WebSphere® software

IBM

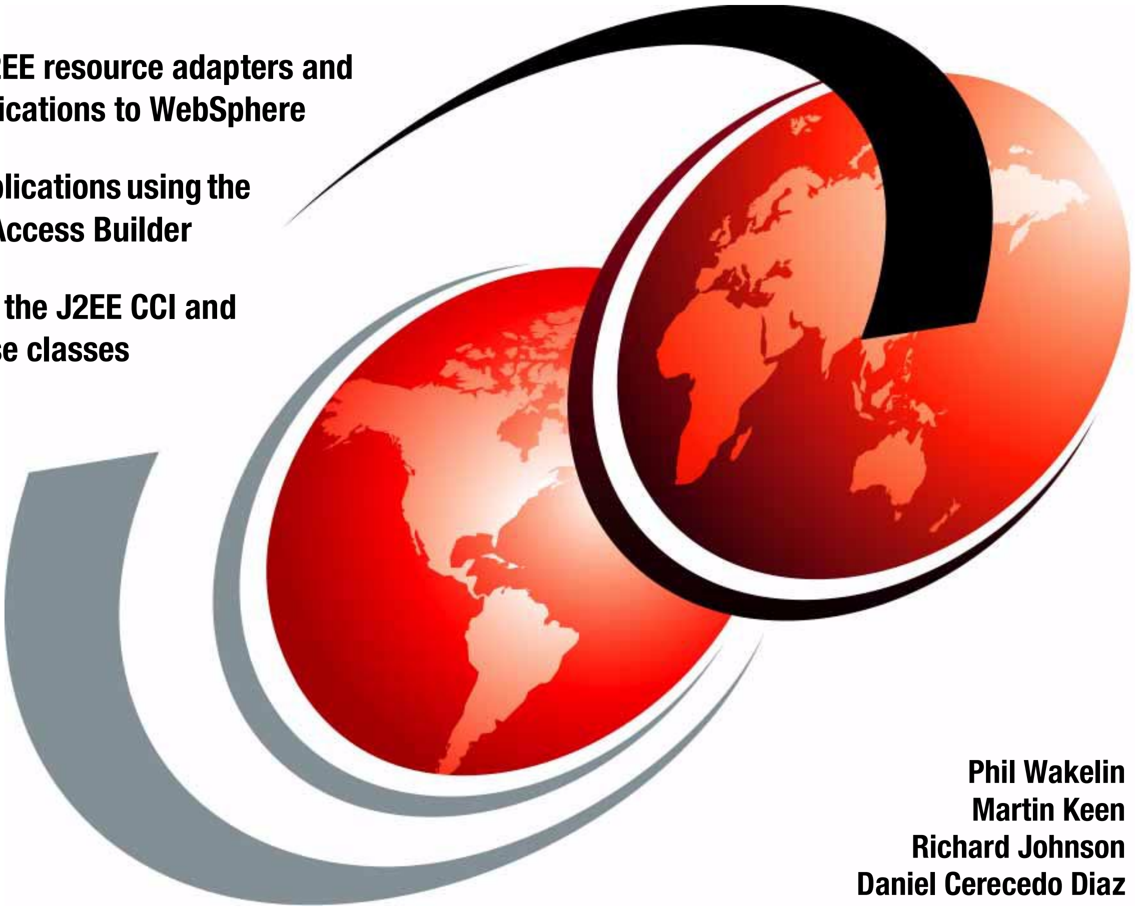
Java Connectors for CICS

Featuring the J2EE Connector Architecture

Use CICS J2EE resource adapters and deploy applications to WebSphere

Develop applications using the Enterprise Access Builder

Understand the J2EE CCI and the CTG base classes



Phil Wakelin
Martin Keen
Richard Johnson
Daniel Cerecedo Diaz

Redbooks

ibm.com/redbooks

International Technical Support Organization

Java Connectors for CICS: Featuring the J2EE Connector Architecture

March 2002

SG24-6401-00

Take Note! Before using this information and the product it supports, be sure to read the general information in “Special notices” on page vii.

First Edition (March 2002)

This edition applies to CICS Transaction Gateway V4.0.1, VisualAge for Java V4.0, WebSphere Studio Application Developer V4, WebSphere Application Server Advanced Edition V4.0.1

Comments may be addressed to:
IBM Corporation, International Technical Support Organization
Dept. QXXE Building 80-E2
650 Harry Road
San Jose, California 95120-6099

Contents

Special notices

IBM trademarks

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

Other company trademarks

Preface

What is the best method for connecting a Java application to CICS? There are a wealth of options that are available, ranging from using the Java class libraries that are shipped with the CICS Transaction Gateway (CTG), to using the Common Client Interface (CCI) component of the Java 2 Enterprise Edition (J2EE) Connector Architecture. There are also important application development choices to make, such as whether to code to an API directly, or to use a tool such as VisualAge for Java's Enterprise Access Builder.

This IBM Redbook examines the strategic Java connection methods for CICS. The focus is on the use of the J2EE Connector Architecture, which is a new Java standard for connecting to legacy Enterprise Information Systems such as CICS. This builds upon the previous IBM Common Connector Framework (CCF) and provides enhanced facilities for deploying into a managed environment, where connection pooling, transactions, and security are managed by a J2EE capable application server such as WebSphere Application Server.

This redbook begins by providing an overview of the CTG, which is the basis for the new CICS J2EE resource adapters provided in V4.0.1 of the CTG. Afterwards, a set of practical programming examples are provided that detail how to build applications using either the existing Java classes offered by the CTG, or using the new J2EE CCI.

Included in this redbook are comprehensive examples of how to develop applications using both the External Call Interface (ECI) for calling COMMAREA based CICS programs, and the External Presentation Interface (EPI) for invoking 3270 based CICS transactions. The usage of the ECIRequest, ESIRequest, EPI support classes provided by the CTG is featured, as well as the CCI, which is required when developing to the J2EE Connector Architecture specification.

Additionally, there is information on how to develop and deploy a full-scale J2EE application using the CCI, which is deployed into the managed environment offered by WebSphere Application Server Advanced Edition V4, using the CICS ECI J2EE resource adapter and the CTG V4.0.1.

All the code developed in the book is packaged into a set of simple samples and is available for download from the ITSO Web sites.

The team that wrote this redbook

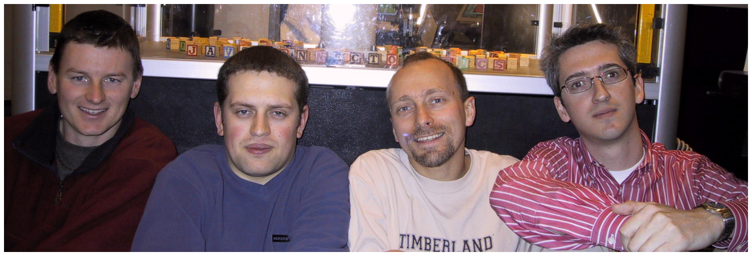
This redbook was produced by a team of specialists from around the world working at the International Technical Support Organization, San Jose Center.

Phil Wakelin is a senior IT specialist at the International Technical Support Organization, San Jose Center. Phil joined IBM in 1990, originally working in the System Test Department at IBM Hursley, on many of the platforms and versions of CICS before joining the Installation Support Center as a pre-sales support specialist for CICS client-server. Phil has been at the ITSO since 1999 and writes and speaks extensively on CICS Web-enablement. He is an IBM Certified Solutions Expert in CICS Web Enablement, and holds a BS degree in Applied Biology from the University of Bath, UK.

Martin Keen is an advisory IT specialist, working as a consultant in Software Group Services at IBM UK's Hursley Laboratory. He writes and teaches extensively in the area of CICS Java Web-enablement, and also provides consultancy for WebSphere Application Server for z/OS. Martin is an IBM Certified Solutions Expert in CICS Web Enablement, and holds a BS in Computer Studies from Southampton Institute.

Richard Johnson is a CICS technical consultant, working in Software Group Services at IBM UK's Hursley Laboratory. His areas of expertise include the CICS Transaction Gateway, Web, Java, EJB, CICS and WebSphere Application Server. He has two years of previous experience at the Functional Test Department of CICS development. He holds a Master's in Chemistry from the University of Oxford.

Daniel Cerecedo Diaz is an IT specialist working in IBM Global Services in Madrid. He has two years of experience in Java and Web development, object-oriented design and development methodologies. He holds a Master's degree in Computer Science from Deusto University. His areas of expertise include Java, XML and related technologies, and WebSphere Business Components Composer.



Thanks to the following people for their contributions to this project:

Chris Smith, Geoff Sharman and Ken Davies, IBM Hursley for supporting this project.

Dave Kelsey, Peter Masters, Jonathan Lawrence, David Radley, Stephen Hurst, Daniel McGinnes, Kate Robinson IBM Hursley, for being a valuable source of reference regarding the CICS Transaction Gateway.

Kevin Sutter and Kevin Kelle IBM Rochester, for great advice on WebSphere.

Mike Andrea and John Green IBM Toronto for advice on VisualAge for Java and associated products.

Carol Shanesy, IBM Dallas for help with CICS 3270 questions, and Dennis Weiand for providing such detailed reviews.

Maritza Dubec of the International Technical Support Organization, for providing editing support.

Notice

This publication is intended to help application developers to design and develop Java applications to access legacy CICS applications. Information in this publication is not intended as the specification of any programming interfaces that are provided by CICS Transaction Server, the CICS Transaction Gateway, or VisualAge for Java. See the PUBLICATIONS section of the IBM Programming Announcement for these products for more information about what publications are considered to be product documentation.

Comments welcome

Your comments are important to us!

We want our Redbooks to be as helpful as possible. Send us your comments about this or other Redbooks in one of the following ways:

- Use the online **Contact us** review redbook form found at:

  `ibm.com/redbooks`

- Send your comments in an Internet note to:

  `redbook@us.ibm.com`

- Mail your comments to the address on page ii.

Part 1

Introduction

In part one, we provide an overview of the Java connectors available for use with CICS, including a detailed description of the CICS Transaction Gateway, and its features and functions. We also provide a summary of the J2EE Connector Architecture and the Common Client Interface.

1

Java connectors for CICS

This chapter gives a broad overview of the Java connectors that are available for use with CICS.

This chapter provides details on the following information:

- CTG base classes
- CTG EPI beans
- CCF CICS connector
- CICS connector for CICS TS
- J2EE connectors, including:
  - J2EE CICS resource adapters
  - WebSphere for z/OS CICSEXCI connector

1.1 CICS Transaction Gateway

The CICS Transaction Gateway (CTG) has a long heritage as a Java connector for CICS, originally being provided as the CICS Gateway for Java, which was available as a free download for use with the CICS Client. Since then the CTG has advanced along with the Java world and now provides three principal interfaces for communication with CICS:

- Base classes
- Common Connector Framework API
- J2EE Common Client Interface

These interfaces, and the other connectors that have also implemented these interfaces, are discussed in the following sections.

For further detail on the features and facilities provided by the CTG, refer to Chapter 2, “CICS Transaction Gateway” on page 9.

1.2 CTG APIs

The CTG provides its own set of base classes, as well as a set of EPI support classes and EPI beans.

CTG base classes

The CTG provides a set of base classes that offer a simple, but low-level interface to CICS. They are relatively easy to use, but they require a reasonable understanding of CICS to implement. There are three request classes that are part of the CTG base classes:

ECIRequest	This provides a Java interface to the ECI, and it is used for calling COMMAREA based CICS applications. For details on how to use the `ECIRequest` class to develop test applications, refer to Chapter 4, “ECI and ESI applications” on page 35.
EPIRequest	This provides a Java interface to the EPI, and it is used for invoking 3270 based transactions. Due to its low-level nature, using it for developing EPI applications requires a strong knowledge of CICS and 3270 datastreams. For this reason, there are no examples in this book, and it is advised that you use the EPI support classes instead.

ESIRequest This provides a Java interface to the ESI that invokes the Password Expiration Management (PEM) functions in CICS. This allows for the verification and changing of passwords. For details on how to use the `ESIRequest` class to develop test applications, refer to Chapter 4, "ECI and ESI applications" on page 35.

Restriction: Note that the CCF and J2EE connectors do not provide an alternative interface to the `ESIRequest` class (as they do for the EPI and ECI interfaces.)

EPI support classes

The EPI support classes provide high-level constructs for handling 3270 data streams. You do not need a detailed knowledge of 3270 data streams to use these classes, and they are considerably easier to use than the `EPIRequest` class.

A wide range of classes is provided including `AID`, `FieldData`, `Screen`, `Terminal`, `Map` and `MapData`. These are used to represent the interface to a CICS 3270 terminal, and the resulting 3270 response. For details on how we used these support classes to develop test applications, refer to Chapter 7, "EPI support classes" on page 155.

EPI beans

The EPI beans are based on the EPI support classes and JavaBean development environment. They allow you to create EPI applications in a visual development environment, using one of the visual application builder tools, such as VisualAge for Java.

For further information on using the EPI beans refer to the IBM Redbook, *CICS Transaction Gateway and More CICS Clients Unmasked*, SG24-5277.

1.3 CICS CCF connector

The IBM Common Connector Framework (CCF) is an architecture that provides Java developers with a standardized set of interfaces to access Enterprise Information Systems (EIS). The CCF connector classes implement the CCF interfaces and programming model. The CCF is comprised of three core components:

- **CCF classes**
  The principal component are the `CICSConnectionSpec`, the `ECIInteractionSpec`, and the `EPIInteractionSpec`, which are used to control the interaction with the CICS EIS.
- **Java Record Framework**
  The Java Record Framework is used to build `Record` objects to wrap data structures such as a COBOL COMMAREA or a BMS map. It also provides a powerful set of marshalling options for the encoding of data, and for retrieving fields from the COMMAREA or BMS datastreams.
- **Enterprise Access Builder (EAB)**
  The EAB is a function of VisualAge for Java. Through a set of SmartGuides, the EAB allows *Command beans* to be developed that encapsulate the CCF classes, and the Java Record Framework `Records`.

The CICS Transaction Gateway provides the `CICSConnectionSpec`, `ECIInteractionSpec`, and `EPIInteractionSpec` classes that implement the CCF connector for CICS. These classes are provided in the CTG class library `ctgclient.jar`, and are also provided by VisualAge for Java, and by CICS Transaction Server for z/OS V2 (CICS TS V2). For further information on the CCF, refer to the redbook *CCF Connectors and Databases Connections using WebSphere Advanced Edition*, SG24-5514.

The J2EE Connector Architecture has now replaced the CCF as the strategic way of connecting to EIS. If you are developing new applications that connect to EIS, it is strongly recommend that you use the J2EE Connector Architecture instead of the CCF.

1.4 CICS Connector for CICS TS

In CICS TS V2.1 and V2.2, a Java program, or an enterprise bean running within CICS, can use the new *CICS Connector for CICS TS* to link to any CICS program with a COMMAREA interface. This connector runs within the CICS region and offers the same CCF connector interface as provided by the CTG. However, when using the CICS Connector for CICS TS, the CTG is not required, because the connector runs within the CICS TS V2 environment.

Since the CICS connector for CICS TS is based on the technology in the CTG class library `ctgclient.jar`, it is also possible to develop and deploy Java applications into CICS TS V2 that use the CTG `JavaGateway` and `ECIRequest` objects, instead of using the CCF `CICSConnectionSpec` and `ECIInteractionSpec` objects.

Note: As an alternative to the CICS connector for CICS TS, you can use the `link()` method provided in the JCICS `ibm.cics.server.Program` class. This is a lower-level interface, and if used in a session bean, it will prevent the session bean from being deployed in an environment other than CICS.

For more information on the CICS connector for CICS TS, refer to the IBM Redbook, *Enterprise JavaBeans for z/OS and OS/390 CICS Transaction Server V2.1,* SG24-6284.

1.5 J2EE connectors

The J2EE Connector Architecture defines a standard architecture for connecting the Java 2 Platform Enterprise Edition (J2EE) platform to a heterogeneous EIS such as CICS. Java applications interact with resource adapters using the Common Client Interface (CCI) (which is largely based on the CCF) but, it is a standard that is open to the entire Java community. For further details on the J2EE connector architecture, refer to Chapter 3, “CICS and the J2EE Connector Architecture” on page 17.

The connector architecture enables an EIS vendor to provide a standard *resource adapter* for its EIS. A resource adapter is the middle-tier between a Java application and an EIS, which permits the Java application to connect to the EIS.

There are three resource adapters currently available for use with CICS:

- **CICS ECI resource adapter** and **EPI resource adapter**

  The CTG V4.0.1 offers an ECI and EPI resource adapter. Together they provide the ability to call both COMMAREA based CICS programs and start 3270 based CICS transaction from a J2EE environment. They can be used in any Java application as a non-managed environment, or from a session bean in a managed environment, such as in WebSphere Application Server Advanced Edition V4.

These two resource adapters, along with WebSphere Application Server Advanced Edition, are the main subjects of this redbook. Further details on how to develop ECI based applications are provided in Part 2, “Connecting to COMMAREA based CICS programs” on page 33, and how to develop EPI based applications in Part 3, “Connecting to 3270 based CICS transactions” on page 153.

These connectors are based on the ECI and EPI Java interfaces, previously provided by the CTG. Since the ECI is a one phase commit protocol calls using the ECI resource adapter can only be transactional if the Application Server supports the Local Transaction interface. For further details refer to 3.3.2, “Transaction management” on page 25.

Attention: Support for the CICS ECI resource adapter running in WebSphere V4 for z/OS is currently *not* available with the CTG, but will be made available in V4.0.2 of the CTG. Since this connector will use the CICS External Call Interface (EXCI) to establish a connection to a CICS AOR, the calls to a CICS program can be transactional, as long as the AOR resides on the same MVS image as the CTG and WebSphere. Updates to recoverable resources will be will be coordinated using MVS Resource Recovery Services (RRS)

- **WebSphere for z/OS CICSEXCI connector**

  This connector comes with WebSphere for z/OS V4.0.1, and allows enterprise beans running in a WebSphere for z/OS J2EE server region to make calls to COMMAREA based CICS programs. This connector uses the EXCI to establish connection to a CICS AOR (which must reside on the same MVS image as the WebSphere for z/OS J2EE server.) The calls to CICS programs are transactional. Updates to recoverable resources will be coordinated using MVS Resource Recovery Services (RRS).

Note: The WebSphere for z/OS CICSEXCI connector is a beta J2EE connector, and its function will be superseded by the CICS ECI resource adapter when it is made available.

2

CICS Transaction Gateway

This chapter gives a broad overview of the CICS Transaction Gateway (CTG) and the features and functions it provides.

The following topics are discussed:

- CTG interfaces
- Gateway daemon
- Client daemon
- Configuration tool
- Terminal Servlet

2.1 CTG: interfaces

All the principal interfaces provided by the CTG fall into one of three categories, based on the function being invoked in CICS:

- External Call Interface (ECI)
- External Presentation Interface (EPI)
- External Security Interface (ESI)

External Call Interface

The ECI is used for calling COMMAREA based CICS applications. The COMMAREA is the buffer that is used for passing the data between the client and the CICS server. CICS sees the client request as just another distributed program link (DPL) request.

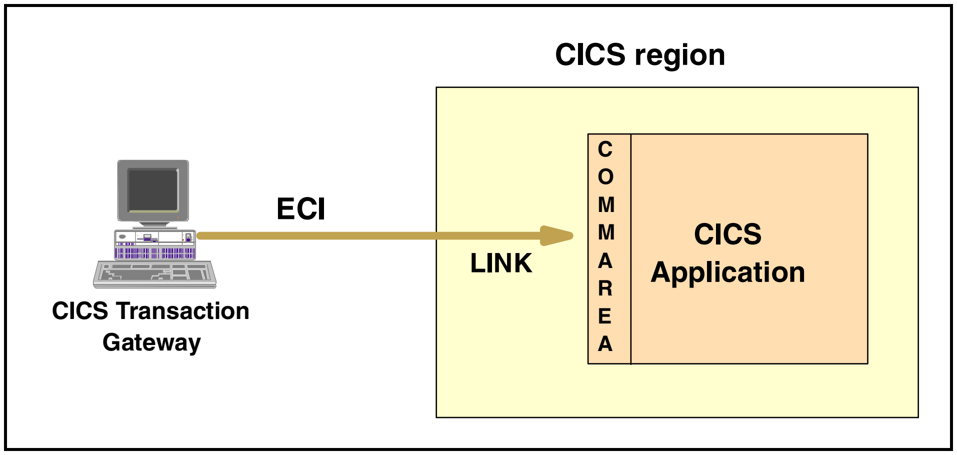


Figure 2-1 External Call Interface

For further details on programming with the ECI refer to Chapter 4, “ECI and ESI applications” on page 35.

External Presentation Interface

The EPI is used for invoking 3270 based transactions. A terminal is installed in CICS, and CICS sees the request as running on a remote terminals controlled by the CTG. For further details on programming with the EPI refer to Chapter 7, “EPI support classes” on page 155.

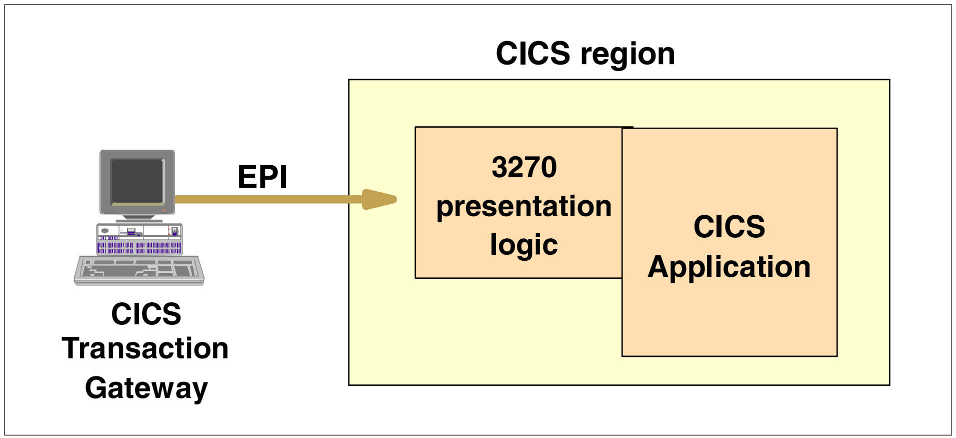


Figure 2-2 External Presentation Interface

External Security Interface

The ESI is used for verifying and changing of the user ID and password information held in the CICS External Security Manager (ESM), such as RACF. It is based on the CICS Password Expiration Management (PEM) function. For further details on programming with the ESI refer to 4.4, “ESI calls” on page 53.

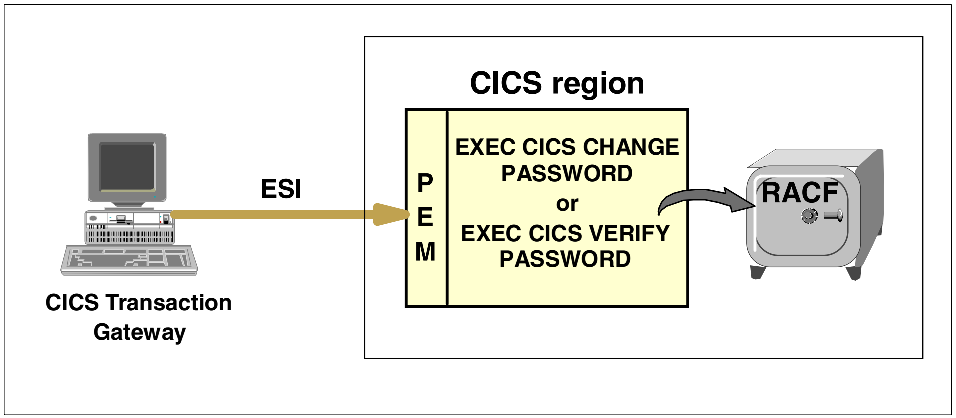


Figure 2-3 External Security Interface

2.2 CTG: infrastructure

The CICS Transaction Gateway (CTG) is a set of client and server software components that allow a Java application to invoke services in a CICS region. The Java application can be an applet, a servlet, an enterprise bean, or any other Java application (Figure 2-4).

The latest edition of the CTG is V4.01, and the currently supported platforms are OS/390, Linux for S/390, AIX, HP-UX, Sun Solaris, Windows NT, and Windows 2000.

The CTG is supported for use with CICS/ESA V4.1, CICS/VSE 2.3 and CICS Transaction Server for VSE/ESA V1, but only if the CTG runs on a distributed platform. For use with CICS Transaction Server for OS/390 or CICS Transaction Server for z/OS V2, the CTG can run on OS/390 or a distributed platform.

For product information on using CTG, refer to the *CICS Transaction Gateway Administration Guides*. For information on configuring the CTG, refer to *CICS Transaction Gateway V3.1, The WebSphere Connector for CICS*, SG24-6133.

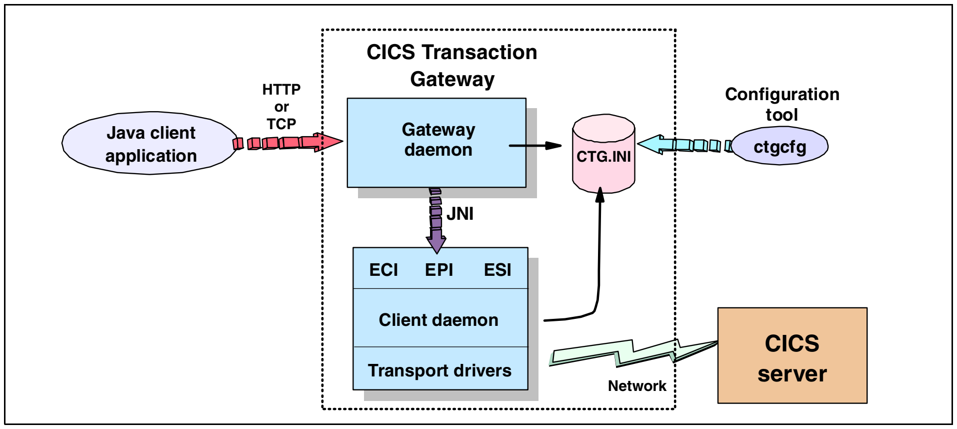


Figure 2-4   CICS Transaction Gateway components

The CICS Transaction Gateway consists of the following principal components:

- Gateway daemon
- Client daemon
- Configuration tool (ctgcfg)
- Terminal servlet
- Java class library

2.2.1 Gateway daemon

The gateway daemon is a long-running process that functions as a server to network-attached Java client applications (such as applets or remote applications) by listening on a specified TCP/IP port. The CTG supports four different CTG *network protocols* (*TCP, SSL, HTTP,* or *HTTPS*); each of which requires a different CTG *protocol handler* to be configured to listen for requests (Figure 2-5).

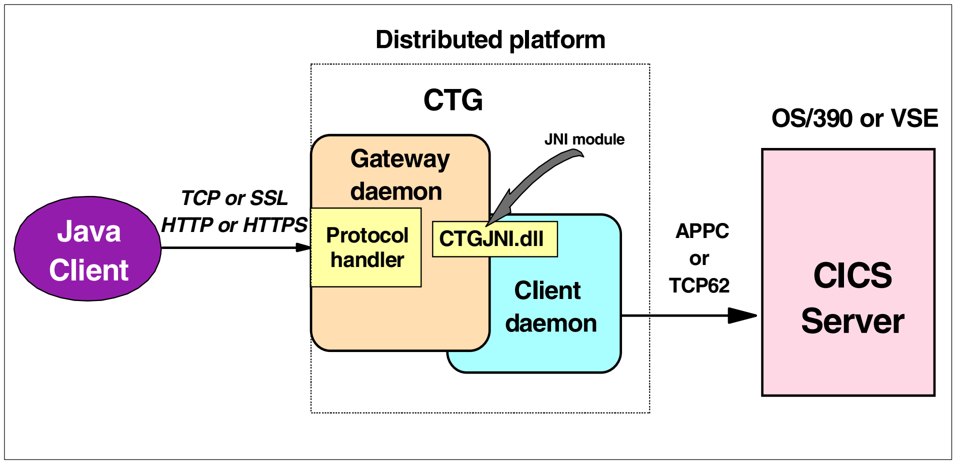


Figure 2-5 CICS Transaction Gateway: distributed platform

The structure of the gateway daemon is slightly different on OS/390 and on distributed platforms. On distributed platforms (including Linux for S/390), the CTG provides equivalent functions to that provided by the CICS Universal Client. There are three basic interfaces that are provided to Java client applications:

External Call Interface (ECI)	A call interface to COMMAREA based CICS applications
External Presentation Interface (EPI)	An API to invoke 3270 based transactions
External Security Interface (ESI)	Allows password expiration management (PEM) functions to be invoked in CICS, in order to verify and change user IDs and passwords

On OS/390, the External CICS Interface (EXCI) is used in place of the client daemon, and provides access to COMMAREA based CICS programs. Consequently, the EPI and ESI interfaces are not available with the OS/390 CTG. There are a few differences between the OS/390 ECI support, and the ECI support using distributed platforms (see Figure 2-6.)

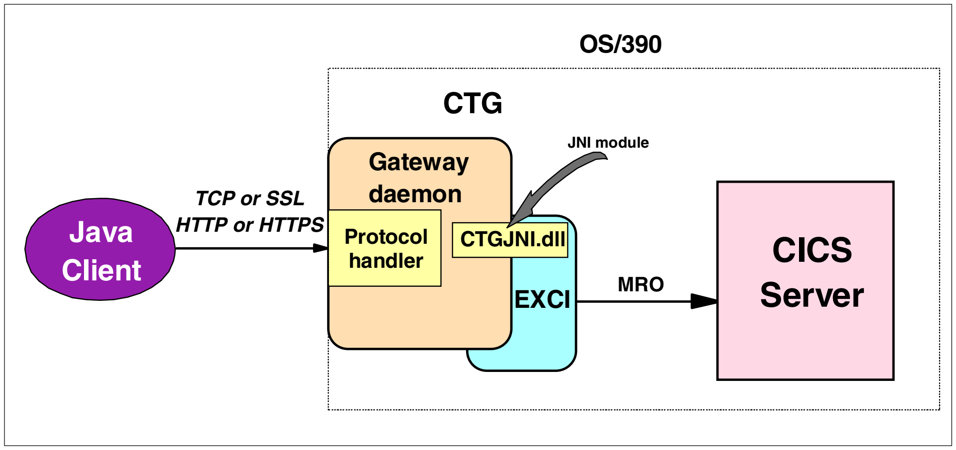


Figure 2-6 CICS Transaction Gateway: OS/390

The primary differences in the ECI support offered when the CTG is running on OS/390, are as follows:

- When using asynchronous calls, specific reply solicitation calls are not supported.
- The user ID and password flowed on ECI requests are verified within the CTG with RACF; afterwards the verified user ID is then flown onto CICS.
- The `ECI_Listsytems` function does not return the list of defined servers, since any CICS region within the OS/390 Parallel Sysplex can be reached.

> **Note**: The gateway daemon is not usually required when a Java application executes on the same machine as where the CTG is installed. In this situation, the CTG *local:* protocol can be used, which directly invokes the underlying transport mechanism using native code.

2.2.2 Client daemon

The CTG client daemon is an integral part of the CTG on all distributed platforms. It provides the CICS client-server connectivity using the same technology as previously provided by the CICS Universal Client. On distributed platforms, connections to the following CICS servers are supported

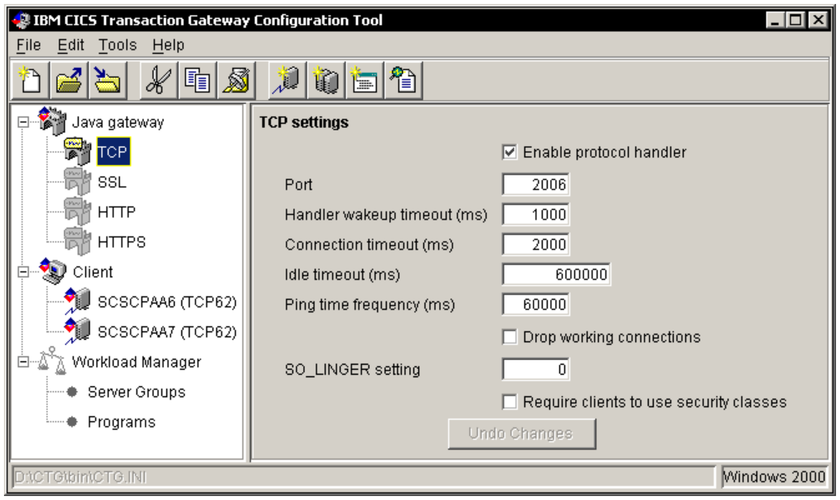
- APPC connections from Windows and AIX platforms to all CICS platforms
- TCP62 (LU6.2 over IP) connections to CICS/ESA V4.1, CICS TS V1.2 and CICS TS V1.3 for OS/390, and CICS TS for z/OS V2
- TCP/IP connections to the TXSeries CICS Servers (AIX, Sun Solaris, Windows NT, and HP-UX), CICS TS for z/OS V2.2 and CICS TS for VSE/ESA V1.1.1 and CICS OS/2 Transaction Server

For further details on supported platforms and required service levels, refer to the appropriate announcement letters available at the following URL:

http://www-4.ibm.com/software/ts/cics/announce/

2.2.3 Configuration tool

The configuration tool (ctgcfg) is a Java-based graphical user interface (GUI) supplied by the CTG on all platforms. It is used to configure the gateway daemon and client daemon properties, which are stored in the `CTG.INI` file. In previous versions of the CTG, the `cicscli.ini` and `gateway.properties` files were used to store the configuration parameters now stored in `CTG.INI`. Figure 2-7 illustrates the graphical user interface of the configuration tool.



TCP settings

[✓] Enable protocol handler

Port	2006
Handler wakeup timeout (ms)	1000
Connection timeout (ms)	2000
Idle timeout (ms)	600000
Ping time frequency (ms)	60000

[ ] Drop working connections

SO_LINGER setting 0

[ ] Require clients to use security classes

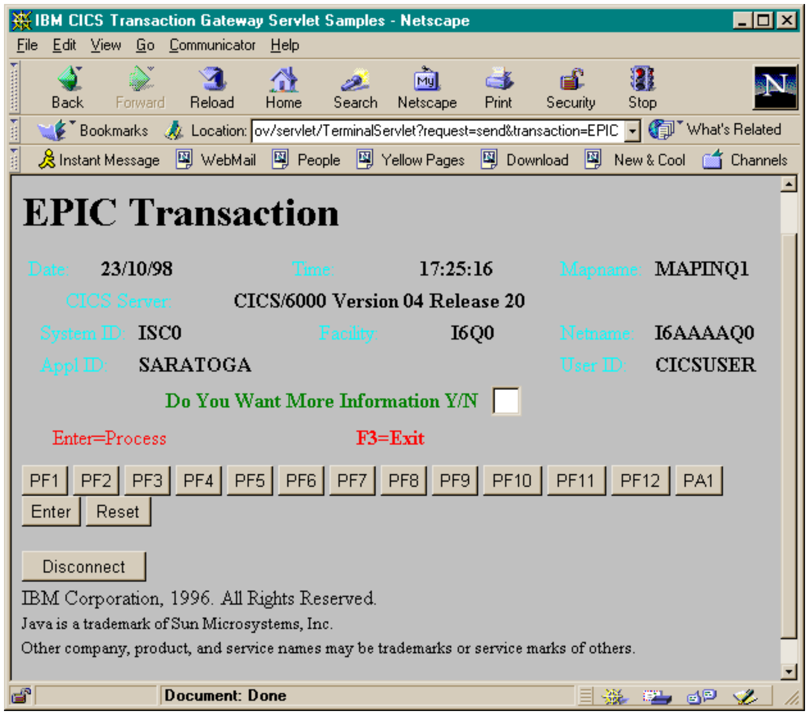
D:\CTG\bin\CTG.INI   Windows 2000

Figure 2-7   CTG configuration tool

2.2.4 Terminal Servlet

The Terminal Servlet is a supplied Java servlet that allows you to use a Web browser as an emulator for a 3270 CICS application. It dynamically converts 3270 output into HTML for display on a Web browser, and is a non-programmatic solution for Web-enabling 3270 applications.

The Terminal Servlet is similar in function to the CICS supplied 3270 Web bridge, in that it provides a turn-key solution to Web-enabling 3270 based applications. It also has the ability to use the same HTML templates as the 3270 Web bridge to display the output of CICS BMS maps as HTML forms. In addition, it provides a basic terminal emulation capability, and the ability to use *server-side includes* to display information from a CICS screen. The HTML template interface offered by the Terminal Servlet is shown in Figure 2-8.


IBM CICS Transaction Gateway Servlet Samples - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Stop
Bookmarks
Instant Message WebMail People Yellow Pages Download New & Cool Channels
What's Related
PF1 PF2 PF3 PF4 PF5 PF6 PF7 PF8 PF9 PF10 PF11 PF12 PA1
Enter Reset
Disconnect
Document: Done

Location: ov/servlet/TerminalServlet?request=send&transaction=EPIC

EPIC Transaction

Date: 23/10/98 Time: 17:25:16 Mapname: MAPINQ1

CICS Server: CICS/6000 Version 04 Release 20

System ID: ISC0 Facility: I6Q0 Netname: I6AAAAQ0

Appl ID: SARATOGA User ID: CICSUSER

Do You Want More Information Y/N

Enter=Process F3=Exit

IBM Corporation, 1996. All Rights Reserved.

Java is a trademark of Sun Microsystems, Inc.

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

Figure 2-8 CTG Terminal Servlet

3

CICS and the J2EE Connector Architecture

This chapter describes the J2EE Connector Architecture and how it can be used to access CICS applications. It introduces the J2EE Connector Architecture, and explores some of its components. This chapter closes with a discussion of the CICS specific resource adapters that are provided with the CICS Transaction Gateway.

This chapter covers the following subjects:

- J2EE Connector Architecture, including an introduction to this specification
- Common Client Interface, describing the client API for working with the J2EE Connector Architecture
- System contracts that can automate tasks, such as transactionality and security
- CICS resource adapters provided by the CICS Transaction Gateway

3.1 J2EE Connector Architecture

The J2EE Connector Architecture defines a standard architecture for connecting the Java 2 Platform Enterprise Edition (J2EE) platform to heterogeneous Enterprise Information Systems (EIS). Examples of EIS include transaction processing systems, such as CICS Transaction Server, and Enterprise Resource Planning systems such as SAP.

Note: The complete J2EE Connector Architecture specification defined by the Java Community Process can be downloaded at:

`http://java.sun.com/j2ee/download.html#connectorspec`

The connector architecture enables an EIS vendor to provide a standard *resource adapter* for its EIS. A resource adapter is a middle-tier between a Java application and an EIS, and permits the Java application to connect to the EIS. The types of Java applications include applets, servlets, and enterprise beans. A resource adapter plugs in to any application server supporting the J2EE Connector Architecture.

An application server vendor only needs to extend its system once to support the J2EE Connector Architecture, and is then assured of connectivity to multiple EISs. Likewise, an EIS vendor provides a standard resource adapter, and it has the capability to plug-in to any application server that supports the J2EE Connector Architecture. This is shown in Figure 3-1.

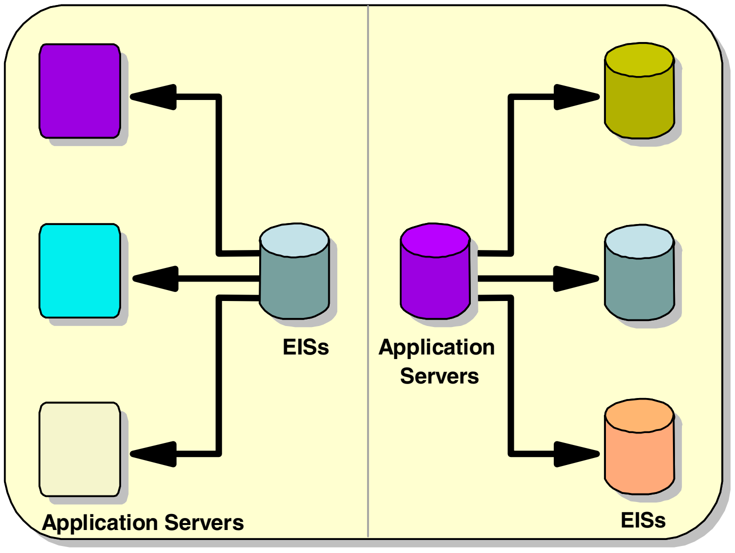


Figure 3-1 The role of resource adapters

3.1.1 Components of the J2EE Connector Architecture

Version 1.0 of the J2EE Connector Architecture defines a number of components that make up this architecture (see Figure 3-2):

- **Common Client Interface (CCI)**

  The CCI defines a common API for interacting with resource adapters. It is independent of a specific EIS. A Java developer communicates to the resource adapter using this API. See 3.2.1, “CCI overview” on page 22.

- **System contracts**

  A set of system-level contracts between an application server and EIS. These extend the application server to provide:

  – Connection management
  – Transaction management
  – Security management

  These system contracts are transparent to the application developer, meaning, they do not implement these services themselves.

- **Resource adapter deployment and packaging**

  A resource adapter provider develops a set of Java interfaces/classes as part of its implementation of a resource adapter. The Java interfaces/classes are packaged together with a deployment descriptor to create a *Resource Adapter Archive* (represented by a file with an extension of `rar`). This Resource Adapter Module is used to deploy the resource adapter into the application server.

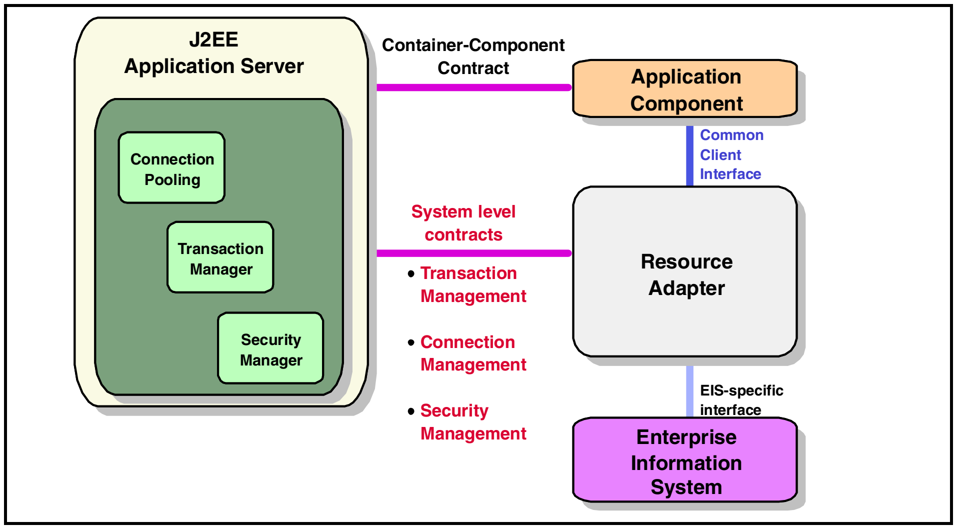


Figure 3-2 J2EE Connector Architecture components

3.1.2 Managed and non-managed environments

There are two different types of environments that a Java application using J2EE connectors can run in:

- **Managed environment**

  The Java application accesses a resource adapter through an application server such as WebSphere Application Server. Management of connections, transactions, and security is provided by this application server. The Java application developer does not have to code this management manually.

- **Non-managed environment**

  In a non-managed environment you do not have to use an application server. Instead, the Java application directly uses the resource adapter to access an EIS. In this case *management* of connections, transactions, and security must be handled manually by the Java application.

Typically, when using a resource adapter for the first time, it is advisable to develop a Java application for a non-managed environment first. This allows you to become familiar with the resource adapter without having to deploy your Java application to an application server. Once you are sure that the Java application is working correctly, you can modify it to use a managed environment, and deploy it to an application server. (See Chapter 5, "CCI applications: ECI based" on page 71, and Chapter 6, "CCI applications in a managed environment" on page 111).

3.1.3 The Common Connector Framework

Before the existence of the J2EE Connector Architecture, IBM recognized a need for a common way to connect to EIS. The IBM Common Connector Framework (CCF) was introduced to provide this function.

Before the CCF, a Java application developer wishing to connect to multiple EISs would have to learn an API specific to each EIS connector. Now, the CCF offers a common API for every EIS connector supported within this framework. Additionally, tools such as VisualAge for Java Enterprise Edition can generate code adhering to the CCF specification, allowing Java developers to create connections to an EIS without having to write a single line of code.

However, the CCF has a limitation; it is not an open standard. The only connectors supported by the CCF are those IBM chooses to add. The J2EE Connector Architecture provides similar function to the CCF, but is an open specification that can be implemented by anyone.

IBM played a significant role in the development of the J2EE Connector Architecture specification, and it is based heavily on the CCF. A developer already familiar with the CCF will be able to become productive quickly using J2EE resource adapters.

The J2EE Connector Architecture has now replaced the CCF as the strategic way of connecting to an EIS. If you are developing new applications that connect to EISs, we strongly recommend using the J2EE Connector Architecture instead of the CCF.

VisualAge for Java Enterprise Edition V4 provides tooling that can migrate existing CCF based applications to the J2EE Connector Architecture standard. Additionally, the same tooling that generated CCF code automatically can be used to generate code adhering to the J2EE Connector Architecture in VisualAge for Java V4.

3.2 Common Client Interface

The Common Client Interface (CCI) defines a standard client API so that application components can access multiple resource adapters (Figure 3-3). This API can be used directly, or enterprise application integration frameworks can be used to generate EIS access code for the developer. The CCI is designed to be an EIS independent API, so that an enterprise application development tool can produce code for any J2EE compliant resource adapter that implements the CCI interface. Such tools include the VisualAge for Java Enterprise Access Builder.

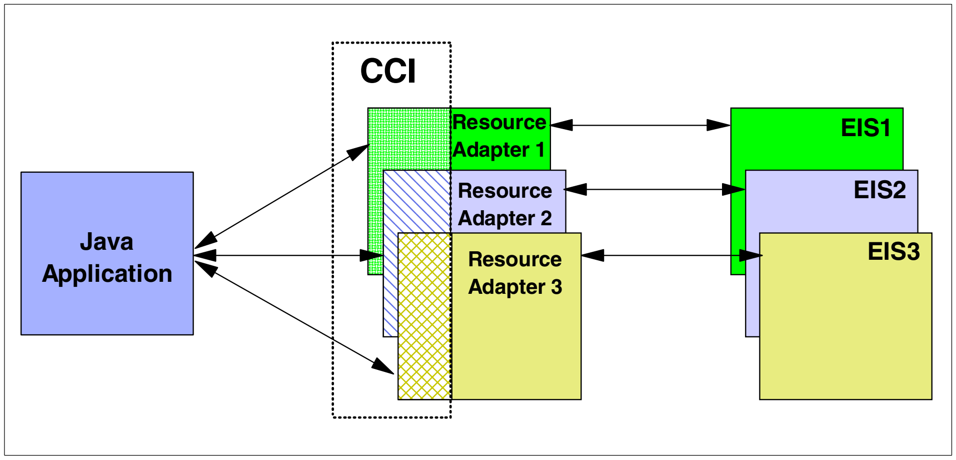


Figure 3-3   CCI with multiple resource adapters

3.2.1 CCI overview

The CCI has the following characteristics:

- It is independent of a specific EIS. It forms a base-level API for EIS access on which higher-level functionality, specific to an EIS, can be built.
- It provides an API that is consistent with other APIs in the J2EE platform, such as JDBC.
- It is targeted primarily towards application development tools and enterprise application integration frameworks, rather than Java developers using the CCI API directly.

One goal of the CCI is to complement, rather than replace, the JDBC API. The CCI programming model matches up with the JDBC programming model, but both APIs serve the following different purposes:

- The JDBC API is used to access relational databases.
- The CCI API is used to access an EIS (which are not databases).

3.2.2 Using the CCI classes

The CCI provides two distinctive types of classes:

- Framework classes
- Input and output classes

Framework classes

These are used to define the communication with the resource adapter. The Framework classes enable you to:

- Connect to a resource adapter and disconnect
- Specify the interaction to make to an EIS
- Execute the interaction to an EIS; passing input and retrieving output

Figure 3-4 is an example of simple code that shows how to interact with an EIS using the CCI API.

```
1 ConnectionFactory cf = <lookup from JNDI namespace>
  Connection connection = cf.getConnection();
2 Interaction interaction = connection.createInteraction();
3 interaction.execute(<input and output data>);
4 interaction.close();
  connection.close();
```

Figure 3-4 A basic CCI sample

Note: The interfaces used in the example in Figure 3-4 all belong to the `javax.resource.cci` package.

The following steps summarizes the logic of the code in Figure 3-4:

1. **Create a `Connection` object to connect to the EIS**

   The `ConnectionFactory` object is used to generate a `Connection` object. The `ConnectionFactory` retrieves information on how to connect to the EIS by performing a JNDI lookup. This information might include the location of the resource adapter to use, and the name of the EIS to connect to.

2. **Create an `Interaction` object**

   An `Interaction` object is used to perform specific interactions with an EIS across a connection. The `Connection` object is used to create an `Interaction`.

3. **Execute the `Interaction` with the EIS**

   The `Interaction` object is used to execute the interaction with the EIS. This will make a call to the resource adapter over the specified connection, which will in turn, make a call to the EIS.

4. **Close the `Interaction` and `Connection`**

Input and output classes

Input and output classes are used to pass or retrieve specific information to both the framework classes, and to the EIS. There are three types of input and output classes:

- **ConnectionSpec objects**

  Actions that a resource adapter will perform on a connection to a EIS. For example, the user ID and password to flow on a connection.

- **InteractionSpec objects**

  Actions that a resource adapter will perform on an interaction with a EIS. For example, the program name to run in the EIS.

- **Record objects**

  Stores the input and output data to be used during the interaction with a EIS. The input Record will contain data to pass to the EIS, and the output Record will be used to store data generated by the EIS.

A resource adapter will implement the spec objects to create getter and setter methods that have specific relevance to an EIS.

Figure 3-5 shows a more complete overview of using CCI, with input and output classes included for a fictional *xyz* resource adapter.

```
ConnectionFactory cf = <lookup from JNDI namespace>
xyzConnectionSpec cs = new xyzConnectionSpec();
cs.setXXX();
Connection connection = cf.getConnection();
Interaction interaction = connection.createConnection();
xyzInteractionSpec is = new xyzInteractionSpec();
is.setXXX();
RecordImpl in = new RecordImpl();
RecordImpl out = new RecordImpl();

interaction.execute(is, in, out);
interaction.close();
connection.close();
```

Figure 3-5 A CCI sample with input and output classes

For information about using CCI with the CICS ECI resource adapter, refer to Chapter 5, "CCI applications: ECI based" on page 71. For information about using the CCI with the CICS EPI resource adapter, see Chapter 8, "CCI applications: EPI based" on page 181.

3.3 System contracts

To achieve the ease of interaction between the application server and EIS, the J2EE Connector Architecture defines a set of system contracts. The application server uses a resource adapter to support these contracts. The resource adapter implements the system contracts to collaborate with the application server and uses an EIS-specific API to communicate with the EIS. Thus, a resource adapter is specific to an EIS, however, because it implements the system contracts, it can be plugged into any J2EE compliant application server.

3.3.1 Connection management

The connection management contract gives an application component a connection to an EIS. To deliver performance and scalability, the connection management contract should support connection pooling and management.

When retrieving data from an EIS, a large portion of the time (from making the connection, to receiving the data, and closing the connection) is in the creation of the connection itself. Connection pooling alleviates this bottleneck. When you call for a connection, you are passed a handle to the next available connection that is in a ready-to-use state. This increases performance greatly by removing the actual connection time, and scalability is handled by predefining as many connections in the pool as you need.

Connection pooling is a *quality of service* offered by the application server. An application server uses the connection management contract to implement a connection pooling mechanism in its own implementation-specific way.

3.3.2 Transaction management

Before discussing transaction management, the following terms should be defined:

Resource manager	The resource adapter and underlying EIS. It may participate in transactions that are externally controlled by a transaction manager.
Transaction manager	Controls and coordinates transactions across multiple resource managers

A resource manager has three options for supporting transactions:

- **No support**

  The resource manager does not support transactions.
- **Local transactions**

  These are transactions that are managed internally by the resource manager. The coordination of such transactions involves no external transaction manager.
- **Global transactions**

  There are multiple resources managers involved, and an external transaction manager must be used to coordinate the transaction using two-phase commit.

Local transactions

These transactions are managed internally by the resource manager without the need for an external transaction manager, and can be utilized when only one resource manager is involved. Local transactions only support one-phase commit, because they only reference one EIS.

To support local transactions, the resource adapter must implement the `javax.resource.spi.LocalTransaction` interface. If the resource adapter supports the CCI, then it will also send a number of transaction events to the application server.

The application server is also required to implement the interface `javax.resource.spi.ConnectionEventListener`, which among other events, allows the application server to hear and react to the following local transaction events:

- LOCAL_TRANSACTION_STARTED
- LOCAL_TRANSACTION_COMMITTED
- LOCAL_TRANSACTION_ROLLEDBACK

By listening for these events, the application server can do various functions such as local transaction cleanup.

Global transactions

Global transactions are also referred to in the J2EE Connector Architecture specification as JTA transactions, and they are supported by the resource adapter implementing the `javax.transaction.xa.XAResource` interface.

In a managed environment, the application server uses a transaction manager to coordinate the transaction. The application server will provide the following functions:

- Inform the transaction manager when a transaction begins.
- Perform the work of the transaction.
- Tell the transaction manager to commit the transaction.

The transaction manager uses the `XAResource` interface of the resource adapters to coordinate the two-phase commit process across multiple resource managers. Two-phase commit works as follows: (See Figure 3-6.)

1. In phase one, the transaction manager asks all resource managers to *prepare to commit* their work. If a resource manager can commit its work, it replies affirmatively, and hardens its recoverable data to permanent storage. A negative reply reports an inability to commit for any reason.
2. In phase two, the transaction manager directs all resource managers either to *commit* or *rollback* work done on behalf of the global transaction, based on the replies from phase one.

In a non-managed environment, the Java application is responsible for managing transactions, through the local transaction interface (providing that the resource adapter supports this). By using the managed environment, the programmer does not even need to think about managing the transaction, because the transaction manager is one of the *quality of services* provided.

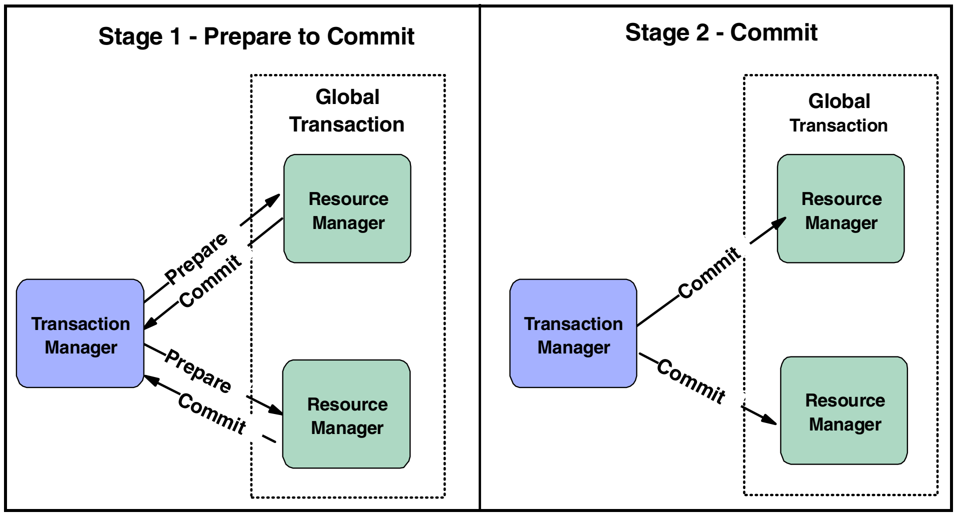


Figure 3-6 Global transactions

If the resource adapter does implement *XAResource* (and does support global transactions) it must also implement support for one-phase commit. This allows the transaction manager to do one-phase commit optimization.

Local transaction optimization

Local transaction optimization is forcing the use of one-phase commit in the situation when two-phase commit is not needed for a global transaction. This is when only one resource manager was referenced, so two-phase commit is an unnecessary overhead.

When the application server needs to do a global transaction, it informs the transaction manager of its intention to begin a transaction. The application server then performs the necessary operations, and when finished, informs the transaction manager to commit the transaction. The transaction manager now has the ability to do one-phase optimization. If the EIS referenced is only one, the transaction manager skips the prepare statement, and goes straight to commit or rollback. WebSphere Application Server Advanced Edition V4 does not currently provide this support.

Last resource optimization

It is possible for a resource adapter that does not implement the `XAResource` interface to participate in a global transaction using last resource optimization (also known as last-agent optimization). This allows the use of a single one-phase commit resource in a global transaction, along with any number of two-phase commit resources. At transaction commit, the two-phase commit resources will first be prepared. If this is successful, the one-phase commit resource will be called to commit, followed by the call to commit for two-phase commit resources. This is shown in Figure 3-7.

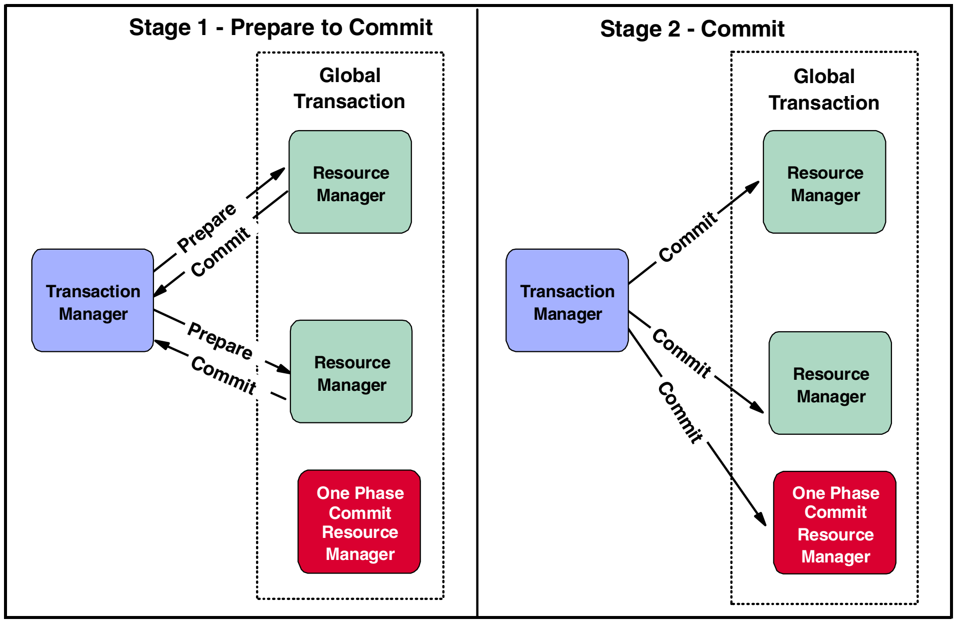


Figure 3-7 Last resource optimization

Last resource optimization falls out of the scope of the J2EE Connection Architecture specification. It is up to each application server to decide if it is supported. Currently, WebSphere Application Server Advanced Edition V4 does not provide this support.

3.3.3 Security management

The J2EE Connector Architecture security contract extends the J2EE security model to provide secure connections to EIS. To create a connection to an EIS, there must be some form of signing on to the EIS, to authenticate the connection requester. Re-authentication can also take place if supported by the EIS. This occurs when the security context is changed after a connection is made. (For example, connection pooling could cause a re-authentication when the connection is redistributed.)

Performing the signon generally involves one or more of the following steps:

1. Determine the resource principal under whose security context the connection will be made.
2. Authenticate the resource principal.
3. Establish secure communications.
4. Determine authorization (access control).

3.4 CICS resource adapters

The CICS Transaction Gateway provides two CICS resource adapters:

- **ECI resource adapter**

  For making calls to CICS COMMAREA-based programs
- **EPI resource adapter**

  For making calls to CICS 3270-based transactions

The CICS Transaction Gateway provides implementations of the CCI interfaces for these two resource adapters in the `com.ibm.connector2.cics` package. These are CICS specific implementations of CCI interfaces. For instance, the `javax.resource.cci.ConnectionSpec` interface has been implemented by the `com.ibm.connector2.cics.ECIConnectionSpec` class. This contains methods relating specifically to the CICS resource adapters, such as `setCommareaLength()` and `setFunctionName()`.

For more information on the CICS Transaction Gateway, refer to Chapter 1, “Java connectors for CICS” on page 3.

3.4.1 ECI resource adapter

The CICS ECI resource adapter uses the External Call Interface (ECI) of the CICS Transaction Gateway to communicate with CICS. It can link to CICS programs, passing data in a buffer called a COMMAREA.

Transaction management support

The CICS ECI resource adapter implements the `LocalTransaction` interface, and supports *local* transactions. *Global* transactions are not supported. If you use an application server than supports last resource optimization, the resource adapter can participate in a global transaction, provided that it is the only local transaction resource in the global transaction. WebSphere Application Server Advanced Edition V4 does not currently support last resource optimization.

Security management support

To communicate with a secure CICS region using the ECI, you must send a valid user ID and password. These can be supplied by the application, or by the container (if the container supports this).

Restriction: WebSphere Application Server Advanced Edition V4.02 does *not* currently provide the container mechanism for supplying the user ID and password.

Connection management support

Support is provided by the ECI resource adapter for pooling connections from the EJB server to the gateway daemon. This allows for optimization, by reducing the amount of socket open and closes required for a series of ECI calls.

3.4.2 EPI resource adapter

The CICS EPI resource adapter uses the External Presentation Interface of the CICS Transaction Gateway to communicate with CICS. It can start CICS transactions by interacting with a *virtual terminal*. The virtual terminal represents a 3270 terminal to the CTG user.

Transaction management support

The EPI deals only with the CICS 3270 interface, therefore, recoverable work is not performed on the client. For this reason the CICS EPI resource adapter does not support transactions.

Security management support

To communicate with a secure CICS region using the EPI, you must supply a valid user ID and password. If you are using *signon incapable terminals*, then a user ID and password are required for each request. Support for *signon capable terminals* is provided by the `LogonLogoff` interface, which allows a Java developer to define a signon procedure in CICS (such as starting the CESN transaction). For further details on the difference between signon and signon incapable terminals when using the EPI resource adapter, refer to Example 8.3 on page 212.

Connection management support

Support is provided by the EPI resource adapter for pooling terminals installed on the CICS region. This provides for optimization by reducing the amount of network flows and CICS terminal install operations required for a series of EPI calls.

Part 2

Connecting to COMMAREA based CICS programs

Part 2 describes how to develop ECI-based applications to invoke CICS COMMAREA based programs. Information is provided on using the base CTG `ECIRequest` class, and the J2EE CCI in a both a non-managed and a managed environment.

4

ECI and ESI applications

This chapter discusses how to the use the `JavaGateway` and `ECIRequest` objects to call COMMAREA-based CICS applications, and the `ESIRequest` object to invoke security functions in CICS.

This chapter covers the following topics:

- Base classes overview
- Synchronous ECI calls
- Asynchronous ECI calls
- ESI calls
- Extended logical units of work
- Tracing
- Exception handling

The sample applications shown in this chapter, are a set of classes in the packages `itso.cics.eci` and `itso.cics.esi`. Instructions on how to download the sample code are in Appendix D., “Additional material” on page 261.

4.1 Base classes overview

The CTG itself provides a Java class library consisting of a set of *base classes*, namely the `JavaGateway`, `ECIRequest`, `EPIRequest`, `ESIRequest` and `CicsCpRequest` classes, in the `com.ibm.ctg.client` package. These classes are relatively simple to use, but do require a greater understanding of the workings of CICS, compared to the use of J2EE CCI or the EPI support classes.

A Java application can use the base classes to communicate with a CICS system through the CTG. The Java API is provided in the JAR file `ctgclient.jar`, and is used to make a network connection to a gateway daemon. The gateway daemon (coded in Java) listens for remote requests from Java clients, and invokes the services of the underlying client daemon (code in native code) using the Java Native Interface (JNI). This scenario is illustrated in Figure 4-1 for a Java client using a CTG on Windows NT. On OS/390 the EXCI is used in place of the client daemon to provide an ECI interface to Java applications.

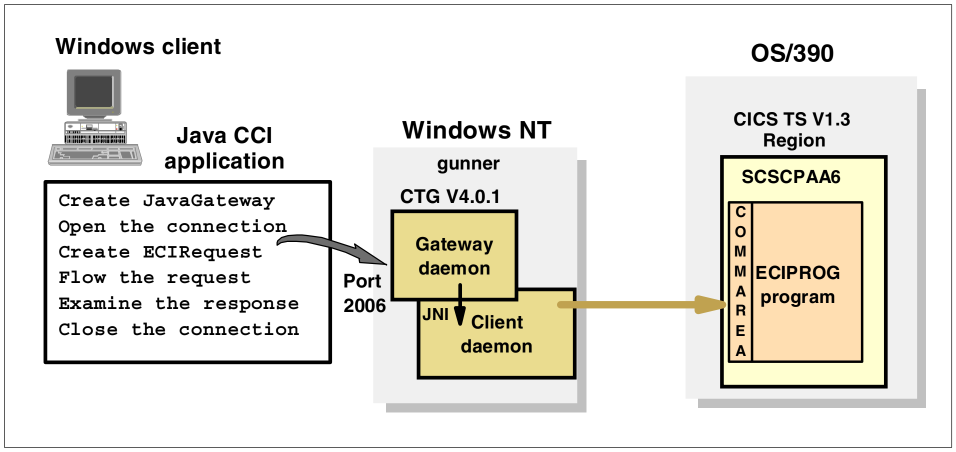


Figure 4-1   CTG scenario for ECIRequest application

The Java code may be a simple application on a workstation deployed as a two-tier application, or in an Web application server as part of a three-tier application architecture. The CTG may be installed on a separate workstation as the Web application, on a S/390 host, or on the same machine as the Java application runs. For more details on the infrastructure of the CTG, refer to Chapter 2, “CICS Transaction Gateway” on page 9.

4.1.1 JavaGateway overview

The basis of the communication from the Java application to the CTG is provided by the `JavaGateway` class, which encapsulates the behavior of the CTG itself, and acts as an interface for it. The actual `JavaGateway` is just a wrapper class, and there are several underlying classes, also provided in the `ctgclient.jar file`, that do the real work.

The `JavaGateway` object represents a logical connection between your program and the CTG. This applies equally when you specify a network protocol such as `tcp://gunner:2006`, or if you use the *local* gateway by specifying the protocol `local:`.

The *local* gateway is a special type of `JavaGateway` that allows you to use a CTG installed on the same machine as the Java application. It allows you to bypass the gateway daemon and the associated network overheads, and uses the JNI to directly invoke the services of the CTG client daemon.

Program flow

The flow of a Java program that uses the `JavaGateway` to connect to a CTG follows these steps:

1. The Java program instantiates a `JavaGateway` object. Two `JavaGateway` constructors simplify the creation of a `JavaGateway` by setting the relevant properties and implicitly calling the `open()` method for you. On return from a successful call to one of these constructors, the resultant `JavaGateway` is opened, and the underlying connection to the CTG will have been made.
2. The Java program creates an instance of one of the gateway request classes containing these requests:
   a. An `ECIRequest` object is created for an ECI request.
   b. An `ESIRequest` object is created for an ESI request.
   c. An `EPIRequest` object is created for an EPI request. This is not discussed in this book. For details on the alternative EPI support classes refer to Chapter 7, “EPI support classes” on page 155).
   d. A `CicsCpRequest` object is created for querying the code page of the CTG the `CicsCpRequest` object is using. (Refer to CicsCpRequest object in Appendix B.2, “Conversion within CICS: DFHCNV templates” on page 232.)
3. The Java application then flows the request to the CICS Transaction Gateway using the `flow()` method of the `JavaGateway` object.
4. The Java program checks the return code of the flow operation to see whether the request was successful.

5. The program continues to create request objects and flow them through the `JavaGateway` object, as appropriate.
6. The Java program then closes the `JavaGateway` object, using the `close()` method.

Supported protocols

The `JavaGateway` supports the following network protocols from remote Java clients:

TCP	This is a simple TCP/IP socket-based communication, and is the easiest to use while providing the fastest network connections.
SSL	This is a secure sockets layer (SSL) version of the TCP protocol for secure communications.
HTTP	This provides HTTP support to allow communication through a HTTP proxy in a firewall configuration.
HTTPS	This is a SSL version of the HTTP protocol. It can be used through a Java applet, or from a Java application, if the Java environment provides support.

There is also support for a *local gateway* using the following protocol:

local	This is used when the Java application and the CTG reside on the same host (often the case in a servlet configuration). It provides optimized communication by removing the network latency associated with TCP/IP calls to the gateway daemon.

Configuration

When you create a `JavaGateway`, you determine the protocol to use, and if required, the connection details of the remote CTG. To do so, the `JavaGateway` provides the following methods:

`setProtocol()`	Sets the protocol to be used for the connection with the CTG. Possible values are: `tcp`, `http`, `https`, `ssl` or `local`
`setAddress()`	Sets the IP address were the CTG resides on. It may be a hostname or a numerical IP address
`setPort()`	Sets the port number on which the gateway daemon is listening
`setURL()`	To simplify you may use this method to set the former three attributes through an URL parameter in the form: `protocol://address:port/`.

`open()`	This method opens the connection between the Java client and the CTG.

Tip: Remember that the `JavaGateway` provides constructors that simplify its configuration by setting the relevant properties, and implicitly calling the `open` method for you.

`close()`	Closes an open connection
`isOpen()`	This is a useful method to check if the connection is still alive.
`flow()`	This method is used to flow or pass a request object to the CTG once the gateway is opened.

4.1.2 ECIRequest overview

The External Call Interface (ECI) allows non-CICS applications to call a CICS program on a CICS server. ECI calls are made using an `ECIRequest` object. These calls are of three types (the type that is controlled by setting the `Call_type` parameter in the `ECIRequest` object to the proper value):

- Program link calls
- Status information calls
- Reply solicitation calls.

Program link calls

A program link call causes a CICS program to be called on a CICS server. These calls can be one of two types:

Synchronous	This is a blocking call, and the client application is suspended until the called program has finished. For further information refer to 4.2, “Synchronous ECI calls” on page 40.
Asynchronous	This is a non-blocking call, and the calling application gets control back without reference to the completion of the called program. The application can ask to be notified later when the information is available by using a reply solicitation call to determine the outcome of the asynchronous request. For details refer to 4.3, “Asynchronous ECI calls” on page 44.

Status information calls

Status information calls retrieve status information about the type of system on which the application is running, and its status. These calls can be either synchronous or asynchronous, although there is little value in using an asynchronous call type to query the status of a server.

Restriction: During this redbook project we found that status calls did not provide reliable information about the status of the CICS server in all circumstances. Therefore, our advice is to use exception handling to check the server status, instead of a status call. A simple method would be to create a dummy CICS program and use this to obtain the status through polling.

Reply solicitation calls

Reply solicitation calls are used to get information back after asynchronous calls have been made. Reply solicitation calls can be one of two types:

General	These retrieve information for any outstanding pieces of work
Specific	These retrieve information for a named asynchronous request. An application that uses the asynchronous method of calling may have several program link and status information calls outstanding at any time.

4.1.3 ESIRequest overview

The ESIRequest class provides access to password expiration management (PEM) security services in an attached CICS region. These services are implemented in CICS by the External Security Manager (ESM), for which RACF is often used.

ESI functions

The ESIRequest class provides the following PEM functions:

Verify Password	Allows a client application to verify that a password matches the password for a given user ID stored by the CICS ESM
Change Password	Allows a client application to change the password held by the CICS ESM for a given user ID

For further application development details refer to 4.4, “ESI calls” on page 53.

4.2 Synchronous ECI calls

To make synchronous ECI calls to CICS, you need to create an `ECIRequest` objects and flow this using a `JavaGateway` object. The `JavaGateway` must be correctly configured before the request is flowed. This involves setting the attributes such as the CICS server name, the program name, and the user ID and password if required.

4.2.1 ECIRequest configuration

There are several attributes of the `ECIRequest` object that must be supplied with proper values before flowing the request. These attributes are:

`Server`	The CICS region where the program resides
`Program`	The name of the program to execute
`Extended_Mode`	Whether this call is to be extended through several client/server interactions, or not. Possible values are:

- **`ECI_EXTENDED`**
  Indicates that this call is the first of several program link calls that will constitute a single logical unit-of-work (LUW). These series of calls need to be finished with a *commit* or *rollback*.
- **`ECI_NO_EXTEND`**
  Indicates that this is a single program link call that is, by itself, a complete LUW. It does not need a commit or backout. If the call ends with no error, the changes will be committed, otherwise they will be rolled back.
- **`ECI_COMMIT`**
  Commits all the changes since the beginning of this LUW
- **`ECI_BACKOUT`**
  Rolls back all the changes since the beginning of this LUW

`Luw_Token`	A token that identifies a unique LUW. All LUWs including single program link calls, begin with a value of `ECI_NEW_LUW`. When using extended LUWs, the CTG returns a unique value for this attribute to identify the specific LUW.
`Commarea`	The byte array holding the COMMAREA input and output data.
`Commarea_length`	The length of the COMMAREA byte array
`Userid`	The user ID of the CICS server. This is only required if security is enabled in CICS.
`Password`	The password for the specific user ID. This is only required if security is enabled in CICS.

4.2.2 Program flow

Synchronous ECI requests are the simplest way of calling CICS programs using the CTG. When a synchronous ECI request is made, the calling application *blocks* until the reply is received from CICS.

Figure 4-2 illustrates the program flow from the creation of a `JavaGateway` object to the synchronous ECI call. This sample code executes the CICS program ECIPROG on the CICS server SCSCPAA6, connecting through *TCP/IP,* to the CTG listening on port *2006,* on the host *gunner*. This code is supplied as the sample class `SyncECI` provided with this book.

The CICS program ECIPROG requires an input COMMAREA of 27 bytes in length. No input data is required, and ECIPROG will return the 8 character CICS APPLID, and the date and time on CICS. The code for ECIPROG is supplied in C.2, "ECIPROG" on page 244.

```
try {
1  JavaGateway jg= new JavaGateway("tcp://gunner",2006);
2  byte commarea[]=("---------------------------").getBytes("IBM037");
3  ECIRequest req= new ECIRequest(ECIRequest.ECI_SYNC, //sync or async
   "SCSCPAA6", //CICS server name
   null, null, //userid & password
   "ECIPROG", //program name
   "CPMI", //transaction ID
   commarea, commarea.length, //commarea data & length
   ECIRequest.ECI_NO_EXTEND, //extended mode
   ECIRequest.ECI_LUW_NEW); //LUW token
   System.out.println("Comm   in: " + new String(req.Commarea, "IBM037"));
4  jg.flow(req);
5  System.out.println("Comm out: " + new String(req.Commarea, "IBM037"));
   System.out.println("Rc: " + req.getRc());
6  jg.close();
}catch (IOException ioe) {
   System.out.println("Handled exception: " + ioe.toString());
}
```

Figure 4-2 Synchronous ECI call

The import statements required for this sample code are shown below:

```
import com.ibm.ctg.client.JavaGateway;
import com.ibm.ctg.client.ECIRequest;
import java.io.IOException;
```

The logic in the code is as follows:

- 1 Instantiate the `JavaGateway` object that will provide the connection to the CTG, and configure the connection through the constructor.

  The constructor opens the `JavaGateway` for us, so we do not need to explicitly call the `open()` method.

- 2 Construct a byte array data structure for the COMMAREA and initialize the COMMAREA input data.

Initialize the input to 27 characters in order to illustrate the difference between input and output data when printing to the console.

- **3** Instantiate a request object.

  Create an `ECIRequest` object and configure it through its constructor. Use the extended constructor and supply the call type, server name, program name, transaction ID, COMMAREA data and its length, and information on the logical unit of work token. The attribute that identifies this request as synchronous is the `Call_type` with value `ECI_SYNC`.

  Our example specifies the default transaction ID *CPMI*. In a real-life scenario you should contact your CICS systems programmer to determine if the use of a customized mirror transaction ID is required.

- **4** Flow the request.

  The request is flowed to CICS through the CTG by invoking the `flow()` method on the `JavaGateway` object. The program flow blocks at this point until a reply is received back with the return data in the COMMAREA.

- **5** Write the COMMAREA output and ECI return code to the console. The COMMAREA data is converted from the EBCDIC IBM037 code page to a `String`, which is Unicode. For more details on different options for handling data conversion, refer to Appendix B, “Data conversion” on page 227.

- **6** Close the gateway connection by invoking the `close()` method on the `JavaGateway` object. This closes the underlying socket connection to the CTG.

The output for this code should be as shown in Example 4-1.

Example 4-1 SyncECI output

```
Comm   in: ---------------------------
Comm out: SCSCPAA6 13/12/01 20:54:08
Rc: 0
```

4.3 Asynchronous ECI calls

Unlike synchronous calls, asynchronous ECI calls do not block, so control is returned to the application before the response is returned from the CTG. To receive the response, it is necessary to explicitly flow another request in order to obtain the reply. Therefore, an asynchronous call involves two steps, the *call* and the *reply*.

An asynchronous call is specified by setting the `Call_type` attribute set to `ECI_ASYNC` on an `ECIRequest` object. The request object can be reused on a subsequent call to receive the response, by setting its `Call_type` attribute to one of the possible values for reply solicitation calls, such as `ECI_GET_SPECIFIC_REPLY_WAIT`.

There is also another way to implement asynchronous calls using just one request. This is done through `callback` objects. The `callback` object is associated with the asynchronous request, so that the `callback` object is notified when the reply is ready.

4.3.1 Reply solicitation calls

Reply solicitation calls are used to solicit a reply for a previous asynchronous ECI call. The CTG queues the asynchronous requests and responses. There are four different call types for reply solicitation calls, classified into two types:

- Generic reply solicitation calls
- Specific reply solicitation calls

General reply solicitation calls

These call types retrieve any pending response from the CTG. There are two call types that you can use:

`ECI_GET_REPLY`	Provides a non-blocking reply solicitation call
`ECI_GET_REPLY_WAIT`	Provides a blocking reply solicitation call

Note: We recommend that you do not use general reply solicitation calls because they can receive any response queued in the CTG, therefore, one application can receive a reply destined for a different application. Their use is discouraged when using a remote CTG. Instead, it is recommended you use specific reply solicitation calls, or `callback` objects.

Attention: To use general reply solicitation calls with the CTG, you have to enable this support by selecting the box labelled: *Let Java clients obtain generic ECI replies* in the CTG configuration tool.

Specific reply solicitation calls

These call types retrieve a specific response pending in the CTG. To identify the specific response, you need to supply a *message qualifier*. This is specified using an `int` value that uniquely identifies the ECI call and its associated response. There are two call types which can be used:

- **`ECI_GET_SPECIFIC_REPLY`**

  Provides a reply solicitation call to return reply information for a specific asynchronous request. The specific request is referred through the message qualifier of the request. If there is no such reply, `ECI_ERR_NO_REPLY` is returned.

- **`ECI_GET_SPECIFIC_REPLY_WAIT`**

  Provides a reply solicitation call to return reply information for a specific asynchronous request. If there is no such reply, the request will block until one is ready.

Restriction: Message qualifiers are not supported when communicating with a CTG on OS/390, either using a gateway daemon on OS/390, or using the local protocol from an application on OS/390. A specific reply solicitation call will return the response ECI_ERR_NO_REPLY if used.

4.3.2 Program flow

Using an asynchronous ECI request enables the calling Java application to make non-blocking calls to CICS. The program flow steps to make a successful asynchronous ECI call are:

1. Invoke the `setAutoMessageQual()` method on the `ECIRequest` object, passing a `true` value as the input parameter. This method ensures that unique message qualifiers will be assigned.
2. Flow the asynchronous ECIRequest.
3. Store the message qualifier returned from the ECI request in order to retrieve the asynchronous response later.

The simplest way to do this is to reuse the `ECIRequest` object for the subsequent reply solicitation call, as the message qualifier is returned within this object. However, it is also possible to invoke the `getMessageQualifier()` method on the `ECIRequest` object, which will return the message qualifier that can then be saved, allowing the `ECIRequest` object to be reused for other purposes.

4. Make the reply solicitation call using an `ECIRequest` object, with its message qualifier set to the value previously obtained from the flowed program link call.

Sample code is shown in Figure 4-3 for making an asynchronous ECI call to invoke the ECIPROG program on the CICS server `SCSCPAA6`, using the CTG on `tcp://gunner`, and a specific reply solicitation call. Another request object was *not* created for the reply solicitation call, instead the `ECIRequest` object used for the program link call was reused. After the flow of the program link, the `ECIRequest` message qualifier value changes. As you reuse the same instance, do not reset this value for the specific reply solicitation call. This code is supplied in the sample class As`yncECI` provided with this book.

```
try {
1  JavaGateway jg= new JavaGateway("tcp://gunner",2006);
2  byte commarea[]=("--------------------------").getBytes("IBM037");
3  ECIRequest req= new
      ECIRequest(ECIRequest.ECI_ASYNC, //call type: sync or async
      "SCSCPAA6", //CICS server name
      null, null, //userid & password
      "ECIPROG", //program name
      "CPMI", //transaction id
      commarea, commarea.length, //commarea data & length
      ECIRequest.ECI_NO_EXTEND, //extended mode
      ECIRequest.ECI_LUW_NEW); //LUW token
   System.out.println("(Main) Comm   in: " + new String(req.Commarea,
      "IBM037"));
4  req.setAutoMsgQual(true);
5  jg.flow(req);
   //we reuse the request object as many attributes remain unchanged
6  req.Call_Type= ECIRequest.ECI_GET_SPECIFIC_REPLY_WAIT;
7  jg.flow(req);
8  System.out.println("Comm out: " + new String(req.Commarea,
      "IBM037"));
   System.out.println("Rc: " + req.getRc());
9  jg.close();
}catch (IOException ioe){
   System.out.println("(Main) Handled exception: " + ioe.toString());
}
```

Figure 4-3 Asynchronous call with a specific reply solicitation call

The output should resemble Example 4-1 on page 43. The logic in this code is as follows:

- **1** Instantiate the `JavaGateway` object that will provide the connection with the CTG and configure the connection through the constructor.

  The constructor will implicitly open the connection to the gateway, so that future ECI requests may be flown.
- **2** Construct a byte array data structure for the COMMAREA, and initialize the COMMAREA data.

  Initialize the byte array to '-' chars so that any updates to the COMMAREA are highlighted when the printing to the console. The length of the COMMAREA is 27 bytes.
- **3** Instantiate the `ECIRequest` object.

  Configure the `ECIRequest` object through its constructor. The call type, server name, program name, transaction ID, the COMMAREA data, and its length and information on the logical of unit of work, are all supplied. Specify the default CICS mirror transaction ID CPMI. If required, this can be overridden by a user private mirror transaction. This may be useful for monitoring or security purposes.
- **4** Invoke `setAutoMessageQualifier()` method with the value `true`.

  This will ensure safe message qualifier generation when the request is flowed.
- **5** Flow the request to CICS through the gateway.

  The request is passed to the CTG and it flows to the CICS server. At this point, the request will get a message qualifier assigned from the CTG. If you do not want to reuse the `ECIRequest` object (like the examples), store the value of the message qualifier after flowing the request in order to place the specific reply solicitation call later.
- **6** Reusing the `ECIRequest` object, reinitialize it to place a specific reply solicitation call.
- **7** Flow the specific reply solicitation call.
- **8** Write the COMMAREA returned to the console. There will be a conversion of the COMMAREA data, from the EBCDIC IBM037 code page, to a Unicode string.
- **9** Close the gateway connection, as it is no longer needed.

  This closes the underlying TCP/IP socket connection to the gateway daemon.

4.3.3 Callback objects

The `ECIRequest` class supports `callback` objects. A `callback` object must implement the `Callbackable` interface and is associated with an `ECIRequest` object invoking `setCallback()` on the request object

Once the association is made, and the request flown, the `callback` object will be notified and provided with the data, when the reply for the associated request is ready. This makes the `callback` object the perfect approach for asynchronous calls.

You can specify a `callback` object for all ECIRequest call types (not just asynchronous call types). In the case of synchronous calls, the results are passed to your `Callbackable` object, as well as to your `ECIRequest` object, in the flow request.

The `Callbackable` interface that the `callback` object must implement, defines these two methods:

`setResults()`	This method is invoked when the response for the `ECIRequest` is ready. It will provide your implementation with the COMMAREA data as a byte array typed parameter. Your implementation of this method should be able to store the COMMAREA data for later use.
`run()`	After invoking the `setResults()` method, the `JavaGateway` will create a thread that will be able to execute the `run()` method. This method is inherited from the `Runnable` interface that the `Callbackable` interface extends. Your implementation of this method should be able to do all the result data management.

For our sample application, a `callback` object has been developed. The code for this `callback` example is provided in package `itso.cics.eci` in the classes: `AsyncCallbackECI`, `Callback`, and the interface: `Sleeper`.

Timestamps in both the application and the `callback` object have been added to show how the program flows through time. The `callback` object implementation is shown in Figure 4-4.

Attention: If you try to build your own sample application using a `callback` object, you may encounter a problem during testing. Your sample application may flow requests from its `main()` method. Due to the asynchronous behavior of the `callback` object, your application may terminate before the `callback` object performs any task. To avoid this problem, we implemented a synchronization mechanism so that the application does not terminate until the `run()` method of the `callback` object finishes. This is achieved through the `Sleeper` interface shown in Figure 4-5.

Note that in the `run()` method, a timestamp is written to the console, which wakes up the main thread.

```
public class Callback implements Callbackable {
private byte[] results = null;
private Sleeper callbackListner = null;
public Callback(Sleeper listener) {
   super();
}
public Callback(Sleeper listener) {
   callbackListner=listener;
}
public void run() {
   System.out.println("(Callback) Response ready at: " + (new
      SimpleDateFormat("HH:mm:ss:SSS")).format(new Date()));
   try {
      System.out.println("(Callback)\t\tCOMMAREA: " + new String(results,
         "IBM037"));
   } catch (UnsupportedEncodingException uee) {
   }
   ((Sleeper)callbackListner).wakeUp();
}

public void setResults(com.ibm.ctg.client.GatewayRequest req) {
   results=((ECIRequest)req).Commarea;
}
}
```

Figure 4-4 The callback object implementation

The import statements used for this class are those shown below:

```
import com.ibm.ctg.client.Callbackable;
import com.ibm.ctg.client.ECIRequest;
import java.io.UnsupportedEncodingException;
import java.text.SimpleDateFormat;
import java.util.Date;
```

The Sleeper interface shown in Figure 4-5 defines three methods to be implemented by the sample application, in order to perform synchronization tasks with the callback object.

```
public interface Sleeper {
   void sleep() throws InterruptedException;
   void sleep(int timeout) throws InterruptedException;
   void wakeUp();
}
```

Figure 4-5 The Sleeper interface

Synchronization is performed through a wait/notify mechanism. See the application sample code in Figure 4-6 to see how these methods are implemented.

```
boolean notified=false;
public void sleep()throws InterruptedException {
   synchronized (this){
      if(!notified){
         wait();
      }
      notified=false;
   }
}
public void sleep(int timeout)throws InterruptedException {
   synchronized (this){
      if(!notified){
         wait(timeout);
      }
      notified=false;
   }
}
public void wakeUp(){
   synchronized (this){
      notified=true;
      notify();
   }
}
```

Figure 4-6 Implementation of the Sleeper interface

The sample application code in Figure 4-7 shows how to write a sample application that asynchronously invokes the CICS program ECIPROG on the CICS region SCSCPAA6, using a callback object.

```
try {
   JavaGateway jg= new JavaGateway();
   jg.setURL("tcp://gunner:2006");
   jg.open();
   byte commarea[]= new byte[27];
   commarea= ("--------------------------").getBytes("IBM037");
1  ECIRequest req= new ECIRequest(ECIRequest.ECI_ASYNC,
      "SCSCPAA6", //CICS server name
      null, //userid
      null, //password
      "ECIPROG", //program name
      "CPMI", //transaction id
      commarea, //commarea data
      commarea.length, //commarea length
      ECIRequest.ECI_NO_EXTEND, //New LUW
      ECIRequest.ECI_LUW_NEW); //LUW token
2  req.setCallback(new Callback(this));
   SimpleDateFormat df=new SimpleDateFormat("HH:mm:ss:SSS");
   System.out.println("(Main) Before  flowing: " + df.format(new Date()));
   System.out.println("(Main)\t\tCOMMAREA: " + new String(req.Commarea,
      "IBM037"));
3  jg.flow(req);
   System.out.println("(Main) After   flowing: " + df.format(new Date()));
   System.out.println("(Main)\t\tCOMMAREA: " + new String(req.Commarea,
      "IBM037"));
   System.out.println("(Main)\t\tRc: " + req.getRc());
   System.out.println("(Main) After response: " + df.format(new Date()));
   System.out.println("(Main)\t\tCOMMAREA: " + new String(req.Commarea,
      "IBM037"));
   System.out.println("(Main) Rc: " + req.getRc()+"; "+req.getRcString());
4  sleep();
   jg.close();
}catch (IOException ioe) {
   System.out.println("(Main) Handled exception: " + ioe.toString());
}catch (InterruptedException ie) {}
```

Figure 4-7 Java code for asynchronous request with callback

The import statements for this asynchronous request with `callback` are shown below:

```
import com.ibm.ctg.client.JavaGateway;
import com.ibm.ctg.client.ECIRequest;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.io.IOException;
```

The logic in the code in Figure 4-7 is as follows:

- **1** After creating a COMMAREA instantiate and configure the ECIRequest object through its constructor.
- **2** Associate the callback object to the request.

  It is passing the application object, this, to the callback object constructor for synchronization purposes. The application object is to be notified by the callback object when the reply is processed.
- **3** Flow the request.

  No other request is flown apart from the ECI_ASYNC program link call. No reply solicitation call is needed. The CTG will inform the callback object as soon as a response is ready.
- **4** Invoke the sleep() method. (Refer to Figure 4-6 on page 50 to view how the application class implements this method from the Sleeper interface).

  All the synchronization that is implemented through an interface is due to the fact that both the wait() and notify() methods must be synchronized on the same object, or monitor. That object monitor should be owned by the waiting object. Any Java object may be a monitor. An object owns a monitor when it is the monitor (this), or when it created it.

  The sleep() method invokes a wait() synchronized on the application object (the application object acts as a monitor). To unlock the application a notify(), synchronized on the same monitor (the application object), must be invoked on the waiting object (also the application object). This unlocking is performed by the wakeUp() method.

  The waiting object (the application waits until the callback object has finished processing the reply) and the object that performs a notify on it (the callback object will tell the application when he has finished the processing) must know the object where synchronization takes place.

The output from the execution of the sample code is shown in Example 4-2.

Example 4-2 AsyncCallbackECI output

```
(Main) Before  flowing: 18:28:03:416
(Main) COMMAREA: --------------------------
(Main) After   flowing: 18:28:03:526
(Main) COMMAREA: --------------------------
(Main) Rc: 0
(Main) After response: 18:28:03:526
(Main) COMMAREA: --------------------------
(Main) Rc: 0; OK
(Callback) Response ready at: 18:28:03:666
(Callback)COMMAREA: SCSCPAA6 13/12/01 21:28:18
```

4.4 ESI calls

The External Security Interface (ESI) offered by the CTG is a Java interface to Password Expiration Management (PEM). This is used to verify and change the user ID and password information stored in CICS. PEM is part of the APPC architecture. A PEM server, such as CICS, is written to send and receive the architected PEM flows using the APPC communication protocol. The two architected functions are *verification of passwords*, and *changing of passwords*, and these invoke the corresponding CICS commands EXEC CICS VERIFY PASSWORD and EXEC CICS CHANGE PASSWORD, which interface with the CICS External Security Manager (ESM) such as RACF, in order to perform the desired operation (Figure 4-8).

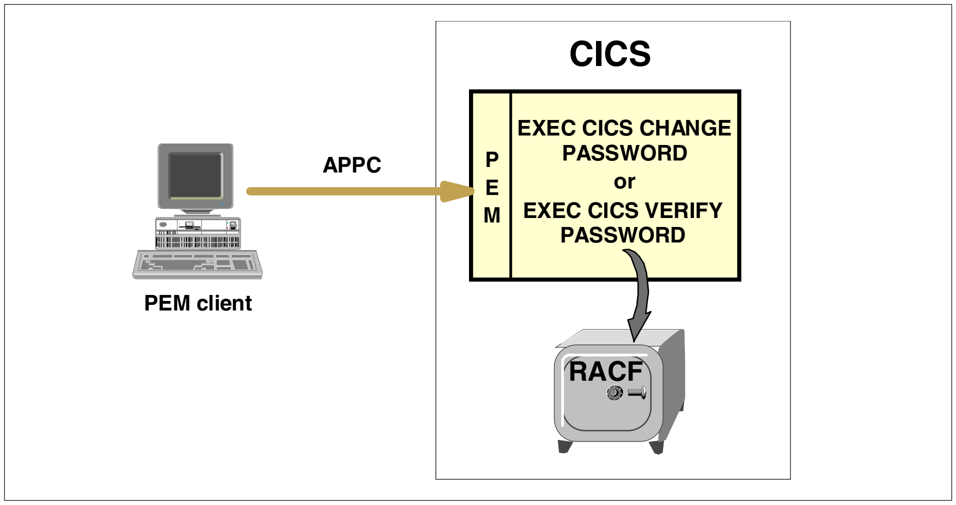


Figure 4-8 CICS Password Expiration Management

Note: Because PEM is based on the APPC architecture, the ESI can only be used over APPC or TCP62 connections to a CICS/ESA, or CICS TS region on OS/390 or VSE/ESA. For further details on PEM, refer to the *CICS RACF Security Guide*, SC33-1701.

The ESI provides access to security services implemented in CICS through the `com.ibm.ctg.client.ESIRequest` class. The configuration of the `ESIRequest` object can be done through two static methods that generate an `ESIRequest` that is properly configured for the request you want to flow.

`verifyPassword()` This method takes as arguments three `String` objects: the user ID, the password, and the server. The method is static and returns an `ESIRequest` object, which when flown, will cause CICS to check the password for the given user ID.

changePassword()	This method takes as arguments four `String` objects: the user ID, the current password, the new password, and the server. This method is static and returns an `ESIRequest` object, which when flown, will attempt to change the current password of the specified user ID in the specified server.

The `ESIRequest` object also provides a few utility methods, to be invoked once the request has been flown, such as:

getExpiry()	This method retrieves a `java.util.Calendar` object with the expiration date of the specified password.
getLastAccess()	This method retrieves a `java.util.Calendar` object with the date of the last access to the server by the specified user ID.
getLastVerified()	This method retrieves a `java.util.Calendar` object with the date of the last time a password verification was flown for the specified user ID.

Figure 4-9 provides sample code on how to verify a user ID/password pair. The sample code verifies the user ID `CICSRS3` with password `PASSWORD` on server SCSCPAA7, connecting through the CTG on `tcp://gunner:2006/`.

```
try{
  JavaGateway jg= new JavaGateway("tcp://gunner", 2006);
  ESIRequest
  req=ESIRequest.verifyPassword("CICSRS9","PASSWORD","SCSCPAA7");
  jg.flow(req);
  System.out.println("Expiry date:  "
    +((req.getExpiry()!=null)?""+req.getExpiry().getTime():"(empty)"));
  System.out.println("Last access: "
    +((req.getLastAccess()!=null)?""+req.getLastAccess().getTime()
    :"(empty)"));
  System.out.println("Last verified:  "
    +((req.getLastVerified()!=null)?""+req.getLastVerified().getTime()
    :"(empty)"));
  System.out.println("Rc: "+req.getRc());
  jg.close();
}catch(IOException ioe){
   System.out.println("Handled exception: "+ioe.getMessage());
}
```

Figure 4-9 Password verification

This sample code requires the following import statements:

```
import java.io.IOException;
import com.ibm.ctg.client.JavaGateway;
import com.ibm.ctg.client.ESIRequest;
```

This sample code is provided with this book in package `itso.cics.esi` in class `VerifyESI`. The output for this sample code should be as shown in Example 4-3.

Example 4-3 VerifyESI output

```
Expiry date:  Wed Dec 31 00:00:00 PST 0002
Last access: Tue Dec 11 17:44:20 PST 2001
Last verified:  Thu Dec 13 21:37:22 PST 2001
Rc: 0
```

Figure 4-10, also provides sample code for the changing of a password. The sample code changes the CICSRS9 user's password from `PASSWORD` to `NOV2001` on the CICS server SCSCPAA7, connecting through the CTG on `tcp://gunner:2006/`.

```
try {
   JavaGateway jg= new JavaGateway("tcp://gunner", 2006);
   ESIRequest req=
      ESIRequest.changePassword("CICSRS9","NOV2001","PASSWORD","SCSCPAA7");
   jg.flow(req);
   System.out.println("Rc: "+req.getRc());
   jg.close();
} catch (IOException ioe) {
   System.out.println("Handled exception: "+ioe.getMessage());
}
```

Figure 4-10 Password changing

This code is also in package `itso.cics.esi` as the class `ChangeESI`.

4.5 Extended logical units of work

Before considering how to extend logical units of work (LUWs) we should first explain what one is. In CICS terms a l*ogical unit of work* (also termed a *unit of work*) is a recoverable sequence of operations within a transaction. The start and end of which are marked by synchronization points, where the LUW is either committed or rolled-back. If there are no explicit synchronization points within a transaction then the *start* and *end* of the transaction are themselves implicit synchronization points.

To illustrate the concept, imagine a bank transaction that consists of debiting money from one account, and crediting it to another. If you debit the money from one account, but the credit process fails, the data falls into an inconsistent state according to where the money was lost. So, the action of subtracting from one account, and adding the same amount to another account, needs to be carried out as one LUW.

If this operation is performed by a series of ECI calls, one would need to *extend* the LUW created by the first program, so that the second ECI call is executed within the scope of the logical unit of work that was created by the first ECI call. This is achieved within CICS by causing the mirror program, which executes ECI requests to remain active, after an ECI request has terminated. (The mirror program usually terminates at the end of each ECI request). The mirror program only terminates when a request is received to end the LUW.

Attention: In computing literature, the term *transaction* is often used to refer to a recoverable unit of work. However, in CICS, the term *transaction* has traditionally referred to a task running within CICS, which can be composed of one or more recoverable units of work.

To extend a logical unit of work in an ECI request is a simple matter of setting the `Extended_Mode` and `Luw_Token` attributes on the `ECIRequest` object. The `Extended_Mode` should be set to the constant `ECI_EXTENDED` and the `Luw_Token` is then taken from the first request and used to identify the extended LUW in further ECI calls.

The sample application ECIADDER

In this section we utilize a sample COBOL application called ECIADDER; the input and output COMMAREAs are shown in Figure 4-11 and Figure 4-12.

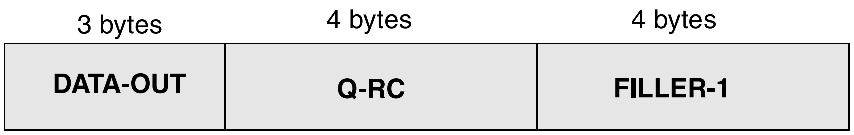


Figure 4-11   ECIADDER output COMMAREA

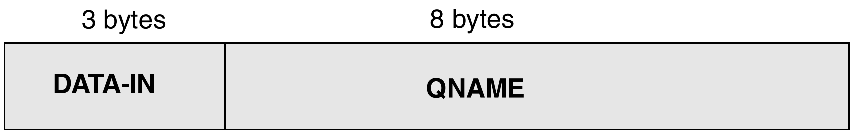


Figure 4-12   ECIADDER input COMMAREA

The code for these COBOL structures is shown below, and the full COBOL source code for the ECIADDER program can be found in C.1, “ECIADDER” on page 242. This is shipped as a sample with this redbook.

```
01 COMMAREA-RETURN.
          03 DATA-OUT                  PIC S9(3) DISPLAY SIGN IS
                                       LEADING SEPARATE CHARACTER.
          03 Q-RC                      PIC ZZZ9 DISPLAY.
          03 FILLER-1                  PIC X(4) VALUE SPACE.

       LINKAGE SECTION.
       01 DFHCOMMAREA.
          03 DATA-IN                   PIC S9(3) DISPLAY SIGN IS
                                       LEADING SEPARATE CHARACTER.
          03 QNAME                     PIC X(8).
```

This COMMAREA is character based, and takes as input an integer value (`DATA-IN`) and a temporary storage queue name (`QNAME`). It adds the integer to the value already stored in the queue, and then returns the result (`DATA-OUT`) along with a return code (`Q-RC`).

You can subsequently call ECIADDER to see how the value of the queue increases (or decreases if provided with negative integers) at each call. If you call ECIADDER, as we have learned from 4.2, “Synchronous ECI calls” on page 40, you will be executing single LUWs. Now we will show you how to make several calls to ECIADDER. See how the value on the queue has increased, and finally undo the changes on the queue to bring it back to its first state, or commit the changes to confirm them. The code example for this is shown in Figure 4-13 and is provided as the class `ExtendedLUW`.

```
try {
1  JavaGateway jg= new JavaGateway("tcp://gunner",2006);
2  ECIRequest req= new ECIRequest(ECIRequest.ECI_SYNC, //sync.async
      "SCSCPAA6", //CICS server name
      null, null, //userid & password
      "ECIADDER", //program name
      "CPMI", //transaction id
3     ("+000RMYQUEUE").getBytes("IBM037"),12, //commarea data & length
4     ECIRequest.ECI_NO_EXTEND, //will extended LUW
5     ECIRequest.ECI_LUW_NEW); //LUW token
6  jg.flow(req);
7  System.out.println("Queue starting value:"+(new
      String(req.Commarea,"IBM037")).substring(0,4));
8  req.Commarea=("+010RMYQUEUE").getBytes("IBM037");

9  req.Extend_Mode=ECIRequest.ECI_EXTENDED;
   req.Luw_Token=ECIRequest.ECI_LUW_NEW;
   jg.flow(req);
   System.out.println("Queue after adding 10: "+(new
      String(req.Commarea,"IBM037")).substring(0,4));

10 req.Extend_Mode=ECIRequest.ECI_BACKOUT;
   jg.flow(req);
11 req.Commarea=("+000RMYQUEUE").getBytes("IBM037");
   req.Extend_Mode=ECIRequest.ECI_NO_EXTEND;
   req.Luw_Token=ECIRequest.ECI_LUW_NEW;
   jg.flow(req);
   System.out.println("Queue after rollback: "+(new
      String(req.Commarea,"IBM037")).substring(0,4));

12 req.Commarea=("+010RMYQUEUE").getBytes("IBM037");
   req.Extend_Mode=ECIRequest.ECI_EXTENDED;
   req.Luw_Token=ECIRequest.ECI_LUW_NEW;
   jg.flow(req);
   req.Commarea=("-007RMYQUEUE").getBytes("IBM037");
   jg.flow(req);

13 req.Extend_Mode=ECIRequest.ECI_COMMIT;
   jg.flow(req);
   System.out.println("Queue after adding (10-7): "+(new
   String(req.Commarea,"IBM037")).substring(0,4));
14 jg.close();
}catch (IOException ioe) {
   System.out.println("Handled exception: " + ioe.toString());
}
```

Figure 4-13 Extending an LUW

The logic in this sample is described below, and the sample code for this example is supplied in package `itso.cics.eci` in class `ExtendedLUW`.

- **1** Instantiate the gateway, configure, and open it using the relevant constructor.
- **2** Instantiate the `ECIRequest` object that will be reused throughout all the sample code. Configure it through its constructor:
  - **3** Set the COMMAREA to match the `DFHCOMMAREA` fields in the ECIADDER program, and inform the request object about the size of the COMMAREA. For details of the ECIADDER source code refer to Appendix C.1, "ECIADDER" on page 242.

    The COMMAREA for ECIADDER is character data, and has a 12-byte length. The first four characters represent the signed value to be added to the value that is currently stored in the queue. One character is for the sign, and the other three are numeric characters. The 8 remaining bytes are the name of the queue. Since it is the character data, it must be formatted to the correct code page. Use the EBCDIC code page IBM037, as it is the CICS server code page.

    If the numeric value is 0, then the application will effectively just read the contents of the queue.

    The queue that is accessed throughout the sample code is called RMYQUEUE; the initial character of R denotes the queue as recoverable due to a CICS TSMODEL definition that signifies all queues prefixed with R as being recoverable.
  - **4** Set the `ECIRequest Extended_Mode` attribute to `ECI_NO_EXTEND`.

    The first call to ECIADDER is to check the initial value of the queue, therefore, do not extend this call, as it is not recoverable.
  - **5** Set the `Luw_Token` to `ECI_NEW_LUW` to indicate the CTG that this is the first call of this LUW.

    Set the `Extended_Mode` attribute to `ECI_NO_EXTEND`; this will be the first and only call of this LUW.
- **6** Flow the request to read the value stored in the queue, the result will be:

  ```
  Queue starting value: +000
  ```
- **7** When printing the result, convert the COMMAREA from EBDCIC code page IBM037 to a Unicode string. The numeric result is returned in the four first bytes of the COMMAREA.
- **8** Create a new COMMAREA in order to add 10 to the value currently stored in RMYQUEUE.

- 9 Configure the ECIRequest `Extended_Mode` attribute to `ECI_NO_EXTEND`ED and the Luw_token to `ECI_NEW_LUW`. Then flow the request passing the COMMAREA created in line 8.

  Extend the ECI call. However, in this case, after adding 10 to the value stored in RMYQUEUE, undo the changes with a backout, so the result will be:

  ```
  Queue after adding 10: +010
  ```

- 10 Configure the ECIRequest object `Extended_Mode` attribute to `ECI_BACKOUT` and flow the ECI request. This will undo all the changes in this LUW.

  This means that the action of adding 10 to the value in RMYQUEUE will have no affect, since it will be rolled back, and the value in the queue will still be the initial value.

- 11 After the backout, read the queue; this will confirm that no changes actually took place, so the result will be:

  ```
  Queue after rollback: +000
  ```

- 12 Again, create a new extended LUW. With the first flow, add 10 to the value stored in RMYQUEUE, next subtract 7 (by adding -7) and finally, confirm these actions with a commit.

  The request still has an `ECI_EXTENDED` value for `Extended_Mode` and the `Luw_Token` value remains unchanged since the last flow. The value of the `Luw_Token` has been supplied by the CTG on the last flow. If you are using different request objects, you should set the other request's `Luw_Token` attribute to the value returned by the CTG, as it is this value that identifies the LUW throughout all the flows.

  If you are reusing the request objects, none of these attributes must be changed.

- 13 Reconfigure the `ECIRequest` object, changing the `Extend_Mode` attribute to `ECI_COMMIT`, and flow the request.

  This request will confirm the previous request made to the queue (adding 10 and subtracting 7) since starting this extended LUW in step 12. The CICS mirror program will perform a synchronization point and commit the updates made to the queue, so that the result will be:

  ```
  Queue after adding (10-7): +003
  ```

Attention: You should be careful when extending a logical unit of work across multiple program link calls that may span a long time (for example, across user-think time). The reason is that the extended logical unit of work suspends the mirror transaction, which will hold various resources on the CICS server. This may have a knock on the effect, and cause delays to other users who are waiting for those same locks and resources.

- **14** Finally, close the gateway, which closes the socket connection to the gateway daemon.

Logical unit of work (LUW) IDs, and message qualifiers can only be used on the same `JavaGateway` instance that created or assigned them. This is a security feature. It stops different clients that are connected to the same CTG from manipulating the LUW IDs of another application, or from using the message qualifier to request its messages. For example, attempts to get a specific reply to a message from a different `JavaGateway` will result in an ECI_ERR_NO_REPLY return code.

Only one program link call per logical unit of work can be outstanding at any time. An asynchronous program link call is outstanding until a reply solicitation call has processed the reply. This means that you can not invoke programs concurrently within the same LUW.

4.6 Tracing

Within the CTG class library, tracing is controlled by the `com.ibm.ctg.client.T` class. Tracing is activated using the API calls in this class. All the methods in the T class are static, so no instantiation is needed. The methods defined by the T class are as follows:

`setOn()`	The standard option for application tracing. By default, it displays only the first 128-bytes of any data blocks (for example, the COMMAREA, or network flows).
`setDebugOn()`	The debugging option for application tracing. A *boolean* supplied as an argument determines the effect. By default, it traces out the whole of any data blocks. The trace contains more information about the CICS Transaction Gateway than the standard trace level. Calls to `setTruncationSize()` and `setDumpOffset()` should be made after the `setDebugOn()` call.
`setStackOn()`	The exception stack option for application tracing. A *boolean* supplied as an argument determines the effect. It traces most Java exceptions, including exceptions that are expected during normal operation of the CTG. No other tracing is written.
`setTimingOn()`	Specifies whether or not to display timestamps in the trace. A *boolean* supplied as an argument determines the effect. If on, timing information is appended to all messages.

setfullDataDump()	Specifies whether or not all the content for any data blocks is to be traced. A *boolean* supplied as an argument determines the effect. To determine precise data tracing use `setTruncationSize()` and `setDumpOffset()`.
setTruncationSize()	The integer value supplied as an argument specifies the maximum size of any data blocks that will be written in the trace. Any positive integer is valid. If you specify a value of 0, then no data blocks will be written in the trace.
setDumpOffset()	The integer value supplied as an argument specifies the offset from which displays of any data blocks will start. If the offset is greater than the total length of data to be displayed, an offset of 0 will be used.
setTFile()	The value filename specifies a file location for writing of trace output. This is as an alternative to the default output on stderr. Long filenames must be surrounded by quotation marks (for example: `trace output file.log`).

Alternatively, tracing can be activated using Java system directives. The Java directives are processed when this class is loaded. The relevant directives are listed below. The first column represents the directive name and the possible parameters, and the second the equivalent API call.

gateway.T = on|off	`setDebugOn()`
gateway.T.stack = on|off	`setStackOn()`
gateway.T.timing = on|off	`setTimingOn()`
gateway.T.fullDataDump = on|off	`setfullDataDump()`
gateway.T.setTruncationSize = integer	`setTruncationSize()`
gateway.T.setDumpOffset = integerValue	`setDumpOffset()`
gateway.T.setTFile = StringValue	`setTFile()`

A simple code snippet illustrates the enabling of tracing as shown in Figure 4-14.

```
....
com.ibm.ctg.client.T.setOn(true);
com.ibm.ctg.client.T.setfullDataDumpOn(false);
....
```

Figure 4-14   T class tracing

A sample trace snippet (using the `T` class) of an ECI call is shown in Example 4-4.

Example 4-4 T class tracing output

```
16:28:58:197 : main: S-C: CCL6603I: # Dump: 51/51 bytes : Offset = 0 Outbound Flow
main: S-C: CCL6603I: # 00000: 47 61 74 65 00 40 00 00 00 00 00 05 00 00 00 00
Gate.@..........
main: S-C: CCL6603I: # 00016: 00 00 00 00 00 00 00 04 42 41 53 45 00 00 00 00
........BASE....
main: S-C: CCL6603I: # 00032: 12 00 02 65 6E 00 02 55 53 00 00 00 00 00 00 00
...en..US.......
main: S-C: CCL6603I: # 00048: 00 00 00                                        ...
16:28:58:257 :
LL-Socket[addr=9.1.38.179,port=2006,localport=1818]com.ibm.ctg.client.TcpJavaGateway@314e: S-C:
CCL6602I: GatewayRequest type = BASE, flow version = 4194304, flow type = 5, Gateway return
code = 0, length of data following the header = 48.
Commarea in: --------------------------
16:28:58:417 : main: S-C: CCL6720I: ECIRequest: Call_Type = ECI_SYNC, Server = SCSCPAA6,
Program = ECIPROG, Transid = CPMI, Extend_Mode = ECI_NO_EXTEND, Luw_Token = 0,
Message_Qualifier = 0, Callbackable = false.
16:28:58:477 : main: S-C: CCL6727I: ECIRequest: Commarea_Length = 27.
16:28:58:487 : main: S-C: CCL6603I: # Dump: 80/137 bytes : Offset = 0 Outbound Flow
main: S-C: CCL6603I: # 00000: 47 61 74 65 00 40 00 00 00 00 00 01 00 00 00 00
Gate.@..........
main: S-C: CCL6603I: # 00016: 00 00 00 00 00 00 00 03 45 43 49 00 00 00 00 69
........ECI....i
main: S-C: CCL6603I: # 00032: 00 00 00 01 00 00 00 00 00 00 00 00 00 00 00 00
................
main: S-C: CCL6603I: # 00048: 00 53 43 53 43 50 41 41 36 00 00 00 00 00 00 00
.SCSCPAA6.......
main: S-C: CCL6603I: # 00064: 00 00 00 00 00 00 00 00 00 2A 2A 2A 2A 2A 2A 2A
.........*******
16:28:58:617 :
LL-Socket[addr=9.1.38.179,port=2006,localport=1818]com.ibm.ctg.client.TcpJavaGateway@314e: S-C:
CCL6602I: GatewayRequest type = ECI, flow version = 4194304, flow type = 3, Gateway return code
= 0, length of data following the header = 52.
16:28:58:627 :
LL-Socket[addr=9.1.38.179,port=2006,localport=1818]com.ibm.ctg.client.TcpJavaGateway@314e: S-C:
CCL6721I: ECIRequest: Call_Type = ECI_SYNC, Cics_Rc = ECI_NO_ERROR, Abend_Code =     ,
Luw_Token = 0, Message_Qualifier = 0.
16:28:58:637 :
LL-Socket[addr=9.1.38.179,port=2006,localport=1818]com.ibm.ctg.client.TcpJavaGateway@314e: S-C:
CCL6727I: ECIRequest: Commarea_Length = 27.
16:28:58:647 :
LL-Socket[addr=9.1.38.179,port=2006,localport=1818]com.ibm.ctg.client.TcpJavaGateway@314e: S-C:
CCL6603I: # Dump: 27/27 bytes : Offset = 0 Inbound Commarea
LL-Socket[addr=9.1.38.179,port=2006,localport=1818]com.ibm.ctg.client.TcpJavaGateway@314e: S-C:
CCL6603I: # 00000: E2 C3 E2 C3 D7 C1 C1 F6 40 F1 F7 61 F0 F1 61 F0   ........@..a..a.
```

```
LL-Socket[addr=9.1.38.179,port=2006,localport=1818]com.ibm.ctg.client.TcpJavaGateway@314e: S-C:
CCL6603I: # 00016: F2 40 F1 F9 7A F3 F1 7A F1 F3 40                .@..z..z..@
Commarea out: SCSCPAA6 17/01/02 19:31:13
Rc: 0
16:28:58:667 :
LL-Socket[addr=9.1.38.179,port=2006,localport=1818]com.ibm.ctg.client.TcpJavaGateway@314e: S-C:
CCL6602I: GatewayRequest type = BASE, flow version = 4194304, flow type = 4, Gateway return
code = 61444, length of data following the header = 0.
```

4.7 Exception handling

This section provides details on how to deal with error handling when using the `ECIRequest` object, and how to build an exception handling framework for use with the ECI.

When errors are encountered while making ECI calls, no exceptions are thrown. Instead, it is up to you to check the return code in the `Cics_Rc` field in the `EciRequest` object using the constants defined in the supplied `ECIReturnCodes` interface. The `getCicsRcString()` method may also be used to translate the `Cics_Rc` from an integer into the error code strings defined in the `ECIReturnCodes` interface, which may be of more use in error messages than the integer `Cics_Rc`.

In addition to the return code, the `Abend_code` field may *also* provide useful information in the form of the four character CICS abend code returned from the called CICS server.

The only case when an exception will be generated, is when there is a network problem communicating with the gateway daemon. In this case, a `java.io.IOException` will be thrown.

4.7.1 ECI return codes

The most common ECI error codes are as follows:

ECI_ERR_CICS_DIED	The specified server is no longer available.
ECI_ERR_MAX_SESSIONS	There are not enough communication resources to satisfy requests.
ECI_ERR_NO_CICS	The CICS system is not available.
ECI_ERR_RESOURCE_SHORTAGE	There are not enough resource to complete the request (usually a communications problem with SNA sessions or EXCI pipes).

ECI_ERR_SECURITY_ERROR	Either CICS rejected the request because the user ID and password were not valid, or they were missing.
ECI_ERR_TRANSACTION_ABEND	The CICS transaction ended abnormally.
ECI_ERR_UNKNOWN_SERVER	The requested server could not be located by the CTG.

These and other error codes are defined as constants in `ECIReturnCodes` interface, which should be used for any error determination.

4.7.2 ESI return codes

The ESI has a very similar structure for error handling as the `ECIRequest` class in that the `Cics_Rc` field contains the return code, and the `getCicsRcString()` can be used to translate the return code into a string. However, the `Abend_code` field is not provided.

The most common ESI error codes are:

ESI_ERR_PASSWORD_EXPIRED	The password has expired and needs to be changed.
ESI_ERR_PASSWORD_INVALID	The password is not valid for the given user ID on the specified server.
ESI_ERR_PASSWORD_REJECTED	The password was revoked by the CICS administrator. The user ID must ask the administrator to enable his account on the specified server.
ESI_ERR_USERID_INVALID	The user ID does not exist in the specified server.

These error codes are defined as constants in `ESIReturnCodes` interface, which should also be used for any error determination.

4.7.3 Implementing an exception handling framework

Based on our experience, you are encouraged to build a small exception handling framework for your applications. Figure 4-15 through Figure 4-17 on page 68 show a simple exception handling framework. It resembles the Java event model, but for simplicity, it is limited to exception notification for one single exception handler, because only one handler may be registered at each moment. This limitation is not implicit in the interfaces provided in the framework, but in the `CICSECIExceptionNotifierImpl` class implementation, and in the lack of some other classes and interfaces that may be needed to remedy the limitation. Also, the example framework shown does not handle all the exceptions.

Next, we define an exception handler interface, named `CICSECIExceptionHandler`, shown in Figure 4-15. This interface should be implemented by a specific error handling class, which should provide a method for each error defined. The sample code provides a class called `CICSECIExceptionHandlerImpl` with default message logging for each error situation.

```
public interface CICSECIExceptionHandler {
   void handleECI_ERR_CICS_DIED();
   void handleECI_ERR_RESPONSE_TIMEOUT();
   void handleECI_ERR_ROLLEDBACK();
   void handleECI_ERR_SECURITY_ERROR();
   void handleECI_ERR_TRANSACTION_ABEND();
   ....
   void handleUnexpected();
}
```

Figure 4-15 ECIRequest exception handling framework, part 1

Tip: If you think you need more information to handle your exceptions, you may want to pass a class of your own, with the information that you require as a parameter to each handler method.

And finally, we defined an exception notifier interface, named `CICSECIExceptionNotifier`, shown in Figure 4-16. Any class interacting with the CICS server should implement this interface, as these classes may obtain error codes from CICS.

```
public interface CICSECIExceptionNotifier {
   void addExceptionHandler(CICSECIExceptionHandler eh);
   boolean handleException(int errorCode);
   void removeExceptionHandler(CICSECIExceptionHandler eh);
}
```

Figure 4-16   ECIRequest exception handling framework, part 2

Usually, the implementation code for notifiers in this event model is very similar (if not the same) for all the notifiers. It is a good practice to provide developers with an implementation class of the notifier interface.

In this case, your classes should still implement the notifier interface, and have the implementation class as an aggregate. The methods inherited from the implemented interface will work as wrappers for the aggregated class methods, where the real code is.

In Figure 4-17 we show an example of an implementation class, named `CICSECIExceptionNotifierImpl`, for the notifier interface `CICSECIExceptionNotifier`.

```
import com.ibm.ctg.client.*;

public class CICSECIExceptionNotifierImpl
    implements ECIReturnCodes, CICSECIExceptionNotifier {

    private CICSECIExceptionHandler exceptionHandler;

    public CICSECIExceptionNotifierImpl() {
        super();
    }
    public void addExceptionHandler(CICSECIExceptionHandler eh) {
        exceptionHandler = eh;
    }
public boolean handleException(int errorCode) {
    if (exceptionHandler != null) {
        switch (errorCode) {
            case ECI_ERR_CICS_DIED :
                {
                    exceptionHandler.handleECI_ERR_CICS_DIED();
                    break;
                }
            case ECI_ERR_ROLLEDBACK :
                {
                    exceptionHandler.handleECI_ERR_ROLLEDBACK();
                    break;
                }
            case ECI_ERR_TRANSACTION_ABEND :
                {
                    exceptionHandler.handleECI_ERR_TRANSACTION_ABEND();
                    break;
                }
            default :
                {
                    exceptionHandler.handleUnexpected();
                    break;
                }
        }
    }
    return errorCode != ECI_NO_ERROR;
}
    public void removeExceptionHandler(CICSECIExceptionHandler eh) {
        exceptionHandler = null;
    }
}
```

Figure 4-17 Exception handling framework, part 3

Given that your class implements the `CICSECIExceptionNotifier`, the actual exception handling framework should be invoked as shown in Figure 4-18.

```
public class ExceptionECI implements CICSECIExceptionNotifier {

private CICSECIExceptionNotifierImpl notifier =  new CICSECIExceptionNotifierImpl();
private CICSECIExceptionHandler handler = new CICSECIExceptionHandlerImpl();

public void addExceptionHandler(CICSECIExceptionHandler eh) {
        notifier.addExceptionHandler(eh);
}
public void execute() {
    try {
            JavaGateway jg = new JavaGateway("tcp://gunner.almaden.ibm.com", 2006);
    byte commarea[] =("A-------------------------").getBytes("IBM037");
    ECIRequest req = new ECIRequest(ECIRequest.ECI_SYNC,
       "SCSCPAA6", //CICS server name
        null, null, //userid &password
        "ECIPROG", //program name
        "CPMI", //transaction ID
        commarea, commarea.length, //commarea data &length
        ECIRequest.ECI_NO_EXTEND, //extended mode
        ECIRequest.ECI_LUW_NEW); //LUW token
        System.out.println("Comm in:" + new String(req.Commarea, "IBM037"));
        jg.flow(req);

        addExceptionHandler(handler);
        if (handleException(req.getRc())) {
          System.out.println("CICS Error");
        } else {
          System.out.println("Comm out:" + new String(req.Commarea,"IBM037"));
        }
          jg.close();
    } catch (IOException ioe) {
            System.out.println("Handled exception:" + ioe.toString());
    }
}
public boolean handleException(int errorCode) {
   return notifier.handleException(errorCode);
}
public static void main(String[] args) {
        (new ExceptionECI()).execute();
}
public void removeExceptionHandler(CICSECIExceptionHandler eh) {
        notifier.removeExceptionHandler(eh);
}
}
```

Figure 4-18 Invoking the exception handling framework

The following classes are provided in the package `itso.cics.eci`, which is offered with the sample code for this redbook.

`CICSECIExceptionHandler`	Handler interface
`CICSECIExceptionHandlerImpl`	Implementation of handler
`CICSECIExceptionNotifier`	Notifier interface
`CICSECIExceptionNotifierImpl`	Implementation of notifier
`ExceptionECI`	Class invoking exception handling framework

5

CCI applications: ECI based

This chapter describes how to develop non-managed applications that use the ECI resource adapter. There are examples that introduce two ways to write these applications; using the Common Client Interface (CCI) classes directly and using the Enterprise Access Builder (EAB) provided by VisualAge for Java.

This chapter focuses on writing non-managed applications; that is, an application server is not managing the connection to the resource adapter. It is recommend that you first develop a non-managed application for simplicity before writing and deploying to a managed environment. For more information on the managed environment, see Chapter 6, “CCI applications in a managed environment” on page 111.

The following information on how to develop a simple application is in this chapter:

- Using the CCI
- Using the Enterprise Access Builder
- Asynchronous calls
- Extended logical units of work
- Exception handling

All the sample code for this chapter is available in the package `itso.cics.j2ee`.

5.1 Using the CCI

The Common Client Interface (CCI) defines an application programming interface (API) for resource adapters connecting to Enterprise Information Systems (EIS). This section focuses on using this interface directly. Alternative tooling, such as VisualAge for Java, can generate CCI code. This is discussed in 5.2, "Using the Enterprise Access Builder" on page 79.

The CICS ECI resource adapter class diagram (Figure 5-1) shows the interactions between the different J2EE Connector Architecture classes used to execute a transaction, and their relationship to the CCI. The class names in bold are those used in an ECI program call. The methods used to drive an interaction are inherited by these classes from the interfaces provided by the CCI.

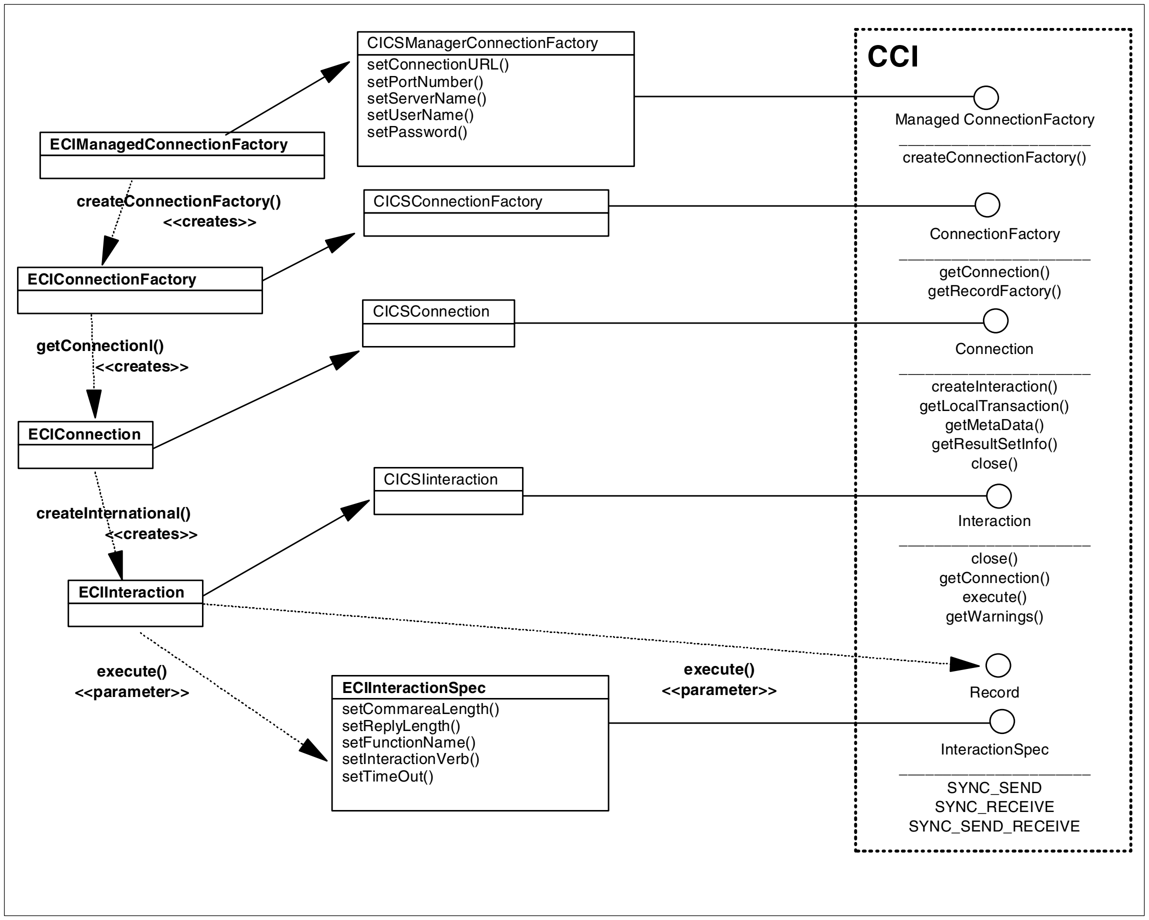


Figure 5-1 CICS ECI resource adapter class diagram

5.1.1 Writing a simple CCI application

This section describes how to write a simple Java CCI program that calls a CICS COMMAREA based program, passing a Java Record as input and output. The tasks the program performs are illustrated in Figure 5-2. The class `EciprogTest` in the package `itso.cics.eci.j2ee` is a completed solution of our program. To obtain it refer to "Additional material" on page 261.

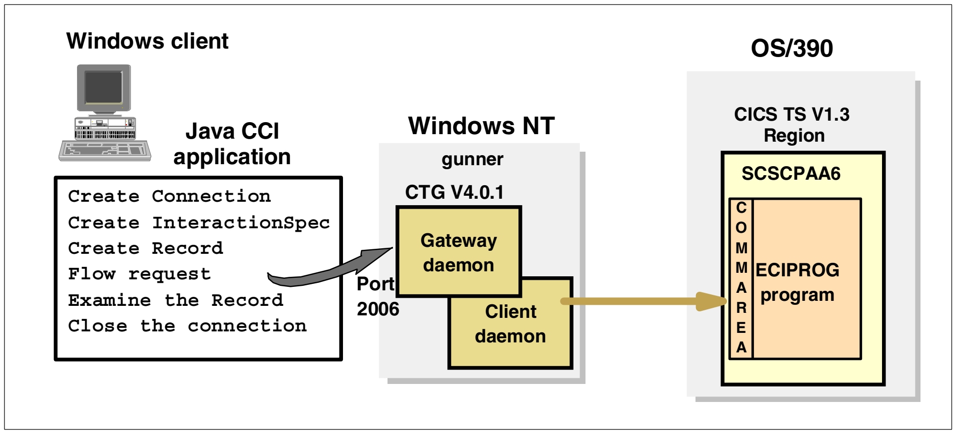


Figure 5-2 CTG scenario for ECI CCI application

Import statements

Our programs uses classes from these three Java packages:

`javax.resource.cci`	A collection of CCI interfaces, described in the J2EE Connector Architecture specification.
`com.ibm.connector2.cics`	Classes specific to the CICS resource adapters, which implement the CCI interfaces
`java.io`	The `UnsupportedEncodingException` class, which may be thrown when performing data conversion.

To use classes from these packages, you must either explicitly reference their package names in the code, or (more conveniently) use an import statement. The import statements shown below were used:

```
import com.ibm.connector2.cics.ECIManagedConnectionFactory;
import com.ibm.connector2.cics.ECIInteractionSpec;
import javax.resource.cci.ConnectionFactory;
import javax.resource.cci.Connection;
import javax.resource.cci.Interaction;
import javax.resource.ResourceException;
import java.io.UnsupportedEncodingException;
```

Connecting to CICS

Our `EciprogTest` class calls the CICS program ECIPROG on the CICS server, SCSCPAA6, using the facilities of a CTG running on the host `gunner`, listening on port 2006. This code is shown in Figure 5-3, and is available as a sample class.

```
try {
   //Create and set values for ECI managed connection factory
1  ECIManagedConnectionFactory mcf= new ECIManagedConnectionFactory();
2  mcf.setConnectionURL("tcp://gunner");
3  mcf.setPortNumber("2006");
4  mcf.setServerName("SCSCPAA6");

   //Create a connection factory connection object
5  ConnectionFactory cxnf=(ConnectionFactory)mcf.createConnectionFactory();
6  Connection cxn= cxnf.getConnection();
   //create an interaction with CICS to start program ECIPROG
7  Interaction ixn= cxn.createInteraction();
8  ECIInteractionSpec ixnSpec= new ECIInteractionSpec();
9  ixnSpec.setInteractionVerb(ixnSpec.SYNC_SEND_RECEIVE);
10 ixnSpec.setFunctionName("ECIPROG");

   //Create a new record for handling the COMMAREA byte array
11 GenericRecord record = new
      GenericRecord(("--------------------------").getBytes("IBM037"));
   System.out.println("Comm in: "+new String(record.getCommarea(),"IBM037"));

   //Finally execute and flow the request to CICS
12 ixn.execute(ixnSpec, record, record);
   //Close the interaction and the connection
13 ixn.close();
   cxn.close();
   System.out.println("Comm out: "+new String(record.getCommarea(),"IBM037"));
}catch (ResourceException re) {   System.out.println("Error: "+re.getMessage()); }
}catch (IOException ioe)      {   System.out.println();}
```

Figure 5-3   ECI CCI example EciprogTest

The logic in this code is as follows:

- 1 Instantiate the `ECIManagedConnectionFactory` class and configure the object.

  In the example, this class is directly instantiated. This class configures the main attributes of the *non-managed* connection between the Java code and the CICS Transaction Gateway, such as:

  – 2 The URL for the gateway daemon, through `setConnectionURL()`

- 3 The port number where the gateway daemon is listening, through `setPortNumber()`
- 4 The CICS server name (as known to the CTG) through `setServerName()`
- The security attributes, through `setUserName()` and `setPassword()`

Security: The user ID and password specified using `setUserName()` and `setPassword()` are flowed with each ECI request.

If using the OS/390 CTG, these can be verified within the CTG itself, and only the user ID is flowed on to CICS. If using a distributed CTG, both the user ID and password are flowed with each ECI request into CICS, and they are verified within CICS for each ECI request.

- 5 Instantiate the `javax.resource.cci.ConnectionFactory` object

  Use the `ECIManagedConnectionFactory` object that you already created to instantiate a class implementing the `ConnectionFactory` interface. To do so, invoke the method `createConnectionFactory()` on the `ECIManagedConnectionFactory` object.

Important: The J2EE Connector Architecture states that the `ConnectionFactory` object should be retrieved by making a JNDI lookup, rather than explicitly creating it in the code. JNDI is not used here for simplicity. For information on how to incorporate JNDI to lookup a `ConnectionFactory` object, see 5.5, “Using JNDI” on page 100.

- 6 Instantiate the `javax.resource.cci.Connection` class.

  Invoke the `getConnection()` method on your `ConnectionFactory` object to obtain an instance of `Connection`, configured as stated by the configuration of your `CICSManagedConnectionFactory`.

- 7 Instantiate the `javax.resource.cci.Interaction` object.

  Create an `Interaction` object by invoking `createInteraction()` on your `Connection` object. The `Connection` object is responsible for the creation of the `Interaction` object, which will execute interactions with the CICS server.

- 8 Instantiate and configure the specific object for the kind of interaction that is required.

  In this case, use the `com.ibm.connector2.cics.ECIInteractionSpec` class. This class implements the `javax.resource.cci.InteractionSpec` interface. It is used for the configuration of the specific interaction to be executed. This configuration is performed by invoking its setter methods in order to populate the attributes of the object as follows:

– **9** The type of the interaction, whether it is synchronous or asynchronous, through the `setInteractionVerb()` method

– **10** The name of the program to be run on the CICS server, through the `setFunctionName()` method

► **11** Instantiate the object implementing the `javax.resource.cci.Record` and `javax.resource.cci.Streamable` interfaces.

Create your our own `GenericRecord` class, which implements the `Record and Streamable` interfaces. `Record` is an empty interface; it has no methods to implement. The `Streamable` interface provides several methods, the most important ones being: `read(InputStream)` and `write(OutputStream)`.

```
public class GenericRecord
      implements javax.resource.cci.Streamable,
      javax.resource.cci.Record {
}
```

The tasks that this `GenericRecord` must perform are:

– Provide a byte array to hold the CICS COMMAREA. Declare a byte array for this purpose as follows:

```
private byte commarea[]=null;
```

– Provide a method to write data into the data structure supporting the COMMAREA storage.

Use the inherited method `read()` from the `Streamable` interface.

```
public void read(java.io.InputStream in) throws java.io.IOException {
   commarea[]=null byte[in.available()];
   in.read(commarea);
}
```

Tip: When supplying data to the `Record`, (from the `Record's` point-of-view) the `Record` is reading the data. To provide the `Record` with data, you should invoke its `read()` method, so that the `Record` is able to get the data from the `InputStream` you will supply it with.

– Provide a method for the CCI to read the COMMAREA data from the `Record`, and to flow it in the `ECIInteractionSpec` object. Use the inherited method `write()` from the `Streamable` interface for this:

```
public void write(java.io.OutputStream out) throws
      java.io.IOException {
   out.write(commarea);
   out.flush();
}
```

For simplicity, create a constructor, and a getter and setter to work with the *commarea* field as a byte array, and not as an `InputStream` or `OutputStream`.

```
public GenericRecord(byte[] comm) {
   setCommarea(comm);
   }
public void setCommarea(byte[] comm) {
try {
   read(new java.io.ByteArrayInputStream(comm));
   }catch (java.io.IOException ioe) {
   }
}
public byte[] getCommarea() {
   return commarea;
}
```

Tip: Note that in our example, we pass the constructor a byte array of length 27 containing contain '-' chars. Actually, ECIPROG does not require any input, so this input COMMAREA is just ignored. We only supply this specific byte array to clarify the input.

Use a `clone()` method, as this method also is defined in the `javax.resource.cci.Record` interface.

```
public Object clone() throws CloneNotSupportedException{
    return super.clone();
}
```

When supplying the byte array to the `GenericRecord` constructor, use the EBCDIC code page (IBM037), in order to convert data from the Unicode string to an EBCDIC byte array expected by CICS on S/390.

Note: Instead of developing a custom `GenericRecord`, you can use the EAB feature of VisualAge for Java to generate `Records` using the *Import COBOL to Record Type* SmartGuide. This provides getters and setters for accessing each Record within the COMMAREA, and built-in data conversion for all numeric and character data types. For further details on how to use the EAB refer to 5.2.1, "Creating a Record out of a COMMAREA" on page 79.

- **12** Execute the transaction.

The `execute()` method invoked on the `Interaction` object opens the actual TCP/IP socket connection to the gateway daemon, and causes the ECI request to be flowed to CICS. Three parameters must be supplied to the `execute()` method as follows:

- An `InteractionSpec`, created previously
- A `Record` object as both input and output

Restriction: Notice that even though we are using the same `Record` for input and output, we used the `execute()` method with a three parameter signature since the CICS ECI resource adapter does not support the two parameter `execute()` method.

- **13** Finally, close both the `Interaction` and the `Connection`.

  The `close()` method on the `Connection` object is the final flow, and in doing so, closes the underlying TCP/IP socket connection to the gateway daemon. It must be executed after the `close()` method on the `Interaction`.

The output for this sample code is shown in Example 5-1.

Example 5-1 EciProgTest output

```
Comm in: ---------------------------
Comm out: SCSCPAA6 13/12/01 22:13:11
```

5.1.2 Tracing

CCI tracing is set using the `ECIManagedConnectionFactory` object. Two steps must be taken to enable tracing:

1. Turn logging on through the `setLogWriter()` method.
2. Specify the level of tracing using the `setTraceLevel()` method as shown:

```
ECIManagedConnectionFactory mcf = new ECIManagedConnectionFactory();
mcf.setLogWriter(new java.io.PrintWriter(System.out));
mcf.setTraceLevel(new Integer(mcf.RAS_TRACE_INTERNAL));
```

Valid levels of tracing are as follows, with each level of trace building upon the previous level, therefore, ENTRY_EXIT includes everything in ERROR_EXCEPTION, and INTERNAL includes all trace levels:

RAS_TRACE_OFF	Disable all tracing
RAS_TRACE_ERROR_EXCEPTION	Output exception trace stacks
RAS_TRACE_ENTRY_EXIT	Output method entry and exit stack traces
RAS_TRACE_INTERNAL	Output debug trace entries

Tip: We found that the CCI trace output was not particularly useful as it only traced the execution of methods within the CCI itself. Therefore, we suggest that you use the `com.ibm.ctg.client.T` class if you wish to debug your application. For details on using the `T` class refer to 4.6, “Tracing” on page 61.

5.2 Using the Enterprise Access Builder

The Enterprise Access Builder (EAB) is a tool provided by VisualAge for Java, and is capable of automatically creating Record objects out of COMMAREA definitions, and creating Command beans that encapsulate all the interaction CICS. The EAB was originally built to use the Common Connector Framework (CCF), but there is also support for the beta set of J2EE Connectors.

Attention: IBM has just released a new development toolkit, named *WebSphere Studio Application Developer Integration Edition*. This provide for integration of J2EE applications with Enterprise Information Systems, as well as Web services. This supersedes the function provided by the EAB in VisualAge for Java, and embeds the complete technology provided by WebSphere Studio Application Developer. This new toolkit officially supports the J2EE connectors, but does allow you to run your existing CCF connector applications in the new environment. Tooling for modifying your CCF application is supported only by the EAB. For further refer to the URL:

`http://www.ibm.com/software/ad/studiointegration/`

5.2.1 Creating a Record out of a COMMAREA

The most likely scenario when connecting to CICS is to call an existing CICS program, passing a COMAREA as input and output. The COMMAREA is the data interface for all ECI calls, and will often be a complex structure with many different types of Records within it. The content of these Records will be what determines the course of action to be taken by the called CICS program.

The EAB tool uses the Java Record Framework to import the COMMAREA as a class implementing the `javax.resource.cci.Record` interface. This class provides a series of getter and setter methods to access fields within the COMMAREA, and can also be used to perform data conversion.

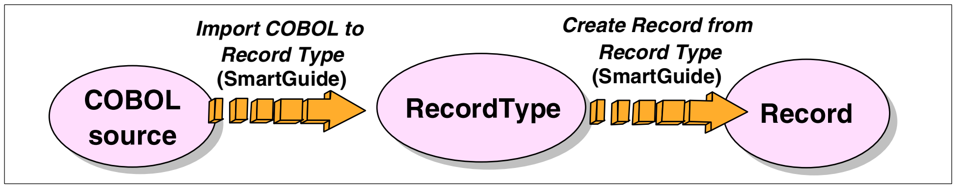

COBOL source
Import COBOL to Record Type (SmartGuide)
RecordType
Create Record from Record Type (SmartGuide)
Record

Figure 5-4 EAB Record generation

Note: For instructions on installing the EAB feature, and the J2EE connector support in Visual Age for Java, refer to Appendix A, "Configuring the CICS connectors in VisualAge for Java" on page 219.

The COMMAREA declaration for our sample ECIPROG CICS application is shown in Figure 5-5. (For the complete COBOL source refer to Appendix C.2, “ECIPROG” on page 244).

```
01 DFHCOMMAREA.
   03 APPLID                    PIC X(8).
   03 FILLER-1                  PIC X(1) VALUE SPACE.
   03 DATE-AREA                 PIC X(8).
   03 FILLER-2                  PIC X(1) VALUE SPACE.
   03 TIME-AREA                 PIC X(8).
   03 FILLER-3                  PIC X(1) VALUE SPACE.
```

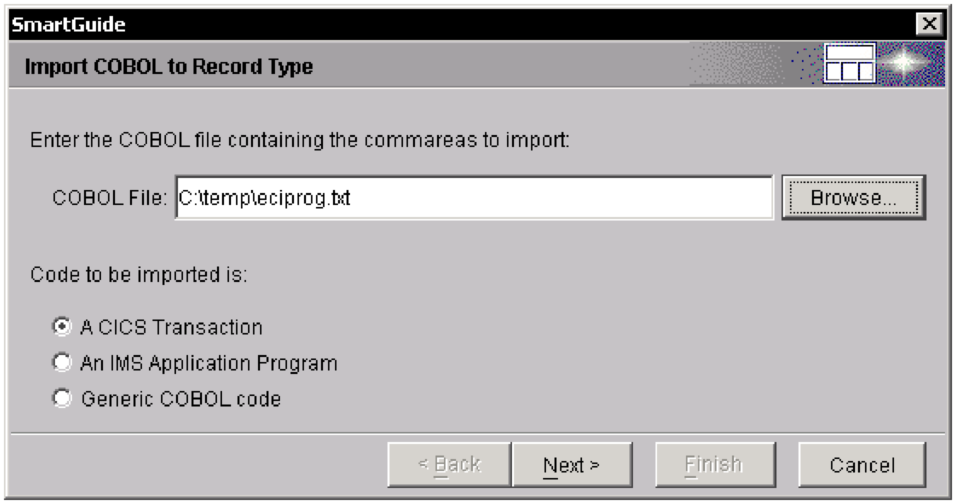
Figure 5-5 COMMAREA for ECIPROG

ECIPROG is a sample that takes no input, and merely returns the CICS region APPLID, and the date and the time in a 27 character string. Therefore, our EAB generated `Record` class will provide getter and setter methods for each of the fields: APPLID, FILLER-1, DATE-AREA, FILLER-2, TIME-AREA; and FILLER-3 (present in the original COMMAREA.)

Restriction: The EAB tool only supports COBOL coded CICS programs. If you wish to use it with PL/I, Assembler, or C CICS programs, you will need to create a dummy COMMAREA structure in COBOL and use this with EAB, or use the Record editor to build a Record from scratch.

You must import the COMMAREA into VisualAge for Java using the EAB tool. You will need a project and a package for the class to reside in; both of them must be open editions. In this book, the project *CICS Connectors Redbook* and a package named `itso.cics.eci.j2ee` is used. Once you have a `RecordType` class, process it to create the `Record` class responsible for the exchange of data with the CICS server. The following steps are needed to create a `RecordType`:

1. This is the first step is to create the `RecordType` from the COBOL COMMAREA:
   a. In VisualAge for Java, select **Workspace -> Tools -> Enterprise Access Builder -> Import COBOL to Record Type**. A SmartGuide will open to guide you through the process, starting with the screen shown in Figure 5-6.


SmartGuide

Import COBOL to Record Type

Enter the COBOL file containing the commareas to import:

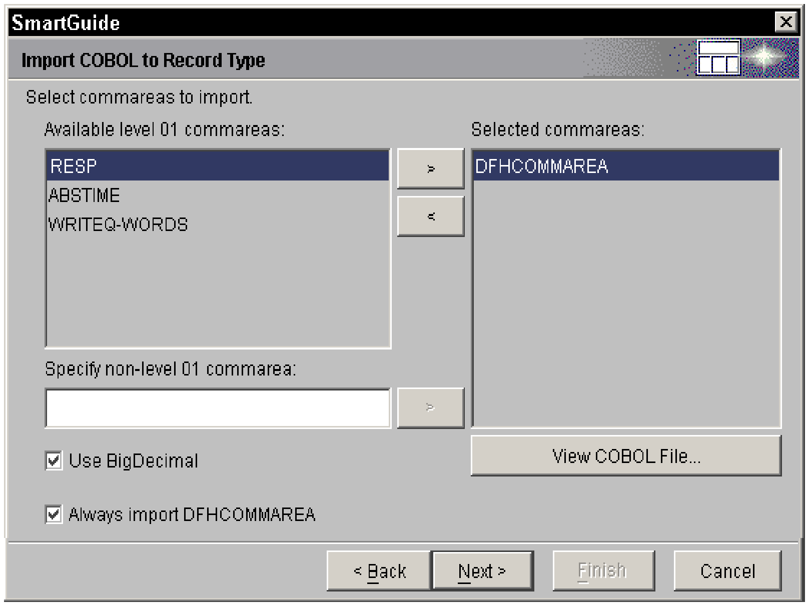
COBOL File: C:\temp\eciprog.txt

Code to be imported is:

[✓ A CICS Transaction]
An IMS Application Program
Generic COBOL code

Figure 5-6 Import COBOL to Record Type, part 1

b. Provide the name of the COBOL file containing the COMMAREA declaration. This example uses the file `eciprog.txt`. Select **A CICS Transaction** as **Code to be imported**. Then click **Next**. The window in Figure 5-7 will be displayed.


SmartGuide

Import COBOL to Record Type

Select commareas to import.

Available level 01 commareas:
RESP
ABSTIME
WRITEQ-WORDS

Selected commareas:
DFHCOMMAREA

Specify non-level 01 commarea:

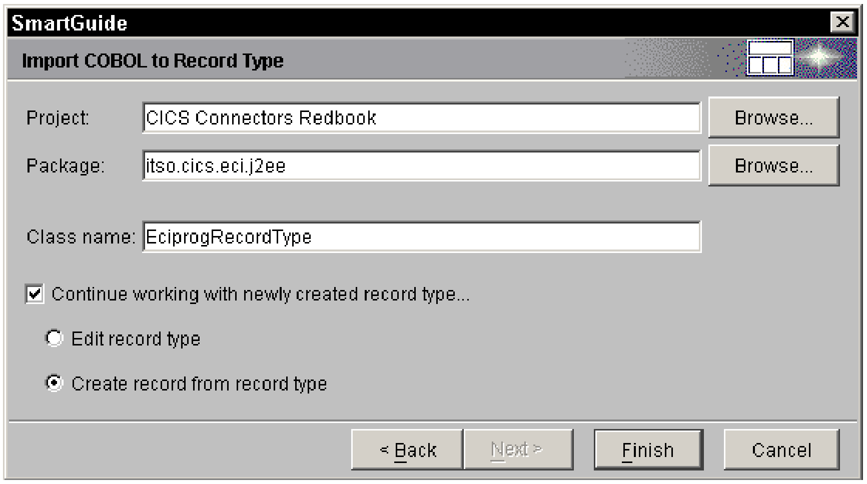
[✓] Use BigDecimal

[✓] Always import DFHCOMMAREA

Figure 5-7 Import COBOL to Record Type, part 2

c. The screen in Figure 5-7 shows the correct values by default, so nothing needs to be changed. Nevertheless, the left pane should contain all level-one data structures declared in the COBOL source code, and the right pane has what you want to import into VisualAge for Java. By selecting a data structure on the left and clicking on the left to right arrow (**>**)button on top of the screen, you add data to be imported as a `RecordType`. Select the checkbox **Always import DFHCOMMAREA** and the SmartGuide will automatically import the COMMAREA declaration in your COBOL files. When finished, click **Next**. The window on Figure 5-8 is displayed.

Attention: Notice that all COMMAREA declarations in any COBOL source code file have the same name: DFHCOMMAREA. This does not mean that you should always import the COMMAREA declared as DFHCOMMAREA, because in some COBOL programs the COMMAREA appears as an unformatted field that later on will be formatted into the proper data structure.



Project: CICS Connectors Redbook

Package: itso.cics.eci.j2ee

Class name: EciprogRecordType

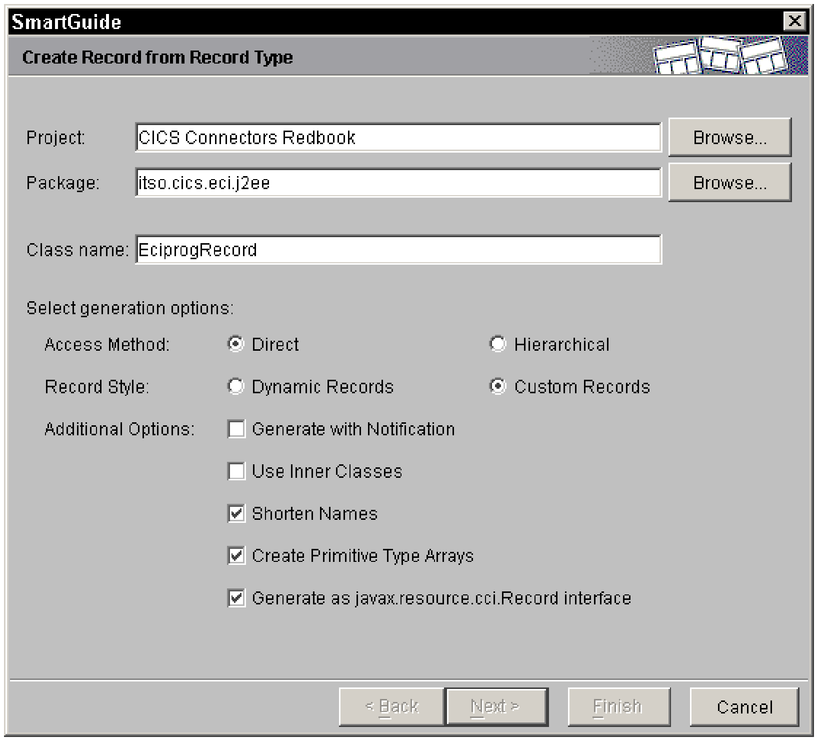
[✓] Continue working with newly created record type...

○ Edit record type

[● Create record from record type]

Figure 5-8 Import COBOL to Record Type, part 3

d. Finally, specify the project and package where you want to create your new Record type and name the `RecordType`: *EciprogRecordType*. Click **Finish**. The new `EciprogRecordType` class will now be added to the `itso.cics.eci.j2ee` package in your workspace, and the *Create Record from Record Type Smart Guide* will be started (see Figure 5-9.)


SmartGuide

Create Record from Record Type

Project: CICS Connectors Redbook

Package: itso.cics.eci.j2ee

Class name: EciprogRecord

Select generation options:

Access Method: [✓ Direct] Hierarchical

Record Style: Dynamic Records [✓ Custom Records]

Additional Options:

- [ ] Generate with Notification
- [ ] Use Inner Classes
- [x] Shorten Names
- [x] Create Primitive Type Arrays
- [x] Generate as javax.resource.cci.Record interface

Figure 5-9 Create Record from Record Type, part 1

e. You will see that the project and package text boxes are already filled in; these can be changed as needed, but in this example create your `Record` class in the same package as the `RecordType` class. Please complete the following steps:

   i. You must provide a name for your `Record` class. Name it *EciprogRecord*.

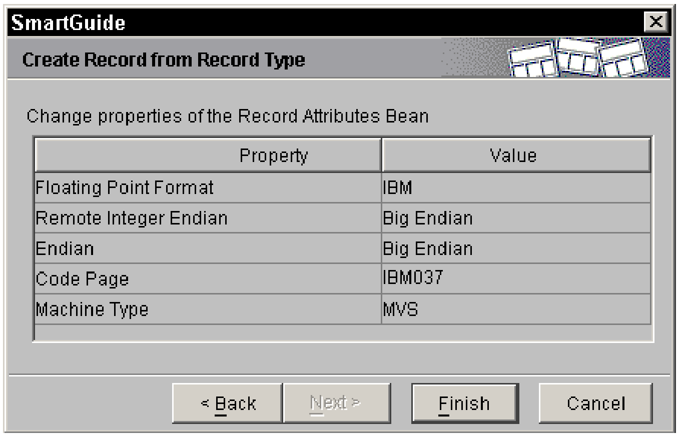
   ii. Select **Access Method** to **Direct**. This flattens a Record and puts all fields, including nested ones, at the same access level.

   iii. Select **Record Style** to **Custom**. This option generates offset information into the code itself, rather than placing the offset information outside the generated code. When the offset information is placed outside the code, there is a lookup cost to find the offset information each time a request for data is made.

   iv. Select **Shorten Names;** this option creates names that are more readable.

   v. Select **Create Primitive Type Arrays**; this is a more efficient way for accessing arrays with a primitive data type.

vi. Be sure to select **Generate as javax.resource.cci.Record interface**, as the interface for your Records. This is an interface required for your Record to be J2EE compliant. When finished, click **Next,** and the screen on Figure 5-10 appears.



Change properties of the Record Attributes Bean

Property	Value
Floating Point Format	IBM
Remote Integer Endian	Big Endian
Endian	Big Endian
Code Page	IBM037
Machine Type	MVS

Figure 5-10 Create Record from Record Type, part 2

f. Next, configure how data conversion is to be performed with the following steps:

i. Change *Floating Point Format* to *IBM*.

ii. Change *Remote Integer Endian* and *Endian to Big Endian*.

iii. Set the *Code Page* to *IBM037*, or another EBCDIC code page suitable for mainframe CICS.

iv. And finally, set the *Machine Type* to MVS.

v. This set of values provides the correct data conversion for most data types. Click **Finish** to end.

Tip: For further details on what all these data conversion parameters really mean, refer to Appendix B, “Data conversion” on page 227.

Finally, if you take a look at your package, two new classes should have been created: `EciprogRecord` and `EciprogRecordBeanInfo`. If you explore the methods on the `EciprogRecord` class you will notice that it has setters and getters for the fields that were declared in the COMMAREA.

The code in our previous EciprogTest example (Figure 5-3 on page 74) can now be modified to use this new `Record` class as shown in Figure 5-11.

```
....
EciprogRecord record= new EciprogRecord();
record.setRawBytes(("---------------------------").getBytes("IBM037"));

//make the call to CICS
...

//examine the output record
System.out.println("APPLD: "+record.getApplid());
System.out.println("DATE: "+record.getDate__Area());
System.out.println("TIME: "+record.getTime__Area());
....
```

Figure 5-11 ECI CCI using EAB Record

This sample code is provided in package `itso.cics.eci.j2ee` in classes: `EABEciProgTest`, `EciprogRecord`, `EciprogRecordBeaninfo` and `EciprogRecordType`.

The output from the running of `EABEciProgTest` is shown in Example 5-2.

Example 5-2 EABEciprogTest output

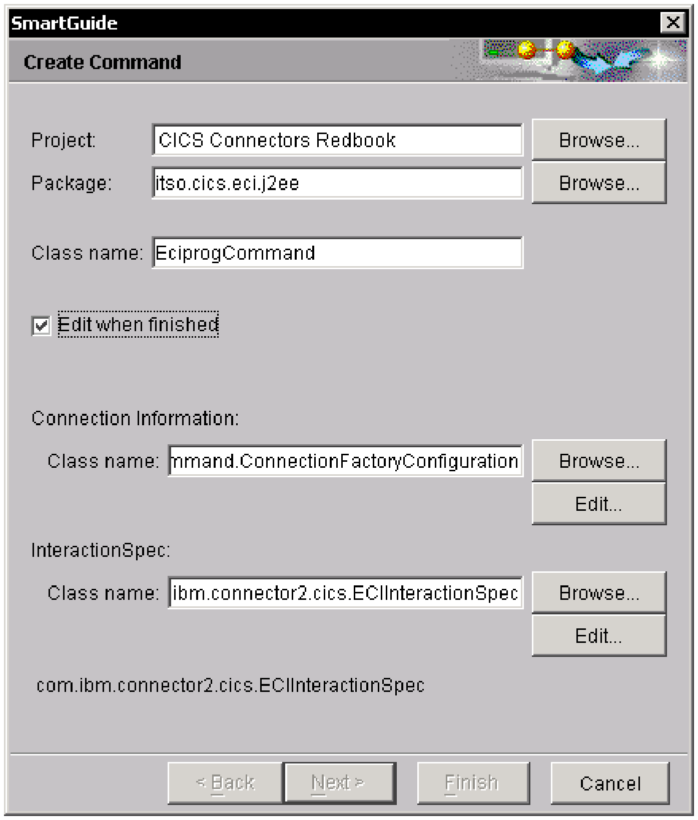
```
APPLID:SCSCPAA6
DATE: 03/12/01
TIME: 19:55:13
```

5.2.2 Creating a Command bean

The EAB is not only capable of abstracting data, but can also encapsulate all the coding through the use a *Command bean*. A Command bean encapsulates all the necessary coding of an ECI request, and allows a Java literate programmer to swiftly and easily invoke a CICS program without having to understand the interactions involved. This is achieved using the EAB *Create Command SmartGuide*. In our case, our CCI code example (Figure 5-3 on page 74) can be significantly reduced to a few lines by encapsulating that code in a Command bean.

Before creating a Command bean, you need to provide the input and output Records (which might be the same) for the interaction. These should have been built previously using the EAB.

1. In VisualAge for Java, select **Workspace -> Tools -> Enterprise Access Builder -> Create Command**. The *Create Command SmartGuide* shown in Figure 5-12 will open to guide you through the process.



SmartGuide

Create Command

Project: CICS Connectors Redbook

Package: itso.cics.eci.j2ee

Class name: EciprogCommand

[✓] Edit when finished

Connection Information:

Class name: mmand.ConnectionFactoryConfiguration

InteractionSpec:

Class name: ibm.connector2.cics.ECIInteractionSpec

com.ibm.connector2.cics.ECIInteractionSpec

Figure 5-12 Create Command 1

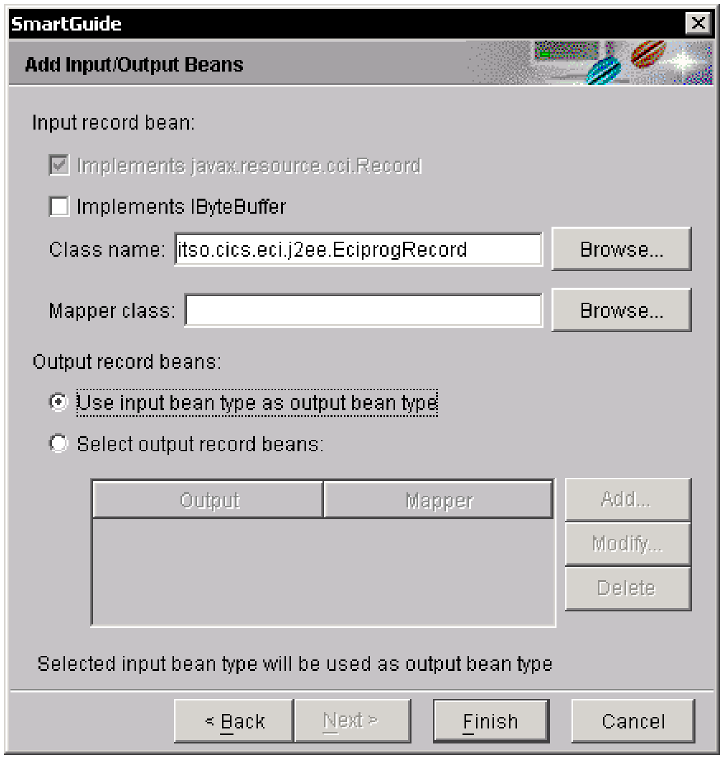
a. Select the **CICS Connector Redbook** project and the **`itso.cics.eci.j2ee`** package as the place to create the new Command bean. Name the Command bean `EciprogCommand`.

b. Select the **`com.ibm.ivj.eab.command.ConnectionFactoryConfiguration`** class in the Connection information text box (click the **Browse** button for this task). This class will be used to configure the connection to the CICS server.

> **Attention:** Do *not* set the Connection to `CICSConnectionSpec`, rather use the `ConnectionFactoryConfiguration` class. Using the `CICSConnectionSpec` will cause your Command bean to use the CCF-based connector, and not the new J2EE connector.

c. Select the **`com.ibm.connector2.cics.ECIInteractionSpec`** class in the InteractionSpec text box (click the **Browse** button for this task).

d. Select **Next** and the *Add Input/Output Beans* windows appears.


SmartGuide

Add Input/Output Beans

Input record bean:

[✓] Implements javax.resource.cci.Record

[ ] Implements IByteBuffer

Class name: itso.cics.eci.j2ee.EciprogRecord

Mapper class:

Output record beans:

(●) Use input bean type as output bean type

( ) Select output record beans:

Output	Mapper

Selected input bean type will be used as output bean type

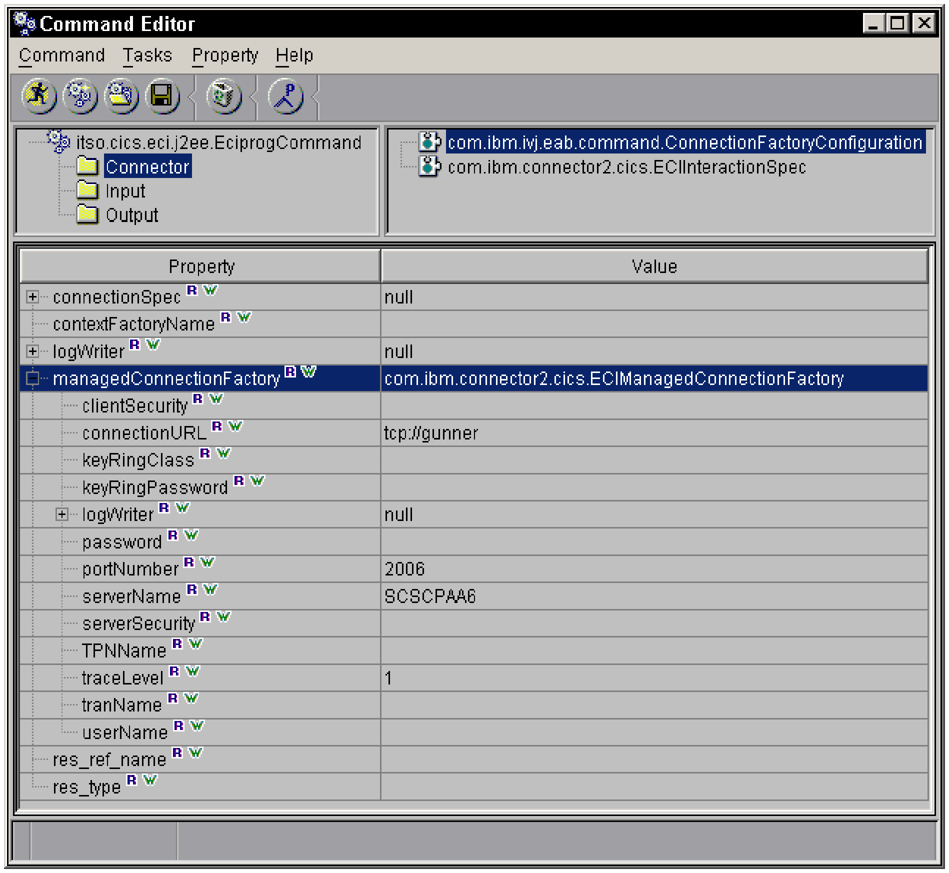
Figure 5-13 Create Command 2

2. In the *Add Input/Output Beans* window shown in Figure 5-13, the *Implements java.resource.cci.Record* checkbox at the top appears checked, indicating that you are building a J2EE Connector Architecture compliant Command bean. Next, tell the SmartGuide the name of the Record to be used as input for the ECI call. Use the `EciprogRecord` class that was previously created.

   a. First, click the **Browse** button next to the text box labeled *Class name* and select the **`itso.cics.eci.j2ee.EciprogRecord`** class.

   b. Since you are using the same Record for input and output, select **Use input bean type as output bean type**. The other possible choice is to select **Output record beans**. This would be used when the COMMAREA returned by the CICS program is different to the input COMMAREA.

   c. When finished click **Finish,** which invokes the EAB *Command Editor* as shown in Figure 5-14.



itso.cics.eci.j2ee.EciprogCommand
- Connector
- Input
- Output

com.ibm.ivj.eab.command.ConnectionFactoryConfiguration
com.ibm.connector2.cics.ECIInteractionSpec

Property	Value
connectionSpec	null
contextFactoryName	
logWriter	null
managedConnectionFactory	com.ibm.connector2.cics.ECIManagedConnectionFactory
clientSecurity	
connectionURL	tcp://gunner
keyRingClass	
keyRingPassword	
logWriter	null
password	
portNumber	2006
serverName	SCSCPAA6
serverSecurity	
TPNName	
traceLevel	1
tranName	
userName	
res_ref_name	
res_type	

Figure 5-14 EAB Command Editor 1

3. At this stage, the Command bean has been generated. Now, set the correct properties on the Command bean. First you need to set the connection factory fields:

   – Highlight **Connector** in the top left pane, then select **com.ibm.ivj.eab.command.ConnectionFactoryConfiguration** in the top right frame. This will display properties and values relevant to the connection to CICS.

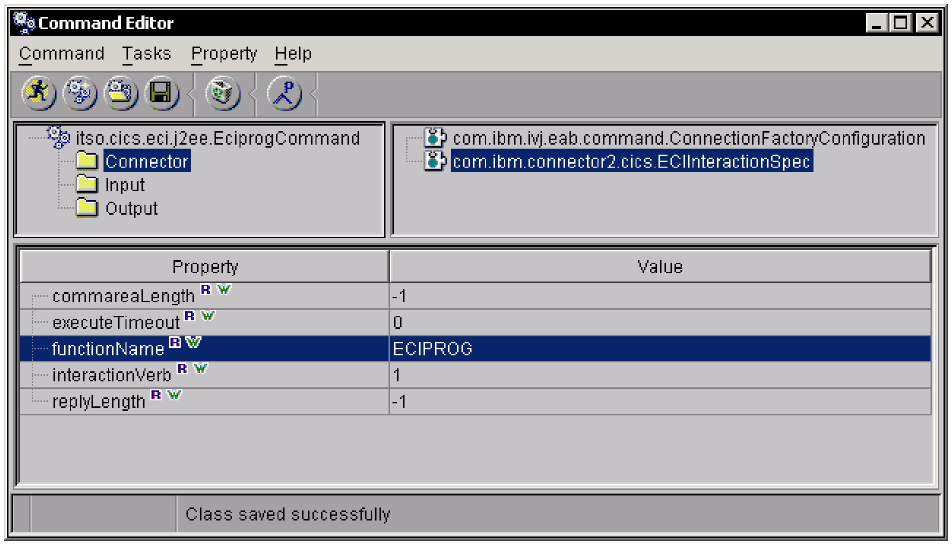
   – Select the **null value** for the managedConnectionFactory property. This presents a pull-down menu of valid managed connection factories. Select **ECIManagedConnectionFactory**.

   – To the left of the managedConnectionFactory property click on **+** to expand this property. Numerous sub-properties of managedConnectionFactory are displayed. Set the following:

- Set the connectionURL to point to the gateway daemon. We used `tcp://gunner` for our CTG. If the protocol is omitted, `tcp` is assumed.
- Note the portNumber is by default `2006`. This has not been changed.
- Change the serverName to the CICS server name where ECIPROG will run. We used SCSCPAA6.

Important: The J2EE Connector Architecture states that the `ConnectionFactory` object should be retrieved by making a JNDI lookup, rather than explicitly creating it in the code. We did not use the JNDI in our example for reasons of simplicity. For information on how to utilize the JNDI to perform the lookup of the `ConnectionFactory` object, refer to 5.5, "Using JNDI" on page 100.

4. Now, set the InteractionSpec properties. To do this click on **`com.ibm.connector2.cics.ECIInteractionSpec`** in the top right pane. We wanted to call the program ECIPROG, so we set the `functionName` property to ECIPROG (Figure 5-15).


Command Editor
Command
Tasks
Property
Help
itso.cics.eci.j2ee.EciprogCommand
Connector
Input
Output
com.ibm.ivj.eab.command.ConnectionFactoryConfiguration
com.ibm.connector2.cics.ECIInteractionSpec

Property	Value
commareaLength	-1
executeTimeout	0
functionName	ECIPROG
interactionVerb	1
replyLength	-1

Class saved successfully

Figure 5-15 EAB Command Editor 2

5. It is necessary to expose any *Record* fields that you wish the user of the Command bean to be able to have access to, or set. To expose the `applid`, `date_Area`, and `time_Area` output Record fields do as follows:
   – Highlight **Output** in the top left pane, then highlight **`itso.cics.eci.j2ee.EciprogRecord`** in the top right pane.

- Right click on the **applid** property and select **Promote Property**. This adds a green dot next to applid to indicate that a getter and setter method will be generated for it.
- Promote date_Area and time_Area using the same method.

6. The Command bean is now complete. To save the values set using this editor select **Command -> Save**.

Now, lets take a look at package `itso.cics.eci.j2ee` in project *CICS Connectors Redbook*. You should now discover two new classes: `EciprogCommand` and `EciprogCommandBeanInfo`. It is a simple matter to create an application to invoke the EciprogCommand bean, and retrieve the data from EciprogRecord. This application is shown in Figure 5-23 on page 98.

```
....
try {
     //Create the command and execute it
     EciprogCommand command = new EciprogCommand();
     command.execute();

     //Get the results from the getters in the Record
     System.out.println("APPLID:" + command.getApplid());
     System.out.println("DATE: " + command.getDate__Area());
     System.out.println("TIME: " + command.getTime__Area());

     } catch (Exception e) {
          System.out.println(e);
     }....
```

Figure 5-16 Using a Command bean

The promoted fields of the output Record can be accessed individually through automatically generated setters and getters in the Command bean, and no data conversion is required.

You will find this code as the sample class `CommandBeanTest` in the package `itso.cics.eci.j2ee` in the sample code supplied with this book. The output of this test application will be the same as shown in the `EABEciprogTest` in Example 5-2 on page 85.

5.2.3 Migrating a CCF application

CCF compliant Records and commands can be migrated to the J2EE Connector Architecture using an EAB SmartGuide. Before migrating any classes, it is recommended that you version them *first* in VisualAge for Java.

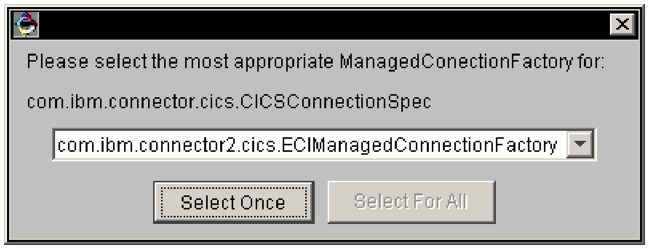
The command migrator performs the following actions:

- Changes the connection information (if any was supplied) from using the CCF-based `CICSConnectionSpec` class, to the J2EE Connector Architecture based `ConnectionFactoryConfiguration` class.
- Changes the interaction spec from the CCF-based `ECIInteractionSpec` class, to the J2EE Connector Architecture based `ECIInteractionSpec` class.

The Record migrator performs the following actions:

- Modifies the class to implement the `javax.resource.cci.Record` and `javax.resource.cci.Streamable` interfaces.
- Implements the methods that these interfaces provide.

To migrate a command, right click on it and select **Tools -> Enterprise Access Builder -> Migrate Commands**. The *Migrate to Connector Architecture* SmartGuide will appear. You can optionally add additional commands to migrate here. Click **Finish** to start the migration. If the command contains connection information, the window shown in Figure 5-17 will appear.


Please select the most appropriate ManagedConectionFactory for:

com.ibm.connector.cics.CICSConnectionSpec

com.ibm.connector2.cics.ECIManagedConnectionFactory

Figure 5-17 Migrating a command

Select **ECIManagedConnectionFactory**.

The following CICSConnectionSpec properties cannot be migrated:

- reapTime
- realm
- unusedTimeout
- maxConnections
- minConnections
- connectionTimeout

If any of these properties have been set, a warning will be issued during the migration. The VisualAge for Java log will list all populated properties omitted from the migration.

Once the command migration is complete, you will be asked if you wish to migrate all associated input and output Records. You should answer `Yes` to this, otherwise, using CCF-based Records in a migrated Command bean will cause `java.lang.ClassCastException` to be thrown at runtime.

5.3 Asynchronous calls

The support for asynchronous calls is built into the CICS ECI resource adapter. This allows your Java application to call a CICS program without blocking, while waiting for the return of a response from CICS.

However, asynchronous calls using the CICS ECI resource adapter also have their limitations. You cannot make several concurrent calls, and then wait for the response. You must take the response of each previous call, before making another call. This applies to all the calls made on the same CTG connection.

Restriction: We found during testing that the OS/390 CTG does *not* support asynchronous ECI calls when using the CCI. In this situation, the error CTG9631E, ECI_ERR_INVALID_CALL_TYPE is returned. This restriction should be removed in a subsequent release of the CTG. This restriction does not apply if the CTG is running on other platforms, and does not apply if using asynchronous calls with the `ECIRequest` class.

The code shown in Figure 5-18 is an extension of our earlier application `EciprogTest` in Figure 5-3 on page 74. It shows the changes require to make a single asynchronous call, and is described in the following text.

```
   Interaction ixn= cxn.createInteraction();
   ECIInteractionSpec ixnSpec= new ECIInteractionSpec();

   //Configure the first parameter
1  ixnSpec.setInteractionVerb(ECIInteractionSpec.SYNC_SEND);
   ixnSpec.setFunctionName("ECIPROG");
   //...a Record, the second parameter
   EciprogRecord record= new EciprogRecord();

   //And finally execute!
2  ixn.execute(ixnSpec, record, null);
   //Now prepare to get the response for the server
3  ixnSpec.setInteractionVerb(ixnSpec.SYNC_RECEIVE);
4  ixnSpec.setCommareaLength(record.getSize());
5  ixn.execute(ixnSpec, null, record);
   //Close both the interaction and the conection
   ixn.close();
   cxn.close();
System.out.println("APPLID:" + record.getApplid());
System.out.println("DATE: " + record.getDate__Area());
System.out.println("TIME: " + record.getTime__Area());
```

Figure 5-18 Sample code for synchronous calls

Attention: This sample is incomplete since only the code required to implement asynchronous calls is shown. It is meant to supersede the example in Figure 5-11 on page 85.

This sample does not use any data conversion, so no `IOException` may be thrown. Therefore, you would need to remove the `catch` statement for this exception from the previous example. Also, do not forget to specify the same import statements that were used previously.

- **1** Notice that the interaction verb has changed from the previous Figure 5-3 on page 74. Now, instead of a `SYNC_SEND_RECEIVE` (send-and-wait for the response) use `SYNC_SEND` (send-without-waiting).
- **2** Now the calling of the ECIPROG program has changed from one method to two methods (line **2** send, and line **5** receive). Execute the first interaction with a `SYNC_SEND` interaction verb.
- **3** Configure the next interaction to be a `SYNC_RECEIVE` in order to receive the asynchronous response from the CTG.

- **4** Notice that you are telling the `InteractionSpec` object about the size of the COMMAREA (which is the size of the Record). Otherwise, you will get an exception.
- **5** Finally, execute the receive, and look at the response on the `Record` instance.

Sample code is provided in package `itso.cics.eci.j2ee` in the class `AsyncEciJca`. The output for this example is the same as that in Example 5-2 on page 85.

Figure 5-19 shows the code for making concurrent calls. Notice that in this code each call is placed on its own `Connection` object.

```
...
Connection cxnA= cxnf.getConnection();
Connection cxnB= cxnf.getConnection();
Interaction ixnA= cxnA.createInteraction();
Interaction ixnB= cxnB.createInteraction();

ECIInteractionSpec ixnSpec= new ECIInteractionSpec();
ixnSpec.setInteractionVerb(ixnSpec.SYNC_SEND);
ixnSpec.setFunctionName("ECIPROG");
EciprogRecord record= new EciprogRecord();
//And finally execute!
ixnA.execute(ixnSpec, record, null);

//place another call
ixnB.execute(ixnSpec, record, null);

ixnSpec.setInteractionVerb(ixnSpec.SYNC_RECEIVE);
ixnSpec.setCommareaLength(record.getSize());
ixnA.execute(ixnSpec, null, record);
System.out.println("APPLID1:" + record.getApplid());
System.out.println("DATE1: " + record.getDate__Area());
System.out.println("TIME1: " + record.getTime__Area());

ixnB.execute(ixnSpec, null, record);
System.out.println("APPLID2:" + record.getApplid());
System.out.println("DATE2: " + record.getDate__Area());
System.out.println("TIME2: " + record.getTime__Area());

ixnA.close();
ixnB.close();
cxnA.close();
cxnB.close();
```

Figure 5-19 Sample code for concurrent calls

This code is provided in the package `itso.cics.eci.j2ee` in the class `AsyncEciJcaMultiple`. The output for this sample code is shown in Example 5-3 on page 95.

Example 5-3 AsyncEciJcaMultiple output

```
APPLID1:SCSCPAA6
DATE1: 14/12/01
TIME1: 12:56:10
APPLID2:SCSCPAA6
DATE2: 14/12/01
TIME2: 12:56:10
```

5.4 Extended logical units of work

Before considering how to extend logical units of work (LUWs), you should first know what one is. In CICS terms, a *logical unit of work* (also termed a *unit of work*) is a recoverable sequence of operations within a transaction. The start and end of which are marked by synchronization points, where the LUW is either committed, or rolled-back. If there are no explicit synchronization points within a transaction, then the start and end of the transaction are considered implicit synchronization points.

To illustrate the concept, imagine a bank transaction that consists of debiting money from one account, and crediting it to another. If you debit the money from one account, but the credit process fails, the data falls into an inconsistent state where the money was lost. So, the action of subtracting from one account and adding the same amount to another account, needs to be carried out as one atomic operation, or as a logical unit of work.

If this operation was being performed by a series of ECI calls, you would need to *extend* the LUW created by the first program, so that the second ECI call was executed within the scope of the logical unit of work that was created by the first ECI call. This is achieved within CICS by causing the mirror program that executes ECI requests to remain active, after an ECI request has terminated. (The mirror program usually terminates at the end of each ECI request.) The mirror program only terminates when a request is received to end the LUW.

The sample application ECIADDER

This section utilizes a sample COBOL application called ECIADDER. The input and output COMMAREAs are shown in Figure 5-20 and Figure 5-21.

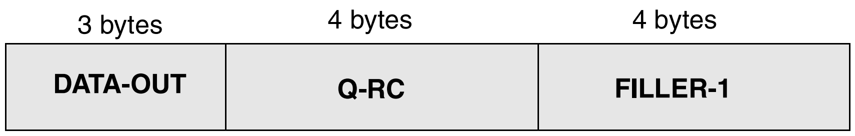


Figure 5-20 ECIADDER output COMMAREA

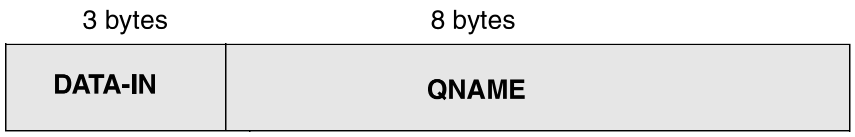


Figure 5-21 ECIADDER input COMMAREA

The code for these COBOL structures is shown below, and the full COBOL source code for the application can be found in Appendix C.1, “ECIADDER” on page 242. It is also shipped as a sample with this redbook.

```
01 COMMAREA-RETURN.
          03 DATA-OUT                   PIC S9(3) DISPLAY SIGN IS
                                        LEADING SEPARATE CHARACTER.
          03 Q-RC                       PIC ZZZ9 DISPLAY.
          03 FILLER-1                   PIC X(4) VALUE SPACE.

       LINKAGE SECTION.
       01 DFHCOMMAREA.
          03 DATA-IN                    PIC S9(3) DISPLAY SIGN IS
                                        LEADING SEPARATE CHARACTER.
          03 QNAME                      PIC X(8).
```

This COMMAREA is character-based and takes as input an integer value (`DATA-IN`) and a temporary storage queue name (`QNAME`). It adds the integer to the value already stored in the queue, and then returns the result (`DATA-OUT`) along with a return code (`Q-RC`).

You can subsequently call ECIADDER to see how the value of the queue increases (or decreases, if provided with negative integers) at each call. If you call ECIADDER (as you have learned from 5.1.1, “Writing a simple CCI application” on page 73) you will be executing single LUWs. Now, we will show how to make several calls on ECIADDER, to see how the value on the queue has increased, and finally, to undo the changes on the queue in order to bring it back to its first state, or commit the changes.

The extension of the LUW is performed using a `LocalTransaction` object obtained from the `Connection` object by invoking the `getLocalTransaction()` method. Once you have obtained the `LocalTransaction` instance, any interaction executed after the invocation of `begin()` will be part of the same LUW, and ends upon execution of a `commit()` or `rollback()` invoked on `LocalTransaction`. The Java code that calls ECIADDER is shown in Figure 5-22 and Figure 5-23. Notice that the code is not complete, because it is quite similar to that in Figure 5-3 on page 74. The complete code can be found in the sample class `XluwTest` in package `itso.cics.eci.j2ee` supplied with this book.

```
try{
   ......
   ECIInteractionSpec ixnSpec= new ECIInteractionSpec();
   ixnSpec.setInteractionVerb(ECIInteractionSpec.SYNC_SEND_RECEIVE);
   ixnSpec.setFunctionName("ECIADDER");
   XLUWInputRecord input= new XLUWInputRecord();
   XLUWOutputRecord output=new XLUWOutputRecord();

   input.setData__In((short)0);
   input.setQname("RADDINGQ");
1  ixn.execute(ixnSpec, input, output);
2  System.out.println("Queue starting value: "+output.getData__Out());

3  cxn.getLocalTransaction().begin();
   input.setQname("RADDINGQ");
   input.setData__In(10);

4  ixn.execute(ixnSpec, input, output);
   System.out.println("Queue after adding 10: " + output.getData__Out());
```

Figure 5-22 Extended calls to ECIADDER, part 1

```
5  cxn.getLocalTransaction().rollback();

   input.setData__In((short)0);
   input.setQname("RADDINGQ");
6  ixn.execute(ixnSpec, input, output);
   System.out.println("Queue after rollback: " + output.getData__Out());

7  cxn.getLocalTransaction().begin();

   input.setQname("RADDINGQ");
   input.setData__In((short)10);
8  ixn.execute(ixnSpec, input, output);
   input.setQname("RADDINGQ");
   input.setData__In((short)-7);
9 ixn.execute(ixnSpec, input, output);

10 cxn.getLocalTransaction().commit();

   System.out.println("Queue after adding (10-7):"+output.getData__Out());
...
}catch(ResourceException re){
   ....
}
```

Figure 5-23 Extended calls to ECIADDER, part 2

To test this program, you must require the ECIADDER COBOL program to be deployed on your CICS server. Once the ECIADDER program is ready to run, import the COMMAREA, declared as DFHCOMMAREA in the COBOL source, and the COMMAREA-RETURN, into VisualAge for Java as different `Records`. Name them `XLUWInputRecord` and `XLUWOutputRecord` respectively.

Note: The `DFHCOMMAREA` is the only interface between the CICS program and the rest of the world. So, why import two different `Records`?

Although the DFHCOMMAREA structure is the only interface for input and output, it is often convenient to have another structure (such as our `COMMAREA-RETURN`) that is the *same size* as the DFHCOMMAREA, but contains different fields that represent the returned data. This structure is then moved over the original COMMAREA before the program returns.

Now, let us take a deeper look at the sample code.

- **1** First of all invoke ECIADDER as usual with an input value of 0, so that you do not actually modify the value in the CICS temporary storage queue named RADDINGQ. As any call to ECIADDER returns the value in the queue, what you are really doing is just reading the queue.
- **2** This line will print the actual value stored in the queue. Assuming the value of the queue is zero, the result will be:

  ```
  Queue starting value: 0
  ```
- **3** Now, go to the `LocalTransaction`, and invoke the `begin()` method to create an extended logical unit of work.
- **4** Now call ECIADDER with the value +10, so the result will be:

  ```
  Queue after adding 10: 10
  ```

  Attention: Note that it is important to reset the queue name before every `execute()` method invocation, since the previous invocation of this method clears the value.

- **5** Now, invoke the `rollback()` method on the `LocalTransaction` instance. This will undo all the changes since the invocation of the `begin()` method.
- **6** Now, execute ECIADDER with the value 0 in order to read the queue after the rollback. The output will be:

  ```
  Queue after rollback: 0
  ```
- **7** Begin a new transactional unit of work.
- **8**,**9** Within the same logical unit of work, you will now call ECIADDER twice, once with the value +10, and once with the value -7. The output will now be:

  ```
  Queue after adding (10-7): 3
  ```
- **10** Confirm the changes made on the queue, and close the logical unit of work by invoking the `commit()` method. Now, you cannot undo the changes, and the value 3 remains in the queue.

5.5 Using JNDI

When deploying J2EE applications in a non-managed environment, there are two ways to obtain a `ConnectionFactory` object:

- Manually create a `ConnectionFactory` object:
  - Instantiate a `ManagedConnectionFactory` object.
  - Populate the `ManagedConnectionFactory` object with deployment values.
  - Use the `ManagedConnectionFactory` `createConnectionFactory()` method to create a `ConnectionFactory` object.
- Use the JNDI to lookup a `ConnectionFactory` object, which has been created earlier following the same steps as above, then bound into the JNDI namespace.

The J2EE Connector Architecture specification indicates that all applications deployed to a non-managed environment must use the JNDI to obtain a `ConnectionFactory` object. There are two reasons for this:

- Applications deployed to a managed environment use the JNDI to lookup a `ConnectionFactory` object. If applications are coded to perform this JNDI lookup, it makes moving applications between managed and non-managed environments easier.
- `ConnectionFactory` objects contain deployment values (such as the connectionURL of the resource adapter) which may frequently change. Without using JNDI, any changes to these deployment values means having to change the Java application that uses them. If JNDI is used, a new `ConnectionFactory` object can be bound to the JNDI namespace with the new deployment values, and the application that uses them remains unchanged.

Tip: We do not recommend using JNDI in a non-managed test environment for applications that will ultimately be deployed to a managed environment. Binding `ConnectionFactory` objects into a non-managed environment is a time consuming exercise, and adds unnecessary complexity. In contrast, the WebSphere Application Server V4 managed environment automates the process of binding `ConnectionFactory` objects to a JNDI namespace. However, if you are considering using a non-managed environment in production, rather than in a test, then we recommend that you do use JNDI.

The examples used in this section are based on creating a `ConnectionFactory` object for use with the CICS ECI resource adapter. They are all part of the `itso.cics.eci.j2ee.jndi` package supplied with this book. For instruction on obtaining the sample code, see Appendix D, “Additional material” on page 261.

5.5.1 Using JNDI with the CCI

There are two steps to utilize the JNDI to obtain a `ConnectionFactory` object:

1. Create and bind a `ConnectionFactory` object to the JNDI namespace.
2. Lookup the `ConnectionFactory` object in the Java application.

> **Further information:** The whitepaper "*Using J2EE Resource Adapters in a Non-managed Environment*" provides an in-depth discussion of using JNDI with the CCI in a non-managed environment. It is available at the following URL:
>
> `http://www7b.boulder.ibm.com/wsdd/library/techarticles/0109_kelle/0109_kelle.html`

Create and bind a ConnectionFactory object

For simplicity, these instructions are referred to *publishing a ConnectionFactory object in the JNDI namespace,* and *performing a lookup to retrieve that object*. In practice, a `ConnectionFactory` object is stored in a `javax.naming.Reference` object, and it is the `Reference` object that is bound into the JNDI namespace. The `Reference` object contains both the `ConnectionFactory` object, and the `javax.naming.spi.ObjectFactory` implementation, which defines how to extract the `ConnectionFactory` object from the `Reference` when it is retrieved from the JNDI namespace.

This section uses the WebSphere Application Server JNDI naming server, which is supplied with the application server, and with WebSphere Studio Application Developer as part of the WebSphere Test Environment.

To use the WebSphere JNDI naming server in a non-managed environment, the following JAR files from the `WebSphere\AppServer\lib` directory are needed in the CLASSPATH at runtime:

- `iwsorb.jar`
- `jca.jar`
- `ns.jar`
- `ras.jar`
- `ujc.jar`
- `utils.jar`
- `websphere.jar`

The class `itso.cics.eci.j2ee.jndi.ConFacPublish` contains sample code to publish a `ConnectionFactory` object to the JNDI. It consists of two methods: `main()` and `getJNDIContext()`. The `getJNDIContext()` method is shown in Figure 5-24.

```
public static Context getJNDIContext() throws NamingException {
     InitialContext ctx = null;
     Properties p = new Properties();

1    p.put( Context.INITIAL_CONTEXT_FACTORY,
                 "com.ibm.websphere.naming.WsnInitialContextFactory");
2    p.put( Context.PROVIDER_URL, "iiop://localhost:900");

     try {
        ctx = new javax.naming.InitialContext(p);
     } catch (NamingException ne) {
        System.out.println("caught NamingException getting InitialContext");
        ne.printStackTrace();
        throw ne;
     }
     return ctx;
     }
```

Figure 5-24 The getJNDIContext() method

The above code returns a `Context` object, which is used to make interactions with the JNDI namespace. The initial context factory (1) is set for that of the WebSphere JNDI naming server, and the provider URL (2) specifies how to connect to the name server.

The `main()` method performs the following functions:

- Instantiates an `ECIManagedConnectionFactory` object and populates the fields of this object with deployment values
- Uses the `ECIManagedConnectionFactory` `createConnectionFactory()` method to create a `ConnectionFactory` object
- Instantiates a `Reference` object, specifying the `ConnectionFactory` object as a byte array and the class name of the `ObjectFactory` implementation
- Uses the `getJNDIContext()` method to get the `Context` object and bind the `Reference` object to the JNDI namespace, specifying a resource reference name

Lookup a ConnectionFactory object

The class `itso.cics.eci.j2ee.jndi.EciprogTestJNDI` performs a JNDI lookup to retrieve a `ConnectionFactory` object, and uses it to create a connection to the CICS ECI resource adapter. The lookup portion of this code is shown in Figure 5-25.

```
  ConnectionFactory connFac = null;
  ConFacPublish cfp = new ConFacPublish();

  try{
1        Context ctx = cfp.getJNDIContext();
2        connFac = (ConnectionFactory)ctx.lookup("eis/gunner/SCSCPAA6");
  } catch (NamingException ne) {
  ne.printStackTrace();
  }
```

Figure 5-25 Performing the JNDI lookup

The following list summarizes the logic in this example:

- **1** The `getJNDIContext()` method (shown in Figure 5-24) is used to create a `Context` object.
- **2** The `Context` object is used to lookup the `ConnectionFactory`, using the resource reference name (`eis/gunner/SCSCPAA6`) specified when the object was bound. The class `itso.cics.eci.j2ee.jndi.CFNamingHelper` will be used to retrieve the `ConnectionFactory` from the `Reference`.

5.5.2 Using JNDI with the EAB

There are two steps for incorporating JNDI to obtain a `ConnectionFactory` object:

1. Create and bind a `ConnectionFactory` object to the JNDI namespace.
2. Lookup the `ConnectionFactory` object in the EAB Command bean.

Create and bind a ConnectionFactory object

VisualAge for Java provides a servlet called `JNDIDeployer` to bind a ConnectionFactory object into the JNDI namespace provided by the Persistent Name Server portion in the WebSphere Test Environment. Follow the instructions below to use this servlet.

- Add the *IBM Enterprise Access Builder WebSphere Samples* and *IBM WebSphere Test Environment* projects to the workspace.
- In the *IBM Enterprise Access Builder WebSphere Samples* project, open the `com.ibm.ivj.eab.sample.ws.j2ee.servlet.LookupDeployerHelper` class. This class is used to both bind and lookup references in the namespace.
- Examine the `createBinding()` method of `LookupDeployerHelper`. This method allows you to specify values for the `ECIManagedConnectionFactory` (which is used to create a `ConnectionFactory` object) and the `DefaultConnectionPoolProperties`, which is used to configure connection

pooling. Use the setter methods of these classes to set deployment values, including:

- `setServerName()`, which sets the name of the CICS server to call
- `setConnectionURL()`, which specifies the connection URL of the CTG

Additionally, you can customize the resource reference name by modifying the `rebind()` method of the `ctx` object. The supplied code sets this to CICSECI_A.

Save the changes made to the `LookupDeployerHelper` class.

- Start the WebSphere Test Environment by selecting **Workspace -> Tools -> WebSphere Test Environment**.
- In the list of servers, highlight **Servlet Engine**. Select **Edit Class Path** and select **Select All** to select all projects; then select **OK**. Click the **Start Servlet Engine** button, and wait for the servlet engine to start.
- In the list of servers, highlight **Persistent Name Server**. This will be the JNDI server used. Ensure that the *Database driver* is set to *jdbc.idbDriver*, then select **Start Name Server**.
- Once the servlet engine and name server have started, open a browser and enter the URL:

  `http://localhost:8080/servlet/com.ibm.ivj.eab.sample.ws.j2ee.servlet.JNDIDeployer`

  This will attempt to bind a reference to the JNDI namespace. The servlet will respond with the following message

  "`ConnectionFactory added to the JNDI context as: CICSECI_A` "

  This message will be displayed regardless of the success of the operation. To determine the *real* outcome of this operation, check the servlet engine thread in the VisualAge for Java console for messages:

  - A stack trace indicates an exception was thrown and the operation was not successful.
  - A message similar to the one shown in Example 5-4 indicates that the resource reference name was successfully bound to the namespace.

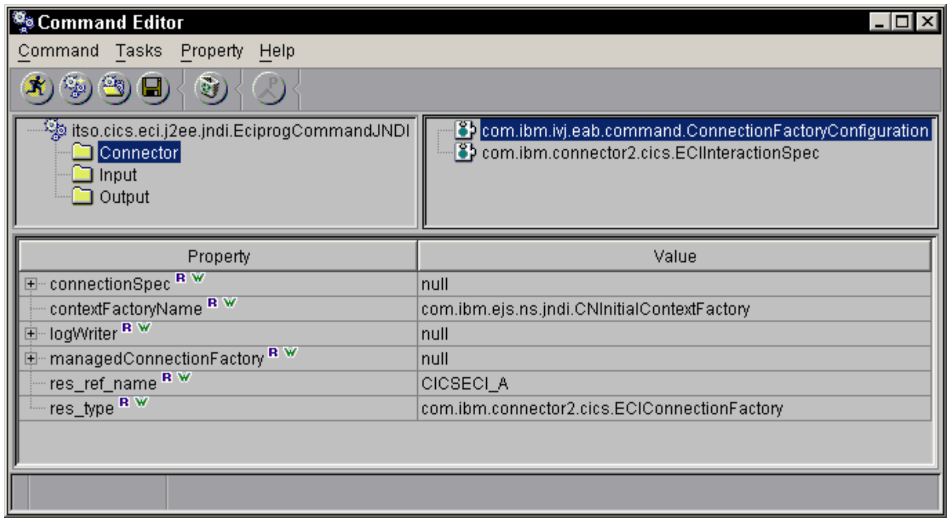
Example 5-4 Output from the JNDIDeployer servlet

```
Reference Class Name:
com.ibm.ivj.eab.sample.ws.j2ee.servlet.LookupDeployerHelper
Address Type: obj0
AddressContents: ffffffac ffffffed 0 5 73 72 0 33 63 6f 6d 2e 69 62 6d 2e 63 6f
6e 6e 65 63 74 6f 72 32 2e 63 69 63 73 2e  ...
Address Type: obj1
AddressContents: ffffffac ffffffed 0 5 73 72 0 36 63 6f 6d 2e 69 62 6d 2e 63 6f
6e 6e 65 63 74 6f 72 32 2e 73 70 69 2e 44  ...
```

5.5.3 Using JNDI with a Command bean

To use JNDI in a Command bean, perform the following steps:

- Edit the Command bean, by right clicking on it and selecting **Tools -> Enterprise Access Builder -> Edit Command**. This opens the Command Editor.
- Highlight **Connector** in the left pane, and ConnectionFactoryConfiguration in the right pane. Set the following properties, as shown in Figure 5-26:
  - Set *contextFactoryName* to *com.ibm.ejs.ns.jndi.CNInitialContextFactory*.
  - Ensure *managedConnectionFactory* is set to null. You do not need a `ManagedConnectionFactory` object because we are retrieving a `ConnectionFactory` object from the JNDI namespace.
  - Set *res_ref_name* to *CICSECI_A* (or the value you set the resource reference name to be in the `LookupDeployerHelper` class).
  - Set *res_type* to *com.ibm.connection2.cics.ECIConnectionFactory*.


Command Editor
Command
Tasks
Property
Help
itso.cics.eci.j2ee.jndi.EciprogCommandJNDI
Connector
Input
Output
com.ibm.ivj.eab.command.ConnectionFactoryConfiguration
com.ibm.connector2.cics.ECIInteractionSpec

Property	Value
connectionSpec R W	null
contextFactoryName R W	com.ibm.ejs.ns.jndi.CNInitialContextFactory
logWriter R W	null
managedConnectionFactory R W	null
res_ref_name R W	CICSECI_A
res_type R W	com.ibm.connector2.cics.ECIConnectionFactory

Figure 5-26 JNDI properties in the Command Editor

- Save the Command bean. It is now configured to use JNDI to lookup a `ConnectionFactory` object.

An example of a Command bean that uses JNDI, and a client that uses this Command bean, is included with this book in the classes `EciProgCommandJNDI` and `CommandBeanTestJNDI` in the package `itso.cics.eci.j2ee.jndi`.

5.6 Exception handling

This section analyzes the CICS resource adapter return codes and describes their possible causes. All the exceptions generated when using the ECI extend to the `javax.resource.ResourceException` class, but it is often desirable to catch the specific subclass of an exception, and use this to determine the specific error situation. Figure 5-27 shows the hierarchy of the supplied ResourceException subclasses.

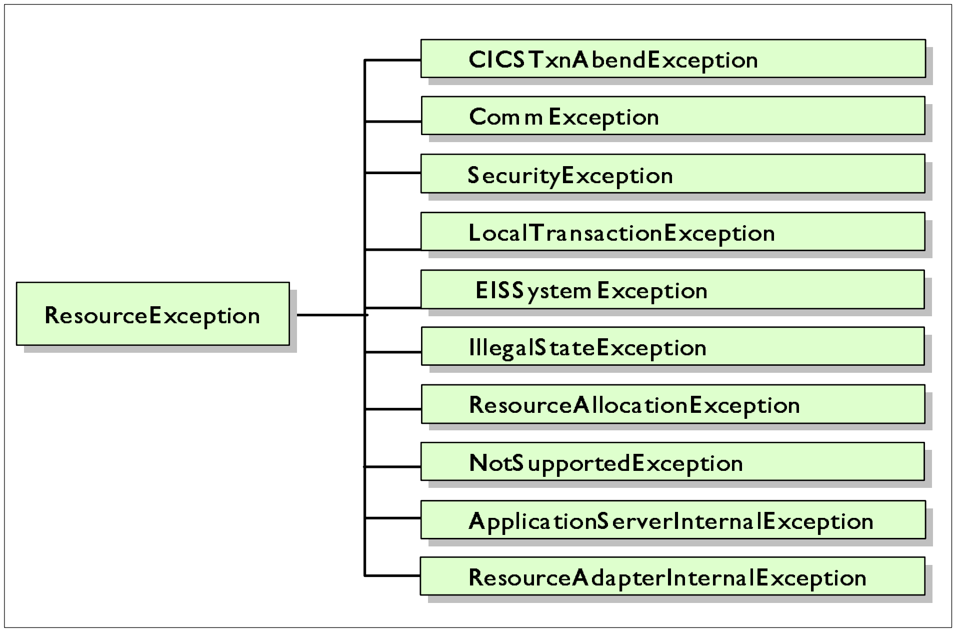


Figure 5-27 Hierarchy of the ECI CCI exception classes

Further details on these exception classes can be found in the J2EE connector specification available for download at:

`http://java.sun.com/j2ee/download.html#connectorspec`

Table 5-1 shows how these exceptions map to the CICS ECI return codes.

Table 5-1 Exceptions thrown by ECI return codes

ECI return code	**ResourceException**
ECI_ERR_INVALID_DATA_LENGTH	CICSUserInputException
ECI_ERR_INVALID_EXTEND_MODE	ResourceAdapterInternalException
ECI_ERR_NO_CICS	CommException
ECI_ERR_CICS_DIED	EISSystemException
ECI_ERR_REQUEST_TIMEOUT	ResourceAdapterInternalException
ECI_ERR_NO_REPLY	ResourceAdapterInternalException
ECI_ERR_RESPONSE_TIMEOUT	EISSystemException
ECI_ERR_TRANSACTION_ABEND	CICSTxnAbendException
ECI_ERR_LUW_TOKEN	ResourceAdapterInternalException
ECI_ERR_SYSTEM_ERROR	ResourceAdapterInternalException
ECI_ERR_NULL_WIN_HANDLE	n/a
ECI_ERR_NULL_MESSAGE_ID	n/a
ECI_ERR_THREAD_CREATE_ERROR	ResourceAllocationException
ECI_ERR_INVALID_CALL_TYPE	ResourceAdapterInternalException
ECI_ERR_ALREADY_ACTIVE	LocalTransactionException
ECI_ERR_RESOURCE_SHORTAGE	ResourceAllocationException
ECI_ERR_NO_SESSIONS	ResourceAllocationException
ECI_ERR_NULL_SEM_HANDLE	n/a
ECI_ERR_INVALID_DATA_AREA	ResourceAdapterInternalException
ECI_ERR_INVALID_VERSION	ResourceAdapterInternalException
ECI_ERR_UNKNOWN_SERVER	ResourceAllocationException
ECI_ERR_UNKNOWN_SERVER	ResourceAdapterInternalException
ECI_ERR_CALL_FROM_CALLBACK	ResourceAdapterInternalException
ECI_ERR_MORE_SYSTEMS	ResourceAdapterInternalException
ECI_ERR_NO_SYSTEMS	n/a
ECI_ERR_SECURITY_ERROR	SecurityException

ECI return code	ResourceException
ECI_ERR_MAX_SYSTEMS	ResourceAllocationException
ECI_ERR_MAX_SESSIONS	ResourceAllocationException
ECI_ERR_ROLLEDBACK	LocalTransactionException

The `ResourceException` class provides the following methods that can be used to obtain further information about the error condition.

- **`getMessage()`**

  The `getMessage()` returns a String containing the CTG error code and a description of the error. An example of such a String is:

  ```
  CTG9630E IOException occurred in communication with CICS
  ```

  All the error messages produced by the J2EE resource adapters can be found in Appendix C, “J2EE Messages” of the *CICS Transaction Gateway, Gateway Programming,* SC34-5938.

- **`getErrorCode()`**

  This method returns a String that contains either an ECI return code, or a CICS abend code.

  – If the result is an abend code then a `CICSTxnAbendException` will have been generated. An abend code is a 4-character code returned by the CICS server program. These are architected codes provides by CICS to help determine the cause of the abend. An example being `AEIO, which` indicates that the desired CICS program was not found (PGMIDERR).

  – If the result is an ECI return code, it should be converted to an integer and then compared to the constants in the `ECIReturnCodes` interface. For further details on using this interface refer to 4.7.1, “ECI return codes” on page 64.

- **`getLinkedException()`**

  This method returns a wrapped exception that may have generated the error. If there is no wrapped `exception`, null is returned; for instance, when there is a network problem while communicating with the gateway daemon. The primary exception will be a `CommException`, but the linked exception will be a `java.io.IOException`.

5.6.1 Developing an exception handling routine

It is encouraged that you build an exception handling routine for your applications. Figure 5-28 and Figure 5-29 show a set of try/catch blocks that were developed for catching the most common errors in a ECI CCI application. The full version of this sample is available within the `ECIProgTest` class that is provided in the `itso.cics.eci.j2ee` package with this redbook.

In addition to the sub-classes of `ResourceException`, the `UnsupportedEncodingException` is also catched, since this can be generated by the `getBytes()` method that is used to store data in the `GenericRecord` class.

```
} catch (com.ibm.connector2.cics.CICSTxnAbendException re) {
        System.out.println("Error - CICS Abend: " + re.getErrorCode());
        if (re.getErrorCode().equals("AEIO")) {
            System.out.println("Msg: CICS Program not found");
        } else {
            System.out.println("Msg: Check CICS log");
        }
} catch (javax.resource.spi.SecurityException re) {
        System.out.println("Error - Security: " + re.getErrorCode());
        if (Integer.valueOf(re.getErrorCode()).intValue()
            == ECIResourceAdapterRc.ECI_ERR_SECURITY_ERROR) {
            System.out.println("Msg: Check credentials for userid ");
        } else {
            System.out.println("Msg: " + re.getMessage());
        }
} catch (javax.resource.spi.CommException re) {
        System.out.println("Error - Communication: " + re.getErrorCode());
        if (re.getLinkedException() != null) {
            if (re.getLinkedException() instanceof java.io.IOException) {
                System.out.println("Msg: IO error: check gateway daemon");
            } else {
                System.out.println("Linked exception = " + re.getLinkedException());
            }
        }
        if (re.getErrorCode() != null) {
            if (Integer.valueOf(re.getErrorCode()).intValue()
                == ECIResourceAdapterRc.ECI_ERR_NO_CICS) {
                System.out.println("Msg: CICS server is stopped");
            } else {
                System.out.println("Msg: " + re.getMessage());
            }
        }
```

Figure 5-28 ECI CCI application Exception handling, part 1

```
} catch ...
} catch (ResourceException re) {
        System.out.println("Unknown Error: " + re.getMessage());
        if (re.getLinkedException() != null) {
            System.out.println("Linked exception = " + re.getLinkedException());
        }
} catch (UnsupportedEncodingException ue) {
        System.out.println("Error - UnsupportedEncoding");
        System.out.println("CodePage: " + ue.getMessage());
    }
}
```

Figure 5-29 ECI CCI application Exception handling, part 2

6

CCI applications in a managed environment

This chapter discusses using WebSphere Application Server Advanced Edition V4 in a J2EE Connector Architecture managed environment. It is split into five sections:

- **WebSphere managed environment**
  This discusses the WebSphere V4 support implemented for the managed environment, including the restrictions of this release.
- **Configuring WebSphere Application Server**
  This provides a step-by-step guide to enabling and installing the CICS resource adapters, provided by the CTG, into a managed environment.
- **Creating the CCI application**
  This describes how to build an enterprise application in WebSphere Studio Application Developer that uses the CCI to interact with a CICS stock trading program called Trader.
- **Testing the enterprise bean**
  This shows how to test the Trader enterprise bean in WebSphere Studio Application Developer using the EJB test client.
- **Deploying the application to WebSphere**
  This documents how to deploy the Trader enterprise application to the WebSphere managed environment.

6.1 WebSphere managed environment

Use of the J2EE Connector Architecture managed environment implies a scenario where a Java application (a component) accesses a resource adapter through an application server. Management of connections, transactions, and security is provided by this application server. The component developer does not have to code this management manually.

WebSphere Application Server Advanced Edition V4 is the first version of this application server to provide a J2EE Connector Architecture managed environment. This support was initially added as an optional add-on known as the *Connector Architecture for WebSphere Application Server (technology preview)*. The technology preview is an additional download that can be plugged into the application server to provide a managed environment that adheres to the Proposed Final Draft #2 of the J2EE Connector Architecture specification.

Subsequent to writing this book, PTF V4.02 was released for Websphere Application Server. This incorporates the J2EE Connector Architecture technology preview within the base product.

Tip: The WebSphere Application Server InfoCenter contains extensive documentation on the application server, including the J2EE Connector Architecture technology preview. The InfoCenter is available online at:

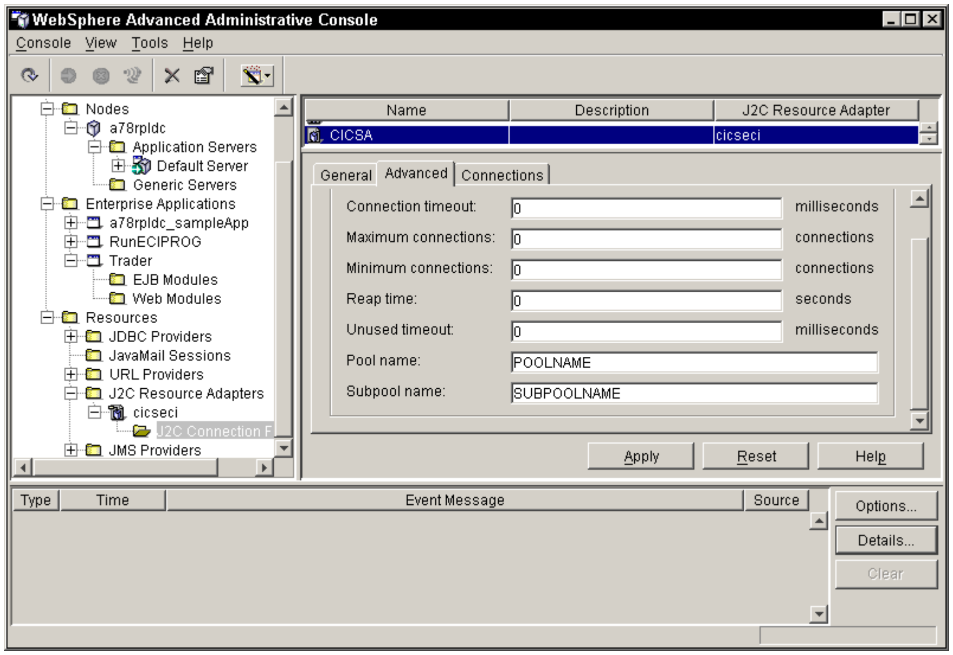
`http://www.ibm.com/software/webservers/appserv/infocenter.html`

The J2EE Connector Architecture managed environment support provided by WebSphere Application Server Advanced Edition V4 with the technology preview is summarized in this section.

Connection management

Connection pooling is implemented to allow the reuse of CICS Transaction Gateway connections. Connection pool settings are specified in the *Advanced* pane of the connection factory in the Administrative Console (Figure 6-1).

Note: The EJB container in WebSphere V4 supports managed connections. Therefore, enterprise beans can be used in a managed environment with this release. This support is not extended to the Web container, therefore, servlets cannot directly use the managed environment. Instead, they must make a call to an enterprise bean to utilize the managed environment.


WebSphere Advanced Administrative Console

Console View Tools Help

- Nodes
  - a78rpldc
    - Application Servers
      - Default Server
    - Generic Servers
- Enterprise Applications
  - a78rpldc_sampleApp
  - RunECIPROG
  - Trader
    - EJB Modules
    - Web Modules
- Resources
  - JDBC Providers
  - JavaMail Sessions
  - URL Providers
  - J2C Resource Adapters
    - cicseci
      - J2C Connection F
  - JMS Providers

Name	Description	J2C Resource Adapter
CICSA		cicseci

General | Advanced | Connections

Connection timeout:	0	milliseconds
Maximum connections:	0	connections
Minimum connections:	0	connections
Reap time:	0	seconds
Unused timeout:	0	milliseconds
Pool name:	POOLNAME	
Subpool name:	SUBPOOLNAME	

Apply Reset Help

Type	Time	Event Message	Source

Options... Details... Clear

Figure 6-1 Administrative console with connection pooling

For each request made to the CICS ECI resource adapter, WebSphere Application Server will look for an available connection to the CTG from its pool of connections. If a connection is available, it will use it for the request, thus saving the overhead of creating a new connection. If no connection is available, a new connection to the CICS Transaction Gateway will be created and added to the pool.

You can limit the number of connections that exist at any one time by specifying a value in the *Maximum Connections* field of the connection factory. After this number is reached, no new connections are created, and the exception `javax.resource.spi.ResourceAllocationException` is thrown. Alternatively, you can specify that new connection requests wait a period of time for a connection to become available. This period of time is specified in the *Connection Timeout* field of the connection factory.

Restriction: We found that the connection timeout feature did not work as expected in the technology preview. In this scenario, when the maximum connections value was exceeded, further attempted connections threw `ResourceAllocationException`, regardless of the connection timeout value specified. The connection timeout feature will be fixed in a subsequent release of WebSphere Application Server.

If a connection to the CICS Transaction Gateway remains idle for a given period of time, the connection is disconnected. This period of time is controlled by the `idletimeout` value in the `CTG.INI` file.

Transaction management

Support is provided by WebSphere Application Server for *NoTransaction*, *LocalTransaction*, and *XATransaction*. The CICS ECI resource adapter supports *LocalTransaction*. The CICS EPI resource adapter supports *NoTransaction*.

Local transaction optimization and *last resource optimization* are not currently supported in WebSphere V4. To use last resource optimization, the CICS ECI resource adapter must be deployed in a managed application server environment that also provides this support. For more information refer to 3.3.2, “Transaction management” on page 25.

The *get/use/close* programming model is recommended for using the connector inside WebSphere. This model dictates that a connection to a resource adapter is retrieved, used, and closed with a transaction. The connection object should not exceed transaction boundaries. The *get/use/cache* programming model, which caches connections across transaction boundaries, is not currently supported in WebSphere V4.

Important: The *get/use/cache* programming model, local transaction optimization, last resource optimization, and container managed authentication are not currently supported in WebSphere V4, but future versions will provide these functions.

Security management

Only component managed sign on (Option C in the J2EE Connector Architecture specification) is supported. This requires the component to pass a user ID and password credentials through the `ConnectionSpec` to CICS. If the credentials in the `ConnectionSpec` are not set, then the credentials in the `ManagedConnectionFactory` are used to authenticate to CICS.

Container managed sign on, where the component relies on the application server to provide security credentials instead of manually specifying them, is not supported. The `<res-auth>` element of the deployment descriptor, which determines whether component or container managed security should be used, is ignored. In order to be consistent with future releases that may support container managed signon, the `<res-auth>` element should be set to `Application`.

6.2 Configuring WebSphere Application Server

This section shows how to configure WebSphere Application Server Advanced Edition V4 to support the CICS resource adapters. It contains the following steps:

1. Download and install the J2EE connector support.
2. Install the CICS resource adapters.
3. Configure a connection factory.
4. Test the configuration.

Download and install the J2EE connector support

The Connector Architecture for WebSphere Application Server (technology preview) provides an early implementation of the J2EE Connector Architecture Specification, Proposed Final Draft #2. It was the first J2EE Connector Architecture support available for WebSphere Application Server V4.

To install it, do the following:

1. Download the Connector Architecture technology preview for Windows from:

   `http://www6.software.ibm.com/dl/connarch/connarch-p`

   Note: Subsequent to the writing of this redbook, PTF V4.0.2 of WebSphere Application Server Advanced Edition was released. This incorporates the J2EE Connector Architecture technology preview within the actual product, and provides the same functionality documented here.

2. The file `installJ2C_AE.zip` will be downloaded in the technology preview. Extract the contents of this zip file to a temporary directory. Ensure this temporary directory is not on a remote drive to the WebSphere Application Server installation. The use of a network drive may cause installation errors.

   Note: The acronym *J2C* is used to represent IBM's implementation of the J2EE Connector Architecture, in both the technology preview and WebSphere Application Server help files.

3. Before running the installation script, stop all WebSphere Application Server processes, including:
   – IBM HTTP Administration
   – IBM HTTP Server
   – IBM WS AdminServer V4.0

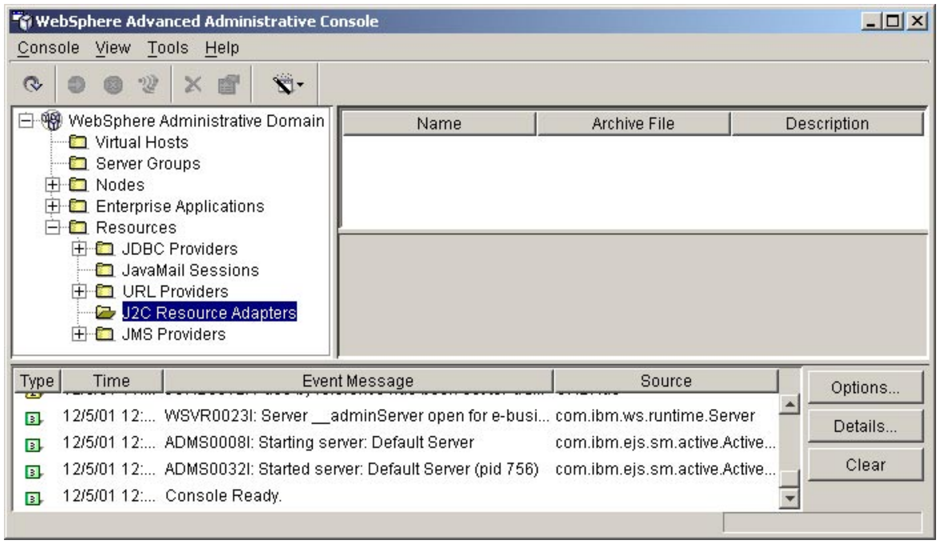
4. Run the installation script. From the Windows command prompt, move to the temporary directory where you extracted the zip file, and run `installJ2C.bat`. If you encounter errors, consult `<WAS_ROOT>\logs\j2Cinstall.log` for more information.
5. After a successful installation, the `<WAS_ROOT>\lib` directory will contain:
   – `j2c.jar`
   – `jca.jar`
   – `eablib.jar`
   – `recjava.jar`
   – `ccf.jar`
6. Restart the WebSphere Application Server processes.

Install the CICS resource adapters

The CTG provides two CICS resource adapters: the *ECI resource adapter* and the *EPI resource adapter*. This section describes how to install one or both of these resource adapters into WebSphere Application Server.

This section assumes that the CICS Transaction Gateway V4.0.1 is already installed. Please follow these steps to install the CICS resource adapters:

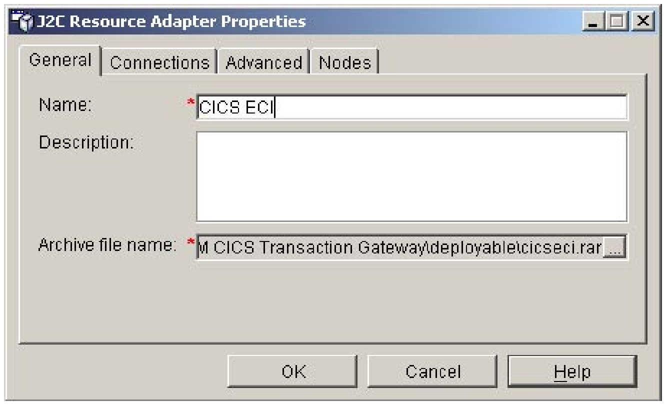
1. Open the WebSphere Advanced Administrative Console. Expand **WebSphere Administrative Domain** -> **Resources**. Notice the *J2C Resource Adapters* folder (Figure 6-2). This is where the resource adapters are defined to the application server.



```
12/5/01 12:...  WSVR0023I: Server __adminServer open for e-busi...  com.ibm.ws.runtime.Server
12/5/01 12:...  ADMS0008I: Starting server: Default Server           com.ibm.ejs.sm.active.Active...
12/5/01 12:...  ADMS0032I: Started server: Default Server (pid 756)  com.ibm.ejs.sm.active.Active...
12/5/01 12:...  Console Ready.
```

Figure 6-2 Administrative console J2C resource adapters

2. Resource adapters are installed into application servers using a *Resource Adapter Module* (represented by a file with an extension of *rar*). This RAR file contains a collection of JAR files relating to the resource adapter, and a deployment descriptor (`ra.xml`) that describes the deployment properties for the resource adapter. The CTG provides a RAR file for each CICS resource adapter. To install the CICS ECI resource adapter, follow these steps:
   a. Right click on **J2C Resource Adapters** and select **New**. This opens the *J2C Resource Adapter Properties* window.
   b. In the *General* pane, enter a name for the resource adapter in the *Name* field. Use CICS ECI.
   c. In the *General* pane, expand the Archive file named *textbox*.
   d. This opens the *Install Driver* window. Select the node you wish to install the resource adapter to, then click the **Browse** button to select the RAR file to use.
   e. To install the ECI resource adapter, move to the `<CTG_HOME>\deployable` directory and select **cicseci.rar**. Select **Open** to select this file, then select **Install**.
   f. The completed *J2C Resource Adapter Properties* window is shown in Figure 6-3. Select the **Connections** and **Advanced** panes to view information about the resource adapter.
   g. Select **OK** to finish. If the resource adapter was installed correctly, a dialog box will display saying *Command "J2CResourceAdapter.create" completed successfully*.


J2C Resource Adapter Properties

General | Connections | Advanced | Nodes

Name: * CICS ECI

Description:

Archive file name: * M CICS Transaction Gateway\deployable\cicseci.rar

Figure 6-3 J2C resource adapter properties

3. If you wish to install the EPI resource adapter, repeat the above process, selecting **cicsepi.rar** as the archive file name.

Configure a connection factory

Every resource adapter installed in WebSphere Application Server should have one or more connection factories associated with it. A connection factory contains deployment specific connection information. In the case of the CICS resource adapters, this information includes:

- The connection URL of the CICS Transaction Gateway
- The CICS server to communicate with
- The CICS program to call, or the CICS transaction to start
- A user ID and password to flow to CICS

At least one connection factory is required to use the resource adapter in a managed environment. Setting up multiple connection factories allows you to configure a collection of possible connection options. When you deploy an enterprise bean into WebSphere Application Server, you must select which connection factory you wish to use.

This section explains how to create and configure a connection factory for the CICS ECI resource adapter. This connection factory will be used in the next section to test the J2EE Connector Architecture support in the application server.

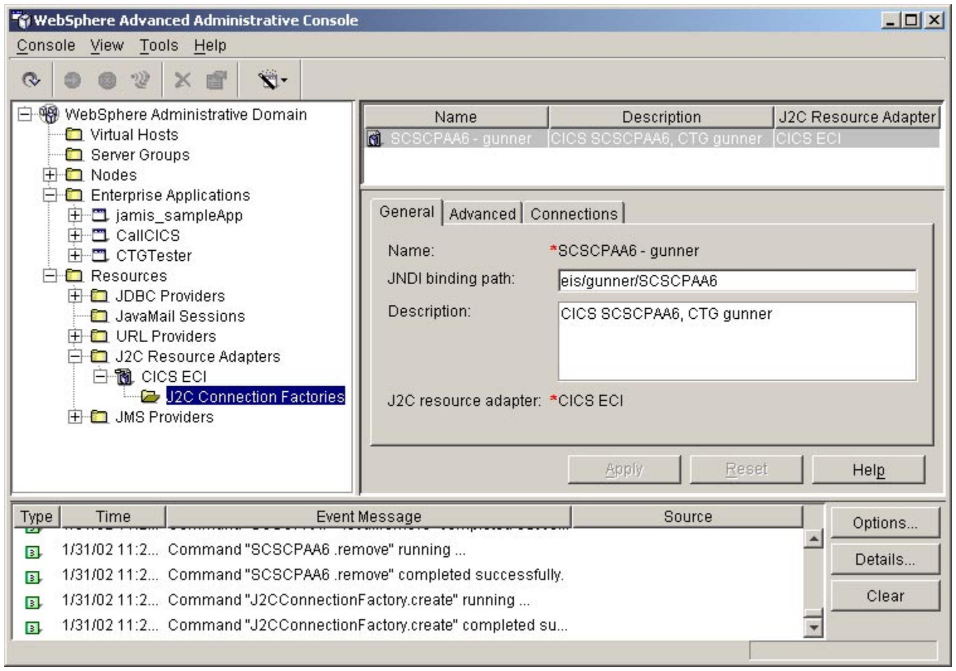
- In order to open the WebSphere Administrative Console, take the following steps:
  a. Expand **WebSphere Administrative Domain -> Resources -> J2C Resource Adapters**. This will display the resource adapter you have installed.
  b. Expand the CICS ECI resource adapter, and select **J2C Connection Factories**. A list of all connection factories created for this resource adapter (if any) will be displayed in the right pane.
- To create a new connection factory, take the following steps:
  a. Right click **J2C Connection Factories** and select **New**. This will open the *J2C Connection Factory Properties* window.
  b. In the *General* pane enter the following:
     - In the *Name* field, enter a name for the connection factory. Use *SCSCPAA6 - gunner*.
     - Optionally, enter a path in the *JNDI binding path* field. We used *eis/gunner/SCSCPAA6***.** If you leave this blank, a JNDI binding path will be created for you in the format of `eis/<connection factory name>`.
     - Notice the *J2C Resource Adapter* field is set to *CICS ECI*.
  – The *Advanced* pane allows you to set connection pooling information. Do not change any fields in this pane.

– The *Connections* pane allows you to set connection specific information. Set the following:

   - *ConnectionURL* sets the connection URL of the CICS Transaction Gateway to use. Use `tcp://gunner`. If you wish to connect to a local CTG specify `local:` here.
   - *ServerName* sets the name of the CICS server to connect to. Use *SCSCPAA6*.
   - If your CICS region requires security credentials, specify values for the *UserName* and *Password* parameters.

– Select **OK** when finished. If the connection factory was installed correctly, a dialog box will display saying `Command “J2CConnectionFactory.create” completed successfully`. This will publish a reference to the connection factory in the JNDI namespace.

► The newly created connection factory will be displayed in the right pane (Figure 6-4).



Name	Description	J2C Resource Adapter
SCSCPAA6 - gunner	CICS SCSCPAA6, CTG gunner	CICS ECI

Name: *SCSCPAA6 - gunner
JNDI binding path: eis/gunner/SCSCPAA6
Description: CICS SCSCPAA6, CTG gunner
J2C resource adapter: *CICS ECI

```
Type  Time          Event Message                                                    Source
      1/31/02 11:2...  Command "SCSCPAA6 .remove" running ...
      1/31/02 11:2...  Command "SCSCPAA6 .remove" completed successfully.
      1/31/02 11:2...  Command "J2CConnectionFactory.create" running ...
      1/31/02 11:2...  Command "J2CConnectionFactory.create" completed su...
```

Figure 6-4 Administrative console with the Connection Factory

Test the configuration

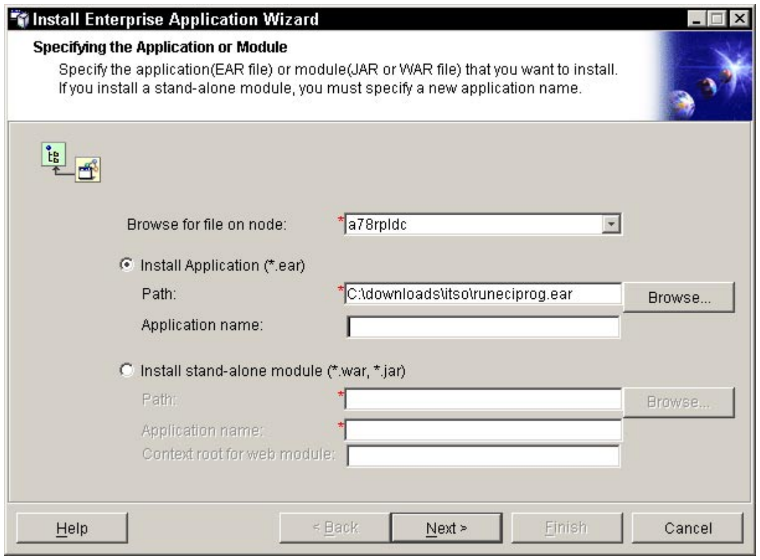
Now that the J2EE connector support has been installed, and a connection factory has been created, the environment is ready to be tested. We show you how to create an enterprise application that tests the CICS ECI resource adapter in a managed environment. This provides a quick and easy way to check that the environment has been configured correctly. The enterprise application calls the CICS program ECIPROG used in Chapter 5, “CCI applications: ECI based” on page 71. The enterprise application consists of:

- The HTML page `Start.html` contains a form that starts the servlet `RunECIPROGServlet`. Two parameters are sent to the servlet: the name of the CICS program to call (in most cases this will simply be ECIPROG), and the encoding to use.
- The servlet passes these parameters on to the RunECIPROG session bean.
- The session bean makes the call to CICS using the CICS ECI resource adapter, and returns the response to the servlet. The servlet generates an HTML page containing this response.

This enterprise application is stored in `runeciprog.ear`. To obtain this file, see Appendix D, “Additional material” on page 261.

Follow the instructions below to test the J2EE connector configuration. These instructions assume you have already created a connection factory called *SCSCPAA6 - gunner* with the JNDI name *eis/gunner/SCSCPAA6*, as described in, “Configure a connection factory” on page 118.

- From the WebSphere Advanced Administrative Console, select **Console -> Wizards -> Install Enterprise Application**.
- This opens the Install Enterprise Application Wizard window (Figure 6-5). Select **Install application (*.ear)** then select **Browse**. Move to the directory where you downloaded `runeciprog.ear`, select this file, then select **Open**.


Install Enterprise Application Wizard

Specifying the Application or Module

Specify the application(EAR file) or module(JAR or WAR file) that you want to install.
If you install a stand-alone module, you must specify a new application name.

Browse for file on node: * a78rpldc

(●) Install Application (*.ear)

Path: * C:\downloads\itso\runeciprog.ear

Application name:

( ) Install stand-alone module (*.war, *.jar)

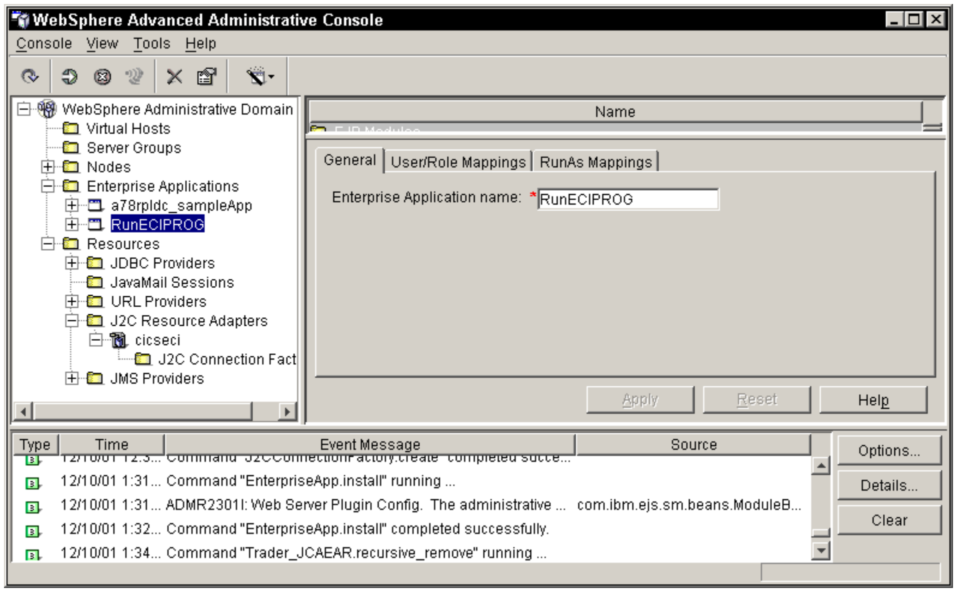
Path: *

Application name: *

Context root for web module:

Figure 6-5 Installing an EAR file into WebSphere Application Server

- You should use the default values specified in the EAR file to deploy this enterprise application. Click **Next** continually until the *Completing the Application Installation* Wizard screen appears. At this screen click **Finish**. When asked if you would like to regenerate code select **No**. If the enterprise application was installed successfully, a dialog box will display saying `Command “EnterpriseApp.install” completed successfully.`
- The enterprise application RunECIPROG will be created. To view it in the Administrative Console, expand **WebSphere Administrative Domain -> Enterprise Applications**. See Figure 6-6.



Enterprise Application name: * RunECIPROG

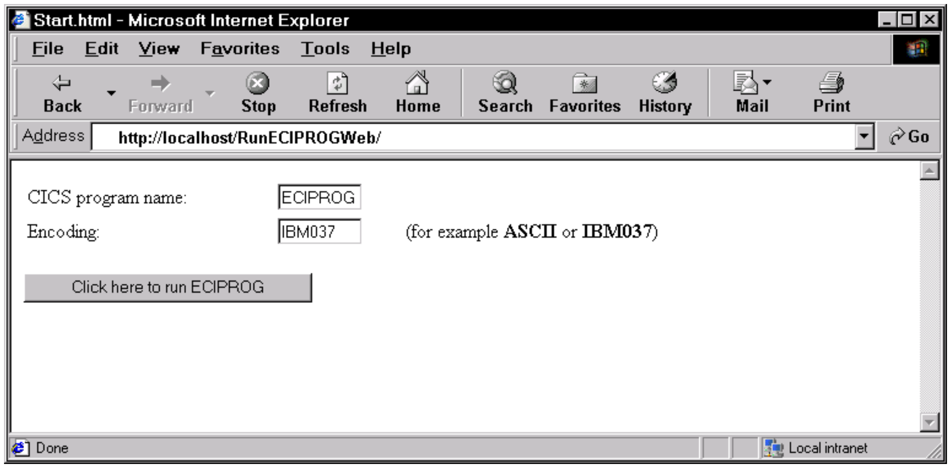
Type	Time	Event Message	Source
	12/10/01 12:3...	Command "J2CConnectionFactory.create" completed succe...	
	12/10/01 1:31...	Command "EnterpriseApp.install" running ...	
	12/10/01 1:31...	ADMR2301I: Web Server Plugin Config. The administrative ...	com.ibm.ejs.sm.beans.ModuleB...
	12/10/01 1:32...	Command "EnterpriseApp.install" completed successfully.	
	12/10/01 1:34...	Command "Trader_JCAEAR.recursive_remove" running ...	

Figure 6-6 Installed RunECIPROG application

- Now the enterprise application has been installed, the Webserver Plugin must be notified. To do this, expand **WebSphere Administrative Domain -> Nodes**. Right click on the node under which you installed the enterprise application and select **Regen Webserver Plugin**.
- Before running this enterprise application, restart the default application server. To do this, expand **WebSphere Administrative Domain -> Nodes -> <*your node name*> -> Application Servers**. Right click on **Default Server**. Stop the server if it is running, then select **Start** to restart the default server.
- To run the enterprise application, open a Web browser and enter the URL:

  `http://localhost/RunECIPROGWeb/`

  You may need to specify a port in the URL, depending on your WebSphere Application Server configuration. This should display the page shown in Figure 6-7.


Start.html - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail Print
Address http://localhost/RunECIPROGWeb/
Go
Click here to run ECIPROG
Done
Local intranet

CICS program name: ECIPROG

Encoding: IBM037 (for example **ASCII** or **IBM037**)

Figure 6-7 RunECIPROG enterprise application

- Notice that there are two parameters that can be customized:
  - **CICS program name**
    This defaults to ECIPROG. If ECIPROG has been defined to CICS with a program resource name other than ECIPROG, specify that name here.
  - **Encoding**
    CICS returns the COMMAREA as a byte array. The servlet will convert this byte array into a string, using the encoding value specified here. Set this encoding value to the code page your CICS server is using (typically IBM037). However, if CICS is performing data conversion for ECIPROG using a DFHCNV template, specify the encoding value to match the output code page of DFHCNV (typically an ASCII code page such as 8859_1). For further details, refer to Appendix B, "Data conversion" on page 227.
- Click the **Click here to run ECIPROG** button. This will use the CICS ECI resource adapter and CICS server you specified in the connection factory to call the ECIPROG CICS program. If the connection was successful, the following response will be returned in the HTML form (Example 6-1).

Example 6-1 Output from RunECIPROG

```
Call ECIPROG Servlet
CICS Program name: ECIPROG
Encoding used: IBM037
Values returned from CICS
CICS server: SCSCPAA6
Date: 20/12/01
```

6.3 Creating the CCI application

This section describes how to develop a J2EE application that uses the CICS ECI resource adapter in a managed environment. We show you how to create a Web application that uses the business logic component of the backend CICS program TRADERBL (see Appendix C.5, “TRADER” on page 253). The completed application consists of a number of JSPs, data beans, a servlet, and an enterprise bean. Most of the components are slightly modified versions of those created for the book *Enterprise JavaBeans for z/OS and OS/390 CICS Transaction Server V2.1,* SG24-6284. However, the enterprise bean was freshly created for this example. The session bean contains all of the code for connecting to the CICS server, therefore, this section concentrates on describing how to develop this particular component.

It is suggested that if you intend to follow the instructions in this section to build your own Trader enterprise application, you should first download all of the ancillary classes used in this chapter. See Appendix D, “Additional material” on page 261 for instructions on how to do this.

Figure 6-8 illustrates the complete architecture of the Web application that was created.

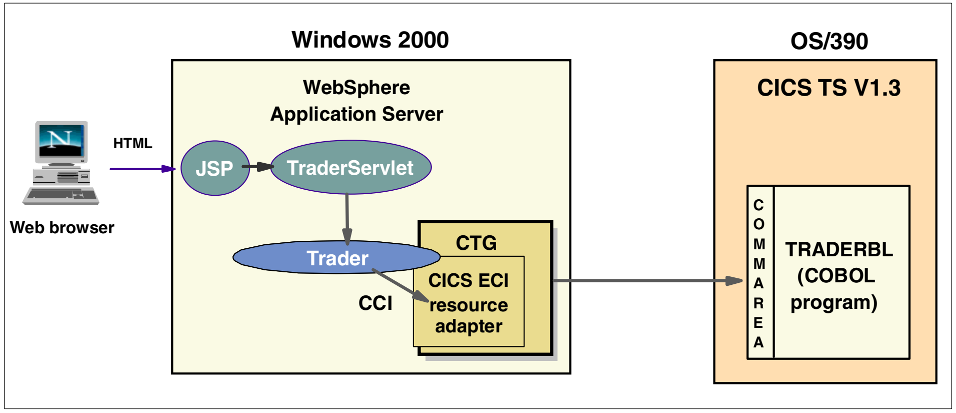


Figure 6-8 Application architecture of completed Trader Web application

Application overview

To understand how to create the session bean, and how it connects to CICS, it is not necessary to know the internal workings of the remaining components of the final Web application. However, an appreciation of the overall flow of the application is beneficial. The following list explains the sequence of events that occur when the end user interacts with the application through a Web browser:

1. The end user presses a button on a Web page that submits a form to the servlet.
2. The servlet receives the request for action and calls an appropriate method on the remote interface of the Trader session bean.
3. The session bean connects uses the J2EE CCI to call the CICS program TRADERBL, using the facilities of the CTG and the CICS resource adapter.
4. The session bean returns any output data from TRADERBL back to the servlet in the form of a data bean object.
5. The servlet forwards the request to a JSP, which displays the contents of the data bean to the end user.

We developed and tested our application using WebSphere Studio Application Developer. We chose this environment instead of VisualAge for Java V4, because it supports editing of EJB 1.1 deployment descriptors, and can generate deployment code suitable for the WebSphere Application Server V4 container. It can export deployed application files directly, ready for installation into WebSphere Application Server V4. These application files include J2EE enterprise archives (EAR files), Web application archives (WAR files), and EJB 1.1 deployed JAR files. Application Developer also provides a suitable local test environment for testing your application components because it uses WebSphere Application Server Advanced Single Server V4. We also used the EJB Test Client from Application Developer to test the session bean.

As an alternative to using Application Developer, you can use VisualAge for Java in combination with the Application Assembly Tool (AAT) to produce deployable enterprise beans for WebSphere Application Server Advanced Edition.

6.3.1 Configuring WebSphere Studio Application Developer

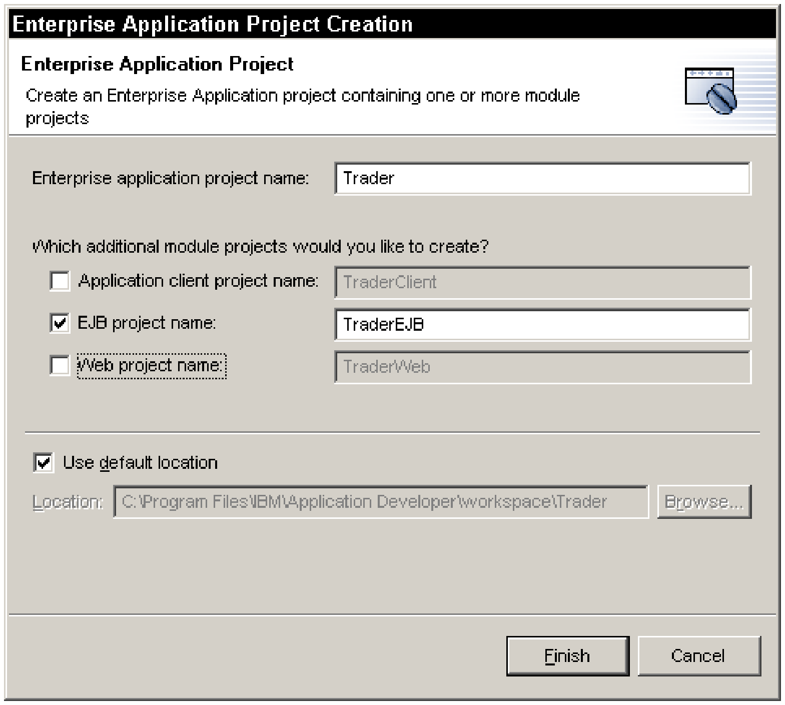
This section begins by showing you how to configure Application Developer to develop a Web application that uses the J2EE Connector Architecture support. We started with a fresh installation of Application Developer, and created projects that contain the J2EE application. You must perform the following steps:

1. Create projects for the application.
2. Modify the Java build path for our EJB project.
3. Import the Record class and data bean classes.
4. Create an enterprise bean.

Creating the application projects

This session creates an enterprise application project for a J2EE application that contains a Web module project to hold the session bean. The following steps show how to create these two projects in your workspace:

1. Switch to the J2EE perspective in Application Developer by choosing **Perspective -> Open -> J2EE** from the menus.
2. Choose **File -> New -> Enterprise Application Project**. This opens the Enterprise Application Project Creation dialog box. At this point you can specify which sub-projects the enterprise application project will contain. Fill in the dialog as shown in Figure 6-9. Click **Finish** to create the projects in your workspace.



Enterprise Application Project Creation

Enterprise Application Project

Create an Enterprise Application project containing one or more module projects

Enterprise application project name: Trader

Which additional module projects would you like to create?

- [ ] Application client project name: TraderClient
- [x] EJB project name: TraderEJB
- [ ] Web project name: TraderWeb

- [x] Use default location

Location: C:\Program Files\IBM\Application Developer\workspace\Trader

Figure 6-9 Creating an enterprise application project

Note: An enterprise application project can consist of both Web module projects (containing servlets and JSPs) and EJB module projects (containing enterprise beans). The enterprise application project that was initially created contained only an EJB project for the session bean. Once the session bean is complete, add a Web module project to the enterprise application by importing the pre-existing Web components from a WAR file.

Modifying the Java build path

Now that the projects have been created, it is necessary to add some JAR files to the Java build path, which contains the classes that your project will be using. The following steps shows how to add the JARs to the project's Java build path:

1. Open the *Navigator* view from the J2EE Perspective page by choosing **Perspective -> Show View -> Navigator** from the menus.
2. Right click on the **TraderEJB** project in the *Navigator* view and select **Properties**.
3. Choose **Java Build Path** from the options list on the left of the dialog box, and then select the **Libraries** tab from the right hand panel.
4. Click the **Add External JARs** button to browse to the location of a JAR file in order to add the Java build path, and click **Open**. Repeat this step for each of the JAR files that you require.
5. The session bean makes use of the CICS ECI resource adapter classes and the EAB Record object classes. Therefore, you must add the JAR files that contain these referenced classes to the TraderEJB project's Java build path.

   Add the following JARs from a CTG V4.0.1 installation:

   – `classes\connector.jar`
   – `classes\cicsj2ee.jar`

   and the following two JARs from a WebSphere Advanced V4 installation:

   – `lib\recjava.jar`
   – `lib\eablib.jar`
6. When you have finished adding all of the JAR files that your EJB will reference to the Java build path, click **OK**.

Tip: If you do not want to learn how we developed our enterprise bean, you can import the sample `TraderEJB.jar` shipped with this book, and skip straight to 6.4, "Testing the enterprise bean" on page 139.

Importing a Record class and data bean classes

To be able to use the CCI we needed to use a Record class to represent the COMMAREA of the TRADERBL program. We also used data bean classes to store output information from the backend CICS TRADERBL program and to enable the JSPs to display this data. The Record class and data beans classes must be imported into the EJB project so that they can be referenced by the session bean.

Importing the Record class

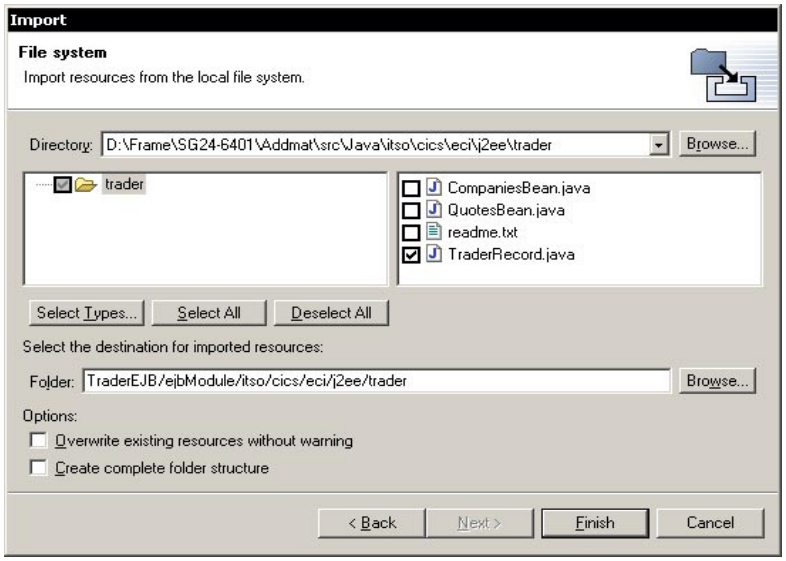
Application Developer does not currently provide a tool for creating these Record objects so you will need to use the EAB toolkit in VisualAge for Java. To use the EAB follow the same procedure as detailed in 5.2.1, “Creating a Record out of a COMMAREA” on page 79 to import the COBOL source. This will create a Record class called `itso.cics.eci.j2ee.trader.TraderRecord` which should then be exported the to the local file system. Note, if you wish you can skip the creation of the Record class, and use the `TraderRecord` sample class supplied with the redbook.

The following steps describe how to import the `Record` into Application Developer (Figure 6-10):

1. From the *J2EE* perspective *Navigator* view, click on the **TraderEJB** project. Choose **File -> Import** from the menu system.
2. Select **File system** as the import source and click **Next**.
3. Click **Browse** to locate the directory containing `TraderRecord.java`. Once you have located the directory, click **OK**.
4. Click on the directory name in the left hand side part of the panel and then click the **`TraderRecord.java`** box on the right hand side. Set the folder name to:

   ```
   TraderEJB/ejbModule/itso/cics/eci/j2ee/trader
   ```

5. Set all of the other remaining options shown in Figure 6-10, then click **Finish** to complete the import.


Import

File system

Import resources from the local file system.

Directory: D:\Frame\SG24-6401\Addmat\src\Java\itso\cics\eci\j2ee\trader

[✓] trader

[ ] CompaniesBean.java
[ ] QuotesBean.java
[ ] readme.txt
[✓] TraderRecord.java

Select the destination for imported resources:

Folder: TraderEJB/ejbModule/itso/cics/eci/j2ee/trader

Options:
[ ] Overwrite existing resources without warning
[ ] Create complete folder structure

Figure 6-10 Importing a Record class into Application Developer

Importing data bean classes

Once you import the Record class, you must import the following classes into the same directory as the `TraderRecord`. Follow the same procedure again to import these extra classes from the `itso.cics.eci.j2ee.trader` package:

- CompaniesBean.java
- QuotesBean.java

6.3.2 Creating an enterprise bean

The Trader application requires an enterprise bean component that will perform the task of calling the CICS program TRADERBL. We created a session bean that implements the following business methods, and uses the CCI to make connections to CICS:

`logon()`	To logon to the Trader application
`logoff()`	To logoff from the application
`getCompanies()`	To query the companies to trade with
`getQuotes()`	To retrieve quotes for a specific company
`buy()`	To buy shares of a specific company
`sell()`	To sell shares of a specific company

These business methods are responsible for creating input data and handling output data for the calls to CICS. There is a delegation for the sending and receiving of data to additional private methods in the enterprise bean. Much of the code that was created is the same as described in 5.1.1, "Writing a simple CCI application" on page 73, but in this example, the resource adapter is used in a managed environment. This section concentrates on the additional code that is required by the managed environment, and does not discuss in detail the areas that are common to the non-managed scenario.

You will need to follow these steps to create this enterprise bean:

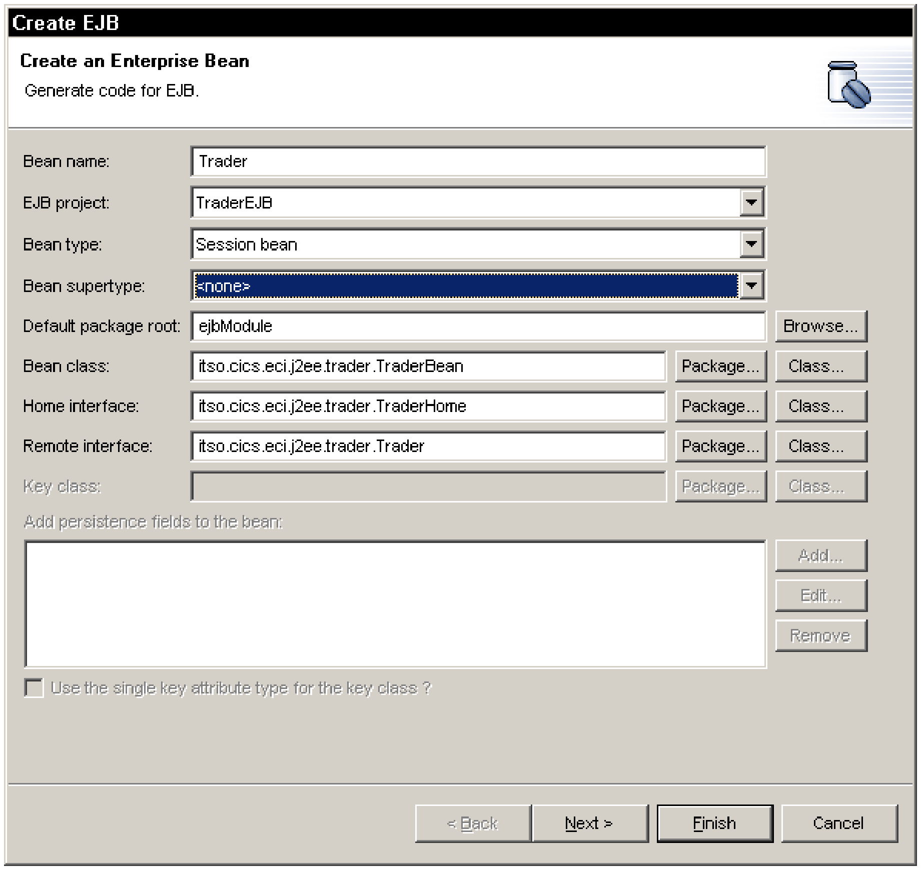
1. Add a new enterprise bean to the `TraderEJB` project.
2. Add private methods that use the CCI to connect to CICS.
3. Add the business methods.

Adding a new enterprise bean

Follow the steps below to create the basic framework for the bean in Application Developer:

1. From the J2EE perspective *Navigator* view, right click on the **TraderEJB** project and choose **New -> Enterprise Bean**. This opens the *Create EJB* dialog. Enter the bean name as Trader, and then insert the full package name of `itso.cics.eci.j2ee.trader` in front of `TraderBean` in the bean class entry

box. This will automatically complete the package names for the home and remote interfaces. The completed dialog box is shown in Figure 6-11. Click **Finish** to create the enterprise bean.


Create EJB

Create an Enterprise Bean

Generate code for EJB.

Bean name:	Trader
EJB project:	TraderEJB
Bean type:	Session bean
Bean supertype:	<none>
Default package root:	ejbModule
Bean class:	itso.cics.eci.j2ee.trader.TraderBean
Home interface:	itso.cics.eci.j2ee.trader.TraderHome
Remote interface:	itso.cics.eci.j2ee.trader.Trader
Key class:	

Add persistence fields to the bean:

Use the single key attribute type for the key class ?

Figure 6-11 Creating the Trader enterprise bean

2. From the *Navigator* view, expand the *ejbModule* folder in the *TraderEJB* project, and then expand the package tree for your session bean. You should see the three Java files that comprise the bean with its home and remote interfaces; in our case: `TraderBean.java`, `TraderHome.java` and `Trader.java` respectively.

Adding private methods that connect to CICS

Now that the framework for an enterprise bean is in place, you can start adding private methods that will make the connections to CICS using the CCI. These methods are responsible for flowing the input and receiving the output data. The business methods that we will add subsequently are responsible for creating the correct input data for TRADERBL, and for handling the output data.

This section describes how to build the enterprise bean in stages. To edit the enterprise bean, double click on the **TraderBean.java** file to open it in the Java editor. Here you can add the methods shown in this section, as they are described. When you save your file using the **File -> Save** menu option, Application Developer will try to compile your code, and any errors will be highlighted in the left hand margin of the Java editor.

Obtaining the ConnectionFactory object using JNDI

In a managed environment you *must* use JNDI to look up the managed `ConnectionFactory` object that is provided by the application server, rather than instantiate your own non-managed instance from a hard-coded set of parameters.

A private method in the EJB is implemented to do the look up of the `ConnectionFactory` in JNDI, and is called in `getConnectionFactory()`. Figure 6-12 shows the code used in the method, and also illustrates some changes to the bean class made to support this method.

```
package itso.cics.eci.j2ee.trader;
1
import javax.naming.*;
import javax.resource.cci.*;
import com.ibm.connector2.cics.*;

public class TraderBean implements javax.ejb.SessionBean {

2 // We have omitted the other methods of the EJB from this figure.

3  private ConnectionFactory cf = null;

   private void getConnectionFactory() throws Exception {
      try {
4        Context ic = new InitialContext();
5        cf = (ConnectionFactory) ic.lookup("java:comp/env/eis/ECICICS");
      } catch (Throwable t) {
         t.printStackTrace();
         throw new Exception("Lookup for ConnectionFactory failed.");
      }
   }
}
```

Figure 6-12   Obtaining the ConnectionFactory using JNDI

- 1 Add import statements for packages that were used in the session bean.
- 2 Other methods that exist in the bean class are not shown, to keep this figure concise.
- 3 The `ConnectionFactory` object should be kept in a private field belonging to the session bean.
- 4 Obtain an initial context to the JNDI namespace.
- 5 Create the `ConnectionFactory` instance by looking up a resource reference.

  A session bean uses a resource reference to represent a connection factory object. The JNDI string must match with the resource reference name that is declared in the EJB JAR deployment descriptor. For further information about declaring resource references, and an example of how to create one for your EJB, see 6.3.3, “Editing the EJB deployment descriptor” on page 137.

You can use any value for the lookup string, as long as it begins with java:/comp/env/. It is recommend that you also add eis to the name to indicate that the resource reference is a connection factory object for an EIS connector. The lookup() method return type is defined as Object, so a cast is used to change this to the specific class of ConnectionFactory. The ConnectionFactory is used by other methods in the enterprise bean, so it is necessary to store its value in the private field declared in 3.

Caching the connection and interaction with get/use/close

Now that you have created a method for obtaining the ConnectionFactory, it is necessary to add methods for obtaining the ECIInteractionSpec, Connection and Interaction objects. The code to create these objects is the same as shown in 5.1.1, "Writing a simple CCI application" on page 73.

However, in a managed environment it is necessary to be careful about caching these objects. The ECIInteractionSpec in the ejbCreate() method of our EJB was created and this was then cached for the lifetime of the bean in a private field. However, do not use caching for the Connection and Interaction objects, because WebSphere Application Server Advanced V4 only supports the use of a get/use/close model when using these objects. This means that each usage (getConnection(), Interaction usage, and close()) of a connection should be completed within a transaction. You should not cache the objects across transaction boundaries.

To support this programming model, add two methods, getConnection() and dropConnection(), which should be used before and after any call to CICS made using the execute() method of the Interaction object. The extra code added to the bean class is shown in Figure 6-13, which also shows the required import statements and fields. For further explanation about the code, see 5.1.1, "Writing a simple CCI application" on page 73.

```
private Connection eciConn = null;
private Interaction eciInt = null;

private void getConnection() throws Exception {
   eciConn = cf.getConnection();
   eciInt = eciConn.createInteraction();
}
private void dropConnection() throws Exception {
   eciInt.close();
   eciConn.close();
   eciInt = null;
   eciConn = null;
}
```

Figure 6-13 Methods for getting and dropping the connection

ejbCreate() method

Add the code to the `ejbCreate()` method. This creates and initializes the objects that will be cached for the lifetime of the EJB. Figure 6-14 shows how to create the `ECIInteractionSpec`, `ConnectionFactory` and `TraderRecord` objects. Set the parameters for the `ECIInteractionSpec` in the same way as for an application in a non-managed environment, but with values specific to the TRADER application. Set the function name to TRADERBL, which is the name of the CICS program you wish to call. Create a `TraderRecord` object that can be used later with the `Interaction` object's `execute()` method.

```
private ECIInteractionSpec eSpec = null;
private TraderRecord traderRecord = null;
public void ejbCreate() throws javax.ejb.CreateException {
   eSpec = new ECIInteractionSpec();
   eSpec.setCommareaLength(372);
   eSpec.setReplyLength(372);
   eSpec.setFunctionName("TRADERBL");
   traderRecord = new TraderRecord();
   try {
      eSpec.setInteractionVerb(ECIInteractionSpec.SYNC_SEND_RECEIVE);
      getConnectionFactory();
   } catch (Throwable t) {
      t.printStackTrace();
      throw new javax.ejb.CreateException("EJB creation failed.");
   }
}
```

Figure 6-14   ejbCreate() method

Calling the TRADERBL CICS program

Add the `callTraderBackend()` method as shown in Figure 6-15 to make the call to CICS. This method uses the get/use/close model and calls CICS in the same way as a non-managed environment.

```
private void callTraderBackend() throws Exception {
   try {
      getConnection();
      eciInt.execute(eSpec, traderRecord, traderRecord);
      dropConnection();
   } catch (Throwable t) {
      t.printStackTrace();
      throw new Exception("Error calling CICS: "+t.getMessage());
}}
```

Figure 6-15   callTraderBackend() method

Add business methods

Now that you have completed adding the private methods required for making the connections to CICS, you need to add the methods that will perform the business logic. Figure 6-16 and Figure 6-17 show the remaining code that must be added.

```
private static final int NUM_OF_COMPANIES=4;
private String ivUserID;
private String ivPassword;

public CompaniesBean getCompanies() throws Exception {
   CompaniesBean companies = new CompaniesBean();
   traderRecord.setRequest__Type("Get_Company");
   try {
      callTraderBackend();
   } catch (Throwable t) {
      t.printStackTrace();
      throw new Exception("Error getting companies: " + t.getMessage());
   }
   for (int i = 0; i < NUM_OF_COMPANIES; i++) {
      companies.addCompany(traderRecord.getCompany__Name__Tab(i));
   }
   return companies;
}
  public QuotesBean getQuotes(String company) throws Exception {
   QuotesBean quotes = new QuotesBean();
   traderRecord.setRequest__Type("Share_Value");
   traderRecord.setCompany__Name(company);
   traderRecord.setUserid(ivUserID);
   try {
      callTraderBackend();
   } catch (Throwable t) {
      t.printStackTrace();
      throw new Exception("Error getting quotes: " + t.getMessage());
   }
 quotes.setUnitSharePrice(traderRecord.getUnit__Share__Price());
   quotes.setUnitValue1Days(traderRecord.getUnit__Value__1__Days());
   quotes.setUnitValue2Days(traderRecord.getUnit__Value__2__Days());
   quotes.setUnitValue3Days(traderRecord.getUnit__Value__3__Days());
   quotes.setUnitValue4Days(traderRecord.getUnit__Value__4__Days());
   quotes.setUnitValue5Days(traderRecord.getUnit__Value__5__Days());
   quotes.setUnitValue6Days(traderRecord.getUnit__Value__6__Days());
   quotes.setUnitValue7Days(traderRecord.getUnit__Value__7__Days());
   quotes.setCommissionCostSell(traderRecord.getCommission__Cost__Sell());
   quotes.setCommissionCostBuy(traderRecord.getCommission__Cost__Buy());
```

Figure 6-16 Trader business methods, part 1

```
   quotes.setNumberOfShares(traderRecord.getNo__Of__Shares());
   quotes.setTotalShareValue(traderRecord.getTotal__Share__Value());
   return quotes;
}

public void buy(String company, int numberOfShares) throws Exception {
     trade(company, numberOfShares, true);
}
public void sell(String company, int numberOfShares) throws Exception {
   trade(company, numberOfShares, false);
}
private void trade(String company, int numberOfShares, boolean buy)
   throws Exception {

   traderRecord.setRequest__Type("Buy_Sell");
   traderRecord.setCompany__Name(company);
   traderRecord.setUserid(ivUserID);
   traderRecord.setNo__Of__Shares__Dec((short) numberOfShares);
   if (buy == true) // if buy
      traderRecord.setUpdate__Buy__Sell("1");
   else // if sell
      traderRecord.setUpdate__Buy__Sell("2");
   try {
      callTraderBackend();
   } catch (Throwable t) {
      t.printStackTrace();
      throw new Exception("Error getting quotes: " + t.getMessage());
   }
}
public void logon(String userid, String password) {
   ivUserID = userid;
   ivPassword = password;
}
public void logoff() {
}
```

Figure 6-17   Trader business methods, part 2

After adding these methods to our enterprise bean, add the `buy()`, `getCompanies()`, `getQuotes()`, `logoff()`, `logon()`, and `sell()` business methods to the EJB remote interface. To do this, right click on each method name in the *Outline* view panel, and choose **Enterprise Bean -> Promote to Remote Interface**.

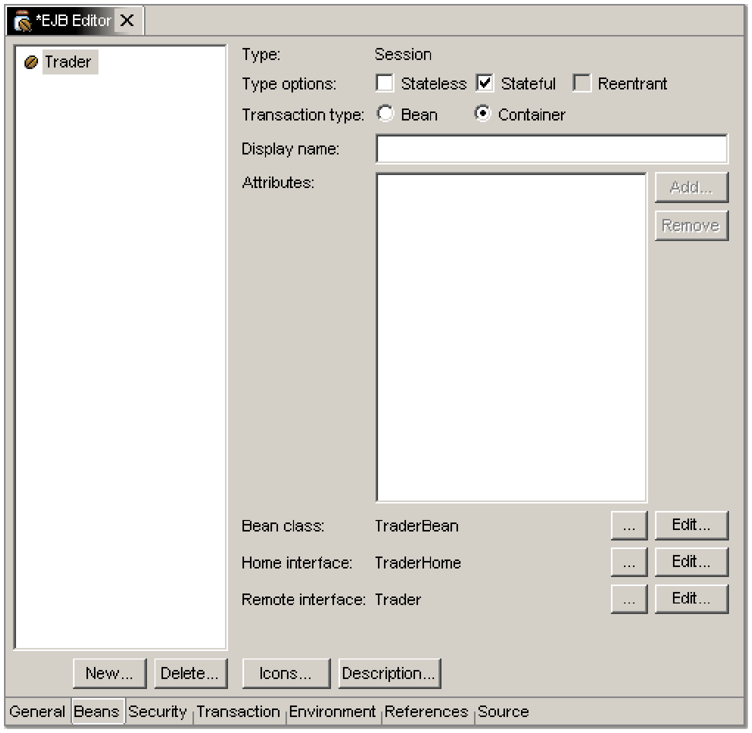
6.3.3 Editing the EJB deployment descriptor

Now that the session bean is complete, you must make some changes to its deployment descriptor. The following steps explain how to make these changes:

1. From the *Navigator* view, right click on the **ejb-jar.xml** deployment descriptor file found in the following folder location:

   ```
   TraderEJB/ejbModule/META-INF/ejb-jar.xml
   ```

2. Select **Open With -> EJB Editor** from the popup menu.
3. This launches the deployment descriptor editor. Click the **Beans** tab at the bottom of the editor.
4. Now click on the **Trader** bean in the left panel of the editor to display the bean properties. You have created a stateful session bean, so change the type option to **Stateful** by clicking the checkbox to ensure that all the remaining values are as shown in Figure 6-18.



Type: Session
Type options: [ ] Stateless [✓ Stateful] [ ] Reentrant
Transaction type: ( ) Bean [✓ Container]
Display name:
Attributes:
Bean class: TraderBean
Home interface: TraderHome
Remote interface: Trader

Figure 6-18 Setting the bean properties

5. Now click on the **References** tab and select the **Resource Reference** radio button. Click on the **Trader** entry in Resource Name column in the table. Click the **Add** button to the right of the table.

6. Complete each of the columns with the values shown below:
   - Resource name: `eis/ECICICS`
   - Type: `javax.resource.cci.ConnectionFactory`
   - Authentication: `Application`
   - Description: `CICS ECI resource adapter`

   Important: The Resource Name value must match what comes after `java:comp/env/` in the JNDI lookup string used to locate the `ConnectionFactory` object. Our JNDI lookup string was:

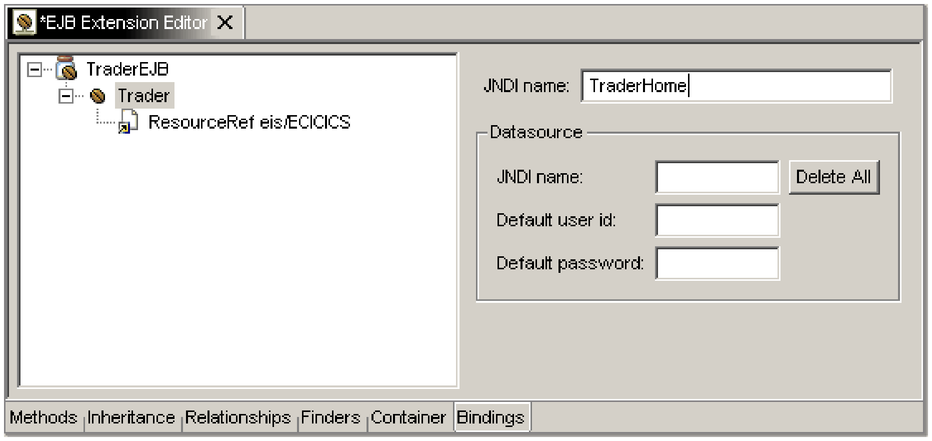
   ```
   java:comp/env/eis/ECICICS
   ```

   and so we set the Resource Name to:

   ```
   eis/ECICICS
   ```

   See "Obtaining the ConnectionFactory object using JNDI" on page 131 for further details.

7. Press Ctrl-S to save the changes to the `ejb-jar.xml` file and close the editor.
8. Now right click on the **ejb-jar.xml** file again and this time select **Open With -> EJB Extension Editor**. This will launch the *EJB Extension Editor*, which allows you to add extra deployment information to the EJB JAR that is not contained in the standard deployment descriptor.
9. Click on the **Bindings** tab at the bottom of the editor and expand the tree under *TraderEJB* to display the *Trader* session bean and the *eis/ECICICS* resource reference. Select the **Trader** session bean and enter a JNDI name in the right hand panel of the editor as shown in Figure 6-19. We used the name *TraderHome*.



JNDI name: TraderHome

Figure 6-19 Adding the JNDI name of the session bean

10. In a similar way, click on the **ResourceRef** and give it a JNDI name of **ECICICS**. Press Ctrl-S to save the file. This will create an additional file named `ibm-ejb-jar-bnd.xmi` in the EJB project in the same folder as `ejb-jar.xml`. This file contains the EJB extensions JNDI bindings information that were just created. Close the EJB Extensions Editor.

Note: It should *not* be necessary to create these JNDI bindings at this time because they can be overridden when installing the EJB JAR file into WebSphere Application Server. However, there will be an exception when trying to start the enterprise bean in WebSphere if you do not create the EJB extensions file prior to installation. See 6.5.3, “Installing the EAR file into WebSphere” on page 150 for information about how the JNDI binding information is used when the enterprise bean is installed into WebSphere.

6.4 Testing the enterprise bean

Now that the session bean is completed, it is necessary to test your application. Application Developer provides the runtime environment of WebSphere Application Server Advanced Single Server V4. This single server version of WebSphere V4 does not provide the runtime environment for J2EE resource adapters, so testing is limited to using a non-managed environment. However, it still proves very useful for testing the other parts of an application; this is explained in this chapter.

To perform a complete test of the application in a managed environment, it is necessary to deploy the application into WebSphere Application Server, as detailed in 6.5, “Deploying the application to WebSphere” on page 149.

Creating a non-managed test environment

We extended our `getConnectionFactory()` method with the capability to manually create a `ConnectionFactory` object instead of using JNDI. This meant that we could choose between using JNDI when in a managed environment or manually creating the `ConnectionFactory` in our application when in the non-managed test environment. We added the same code as we had previously used in 5.1.1, “Writing a simple CCI application” on page 73 to manually create the `ConnectionFactory` object. Then by adding a Boolean value called `isManaged` we were able to switch between what type of environment we were running in. For testing inside Application Developer set this value to `false`. When you are ready to deploy into the managed environment change it to `true`. Figure 6-20 shows the new version of the `getConnectionFactory()` method.

```
private void getConnectionFactory() throws Exception {
   boolean isManaged = false;
   if (isManaged) {
      try {
         Context ic = new InitialContext();
         cf = (ConnectionFactory) ic.lookup("java:comp/env/eis/ECICICS");
      } catch (Throwable t) {
         t.printStackTrace();
         throw new Exception("Lookup for ConnectionFactory failed.");
      }
   } else {
      try {
         ECIManagedConnectionFactory mcf =
            new ECIManagedConnectionFactory();
         mcf.setConnectionURL("local:");
         mcf.setPortNumber("2006");
         mcf.setServerName("SCSCPAA6");
         cf = (ConnectionFactory) mcf.createConnectionFactory();
      } catch (Throwable t) {
         t.printStackTrace();
         throw new Exception("Failed to get ConnectionFactory.");
      }
   }
}
```

Figure 6-20 Modified getConnectionFactory() method used for testing

You will need to change the `ECIManagedConnectionFactory` properties to values that are valid for your system. Note that this sample uses the CTG in local mode.

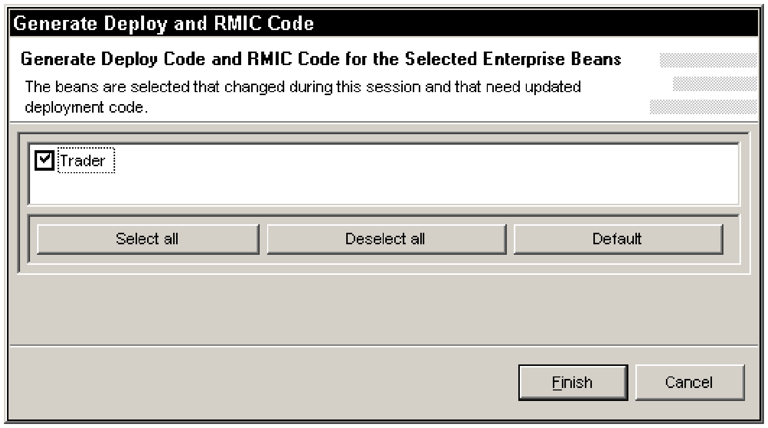
Note: The simplest method of testing in a non-managed environment is to hard code the connection details as shown here. However, the J2EE Connector Architecture specification requires you to use JNDI in both non-managed and managed environments to avoid the need for different code in each case. If you decide to also use JNDI in your non-managed test environment, then you must follow the procedure explained in 5.5, “Using JNDI” on page 100. If you are considering using a non-managed environment in production rather than test, it is recommend that you use JNDI.

Using the EJB test client in Application Developer

Now that you have the necessary code in place, you are ready to test our session bean in the test environment provided by Application Developer. The following steps show how this is done:

Attention: If you are running WebSphere Application Server on the same machine as you are using for WebSphere Studio Application Developer, you will need to either stop WebSphere Application Server, or modify the port 900 used by the JNDI Server in Application Server to prevent a port clash.

1. Before the session bean can be run in an application server, its deployment code must be generated along with the stubs and ties. From the J2EE perspective *Navigator* view, right click on the **TraderEJB** project and select **Generate -> Deploy and RMIC code**. Select the checkbox next to the Trader enterprise bean as shown in Figure 6-21, and click **Finish** to generate the code. This will create all of the necessary deployment classes. You must perform this step whenever you have made changes to the bean code.


Generate Deploy and RMIC Code

Generate Deploy Code and RMIC Code for the Selected Enterprise Beans

The beans are selected that changed during this session and that need updated deployment code.

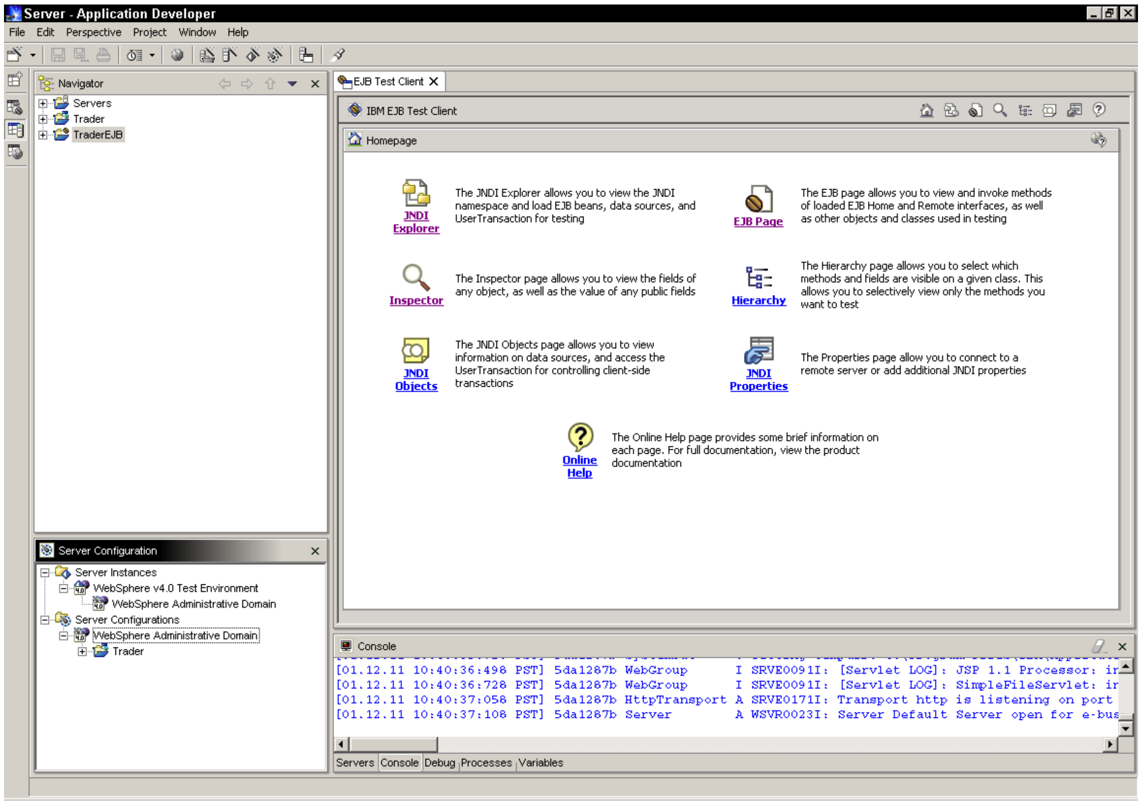
☑ Trader

Figure 6-21 Generating bean deployment code

2. From the J2EE perspective *Navigator* view, right click on the name of the EJB project, in our case **TraderEJB**. Select **Run on Server**.
3. If you have not used the test environment previously, then this action will create the default WebSphere v4.0 Test Environment server instance for you. The *Server* perspective will be opened and the server begins to start. This perspective contains the *Console* view that shows the server log information. Once the server has started, you should see the following message at the bottom of the log:

```
WSVR0023I: Server Default Server open for e-business
```

The EJB Test Client will also be displayed and your screen should look similar to that shown in Figure 6-22.



The JNDI Explorer allows you to view the JNDI namespace and load EJB beans, data sources, and UserTransaction for testing

The EJB page allows you to view and invoke methods of loaded EJB Home and Remote interfaces, as well as other objects and classes used in testing

The Inspector page allows you to view the fields of any object, as well as the value of any public fields

The Hierarchy page allows you to select which methods and fields are visible on a given class. This allows you to selectively view only the methods you want to test

The JNDI Objects page allows you to view information on data sources, and access the UserTransaction for controlling client-side transactions

The Properties page allow you to connect to a remote server or add additional JNDI properties

The Online Help page provides some brief information on each page. For full documentation, view the product documentation

```
[01.12.11 10:40:36:498 PST] 5da1287b WebGroup      I SRVE0091I: [Servlet LOG]: JSP 1.1 Processor: ir
[01.12.11 10:40:36:728 PST] 5da1287b WebGroup      I SRVE0091I: [Servlet LOG]: SimpleFileServlet: ir
[01.12.11 10:40:37:058 PST] 5da1287b HttpTransport A SRVE0171I: Transport http is listening on port
[01.12.11 10:40:37:108 PST] 5da1287b Server        A WSVR0023I: Server Default Server open for e-bus
```

Figure 6-22 Testing the session bean using the EJB test client

4. If you inspect the console log at this point, you will see that the following exception occurred during startup:

```
java.lang.NoClassDefFoundError: javax/resource/cci/InteractionSpec
```

This occurred because the default server instance that was created does not have access to the necessary runtime classes to support the J2EE Connector Architecture and the CICS ECI resource adapter. You must now add these JARs to the server class path by modifying the server instance.

Note: Although the server instance created by the Application Developer does not have the required JAR files on its class path, it was the easiest way to initially create the server configuration. Once it had been created, you will be able to add these JARs and restart the server, as detailed below.

5. Double click on the **WebSphere v4.0 Test Environment** server instance in the *Server Configuration* view of the *Server* perspective. This will open the editor for the server instance.
6. Click on the **Paths** tab at the bottom of the editor to display the class path panel.
7. Click the **Add External JARs** button. Locate the JAR file that you wish to add and click **Open**. Continue adding JARs in this way until all have been added.

   The following JARs from a CTG V4.0.1 installation were added:

   ```
   classes\connector.jar
   classes\cicsj2ee.jar
   classes\ctgclient.jar
   classes\ccf2.jar
   classes\ctgserver.jar
   ```

   and these JARs were added from a WebSphere Advanced V4 installation:

   ```
   lib\eablib.jar
   lib\recjava.jar
   ```

8. Save the server configuration by pressing Ctrl-S and then close the editor.
9. You must now re-start the test environment. Click on the **Servers** tab underneath the Console window to show the list of servers. Right click on the **WebSphere v4.0 Test Environment** server instance and select **Restart**.
10. Wait for the server to restart. This time the console log should show no errors or exceptions on startup, and you can now begin to use the EJB Test Client. Start by clicking on the **JNDI Explorer** icon from within the *Test Client* view. This opens a view of the JNDI name space as shown in Figure 6-23.

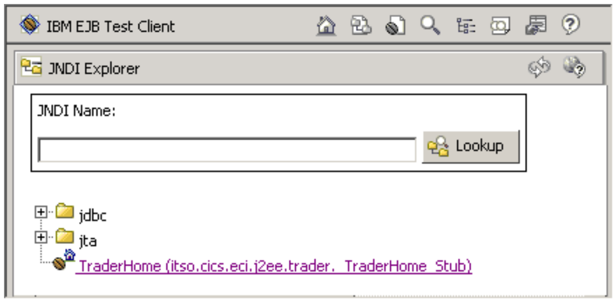
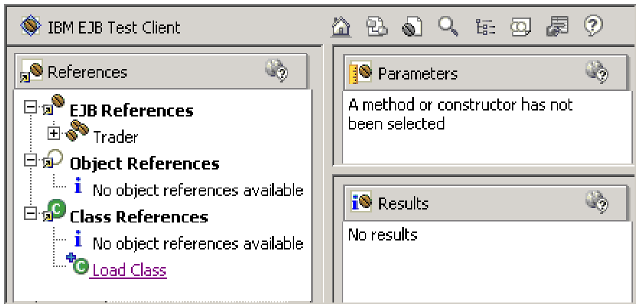


Figure 6-23 JNDI Explorer in the EJB Test Client

11. The home interface of the `Trader` enterprise bean is displayed with the JNDI name `TraderHome` that was specified in 6.3.3, “Editing the EJB deployment descriptor” on page 137. Click on the **TraderHome** entry to open the EJB page of the test client as shown in Figure 6-24.


IBM EJB Test Client

References

EJB References
Trader
Object References
No object references available
Class References
No object references available
Load Class

Parameters

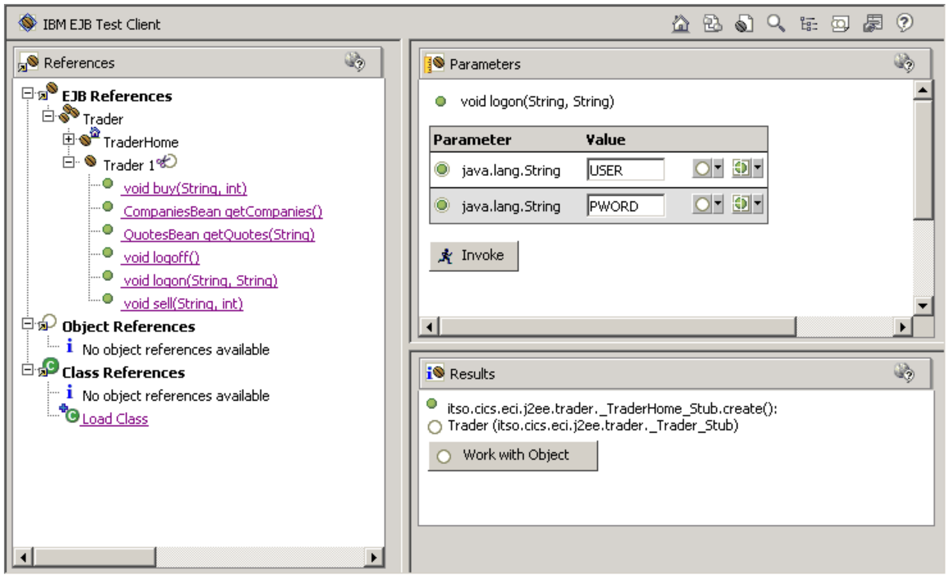
A method or constructor has not been selected

Results

No results

Figure 6-24 EJB page of the EJB Test Client

12. In the *References* panel, fully expand the tree of the Trader *EJB References*, and then click on the home interface’s **create()** method.
13. This will add the `create()` method to the *Parameters* panel. This panel is used to supply any required values to a method before invoking it. As the `create()` method takes no input, just click the **Invoke** button to create an instance of the session bean. The *Results* panel will display the new session bean object. Click the **Work with Object** button that also appears in the *Results* panel at the bottom.
14. This will add an instance of the `Trader` bean named `Trader1` to the *EJB References* section of the *References* panel. Expand the tree on this reference to see the available methods from the bean’s remote interface.
15. At this point, you can choose a method to invoke. Clicking on our `logon()` method causes the method to be displayed in the *Parameters* panel, as shown in Figure 6-25.


IBM EJB Test Client

References

- EJB References
  - Trader
    - TraderHome
    - Trader 1
      - void buy(String, int)
      - CompaniesBean getCompanies()
      - QuotesBean getQuotes(String)
      - void logoff()
      - void logon(String, String)
      - void sell(String, int)
- Object References
  - No object references available
- Class References
  - No object references available
  - Load Class

Parameters

void logon(String, String)

Parameter	Value
java.lang.String	USER
java.lang.String	PWORD

Invoke

Results

itso.cics.eci.j2ee.trader._TraderHome_Stub.create():
Trader (itso.cics.eci.j2ee.trader._Trader_Stub)

Work with Object

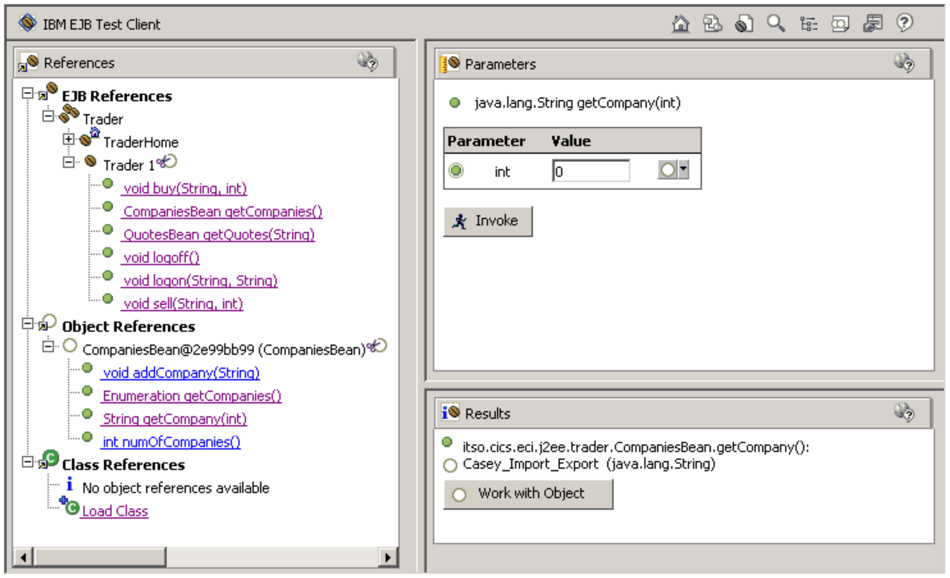
Figure 6-25 Invoking methods on the bean's remote interface

16. This method takes two strings as input. Enter a username into the first box and a password into the second, and click **Invoke,** which causes the method to run successfully, as reported in the *Results* panel.

17. Next, try the `getCompanies()` method, which actually makes a connection to CICS. Click on the method name and then click the **Invoke** button in the same way as the `logon()` method. The `getCompanies()` method returns an object of type `CompaniesBean`, which is one of the data bean classes that was used to hold information about the companies available for trading.

18. The `CompaniesBean` object now appears in the *Results* panel. Click on the **Work with Object** button so that you can look at the data contained by this bean. This adds a CompaniesBean instance to the Object References tree in the *References* panel. By expanding the tree on this object, you can choose one of its available public methods in the same way as you chose a remote interface method on the session bean. Click on the **getCompany()** method.

- Set the input `int` parameter in the *Parameters* panel to **0** and click **Invoke**. This returns the name of the first company, Casey_Import_Export, as shown in Figure 6-26.



java.lang.String getCompany(int)

Parameter	Value
int	0

itso.cics.eci.j2ee.trader.CompaniesBean.getCompany():
Casey_Import_Export (java.lang.String)

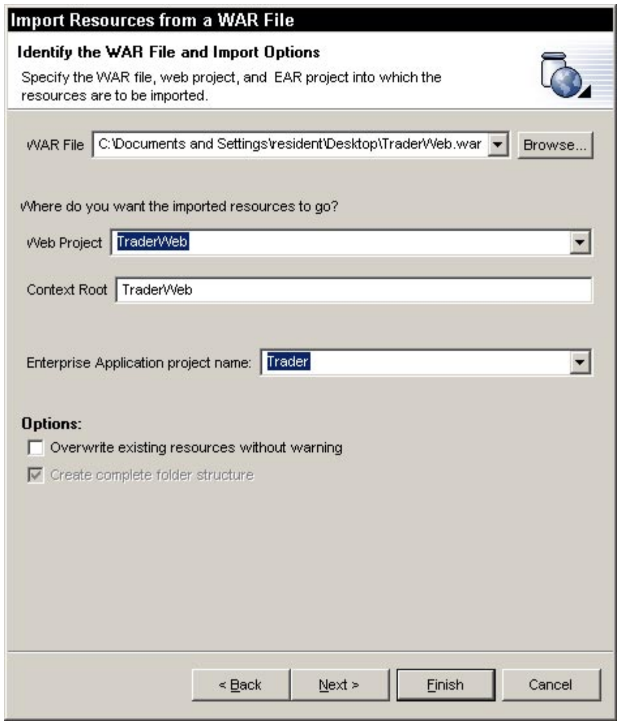
Figure 6-26 Testing the getCompany() method

19. By using the same techniques, you will be able to use the test client to test each of the remaining methods in your session bean.

Importing the Web components of the application

Now that the session bean is completed and successfully working, you can add the Web components of the application to the workspace, and then test the completed Web application. The following steps explain how to do this:

1. Choose **File -> Import** from the menu system.
2. Select **WAR file** as the import source and click **Next**.
3. Click the **Browse** button and locate the supplied file `TraderWeb.war`. This contains the servlet and JSPs for our application, as well as additional data bean classes. Select the file and click **Open**. Complete the remainder of the dialog box as shown in Figure 6-27.


Import Resources from a WAR File

Identify the WAR File and Import Options

Specify the WAR file, web project, and EAR project into which the resources are to be imported.

WAR File: C:\Documents and Settings\resident\Desktop\TraderWeb.war

Where do you want the imported resources to go?

Web Project: TraderWeb

Context Root: TraderWeb

Enterprise Application project name: Trader

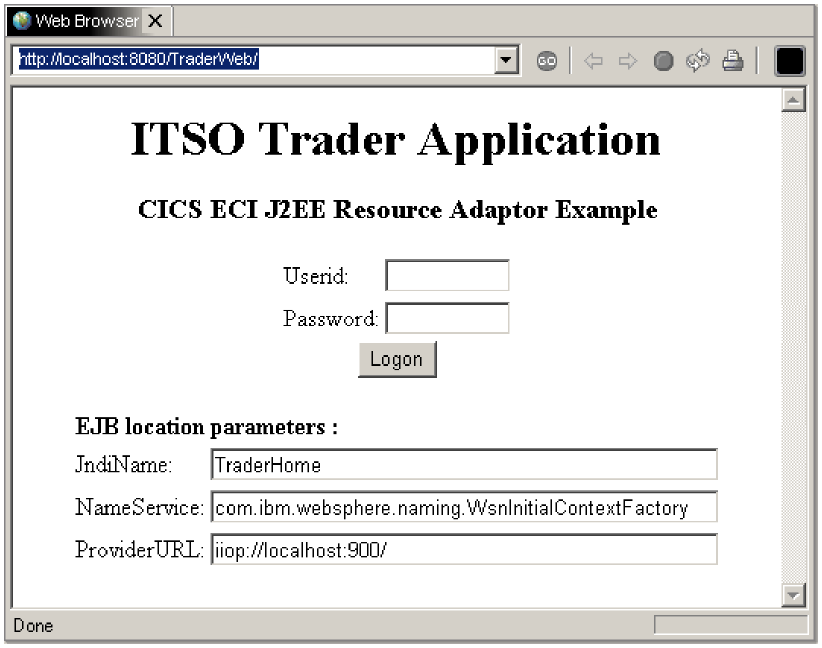
Options:

☐ Overwrite existing resources without warning

☑ Create complete folder structure

Figure 6-27 Importing the WAR file

4. Click **Next** to display the *Module Dependencies* dialog.
5. Click the checkbox for `TraderEJB.jar` to add the session bean to the servlet's build and runtime classpaths. Click **Next** to display the *Define Java Build Settings* dialog.
6. The *TraderEJB* project has already been added to the build class path in the previous step, so click **Finish** as there is no need to add any further resources.
7. The workspace will now contain the completed enterprise application project *Trader*, and its two module projects *TraderEJB* and *TraderWeb*. You can now test the application. Restart the test environment like before by switching to the *Servers* view; right clicking on the server instance, and choosing **Restart**.
8. Check the *Console* view to ensure that the server started without any errors.
9. This time, right click on the **TraderWeb** project in the *Navigator* view and choose **Run on Server**. This will open the welcome page, `Logon.html`, of the Web application in a *Web Browser* view. Your screen should look similar to that shown in Figure 6-28.


Web Browser

http://localhost:8080/TraderWeb/

ITSO Trader Application

CICS ECI J2EE Resource Adaptor Example

Userid:

Password:

Logon

EJB location parameters :

JndiName: TraderHome

NameService: com.ibm.websphere.naming.WsnInitialContextFactory

ProviderURL: iiop://localhost:900/

Done

Figure 6-28 Testing the Web application

10. Enter a username and password, and click **Logon** to begin testing the completed application.

Note: The EJB location parameters tell the servlet that processes the request where it can find the session bean in a JNDI namespace. If you used a different name than `TraderHome` for the enterprise bean's JNDI binding, you must change the *JndiName* value in this form to match the name you chose. You also have the ability to change the *NameService* and *ProviderURL* values, but the defaults are correct for the scenario that is described.

6.5 Deploying the application to WebSphere

Now that the application is complete and tested in the non-managed environment of Application Developer, you are ready to deploy to a managed environment. We used WebSphere Application Server Advanced Edition V4.0.1 on a Windows 2000 server to provide the managed environment for the resource adapter and to host your application. Deployment of your application involves the following steps:

1. Enable JNDI in the application.
2. Export the application from Application Developer.
3. Install the EAR file into WebSphere.

6.5.1 Enabling JNDI in the application

Before the application is ready to export, you must make one small change to be able to run in the managed environment. The `getConnectionFactory()` method in `TraderBean.java` needs changing so that it uses JNDI to find the `ConnectionFactory` object. Take the following steps:

1. Double click on the **`TraderBean.java`** file from the *Navigator* view to open the Java editor.
2. Change the `isManaged` boolean declaration at the beginning of the method to set the value to true, as shown:

   ```
   boolean isManaged = true;
   ```

3. Save the changes by pressing Ctrl-S and then regenerate the deployment code, stubs, and ties as described in, “Using the EJB test client in Application Developer” on page 141.

6.5.2 Exporting the application from Application Developer

The application is now ready to export. Follow these steps to export the application as an EAR file. The EAR file will contain two components, an EJB JAR file and a WAR file, which represents the EJB module project and the Web module project respectively.

Note: Sometimes it is necessary to rebuild the projects before exporting them from Application Developer. Doing this will force a re-compilation of all of the code and sometimes fixed problems during runtime. To rebuild a project, select it from the *Navigator* view, and choose **Project -> Rebuild All** from the menu system. Do this for each of the three projects in the workspace. It is also recommended that you re-generate the EJB deployment code after doing a rebuild of the TraderEJB project.

1. Choose **File -> Export** from the menu system, and choose **EAR file** as the export destination.
2. Select the enterprise application project **Trader** from the drop down box as the resource to be exported. Click **Browse** to choose where to save the file. The examples saves it as `Trader.ear`.
3. Click **Finish** to generate the EAR file.

6.5.3 Installing the EAR file into WebSphere

We prepared our WebSphere Application Server installation by installing the CICS ECI resource adapter described in Chapter 6.2, “Configuring WebSphere Application Server” on page 115. We also added a connection factory instance to connect to a secure CICS server through a local gateway. The following parameters were set on the connection factory using the menu **WebSphere Administrative Domain** -> **Resources -> J2C Resource Adapters -> J2C Connection Factory -> CICS ECI > New**

- **Name:** SCSCPAA7 - local
- **JNDI binding path:** eis/local/SCSCPAA7
- **ConnectionURL:** local:
- **ServerName:** SCSCPAA7
- **UserName:** CICSRS2
- **Password:** PASSW0RD

The enterprise application can now be deployed to WebSphere Application Server Advanced V4. The following instructions show how to install the EAR file:

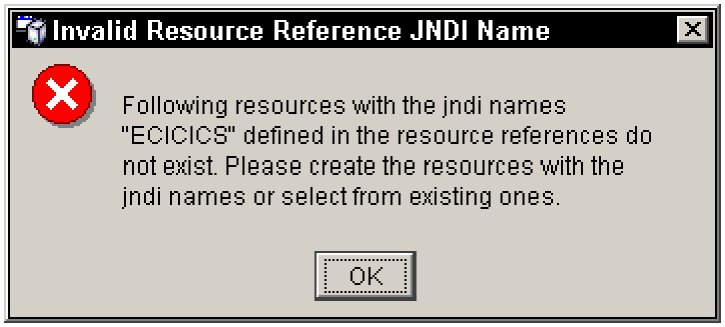
1. Start the WebSphere Admin Server process and launch the Administrative Console.
2. From the Administrative Console menu system choose **Console -> Wizards -> Install Enterprise Application**.
3. Choose the **Install Application (*.ear)** radio button and click **Browse** to locate the `Trader.ear` file that was previously exported from Application Developer. Select the file and click the **Open** button.

Note: We found that the EAR file must be located on the local hard disk. Any attempt to use a network drive resulted in an exception box being displayed.

4. Enter **Trader** for the application name and click **Next**.
5. Click **Next** two more times to display the *Binding Enterprise Beans* to JNDI *Names* dialog. This allows you to change the JNDI name under the session bean that is bound in the JNDI namespace. The JNDI name will already be

set to whatever was previously specified in the EJB Extension Editor bindings. See 6.3.3, “Editing the EJB deployment descriptor” on page 137, which is TraderHome. Click **Next** to accept this value.

- Click **Next** again to display to the *Mapping Resource References to Resources* dialog. This will result in the error message shown in Figure 6-29.


Invalid Resource Reference JNDI Name

Following resources with the jndi names "ECICICS" defined in the resource references do not exist. Please create the resources with the jndi names or select from existing ones.

Figure 6-29 Invalid Resource Reference JNDI name error message

This error arises because the JNDI binding name ECICICS that are specified for the eis/ECICICS resource reference (see 6.3.3, “Editing the EJB deployment descriptor” on page 137) does not match with the JNDI name of an existing connection factory resource in WebSphere. At this point you can either cancel the installation, or you can specify that you want to use the JNDI name of a resource that does exist. We suggest that you use the connection factory that was configured previously with the JNDI name eis/local/SCSCPAA7. To do this click **OK** and follow the remaining instructions.

6. Click the **Select Resource** button, and choose the name of the connection factory resource that you want to use. The name of the resource in WebSphere, rather than its JNDI name, is displayed. Chose the *SCSCPAA7 - local* resource that was previously created. Click **OK** and click **Next** again.
7. Click **Next** four more times to accept the remaining defaults, and then click **Finish**. When prompted whether to regenerate the application code for installation choose **No** because this was already created in Application Developer.
8. A dialog box will appear once installation has completed. Click **OK**.
9. The Web Server plugin must now be regenerated. From the Administrative Console, right click on your server node found under the Nodes tree and select **Regen Webserver Plugin**.
10. If the server is currently running, you must stop it and then restart it. If it is already stopped, then you can skip this step. Open the Application Servers

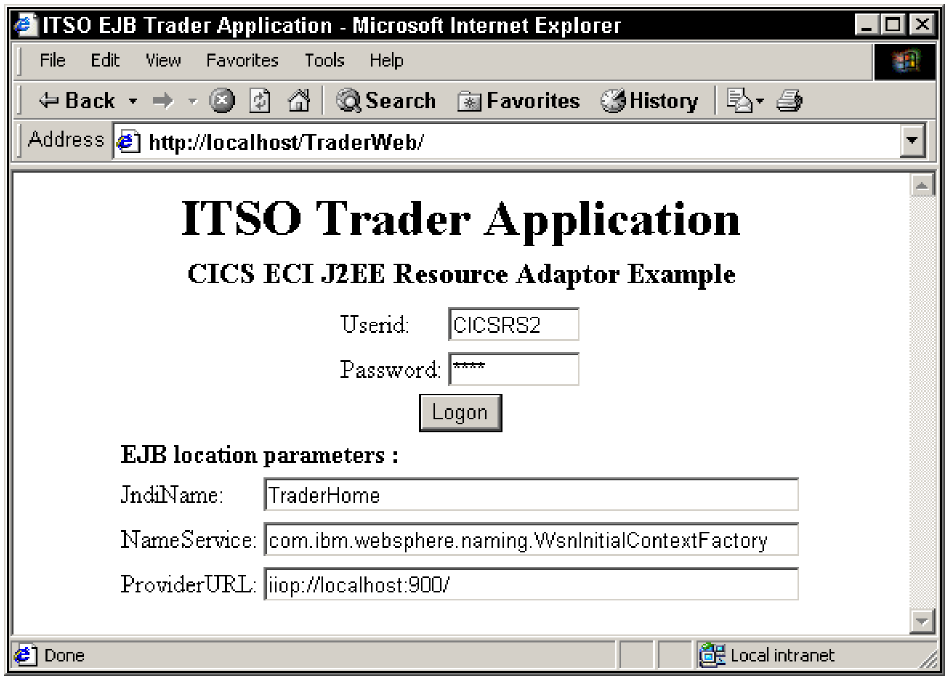
tree under your server node, right click on the application server (the default Server in this case) and choose **Stop**. Wait for the dialog box to indicate that the server has stopped, and click **OK**.

11. Right click on the application server again, and select **Start** to start the server. Once the server has started, click **OK** to dismiss the dialog box.

    At this point you can start a Web browser and enter the following URL to begin using the Trader application:

    ```
    http://<server_name>:<port>/TraderWeb
    ```

    where <server_name> and <port> are the details for your server. Figure 6-30 shows the login screen of the completed application.


ITSO EJB Trader Application - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address
Logon
Done
Local intranet

http://localhost/TraderWeb/

ITSO Trader Application

CICS ECI J2EE Resource Adaptor Example

Userid: CICSRS2

Password: ****

EJB location parameters :

JndiName: TraderHome

NameService: com.ibm.websphere.naming.WsnInitialContextFactory

ProviderURL: iiop://localhost:900/

Figure 6-30 Completed Trader application

You are now ready to start using the Trader application to interact with CICS.

Part 3

Connecting to 3270 based CICS transactions

In Part 3 we provide information on how to develop applications that invoke 3270 based CICS transactions using either the EPI support classes, or the CCI and the CICS EPI resource adapter.

7

EPI support classes

Several Java classes are supplied with the CICS Transaction Gateway (CTG) that facilitate the creation of Java EPI applications. These classes are known as the *EPI support classes* and are used to connect from a Java environment to CICS transactions that were originally created for a 3270 terminal. Because of their ease-of-use you should choose to use them in place of the lower level programming interface classes, which require you to construct `EPIRequest` objects. In this chapter we describe how to use these EPI support classes in your applications.

We provide examples of:

- Using the basic EPI support classes to connect to a CICS transaction
- Using the `Map` class and the **`BMSMapConvert`** utility
- Using the provided exception classes for error handling
- Configuring EPI applications that call secured CICS transactions

We begin by building a basic EPI application that connects to the simple CICS transaction EPIP, which runs our sample server program EPIPROG. We then show how to use each of the EPI support classes to create a more complex EPI application.

The sample EPI applications shown in this chapter are offered as a set of classes in the package `itso.cics.epi`. Instructions on how to download the sample code are in Appendix D, “Additional material” on page 261.

7.1 Creating a simple EPI application

This section explains each of the basic EPI support classes, and demonstrates how they can be used to construct a simple EPI application that calls a CICS transaction. The sample code for this EPI application is provided as the class `itso.cics.epi.Simple`. To download this class, refer to Appendix D, “Additional material” on page 261.

Important: To be able to use the EPI support classes, your application must include the `ctgclient.jar` file in its classpath. This file can be found in the `classes` subdirectory of a CTG installation. In addition, we recommend that your application includes an `import` statement like the one shown below, which allows you to reference the EPI classes without fully qualifying their package names:

```
import com.ibm.ctg.epi.*;
```

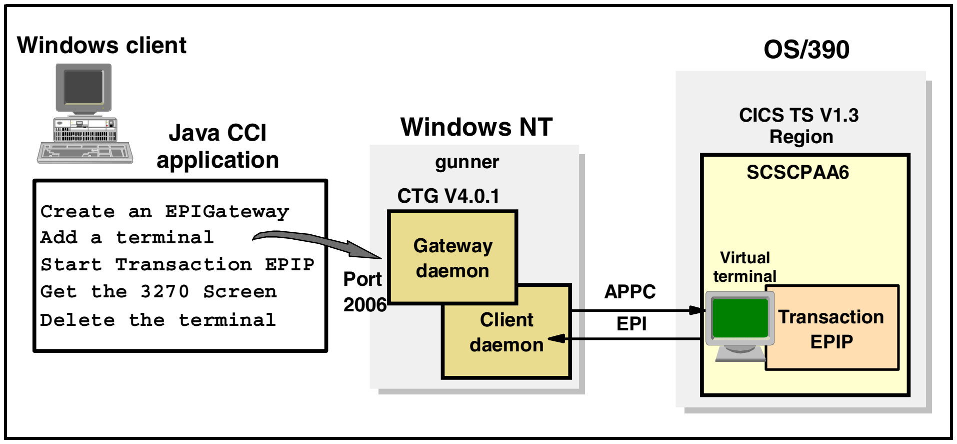


Figure 7-1 CTG scenario for EPI support classes

All of the examples in this chapter use the CTG V4.0.1 on a Windows NT server for our gateway, and CICS TS V1.3 for our CICS server.

In this section we show how to use the following classes:

- `com.ibm.ctg.epi.EPIGateway`
- `com.ibm.ctg.epi.Terminal`
- `com.ibm.ctg.epi.Screen`
- `com.ibm.ctg.epi.Field`

7.1.1 Using the EPIGateway class

For an application to be able to send and receive data from a CICS server, it needs to connect to a CICS Transaction Gateway. An EPI application begins by creating an object to represent this connection. This object can be an instance of either the standard `JavaGateway` class, or its enhanced subclass `EPIGateway`, which is provided by the EPI support classes. The `EPIGateway` class extends the `JavaGateway` class by offering the following additional methods that return information about what CICS servers have been defined to the CTG.

- `serverCount()`
- `serverName()`
- `serverDesc()`

These methods allow an application to find out information about what servers have been defined to the CTG. If you do not wish to use these methods, then a `JavaGateway` object will suffice for creating the connection to the CTG.

Note: Any information about a server definition that is returned by these `EPIGateway` methods contains no indication of the actual *status* of the server. The existence of a server definition in the CTG configuration file is no guarantee of the availability of that CICS server. The CICS server may be incorrectly configured and/or unavailable. You can find out the status of the CICS server by flowing a test ECI request. See “Status information calls” on page 39 for details.

An `EPIGateway` can be created by using its default constructor, and then calling setter methods for each of its properties. Once these are set, the `open()` method is called and creates the connection with the CTG. There is also an overloaded constructor that simplifies the process by setting each of the properties, and implicitly calling `open()` for you. Figure 7-2 shows how we created an `EPIGateway`, and used this to connect to the CTG and to list the defined servers and their descriptions.

```
try {
1  EPIGateway epiGate = new EPIGateway("tcp://gunner", 2006);

2  int numSystems = epiGate.serverCount();
   System.out.println("Number of servers:" + numSystems);

   for (int sysCount = 1; sysCount <= numSystems; sysCount++) {
3     System.out.println("Server Name=" + epiGate.serverName(sysCount));
4     System.out.println("Description=" + epiGate.serverDesc(sysCount));
   }

5  epiGate.close();
}
6 catch (EPIException epiEx) { epiEx.printStackTrace(); }
catch (java.io.IOException ioEx) { ioEx.printStackTrace();
}
```

Figure 7-2 Using the EPIGateway class to connect to the CTG

The following list explains the code we used:

- 1 Create a new `EPIGateway` object called `epiGate`. In our example, the URL and port number are passed into the constructor, which uses these to open the socket connection to the gateway daemon.
- 2 Call the `serverCount()` method; this returns the number of servers that are defined to the gateway as an integer.
- 3 Call the `serverName()` method to return the name of each CICS server as a String.
- 4 Get the server description from the CTG by calling the `serverDesc()` method. This step, and the previous one, are both repeated for all of the servers that have been defined to the CTG.
- 5 Close the gateway connection.
- 6 Catch any exceptions. Exception handling is discussed in 7.2.2, “Exception handling” on page 169.

An example of the output produced by this program is shown in Example 7-1.

Example 7-1 Sample output from the code shown in Figure 7-2

```
Number of servers:2
Server Name=SCSCPAA6
Description=CICS TS SCSCPAA6
Server Name=SCSCPAA7
Description=CICS TS SCSCPAA7
```

7.1.2 Using the Terminal class

Once the gateway has been opened, the EPI application now needs to establish a terminal resource in CICS. Any transactions initiated using the EPI interface will be associated with this terminal when running in CICS. The EPI support classes supply a class, `com.ibm.ctg.epi.Terminal` to create this required CICS terminal resource.

A default constructor is provided to initially create the `Terminal` object and setter methods are then used to specify the values of any relevant parameters. Once these parameters have been set, you must call the `connect()` method, which causes the terminal resource to be created on the CICS server. Alternatively, you can use one of the overloaded constructors that will set the required parameters. Whenever the `connect()` method is invoked, it causes the CICS supplied transaction CTIN to be driven on the server and installs the requested terminal. See Figure 7-10 on page 174 for further details about the CTIN transaction.

There are two different types of `Terminal` objects that can be created, either a *basic terminal* or an *extended terminal*. We describe each type below.

Basic terminals

A basic terminal offers a subset of the features provided by an extended terminal. It is created using the `Terminal` object by specifying just the gateway to use and the name of the CICS server. You can also optionally specify the netname and device type that you want CICS to use when it auto-installs the terminal. By setting both values to `null`, you can allow CICS to automatically decide what values to use.

There are two ways of creating a basic `Terminal` object. The first way is to create the object using the default constructor (before using the setter methods shown below) to set each of the properties:

- `setGateway()`
- `setServerName()`
- `setDeviceType()`
- `setNetName()`

Once you have set each value, you must then call the `connect()` method which will send your terminal install request to CICS.

The second way is to use the overloaded `Terminal` constructor, which will set each value and implicitly call `connect()` for you. We recommend that you use this for convenience. We have used this constructor in all of our examples that use a basic terminal.

Note: If you specify a device type of `null` when creating the `Terminal` object then the CTG that you connect to will use the model terminal definition parameter as specified on the MODELTERM parameter in the `CTG.INI` file. A different MODELTERM value can be set for each CICS server that is defined to the CTG. If no value has been set then the value used will be a CICS server-specific default value. Because we set the value to `null` in our application and also did not define a MODELTERM value in the CTG then we allowed our CICS server to use its default value.

Extended terminals

An extended terminal allows you to set the same values as for a basic terminal but also offers the following extra features:

- Signon capability
- Install timeout
- Read timeout
- Data stream codepage (encoding)
- Security credentials (user ID and password)

Setter methods similar to those used for the basic terminal are provided for each of these extra parameters. If you set any of these values, which are not provided by a basic terminal, then your terminal resource will automatically become an extended terminal. Once again, you can choose between using the default constructor and then the setter methods, or calling the overloaded constructor. However, unlike the one for a basic terminal, the overloaded constructor for an extended terminal does not call `connect()` for you, so you must call this method after creating the extended `Terminal` object.

Using a basic terminal

In the following example, we show you how to create a basic terminal. For information on extended terminals refer to 7.3, “Connecting to secured CICS transactions” on page 174.

As well as enabling an application to create a terminal resource in CICS, the `Terminal` class also has a `send()` method, which allows you to start a CICS transaction. From your EPI application, you can begin the transaction as if it had been initiated, by entering its name at a real 3270 terminal. The example in Figure 7-3 shows how we used the `send()` method of the `Terminal` class to start transaction EPIP, which ran our example program EPIPROG. When creating the `Terminal` object, we used the overloaded constructor for a basic terminal that allowed us to specify the gateway, server name, netname, and device type parameters. The netname and device type parameters allow you to explicitly state which netname and terminal model you want to use for installing the terminal on the CICS server.

```
try {
   EPIGateway epiGate = new EPIGateway("tcp://gunner", 2006);

1  Terminal term = new Terminal(epiGate, "SCSCPAA6", null, null);

2  term.send("EPIP", null);
3  term.setPurgeOnDisconnect(true);
4  term.disconnect();

   epiGate.close();
}
catch (EPIException epiEx) { epiEx.printStackTrace(); }
catch (java.io.IOException ioEx) { ioEx.printStackTrace(); }
```

Figure 7-3 Using the Terminal class to start a transaction

The new code that was not in the previous example is explained in the following list:

- 1 Create the CICS terminal.

  This `Terminal` constructor sets the gateway URL to use and the CICS server name, SCSCPAA6. By specifying `null` for the `netname` and `devicetype` parameters, we allowed the defaults to be used. The constructor that we used created a basic terminal because it did not set any of the extended terminal parameters. Note that the constructor also calls the `connect()` method, so at this point, CICS will also install the terminal resource by using the CTIN transaction.

- 2 Start the transaction EPIP by using the `send()` method on our `Terminal` object.

  The first parameter for the `send()` method is the transaction name, and the second allows for any input data to be sent as well. This transaction required no input data, so we set the second parameter to `null`. When the EPIP transaction was started, it ran our example EPIPROG program. This example is extended in 7.1.3, "Using the Screen and Field classes" on page 162, and shows how we retrieved the output data from the transaction.

- 3 Turn on transaction purging for the terminal disconnect.

  By default, any outstanding events or transactions that are still running when the `disconnect()` method is invoked (see 4) will cause a `TerminalException` to be thrown. Invoking `setPurgeOnDisconnect(true)` before invoking `disconnect()` will cause any such outstanding transactions to be purged when the `disconnect()` method is called, and therefore, will allow the terminal to be successfully deleted without an exception being thrown, as well as allow for the normal completion of the Java client application. In our example, there should be no outstanding transactions since the EPIP transaction ends immediately.

- 4 Delete the CICS terminal.

  Although we have managed to initiate a transaction on the CICS server, the `Terminal` class does not allow for simulating more advanced interactions with a 3270 screen. For example, we were not able to retrieve the resulting APPLID, date and time values that would be displayed to an end-user with a real 3270 terminal when running our EPIP transaction. We must look at some extra EPI support class to see how to achieve this.

7.1.3 Using the Screen and Field classes

To be able to simulate more complex interactions with the terminal, we use two more EPI support classes:

- `com.ibm.ctg.epi.Screen`
- `com.ibm.ctg.epi.Field`.

The `Screen` class represents the actual display of the 3270 terminal screen and provides methods to query and set screen information. Similarly, the `Field` class represents a field on a 3270 screen and provides methods to query and set the contents and attributes of the field. The example in Figure 7-4 shows how we used these two classes to retrieve the output from our EPIP transaction. We also used the `Screen` and `Field` classes to input the transaction name, rather than by using the `send()` method of the `Terminal` class as in the previous example.

```
try {
   EPIGateway epiGate = new EPIGateway("tcp://gunner", 2006);
   Terminal term = new Terminal(epiGate, "SCSCPAA6", null, null);

1  Screen scr = term.getScreen();

2  Field fld = scr.field(1);
3  fld.setText("EPIP");
4  scr.setAID(AID.enter);

5  term.send();

6  for (int i = 1; i <= scr.fieldCount(); i++) {
      fld = scr.field(i); // get field by index
      if (fld.textLength() > 0)
         System.out.println("Field " + i + ":" + fld.getText());
   }

   term.disconnect();
   epiGate.close();

} catch (EPIException epiEx) { epiEx.printStackTrace();
} catch (java.io.IOException ioEx) { ioEx.printStackTrace();
}
```

Figure 7-4 Using the Screen and Field classes to send and retrieve data

The new code that was not in the previous examples is explained in the following list:

- **1** Create the `Screen` object.

  The `getScreen()` method on the `Terminal` object returns the current screen associated with our terminal. Since the `Terminal` object was just created, the resulting screen consists of only one empty field that covers the entire screen.

- **2** Create the `Field` object.

  Create a `Field` object by using the `field()` method on the `Screen` object to get the first (and only) field of the screen.

- **3** Set the transaction to run to EPIP.

  Set the text value of the field to EPIP by using the `setText()` method. This is equivalent to typing EPIP in the top left corner of a clear 3270 terminal screen.

- **4** Set the key press to send to CICS.

Each `Screen` object has an attention identifier (AID) that is associated with it that represents the key press that caused the screen to be sent to CICS. We wanted to simulate a 3270 terminal user who is typing the transaction ID, EPIP, and then pressing the Enter key. Therefore, we set the AID of the `Screen` object to the enter key by using the `com.ibm.ctg.epi.AID` class.

- **5** Send the prepared `Screen` to CICS.

  This starts the transaction in CICS.

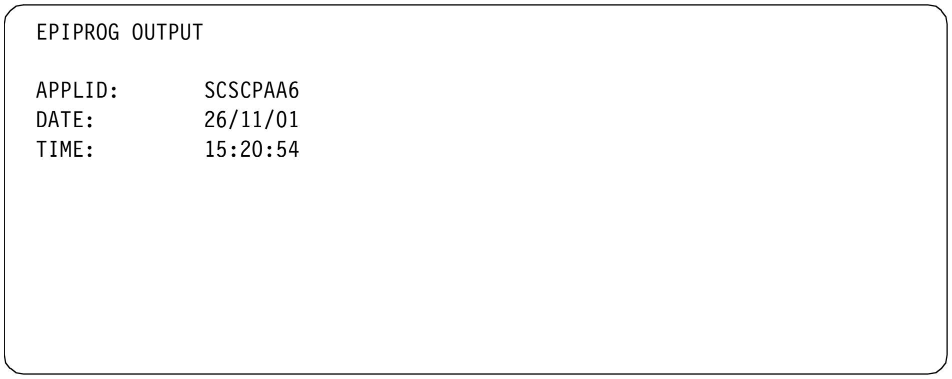
- **6** Retrieve the output data from EPIP.

  After the transaction has executed, it attempts to output data to the terminal it ran on. This caused our `Screen` object, `scr` to be updated to represent the output screen that the EPIP transaction generated. We used the code in this section to retrieve each of the fields from this output screen, and to display the text value of any non-empty fields. An example of the output generated by this application is shown in Figure 7-2.

Example 7-2 Output from our application using the Screen and Field classes

```
Field 1:EPIPROG OUTPUT
Field 2:APPLID:
Field 3:SCSCPAA6
Field 4:DATE:
Field 5:16/11/01
Field 6:TIME:
Field 7:11:39:03
```

The CICS program, EPIPROG, which we ran by invoking transaction EPIP, uses a BMS map when displaying its output data to a 3270 screen. The BMS map we used for EPIPROG was called EPIMAP, and was part of mapset EPIMAPS. The mapset containing this particular map is shown in Appendix C.3, “EPIPROG” on page 246. Figure 7-5 shows the CICS 3270 screen that is created by the BMS map when EPIPROG executes.



```
EPIPROG OUTPUT

APPLID:      SCSCPAA6
DATE:        26/11/01
TIME:        15:20:54
```

Figure 7-5 EPIP 3270 output

The BMS map defines seven fields (seven DFHMDF statements.) These are the seven fields we can see in the example output shown in Example 7-2 on page 164. Our EPI application also makes use of the `getRow()` and `getColumn()` methods on each of the `Field` objects in order to find out the positions at which each of the fields is to be displayed on a 3270 screen. The values returned by these methods correspond to the row and column values specified by the POS attributes of each DFHMDF field in Figure 7-6. Further information about using the EPI with BMS maps can be found in 7.2.1, "Using the Map class and the BMSMapConvert utility" on page 166.

7.2 Extending the EPI application

We have already seen how to connect to a simple CICS transaction, now we will explore some of the advanced features of the EPI support classes including:

- Using information from BMS maps, and connecting to conversational CICS transactions. The sample code for this application is provided as the class `itso.cics.epi.MapClass`.
- Error handling using the EPIException class. The sample code for this application is provided as the class `itso.cics.epi.ErrorHandling`.
- Working with secured CICS applications that require a CICS sign-on. The sample code for this application is provided as the classes `itso.cics.epi.SignonCapable` and `itso.cics.epi.SignonInCapable`.

For instructions on how to download any of our sample code refer to Appendix D, "Additional material" on page 261.

7.2.1 Using the Map class and the BMSMapConvert utility

Many CICS applications use BMS for sending and receiving data to and from a 3270 terminal or terminal emulator. A BMS map controls the display of input and output data by describing where the fields are to be positioned on the screen, and what display attributes they have, such as color. By using a BMS map, the presentation logic is separated from the control logic in the program. The server program controls what data is used to populate the fields, and the BMS map controls the actual display of this data.

It is unlikely that your EPI applications will be concerned with how the data from the server program is intended to be displayed on a regular 3270 terminal. For example, to obtain the date value from the server program EPIPROG, we would have to know that this information was contained in the fourth field on the screen. However, if the BMS map was altered, our EPI application must have changed, because the date value may have been moved to a different field number. What is required is a way of directly accessing the data contained in the date field without, being concerned with the presentation logic that was used when displaying the results on a 3270 terminal. The EPI support classes allow us to do this through the use of two features:

- The EPI `Map` support class
- A converter tool, the `BMSMapConvert` utility

The `BMSMapConvert` utility takes a BMS map and generates from it the source code for a Java class. This class extends the provided Map class and represents the BMS map used by the server program. This generated class is used by our application to achieve the goal of referring to data values by name, rather than indirectly by their screen position. Our client application no longer needs to know about the display layout that the BMS map defined. If changes are made to the BMS map then we simply re-use the converter tool and replace the old version of the map class with the newly generated one.

To demonstrate how the `BMSMapConvert` utility is used, we used the server transaction EPIP that ran our CICS program EPIPROG. We began by using the converter tool on the BMS mapset EPIMAPS that EPIPROG uses to display data (this map can be found on Appendix C.3, “EPIPROG” on page 246). To invoke the converter utility, we first saved the BMS mapset to our workstation with the filename `epimaps.bms`. Then, we added `ctgclient.jar` to the system classpath and ran the `BMSMapConvert` utility as shown in Example 7-3.

Example 7-3 Output from the BMSMapConvert utility

```
set CLASSPATH=%CLASSPATH%;C:\Program Files\IBM\IBM CICS Transaction
Gateway\classes\ctgclient.jar
C:\>java com.ibm.ctg.epi.BMSMapConvert -p itso.cics.epi epimaps.bms
CCL6704I: BMS Map Converter
CCL6706I: Classes will be created in package itso.cics.epi.
```

```
Reading BMS Files
Reading file epimaps.bms
Finished processing BMS Files
Generating Map class for map EPIMAP
```

Each map class name is named by concatenating the original map name with the word *Map*, in our case creating the *EPIMAPMap* class. The converter utility will create a map class for every map in the mapset. In our case there was only one map (EPIMAP) in our mapset (EPIMAPS) so only one class (EPIMAPMap) was generated. As well as supplying the name of the BMS source file, we also specified that we wanted the generated EPIMAPMap class to be in package `itso.cics.epi` by using the -p option. The **`BMSMapConvert`** utility then created a directory structure to match this package name and placed the `EPIMAPMap.java` file into this package subdirectory.

The file `EPIMAPMap.java` was created by the converter and put into the `itso\cics\epi` subdirectory of the directory that contained `epimaps.bms`. We moved this subdirectory structure into the directory where our EPI application was and added the following statement `import itso.cics.epi.EPIMAPMap` to our sample code to reference this new class. We show how our application was then able to make use of this new `EPIMAPMap` class in Figure 7-6.

```
try {
   EPIGateway epiGate = new EPIGateway("tcp://gunner", 2006);
   Terminal term = new Terminal(epiGate, "SCSCPAA6", null, null);
   Screen scr = term.getScreen();
   Field fld = scr.field(1);
   fld.setText("EPIP");
   scr.setAID(AID.enter);
   term.send();

1  EPIMAPMap epimap = new EPIMAPMap(scr);

2  fld = epimap.field(epimap.TIME);
   System.out.println("Time : " + fld.getText());
   fld = epimap.field(epimap.DATE);
   System.out.println("Date : " + fld.getText());
   fld = epimap.field(epimap.APPLID);
   System.out.println("Applid : " + fld.getText());

   term.disconnect();
   epiGate.close();
} catch (EPIException epiEx) { epiEx.printStackTrace();
} catch (java.io.IOException ioEx) { ioEx.printStackTrace();
}
```

Figure 7-6 Using a BMS map class generated by BMSMapConvert

The new code to utilize this `EPIMAPMap` class is explained in the following list:

- **1** Create the `EPIMAPMap` object by using the `Screen` object.

  This constructor creates an instance of our `EPIMAPMap` class and validates the provided `Screen` object to check that it was created by the correct map. An `EPIMapException` is thrown if the screen does not match the map.

- **2** Get the data fields by name rather than by screen position.

  We then accessed each of the fields by calling the `field()` method on our map class, using the field names to indicate which field we wanted. The example shows how we were able to get each field without needing to know the order or position in which they were specified within the BMS map. Example 7-4 shows some sample output from the program.

Example 7-4 Output from Map class program

```
Time : 12:49:46
Date : 14/12/01
Applid : SCSCPAA6
```

7.2.2 Exception handling

A number of exception classes are supplied as part of the EPI support classes, and these should be used in your EPI application to handle potential error scenarios. Figure 7-7 shows the hierarchy of the supplied EPI exception classes.

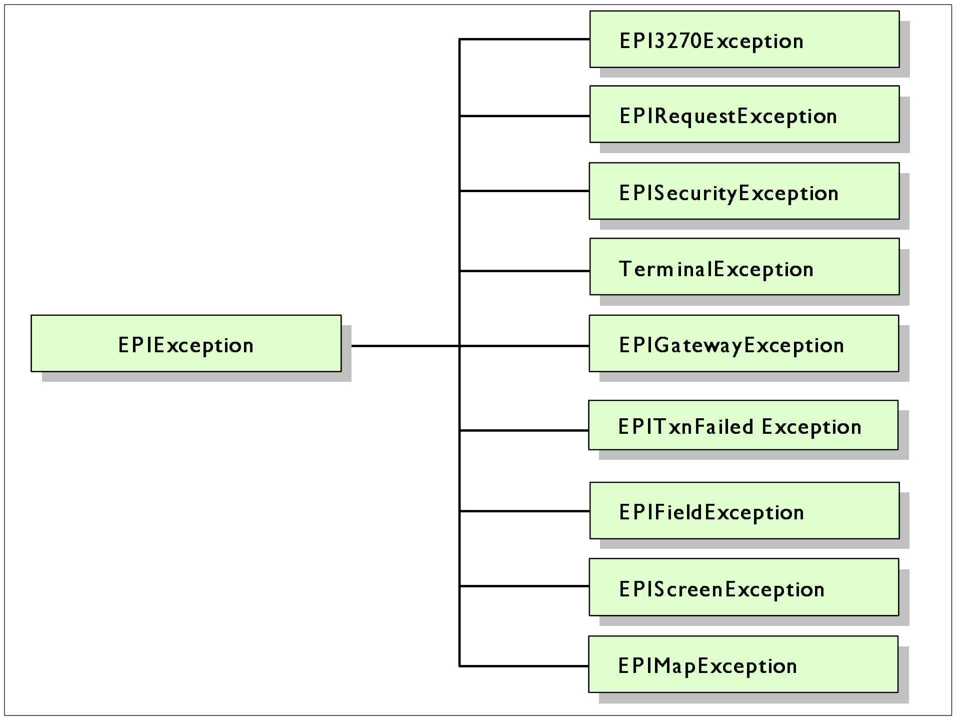


Figure 7-7 Hierarchy of the EPI exception classes.

Exception classes are provided that match most of the EPI support classes we have used, such as `Terminal`, `Screen`, `Field` and `Map`. These exceptions are thrown when an error occurs in a method that belongs to the corresponding support class. If you have multiple catch statements for a given try block then ensure that your catch blocks for the subclass exceptions come before any catch statements for a superclass exception. For example, you should code a catch statement for the `EPISecurityException` before a catch for `EPIException`. If you do not do this, you will receive errors at compile time because of the unreachable catch block.

Another commonly encountered exception when using the EPI support classes is a `java.io.IOException`. This exception is thrown whenever there is a problem opening a socket connection to the gateway daemon. Typically this will occur when first trying to open the gateway connection using the `open()` method of an `EPIGateway` or `JavaGateway` object.

Each of the exception classes provides a `getErrorCode()` method which returns an integer value that corresponds to a reason for the exception. You can use these values in your code to take a particular action based upon what the error code value was. You can also use the `getMessage()` method inherited from the `java.lang.Throwable` class to obtain a string that explains why the exception has occurred. Finally, a stack trace can be obtained from the exception which might be useful for revealing further information about what went wrong. In the following code we show how we used each of these methods to handle some commonly encountered error situations in our code:

```
try {
   EPIGateway epiGate = new EPIGateway("tcp://gunner", 2006);
1  Terminal term = new Terminal(epiGate, "SCSCPAA7", null, null);

2  term.setUserid("CICSRS2");
   term.setPassword("PASSWORD");

3  term.verifyPassword();

   term.disconnect();
   epiGate.close();

4 } catch (EPISecurityException epiSecEx) {
5  if (epiSecEx.getErrorCode() ==
      EPISecurityException.ESI_ERR_PASSWORD_INVALID) {
      System.out.println("ERROR -invalid password was supplied");
   } else System.out.println("Unknown security error encountered");

6 } catch (EPIException epiEx) {
   System.out.println("Unknown EPI error encountered");
7  System.out.println("Error message was :" + epiEx.getMessage());
8  epiEx.printStackTrace();

9 } catch (java.io.IOException ioEx) {
   System.out.println("Unable to connect to the gateway");
   System.out.println("Reason for error : " + ioEx.getMessage());
   ioEx.printStackTrace();
}
```

Figure 7-8 Using the EPI exception classes

The new code shown in Example 7-8, which is not in the previous examples is explained as follows:

- **1** Create the CICS terminal.

  Initially, we created a basic terminal object as in previous examples, but this time the server name was SCSCPAA7 instead of SCSCPAA6. Unlike SCSCPAA6, our SCSCPAA7 server was a CICS region with active RACF security. We needed to use a secure CICS region, because we wanted to call the `verifyPassword()` method of the `Terminal` object which does not work when the region has no security configured.

- **2** Set the security credentials associated with the `Terminal` object.

  By calling the `setUserid()` and `setPassword()` methods, our `Terminal` object was automatically converted from a basic, to an extended terminal, as these values are specific to extended terminals. See 7.3, "Connecting to secured CICS transactions" on page 174 for further information about security and extended terminals.

- **3** Verify the password for user ID CICSRS2.

  We called the `verifyPassword()` method to see if the password we supplied to the `setPassword()` method was valid for user ID CICSRS2. This method throws an exception if the password is invalid for the supplied user ID. The catch statements at the bottom of Figure 7-8 on page 170 handle any exception that could arise.

  Note: The `verifyPassword()` method uses the External Security Interface (ESI) which is described in 4.4, "ESI calls" on page 53.

- **4** Catch any `EPISecurityException`.

  We placed this catch block before the `EPIException` catch block because `EPISecurityException` is a subclass of `EPIException`. If the two catch blocks were the other way around then the `EPIException` catch statement would catch all `EPISecurityException` errors before they could be caught by the actual `EPISecurityException` catch statement. This would result in a compilation error of an *unreachable catch block*.

- **5** Determine what caused the `EPISecurityException`.

  Having caught an `EPISecurityException`, we used the `getErrorCode()` method to check what type of `EPISecurityException` had occurred. This method returns an integer value, which can be compared to one of the pre-defined error field values for the exception class. In our case, the ESI error of ESI_ERR_PASSWORD_INVALID was checked for because this indicates the password was invalid for user ID CICSRS2. Outputting the actual error code number in your error messages is not advisable because the meaning of the number could potentially be changed in a future release of the CTG. Instead, you should always compare the return code to one of the fields that are defined or inherited by the exception class, in the way we have shown here. If the error code is equivalent to ESI_ERR_PASSWORD_INVALID, then the password was invalid and therefore, you should print a specific informational error message to inform the application user about this error. Our code printed `Unknown security error encountered`, when the `EPISecurityException` was caused by some other error.

- **6** Catch any other `EPIException` errors.
- **7** Get the CTG error message.

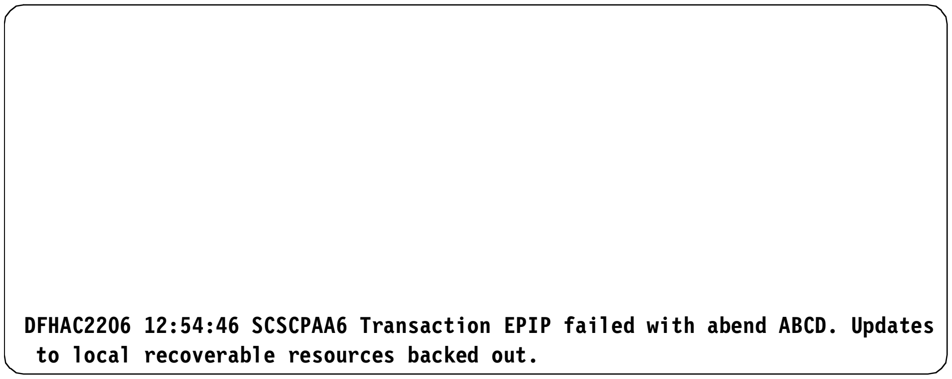
  By using the `getMessage()` method on the exception, we obtained an error code and explanatory message. The *CICS Transaction Gateway Messages* manual for your platform documents explanations for many of these error codes. We could also have used this method in the previous catch block because all of the EPI exception classes inherit from `java.lang.Throwable`, which provides the `getMessage()` method.
- **8** Print a stack trace.

  For this exception we also printed a stack trace by using the `printStackTrace()` method, also inherited from the `java.lang.Throwable` class. This can be helpful in debugging an error in the application, but is less likely to be of use to an end user.
- **9** Catch any errors communicating with the gateway.

  Any problems making a network connection to the gateway result in a `java.io.IOException`. We caught these types of errors in this final catch block. We used some of the same method calls that we had used in the other catch blocks to try to determine why we could not successfully connect.

CICS transaction abends

If a transaction abend occurs in CICS, then no specific exception will be thrown in your EPI application, and no EPI exception class exists for such an error. This is because a CICS transaction that was started using the EPI thinks it is interacting directly with a terminal. If an abend occurs in the transaction, then CICS will try to send an abend notification directly to the terminal screen rather than to the EPI application. An example of how this appears on a 3270 terminal is shown in Figure 7-9.



```
 DFHAC2206 12:54:46 SCSCPAA6 Transaction EPIP failed with abend ABCD. Updates
  to local recoverable resources backed out.
```

Figure 7-9 CICS transaction abend as displayed on a 3270 terminal

This means that a CICS transaction abend will manifest itself as some unexpected screen contents. To handle CICS abends from your EPI application you must be able to cope with the screen displaying this abend error message. You then have the option of retrieving the actual screen contents, which will include the abend message.

One way of coping with this scenario is to use the Map class. If you try to instantiate a `Map` object and the screen contents at the time have not been created by the correct BMS map, then an `EPIMapException` is thrown. You can use this feature to catch unexpected screen contents such as a transaction abend message, and you can then take the appropriate action.

7.3 Connecting to secured CICS transactions

For an EPI application to be able to start secured CICS transactions it must supply security credentials (a user ID and password) for the CICS server to authenticate. There are two options available to do this:

- **Sign on to the CICS terminal.**

  The security credentials determined at signon are then used for any subsequent authorization checks when starting other transactions. A user ID and password need not flow with further requests following the signon. This requires a *signon capable* EPI terminal.

- **Flow a user ID and password with each EPI request.**

  This does not require the user to sign on to CICS, and uses a *signon incapable* EPI terminal.

An EPI application should, therefore, choose between a signon capable or signon incapable terminal. This requires the use of an *extended terminal*. If using the default *basic terminal* there is no ability to set the signon capability, and instead, any security credentials must be hardcoded on each server connection definition in the CTG configuration file.

Server security configuration

The choice between signon capable and incapable terminals can have implications for the security configuration on the CICS server. When CICS installs an EPI terminal it uses the CTIN transaction to perform the operation (Figure 7-10). This transaction can be secured so that only certain user IDs have the authority to run it. Therefore, a transaction attach security check will be made at terminal install time to see if the user ID is authorized to run CTIN. This happens when the `connect()` method is invoked on the `Terminal` object.

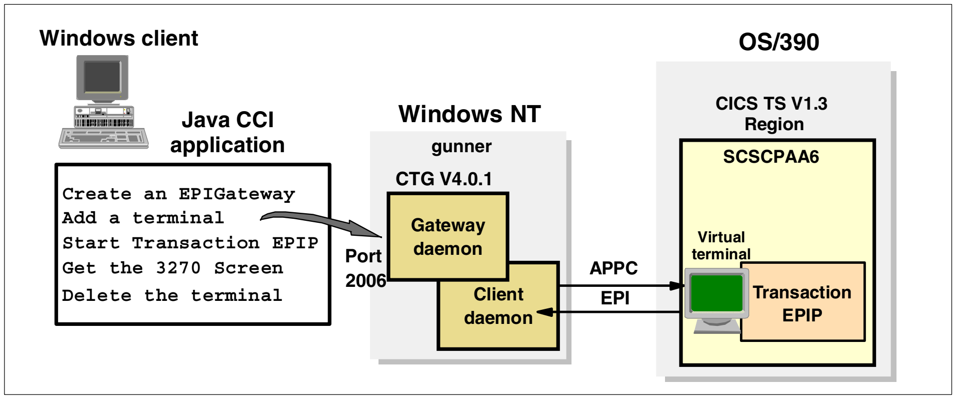


Figure 7-10 CTIN diagram

The behavior of this security check can be changed by modifying the USEDFLTUSER parameter on the CICS server CONNECTION definition. The value of the parameter can be either YES or NO and it determines whether the connection should use the CICS region default user ID for incoming requests that do not specify their own user ID and password.

Therefore, whenever a terminal install request is received with no user ID and password, this parameter is used to decide whether to use the CICS region default user ID to run CTIN. Typically, this user ID will be authorized to run CTIN. If you wish to protect your CICS region from unauthorized terminal install requests then we recommend that you set USEDFLTUSER to NO. When USEDFLTUSER is set to NO, any EPI application that attempts to install a terminal will have to provide a valid user ID and password for CTIN because the default user ID will not be used if they are omitted. You should use the `Terminal` object's `setUserid()` and `setPassword()` methods or set them using its constructor method to ensure that the required user ID and password flow to CTIN when the `connect()` method is called.

However, if USEDFLTUSER is set to NO and you are using *signon capable* terminals your application will be required to pass two security checks as follows:

1. Provide a user ID and password for the `Terminal` object's `connect()` method to successfully run CTIN.
2. Provide another user ID and password (which could be the same) to perform the actual signon.

If you want to implement an application that requires only a single signon then you can set USEDFLTUSER to YES to allow the terminal installation to happen without a security check. If you choose to do this, you should ensure that the default CICS region user ID has minimal access to CICS resources in order to protect your CICS region. If USEDFLTUSER is set to YES then set the user ID and password associated with the `Terminal` object to null before calling its `connect()` method.

7.3.1 Signon capable terminals

If the extended terminal resource is installed as signon capable then the EPI application is responsible for initiating a CICS signon transaction, such as the CICS supplied CESN transaction. The EPI application flows the user ID and password to the CICS signon transaction as 3270 data and the CICS program uses them to perform an EXEC CICS SIGNON. As an alternative to CESN you can use any transaction that will perform the signon command for you. Once your application has signed on, any subsequent calls made to CICS using the EPI will continue to run with the authority of the signed on user ID without the need to re-flow the user ID and password.

In our first security example, we create a *signon capable extended* terminal and ran the CICS supplied CESN transaction to perform a signon. We enable transaction security on our CICS server so that the EPIP transaction is secured and can be run by a user ID with sufficient authority. The code in Figure 7-11 shows how we used a *signon capable* terminal to first sign on to CICS using CESN and to subsequently be able to run the secured EPIP transaction.

```
try {
   EPIGateway epiGate = new EPIGateway("tcp://gunner", 2006);

1  Terminal term = new Terminal(epiGate, "SCSCPAA7", null, null,
      Terminal.EPI_SIGNON_CAPABLE, null, null, 0, null);
2  term.connect();
   Screen scr = term.getScreen();

3  scr.field(1).setText("CESN");
   scr.setAID(AID.enter);
   term.send();

4  scr.field(10).setText("CICSRS2");
   scr.field(16).setText("PASSWORD");
   term.send();

5  scr.setAID(AID.PF3);
   term.send();

6  scr.field(1).setText("EPIP");
   scr.setAID(AID.enter);
   term.send();

7  EPIMAPMap epimap = new EPIMAPMap(scr);
   System.out.println("Time : "+epimap.field(epimap.TIME).getText());
   System.out.println("Date : "+epimap.field(epimap.DATE).getText());
   System.out.println("Applid : "+epimap.field(epimap.APPLID).getText());

   term.disconnect();
   epiGate.close();
} catch (EPIException epiEx) { epiEx.printStackTrace();
} catch (java.io.IOException ioEx) { ioEx.printStackTrace();
}
```

Figure 7-11 Using a signon capable terminal to run CESN

The following list explains the code shown in Figure 7-11.

- 1 Create a signon capable extended `Terminal` object.

This particular constructor creates an extended terminal because it implicitly sets some of the attributes that are not available to basic terminals. We specified that the terminal was signon capable because we wanted to execute the signon transaction CESN. We did not provide a user ID and password to the constructor as we had set USEDFLTUSER to YES on the CICS server connection resource.

- **2** Install the terminal in CICS.

  Because we are using the constructor for an extended terminal we have to call the `connect()` method explicitly. The constructor does not call this for us, and therefore, allows an application the opportunity to do security checks before attempting to connect the terminal.

- **3** Start the CESN transaction.
- **4** Set the user ID and password fields and send this data to CESN.
- **5** End the CESN transaction by simulating a PF3 key press.
- **6** Run the secured EPIP transaction.

  Now that we had signed on, all of our subsequent transactions would be run with the authority of the user ID we used to perform the signon, in this case CICSRS2. We were now able to run the secured EPIP transaction as we had given the user ID CICSRS2 authority to run it when configuring the transaction security on the CICS server.

- **7** Get the results of the EPIP transaction.

  Using the same EPIMAPMap class that we created previously, we were able to obtain the output data from EPIP. If the EPIP transaction does not run successfully, you will see an *EPIMapException,* because the screen contents do not match those of a successful EPIP transaction.

Note: A signon capable terminal is required by any CICS program that issues an EXEC CICS SIGNON and its use is not limited to the CESN transaction. For example, our sample CICS program TRADER (described in Appendix C.5, “TRADER” on page 253) handles authentication of the user ID by issuing an EXEC CICS SIGNON after obtaining the user ID and password from its logon panel. In this scenario, you may want any user to be able to start the TRAD transaction and to be able to obtain its initial logon panel. Therefore, you would not protect the TRAD transaction and hence, would not need to run CESN before starting the TRAD transaction. However, you would still need to use a *signon capable terminal* object, because the TRADER application itself needs to be able to sign on the user.

7.3.2 Signon incapable terminals

If the extended terminal resource is installed as *signon incapable*, a user ID and password must be provided with each and every call to CICS made using the EPI, and a traditional signon is not possible. When using the EPI support classes and a signon incapable terminal, the user ID and password that flow with each request are the values that are associated with the `Terminal` object. These values can be changed by using the `setUserid()` and `setPassword()` methods of the `Terminal` object.

Our second security example shown in Figure 7-12 uses a signon incapable terminal to initiate the secured EPIP transaction. This is achieved by flowing a user ID and password to CICS with each EPI request. Note that when using a signon incapable terminal it is not possible to run a transaction that executes the EXEC CICS SIGNON command, including the CESN transaction.

```
try {
   EPIGateway epiGate = new EPIGateway("tcp://gunner", 2006);

1  Terminal term = new Terminal(epiGate, "SCSCPAA7", null, null,
      Terminal.EPI_SIGNON_INCAPABLE, "CICSRS1", "PASSWORD", 0, null);
   term.connect();
   Screen scr = term.getScreen();

2  term.setUserid("CICSRS2");
   term.setPassword("PASSWORD");

3  scr.field(1).setText("EPIP");
   scr.setAID(AID.enter);
   term.send();

   EPIMAPMap epimap = new EPIMAPMap(scr);
   System.out.println("Time : "+epimap.field(epimap.TIME).getText());
   System.out.println("Date : "+epimap.field(epimap.DATE).getText());
   System.out.println("Applid : "+epimap.field(epimap.APPLID).getText());

   term.disconnect();
   epiGate.close();
} catch (EPIException epiEx) { epiEx.printStackTrace();
} catch (java.io.IOException ioEx) { ioEx.printStackTrace();
}
```

Figure 7-12 Using a signon incapable terminal to run a secured transaction.

The following list explains the code shown in Figure 7-12

- 1 Create a signon incapable extended `Terminal` object.

This time we changed the constructor to create a signon incapable terminal by specifying the signon capability as `Terminal.EPI_SIGNON_INCAPABLE`. We also changed USEDFLTUSER value on the CICS connection definition to NO, so that a user ID and password were required for the terminal install request. Therefore, we added the user ID and password to the constructor to enable the terminal to be installed in CICS. The user ID that is specified (CICSRS1 in our example) should have the authority to run the CTIN client terminal installation transaction on CICS.

- **2** Set the user ID and password for the request to run transaction EPIP.

  Because our terminal was not capable of signing on, we had to pass the user ID and password every time we made an EPI request. When the `send()` method on the `Terminal` object is executed, the current user ID and password values associated with the `Terminal` object are flowed to CICS. We had configured our EPIP transaction so that only the user ID CICSRS2 had permission to run it and CICSRS1 was not authorized. Therefore, we had to update the security credentials by calling the `setUserid()` and `setPassword()` methods on the `Terminal` object.

- **3** Run the secured EPIP transaction.

  Now that the user ID and password have been changed, we started the EPIP transaction. When we called the `send()` method EPIP successfully ran using the updated user ID of CICSRS2.

8

CCI applications: EPI based

This chapter describes how to develop applications that use the CICS EPI resource adapter to connect from a Java environment to CICS transactions, which were originally created for a 3270 terminal. The CICS EPI resource adapter is shipped with the CICS Transaction Gateway (CTG), and acts as the middle tier between a Java application and CICS. The CICS EPI resource adapter adheres to the J2EE Connector Architecture 1.0 specification.

Java applications interface with the CICS EPI resource adapter using the Common Client Interface (CCI). This chapter focuses on:

- Using the CCI API directly to develop applications
- Using the Enterprise Access Builder to develop applications that internally use the CCI API.
- Connecting to secured CICS transactions, relating to both application development methods.

This chapter focuses on writing Java applications that will be deployed in a non-managed environment; this is where an application server is not managing the connection to the resource adapter. For information on managed environments, see Chapter 6, “CCI applications in a managed environment” on page 111.

8.1 Using the CCI

The Common Client Interface (CCI) defines an application programming interface (API) which provides a standard way to communicate with Enterprise Information Systems (EISs) through their specific resource adapters. The CICS EPI resource adapter facilitates communication with CICS transactions. The CCI concepts are described in more detail in “Common Client Interface” on page 21.

This section focuses on using the CCI interfaces directly. Alternatively, tooling such as VisualAge for Java can generate CCI code, and this is discussed in “Using the Enterprise Access Builder” on page 193.

Figure 8-2 shows the classes used to execute a transaction and their relationship to the CCI. Note how the methods used to drive an interaction are inherited by these classes from the interfaces provided by the CCI.

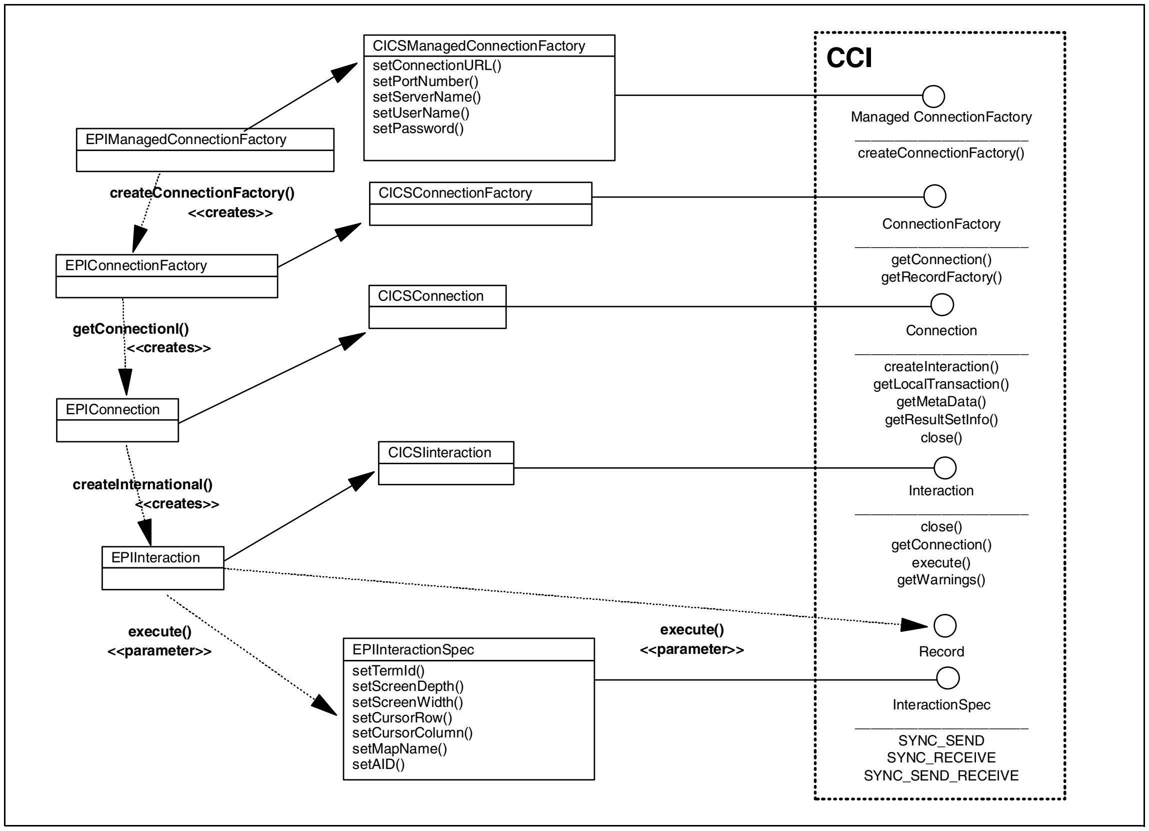


Figure 8-1 CICS EPI resource adapter class diagram

8.1.1 Writing a simple CCI application

The CCI API is typically used with the CICS EPI resource adapter to start a transaction in CICS, and retrieve the fields generated from the CICS 3270 screen using a Record. See Figure 8-2.

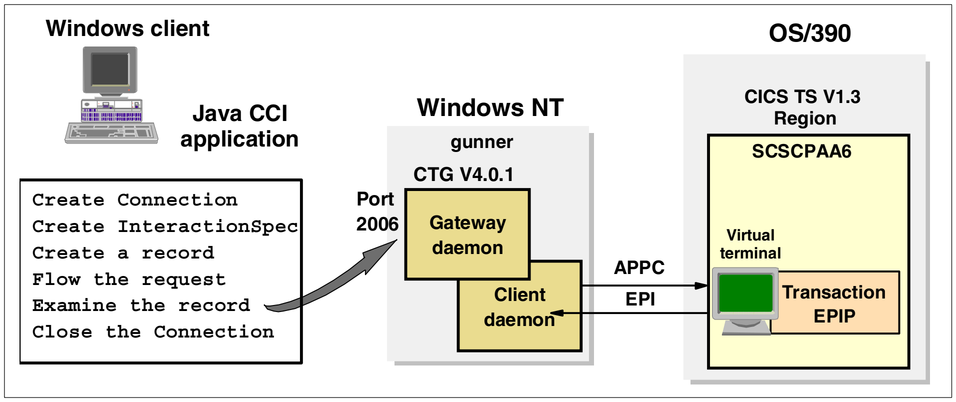


Figure 8-2 CTG scenario: EPI CCI application

We have written a Java application that starts transaction EPIP in CICS to demonstrate the use of the CCI API. EPIP is a simple transaction that runs the CICS program EPIPROG. EPIPROG retrieves the date, time, and CICS server that it is running on, and displays these values as fields on a 3270 screen. Our Java application will display these values to standard output. The source code for EPIPROG is in Appendix C, “Sample CICS programs” on page 241).

The class `itso.cics.epi.j2ee.RunEPIP` contains a completed solution of this Java application. To obtain it see Appendix D, “Additional material” on page 261.

Import statements

Our program uses classes from two Java packages:

javax.resource.cci	A collection of CCI interfaces, described in the J2EE Connector Architecture specification.
com.ibm.connector2.cics	Classes specific to the CICS resource adapters, which implement the CCI interfaces

To use classes from these packages, you must either explicitly reference their package names in the code or, more conveniently, use an import statement. Figure 8-3 shows the import statements we added to our Java application.

```
import javax.resource.ResourceException;
import javax.resource.cci.Connection;
import javax.resource.cci.ConnectionFactory;
import javax.resource.cci.Interaction;
import com.ibm.connector2.cics.EPIManagedConnectionFactory;
import com.ibm.connector2.cics.EPIInteractionSpec;
import com.ibm.connector2.cics.EPIScreenRecord;
import com.ibm.connector2.cics.EPIScreenRecordImpl;
import com.ibm.connector2.cics.EPIFieldRecord;
import com.ibm.connector2.cics.AIDKey;
```

Figure 8-3 Import statements relating to an EPI CCI application

Connecting to a CICS BMS transaction

Figure 8-4 shows the use of the CCI to start the transaction EPIP in CICS, using the EPI resource adapter. It describes an extremely basic interaction with CICS, which can be used as a starting point for developing your own CCI applications.

```
  try{
     //create and set values for a managed connection factory for EPI
1    EPIManagedConnectionFactory mcf = new EPIManagedConnectionFactory();
     mcf.setConnectionURL("tcp://gunner.almaden.ibm.com");
     mcf.setServerName("SCSCPAA6");

     //create a connection object
2    ConnectionFactory cxf=(ConnectionFactory)mcf.createConnectionFactory();
     Connection connection = cxf.getConnection();

     //create an interaction with CICS to start transaction EPIP
3    Interaction interaction = connection.createInteraction();
     EPIInteractionSpec iSpec = new EPIInteractionSpec();
     iSpec.setAID(AIDKey.enter);
     iSpec.setFunctionName("EPIP");

     //create a record to store the response from CICS
4    EPIScreenRecord screen = new EPIScreenRecordImpl();

     //flow the request to CICS
5    boolean rc = interaction.execute(iSpec, null, screen);

     //close the interaction and connection
6    interaction.close();
     connection.close();
7 } catch (ResourceException re) { System.out.println(re.getMessage()); }
```

Figure 8-4 Using the CCI to connect to CICS

The following list summarizes the logic in Example 8-4:

- **1** Create and set values for an EPI managed connection factory.

  The `EPIManagedConnectionFactory` class is used to set EPI specific connection related information, including:

  – The protocol and hostname of the CTG, through the `setConnectionURL()` method. If no protocol is specified, `tcp` is assumed.

  – The port number to use when connecting to the CTG, through the `setPortNumber()` method. If no value is specified port `2006` is assumed.

  – The CICS server to connect to, through the `setServerName()` method.

  – Security related parameters, such as the `setUserName()`, `setPassword()`, and `setLogonLogoffClass()` methods.

  – The device type of the terminal to use, through the `setDeviceType()` method.

  The `EPIManagedConnectionFactory` class is used regardless of whether the Java application will be deployed in a managed or non-managed environment.

- **2** Create a connection object.

  The information supplied in the managed connection factory is used to create a connection object. First, invoke the method `createConnectionFactory()` on the `EPIManagedConnectionFactory` object to create a `ConnectionFactory` object. Then use this `ConnectionFactory` object to create a `Connection` object.

  Important: The J2EE Connector Architecture states that the `ConnectionFactory` object should be retrieved by making a JNDI lookup, rather than explicitly creating it in the code. We have not used JNDI here for simplicity. For information on how to incorporate JNDI to lookup a `ConnectionFactory` object see 5.5, "Using JNDI" on page 100.

- **3** Create an interaction with CICS to start transaction EPIP.

  Once the `Connection` object has been established, an `Interaction` object must be generated. The `Interaction` class provides methods to flow requests to an Enterprise Information System (EIS); in this case CICS. An `EPIInteractionSpec` object is instantiated and set, detailing what the interaction to CICS should do. In this example we perform the equivalent action of a user at a CICS terminal entering the transaction name EPIP and clicking Enter.

- **4** Create a Record to store the response from CICS.

When the CICS EPI resource adapter connects to CICS it will create a *virtual terminal*. A virtual terminal is a terminal with no output device to display to. The virtual terminal is used to communicate with CICS, and when a transaction is started, the output from this transaction will be written to a screen in the virtual terminal.

The fields that make up a virtual terminal can be retrieved in an output Record. The `EPIScreenRecord` class will be used to hold this information.

- **5** Flow the request to CICS.

The `Interaction` object has a method called `execute()` which is used to flow the request to CICS. This creates a connection to the CTG. The `execute()` method used here takes three parameters:

InteractionSpec	The interaction specification. For EPI pass an `EPIInteractionSpec` object.
Record	The input Record describing the fields to send to CICS. When starting a new transaction, specify this value as `null`.
Record	The output Record where the screen fields returned by this interaction will be stored.

The `execute()` method returns a Boolean to indicate the success of the execution.

Note: A return value of `true` from the `execute()` method does not necessarily indicate a successful execution. The occurrence of CICS generated errors, such as the transaction name not being recognized, does not influence the return value.

- **6** Close the interaction and connection.

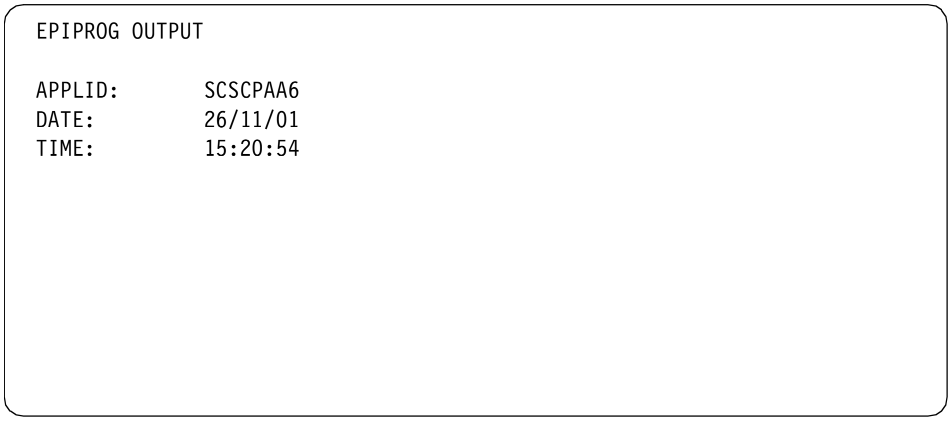
Further interactions are made to CICS by re-using the `Interaction` object. Do not attempt to create a new `Interaction` object because you can only have one `Interaction` per `Connection`. When there are no further interactions to make, both the `Interaction` and `Connection` objects should be closed. This frees up resources held for this `Interaction` in the resource adapter, and closes the connection handle. Note that in a non-managed environment, closing the connection handle will close the connection to the CTG. In a managed environment the connection to the CTG will be returned to the connection pool.

- **7** Catch `ResourceException`

Use the `getMessage()` method of `ResourceException` to retrieve an error code and explanatory message. The *CICS Transaction Gateway Programming* manual for your platform documents provides explanations for these error codes in Appendix C.

Examining the output Record

Upon a successful interaction with CICS, the output Record will contain the fields returned in the 3270 screen. EPIP generates the 3270 screen shown in Figure 8-5



```
EPIPROG OUTPUT

APPLID:       SCSCPAA6
DATE:         26/11/01
TIME:         15:20:54
```

Figure 8-5 EPIP 3270 output

The `EPIScreenRecord` class has a number of useful methods for working with the output Record, including:

getFieldCount()	Returns an `int` containing the number of fields in the output Record.
getField(int index)	Specify a field to retrieve. Returns this field in an `EPIFieldRecord` object.
getFields()	Returns all fields in a `java.util.Iterator` object.

Figure 8-6 retrieves all of the fields returned in the output Record and displays the position and text value of each one. Note there is no way to retrieve fields by name.

```
EPIFieldRecord field;
for (int i=1; i <= screen.getFieldCount(); i++) {
   field = screen.getField(i);
   System.out.println("Pos: " + field.getTextPos() + "=" + field.getText());
}
```

Figure 8-6 Retrieving fields from the output Record

The `EPIFieldRecord` class contains many methods to retrieve information about individual fields including:

getText()	The text value of the field.
getTextPos()	The current position of the text, as if it were displayed on a 3270 screen.
getTextCol()	The column where the text would be displayed on a 3270 screen.
getTextRow()	The row where the text would be displayed on a 3270 screen.
isProtected()	Returns a Boolean stating if the field is protected

Example 8-1 shows the output from running this program.

Example 8-1 Fields from the output Record

```
Pos: 1=EPIPROG OUTPUT
Pos: 161=APPLID:
Pos: 175=SCSCPAA6
Pos: 241=DATE:
Pos: 255=26/11/01
Pos: 321=TIME:
Pos: 335=15:20:54
```

8.1.2 Extending a CCI application

The CCI provides much greater functionality than simply starting a CICS transaction then examining the result. This section discusses extending the functionality of your applications in the following areas:

- Synchronous and asynchronous call support
- Making pseudo-conversation calls to CICS
- Options available for working with input and output Records
- CICS transaction abends

Synchronous calls

The EPI resource adapter only supports synchronous calls to CICS. Asynchronous calls are not supported. The `setInteractionVerb()` method of `EPIInteractionSpec` specifies the type of synchronous call to use:

SYNC_SEND	Sends a call to CICS, but does not return an output Record. However, the call blocks until the transaction has sent all of the information that would appear on a screen.
SYNC_RECEIVE	Retrieves the current contents of the screen in the output Record
SYNC_SEND_RECEIVE	Sends a call to CICS, and returns the response in the output Record. This is the default.

We recommend using SYNC_SEND_RECEIVE.

Pseudo-conversational transactions

Starting a CICS transaction, then retrieving the generated screen, is a somewhat limited way of interacting with CICS. Instead, after starting a transaction, you may want to modify the returned screen and return it to CICS in a pseudo-conversation. This is a more likely scenario.

Figure 8-7 demonstrates an example of a pseudo-conversational interaction with CICS. It calls the SWAP transaction, which generates a screen with two unprotected fields; fields 4 and 6. Values are entered for these fields, then the updated screen is sent back to CICS. The resultant generated screen swaps the value of field 4 with the value of field 6, and vice versa. See Appendix C, “Sample CICS programs” on page 241.

The class `itso.cics.epi.j2ee.RunSWAP` contains a completed solution of this program. To obtain it see Appendix D, “Additional material” on page 261.

```
  //create an interaction with CICS to start transaction SWAP
  Interaction interaction = connection.createInteraction();
  EPIInteractionSpec iSpec = new EPIInteractionSpec();
  iSpec.setAID(AIDKey.enter);
  iSpec.setFunctionName("SWAP");

  //create a record to store the response from CICS
  EPIScreenRecord screen = new EPIScreenRecordImpl();

  //flow the request to CICS
1 interaction.execute(iSpec, null, screen);

  //set values for fields 4 and 6
  String value1 = "ABCDE", value2 = "12345";
2 EPIFieldRecord firstValue = screen.getField(4);
  firstValue.setText(value1);
  EPIFieldRecord secondValue = screen.getField(6);
  secondValue.setText(value2);

  //flow another request to CICS
3 interaction.execute(iSpec, screen, screen);

  //display the new values of fields 4 and 6
4 System.out.println("Value 1: " + value1 + " has become: " +
                     screen.getField(4).getText());
  System.out.println("Value 2: " + value2 + " has become: " +
                     screen.getField(6).getText());

  //flow a request to exit from SWAP
5 iSpec.setAID(AIDKey.PF3);
  interaction.execute(iSpec, screen, screen);
```

Figure 8-7 Making multiple interactions to CICS

The following list summarizes the logic in Figure 8-7:

- **1** Start the SWAP transaction in CICS.

  No input Record is required when starting a CICS transaction, but an output Record is needed to store the screen returned by the transaction.

- **2** Set values for fields 4 and 6.

  The `screen` object (the output Record) contains all of the fields from the screen returned by the SWAP transaction. Fields 4 and 6 are input (unprotected) fields. Set values for these fields using the `setText()` method of `EPIFieldRecord`.

- **3** Flow another request to CICS.

Send the updated `screen` object as the input Record in the `execute()` method. This will send the screen back to CICS with fields 4 and 6 updated, and run SWAP again. The resultant screen will be returned in the output Record.

- **4** Display the new values of fields 4 and 6

  Use the output Record to confirm the fields have swapped. The results are shown in Example 8-2.

- **5** Flow a request to exit from SWAP

  To exit from the SWAP transaction, the user presses the PF3 key. This AID key is set, then flowed to CICS. The input Record remains unchanged because it will not be used by the CICS application. The output Record will contain the response from CICS. A successful response from SWAP is a field saying `Session ended`, which could be used to check for a successful execution.

Note: When interacting with a started CICS transaction, the `execute()` method must specify both an input and output Record.

Example 8-2 Results from RunSWAP

```
Value 1: ABCDE has become: 12345
Value 2: 12345 has become: ABCDE
```

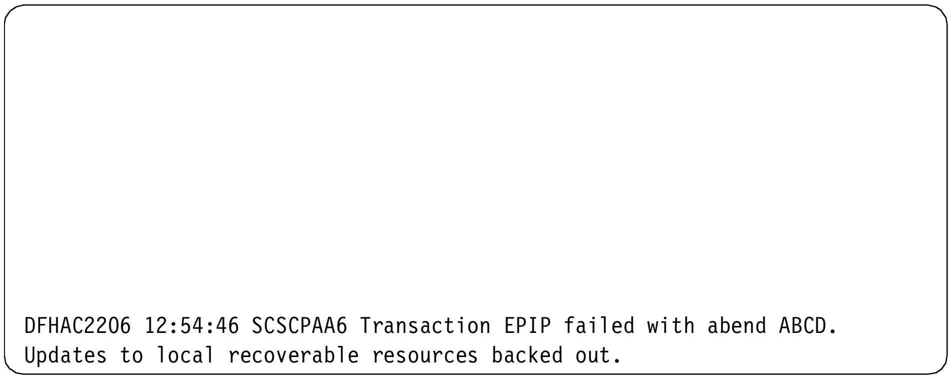
Input and output Records

So far we have used the `EPIScreenRecord` class to work with input and output Records. However, this is only one of three ways of working with Records:

- Write your own custom Record classes. Your classes are responsible for parsing the input and output stream manually. These classes must use either the `Streamable` interface, or the more efficient screenable interfaces. This is a complex option, and we do not recommend taking it.
- Use `EPIScreenRecord` to retrieve output, and modify input of existing screens. It is an implementation of the screenable interfaces. However, it is not a custom screen Record builder. It cannot be used to build your own screen Records from scratch. We have used this option in the above examples.
- Use development tooling to build new Records, and modify the contents of existing ones. VisualAge for Java provides a set of SmartGuides that can be used to produce Record classes, either from scratch or basic mapping support (BMS) source code. This is described in “Building the output Record” on page 195.

CICS transaction abends

If a transaction abend occurs in CICS then no specific exception will be thrown in your application, and no exception class exists for such an error. This is because a CICS transaction that was started using the EPI thinks it is interacting directly with a terminal. If an abend occurs in the transaction, then CICS will try to send an abend notification directly to the terminal screen rather than to your application. An example of how this appears on a 3270 terminal is shown in Figure 8-8.



```
DFHAC2206 12:54:46 SCSCPAA6 Transaction EPIP failed with abend ABCD.
Updates to local recoverable resources backed out.
```

Figure 8-8 CICS transaction abend as displayed on a 3270 terminal

This means that a CICS transaction abend will manifest itself as some unexpected screen contents. To handle CICS abends from your Java application you must check the output Record for these messages manually.

8.1.3 Tracing

Tracing is set using the `EPIManagedConnectionFactory` class. Two actions must be taken to enable tracing:

1. Turn logging on through the `setLogWriter()` method.
2. Specify the level of tracing to use through the `setTraceLevel()` method. Valid levels are tracing are:

RAS_TRACE_OFF	Disable all tracing (except adapter messages)
RAS_TRACE_ERROR_EXCEPTION	Output exception trace stacks
RAS_TRACE_ENTRY_EXIT	Output method entry and exit stack traces
RAS_TRACE_INTERNAL	Output debug trace entries

Each level of trace builds upon the previous level, so ENTRY_EXIT includes everything in ERROR_EXCEPTION, and INTERNAL includes everything in all the trace levels. The code below shows INTERNAL tracing being sent to the stdout destination using `System.out`.

```
EPIManagedConnectionFactory mcf = new EPIManagedConnectionFactory();
mcf.setLogWriter(new java.io.PrintWriter(System.out));
mcf.setTraceLevel(new Integer(mcf.RAS_TRACE_INTERNAL));
```

Attention: We found that the CCI trace output was not particularly useful and only showed the execution of methods within the CCI. If you wish to debug your application you are advised to use the `com.ibm.ctg.client.T` class which provides much more useful information. For details on using the `T` class refer to 4.6, “Tracing” on page 61.

8.2 Using the Enterprise Access Builder

The Enterprise Access Builder (EAB) is a tool provided by VisualAge for Java Enterprise Edition. It provides a set of SmartGuides that simplify communication with EISs such as CICS.

EAB generates code based on properties set in the SmartGuides. Prior to VisualAge for Java V4, the EAB generated code conformed to the IBM Common Connector Framework (CCF). Version 4 introduced support for the J2EE Connector Architecture, in addition to the CCF support. The support for the J2EE Connector Architecture generates code that uses the CCI API. To enable this support in VisualAge for Java V4, see Appendix A, “Configuring the CICS connectors in VisualAge for Java” on page 219.

This section concentrates on using EAB with the J2EE Connector Architecture. It is divided into two sections:

- **Writing a simple EAB application**

  This section describes how to build applications that start CICS transactions, and interrogate the responses returned by these CICS transactions.

- **Extending an EAB application**

  This section describes how to build applications that make pseudo-conversational interactions with CICS transactions.

8.2.1 Writing a simple EAB application

This section describes how to write a program to start a transaction in CICS, and uses the EAB to handle the response. This method is an alternative to using the CCI directly, as described in 8.1.1, “Writing a simple CCI application” on page 183. We have used the CICS transaction EPIP as an example in both sections to ease comparison between these two techniques.

EPIP is a simple transaction that runs the CICS program EPIPROG. EPIPROG retrieves the date, time, and CICS server name it is running on, and displays these values to the screen (see Appendix C, “Sample CICS programs” on page 241).

Two EAB components are required to create a simple interaction with a CICS:

- **Record beans**

  A Record bean maps to the content of a CICS 3270 screen. There are two types of Record bean; input and output. An input Record represents screen content to send to a CICS transaction, and an output Record represents the screen content generated by a CICS transaction.

  Many CICS screens are defined by basic mapping support (BMS). The EAB can take a BMS map and use it to generate a Record bean that implements the `javax.resource.cci.Record` interface (as required by the J2EE Connector Architecture). The generated Record bean will contain getter and setter methods to interact with the screen fields encompassed by the Record.

- **Command beans**

  A Command bean is a Java bean that contains everything necessary to interact with CICS. It contains the following:

  – Connection information, including the URL of the resource adapter, and the name of the CICS server to use.

  – Interaction information, including the CICS transaction to start.

  – The input and output Record beans to use.

  An application that wants to interact with CICS can use a Command bean to do so. The Command bean exposes the getter and setter methods of the input and output Record beans, and the `execute()` method that performs the interaction with CICS. The application programmer needs to know very little about CICS. They only need to know the name of the getter and setter methods, and that the `execute()` method causes the interaction with CICS. All of the other details of the interaction with CICS are handled by the Command bean.

The package `itso.cics.epi.j2ee` contains the classes that make up the completed solution. To obtain it see Appendix D, “Additional material” on page 261. The key classes used are:

EPIPMAPRecord	The output Record
EPIPCommand	The Command bean
EPIPClient	The client which uses the Command bean

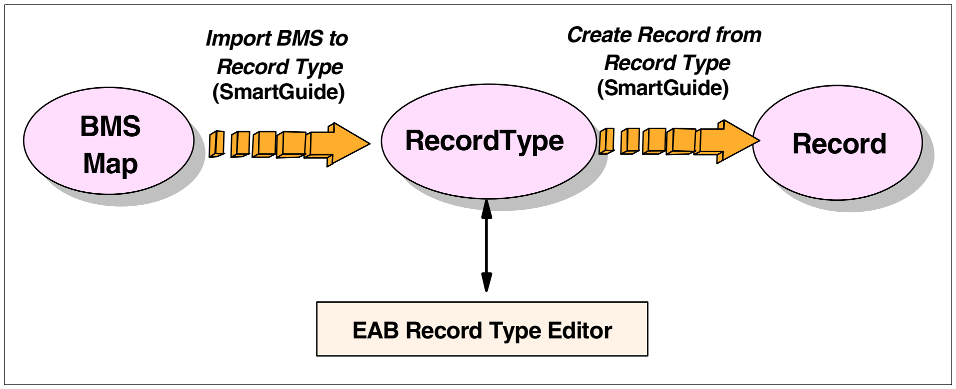
This section describes how to use the EAB to create an application that interacts with the CICS transaction EPIP. It consists of the following stages, which must be completed in order:

- Building the output Record
- Building the Command bean
- Testing the Command bean
- Writing a client to use the Command bean

Building the output Record

An output Record is required to store the content of the screen generated by the EPIP transaction. No input Record is needed because you are not required to specify an input Record when starting a CICS transaction.

This Record can be built from scratch, but if the screen you wish to map has a basic mapping support (BMS) map associated with it, you can instead use this BMS map to automatically generate a Record (Figure 8-9).



BMS Map
Import BMS to Record Type (SmartGuide)
RecordType
Create Record from Record Type (SmartGuide)
Record
EAB Record Type Editor

Figure 8-9 Creating a Record from a BMS map

The following steps describe how to build an output Record using this approach.

1. In VisualAge for Java create a project called *CICS Connectors Redbook*, and a package within this project called `itso.cics.epi.j2ee`.

2. The BMS map source code to build the output Record is contained in the file `epimaps.bms`. (Details of how to obtain it are in Appendix D, “Additional material” on page 261).
3. Right click package `itso.cics.epi.j2ee` and select **Tools -> Enterprise Access Builder -> Import BMS to Record Type**. This will open the *Import BMS to Record Type* SmartGuide.
   – Select the **Add** button, highlight the `epimaps.bms` file from the appropriate directory, then select **Open**.
   – The filename should display in the *Filenames* list. Select **Next**.
4. EPIMAP should appear in the *Available Maps* pane. Select it, then click the **>** button. EPIMAP should now appear in the *Selected Maps* pane as shown in Figure 8-10. Select the **Next** button.

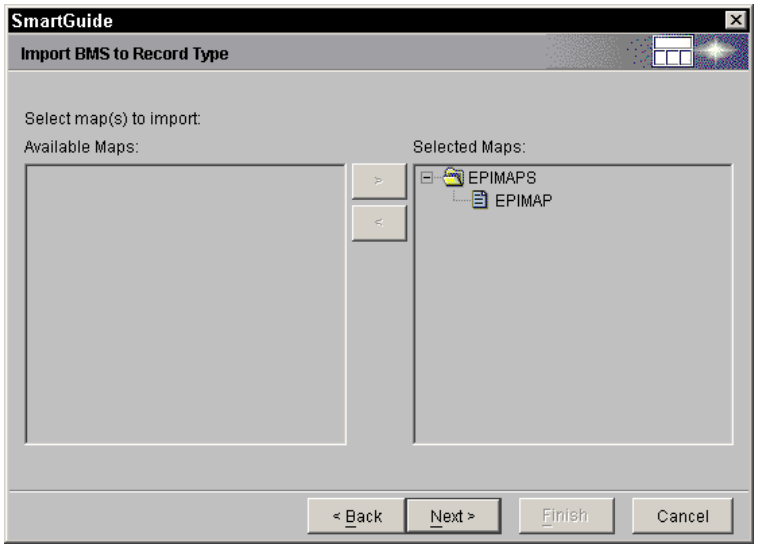
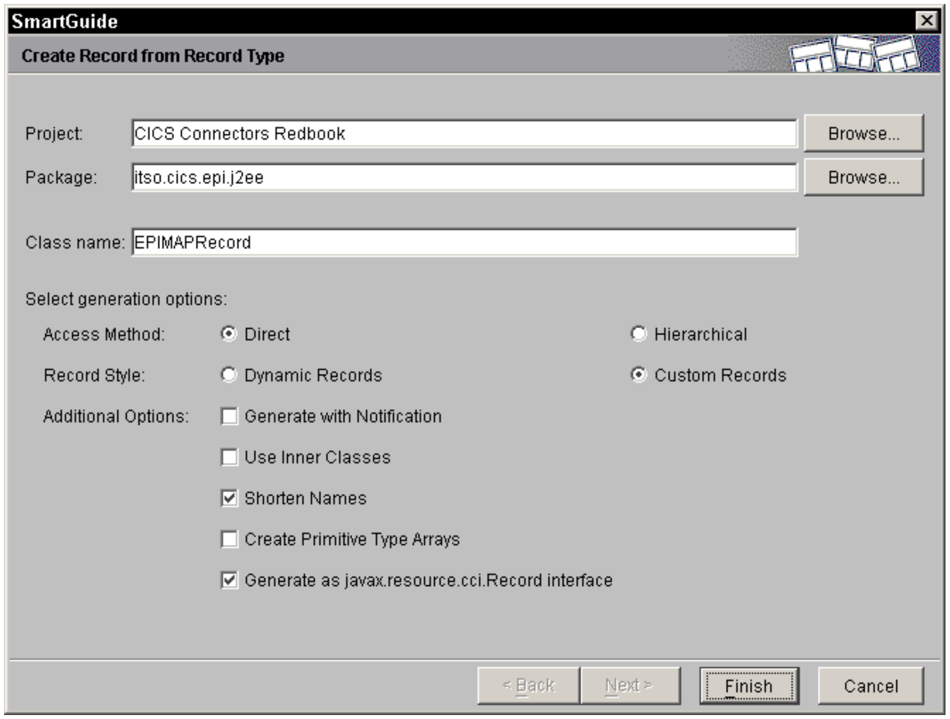


Figure 8-10 Import BMS to Record Type SmartGuide

5. Make sure the project is *CICS Connectors Redbook*, and the package is `itso.cics.epi.j2ee`. Enter a class name of **EPIMAPRecordType**. Ensure **Continue working with newly created record type** and **Create record from record type** are selected. Click **Finish**.
6. The *Create Record from Record Type* SmartGuide will appear. Enter a class name of *EPIMAPRecord.* Perform the following:
   – Select **Access Method** to be *Direct,* and the *Record Style* to use *Custom Records*. This generates the fastest type of Record bean.

- In the *Additional Options* section, ensure *Generate with Notification* is *not* selected. Selecting this option can significantly slow the performance of the Record bean.
- Select the following from the *Additional Options* section:
  - **Shorten Names** (this generates names that are more readable)
  - **Generate as javax.resource.cci.Record interface** (this is an interface the Record bean must implement to be J2EE compliant)

  Figure 8-11 shows a completed SmartGuide. Upon completing this SmartGuide select **Finish**. This will generate a Record bean with getter and setter methods for each field in the BMS map.


SmartGuide

Create Record from Record Type

Project: CICS Connectors Redbook

Package: itso.cics.epi.j2ee

Class name: EPIMAPRecord

Select generation options:

Access Method: [✓ Direct] Hierarchical

Record Style: Dynamic Records [✓ Custom Records]

Additional Options:
- ☐ Generate with Notification
- ☐ Use Inner Classes
- ☑ Shorten Names
- ☐ Create Primitive Type Arrays
- ☑ Generate as javax.resource.cci.Record interface

Figure 8-11 Create Record from Record Type SmartGuide

Building the Command bean

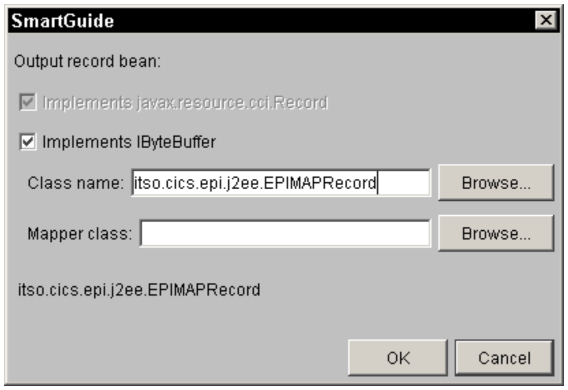
The following steps describe how to build a Command bean which can interact with the CICS transaction EPIP.

1. Right click on the `itso.cics.epi.j2ee` package and select **Tools -> Enterprise Access Builder -> Create Command**. This will open the *Create Command* SmartGuide. Enter the following information:

  - Set the class name to *EPIPCommand*.
  - Select **Edit when finished**.
  - Click **Browse** next to the *Class name* field for *Connection Information*. Select **ConnectionFactoryConfiguration** and click **OK**. This indicates to the SmartGuide that the generated Command bean will adhere to the J2EE Connector Architecture specification.
  - Click **Browse** next to the *Class name* field for *InteractionSpec*. Select **EPIInteractionSpec** and click **OK**. Notice only J2EE Connector Architecture classes (those in the `com.ibm.connector2.cics` package) are available for selection.

    After entering this information click **Next**.

2. Enter information about the input and output Records to use.
  - You do not need an input Record bean because you will be starting a new CICS transaction. Leave the fields in the *Input Record bean* section unchanged.
  - The output Record will store the contents of the screen returned by the EPIP transaction. In the *Output Record beans* section, select **Select output bean records**, then click the **Add** button.
  - A SmartGuide window opens (Figure 8-12). Notice *Implements javax.resource.cci.Record* is selected and greyed out so that it cannot be changed. The Record bean we created earlier adheres to this interface.
    - Select **Browse** next to the *Class name* field.
    - Select **EPIMAPRecord** from package `itso.cics.epi.j2ee` and click **OK**.
    - Click **OK** again, then finally click **Finish** to build the Command bean.


SmartGuide

Output record bean:

[✓] Implements javax.resource.cci.Record

[✓] Implements IByteBuffer

Class name: itso.cics.epi.j2ee.EPIMAPRecord

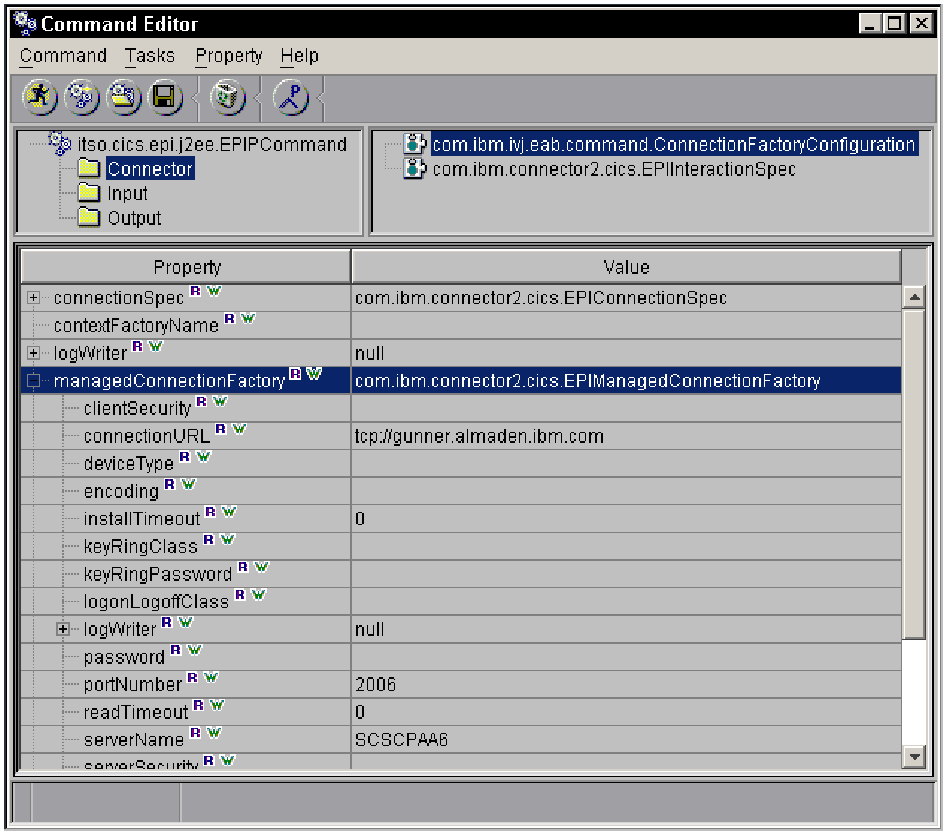
Mapper class:

itso.cics.epi.j2ee.EPIMAPRecord

Figure 8-12 SmartGuide to select the output Record to use

3. The Command bean will be generated, then the *Command Editor* window will appear. This tool lets you set properties in the Command bean. First set the connection factory fields. Highlight **Connector** in the top left pane, then select **com.ibm.ivj.eab.command.ConnectionFactoryConfiguration** in the top right frame. This will display properties and values relevant to the connection to CICS (Figure 8-13).

.


Command Editor
Command Tasks Property Help
itso.cics.epi.j2ee.EPIPCommand
Connector
Input
Output
com.ibm.ivj.eab.command.ConnectionFactoryConfiguration
com.ibm.connector2.cics.EPIInteractionSpec

Property	Value
connectionSpec R W	com.ibm.connector2.cics.EPIConnectionSpec
contextFactoryName R W	
logWriter R W	null
managedConnectionFactory R W	com.ibm.connector2.cics.EPIManagedConnectionFactory
clientSecurity R W	
connectionURL R W	tcp://gunner.almaden.ibm.com
deviceType R W	
encoding R W	
installTimeout R W	0
keyRingClass R W	
keyRingPassword R W	
logonLogoffClass R W	
logWriter R W	null
password R W	
portNumber R W	2006
readTimeout R W	0
serverName R W	SCSCPAA6
serverSecurity R W	

Figure 8-13 Command Editor

4. Perform the following:

   – Select the `null` value for the *managedConnectionFactory* property. This presents a pull-down menu of valid managed connection factories. Select **EPIManagedConnectionFactory**. The `EPIManagedConnectionFactory` class is used regardless of whether the Command bean will be deployed in a managed or non-managed environment.

- To the left of the *managedConnectionFactory* property click on **+** to expand this property. Numerous sub-properties of *managedConnectionFactory* are displayed. Set the following:
  - Set the *connectionURL* to configure the protocol and hostname for the gateway daemon. We used `tcp://gunner.almaden.ibm.com` for our CTG on Windows NT. If the protocol is omitted, `tcp` is assumed.
  - Note the *portNumber* is by default `2006`. Change this if necessary.
  - Change the *serverName* to the CICS server name where EPIP will start. We used *SCSCPAA6*.

Important: The J2EE Connector Architecture states that the `ConnectionFactory` object should be retrieved by making a JNDI lookup, rather than explicitly creating it in the code. We have not used JNDI here for simplicity. For information on how to incorporate JNDI to lookup a `ConnectionFactory` object see 5.5, "Using JNDI" on page 100.

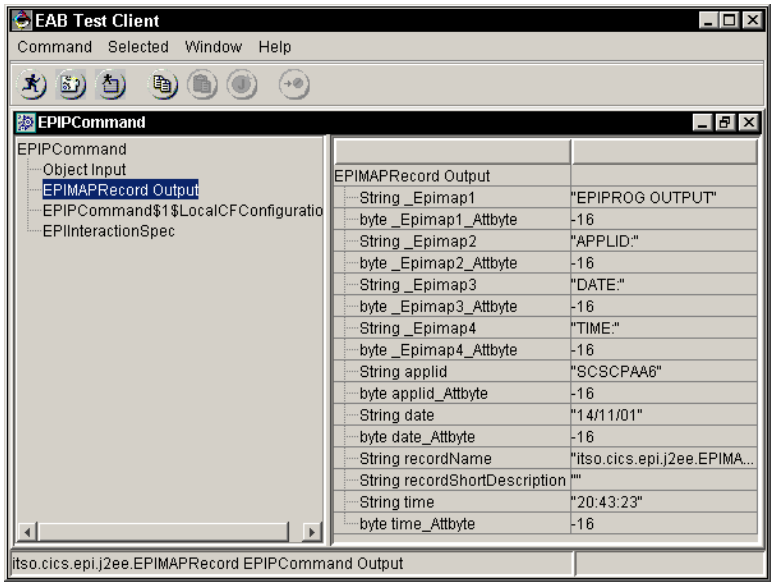
5. Highlight **com.ibm.connector2.cics.EPIInteractionSpec** in the top right pane. These properties set what interaction should take place once connected to CICS.
   - Set the *functionName* property to *EPIP*. This specifies the name of the transaction to start in CICS.
   - Ensure *AID* is set to *enter* which performs the equivalent of pressing Enter on the CICS terminal.
6. The Command bean is now complete. To save the values set using this editor select **Command -> Save**.

Testing the Command bean

VisualAge for Java provides a utility to test a Command bean. We recommend that you use this utility before attempting to integrate Command beans into your applications. Follow the steps below to test EPIPCommand.

1. The *EAB Test Client* can be launched in two ways:
   - If you still have the *Command Editor* open select **Command -> Run Test Client**.
   - Alternatively select **Workspace -> Tools -> Enterprise Access Builder -> Launch Test Client**.
2. Once the *EAB Test Client* has started select **Command -> Create new instance**. Choose class *EPIPCommand* from the `itso.cics.epi.j2ee` package then click **OK**. This will create an instance of the Command bean.

3. Highlight **EPIMAPRecord Output** in the left pane. This displays the properties of the output Record. Click on the **Invoke** button (an icon of a running man) to run the Command bean.
4. When the Command bean completes execution, the output Record properties will be populated. Notice the values of the *applid*, *date*, and *time* properties. Figure 8-14 shows the results of a successful invocation.



EPIMAPRecord Output	
String _Epimap1	"EPIPROG OUTPUT"
byte _Epimap1_Attbyte	-16
String _Epimap2	"APPLID:"
byte _Epimap2_Attbyte	-16
String _Epimap3	"DATE:"
byte _Epimap3_Attbyte	-16
String _Epimap4	"TIME:"
byte _Epimap4_Attbyte	-16
String applid	"SCSCPAA6"
byte applid_Attbyte	-16
String date	"14/11/01"
byte date_Attbyte	-16
String recordName	"itso.cics.epi.j2ee.EPIMA...
String recordShortDescription	""
String time	"20:43:23"
byte time_Attbyte	-16

itso.cics.epi.j2ee.EPIMAPRecord EPIPCommand Output

Figure 8-14 EAB Test Client

Writing a client to use the Command bean

To use `EPIPCommand` in a Java application:

- Create a class called `EPIPClient` in the `itso.cics.epi.j2ee` package and enter the code shown in Figure 8-15 in the `main()` method.

```
public static void main(java.lang.String[] args) {
  try{
     EPIPCommand command = new EPIPCommand();
     command.execute();
     System.out.println("CICS: " + command.getCeOutput0().getApplid());
     System.out.println("Date: " + command.getCeOutput0().getDate());
     System.out.println("Time: " + command.getCeOutput0().getTime());
  } catch(RuntimeException e) {
    System.out.println(e.getMessage());
  }
}
```

Figure 8-15 Testing the Command bean

The `getCeOutput0()` method returns an instance of the `EPIMAPRecord` class. This instance contains the values returned by the EPIP transaction. Notice how you do not need to understand the interaction with CICS when writing the client; all of this logic is handled internally by the Command bean.

Normally, input and output fields can be promoted to be visible from the Command bean, and thus instead of:

```
command.getCeOutput0().getApplid()
```

the *applid* value could be set using:

```
command.getApplid()
```

However, the fields *applid*, *date*, and *time* are defined as protected fields in the BMS map which was used to generate the output Record. A protected field means the value of the field cannot be changed. The EAB cannot promote protected fields, and thus the `getCeOutput0()` method is used instead. An example of promoting unprotected fields in a Record is shown on page 205.

- Set the classpath of `EPIPClient` by right clicking on the **`EPIPClient`** class and select **Properties**. Select the **Class Path** tab, then click **Compute Now**. This will determine the projects that are needed in the classpath to run `EPIPClient`. Once the classpath has been generated, click **OK**.
- Run `EPIPClient`. Example 8-3 shows the output from running this program.

Example 8-3 Results from the EPIPClient test client

```
CICS: SCSCPAA6
Date: 14/11/01
Time: 12:16:48
```

8.2.2 Extending an EAB application

The EAB provides much greater functionality than simply starting a CICS transaction, then examining the result. This section discusses extending the functionality of your applications in the following areas:

- Pseudo-conversational transactions
- Tracing

Pseudo-conversational transactions

We have seen that a Command bean can be used to make a single interaction to CICS. However, to perform many business functions, a user has a series of interactions with a CICS transaction. For example, to work with a CICS application a CICS user might:

- Start a CICS transaction
- Move through a set of menus pseudo-conversationally
- Exit from the application

This series of functions requires multiple Command beans. The EAB contains a *navigator* component which can be used to navigate a path through a series of Command beans. An application programmer that wants to invoke a CICS business function needs only to interact with the navigator by using the getter and setter methods, and calling the `execute()` method of the navigator. The application programmer does not need to know any details of the CICS transactions or Command beans used.

This section generates a navigator that interacts with the SWAP transaction. SWAP runs the CICS program SWAPPER which generates a screen with two unprotected fields; fields 4 and 6. Values are entered for these fields, then the updated screen is sent back to CICS. The resultant generated screen swaps the value of field 4 with the value of field 6, and vice versa. For more information on SWAP see Appendix C, “Sample CICS programs” on page 241.

This represents three interactions with CICS, thus three Command beans are required, as follows:

StartSWAPCommand	Starts transaction SWAP, and returns the generated screen in the output Record
SWAPCommand	Allows modification of two fields in the screen, runs the SWAP transaction, then returns the generated screen in the output Record
EndSWAPCommand	Exits the SWAP transaction by pressing F3. No input or output Records are used.

The following instructions describe how we generated a navigator that controls the execution of the Command beans for the SWAP transaction. If you are unfamiliar with EAB Records and commands, see 8.2.1, “Writing a simple EAB application” on page 194.

- In VisualAge for Java, create a Record that represents the screen format generated by the SWAP transaction.
  a. Using `swapset.bms` (see Appendix D, “Additional material” on page 261), create a Record Type called `SWAPSETRecordType` in the `itso.cics.epi.j2ee.nav` package of the *CICS Connectors Redbook* project.
  b. Use `SWAPSETRecordType` to generate a Record called `SWAPSETRecord`. Remember to select the **Generate as javax.resource.cci.Record interface** checkbox to build a Record bean suitable for use with the EPI resource adapter.
- Build the three Command beans `StartSWAPCommand`, `SWAPCommand`, and `EndSWAPCommand`. Use the information in Table 8-1 to set the correct values.

Note: When specifying a class name for the InteractionSpec, ensure you select the `EPIInteractionSpec` class from the `com.ibm.connector2.cics` (the J2EE connector related package) not from `com.ibm.connector.cics` (the Common Connector Framework related package)

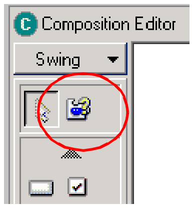
Table 8-1 Command bean properties

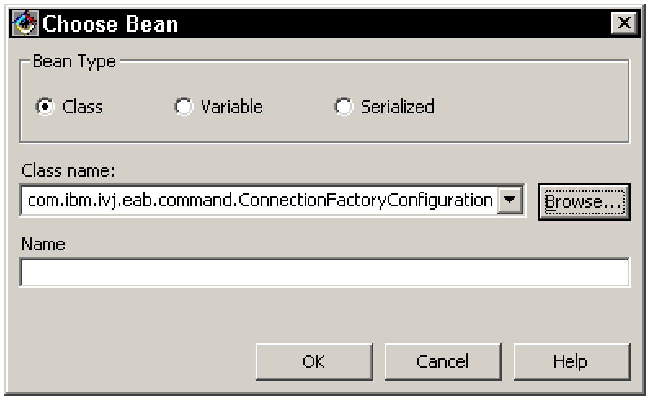
	StartSWAPCommand	**SWAPCommand**	**EndSWAPCommand**
Class name for Connection Information	no value	no value	no value
Class name for InteractionSpec	`EPIInteractionSpec` from the package `com.ibm.connector2.cics`	`EPIInteractionSpec` from the package `com.ibm.connector2.cics`	`EPIInteractionSpec` from the package `com.ibm.connector2.cics`
Input Record bean	no value	SWAPSETRecord	no value
Output Record bean	SWAPSETRecord	SWAPSETRecord	no value
functionName property	SWAP	no value	no value
AID property	enter	enter	PF3

Tip: You do not need to always specify an input and output Record when interacting with a started CICS transaction if you are using the EAB.

- You need to promote the properties you wish to make visible to the navigator. Update the *SWAPCommand* input Record so you can set properties *op1* and *op2*, and update the *SWAPCommand* output Record so you can get properties *op1* and *op2*.
  - Right click on **SWAPCommand** and select **Tools -> Enterprise Access Builder -> Edit Command**.
  - From the Command Editor, select **Input** in the top left pane, and **SWAPSETRecord** in the top right pane.
    - Right click on **op1** and select **Promote Property**.
    - Right click on **op2** and select **Promote Property**.
  - From the Command Editor, select **Output** in the top left pane, and **SWAPSETRecord** in the top right pane. Promote properties **op1** and **op2** as described above.
- A navigator is needed to control the flow of these commands. There are no SmartGuides for creating navigators, so you must use the Visual Composition Editor of VisualAge for Java to construct the navigator. Right click on the **Create Class** SmartGuide to build a new class in the `itso.cics.epi.j2ee.nav` package called `SWAPNavigator`. Specify this class uses **CommunicationNavigator** as its superclass. Select the **Compose the class visually** checkbox.
- The Visual Composition Editor will open. The Visual Composition Editor provides a logical place to build a navigator that will control the flow of three Command beans.

  The first task is to add the beans to use in the navigator.
  - Select the **Choose Bean** button, as shown.
  - In the *Choose Bean* window ensure *Bean Type* is set to *Class*.
  - Click the **Browse** button.
  - Type **ConnectionFactoryConfiguration** and clicking **OK**. The completed window is shown in Figure 8-16.





Class name: com.ibm.ivj.eab.command.ConnectionFactoryConfiguration

Name:

Figure 8-16 Choose Bean window

Click **OK** to add this bean to the navigator. A crosshair will appear, move this crosshair to the white central pane and click here to add the bean.

- Repeat this process for the Command beans you have created:
  - `StartSWAPCommand`
  - `SWAPCommand`
  - `EndSWAPCommand`

  Figure 8-17 shows all four beans added in the Visual Composition Editor.

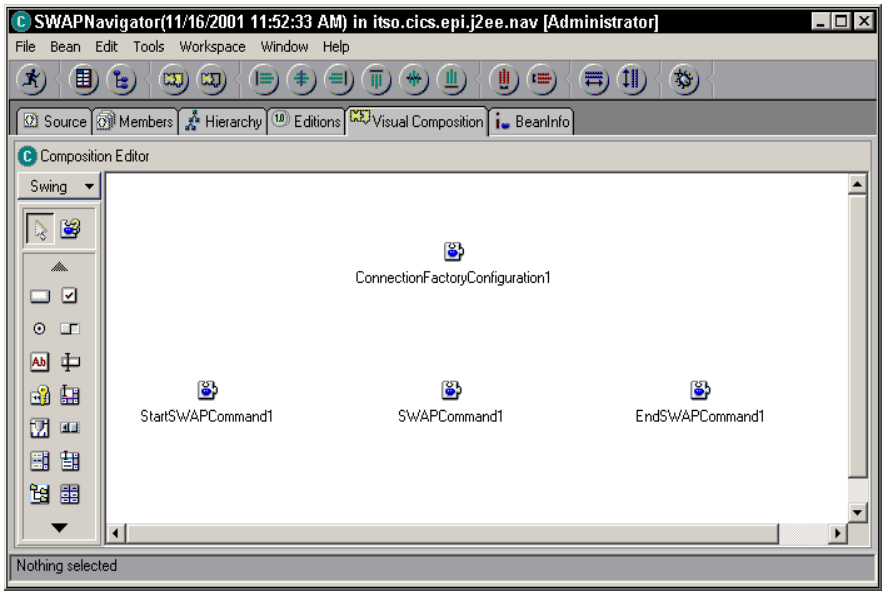
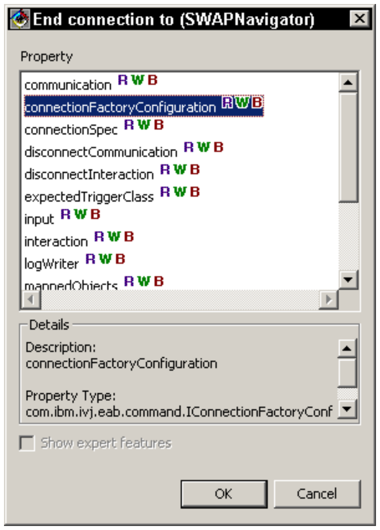


Figure 8-17 Beans in the Visual Composition Editor

- Edit the properties of *ConnectionFactoryConfiguration1* by double clicking on it. Set the *managedConnectionFactory* property to be *EPIManagedConnectionFactory*. Note that *EPIManagedConnectionFactory* is used regardless of whether you are creating a navigator to be deployed in a managed or non-managed environment. Set the following properties within the *EPIManagedConnectionFactory*:
  - **connectionURL** should be set to the network protocol and hostname of the CTG. We used `tcp://gunner`.
  - **serverName** should be set to the CICS server name to connect to. We used *SCSCPAA6*.
- To make the promoted properties in `SWAPCommand1` visible from the navigator:
  - Right click on **SWAPCommand1** and select **Promote Bean Feature**.
  - From the Property list select **OP1**, **OP2**, **OP11** and **OP21** individually and click **>>** to move them to the *Promoted features* list.
  - Click **OK**.
- The next stage is to connect the beans, based on events. The navigator needs some connection information (as we did not specify connection information in the individual Command beans).

  Right click on **ConectionFactoryConfiguration1** and select **Connect -> this**. The cursor changes to a spider. This is the first stage of a connection. To complete the connection click anywhere in the white space (this represents the navigator) to launch the *End connection to (SWAPNavigator)* window, as shown.


End connection to (SWAPNavigator)
Property
communication R W B
connectionFactoryConfiguration R W B
connectionSpec R W B
disconnectCommunication R W B
disconnectInteraction R W B
expectedTriggerClass R W B
input R W B
interaction R W B
logWriter R W B
mappedObjects R W B
Details
Show expert features
OK
Cancel

Description:
connectionFactoryConfiguration

Property Type:
com.ibm.ivj.eab.command.IConnectionFactoryConf

  Select **connectionFactoryConfiguration** and click **OK**. This sets the `connectionFactoryConfiguration` property of the navigator.
- The events of the navigator must be connected to the Command beans to create a logical flow.
  - When the navigator starts it should execute *StartSwapCommand1*. To do this:
    - Right click in the navigator window and select **Connect**, this opens the *Start connection from (SWAPNavigator)* window.

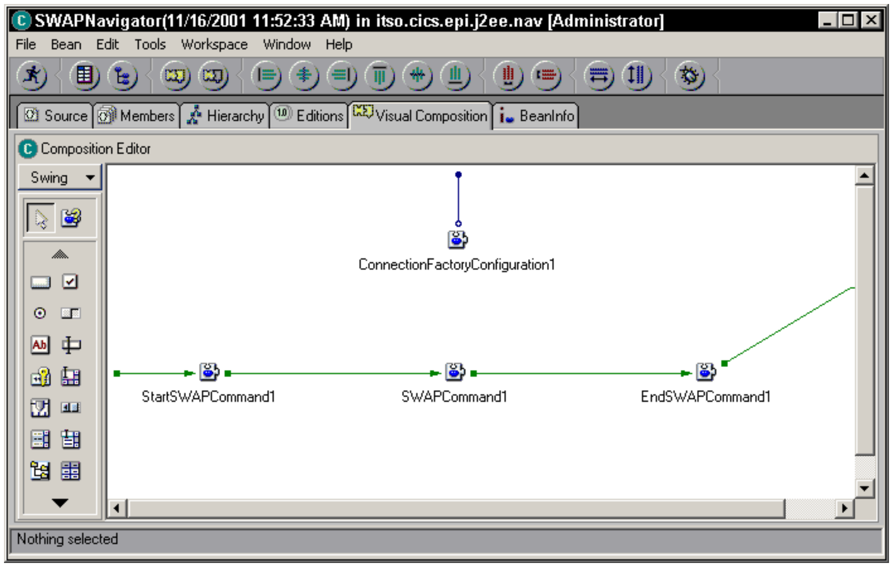
- Ensure **Event** is selected, then highlight the **internalExecutionStarting** event and click **OK**.
- Complete the connection by moving the spider cursor to the **StartSwapCommand1** bean and clicking this.
- From the pop-up menu select **Connectable Features** and select the **execute(CommandEvent)** method. A dotted green line represents the connection.

– After each command completes successfully the next command in the sequence should be executed.

- Connect the *executionSuccessful* event of StartSwapCommand1 with the *execute(CommandEvent)* method of SwapCommand1.
- Connect the *executionSuccessful* event of SwapCommand1 with the *execute(CommandEvent)* method of EndSwapCommand1.
- Connect the *executionSuccessful* event of EndSwapCommand1 with the *returnExecutionSuccessful* method of the navigator.

► The dotted green lines represent an incomplete connection. Double click on each of the four dotted green lines and select **Pass event data**. This will ensure the *CommandEvent* object is flowed between the Command beans.

The navigator is now complete. To generate the navigator select **Bean -> Save Bean**. Figure 8-18 shows a completed navigator.


SWAPNavigator(11/16/2001 11:52:33 AM) in itso.cics.epi.j2ee.nav [Administrator]

Nothing selected

Figure 8-18 A completed navigator

- You can test navigators in the EAB Test Client. To do this, from the workbench right click on the **SWAPNavigator** class and select **Tools -> Enterprise Access Builder -> Launch Test Client**. Further information on using this tool is in “Testing the Command bean” on page 200.

You should create a client to run the navigator. To do this, create a new class in the `itso.cics.epi.j2ee.nav` package called `SWAPClient` and enter the code shown in Figure 8-19 in the `main()` method. Example 8-4 shows the output from running this program.

```
public static void main(java.lang.String[] args) {
  String value1 = "ABCDE";
  String value2 = "12345";
  try{
     SWAPNavigator navigator = new SWAPNavigator();
     navigator.setSWAPCommand1Op1(value1);
     navigator.setSWAPCommand1Op2(value2);
     navigator.execute();
     System.out.println("Value 1: " + value1 + " has become: " +
                        navigator.getSWAPCommand1Op11());
     System.out.println("Value 2: " + value2 + " has become: " +
                        navigator.getSWAPCommand1Op21());
  } catch (RuntimeException e) {
     System.out.println(e.getMessage());
  }
}
```

Figure 8-19 Testing the navigator

Example 8-4 Results from the navigator client

```
Value 1: ABCDE has become: 12345
Value 2: 12345 has become: ABCDE
```

Tracing

To turn on tracing, edit the connection information in the Command bean or navigator. Select the `ConnectionFactoryConfiguration` class. Specify the level of tracing to use through `setTraceLevel()`. Valid tracing levels are 0 - 3:

0: RAS_TRACE_OFF	Disable all tracing (except adapter messages)
1: RAS_TRACE_ERROR_EXCEPTION	Output exception trace stacks
2: RAS_TRACE_ENTRY_EXIT	Output method entry and exit stack traces
3 = RAS_TRACE_INTERNAL	Output debug trace entries

Each level of trace builds upon the previous level, so ENTRY_EXIT includes everything in ERROR_EXCEPTION, and INTERNAL includes everything in all the trace levels.

You specify where the trace output should go in the `logWriter` property. Set this property in the Java client that calls the Command bean, making sure you set it before the `execute()` method is called (Figure 8-20).

```
EPIPCommand command = new EPIPCommand();
command.getCeConnectionSpec().getManagedConnectionFactory().setLogWriter(
                                     new java.io.PrintWriter(System.out));
command.execute();
```

Figure 8-20 Setting the logWriter property

Tip: We found that the CCI trace output was not particularly useful as it only traced the execution of methods within the CCI classes themselves. Therefore, we suggest you use the `com.ibm.ctg.client.T` class if you wish to debug your application. For details on using the `T` class, refer to 4.6, “Tracing” on page 61.

8.2.3 Migrating a CCF application

Prior to Version 4 of VisualAge for Java, the Enterprise Access Builder built classes that adhered to the Common Connector Framework (CCF) specification. CCF-based Records, commands, and navigators can all be migrated to the J2EE Connector Architecture using VisualAge for Java version 4. Before migrating any classes, we recommend you version them first.

Migrating Records and commands

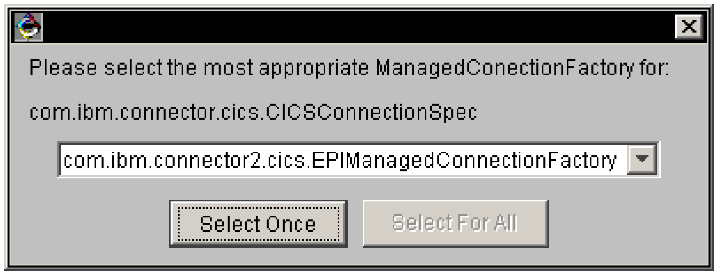
The command migrator performs the following actions:

- Changes the connection information (if any was supplied) from using the CCF-based `CICSConnectionSpec` class to the J2EE Connector Architecture based `ConnectionFactoryConfiguration` class.
- Changes the interaction spec from the CCF-based `EPIInteractionSpec` class to the J2EE Connection Architecure based `EPIInteractionSpec` class.

The Record migrator performs the following actions:

- Modifies the class to implement `javax.resource.cci.Record` and `javax.resource.cci.Streamable`.
- Implements the methods that these interfaces provide.

To migrate a command, right click on it and select **Tools -> Enterprise Access Builder -> Migrate Commands**. The *Migrate to Connector Architecture* SmartGuide will appear. You can optionally add additional commands to migrate here. Click **Finish** to start the migration. If the command contains connection information the window shown in Figure 8-21 will appear.


Please select the most appropriate ManagedConectionFactory for:

com.ibm.connector.cics.CICSConnectionSpec

com.ibm.connector2.cics.EPIManagedConnectionFactory

Figure 8-21 Migrating a command

Select **EPIManagedConnectionFactory**.

The following `CICSConnectionSpec` properties cannot be migrated:

- *reapTime*
- *realm*
- *unusedTimeout*
- *maxConnections*
- *minConnections*
- *connectionTimeout*

If any of these properties have been set a warning will be issued during the migration. The VisualAge for Java log will list all populated properties omitted from the migration.

Once the command migration is complete you will be asked if you wish to migrate all associated input and output Records. You should answer *Yes* to this since using CCF-based Records in a migrated Command bean will cause a `java.lang.ClassCastException` to be thrown at runtime.

Migrating navigators

The navigator migrator performs a single function; it changes the connection information from using the CCF-based `CICSConnectionSpec` class to the J2EE Connector Architecure based `ConnectionFactoryConfiguration` class.

Note that this tool only migrates navigators. The commands and Records that the navigator uses must be migrated separately.

To migrate a navigator do as follows:

- Right click on it and select **Tools -> Enterprise Access Builder -> Migrate Navigators**.
- The *Migrate to Connector Architecture* SmartGuide will appear, you can optionally add additional navigators here to migrate.
- Click **Finish** to start the migration.
- When asked to select the most appropriate ManagedConnectionFactory select **EPIManagedConnectionFactory**.

The migration tool will report that a number of properties could not be migrated. These are the same properties as listed in “Migrating Records and commands” on page 210. However, unlike the command migrator, the navigator migrator will report on all properties it has omitted, regardless of whether they have been set.

When viewing the migrated navigator in the Visual Composition Editor, the `CICSConnectionSpec` bean is not removed and replaced with the `ConnectionFactoryConfiguration` bean, as you might expect. Instead the `initalize()` method of the navigator is changed to ignore the values set in the `CICSConnectionSpec` bean, and creates a `ConnectionFactoryConfiguration` bean to use instead. Therefore, when updating connection parameters you should not modify values in the `CICSConnectionSpec` bean as you would have done before, but instead must modify the code in the `initalize()` method.

8.3 Connecting to secured CICS transactions

For an EPI application to be able to start secured CICS transactions it must supply security credentials (a user ID and password) for the CICS server to authenticate. There are two options available to do this:

- Signon to the CICS terminal before starting a transaction.

  The security credentials are determined at signon using CESN or any other signon transaction. These credentials are then used for all subsequent authorization checks when starting other transactions on this EPI terminal. A user ID and password should not flow with the requests that follow the signon. This option requires the use of a *signon capable terminal*.
- Flow a user ID and password with each EPI request.

  The user ID and password are authenticated for each request and used to authorize each transaction started in CICS. The user can not signon to CICS. This option requires the use of a *signon incapable terminal*.

We recommend you use *signon incapable* terminals if your application allows. There are two main reasons for this:

- Signing on to CICS prevents the resource adapter from controlling security, and instead relies upon a custom `LogonLogoff` class you have written yourself, which can be a security exposure if poorly written.
- If no LogonLogoff class is provided and you use signon capable terminals, the CICS application has to manage signon. However, when using connection pooling, the terminal returned to the pool could still have security credentials set on it. When the connection is reallocated from the pool there is no guarantee that the receiver of the terminal is going to change the credentials and this could circumvent security as they are allocated a terminal with credentials they may not be able to achieve themselves. To protect this scenario, the terminal gets disconnected when returned to the pool. This loses some of the performance benefit gained by pooling.

For further information on handling secure CICS transaction with the EPI refer to 7.3, “Connecting to secured CICS transactions” on page 174.

8.3.1 Signon capable terminals

For transactions that signon to CICS, the EPI resource adapter needs to know how to perform the signon. However the resource adapter does not know how to do this, because signon can vary on different CICS platforms, and you may have written your own custom 3270 signon transaction. Therefore, the resource adapter should invoke a `LogonLogoff` class to determine how to sign on to CICS.

You must write your own `LogonLogoff` class that implements the `com.ibm.connector2.cci.LogonLogoff` interface. This interface defines two methods that must be implemented; `logon()` and `logoff()`. The `logon()` method must drive the signon procedure you wish to use. A `Subject` object is passed to the `logon()` method from which you can obtain the user ID and password to use. The `logoff()` requires only a dummy implementation, because it is never called.

A sample LogonLogoff class is shipped with the CICS Transaction Gateway V4.0.1. It is called `com.ibm.ctg.samples.j2ee.CICSCESNLogon` and it extracts the user ID and password provided in the `Subject` object, and uses these values with the CESN transaction to signon.

When the `LogonLogoff` class is invoked, the resource adapter will pass it the security credentials to be used. In a non-managed environment, the security credentials used will be based in the same order of precedence as described in 8.3.2, “Signon incapable terminals” on page 214.

To specify the `LogonLogoff` class to use, call the `EPIManagedConnectionFactory` method `setLogonLogoffClass()`, specifying the package and class name of the `LogonLogoff` class to use as a `String`. If you are using the EAB, use the Command Editor to edit `ConnectionFactoryConfiguration`. Expand the `managedConnectionFactory` property to set the `logonLogoffClass` property.

Refer to the appropriate *CICS Transaction Gateway: Gateway Programming* manual for your platform for more information on writing a `LogonLogoff` class.

8.3.2 Signon incapable terminals

There are three ways in which security credentials can be passed to the resource adapter. They are listed in order of precedence.

1. **Server supplied credentials**

   In a managed environment, the J2EE server can provide security credentials. WebSphere Application Server Advanced Edition V4 does not currently support this.

2. **ConnectionSpec supplied credentials**

   The `EPIConnectionSpec` class can set the user ID and password to use. We recommend using this method when you are starting multiple EPI transactions that require different user IDs and passwords.

   To do this manually with the CCI, create an instance of `EPIConnectionSpec` and use the `setUserName()` and `setPassword()` methods. Pass the `EPIConnectionSpec` object as a parameter in the `getConnection()` method of the `ConnectionFactory`.

   When using Enterprise Access Builder, use the Command Editor to edit `ConnectionFactoryConfiguration`. Set the `connectionSpec` property to `EPIConnectionSpec`. Expand the `connectionSpec` property to set the `userName` and `password` properties.

3. **Deployed security credentials**

   The `EPIManagedConnectionFactory` class can set the user ID and password to use. We recommend using this method when you want to use a single user ID and password at deployment time for a connection.

   To do this manually with the CCI use the `setUserName()` and `setPassword()` methods of `EPIManagedConnectionFactory`.

   When using the Enterprise Access Builder, use the Command Editor to edit `ConnectionFactoryConfiguration`. Expand the `managedConnectionFactory` property to set the `userName` and `password` properties.

To use any of the above options, you must set the `EPIManagedConnectionFactory` class to specify you intend to work with a signon incapable terminal. The default assumes a signon capable terminal. Set the `setSignonType()` method to an `Integer` value of `1` to represent signon incapable. Alternatively, in Enterprise Access Builder use the Command Editor to edit `ConnectionFactoryConfiguration`. Expand the `managedConnectionFactory` property to set the `signonType` property to `1`.

Part 4

Appendices

A

Configuring the CICS connectors in VisualAge for Java

If you intend to use VisualAge for Java to develop applications like those shown in this book, you must install and configure it correctly. This appendix explains how we setup VisualAge for Java to support J2EE Connector Architecture application development. By configuring VisualAge for Java in this way you will also be able to use the updated version 4.0.1 CICS Transaction Gateway classes when creating applications that use the ECI or EPI interfaces directly.

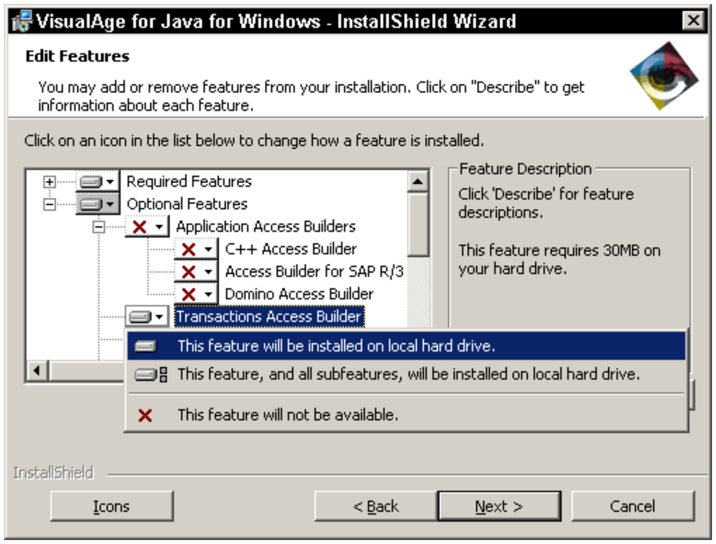
A.1 Installing VisualAge for Java

We began by installing VisualAge for Java V4.0 Enterprise Edition as this version contains the required support for the J2EE Connector Architecture. When installing VisualAge for Java, you must install the Enterprise Access Builder (EAB) support by choosing the following install option *Transactions Access Builder.*

Note: Because we wanted to be able to test servlets, JSPs, and enterprise beans, we also installed the WebSphere Test Environment by adding a further installation feature called *EJB/JSP Development Environment*

If you previously installed VisualAge for Java, but did not install the necessary options, you can still add them by modifying your installation in the following way:

- Version any packages and projects in your workspace and then exit VisualAge for Java.
- Run `setup.exe` located in the root directory of the VisualAge for Java installation CD to start the Installer.
- Choose the **Modify** option to change your installation and click **Next**.
- Install the necessary extras by clicking on the option name and choosing **This feature will be installed on local hard drive** as shown in Figure A-1.


VisualAge for Java for Windows - InstallShield Wizard

Edit Features

You may add or remove features from your installation. Click on "Describe" to get information about each feature.

Click on an icon in the list below to change how a feature is installed.

- Required Features
- Optional Features
  - Application Access Builders
    - C++ Access Builder
    - Access Builder for SAP R/3
    - Domino Access Builder
  - Transactions Access Builder

Feature Description

Click 'Describe' for feature descriptions.

This feature requires 30MB on your hard drive.

- This feature will be installed on local hard drive.
- This feature, and all subfeatures, will be installed on local hard drive.
- This feature will not be available.

Figure A-1 Adding options to a VisualAge for Java installation

A.2 Configuring VisualAge for Java

Once we installed VisualAge for Java with the Enterprise Access Builder, we then added and configured support for the J2EE Connector Architecture. We performed the following steps:

- Updated the EAB to support the J2EE Connector Architecture
- Updated the following connector projects with classes from CTG V4.0.1
  a. J2EE Connector Architecture project
  b. Connector CICS project

A.2.1 Updating the Enterprise Access Builder

An updated version of the EAB that supports the J2EE Connector Architecture is supplied with the beta version of the J2EE Connectors, found on the VisualAge for Java install CD. Information about the J2EE Connector Architecture support is contained in the readme file on the install CD in the following subdirectory:

```
extras\BetaJ2EEConnectors\readme.eab
```

Note: Installing the Beta J2EE Connectors provides two components. First, the EAB tools are enhanced to support J2EE Connector Architecture compliant Records and session beans. Second, classes are provided that implement the beta (proposed final draft version 2) of the J2EE Connector Architecture Specification 1.0.

The J2EE Connector Architecture Specification has now been updated from proposed final draft to final version 1.0, and so these beta level classes must be updated after the EAB has been updated. CICS Transaction Gateway V4.0.1 supplies 1.0 specification classes. We show how we used these to update our workspace in the next section.

The following steps show how to install the required EAB enhancements.

1. Ensure that all of your projects or packages in the workspace are versioned.
2. Exit VisualAge for Java.
3. Run one of the following setup programs, as appropriate for your system, from the VisualAge for Java install CD and follow the installation steps.
   - `extras\BetaJ2EEConnectors\NT\Setup.exe`
   - `extras\BetaJ2EEConnectors\W2000\Setup.exe`
4. When the setup program has completed, start VisualAge for Java
5. Delete any of the following projects that may exist in your workbench. The new versions cannot be added to the workspace until the old versions have been removed.

   - Connector CICS
   - IBM Common Connector Framework
   - IBM Enterprise Access Builder Library
   - IBM Enterprise Access Builder Samples
   - IBM Enterprise Access Builder WebSphere Samples
   - IBM Java Record Library
6. Choose **File -> Quick Start -> Features -> Add Feature** and then add:
   - IBM Enterprise Access Builder Library 4.0
7. Your workbench now contains version 4.0 of the EAB, which supports the J2EE Connector Architecture.

A.2.2 Updating the connector projects with 1.0 specification classes

Now that you have successfully added support for the J2EE Connector Architecture to the EAB you must update the beta classes to the final 1.0 specification. Your workspace may contain one or both of the following projects which must be updated, depending on what features you previously added to your VisualAge for Java workspace:

- J2EE Connector Architecture
- Connector CICS

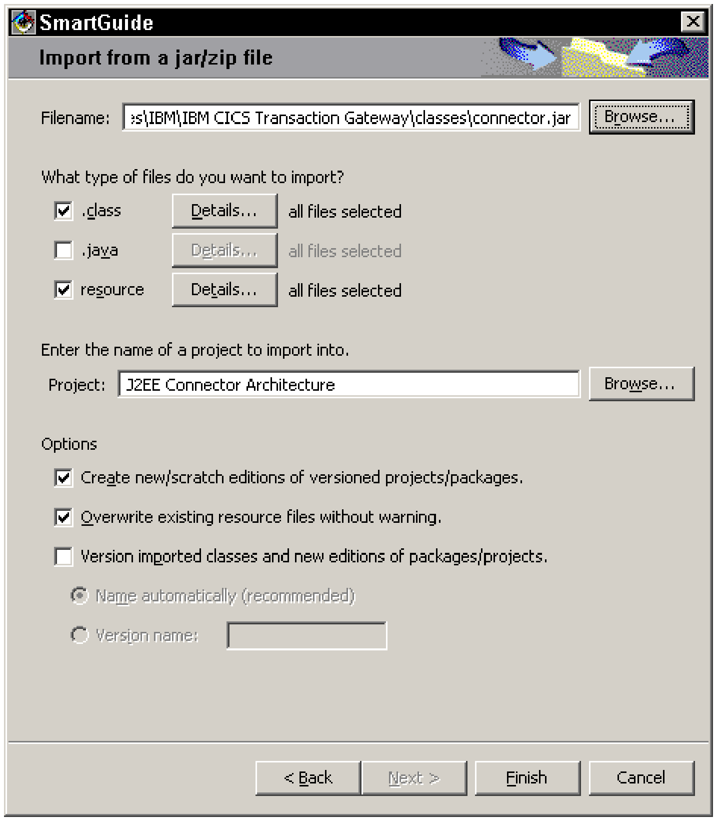
The 1.0 specification classes that we used for these updates are supplied by version 4.0.1 of the CICS Transaction Gateway. You will need to import these classes into your workspace from the `classes` subdirectory of a CTG V4.0.1 installation. Therefore, you must have access to a CTG V4.0.1 installation to perform this update. We installed CTG V4.0.1 onto the same machine that we installed VisualAge for Java on, but you could also obtain the necessary files from any installation of CTG V4.0.1

Updating the J2EE Connector Architecture project

If you have followed the instructions to this point, your workspace will currently contain a project entitled *J2EE Connector Architecture* with a version of *[Proposed Final Draft 2] 1.0*. The following steps show how we updated this project to 1.0 specification by importing new classes into the workspace:

1. Select the **J2EE Connector Architecture** project in the VisualAge for Java workbench.
2. Choose **File -> Import** from the VisualAge for Java workbench. Select **Jar file** as the import source.
3. Specify the full path to the file `connector.jar` in the *Filename* box. The JAR file is found in the `classes` subdirectory of a CTG V4.0.1 installation. In our example this was `C:\Program Files\IBM\IBM CICS Transaction Gateway\classes\connector.jar`

4. Ensure that the project to import to is *J2EE Connector Architecture* and set the remaining import options as shown in Figure A-2



SmartGuide

Import from a jar/zip file

Filename: :s\IBM\IBM CICS Transaction Gateway\classes\connector.jar

What type of files do you want to import?

- [✓] .class — all files selected
- [ ] .java — all files selected
- [✓] resource — all files selected

Enter the name of a project to import into.

Project: J2EE Connector Architecture

Options

- [✓] Create new/scratch editions of versioned projects/packages.
- [✓] Overwrite existing resource files without warning.
- [ ] Version imported classes and new editions of packages/projects.
  - [● Name automatically (recommended)]
  - Version name:

Figure A-2 Updating VisualAge for Java classes

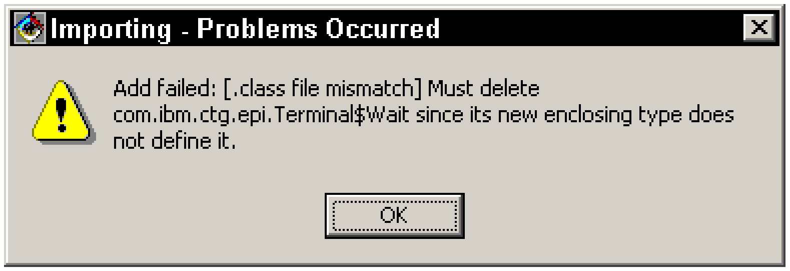
5. Click **Finish** to import the new classes.
6. We recommend that you now version the project and its classes. We right clicked on the **J2EE Connector Architecture** project and selected **Manage -> Create Open Edition**.
7. Right click on the project again and select **Manage -> Version** Click **One Name** from the resulting dialog box, and type **1.0** into the textbox. Click **OK**. Your workspace should now contain the project *J2EE Connector Architecture* with a version number *1.0*

Updating the Connector CICS project

Updating the Connector CICS project is slightly different from the previous two project updates. First, your workspace may already contain one, or even two projects, *Connector CICS* and *Connector CICS Beta*, depending on if you previously installed the CICS Connector. Second, we found that you must delete these existing projects from the workspace before importing files for the update to work correctly; importing into the existing projects will not work. Finally, the Connector CICS classes are found in more than one JAR file so multiple imports must be performed. In addition to these changes, the Connector CICS update also involves importing some Java Beans to the Visual Composition Editor palette, which can be used to visually construct a class. The following steps show how we imported the new classes and updated our workspace.

1. Start the VisualAge for Java workbench.
2. Delete the following projects if they exist in your workspace:
   - Connector CICS
   - Connector CICS Beta

> **Note:** If you do not remove the old Connector for CICS classes from the workspace, you will receive an error while importing the new CTG V4.0.1 classes. If you see an error message like the one shown in Figure A-3 then your workspace contained old CTG classes before you tried to import the new ones. If this happens, check to see if you removed the two projects from the workspace and ensure that there are no packages beginning with `com.ibm.ctg` elsewhere in the workspace.
>
> You can find these packages by right clicking on the projects view of your workspace and selecting **Go To -> Packages**. Selecting the package name will take you to its location in the workspace. Once you have removed these packages, start this step again and you will be able to successfully import the new classes.

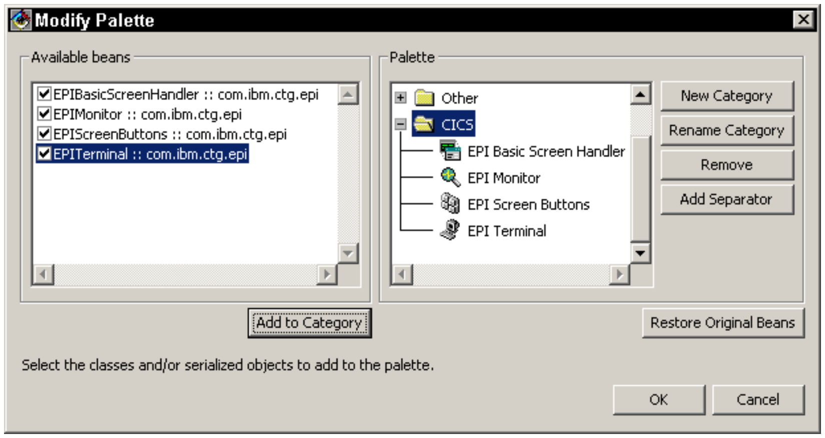


```
Add failed: [.class file mismatch] Must delete
com.ibm.ctg.epi.Terminal$Wait since its new enclosing type does
not define it.
```

Figure A-3 Importing J2EE Connector for CICS classes error message

3. Select **File -> Import** from the VisualAge for Java workbench. Select **Jar file** as the import source.

4. Specify the full path to the file `ctgclient.jar` in the *Filename* box. The JAR file is found in the `classes` subdirectory of a CTG V4.0.1 installation. In our example this was `C:\Program Files\IBM\IBM CICS Transaction Gateway\classes\ctgclient.jar`
5. Enter the project name as *Connector CICS* and once again set the remaining import options as shown in Figure A-2.
6. Click **Finish** to import the new classes.
7. At this point you will be asked to modify the bean palette used by the Visual Composition Editor for visually composing applications using Java Beans.
   - Add a new category to the palette by clicking the **New Category** button and name the new category *CICS*.
   - Now select the checkboxes for all of the new beans and ensure that the *CICS* category is highlighted.
   - Click the **Add to Category** button.
   - Your screen should look like that shown in Figure A-4. Click **OK** to complete the import.



Select the classes and/or serialized objects to add to the palette.

Figure A-4   Adding EPI beans to the palette

8. Now import `cicsj2ee.jar` from the same location as the previous JARs into the `Connector CICS` project by following the same procedure used to import `ctgclient.jar`. This time you do not need to create a new category for the additional beans. Select all the checkboxes for the additional beans as before and highlight the existing **CICS** category. Click **Add to Category** and then **OK**.
9. Import two further JAR files, `screenable.jar` and `ctgserver.jar`, from the same location as the previous ones into the same *Connector CICS* project. Neither of these JARs will require you to further modify the bean palette.

10. Now, we recommend that you version the project and its classes. Right click on the **Connector CICS** project and choose **Manage -> Version**. Select **One Name** from the resulting dialog box and type `4.0.1` into the text box. Click **OK**. Your workspace should now contain project *Connector CICS* with a version number *4.0.1*

Your VisualAge for Java is now configured correctly, so that you to begin building applications that use the J2EE Connector Architecture for connecting to CICS. Additionally, you can now use the version 4.0.1 CTG classes for ECI and EPI applications. Your workspace should contain the following projects and version numbers:

- IBM Enterprise Access Builder Library 4.0
- J2EE Connector Architecture 1.0
- Connector CICS 4.0.1

B

Data conversion

When writing Java applications to invoke CICS programs, data conversion is a key issue since CICS, which runs on IBM's S/390 processors, grew up in an EBCDIC world, whereas Java is based on Unicode which is derived from ASCII character sets that are used in the UNIX and PC worlds.

In this appendix, we provide information on three different strategies you can use to perform data conversions:

- Conversion within Java
- Conversion by CICS: DFHCNV templates
- EAB and Java Record Framework

Each of these options has its own advantages and drawbacks, and we will discuss each of these in the following sections.

B.1 Conversion within Java

If you are an avid Java programmer, the thought of writing you own data conversion code may appeal to you. Even if it does not, understanding how the data needs to be converted from the Unicode used within Java will be a real help.

Character data

All Java strings are stored in Unicode, which is a double byte character set, which is similar to ASCII in that the trailing byte maps to the ASCII code point for the common ASCII characters. Therefore, the character *A* usually represented by the ASCII code point `X ‘41’` is represented in Unicode by `X ‘0041’`.

The COMMAREA flowed to CICS in an `ECIRequest` object has to be a Java byte array (composed of single byte characters), whereas in Java, character data is usually stored in a String which is Unicode. Each time you convert from a String to a byte array you convert each character from Unicode to a single byte character, and therefore, you need to specify the encoding parameter to ensure consistent results.

When converting from a String to a byte array use the `getBytes()` method on the String class, passing the encoding of the byte array you wish to use. In this example we specify the EBCDIC code page IBM037.

```
byte abCommarea[] = new byte[27];
abCommarea = "abcd".getBytes("IBM037");
```

When converting the byte array to a String specify the correct encoding of the data on the String constructor as follows:

```
String strCommarea = new String(abCommarea,"IBM037");
```

This technique means that on the way into CICS data is converted from Unicode to EBCDIC within the JVM, and similarly on the way back data is converted from EBCDIC to Unicode. No data conversion is required within CICS (Figure B-1).

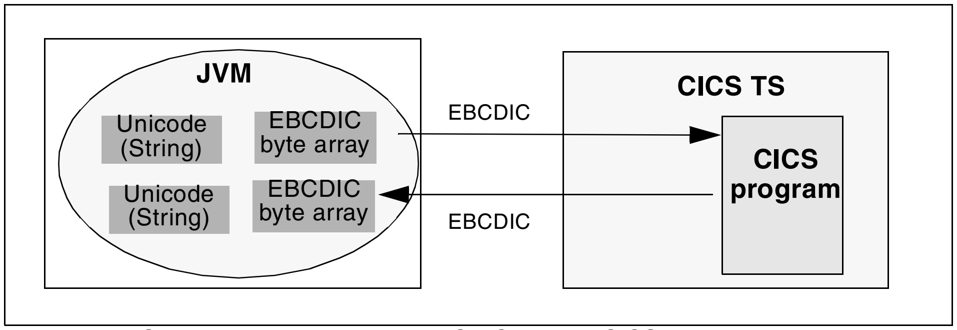


Figure B-1 Code page aware servlet: EBCDIC input to CICS

However, there is an alternative, which is to convert the data to ASCII within the JVM, and then convert from ASCII to EBCDIC within CICS (Figure B-2). This is not as inefficient as it sounds, since data conversion from Unicode to ASCII is an efficient operation in Java, as it involves only the removal of the high-order byte, whereas conversion to EBCDIC requires a table lookup. This means the high cost of EBCDIC conversion can be transferred to CICS, therefore, potentially improving performance within the JVM.

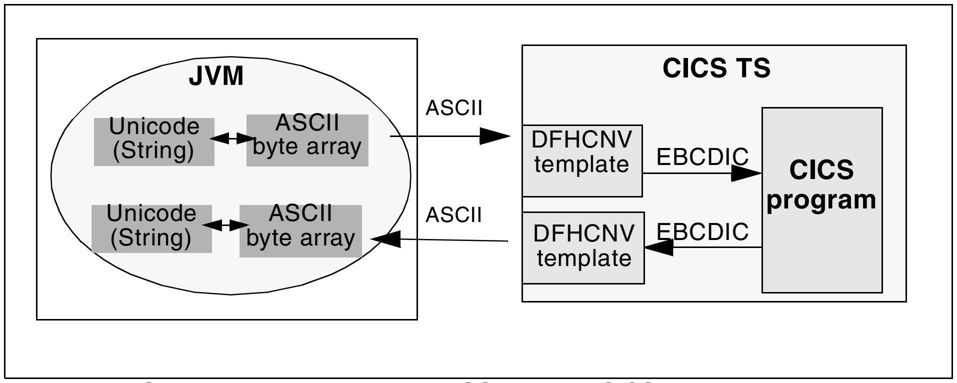


Figure B-2 Code page-aware servlet — ASCII input to CICS

In this case you would use an ASCII code page such as 8859_1 when creating the byte array:

```
byte abCommarea[] = new byte[27];
abCommarea = "abcd".getBytes("8859_1");
```

And having received the byte array back from CICS, convert to a String as follows:

```
String strCommarea = new String(abCommarea,"8859_1");
```

Attention: Specifying an encoding is particularly important if you port your code from Intel to S/390. This is because if you do not specify an encoding, then the default platform encoding will be used, and this will vary between ASCII and EBCDIC platforms.

Numeric data

If you wish to flow numeric data to CICS from a Java application then you may think that this is a relatively simple affair, since both Java and OS/390 store integers in big-endian format. However, you need to consider that all data passed to CICS must flow as byte array. Therefore, it is necessary to convert all integer values into a byte array before they can be passed to CICS.

The following code in Figure B-3 provides a method `getByteData()` to convert a four byte integer to the corresponding byte array. The `int` value is the value of the integer to be converted, and the length is assumed to be 4 bytes, and so will only work for 4 byte integers such as a Java `int` or a COBOL `PIC S9(8) COMP`.

Tip: The COBOL data type `PIC S9(8) COMP` is a 4-byte (full-word) signed numeric data-type. It can store integers ranging in value from -99999999 to +99999999, for further details refer to Table B-1 on page 240.

```
public static byte[] getByteData(int value) {
    byte[] reply = null;
    int length = 4;
    try {
1       java.io.ByteArrayOutputStream bytes = new java.io.ByteArrayOutputStream(length);
        java.io.DataOutputStream data = new java.io.DataOutputStream(bytes);

2       data.writeInt(value);

3       reply = bytes.toByteArray();
        data.close();
    } catch (java.io.IOException io) {
    }
    return reply;
}
```

Figure B-3 Method getByteData()

The logic in Figure C-3 is as follows:

- **1** Instantiate a `ByteArrayOutputStream` object. Then instantiate a `DataOutputStream` object, passing the `ByteArrayOutputStream` in the constructor.
- **2** Use the `DataOutputStream` `writeInt()` method to write the `int` value to the underlying output stream as four bytes, high byte first.
- **3** Use the `ByteArrayOutputStream` `toByteArray()` method to create a byte array from the original int `value`.

The following code in Figure B-4 provides the method `getIntData()` to convert a byte array, into a 4 byte integer. The `int offset` is the offset into the byte array of the integer, the length is assumed to be 4 bytes.

```
public static int getIntData(byte[] commarea, int offset) {

    int length = 4;
1   byte[] raw = new byte[length];
2   System.arraycopy(commarea, offset, raw, 0, length);

    int reply = 0;
    try {

3       java.io.ByteArrayInputStream bytes = new java.io.ByteArrayInputStream(raw);
        java.io.DataInputStream dataStream = new java.io.DataInputStream(bytes);

4       reply = dataStream.readInt();

        dataStream.close();
    } catch (java.io.IOException io) {
        System.out.println("Exception" + io);
    }
    return reply;
}
```

Figure B-4 Method getIntData()

The logic in Figure B-4 is as follows:

- **1** Create a new byte array `raw` of 4 bytes to hold the integer for conversion.
- **2** Populate the array `raw` with 4 bytes from the `commarea` array using the `arraycopy()` method.
- **3** Instantiate a `ByteArrayInputStream`. Then instantiate a `DataInputStream`, passing the `ByteArrayInputStream` object in the constructor.
- **4** Use the `DataInputStream` `readInt()` method to get the integer value from the byte array passed as input.

A sample application `IntConverter` that uses these methods to convert a byte returned from CICS into an integer, is provided in the package `itso.cics.eci.conversion` with this book. For details on how to obtain this code, refer to Appendix D, "Additional material" on page 261.

B.2 Conversion within CICS: DFHCNV templates

ECI applications use the facilities of the CICS mirror program (DFHMIRS) to link to the specified user program, passing a buffer known as the COMMAREA for input and output. The CICS mirror program can invoke the services of the data conversion program (DFHCCNV) to perform the necessary conversion of the inbound and outbound COMMAREA (Figure B-5).

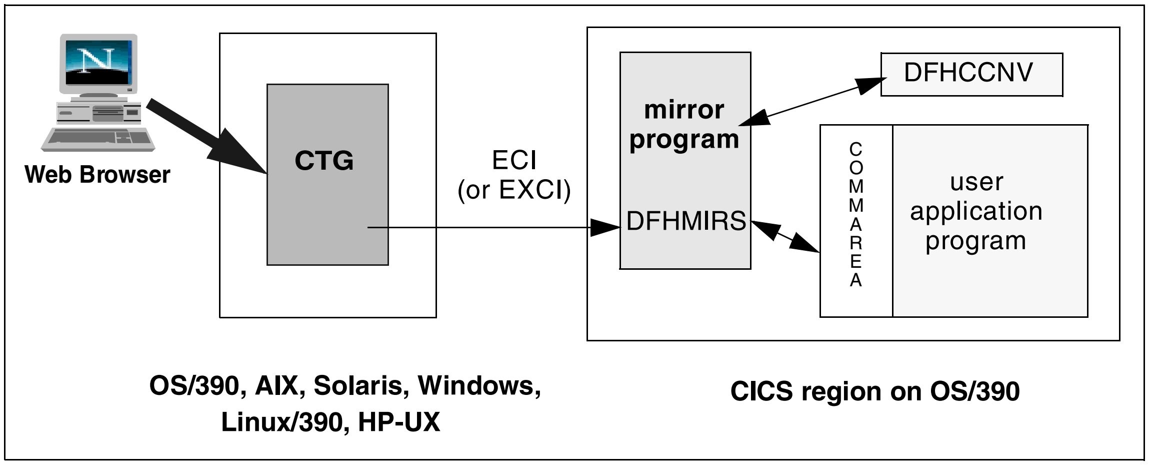


Figure B-5 CICS Transaction Gateway: ECI data conversion

Only *if* DFHCCNV finds a conversion template in the DFHCNV table, which matches the program name, will it perform code page translation for the COMMAREA associated with the ECI request. Templates must specify either character or numeric data as they are dealt with differently by the conversion program DFHCNV.

Character data

To convert character data it is necessary to create a `DATATYP=CHARACTER` template as shown in Figure B-6.

```
         DFHCNV TYPE=INITIAL,CLINTCP=8859-1,SRVERCP=037
         DFHCNV TYPE=ENTRY,RTYPE=PC,RNAME=ECIDCONV
         DFHCNV TYPE=SELECT,OPTION=DEFAULT
         DFHCNV TYPE=FIELD,OFFSET=0,DATATYP=CHARACTER,DATALEN=6,       x
         LAST=YES
         DFHCNV TYPE=FINAL
         END
```

Figure B-6 Sample DFHCNV character conversion template

- The SRVERCP on the TYPE=INITIAL statement should represent the EBCDIC code page in which the data is stored within CICS.
- The CLINTCP on the TYPE=INITIAL statement should be the default code page for client requests, but this is usually overridden by information flowed by from the CTG or CICS Universal Client, which is in turn determined from the code page of the client machine.
- The TYPE=ENTRY statement should specify RTYPE=PC for programs, and the name of the program in RNAME.
- The TYPE=FIELD statement should specify the DATATYP=CHARACTER for character based data. DATALEN should be the maximum length of the COMMAREA that you require to be translated.

Flowing of code page information

The CICS DFHCCNV program dynamically chooses the correct ASCII code page for its conversions based on information flowed to CICS from the CTG client daemon. Each ECI request flowed to CICS contains the code page information in the ECI header. This information is obtained from the CTG.INI parameters CCSID and USEOEMCP. USEOEMCP signifies that the code page in use by the operating system should be used. On Windows this code page can be determined using the **CHCP** command. The CCSID parameter can be used to override the USEOEMCP setting with a code page of your choice.

CicsCpRequest object

The CTG provides a request object, CicsCpRequest, to query the CTG about which code page it will specify in the ECI header. This provides a mechanism to allow proper data conversion within the Java application. The code in Figure B-7 details use of the CicsCpRequest object.

```
try {
1  JavaGateway jg= new JavaGateway();
   jg.setURL("tcp://gunner:2006");
   jg.open();

2   CicsCpRequest cpreq = new CicsCpRequest();
3   jg.flow(cpreq);
4   String jgCodePage = cpreq.getClientCp();
    System.out.println("CICS code page: " + jgCodePage);
    jg.close();
}catch (IOException ioe) {
   System.out.println("(Main) Handled exception: " + ioe.toString());
}
```

Figure B-7 Sample code for CicsCpRequest

The logic in Figure C-7 is as follows:

- **1** Create a `JavaGateway` object, configure it and open the connection.
- **2** Create a `CicsCpRequest` object.
- **3** Flow the request to the CTG.
- **4** After flowing the request invoke the `getClientCp()` method on the `CicsCpRequest` request object to obtain the ASCII code page in use by the worksation on which the CTG is running.

Having discovered the code page in use by the CTG, this should then be used as the ASCII code page when creating an ASCII byte array to flow to CICS. This ASCII byte array must then be subsequently converted to EBCDIC in CICS using a DFHCNV template.

The code in Figure B-8 illustrates how to use the `CicsCpRequest` object to query the CTG code page, and uses this information to create an ASCII byte array to flow to CICS in an ECI request. This code can be found as a working sample called `CodePageCall`, and it is provided in the `itso.cics.eci.conversion` package that comes with the additional material in this book.

```
try {
    JavaGateway jg = new JavaGateway();
    jg.setURL("tcp://gunner:2006");
    jg.open();

1   CicsCpRequest cpreq = new CicsCpRequest();
    jg.flow(cpreq);
    String jgCodePage = cpreq.getClientCp();
    System.out.println("CICS code page: " + jgCodePage);
2   byte abCommarea[];
3   abCommarea = ("---------------------------").getBytes(jgCodePage);

4   ECIRequest eciRequest;
    eciRequest = new ECIRequest("SCSCPAA6", // CICS Server
        "null", // userid
        "null", //password
        "ECIPROG", // Program name
        abCommarea, // Commarea byte array
        ECIRequest.ECI_NO_EXTEND, //extend mode
        ECIRequest.ECI_LUW_NEW); //LUW token
    jg.flow(eciRequest);

5   String strCommarea = new String(abCommarea, jgCodePage);
    System.out.println("COMMAREA returned: " + strCommarea);

}catch (IOException ioe) {
System.out.println("Handled exception: " + ioe.toString());
}
```

Figure B-8 Using the CicsCpRequest code page

The logic in Figure B-8 is as follows:

- **1** Create a `CicsCpRequest` object, to discover the CTG code page.
- **2** Create a byte array `abCommarea`.
- **3** Use the code page `jgCodePage` as the encoding when initializing the byte array with a string of data.
- **4** Create an `ECIRequest` object and use this to flow the COMMAREA to CICS.
- **5** Retrieve the COMMAREA returned from CICS, and use the code page `jgCodePage` when decoding the byte array into a string.

Numeric data

The representation of numeric (or integer data) is different on different computer systems. S/390 and RISC platforms use the *big-endian* format where the most significant byte (big end) is stored first (at the lowest storage address). Thus the unsigned numeric value of 1 is stored as `X'00 01'`.

Intel based machines use the opposite format, called *little-endian*, whereby the most significant byte is stored last (at the highest storage address). Therefore, the unsigned numeric value of 1 is stored as `X'01 00'`.

Attention: Although Intel platforms store integers in little-endian format Java uses a big-endian format. Therefore, you do not need to convert from little-endian to big-endian when passing data from your Java to CICS.

DFHCNV supports the conversion of 2 and 4 byte numeric data from little-endian format to big-endian. This is achieved using `DATATYP=NUMERIC`, such as shown in Figure B-9. A `DATATYP=BINARY` entry signifies the data is already in big-endian format and does not need any conversion. To convert 8 byte numeric values, packed decimal, or floating point values, it is necessary to use a different technique such as the Java Record Framework (see Figure B-10 on page 237) or the CICS user-replaceable conversion program DFHUCNV.

```
        DFHCNV TYPE=ENTRY,RTYPE=PC,RNAME=ECIDCONV
        DFHCNV TYPE=SELECT,OPTION=DEFAULT
        DFHCNV TYPE=FIELD,OFFSET=0,DATATYP=CHARACTER,DATALEN=6
        DFHCNV TYPE=FIELD,OFFSET=6,DATATYP=NUMERIC,DATALEN=2,         x
        LAST=YES
```

Figure B-9 Sample DFHCNV numeric conversion template

The template shown Figure B-9 is similar to the character template shown in Figure B-6 on page 232, except for the following differences:

- The `TYPE=FIELD` statement should specify the `DATATYP=NUMERIC`.
- `OFFSET` is the starting position of the integer within the commarea byte array.
- `DATALEN` should be the length of the integer you require translated (2 or 4 bytes). Each integer must have its own template specified.

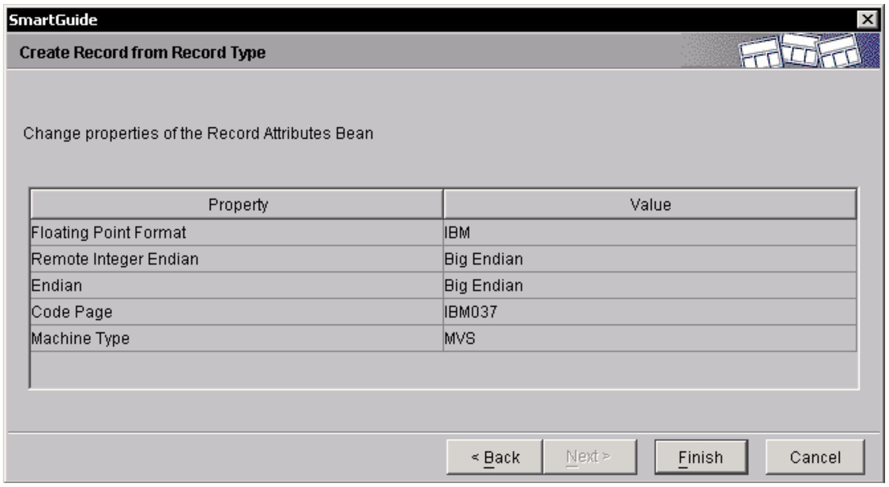
Important: If you code a `TYPE=BINARY` DFHCNV template, data flowed from a Java application through a Windows CTG will still be converted from little-endian to big-endian because CICS overrides the `TYPE-BINARY` field since it thinks the data has originated on a Windows system, and therefore, treats it as if it were little-endian data.

B.3 EAB and the Java Record Framework

The Enterprise Access Builder (EAB) feature of VisualAge for Java consists of set of SmartGuides and helper classes that can be used with connectors such as the CICS connector which allows you to seamlessly and easily access the data within the Enterprise Information System (such as CICS). Within the EAB, is the *Java Record Framework,* which is a powerful utility for marshalling data. This provides a fine-grained set of controls for marshalling all the possible CICS-COBOL data types from Java to S/390 format and vica-versa.

Using the EAB means that the programmer no longer has to concern them self with the offset, encoding, and data marshalling for each field in a COMMAREA. Instead, all the fields can now be correctly accessed using supplied getter and setters.

The *Create Record from Record Type* SmartGuide is used to set the data properties of the Record bean. When using the EAB Record SmartGuide, you are presented with a dialog that allows you to set the numeric data type, the code page, and the machine type for the Record (Figure B-10).



Change properties of the Record Attributes Bean

Property	Value
Floating Point Format	IBM
Remote Integer Endian	Big Endian
Endian	Big Endian
Code Page	IBM037
Machine Type	MVS

Figure B-10 Properties of the Record attributes beans

In the dialog shown in Figure B-10, you are prompted for five different property values that reflect the format of data in the COBOL record.

Floating Point Format

This is the byte representation of internal floating point numeric data types (such as `COMP-2` COBOL data types). Select **IBM** for S/390 format, otherwise select **IEEE**.

Remote Integer Endian

This is the format of 2 and 4 byte binary integers, (such as `PIC S9(4) COMP`, `PIC S9(8) COMP` and `PIC 9(8) COMP`. You should select **Big Endian** if you want to pass big-endian data to your CICS system. You should select **Little Endian** if you want to pass little-endian data. If you pass little-endian data to a CICS TS region on OS/390, you must convert it back to big-endian data by defining TYPE=NUMERIC templates in your DFHCNV macro.

Endian

This controls the format of all binary data types, including 2 and 4 byte integers, as well as 8 byte integers, such as `PIC 9(18) COMP`, and internal decimal (packed decimal) data types such as `PIC S9(5) COMP-3`. We recommend that you select **Big Endian** if you are sending data to a big-endian system, such as a CICS TS region on OS/390 (or a CICS AIX region). If you select **Little Endian**, this will disable conversion of all integers (including 2 and 4 byte integers), and should only be used if sending data to a CICS system running on an Intel platform (Windows or OS/2).

Code Page

This is the code page of character data, (such as `PIC X` or `PIC A` data types), and external floating point data (such as `PIC 9(2).9(2)E+99`). You can specify any code page listed in the JDK internationalization specification. You should use an EBCDIC code page (such as *IBM037*) if you wish to pass EBCDIC data to CICS; or you should use an ASCII code page (such as *8859_1*) if you wish to pass ASCII data to CICS. If you pass ASCII data to a CICS TS region running on OS/390, you must convert this to EBCDIC within CICS by defining DATATYP=CHARACTER templates in your DFHCNV macro.

Machine

This is the target machine type and controls data conversion for external decimal types (such as `PIC S9 DISPLAY`). You can specify the values *MVS, MVSCUSTOM, NT, OS2*, or *AIX*. You should use the value **MVS** for passing data to CICS on S/390, as this property is used to correctly convert the sign byte.

The property *MVSCUSTOM* is a special feature that handles conversion for external decimal types, which have already been converted from ASCII to EBCDIC by the CICS DFHCCNV (using a DFHCNV DATATYP=CHARACTER template). This is useful if you have a large character based COMMAREA, and wish to use DFHCNV templates, but also need to pass a few numeric data types within the COMMAREA.

The values shown in Figure B-10 on page 237 represent the suggested values to use when passing data from an Intel Java client to a CICS TS region running on OS/390. Using these values will delegate all data conversion to the Java client. If you wish to use DFHCNV to convert character data, you should specify the *Code Page* property as an ASCII code page (such as 8859_1). If you wish to use DFHCNV to convert numeric data (2 and 4 byte integers), you should specify the Remote Integer Endian property as *Little Endian*, and the Endian property as *Big Endian.*

The output from this dialog is stored in the constructor method for the Record bean (see Figure B-11), and determines how the getter and setter methods will convert the data to be flowed to the CICS system. This means that the data is not converted until the getter or setter methods are actually called, and provides more optimal performance, since data conversion can be an expensive operation in Java.

```
public EcidconvRecord()
      throws RecordException
   {
      try {
         if (this.initialRecord == null) {
            com.ibm.ivj.eab.record.cobol.CobolRecordAttributes attrs = new
com.ibm.ivj.eab.record.cobol.CobolRecordAttributes(
               "037",
               com.ibm.record.IRecordAttributes.BIGENDIAN,
               com.ibm.record.IRecordAttributes.BIGENDIAN,
               com.ibm.record.IRecordAttributes.IBMFLOATINGPOINTFORMAT,
               com.ibm.ivj.eab.record.cobol.CobolRecordAttributes.MVS,
               com.ibm.ivj.eab.record.cobol.CobolRecordAttributes.VACOBOL);
            this.setRecordAttributes(attrs);
            this.setRecordType(new
CustomRecordType(itso.cics.eci.conversion.EcidconvRecord.class,52));
            this.setRawBytes(new byte[52]);
            this.setInitialValues();
         } else {
            ...
      } catch (Exception e) {
         throw new RecordException(e.getMessage());
      }
   }
```

Figure B-11 Constructor for Record bean

The following Table B-1 summarize each EAB property, and the COBOL data types that each property affects.

Table B-1 COBOL data types and EAB Record attributes

COBOL data type	COBOL examples	Numeric value of +1	Record attribute
Character data	`PIC X or PIC A`	n/a	*Code Page*
External decimal	`PIC S9(4) DISPLAY`	`F0 F0 F0 C1`	*Machine*

Binary (2 or 4 byte) signed or unsigned	`PIC S9(4) BINARY`
	`PIC S9(8) BINARY`
	`PIC 9(4) BINARY`
	`PIC 9(8) BINARY`
		`00 01`
		`00 00 00 01`
		`00 01`
		`00 00 00 01`
			Endian

Binary (8 byte) signed or unsigned	`PIC 9(18) BINARY`	`00 00 00 00 00 00 00 01`	*Remote Integer Endian* plus *Endian*
Internal (packed) decimal	`PIC S9(4) COMP-3`	`00 00 00 1C`	*Remote Integer Endian* plus *Endian*

Internal floating point	`COMP-1`
	`COMP-2`
		`41 10 00 00`
		`41 10 00 00 00 00 00 00`
			FloatingPoint Format

External floating point	`PIC 9(2).9(2)E+99`	`4E F1 F0 4B F0 F0 C5 60 F0 F1`	*Code Page*

These attributes were calculated using a simple COBOL program (ECIDCONV) and an associated Java application (`itso.cics.eci.conversion.EcidconvTest`) and Record (`itso.cics.eci.conversion.EcidconvRecord`). Instructions on how to download these can be found in Appendix D, “Additional material” on page 261.

For further information on COBOL data types refer to the *VisualAge for COBOL Programming Guide*, SC26-9050.

C

Sample CICS programs

In this appendix, we provide the CICS source code for the following CICS programs applications that we used as samples in this book.

ECIADDER	This is a COMMAREA based program that takes a temporary storage queue and an integer as input, and adds the integer to the existing value in the queue.
ECIPROG	This is a simple COMMAREA based application that returns a 27 byte character string from the CICS region.
EPIPROG	This is a simple 3270 application that displays a single BMS map.
SWAP	This is a 3270 pseudo-conversational application that provides a BMS interface, allowing the user to enter numbers that are then swapped between display fields.
TRADER	This is a complex pseudo-conversational CICS application with a 3270 menu front-end. It simulates the buying and selling of shares in different companies. It has both a 3270 presentation interface and a COMMAREA based call interface.

Important: For details on how to download the latest copies of these additional materials refer to Appendix D, “Additional material” on page 261.

C.1 ECIADDER

This is a COMMAREA based program that takes as input a full-word integer and temporary storage queue. It then reads the contents of the queue, takes the integer input and adds the value to the existing value in the queue. This application was used for illustrating extended logical units of work in Chapter 4, "ECI and ESI applications" on page 35, and Chapter 5, "CCI applications: ECI based" on page 71. The source code is shown in Figure C-1.

Example: C-1 ECIADDER COBOL program

```
        IDENTIFICATION DIVISION.

        PROGRAM-ID. ECIADDER.

        ENVIRONMENT DIVISION.
        DATA DIVISION.

        WORKING-STORAGE SECTION.

        01  RESP                         PIC S9(8) BINARY.
        01  RESP-SAVE                    PIC S9(8) BINARY.
        01  APPLID                       PIC X(8).
        01  ITEM                         PIC S9(4) BINARY.
        01  DATA-Q-BIN1                  PIC S9(8) BINARY VALUE ZERO.
        01  Q-COUNT                      PIC S9(8) BINARY VALUE ZERO.
        01  DATA-IN-BIN                  PIC S9(8) BINARY.

       *COMMAREA IS 12 BYTES
        01 COMMAREA-RETURN.
           03 DATA-OUT                   PIC S9(3) DISPLAY SIGN IS
                                         LEADING SEPARATE CHARACTER.
           03 Q-RC                       PIC ZZZ9 DISPLAY.
           03 FILLER-1                   PIC X(4) VALUE SPACE.

        LINKAGE SECTION.

       *COMMAREA IS 12 BYTES, 4 BYTE NUMERIC, 8 BYTE TSQ
       *DATA-IN IS 3 BYTE FIELD WITH LEADING SIGN
        01 DFHCOMMAREA.
           03 DATA-IN                    PIC S9(3) DISPLAY SIGN IS
                                         LEADING SEPARATE CHARACTER.
           03 QNAME                      PIC X(8).

        PROCEDURE DIVISION.

            EXEC CICS ASSIGN APPLID(APPLID)
                      END-EXEC.
```

```
    MOVE DATA-IN TO DATA-IN-BIN.

Q-WRITE.
    MOVE 1 TO ITEM.
    EXEC CICS READQ TS
         QUEUE(QNAME)
         ITEM(ITEM)
         INTO(DATA-Q-BIN1)
         RESP(RESP)
    END-EXEC.
    MOVE RESP TO RESP-SAVE.

    IF RESP-SAVE = DFHRESP(QIDERR) THEN
      EXEC CICS WRITEQ TS
                QUEUE(QNAME)
                FROM(DATA-IN-BIN)
                LENGTH(LENGTH OF DATA-IN-BIN)
                ITEM(ITEM)
                RESP(RESP)
                END-EXEC.

    IF RESP-SAVE = DFHRESP(NORMAL) THEN
      ADD DATA-IN-BIN TO DATA-Q-BIN1
      MOVE 1 TO ITEM
      EXEC CICS WRITEQ TS
                QUEUE(QNAME)
                REWRITE
                FROM(DATA-Q-BIN1)
                LENGTH(LENGTH OF DATA-Q-BIN1)
                ITEM(ITEM)
                RESP(RESP)
                END-EXEC.

    MOVE RESP TO Q-RC.
    MOVE DATA-Q-BIN1 TO DATA-OUT.
    PERFORM CICS-RETURN.


CICS-RETURN.
    MOVE COMMAREA-RETURN TO DFHCOMMAREA.
    EXEC CICS RETURN
              END-EXEC.

    EXIT.
```

C.2 ECIPROG

This is a very simple COMMAREA based application. It merely returns a 27 byte string containing the date, time and APPLID from the CICS region. It is a useful application for testing simple calls to CICS.

Optionally it also takes input, and will either delay for 30 seconds, abend, or shutdown CICS. Depending on whether the first character in the input COMMAREA is D, S, or A. It was used in Chapter 5, “CCI applications: ECI based” on page 71. The COBOL source code is shown in Figure C-2.

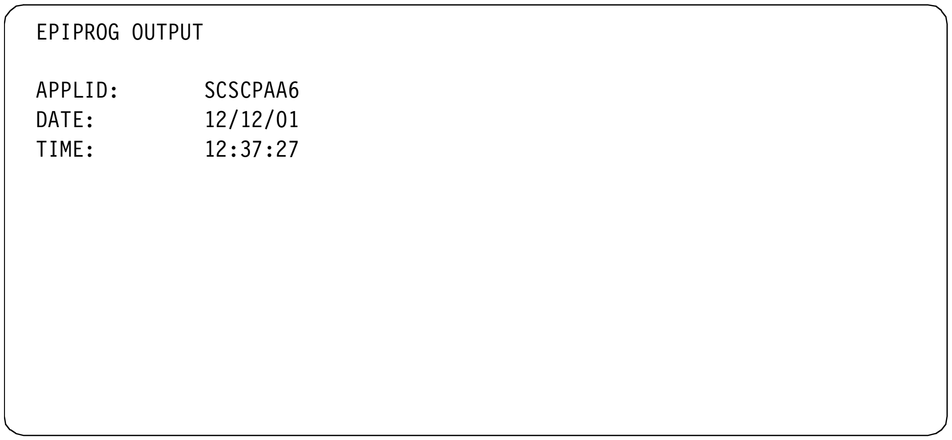
Example: C-2 ECIPROG COBOL program

```
       IDENTIFICATION DIVISION.

       PROGRAM-ID. ECIPROG.

       ENVIRONMENT DIVISION.
       DATA DIVISION.

       WORKING-STORAGE SECTION.

       01  RESP                          PIC S9(8) BINARY.
       01  ABSTIME                       PIC X(8).
       01  DELAY-TIME                    PIC S9(8) COMP VALUE 1.
       01  ABEND-CODE                    PIC X(4) VALUE 'ECIP'.
       01 COMMAREA-IN.
          03 ACTION1                     PIC X(1).
          03 FILLER-1                    PIC X(26) VALUE SPACE.

       LINKAGE SECTION.

       01 DFHCOMMAREA.
          03 APPLID                      PIC X(8).
          03 FILLER-1                    PIC X(1) VALUE SPACE.
          03 DATE-AREA                   PIC X(8).
          03 FILLER-2                    PIC X(1) VALUE SPACE.
          03 TIME-AREA                   PIC X(8).
          03 FILLER-3                    PIC X(1) VALUE SPACE.

       PROCEDURE DIVISION.

           MOVE DFHCOMMAREA TO COMMAREA-IN.
           MOVE SPACES TO DFHCOMMAREA.

           EXEC CICS ASSIGN APPLID(APPLID)
                     END-EXEC.
```

```
        IF ACTION1 = 'D'
          EXEC CICS DELAY FOR SECONDS(30)
                  END-EXEC.

        IF ACTION1 = 'A'
          EXEC CICS ABEND ABCODE(ABEND-CODE)
                  END-EXEC.

        IF ACTION1 = 'S'
          EXEC CICS PERFORM SHUTDOWN IMMEDIATE
                  END-EXEC.

        EXEC CICS ASKTIME ABSTIME(ABSTIME)
                  END-EXEC.

        EXEC CICS FORMATTIME
                  ABSTIME(ABSTIME)
                  DDMMYY(DATE-AREA)
                  DATESEP("/")
                  TIME(TIME-AREA)
                  TIMESEP(":")
                  END-EXEC.

        EXEC CICS WRITEQ TD
                  QUEUE("CSMT")
                  FROM(DFHCOMMAREA)
                  LENGTH(27)
                  RESP(RESP)
                  END-EXEC.

    WS-RETURN SECTION.
        EXEC CICS RETURN
                  END-EXEC.
    WS-RETURN-EXIT.
        EXIT.
```

C.3 EPIPROG

This is a very simple non-conversational 3270 application that returns just one BMS map, containing 3 fields containing the date, time and APPLID from the CICS region. The 3270 screen returned by EPIPROG is shown in Figure C-2



```
EPIPROG OUTPUT

APPLID:       SCSCPAA6
DATE:         12/12/01
TIME:         12:37:27
```

Figure C-1 EPIP 3270 output

It was used in Chapter 8, "CCI applications: EPI based" on page 181 and Chapter 8, "CCI applications: EPI based" on page 181. The COBOL source code is shown in Figure C-3.

Example: C-3 EPIPROG COBOL program

```
IDENTIFICATION DIVISION.

       PROGRAM-ID. EPIPROG.
       ENVIRONMENT DIVISION.
       DATA DIVISION.

       WORKING-STORAGE SECTION.
      * BMS MAP
       COPY EPIMAPS.
      * STANDARD DEFINITIONS FOR BMS
       COPY DFHAID.
       01  ABSTIME                     PIC X(8).
       01  ABSTIMEX                    PIC X(8).
       LINKAGE SECTION.
       PROCEDURE DIVISION.
           MOVE LOW-VALUES TO EPIMAPO.
           EXEC CICS ASSIGN APPLID(APPLIDO)
                     END-EXEC.
           EXEC CICS ASKTIME ABSTIME(ABSTIME)
                     END-EXEC.
           EXEC CICS FORMATTIME
                     ABSTIME(ABSTIME)
                     DDMMYY(DATEO)
                     DATESEP("/")
                     TIME(TIMEO)
                     TIMESEP(":")
                     END-EXEC.
           EXEC CICS SEND MAP('EPIMAP')
                          MAPSET('EPIMAPS')
                          ERASE FREEKB
                     END-EXEC.
           EXEC CICS RETURN
                     END-EXEC.
           EXIT.
```

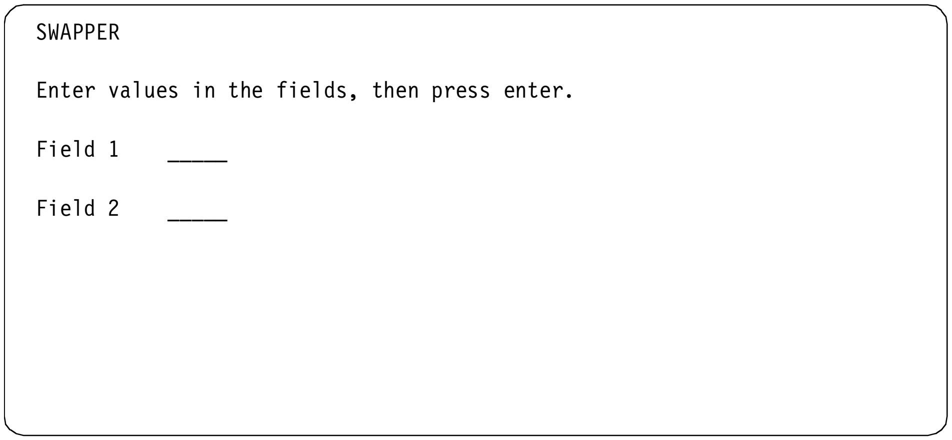
The BMS mapset used by EPIPROG is shown in Figure C-4. Note that a BMS mapset can contain multiple BMS maps but in our example the mapset EPIMAPS contains only one map, EPIMAP.

Example: C-4 EPIMAPS BMS mapset

```
EPIMAPS  DFHMSD TYPE=&SYSPARM,                                         X
               LANG=COBOL,                                             X
               MODE=OUT,                                               X
               CTRL=FREEKB,                                            X
               STORAGE=AUTO,                                           X
               DSATTS=(COLOR,HILIGHT),                                 X
               MAPATTS=(COLOR,HILIGHT),                                X
               TIOAPFX=YES
***********************************************************************
EPIMAP   DFHMDI SIZE=(24,80),                                          X
               LINE=1,                                                 X
               COLUMN=1
***********************************************************************
         DFHMDF POS=(1,1),                                             X
               LENGTH=14,                                              X
               ATTRB=(ASKIP,PROT),                                     X
               INITIAL='EPIPROG OUTPUT'
         DFHMDF POS=(3,1),                                             X
               LENGTH=7,                                               X
               ATTRB=(ASKIP,PROT),                                     X
               INITIAL='APPLID:'
APPLID   DFHMDF POS=(3,15),                                            X
               LENGTH=8,                                               X
               ATTRB=(ASKIP,PROT)
         DFHMDF POS=(4,1),                                             X
               LENGTH=5,                                               X
               ATTRB=(ASKIP,PROT),                                     X
               INITIAL='DATE:'
DATE     DFHMDF POS=(4,15),                                            X
               LENGTH=8,                                               X
               ATTRB=(ASKIP,PROT)
         DFHMDF POS=(5,1),                                             X
               LENGTH=5,                                               X
               ATTRB=(ASKIP,PROT),                                     X
               INITIAL='TIME:'
TIME     DFHMDF POS=(5,15),                                            X
               LENGTH=8,                                               X
               ATTRB=(ASKIP,PROT)
***********************************************************************
         DFHMSD TYPE=FINAL
         END
```

C.4 SWAP

This is a simple pseudo-conversation 3270 application. It returns a single map, containing two input fields. The input to these two fields is then *swapped* when the Enter key is pressed. Note that this application is based on the SWAPPER sample provided by VisualAge for Java V4. The 3270 screen displayed by the application is shown in Figure C-2



```
SWAPPER

Enter values in the fields, then press enter.

Field 1    _____

Field 2    _____
```

Figure C-2 SWAP 3270 output

The sample was used in Chapter 8, "CCI applications: EPI based" on page 181. The COBOL source code is shown in Example C-5.

Example: C-5 SWAP COBOL program

```
IDENTIFICATION DIVISION.

     PROGRAM-ID. SWAP.

     ENVIRONMENT DIVISION.
     DATA DIVISION.

     WORKING-STORAGE SECTION.
     01 TMP PIC A(5).
     01 COMMUNICATION-AREA PIC X.
     01 END-OF-SESSION-MESSAGE PIC X(13) VALUE 'SESSION ENDED'.
    * BMS SYMBOLIC MAP
     COPY SWAPSET.
    * STANDARD DEFINITIONS FOR BMS.
     COPY DFHAID.
     LINKAGE SECTION.
    * USE TO INDICATE WHETHER FIRST INVOCATION OF PROGRAM OR NOT.
```

```
       01  DFHCOMMAREA PIC X.
       PROCEDURE DIVISION.
       START-PARA.
           EVALUATE TRUE

             WHEN EIBCALEN = ZERO
      *        FIRST INVOCATION OF THE PROGRAM
               MOVE LOW-VALUE TO SWAPMAPO
               PERFORM SEND-SWAPMAP-CLEAR
             WHEN EIBAID = DFHCLEAR
               MOVE LOW-VALUE TO SWAPMAPO
               PERFORM SEND-SWAPMAP-CLEAR
             WHEN EIBAID = DFHPA1 OR DFHPA2 OR DFHPA3
               CONTINUE
             WHEN EIBAID = DFHPF3 OR DFHPF12
               PERFORM SEND-TERMINATION-MESSAGE
               EXEC CICS RETURN
               END-EXEC
             WHEN EIBAID = DFHENTER
               PERFORM PROCESS-MAP
             WHEN OTHER
               CONTINUE
           END-EVALUATE.
           EXEC CICS
               RETURN TRANSID('SWAP')
                      COMMAREA(COMMUNICATION-AREA)
                      LENGTH(1)
           END-EXEC.
PROCESS-MAP.
           PERFORM RECEIVE-SWAPMAP.
           PERFORM SWAP-PROCESS.
           PERFORM SEND-SWAPMAP.
      *
       RECEIVE-SWAPMAP.
           EXEC CICS
               RECEIVE MAP('SWAPMAP')
                       MAPSET('SWAPSETM')
                       INTO(SWAPMAPI)
           END-EXEC.
      *
       SEND-SWAPMAP.
           EXEC CICS
               SEND MAP('SWAPMAP')
                    MAPSET('SWAPSETM')
                    FROM(SWAPMAPO)
                    DATAONLY
           END-EXEC.
      *
       SEND-SWAPMAP-CLEAR.
```

```
           EXEC CICS
               SEND MAP('SWAPMAP')
                    MAPSET('SWAPSETM')
                    FROM(SWAPMAPO)
                    ERASE
           END-EXEC.
      *
       SEND-TERMINATION-MESSAGE.
           EXEC CICS
               SEND TEXT FROM(END-OF-SESSION-MESSAGE)
                    ERASE
                    FREEKB
           END-EXEC.
      *
       SWAP-PROCESS.
           MOVE OP1I TO TMP.
           MOVE OP2I TO OP1O.
           MOVE TMP TO OP2O.
```

The BMS mapset used by SWAP is shown in Example C-6. Note that a BMS mapset can contain multiple BMS maps but in our example the mapset SWAPSET contains only one map, SWAPMAP.

Example: C-6 SWAPSET BMS mapset

```
WAPSET  DFHMSD TYPE=&SYSPARM,                                          X
               LANG=COBOL,                                             X
               MODE=INOUT,                                             X
               TERM=3270-2,                                            X
               CTRL=FREEKB,                                            X
               STORAGE=AUTO,                                           X
               DSATTS=(COLOR,HILIGHT),                                 X
               MAPATTS=(COLOR,HILIGHT),                                X
               TIOAPFX=YES
SWAPMAP  DFHMDI SIZE=(24,80),                                          X
               LINE=1,                                                 X
               COLUMN=1
         DFHMDF POS=(1,1),                                             X
               LENGTH=7,                                               X
               ATTRB=(NORM,PROT),                                      X
               COLOR=BLUE,                                             X
               INITIAL='SWAPPER'
         DFHMDF POS=(3,1),                                             X
               LENGTH=45,                                              X
               ATTRB=(NORM,PROT),                                      X
               COLOR=BLUE,                                             X
               INITIAL='Enter values in the fields, then press enter.'
         DFHMDF POS=(5,1),                                             X
```

```
               LENGTH=8,                                               X
               ATTRB=(NORM,PROT),                                      X
               COLOR=GREEN,                                            X
               INITIAL='Field 1 '
OP1      DFHMDF POS=(5,12),                                            X
               LENGTH=5,                                               X
               ATTRB=(NORM,UNPROT,IC),                                 X
               COLOR=TURQUOISE,                                        X
               INITIAL='_____'
         DFHMDF POS=(7,1),                                             X
               LENGTH=8,                                               X
               ATTRB=(NORM,PROT),                                      X
               COLOR=GREEN,                                            X
               INITIAL='Field 2 '
SOP2     DFHMDF POS=(7,12),                                            X
               LENGTH=5,                                               X
               ATTRB=(NORM,UNPROT),                                    X
               COLOR=TURQUOISE,                                        X
               INITIAL='_____'
         DFHMDF POS=(24,1),                                            X
               LENGTH=20,                                              X
               ATTRB=(NORM,PROT),                                      X
               COLOR=BLUE,                                             X
               INITIAL='F3=Exit   F12=Cancel'
DUMMY    DFHMDF POS=(24,79),                                           X
               LENGTH=1,                                               X
               ATTRB=(DRK,PROT,FSET),                                  X
               INITIAL=' '
***********************************************************************
         DFHMSD TYPE=FINAL
         END
```

C.5 TRADER

This appendix describes the 3270 COBOL Trader application used as the basis of the enterprise bean examples provided in Part 3, “Connecting to 3270 based CICS transactions” on page 153 of this redbook.

We start by providing a description of how the original 3270 based version of the Trader application functions. We then provide a summary of what definitions are required when installing the Trader application in your CICS system.

To obtain the sample COBOL Trader application and accompanying JCL, refer to Appendix D, “Additional material” on page 261.

The 3270 Trader COBOL application

Trader, written in COBOL, uses the VSAM access method for file access and the CICS 3270 BMS programming interface. It is a pseudo-conversational application, meaning that a chain of related non-conversational CICS transactions is used to convey the impression of a conversation to the users as they go through a sequence of screens that constitute a business transaction. The application consists of two modules: TRADERPL, which contains the 3270 presentation logic; and TRADERBL, which contains the business logic. TRADERPL invokes TRADERBL using an EXEC CICS LINK and passing a COMMAREA structure for input and output. TRADERBL contains logic to query and write to the persistent VSAM data, stored in two files the *company file* and the *customer file*.

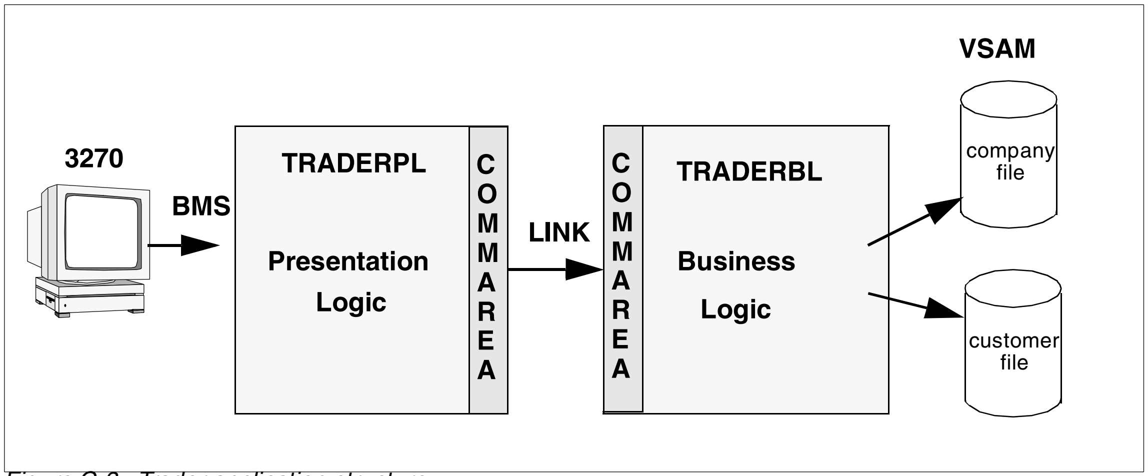


Figure C-3 Trader application structure

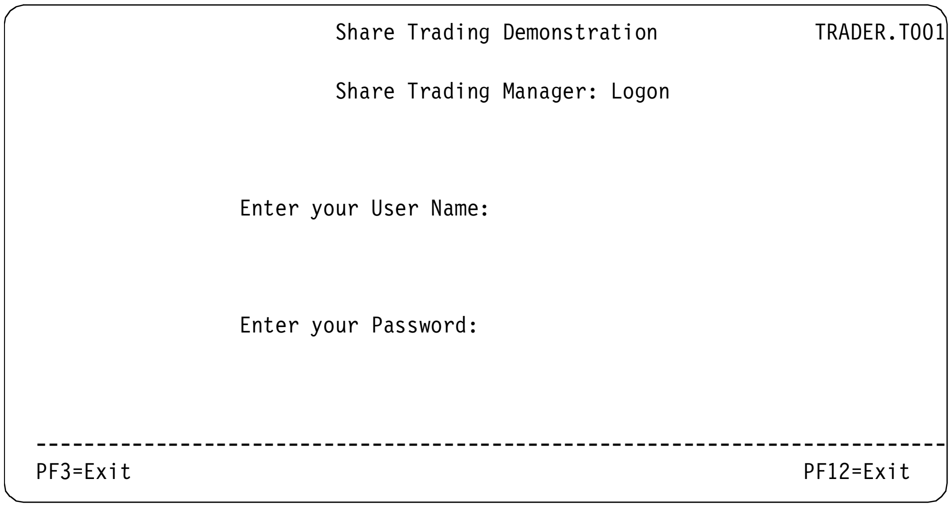
At each step, the application presents a set of options. The user makes a choice, then presses the required key in order to send their selections back to the application. The application performs the necessary actions based on the user's choice and presents the results together with any possible new options. The application has a strict hierarchical menu structure which allows the user to return to the previous step by using the PF3 key.

3270 application flows

In this section, we describe a typical business transaction when using the 3270 Trader application:

1. The program TRADERPL is invoked on a 3270 capable terminal by entering the initial CICS transaction identifier (TRAD). TRADERPL calls TRADERBL, passing an inter-program COMMAREA of 400 bytes. TRADERBL expects the COMMAREA to contain a request type and associated data. There are three request types: *Get_Company* to return a company list, *Share_Value* to return a list of share values, or *Buy_Sell* to buy or sell shares. In this step the request type is *Get_Company*.

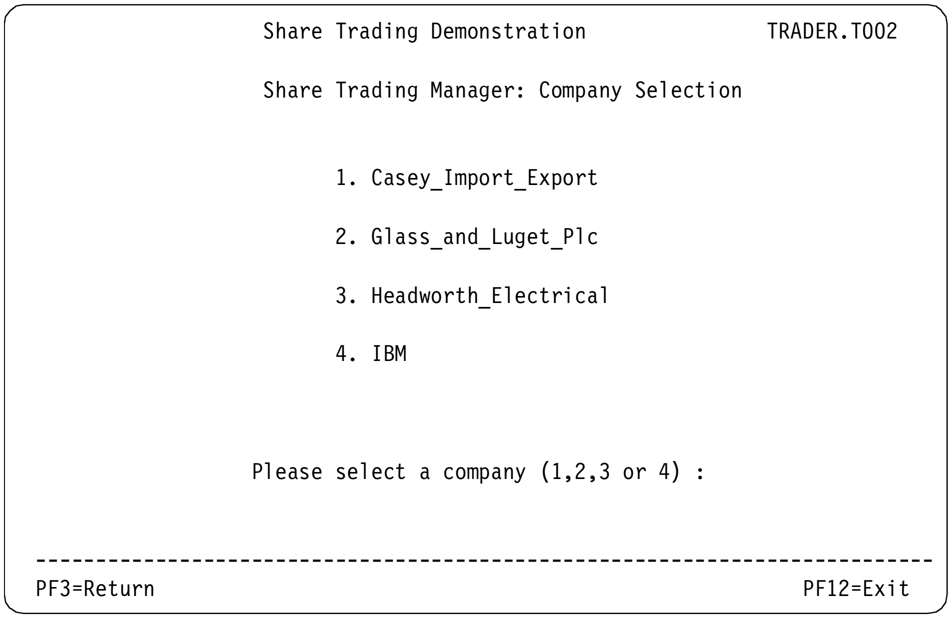
   When TRADERBL receives a *Get_Company* request, it browses the company file and returns the first four entries to TRADERPL. At this point the user has not entered any request, but the application assumes that a Get_Company request will be following. TRADERPL then sends the *signon display* (T001 shown in Figure C-4), which prompts for a user ID and password. The list of companies is stored in the COMMAREA associated with the terminal when the TRAD transaction ends, so that it will be available at the next task in the pseudo-conversational sequence.



```
                    Share Trading Demonstration          TRADER.T001

                    Share Trading Manager: Logon



              Enter your User Name:



              Enter your Password:



 ---------------------------------------------------------------------------
 PF3=Exit                                                        PF12=Exit
```

Figure C-4 Trader signon display

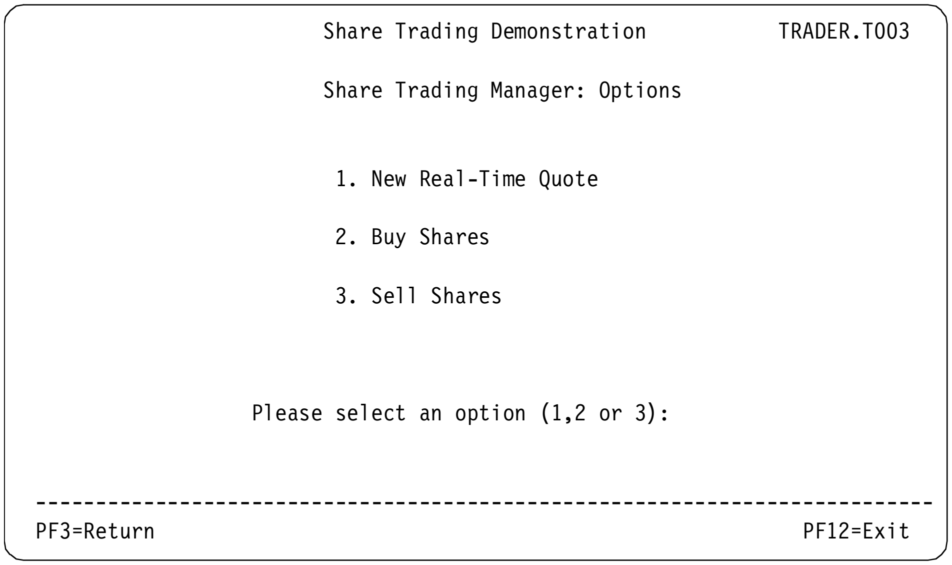
2. The next transaction invokes TRADERPL, which receives the *signon display* (T001) and the saved COMMAREA from step 1. If security is active in the CICS region the user ID and password entered will be used to signon to CICS. Then the using the company data acquired in step 1, TRADERPL sends the *company selection display* (T002), the format of which is shown in Figure C-5. TRADERPL then returns, specifying the next transaction to run and the associated COMMAREA.



```
                    Share Trading Demonstration             TRADER.T002

                    Share Trading Manager: Company Selection


                          1. Casey_Import_Export

                          2. Glass_and_Luget_Plc

                          3. Headworth_Electrical

                          4. IBM


                   Please select a company (1,2,3 or 4) :


 ---------------------------------------------------------------------------
 PF3=Return                                                      PF12=Exit
```

Figure C-5 Company selection display

3. The user selects the company to trade from the *Company Selection* display, and presses Enter. The program TRADERPL is invoked and sends the *Options display* (T003, shown in Figure C-6) to the terminal. The user can now decide whether to buy, sell, or get a new real-time quote. TRADERPL returns, specifying the next transaction to run and the associated COMMAREA.



```
                     Share Trading Demonstration         TRADER.T003

                     Share Trading Manager: Options


                      1. New Real-Time Quote

                      2. Buy Shares

                      3. Sell Shares


                 Please select an option (1,2 or 3):


 ---------------------------------------------------------------------------
 PF3=Return                                                       PF12=Exit
```

Figure C-6   Options menu display

4. The user then selects Option **1** and presses the Enter key. TRADERPL is invoked and determines that the user's request is a *Share_Value* request type. TRADERPL calls TRADERBL, passing the request type and the company selected earlier. TRADERBL reads the customer file to determine how many shares are held, then reads the company file to determine the price history, and returns the information to TRADERPL. TRADERPL uses this data to build a *Real-Time Quote* display (T004) as illustrated in Figure C-7. This display shows the recent history of share values for the company chosen, the number of shares held with this company, and the total value of these shares. TRADERPL returns, specifying the next transaction to run and the associated COMMAREA data.

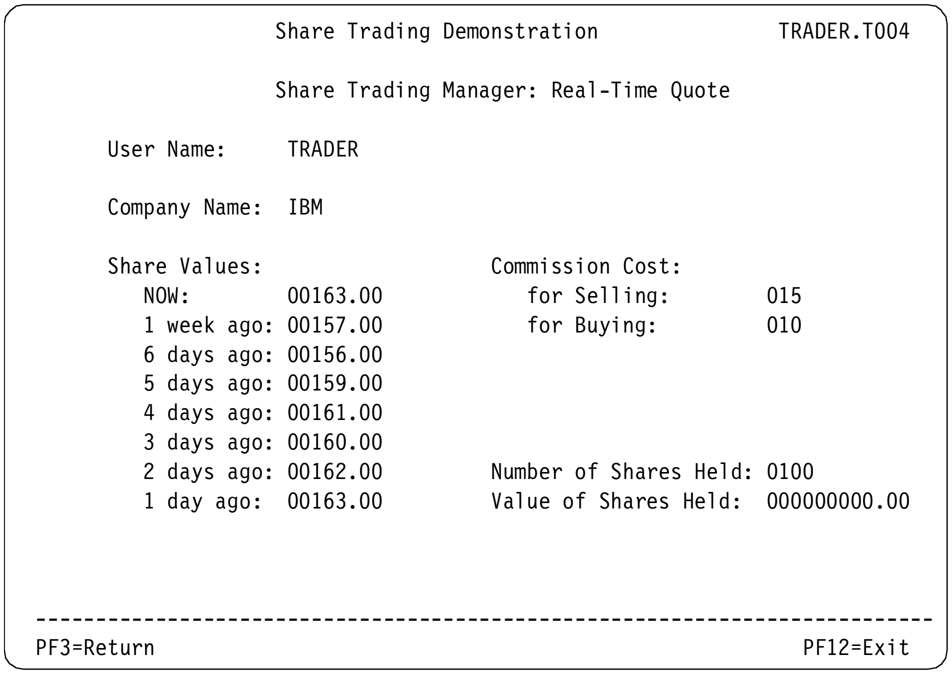
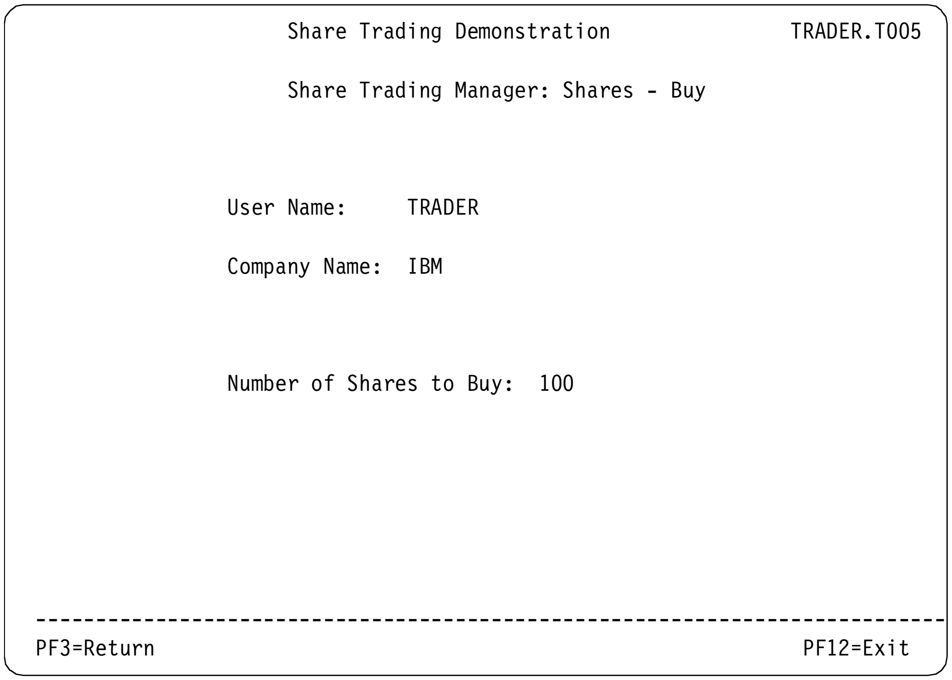

```
                      Share Trading Demonstration               TRADER.T004

                      Share Trading Manager: Real-Time Quote

         User Name:     TRADER

         Company Name:  IBM

         Share Values:                      Commission Cost:
            NOW:        00163.00               for Selling:        015
            1 week ago: 00157.00               for Buying:         010
            6 days ago: 00156.00
            5 days ago: 00159.00
            4 days ago: 00161.00
            3 days ago: 00160.00
            2 days ago: 00162.00            Number of Shares Held: 0100
            1 day ago:  00163.00            Value of Shares Held:  000000000.00



 ---------------------------------------------------------------------------
 PF3=Return                                                        PF12=Exit
```

Figure C-7 Real-time quote display

5. The user now presses **PF3** to go back to the *Options Menu*. TRADERPL is invoked and sends the *Options display* (T003) to the terminal (repeating the actions of Step 3), and returns, specifying the next transaction to run and the associated COMMAREA data.
6. The user now requires to purchase shares, so selects Option **2** and presses the Enter key. Program TRADERPL receives map T003 and determines that the user wants to buy shares, and sends the *Shares-Buy* display (T005) shown in Figure C-8. TRADERPL returns, specifying the next transaction to run and the associated COMMAREA.



```
                    Share Trading Demonstration            TRADER.T005

                    Share Trading Manager: Shares - Buy


               User Name:     TRADER

               Company Name:  IBM


               Number of Shares to Buy:  100








 ----------------------------------------------------------------------------
 PF3=Return                                                       PF12=Exit
```

Figure C-8 Shares - Buy display

7. Program TRADERPL receives the T005 screen and builds a *Buy_Sell* request COMMAREA which is passed to program TRADERBL. TRADERBL reads the company file and then performs a READ for UPDATE and REWRITE to the customer file to update the customers share holdings. The success of the request is returned to TRADERPL in the COMMAREA, and TRADERPL sends the *Options display* (T003) reporting the successful buy to the user. TRADERPL returns, specifying the next transaction to run and the associated COMMAREA.
8. Next the user checks his share holdings by repeating Step 4.
9. The user returns to the options screen by repeating Step 5.
10. The business transaction is completed by the user pressing PF12, which performs an EXEC CICS SEND TEXT to write a message to the terminal and reports that the session is complete. TRADERPL then executes the final EXEC CICS RETURN command. No COMMAREA is specified because the pseudo-conversation is over, and there is no conversation state data to retain.

CICS resource definitions

To install the COBOL Trader application the following CICS resources need to be created:

- **Files**

  Trader uses the following two VSAM files:

  – COMPFILE

    This file is used to store the list of companies and associated share prices. It can be created using the supplied JCL `TRADERCOCJL.TXT` which requires as input the file `TRADERCODATA.TXT`

  – CUSTFILE

    This file is used to store the list of users and share holdings. It can be created using the supplied JCL `TRADERCUJCL.TXT`

- **Transactions**

  The 3270 version of Trader requires just one transaction **TRAD**, which should specify the program TRADERPL

- **Programs**

  CICS program definitions are only required if program autoinstall is not active. The 3270 trader application uses two COBOL programs which will need to compiled and placed in a dataset in your CICS region DFHRPL concatenation. The COBOL source code for these applications is available with the sample source code with this book. For details on how to download this refer to Appendix D, “Additional material” on page 261.

  – **TRADERPL**

    This contains the 3270 presentation logic and is invoked by transaction TRAD.

  – **TRADERBL**

    This contains the business logic and is invoked by program TRADERPL, or it can be linked to from another application that contains its own presentation logic, such as a Java servlet.

- **Mapset**

  Trader uses the Mapset **NEWTRAD** which comprises the maps T001, T002, T003, T004, T005 and T006. The Mapset is supplied in the file `NEWTRAD.TXT` and will need to be assembled and link-edited, and the load module placed in a dataset in your CICS region DFHRPL concatenation.

For further information on creating the resource definitions for Trader, refer to the supplied file `TRADERRDO.TXT`, which contains the output of a CSD extract for the Trader application.

D

Additional material

This redbook refers to additional material that can be downloaded from the Internet as described below.

Locating the Web material

The Web material associated with this redbook is available in softcopy on the Internet from the IBM Redbooks Web server. Point your Web browser to:

`ftp://www.redbooks.ibm.com/redbooks/SG246401`

Alternatively, you can go to the IBM Redbooks Web site at:

`ibm.com``/redbooks`

Select the **Additional materials** and open the directory that corresponds with the redbook form number, SG24-6401.

Using the Web material

The additional Web material that accompanies this redbook contains the following directory structure:

- **CICS-COBOL:** CICS source used in this book, including the following programs:
  - epimap.bms
  - swapset.bms
  - eciadder.txt
  - ecidconv.txt
  - eciprog.txt
  - epiprog.txt
  - swap.txt
- The **EJB_components** directory containing the WAR, and JAR files used in Chapter 6, “CCI applications in a managed environment” on page 111.
- **Java:** Java source code used in this book includes the following packages:
  - itso.cics.eci
  - itso.cics.eci.j2ee
  - itso.cics.eci.j2ee.jndi
  - itso.cics.eci.j2ee.trader
  - itso.cics.epi
  - itso.cics.epi.j2ee
  - itso.cics.esi

  Tip: For information on how to obtain the JAR files (`cicsj2ee.jar`, `ccf2.jar` and `connector.jar`) required by some of our samples refer to Appendix A, “Configuring the CICS connectors in VisualAge for Java” on page 219.

- **Trader-COBOL:** Source code and other files for creating the Trader CICS application.
- The **VAJ_repositories** directory containing the VisualAge for Java repositories for use with Chapter 5, “CCI applications: ECI based” on page 71 and Chapter 8, “CCI applications: EPI based” on page 181. Note that respositories are used for these chapters to store Command bean meta information, which is not stored in the provided Java source code.

System requirements for downloading the Web material

The following system configuration is recommended:

Hard disk space:	2 MB minimum
Operating System:	Windows NT, or 2000.
Processor:	Intel 486 or higher
Memory:	64 MB

How to use the Web material

Create a subdirectory (folder) on your workstation, and unzip the contents of the Web material Zip file into this folder. This will create the directories `CICS-COBOL`, `Trader-COBOL` and `Java` containing the samples and read-me files with further instructions.

Abbreviations and acronyms

AOR	application-owning region
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
AWT	abstract windowing toolkit
BMP	bean-managed persistence
CCI	Common Client Interface
CCF	Common Connector Framework
CMP	container managed persistence
CORBA	Component Object Request Broker Architecture
CTG	CICS Transaction Gateway
CWS	CICS Web support
DBMS	database management system
DNS	Domain Name System
DPL	distributed program link
EAB	Enterprise Access Builder
EAR	Enterprise Application Archive
EBCDIC	Extended Binary Coded Decimal Interchange Code
ECI	External Call Interface
EJB	Enterprise JavaBeans
EJS	Enterprise Java Server
EIS	Enterprise Information Systems
EPI	External Presentation Interface
ESI	External Security Interface
ESM	External Security Manager
EXCI	External CICS Interface
FOR	file-owning region
FTP	File Transfer Protocol
GIOP	General Inter-ORB Protocol
GUI	graphical user interface
HFS	Hierarchical File System
HTML	Hypertext Transfer Protocol
HTTP	Hypertext Markup Language
IDE	Integrated development environment
IDL	interface definition language
IIOP	Internet Inter-ORB Protocol
IOR	interoperable object reference
ISC	inter-system communication
J2EE	Java 2 Enterprise Edition
JAR	Java archive
JDBC	Java Database Connectivity
JDK	Java Developer's Kit
JNDI	Java Naming and Directory Interface
JNI	Java Native Interface
JPDA	Java Platform Debugger Architecture
JSDK	Java Servlet Development Kit
JSP	Java Server Page
JVM	Java Virtual Machine
LDAP	Lightweight Directory Access Protocol
LPAR	logical partition
LUW	logical unit of work
OMG	Object Management Group
OS/390	operating system 390
OTS	object transaction service
PB	Persistence Builder

PEM	Password Expiration Managment
RDBMS	relational database management system
RMI	Remote Method Invocation
RPC	remote procedure call
SDK	Software Development Kit
SQL	structured query language
SQLJ	SQL Java
SSL	secure socket layer
TCB	task control block
TCP/IP	Transmission Control Protocol/Internet Protocol
UML	Unified Modeling Language
UOW	unit of work
URL	Uniform Resource Locator
USS	Unix System Services
VAJ	VisualAge for Java
WAR	Web Application Archive
WTE	WebSphere Test Environment
XMI	XML metadata interchange
XML	Extensible Markup Language

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

IBM Redbooks

For information on ordering these publications, see “How to get IBM Redbooks” on page 268.

- *Revealed! CICS Transaction Gateway with More CICS Clients Unmasked,* SG24-5277
- *CCF Connectors and Databases Connections using WebSphere Advanced Edition*, SG24-5514
- *Revealed! Architecting Web Access to CICS*, SG24-5466
- *CICS Transaction Gateway V3.1 The WebSphere Connector for CICS,* SG24-6133
- *Enterprise JavaBeans for z/OS and OS/390, CICS Transaction Server V2.1*, SG24-6284.

Other resources

These Web sites are relevant as further information sources:

- CICS Transaction Gateway software downloads
  `http://www.ibm.com/software/ts/cics/downloads`
- CICS Transaction Gateway library
  `http://www-3.ibm.com/software/ts/cics/library/manuals/index40.html`
- WebSphere Application Server InfoCenter
  `http://www.ibm.com/software/webservers/appserv/infocenter.html`

Referenced Web sites

These Web sites are also relevant as further information sources:

- Sun J2EE connector specification
  `http://java.sun.com/j2ee/download.html#connectorspec`

- Whitepaper: Using J2EE Resource Adapters in a Non-managed Environment
  `http://www7b.boulder.ibm.com/wsdd/library/techarticles/0109_kelle/0109_kelle.html`
- Connector Architecture for WebSphere Application Server
  `http://www6.software.ibm.com/dl/connarch/connarch-p`
- Coexistence of WebSphere Studio Application Developer with VisualAge for Java
  `http://www7b.boulder.ibm.com/wsdd/library/techarticles/0110_searle/searle.html`
- WebSphere Studio Application Developer Integration Edition
  `http://www.ibm.com/software/ad/studiointegration/`
- CICS announcement letters
  `http://www-4.ibm.com/software/ts/cics/announce/`

How to get IBM Redbooks

You can order hardcopy Redbooks, as well as view, download, or search for Redbooks at the following Web site:

ibm.com/redbooks

You can also download additional materials (code samples or diskette/CD-ROM images) from that site.

IBM Redbooks collections

Redbooks are also available on CD-ROMs. Click the CD-ROMs button on the Redbooks Web site for information about all the CD-ROMs offered, as well as updates and formats.

Index

Redbooks

Java Connectors for CICS: Featuring the J2EE Connector Architecture

(0.5” spine)
0.475”<->0.875”
250 <-> 459 pages

Java Connectors for CICS

Featuring the J2EE Connector Architecture

Use the CICS J2EE resource adapters and deploy applications to WebSphere

Develop applications using the Enterprise Access Builder

Understand the J2EE CCI and the CTG base classes

What is the best method for connecting a Java application to CICS? There are a wealth of options that are available, ranging from using the Java class libraries that are shipped with the CICS Transaction Gateway (CTG), to using the Common Client Interface (CCI) component of the Java 2 Enterprise Edition (J2EE) Connector Architecture. There are also important application development choices to make, such as whether to code to an API directly, or to use a tool such as VisualAge for Java's Enterprise Access Builder.

This IBM Redbook examines the strategic Java connection methods for CICS. The focus is on the use of the J2EE Connector Architecture, which is a new Java standard for connecting to legacy Enterprise Information Systems such as CICS. This builds upon the previous IBM Common Connector Framework (CCF) and provides enhanced facilities for deploying into a managed environment, where connection pooling, transactions, and security are managed by a J2EE capable application server such as WebSphere Application Server.

We provide comprehensive code examples for the J2EE CCI, as well as the CTG base classes, including the ECIRequest, ESIRequest, and the EPI support classes.

SG24-6401-00 ISBN 0738423904