which calls to IBM CICS/TXSeries are executed.

#### **Remark**

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

The default value is empty.

### **AutoCommit (Boolean) / autoCommit**

Defines the transaction handling of the COMPORSYS Connector.

With `autoCommit = false` the transaction context of the Application Server is forwarded to IBM CICS/TXSeries. Multiple calls can be executed over ECI in one Extend Logical Unit of Work of IBM CICS/TXSeries. Further information on the subject of "transactions" can be found in the chapter "Transaction Management".

Valid values are: `true, false`

#### **Remark**

To make use of Extend Logical Unit of Works you will need to communicate using SAN LU 6.2 or Recoverable Resource Service of the OS/390.

Default value is: `false`

### **LogLevel (String) / logLevel**

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:

DEBUG, TRACE, WARN, ERROR

Default value is: ERROR

## **Connection Pooling**

The following parameter influence the Connection Pooling installed in the COMPORSYS Connector. They are only supported if the connector is installed in BEA WebLogic Server 5.1 or 6.0.

### **initialCapacity**

The number of connections established at the start of the Application Server.

#### **Remark**

If you choose a value greater 0 the COMPORSYS Connector must be able to establish a connection to CICS at the start up time of the Application Server or the COMPORSYS Connector will not be properly installed.

Default value is: 0

### **maxCapacity**

The maximal number of connections created and managed in the Connection Pool.

#### **Remark**

The value 0 creates an unlimited Connection Pool.

Default value is: 0

### **loginDelaySeconds**

Time delay in seconds between establishing two connections.

#### **Remark**

Default value is: 0

### **waitTime**

Time in milliseconds the Connector waits for a free connection of the Connection Pool before throwing an exception.

**Remark**

With a value of -1 you the Connector waits infinite time for a free connection.

Default value is: -1

## COMPORSYS CodeGen

Use the COMPORSYS Connector code generation tool COMPORSYS CodeGen to:

- create "Java records" from COBOL copybooks needed for data exchange with IBM CICS. A record allows access to the COMMAREA and converts data types between COBOL and Java.
- create "Access Classes" to work with the CICS programs using the External Call Interface.

### Installation

Add the CODEGEN.JAR Java archive to your CLASSPATH.

### External Call Interface

The ECI allows Java applications to call CICS programs . The application can be connected to multiple servers at the same time and execute various program calls parallel.

The called CICS programs can change resources of the same CICS Server and can call CICS programs on other servers using Distributed Program Link (DPL) or Distributed Transaction Processing (DTP).

The called CICS programs should not execute Terminal I/O Operations, but can read or write other CICS resources . The call for a CICS program appears through ECI as if it came from other programs using EXEC CICS LINK. The parameters are transferred through a COMMAREA.

Multiple ECI calls can be comprehended to one Logical Unit of Work.

## Processing COBOL copybooks

Using the description of the COMMAREA modelled after a COBOL copybook the COMPORSYS Connector allows the creation of corresponding Java classes (*records*).

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

### Command:

```
java de.comporsys.codegen.ImportCobol [-options]
    <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 CICS Server's code page (Default: cp273: EDCDIC (Deutsch))

**Table 3: ImportCobol Parameters**

-innerClass -publicClass	Generate subrecords either as public classes or inner classes. (Default: -innerClass)
-noInit -init	Either do not initialize record (record will contain binary zeros) or use VALUE clauses for record initialisation. (Default: -notInit)

**Table 4: ImportCobol Options**

## 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.

### Supported COBOL keywords

The COBOL copybooks could contain the following COBOL keywords:

- PIC, PICTURE, OCCURS, REDEFINES, A, X, 9, V, S, 0, DISPLAY, COMP, COMP-3, USAGE, IS, ARE, BINARY, PACKED-DECIMAL, DEPENDING, ON, TIMES, POINTER, THROUGH, THRU, 88 Levels

### 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)	long
PIC 9(n=10..18)	java.math.BigDecimal
PIC S9(n=0..4)	int
PIC S9(n=5..9)	long
PIC S9(n=10..18)	java.math.BigDecimal
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**

## Using HIGH-VALUE and LOW-VALUE

The COMPORSYS record and data type classes are supporting Java equivalents to the COBOL HIGH-VALUE and LOW-VALUE constructs.

Each record field can be set to and tested for those special values.

- use `set...(de.comporsys.connector.datatype.Record.HIGH_VALUE)` on the generated record class to set the field to HIGH-VALUE.
- use `set...(de.comporsys.connector.datatype.Record.LOW_VALUE)` on the generated record class to set the field to LOW-VALUE.
- use `get...Value()` to get the current value of the field as an object and use `equals(de.comporsys.connector.datatype.Record.HIGH_VALUE)` to test for HIGH-VALUE.
- use `get...Value()` to get the current value of the field as an object and use `equals(de.comporsys.connector.datatype.Record.LOW_VALUE)` to test for LOW-VALUE.

## Generating access classes for CICS programs

The COMPORSYS Connector includes a tool to generate access classes for CICS programs.

An access class

- encapsulate the call to a CICS program using CCI or UCI,
- contains methods to create parameter classes for CICS programs,
- can be extended through inheritance.

Command for JCA compliant application servers:

```
java de.comporsys.codegen.GenECI
    <cics-program> <input-record-class> <output-record-class>
    <output-root> <package-name>
```

Command for BEA WebLogic Server 5.1 or 6.0:

```
java de.comporsys.codegen.GenECI_WLS
    <cics-program> <input-record-class> <output-record-class>
    <output-root> <package-name>
```

<cics-program>	Name of the CICS Program
<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: GenECI Parameters**

1. The created class has the same name as the CICS program.
2. Lowercase and uppercase letters are pertinent.
3. 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

The access class is named after the CICS program.

```
public class <CICS-PROGRAM> {  
  
    public <CICS-PROGRAM>( 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 CICS program:

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

## Programming with the UCI

With the User Call Interface (UCI) the COMPORSYS Connector has a programming interface to the ECI, EPI and ESI. If you are using the standard Common Client Interface (CCI) only the ECI will be available.

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

Calls to CICS via EPI or ESI always uses the UCI of the COMPORSYS Connector.

### Establishing a connection

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

**Remark:** If the COMPORSYS Connector is used within a BEA WebLogic Server 5.1 environment, the interface has to be

`de.comporsys.connector.cics.appserver.weblogic.ConnectionFactory`.

A connection is needed for calls over ECI, EPI and ESI.

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 `weblogic.properties` 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.*;
// for WLS 5.1, 6.0
import de.comporsys.connector.cics.appserver.weblogic.* ;
// for J2EE 1.3 compliant Application Server
import de.comporsys.connector.cics.* ;
.
.
.
InitialContext context = new InitialContext();
ConnectionFactory cf = (ConnectionFactory) context.lookup("CICS");
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.cics.* ;
.
.
.
ManagedConnectionFactoryImpl mcf = new
    ManagedConnectionFactoryImpl();
mcf.setServerName("CICS");
mcf.setConnectionURL("tcp://localhost");
ConnectionFactory cf = (ConnectionFactory)
    mcf.createConnectionFactory();
Connection con = cf.getConnection();
```

## Transaction Management

The COMPORSYS Connector supports CICS *Extended Logical Unit of Works*.

Multiple calls over ECI can be comprehended to one transaction. The COMPORSYS Connector integrates into the Java Transaction Service (JTS) of the Application Server and propagates the current transaction context to CICS.

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:** COMPORSYS Connector implements the 2-phase-commit upon a local transaction contract. Therefore it **does not guarantee** full distributed transactional behaviour with more than one resource managers (`XATransaction`).

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 *AutoCommit* property of the *Connection* object. For default all connections are created with the *AutoCommit* property value of the *ConnectionFactory* object. The property can also be set for every single connection.

- `public void setAutoCommit( boolean autocommit )`

The actual value can be retrieved by

- `public boolean getAutoCommit()`

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

**Table 7: 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.
- To execute Extended Logical Unit of Works you either apply SNA APPC communication or the Recoverable Resource Services of the OS/390. To do so you may have to do more configurations.

## Managed Environment

The COMPORSYS Connector integrates into the JTS and assures that all calls within a JTS transaction operate through the same Extended Logical Unit of Work in CICS.

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 the CICS and reports the results to CICS.

## 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.

## External Call Interface

The ECI allows Java application to call CICS programs. The Java application can be connected to multiple CICS servers at the same time and execute various program calls parallel.

The called CICS programs should not execute Terminal I/O Operations, but can read and write other CICS resources. The call for a CICS program appears through ECI as if it came from other programs using EXEC CICS LINK. The parameters are transferred through a COMMAREA.

The ECI commands are executed either *synchronous* or *asynchronous*. The synchronous call blocks the client until the result returns from the server. Asynchronous calls immediately give control back to the client and informs him as soon as the server has a result.

Multiple ECI calls can be comprehended to one Logical Unit of Work. Asynchronous Commands can also be comprehended into a logical unit.

The called CICS programs can change resources of the same CICS server and can call CICS programs on other servers using Distributed Program Link (DPL) or Distributed Transaction Processing (DTP).

## Accessing the ECI

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

```
import de.comporsys.connector.cics.eci.* ;  
.  
.  
.  
Statement s = con.createECIStatement();
```

## Synchronous program call

The *Request* object represents the call to a particular CICS program. The *Request* object is created with the name of the CICS program to call and the

COMMAREA to be transferred. The parameterised *Request* object is executed using the *Statement* object.

```
Request request = s.createRequest( "PROGNAME" );  
request.setCommArea( "Text", "cp273" );  
Reply r = s.executeQuery( request );
```

The COMPORSYS Connector offers three different methods to send a *Request* object:

- `public Reply executeQuery( Request r )`  
A read-request is executed and the result is returned by the *Reply* object. (The connector does not guaranty that the called program make any changes. The use of `executeQuery` is for legibility only.)
- `public void executeUpdate( Request r )`  
A write-request is executed that changes CICS/TXSeries resources. The Request object returns no data or data that is of no relevance to the client.
- `public Reply execute( Request r )`  
A request is executed whose type is not further specified. The function is not different from `executeQuery( Request r )`.

## Asynchronous program calls

There are two cases for the use of asynchronous communication:

- A program is called that has a long period of run. The results are not needed immediately.
- A program is called that does not deliver any results. The asynchronous commands make it possible for the client to go on to other tasks fasten.

A asynchronous *Request* object is created using `createAsynchronRequest` and parameterised like a synchronous *Request* object.

The *Request* object must instantiate a *ReplyListener* object of abstract class *ReplyAdapter* before processing the request.

```
AsynchronousRequest request = s.createAsynchronousRequest( "PROGNAME" );
request.setCommArea( "Text", "cp273" );
request.setReplyListener( new ReplyAdapter() {
    public void replyReceived( ReplyEvent e ) {
        Reply r = e.getReply() ;
        if (r.getReturnCode().getCode() != Reply.ECI_NO_ERROR ) {
            // Error handling
        } else {
            // Process results
            // ...
        }
    }
} ) ;
s.executeQuery( request );
```

**Remark**

While using asynchronous calls errors will not be reported by exceptions. The client must check for errors before continuing (p.e. ECI\_ERR\_TRANSACTION\_ABEND, ECI\_ERR\_SECURITY\_ERROR, ECI\_ERR\_ROLLEDBACK).

**Properties of an ECI request**

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

Property	Meaning
ProgramName	Provide the name CICS program to call
TransactionId	Sets the CICS transaction identifier.
CommArea	Sets the COMMAREA to transmit.

<b>Property</b>	<b>Meaning</b>
TPN	<p>If set to true the call will use ECI_TPN otherwise ECI_TRANSID. Default value is "true".</p> <p>Refer to the CICS client/server documentation for more information on the difference between ECI_TRANSID and ECI_TPN.</p>

**Table 8: Properties of an ECI request**

## External Presentation Interface

The EPI allows Java applications to act like a 3270 terminal to the CICS server. Like a CICS *virtual terminal* the application starts transactions of a CICS server using the EPI. An application can access the services of different CICS servers by using as many *virtual terminals* as necessary.

The requested CICS transaction can execute other transactions using CICS START. They will also run under the control of the same *virtual terminal*. The EPI communication between CICS server and Java application runs through 3270 data streams and events. The Java application can present the contents of the 3270 data stream graphically or make automatic data inputs.

### Accessing the EPI

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

```
import de.comporsys.connector.cics.epi.* ;  
.  
.  
.  
Statement statement = con.createEPIStatement();
```

### Accessing a Transaction using EPI

To start a CICS/TXSeries transaction the COMPORSYS Connector sends an EPI *Request*. The *Request* object includes the name of the requested transaction and other properties.

Property	Meaning
Transaction	The name of the transaction accessed through this request.
Data	Additional data given to the transaction.
NetName	The net name of the terminal used to execute the request.
DeviceType	The device type of the terminal used to execute the request.

**Table 9: Properties of an EPI request**

A *Macro* is added to the *Request* object to work on the user dialog (*Screen*) of the transaction. The *Macro* can access all elements of the user dialog and read or write field values.

```
Request request = statement.createRequest( "MENU" ) ;
request.addMacro( new DebugMacro() ) ;
Reply reply = statement.executeQuery( r ) ;
```

The method `addMacro` adds as many *Macros* as necessary to the *Request* object. When the *Request* object is executed the *Macros* are executed in the sequence they were added.

The COMPORSYS Connector offers three different methods to send a *Request* object:

- `public Reply executeQuery( Request r )`

A read-request is executed and the result is returned by the *Reply* object. (The connector does not guaranty that the called program make any changes. The use of `executeQuery` is for legibility only.)

- `public void executeUpdate( Request r )`

A write-request is executed that changes CICS/TXSeries resources. Using the EPI changes cannot be rolled back. Therefore the *Request* object will not be executed immediately and the COMPORSYS Connector delays it until COMMIT. So it is possible to synchronize changes over EPI with the result of a JTS transaction.

## Programming with the UCI

- `public Reply execute( Request r )`

A request is executed whose type is not further specified. The function is not different from `executeQuery( Request r )`.

## Programming with Macros

A *Macro* works on the user dialog (*Screen*) of a transaction. It can access all elements of the user dialog and read or write field values.

### Example

The *DebugMacro* is part of the COMPORSYS Connector and dumps the contents of the user dialog to the console.

```
public class DebugMacro extends Macro {
    public void play( Screen screen ) throws Exception {
        System.out.println("Dumping Screen..." ) ;
        for ( int i=1; i <= screen.getFieldCount(); i++ ) {
            if ( screen.getField(i).getTextLength() > 0 ) {
                System.out.println( "Field " + i + ": " +
                    screen.getField(i).getText() );
                // add each fields value to the reply of this macro
                addReply("Field:"+i, screen.getField(i).getText());
            }
        }
    }
}
```

The *InputProvider* supplies data for the *Macro* of a *Request* object. Every *Macro* has methods to read and write input data.

- `public void addInput( String id, String value )`

Adds the input parameter *value* with the name *id* to the macro.

- `public String getInput( String id )`

Queries the actual value of the input parameter *id*.

```
public class MenuMacro extends Macro {
    public void play( Screen s ) {
        // Set the content of field 9 with the value of TRANSACTION
        s.getField( 9 ).setText( getInput( "TRANSACTION" ) ) ;
        // Set the content of field 11 with the value of NUMBER
        s.getField( 11 ).setText( getInput( "NUMBER" ) ) ;
        // Add the content of field 10 with the name ACCOUNT
        // to the result.
        addReply("ACCOUNT", s.getField( 10 ) ) ;
    }
}
```

Every *Macro* can add new input parameters and forward them to other *Macros*.

Use the *ReplyConsumer* of a *Macro* to add data to the result of a *Request* object.

- `public void addReply( String id, String value )`

Adds the output value *value* with the name *id* to the *Reply* object. All values registered in the *Macro* of a *Request* object with `addReply` can be queried using the *Reply* object.

```
Reply reply = statement.executeQuery( r ) ;
String konto = reply.getValue( "ACCOUNT" ) ;
Set allFiels = reply.getEntries() ;
```

## External Security Interface

The ESI allows Java applications to use the Password Expiration Management (PEM) of Advanced Program-To-Program Communication (APPC).

The APPC PEM together with the CICS server supports APPC based login and offers functions to

- identify a user and authenticate the user identity
- notify the user if the password has expired
- change the user password before or after it has expired
- query the validity of the user password
- query failed logins of a certain user identity

To use APPC PEM the communication protocol to the CICS Server must be APPC. Also an external security manager such as RACF must be used. ESI calls can be combined with calls over ECI or EPI.

## Accessing the ESI

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

```
import de.comporsys.connector.cics.esi.* ;  
.  
.  
.  
Statement s = con.createESISatement();
```

## User authentication

Use the method `verifyPassword` to let IBM CICS authenticate a user.

```
try {
    Status status = s.verifyPassword( "USER", "PASSWORD" )
    .
    .
    .
} catch ( SecurityException e ) {
    if (e.getReturnCode().getCode() == e.ESI_ERR_PASSWORD_INVALID){
        // Wrong password
    } else {
        // see return value
    }
}
```

## Changing the user password

Use the method `changePassword` to synchronize the user password with IBM CICS.

```
try {
    Status status = s.changePassword( "USER", "CURRENT", "NEW" )
    .
    .
    .
} catch ( SecurityException e ) {
    // Wrong username/password combination
}
```

## Query user account information

Use the *Status* object returned by the method `verifyPassword` to query various information about the user account.

```
try {
    Status status = s.verifyPassword( "USER", "PASSWORD" )
    System.out.println( "expiry: " + status.getExpiry() ) ;
    System.out.println( "invalid: " + status.getInvalidCount() ) ;
    System.out.println( "accessed: " + status.getLastAccessed() ) ;
    System.out.println( "verified: " + status.getLastVerified() ) ;
} catch ( SecurityException e ) {
    // Wrong username/password combination or
    // password has expired
    // ...
}
```

## Programming with the CCI

### Setting up the connector for CCI

Set the **CCISupport** property of the `ManagedConnectionFactoryImpl` to **true** to enable the CCI. If you are using the standard Common Client Interface (CCI) only the ECI will be available.

**Note:** The CCI is implemented on top of the UCI, so refer to the corresponding topics their for further details if needed.

### Establishing a connection

The connection to CICS 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("CICS");
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 `ManagedConnectionFactoryImpl` to create a new `ConnectionFactory` object.

```
import de.comporsys.connector.cics.* ;
import javax.resource.cci.* ;
.
.
.
ManagedConnectionFactoryImpl mcf = new
    ManagedConnectionFactoryImpl();
mcf.setServerName("CICS");
mcf.setConnectionURL("tcp://localhost");
ConnectionFactory cf = (ConnectionFactory)
    mcf.createConnectionFactory();
Connection con = cf.getConnection();
```

## Transaction Management

Use the `<transaction-support>` in "ra.xml" and the property "AutoCommitMode" to specify the transactional behaviour.

Please find more information about the transactional support of the connector in the UCI section.

## Managed Environment

The COMPORSYS Connector integrates into the JTS and assures that all calls within a JTS transaction operate through the same Extended Logical Unit of Work in CICS.

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 the CICS and reports the results to CICS.

## 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() ;
}
```

## Call a program using CCI

The *InteractionSpec* object is set with the name of the CICS program and the data to be transferred.

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

```
import de.comporsys.connector.cics.cci.InteractionSpecImpl ;
import javax.resource.cci.* ;
.
.
.
Connection con = cf.getConnection();
// create and setup the specification
InteractionSpecImpl spec = new InteractionSpecImpl() ;
spec.setFunctionName( "TESTPRG" ) ;
// set up the input message as a record
MyRecord in = new MyRecord() ;
in.setText( "MyData" );
// create the output record to hold the reply
MyRecord out = new MyRecord() ;
// create and execute the interaction
Interaction ix = con.createInteraction() ;
ix.execute ( spec, in, out );
// extract the reply 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
FunctionName	Provide the name CICS program to call
InteractionVerb	Specifies the communication model.

	<p>One of the constant value:</p> <p><code>javax.resource.cci.InteractionSpec.SYNC_SEND_RECEIVE</code>  <code>javax.resource.cci.InteractionSpec.SYNC_SEND</code>  <code>javax.resource.cci.InteractionSpec.SYNC_RECEIVE</code></p>
ExecutionTimeout	Sets the milliseconds to wait for a reply.
TransactionId	Sets the CICS transaction identifier.
TPN	<p>If set to true the call will use ECI_TPN otherwise ECI_TRANSID. Default value is "true".</p> <p>Refer to the CICS client/server documentation for more information on the difference between ECI_TRANSID and ECI_TPN.</p>
CommAreaLength	Explicitly sets the CommArea length.

**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 executes the program and blocks until the reply is available.

### **`javax.resource.cci.InteractionSpec.SYNC_SEND`**

The interaction sends the requests and returns immediately. The reply will be received asynchronously by the connector and stored in the Interaction instance which sends the request.

### **`javax.resource.cci.InteractionSpec.SYNC_RECEIVE`**

The interaction returns a asynchronously receives reply or throws an exception if no reply is available.

## Further information

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

This product includes software developed by the Apache Software Foundation (<http://www.apache.org/>).