IBM

Developing Connector Applications for CICS

Archived

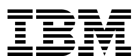
- Java, J2C and C ECIv2 interfaces revealed
- Tooling to build and deploy J2C based clients
- Sample client programs

G Michael Connolly
John Kurian
Lokanadham Nalla
Andrew Smithson
Dennis Weiand

Redbooks

ibm.com/redbooks

Archived



International Technical Support Organization

Developing Connector Applications for CICS

April 2009

Archived

SG24-7714-00

Note: Before using this information and the product it supports, read the information in "Notices" on page vii.

Archived

First Edition (April 2009)

This edition applies to Version 7, Release 2 of CICS Transaction Gateway for z/OS.

© Copyright International Business Machines Corporation 2009. All rights reserved.
Note to U.S. Government Users Restricted Rights -- Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Notices . . . 1
Trademarks . . . 2

Preface . . . 1
The team that wrote this book . . . 1
Become a published author . . . 3
Comments welcome . . . 3

Chapter 1. Introduction to CICS Transaction Gateway . . . 1
1.1 The CICS Transaction Gateway . . . 2
1.1.1 CICS TG Products . . . 2
1.2 CICS Transaction Gateway Components . . . 3
1.2.1 Gateway daemon . . . 4
1.2.2 Client daemon . . . 5
1.2.3 IPIC Driver . . . 5
1.2.4 Configuration tool . . . 5
1.2.5 Resource adapters . . . 6
1.3 Local and remote mode configurations . . . 10
1.4 CICS TG Clients . . . 11
1.4.1 Local CICS TG Clients . . . 11
1.4.2 Remote CICS TG Clients . . . 12
1.4.3 Additional benefits of remote mode . . . 12
1.5 CICS TG supported protocols . . . 13
1.5.1 Gateway daemon protocol handlers . . . 13
1.5.2 Connecting to the CICS TS . . . 13
1.6 CICS TG topologies . . . 14

Chapter 2. Developing CICS TG Applications . . . 23
2.1 Programming interfaces . . . 24
2.1.1 External Call Interface (ECI) . . . 24
2.1.2 External Presentation Interface (EPI) . . . 25
2.1.3 External Security Interface (ESI) . . . 25
2.2 Programming languages . . . 26
2.2.1 Java applications . . . 26
2.2.2 JCA applications . . . 29
2.2.3 C applications - Client API . . . 34
2.2.4 C applications: ECI v2 . . . 36
2.2.5 C++ applications . . . 40
2.2.6 COBOL applications . . . 42

Archived

© Copyright IBM Corp. 2009. All rights reserved.

2.2.7 .NET client applications . . . 44
2.3 Summary . . . 48

Chapter 3. Channels and Containers . . . 49
3.1 Overview . . . 50
3.2 Channels and containers background . . . 50
3.2.1 Channel and container data formats . . . 51
3.2.2 Channel and container data flows . . . 51
3.3 Character Conversion background . . . 52
3.3.1 Double byte character representation . . . 52
3.3.2 Unicode character representation . . . 53
3.3.3 UTF-8 character representation . . . 53
3.3.4 Character representation using Java . . . 53
3.3.5 Coded Character Set Identifier (CCSID) . . . 54
3.4 Channel and container data conversion . . . 54
3.5 Using the CICS TG channel and container API . . . 56
3.5.1 Using channels and containers in a Java client . . . 57
3.5.2 Browsing containers in a Java clients using the CCI . . . 59
3.5.3 Using channels and containers in Java clients using the CICS TG base classes . . . 60
3.5.4 Browsing containers in a Java clients using the CICS TG base classes 62
3.6 Use of channels and containers in a CICS COBOL Program . . . 63
3.7 Conversion problems . . . 64

Chapter 4. Java-based Sample Application . . . 65
4.1 Java application setup . . . 66
4.1.1 Software pre-requisites . . . 66
4.1.2 Configuring the CICS TG . . . 66
4.2 COMMAREA-based applications . . . 67
4.2.1 Sample COMMAREA-based Java client development . . . 67
4.2.2 Sample COMMAREA-based JCA client development . . . 74
4.3 Channel-based applications . . . 83
4.3.1 Sample channel-based Java client development . . . 83
4.3.2 Sample channel-based JCA client development . . . 93

Chapter 5. J2C Application Development With Rational Application Developer . . . 105
5.1 Rational Application Developer . . . 106
5.2 Getting Started . . . 106
5.3 Sample application . . . 108
5.3.1 Generate data binding classes . . . 110
5.3.2 Generate a J2C bean to access CICS . . . 115
5.3.3 Adding a J2C bean method . . . 126

Archived

5.3.4 Creating a WebService from the J2C bean 130
5.4 Developing with channels and containers 140

Chapter 6. Developing a C application using ECIv2 149
6.1 Introduction to the ECIv2 API 150
6.1.1 Current restrictions using ECIv2 API. 150
6.1.2 Characteristics of the ECIv2 API. 150
6.1.3 Documentation 151
6.1.4 Overview of an ECIv2 application 151
6.2 Sample CICS C application overview 151
6.2.1 The COMMAREA structure. 152
6.2.2 Data conversion 154
6.3 Developing the ECIv2 sample application. 155
6.3.1 Sample client application overview 155
6.3.2 The application development environment 157
6.3.3 Connecting to the CICS Transaction Gateway 159
6.3.4 Communicating with the CICS servers 161
6.3.5 Calling the CICS back-end program 162
6.3.6 Closing the ECI connection. 167
6.4 Problem determination 168
6.4.1 CICS server reported errors 168
6.4.2 Application errors 168

Chapter 7. Migrating a Client application to ECIv2 171
7.1 Introduction to application migration 172
7.2 Changing the header files 172
7.3 Managing the Gateway daemon connection 173
7.3.1 Opening connections to the Gateway daemon 173
7.3.2 Working with a connection token 173
7.3.3 Closing connections to the Gateway daemon. 174
7.4 Migrating Client API function calls. 175
7.4.1 Listing available systems 175
7.4.2 ECI parameters 176
7.4.3 Calling the CICS program 177
7.5 Compiling the application 178

Appendix A. Data conversion techniques. 181
Conversion within Java 182
Character data 182
Numeric data. 184
Conversion within CICS: DFHCNV templates 186
Character data 186
Numeric data. 189

Appendix B. Java Sample Code 191
CustProgGetSingleBaseClasses 192
CustProgGetSingleJ2C 197
CustProgMultiRecBaseClasses 202
CustProgMultiRecJ2C 212
B.1 CustProgCommarea 220

Appendix C. Sample ECIv2 client 227
custbrowse.c 228

Appendix D. CICS samples 243
CICS COBOL server program components 244
Executing the COMMAREA-based server program 244
Error situations 245
To Read 245
To Delete 245
To Add 245
To Update 245
Installing the COMMAREA-based sample program on CICS 246
Executing the Channel-based server program 247
Installing the Channel-based sample application on CICS 248
. . . . Compiling and Executing Java Sample Programs 249

Appendix E. Additional material 253
Locating the Web material 253
Using the Web material 254
How to use the Web material 254

Related publications 255
IBM Redbooks 255
Other publications 255
Online resources 255
How to get Redbooks 256
Help from IBM 256

Archived

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:
IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785 U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

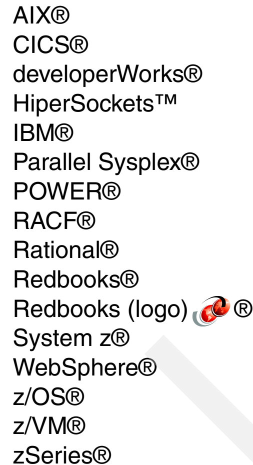
Archived

© Copyright IBM Corp. 2009. All rights reserved.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. These and other IBM trademarked terms are marked on their first occurrence in this information with the appropriate symbol (® or ™), indicating US registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at http://www.ibm.com/legal/copytrade.shtml

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

AIX®
CICS®
developerWorks®
HiperSockets™
IBM®
Parallel Sysplex®
POWER®
RACF®
Rational®
Redbooks®
Redbooks (logo)  ®
System z®
WebSphere®
z/OS®
z/VM®
zSeries®

The following terms are trademarks of other companies:

Interchange, and the Shadowman logo are trademarks or registered trademarks of Red Hat, Inc. in the U.S. and other countries.

SAP, and SAP logos are trademarks or registered trademarks of SAP AG in Germany and in several other countries.

EJB, J2EE, Java, Javadoc, JVM, Solaris, Sun, and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft, Visual Basic, Windows Vista, Windows, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Intel, Itanium, Intel logo, Intel Inside logo, and Intel Centrino logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

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

Archived

Preface

This IBM® Redbooks® publication is intended for the CICS® application programmer who is creating client applications requiring access to CICS back-end resources. This Redbooks publication presents the latest features available with the CICS Transaction Gateway V7.2, including interoperation with the CICS Transaction Server for z/OS® V3.2 channels, and the containers programming model, to allow J2EE™ applications to exchange large amounts of data with CICS programs.

Easy to understand sample client applications are used to demonstrate the programming techniques in base Java™, J2C, and C. Additionally, the use of the Rational® Application Developer tools for building and deploying J2C client applications is presented.

The team that wrote this book

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

G Michael Connolly is an IT consultant at the ITSO, Poughkeepsie Center. He has more than 30 years of IBM software development experience in both distributed systems and the mainframe zSeries®. His areas of expertise include TCP/IP communications, UNIX® System Services, and WebSphere® Application Server for z/OS. He holds a BA in Humanities from Villanova University.

John Kurian is an Advisory Software Engineer in IBM India Software Labs, Bangalore. He has 5 years of experience in TXSeries For Multiplatform (CICS for distributed platforms) field. His areas of expertise include SQL Programming and Java technologies. He holds a Bachelors degree in Applied Electronics and Instrumentation from University of Kerala, India.

Lokanadham Nalla is a Software Engineer in IBM India Software Labs, Bangalore. He has 4 years of experience with the CICS Transaction Gateway. His areas of expertise include Web technologies, Java and J2EE development. He holds Masters of Computer Applications degree from Nagarjuna University, Guntur, AP.

Archived

© Copyright IBM Corp. 2009. All rights reserved.

Andrew Smithson is a Software Engineer in IBM UK in Hursley. He has 5 years of experience with the CICS Transaction Gateway. His areas of expertise include Java, C and J2EE development, security and SSL. He holds a Masters degree in Software Engineering from the University of Southampton.

Dennis Weiand is an Technical Sales Specialist at the IBM Dallas Systems Center. Currently, Dennis works primarily with Web services and Java as they relate to CICS, plus the CICS Transaction Gateway. He holds a masters degree in Computer Science from Tarleton State University, Central Texas.

Figure 0-1 The Redbooks team, from left to right: John Kurian, Lokanadham Nalla, G Michael Connolly, Andrew Smithson

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

Rich Conway, Peter Bertolozzi
International Technical Support Organization, Poughkeepsie Center

Phil Wakelin, CICS Transaction Gateway, Technical Planner
IBM Hursley, UK

Rob Jones, CICS Transaction Gateway, Developer
IBM Hursley, UK

Nigel Williams, STG Sales
IBM Montpellier, France

Archived

Become a published author

Join us for a two- to six-week residency program! Help write a book dealing with specific products or solutions, while getting hands-on experience with leading-edge technologies. You will have the opportunity to team with IBM technical professionals, Business Partners, and Clients.

Your efforts will help increase product acceptance and customer satisfaction. As a bonus, you will develop a network of contacts in IBM development labs, and increase your productivity and marketability.

Find out more about the residency program, browse the residency index, and apply online at:

ibm.com/redbooks/residencies.html

Comments welcome

Your comments are important to us!

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

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

  ibm.com/redbooks

- Send your comments in an e-mail to:

  redbooks@us.ibm.com

- Mail your comments to:

  IBM Corporation, International Technical Support Organization
  Dept. HYTD Mail Station P099
  2455 South Road
  Poughkeepsie, NY 12601-5400

Archived

Archived

1

Introduction to CICS Transaction Gateway

In this chapter, we cover the key concepts of the CICS Transaction Gateway (TG).

Details are provided covering the following topics:

- CICS TG components
- Local and remote mode configurations
- CICS TG supported protocols
- CICS TG topologies
- CICS TG application programming interfaces
- Supported remote and local client applications.

Archived

© Copyright IBM Corp. 2009. All rights reserved.

1.1 The CICS Transaction Gateway

The CICS Transaction Gateway (TG) is a set of client and server software components that allow a remote client application to invoke services in a CICS region. The client application can be either a Java application or a non-Java application. Depending on the platform the non-Java application may use either a C (ECIv1 and ECIv2), C++, COBOL, COM, or .NET interface.

1.1.1 CICS TG Products

The CICS TG is available through the following two orderable products:

- CICS TG on z/OS

  CICS TG on z/OS v7.2 is the latest version for the z/OS platform. It is supported on z/OS V1R8 and later and supports connectivity to a CICS TS on z/OS V2.3 or later release.

  The CICS TG on z/OS provides comprehensive support for two phase-commit transactions along with support for SSL connections, high availability topologies, and advanced systems monitoring. It uses either the external communication interface (EXCI) or the IP Interconnectivity (IPIC) protocol to communicate with CICS regions on z/OS. It provides API support for Java and non-Java applications making ECI calls to COMMAREA or channel- and container-based CICS applications. More information about ECIV2 C applications can be found in Chapter 6, “Developing a C application using ECIv2” on page 149.

- CICS TG for Multiplatforms

  CICS TG on Multiplatforms V7.2 is supported on the following range of operating systems and platforms and is designed to support connectivity to any release of CICS on any platform:

  – Linux® on System z®
  – Linux on Intel®
  – Linux on POWER®
  – AIX®
  – HP-UX (on PA-RISC and Itanium®)
  – Sun™ Solaris™ (on SPARC)
  – Windows® XP, Windows 2003 and Windows Vista®

1.2 CICS Transaction Gateway Components

This section describes the key components of the CICS TG product. Figure 1-1 shows the components of CICS TG for Multiplatforms product and its positioning in the product architecture. The resource adapters are shipped as part of the CICS TG product for deployment to a WebSphere Application Server in support of JCA clients.

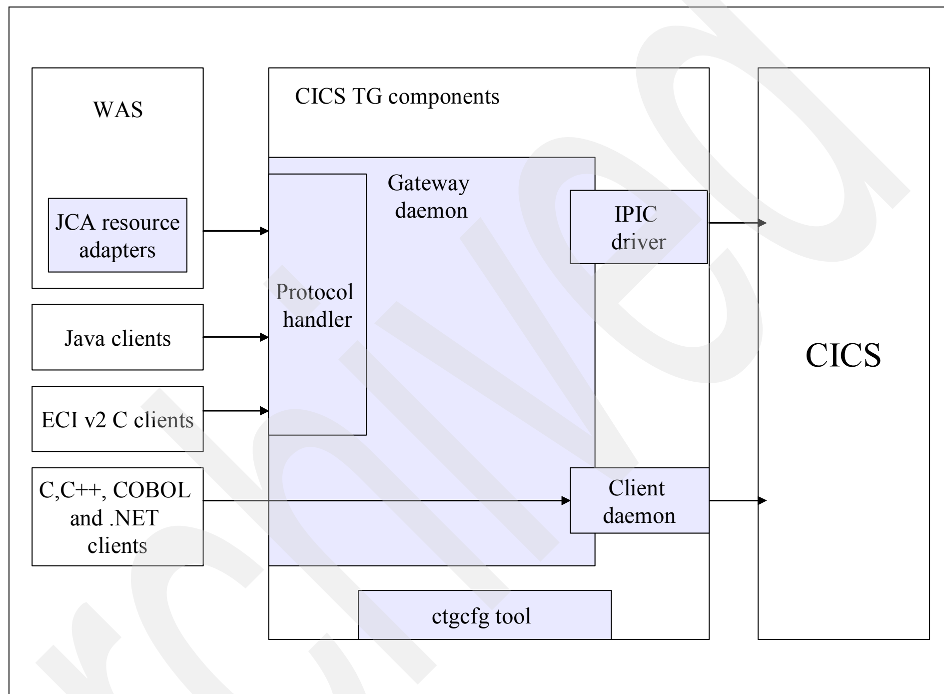


Figure 1-1 Components of CICS TG for Multiplatforms

Archived

The CICS TG for z/OS platform, unlike the distributed platforms, does not include the Client daemon. The CICS TG on z/OS does not require a Client daemon because it communicates with the CICS TS using either the IPIC or EXCI functions provided by CICS TS. Figure 1-2 shows the components of CICS TG on z/OS.

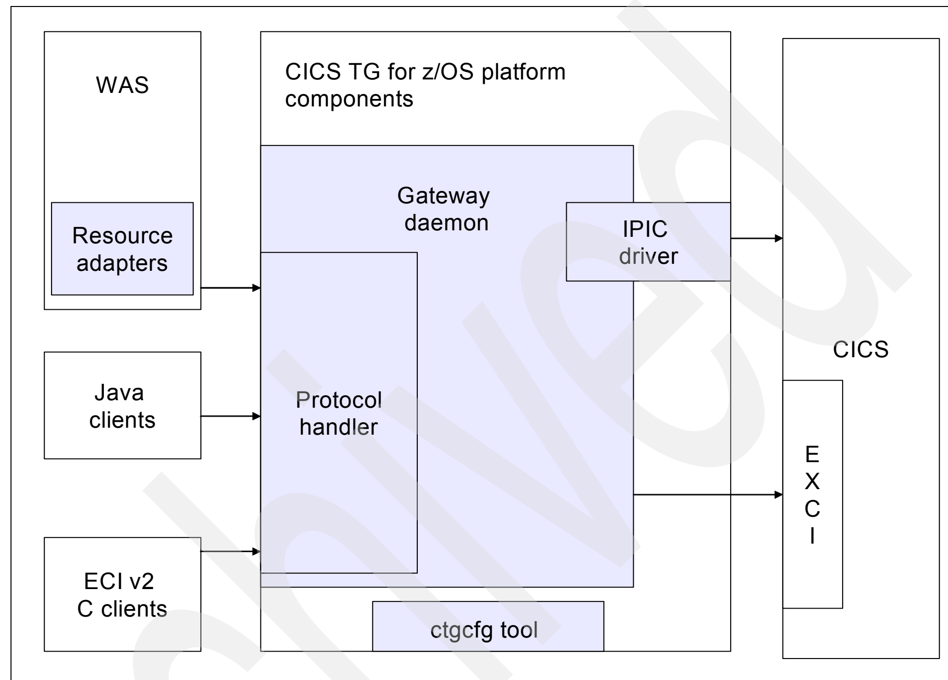


Figure 1-2 Components of CICS TG for z/OS platforms

1.2.1 Gateway daemon

The Gateway daemon listens for incoming remote client requests and manages the threads and connections necessary to ensure good performance. The performance and the features provided by the Gateway daemon running on the z/OS platform differ from the Gateway daemon running on distributed platforms. The Gateway daemon handles remote client requests using its protocol handlers component. Gateway daemon can be configured to support both TCP/IP and SSL protocols. You can configure these protocols and set Gateway daemon specific parameters using the Configuration tool, which is discussed in 1.2.4, “Configuration tool” on page 5.

Archived

1.2.2 Client daemon

The Client daemon is an integral part of CICS TG on all distributed platforms. The Client daemon provides the local interface to C, C++, COM, and COBOL client applications to enable communication with CICS. It provides SNA and TCP/IP protocol connectivity to CICS using the same technology as IBM CICS Universal client. You can configure these protocols with other protocol-specific parameter values using the CICS TG Configuration tool (ctgcfg).

1.2.3 IPIC Driver

The CICS TS for z/OS, Version 3.2 introduced the IPIC protocol, used for intercommunication with CICS regions over TCP/IP. CICS TG V7.1 and later supports the IPIC protocol for making ECI calls to CICS and also includes support for containers and channels, SSL, and two-phase commit. The IPIC support is provided in the CICS TG as a separate component called the IPIC protocol driver. This driver is independent of the Client daemon component. The IPIC configuration for local mode is different from local mode configuration supported by other protocols with Client daemon. More information about local and remote mode configuration is available in 1.3, "Local and remote mode configurations" on page 10.

The following limitations apply when connecting using the IPIC protocol.

- You cannot use the EPI or ESI Application programming interfaces.
- You cannot use the ECI asynchronous calls.
- You cannot use the `cicscli` command for CICS Servers definitions.

You can find more information about the limitations of ECI calls at the CICS TG Infocenter (Programming Guide → Programming in Java → Making External Call Interface calls from Java Client Program → IPIC support for ECI).

1.2.4 Configuration tool

We used the ctg.ini configuration file located in the CICS TG install bin folder for configuring the Gateway daemon and the protocols for the CICS TG. The CICS TG product provides the ctgcfg utility for updating the values within the ctg.ini configuration file. It is suggested to use the ctgcfg utility to modify the property values in the ctg.ini file.

The ctgcfg utility can be used for the following activities:

- CICS TG APPLID, APPLID qualifier, and the default server can be configured in CICS TG main section.
- In the Gateway daemon section, we can configure gateway-supported protocols TCP/IP and SSL. We can also configure Gateway daemon-specific resources, logging, and monitoring facilities.
- In the Client daemon section, we can configure the Client daemon-specific resource and the logging facilities. We can also configure the Workload manager (WLM)-specific parameters. WLM is supported only on the Windows platform.
- In the CICS Servers section, we can add new server connection definitions to CICS by right-clicking the CICS Servers in the leftside panel of the ctgcfg tool. You can select TCP/IP, SNA, IPIC, and named pipes (supported only on the Windows platform) protocols to connect to CICS.

1.2.5 Resource adapters

Resource adapters are used in support of J2EE Connector Architecture (JCA) client applications. To understand resource adapters, it is important to understand JCA.

Purpose of JCA

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

> **Note:** The complete JCA specification 1.5 defined by the Java Community Process can be downloaded at the following Web page:
>
> `http://java.sun.com/j2ee/connector/`

The connector architecture enables an EIS vendor to provide a standard resource adapter. A resource adapter is a middle-tier between a Java application and an EIS, which enables the Java application to connect to the EIS. A resource adapter plugs into application servers supporting the JCA.

Components of JCA

JCA defines a number of components that make up this architecture.See Figure 1-3 on page 8.

Archived

- Common Client Interface (CCI)

  The CCI defines a common API for interacting with resource adapters. It is independent of a specific EIS. A Java application interfaces with the resource adapter using this API.

- System contracts

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

  – Connection management
  – Transaction management
  – Security management

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

- Resource adapter deployment and packaging

  A resource adapter provider includes a set of Java interfaces/classes as part of the resource adapter implementation. These 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 Archive is used to deploy the resource adapter into the application server.

Archived

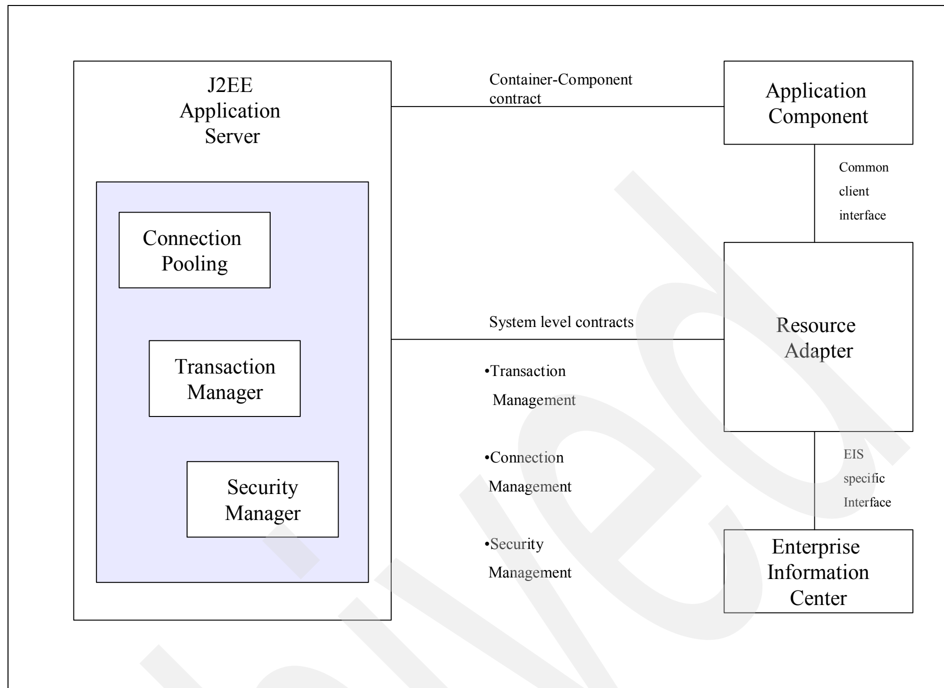


Figure 1-3 JCA Components

Resource adapters are used by JCA client applications. JCA client applications can be run in both managed and non-managed environments.

In the case of a managed environment, the resource adapter is deployed to a WebSphere Application Server. The JCA system contracts (such as connection pooling, transaction management, and security features) are now handled by the WebSphere Application Server, allowing the application developer to concentrate on implementing the business logic.

In a non-managed environment, a client application directly invokes the resource adapter without the support of a WebSphere Application Server. The client application must provide the logic to handle the management of connections, transactions, security, and business logic.

Resource adapters are located in the <CICS TG product install dir>/deployable folder.

ECI Resource Adapter

The ECI resource adapter (cicseci.rar), supplied with the CICS TG for multiplatforms product, provides only single-phase commit transaction support.

The resource adapter supplied with by the CICS TG for z/OS platform provides single-phase transaction support when deployed into WebSphere Application Server on a distributed platform and two-phase transaction support when deployed into a WebSphere Application Server running on a z/OS platform.

ECI XA Resource Adapter

ECI XA resource adapter (cicseciXA.rar) supports two-phase transactional support for the following WebSphere Application Server configurations on any of the supported platforms:

- With CICS TG for z/OS
- With CICS TG Multiplatforms when the IPIC protocol is used in local mode.

The cicseciXA.rar can also support single-phase commit.

EPI resource adapter

The EPI resource adapter (cicsepi.rar) is supplied only with the CICS TG for multiplatforms. It is used for EPI-based JCA client applications.

Archived

1.3 Local and remote mode configurations

The CICS TG supports two modes of operation called local and remote. The mode you use depends on your application topology. If the client application executes on the same server where the CICS TG is running, then local mode is appropriate. If the client application executes on a different server from where the CICS TG is running, then only remote mode is supported. Figure 1-4 shows the local and remote modes configurations.

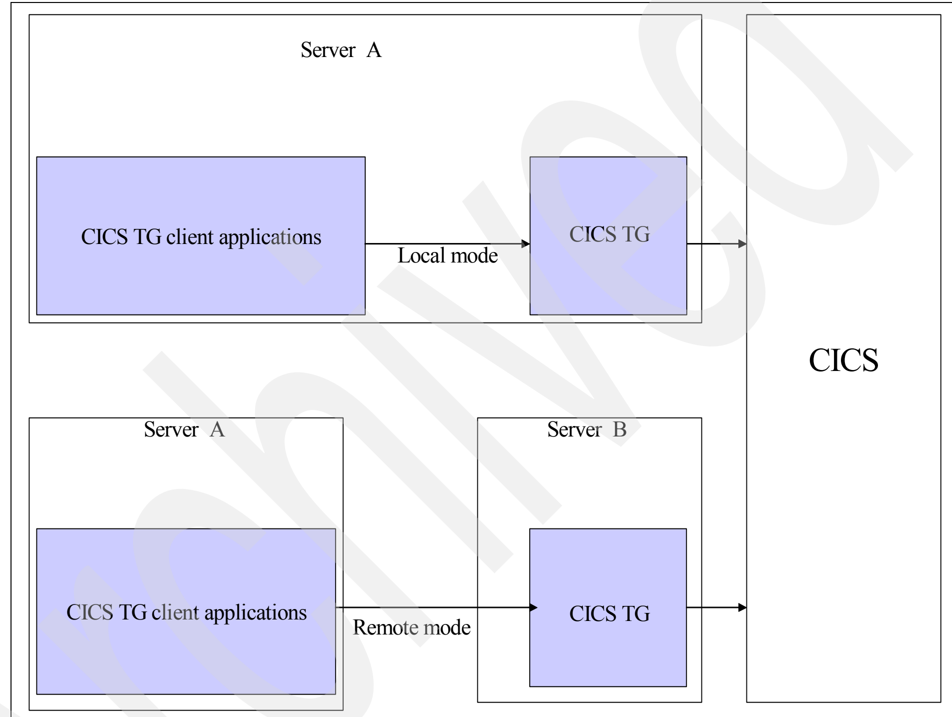


Figure 1-4 CICS TG local and remote mode configurations.

In a local mode configuration there is no need to start the Gateway daemon, however you must set the URL parameter of the ECI request calls to local.

For a remote mode configuration you must configure the Gateway daemon protocol and start the Gateway daemon using the **`ctgstart`** command. The **`ctgstart`** command includes parameters that permit you to override the Gateway daemon properties previously configured using the Configuration tool. Additional details are available for the **`ctgstart`** command by entering **`ctgstart -?`** Make sure that the IP Address or host name is included in the URL parameter of the ECI request calls.

1.4 CICS TG Clients

The CICS TG supports both local and remote clients. The following sections describe the programming languages used for making CICS TG API calls. The difference between local and remote client applications are shown in Figure 1-5. Local clients needs to be collocated on the same server where the CICS TG is installed. Generally, remote clients will be located on a server other than the server hosting the CICS TG.

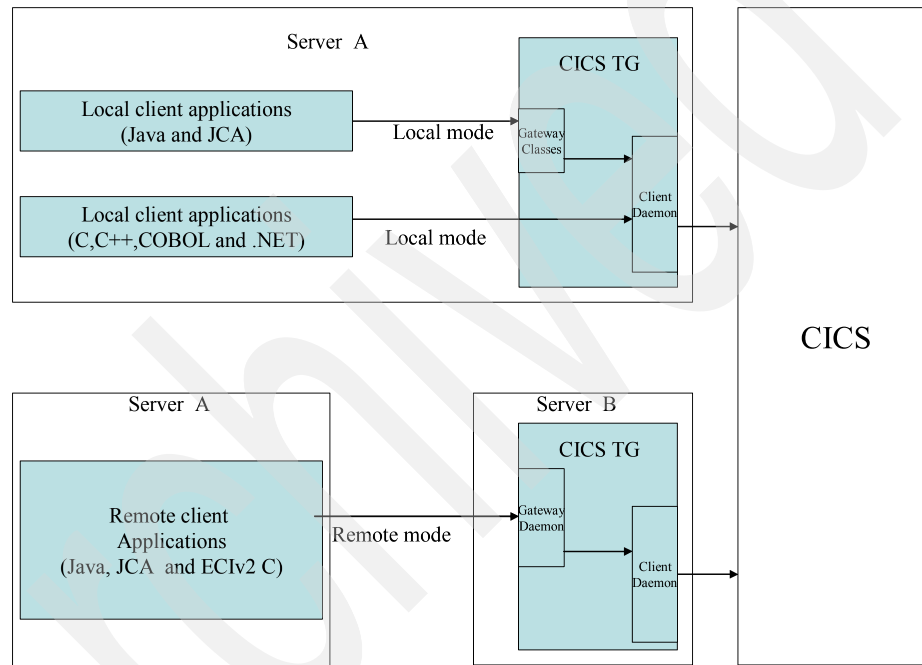


Figure 1-5 Topology of local and remote client applications

1.4.1 Local CICS TG Clients

Local non-Java CICS TG client applications do not use the Gateway daemon. These applications communicate directly with the Client daemon on distributed platforms and with EXCI when communicating with a CICS TG running on a z/OS platform. Local Java client applications do not require the Gateway daemon but use the Gateway classes provided by CICS TG to communicate with a Client daemon on distributed platforms and EXCI when communicating with a CICS TG

on a z/OS platform. The CICS TG supports the following client application types as local clients:

- C and COBOL client applications
- C++ client applications
- COM client applications
- JCA client applications
- Java client applications

1.4.2 Remote CICS TG Clients

Remote CICS TG clients communicate with the Gateway daemon. The CICS TG supports the following client applications as remote clients:

- JCA client applications
- Java client applications
- ECIv2 C client applications

> **Note:** Java and JCA client applications are supported in local mode if the client application and CICS TG are on the same server. However, they will not be able to use the features provided by the Gateway daemon.

1.4.3 Additional benefits of remote mode

Remote mode provides the following additional Gateway daemon capabilities:

- Gateway daemon statistics and monitoring facilities.
- Network concentration for multiple remote clients.
- SSL encryption into z/OS.
- High availability support.

> **Note:** Starting with CICS TG v7.2, when using remote mode, C applications can use these Gateway daemon capabilities.

Archived

1.5 CICS TG supported protocols

In this section, we describe the protocols used to communicate with the CICS TS.

1.5.1 Gateway daemon protocol handlers

Remote client applications communicate with the Gateway daemon using the following protocols

- TCP/IP
- SSL

These protocol handlers can be configured using the Configuration tool. By default the TCP/IP port is bound to 2006 and the SSL port is bound to 8050. Both server authentication and client authentication mechanisms can be used with the SSL protocol.

1.5.2 Connecting to the CICS TS

The following can be used to connect to the CICS TG:

- TCP/IP
- SNA
- IPIC
- Namedpipe (supported only with CICS TG on Windows).
- EXCI (function supported only with CICS TG on z/OS)

These protocols are configured in the CICS Servers section of the Configuration tool. When using the IPIC protocol in a local mode configuration you do not need to configure the server connection definition in the ctg.ini file. This is because the connection definition is included in the server name parameter on the ECI request call.

The External CICS Interface (EXCI) is a function provided by the CICS TS not the CICS TG. The EXCI function is available only when the CICS TG is installed on a z/OS platform.

There are limitations when using the CICS TG APIs. A complete list is available at the CICS TG for Multiplatforms Infocenter: CICS Transaction Gateway for Windows (and UNIX and Linux) → Planning → Communication with CICS Servers.

Archived

1.6 CICS TG topologies

In this section we cover CICS Transaction Gateway topologies, including the use of JCA clients.

Some of the more common topologies are shown in the Figure 1-6.

- Topology 1

  The WebSphere Application Server and CICS TG are both deployed on a distributed platform.

- Topology 2

  The WebSphere Application Server is deployed on a distributed platform and the CICS TG is deployed on a z/OS system.

- Topology 3

  Both the WebSphere Application Server and the CICS TG are deployed on System z.

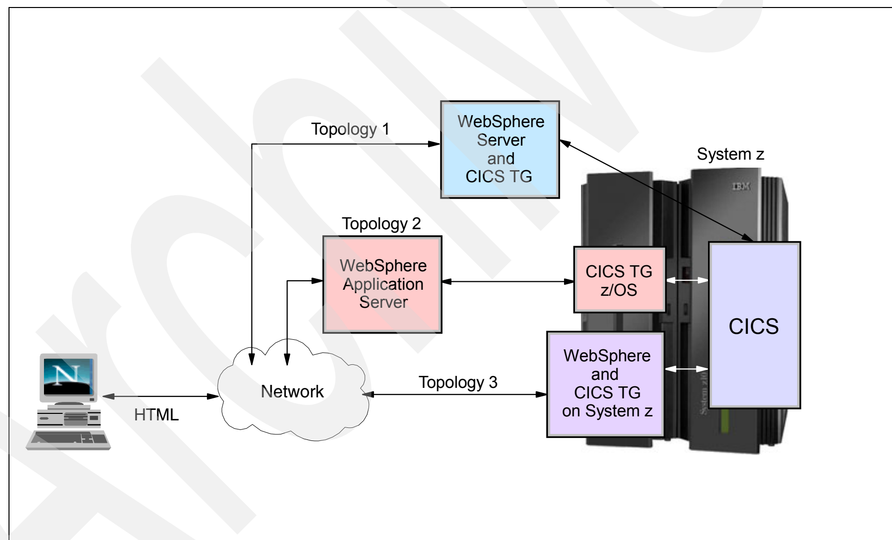


Figure 1-6 CICS TG topologies

Archived

WebSphere Application Server and CICS TG on a distributed platform

In this topology, both the WebSphere Application Server and the CICS TG are deployed on one of the distributed platforms, such as AIX or Linux as shown in Figure 1-7.

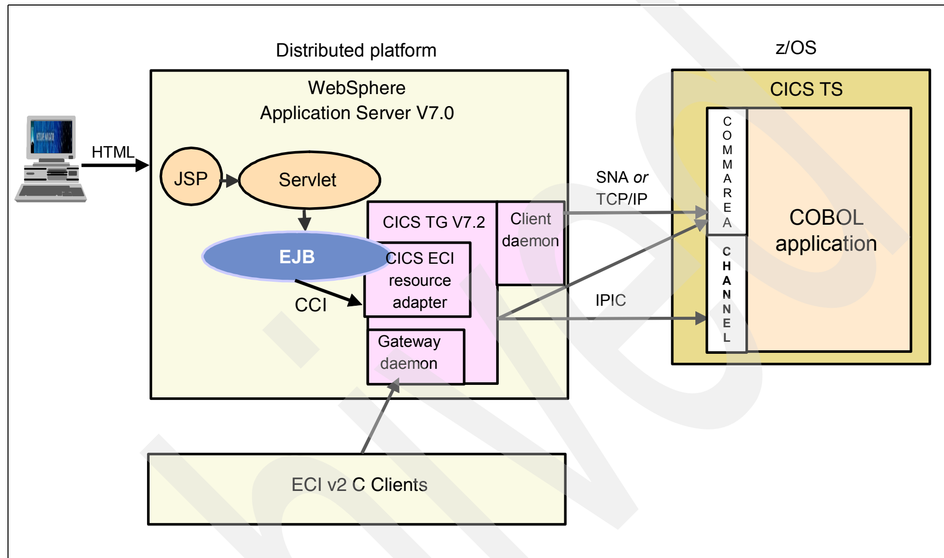


Figure 1-7 WebSphere Application Server and CICS TG on a distributed platform

The Gateway daemon is not required for JCA-based applications because the CICS TG is used in a local mode, which allows the transport drivers to be invoked directly from the J2EE enterprise bean. The ECI requests are routed directly to the CICS TS using either a TCP/IP, an SNA, or IPIC protocol connection. The management of connections, transactions, and security is controlled within the WebSphere Application Server through a combination of configuration parameters and application deployment descriptors.

Note: The qualities of service provided by the JCA may vary depending on the topology.

The specific qualities of service (in terms of the JCA system contracts) that apply to this topology are as follows.

- Connection pooling

  Pooling of connections is provided seamlessly by the pool manager component of WebSphere Application Server, enabling the reuse of connections to the resource adapter between multiple J2EE components.

  > **Note:** The TCP/IP or SNA network connections from the Client daemon into the CICS region are managed and re-used by the Client daemon component of the CICS TG, and are not subject to the JCA connection pooling system contract.

- Transaction management

  In this topology, two-phase commit global transactions are supported using the CICS ECI XA resource adapter and IPIC connections directly into CICS TS V3.2. In addition, the CICS ECI resource adapter (cicseci.rar) can be used with any version of CICS, but it only supports the LocalTransaction interface. As a result, the scope of the transaction is limited to the Resource Manager that is the associated connection factory and the specified CICS server. Resource Manager Local Transactions support single-phase commit processing only. However, if the J2EE application server supports the JCA option of last-resource commit optimization, an ECI interaction can participate in a global transaction, provided that it is the only single-phase-commit resource contained within the global transaction. This function is provided by the last-participant support in WebSphere Application Server V6.0 and later releases.

- Security management

  Security credentials (user ID and password) propagated through to CICS from the WebSphere Application Server can be determined by the application (component-managed) or by the Web or EJB™ container (container-managed). Container-managed sign-on is suggested, as it is good practice to separate the business logic of an application from qualities of service, such as security and transactions. In this topology, however, the principal means of enabling container-managed authentication is by specifying the user ID and password in a JCA authentication entry (also known as an alias), and associating the alias with the resource reference when the J2EE application is deployed.

An ECIv2 C client application on a separate distributed platform can communicate with the CICS TS using the Gateway daemon. In this configuration only the TCP protocol is supported from the ECIv2 C client to the Gateway daemon.

WebSphere Application Server on distributed with CICS TG on z/OS

When a WebSphere Application Server is deployed on one of the distributed platforms, it is possible to access the CICS TS through a Gateway daemon running on z/OS, as in Figure 1-8.

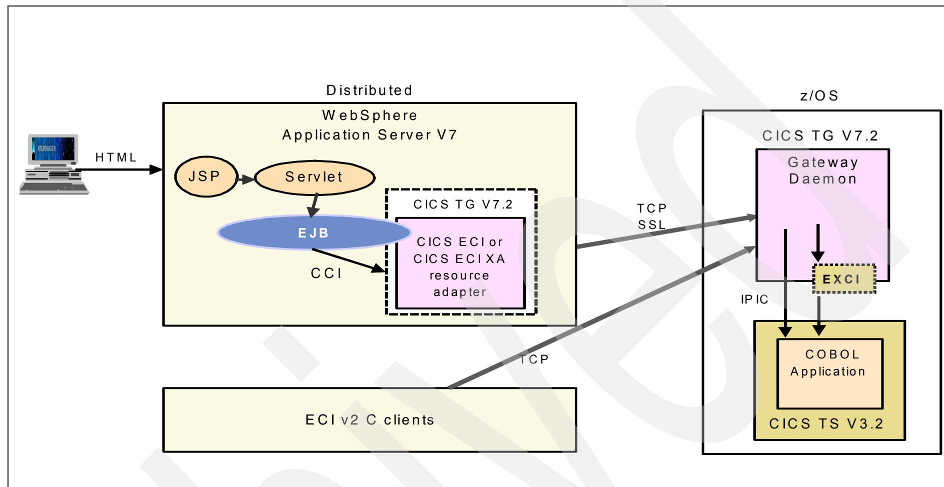


Figure 1-8 WebSphere Application Server on distributed and CICS TG on z/OS

For JCA-based applications, the protocol specified in the connection settings of the connection factory is one of the two supported remote protocols (TCP or SSL). The communication from the Gateway daemon on z/OS to the CICS TS region uses either the cross memory EXCI function, or if communicating with a CICS TS at level V3.2 or later, then the TCP/IP socket-based IPIC protocol may be used.

This topology enables work to be efficiently distributed across a Parallel Sysplex® using the z/OS internet protocol (IP) workload-management functions, including Sysplex Distributor and TCP/IP port sharing. These technologies allow an individual Gateway daemon to be removed as a single point of failure and enables incoming work from remote clients to be efficiently balanced across multiple Gateway daemons running on different z/OS logical partitions (LPARs).

The specific JCA qualities of service that apply to this topology are as follows:

- Connection pooling

  The connection pool represents physical network connections between the WebSphere Application Server and the Gateway daemon on z/OS. In such a configuration, it is essential to have an efficient connection-pooling mechanism because otherwise a significant proportion of the time from

Archived

making the connection to receiving the result from CICS and closing the connection can be in the creation and destruction of the connection itself. The JCA connection-pooling mechanism mitigates this impact by allowing connections to be pooled by the WebSphere Application Server pool manager.

- Transaction management

  In this topology, two-phase commit global transactions are supported, using the CICS ECI XA resource adapter and either EXCI or IPIC connections to the CICS TS. Also, single-phase commit transactions using the CICS ECI resource adapter and extended units-of-work are still supported.

  > **Note:** in either case, if the enterprise bean is deployed with a transactional deployment descriptor (for example, a value of REQUIRED), the resulting ECI request to the CICS TS region uses either transactional EXCI or IPIC depending on the server name protocol configuration.

- Security management

  In this configuration, the Gateway daemon is the entry point into the system on which the CICS TS is running. Therefore, the Gateway daemon would normally perform an authentication check of incoming ECI requests from clients. However, after the user has authenticated to the WebSphere Application Server, a password might not be available to send to the Gateway daemon. Therefore, you must devise a mechanism to establish a trust relationship between the WebSphere Application Server and the Gateway daemon such that the WebSphere Application Server can be trusted to flow only the user ID on the request through to the CICS TG. Solutions such as SSL client authentication and virtual private networks (VPNs) can be used to establish such a trust relationship.

An ECIv2 C client application on a distributed platform can communicate with CICS TS using the Gateway daemon on z/OS. In this configuration only the TCP protocol is supported from the ECIv2 C client to the Gateway daemon.

WebSphere Application Server and CICS TG on System z

In a System z topology, WebSphere Application Server can be deployed on either a z/OS system or a Linux on System z. The qualities of service differ between these two topologies and are discussed separately.

Archived

WebSphere Application Server and CICS TG on z/OS

In this topology (Figure 1-9), only the CICS ECI resource adapters are supported. The most common z/OS configuration uses the local mode of operation. This results in a direct cross-memory connection between WebSphere Application Server and CICS, using either EXCI or optimized fast local sockets. Figure 1-9 shows an application deployed to a WebSphere Application Server on z/OS.

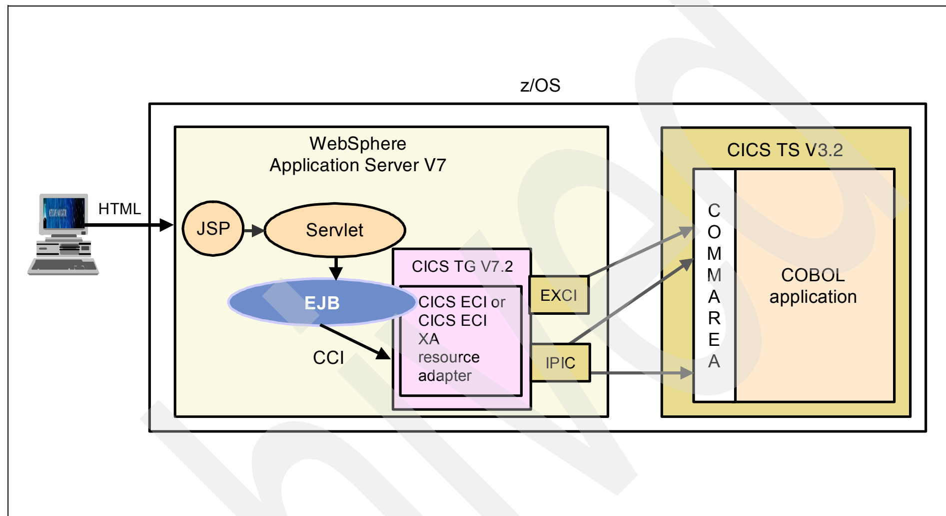


Figure 1-9 WebSphere Application Server and CICS TG on z/OS

The remote mode of operation is required when the WebSphere Application Server executes in a separate LPAR from the CICS and the Gateway daemon. However, the highest qualities of service can be achieved when the CICS TG local mode of operation is used.

The specific JCA qualities of service that apply to this topology are as follows:

- Connection pooling

  The connection pool is a set of connection objects managed by the WebSphere Application Server, which are not directly associated with the EXCI pipes or IPIC sessions used for communicating with CICS.

- Transaction management

  A two-phase commit capability is provided for either EXCI or IPIC connections. When using EXCI, RRS acts as the external transaction coordinator, managing the transaction scope between the WebSphere Application Server and CICS. However, when using IPIC XA, requests are sent directly to CICS.

- Security management

  Both container-managed and component-managed sign-on is supported. In this topology, the ECI resource adapter flows only the user ID to CICS with the ECI request; it is assumed that the user was already authenticated by the WebSphere Application Server. When using container-managed sign-on, a z/OS specific functionality known as *thread identity support* is provided by WebSphere Application Server on z/OS.

> **Note:** *Thread identity support* allows an authenticated RACF® identity from the WebSphere Application Server to be propagated through the CICS TG into the CICS TS for each request.

CICS TG for Linux on System z

A topology where the WebSphere Application Server and the CICS TG are both deployed on System z (Figure 1-10) provides a flexible and scalable environment based on the virtualization capabilities of IBM z/VM® and Linux systems.

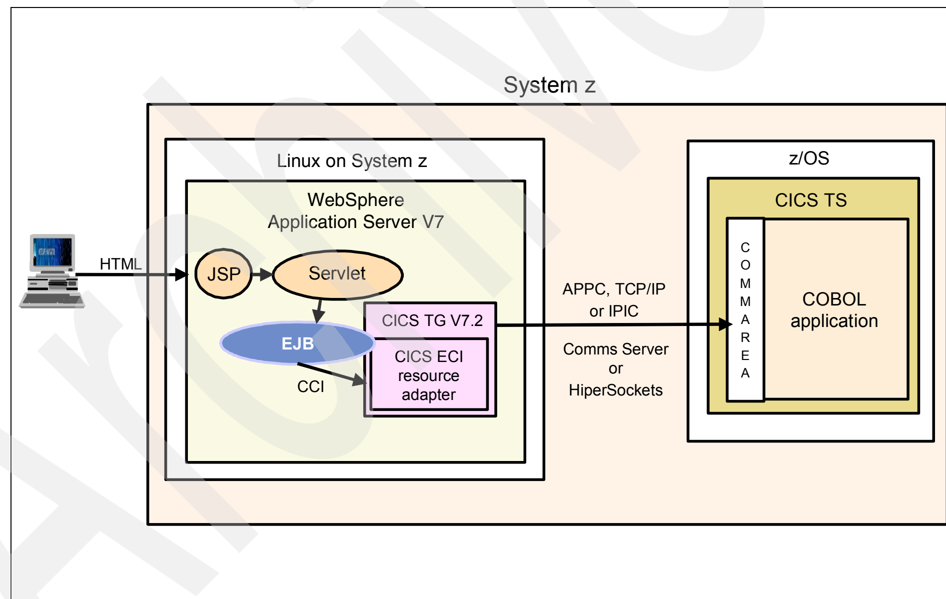


Figure 1-10 WebSphere Application Server and CICS TG for Linux on System z

The JCA qualities of service for this topology are almost identical to those described in “WebSphere Application Server and CICS TG on a distributed platform” on page 15, because Linux on System z (within a JCA and CICS TG scenario) can be treated as a distributed platform. A significant exception to this generalization is that the HiperSockets™ function can be used to provide a highly

efficient cross-memory transport for TCP/IP-based communication into CICS (using either the ECI over TCP/IP function of CICS TS V2.2 and later releases or the IPIC function of CICS TS V3.2). Alternatively, an APPC connection to CICS TS on z/OS or VSE can be provided by the IBM Communications Server for Linux.

You can find more information about the JCA topologies in the IBM white paper, *Integrating WebSphere Application Server and CICS using CICS Transaction Gateway,* under the White papers section on the following Web page:

http://www.ibm.com/software/htp/cics/tserver/v32/library

Archived

Archived

2

Developing CICS TG Applications

In this chapter, we explore the different interfaces and programming languages supported by the CICS TG for developing client applications to access CICS. Information is provided on usage in the following scenarios.

- ECI, EPI and ESI interfaces
- Java - base classes
- Java - JCA
- C
- C++
- COBOL
- .NET

Archived

© Copyright IBM Corp. 2009. All rights reserved.

2.1 Programming interfaces

All the principal interfaces provided by the CICS TG 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)

2.1.1 External Call Interface (ECI)

The ECI (Figure 2-1) is a simple remote procedural call style interface, used for linking to CICS applications. The COMMAREA or channel is the data interface used for the exchange of data between the client application and the CICS.

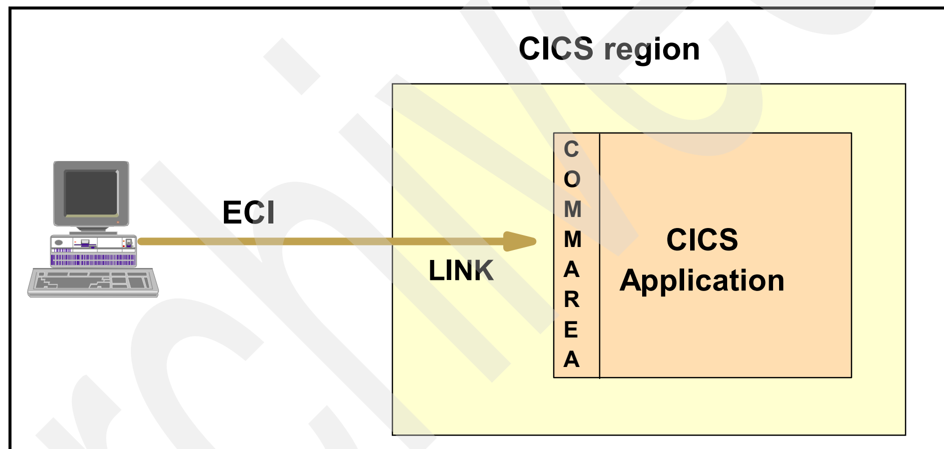


Figure 2-1 External Call Interface

CICS sees the client request as a form of a distributed program link (DPL) request and executes the request using a mirror transaction that mirrors the link command within CICS. Two variants of LINK are supported:

- synconreturn

  Where the CICS mirror program syncpoints before returning control to the client

- extended units of work

  Where multiple links can be chained together into a single unit of work

2.1.2 External Presentation Interface (EPI)

The EPI (Figure 2-2) is used for invoking 3270-based transactions. A virtual terminal is installed in CICS, and CICS sees the request as running on this virtual terminal controlled by the CICS TG. The API is only supported over SNA connections when used with CICS TS on z/OS, but is supported over TCP/IP connections with TXSeries CICS servers

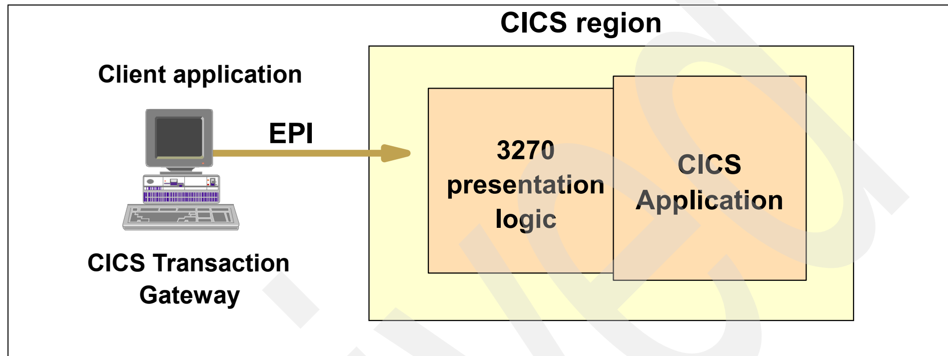


Figure 2-2 External Presentation Interface

2.1.3 External Security Interface (ESI)

The ESI (Figure 2-3) is used for verifying and changing security information held in the CICS External Security Manager (ESM), such as RACF. Due to the underlying support in CICS, it can currently only be used over SNA connections into CICS TS for z/OS.

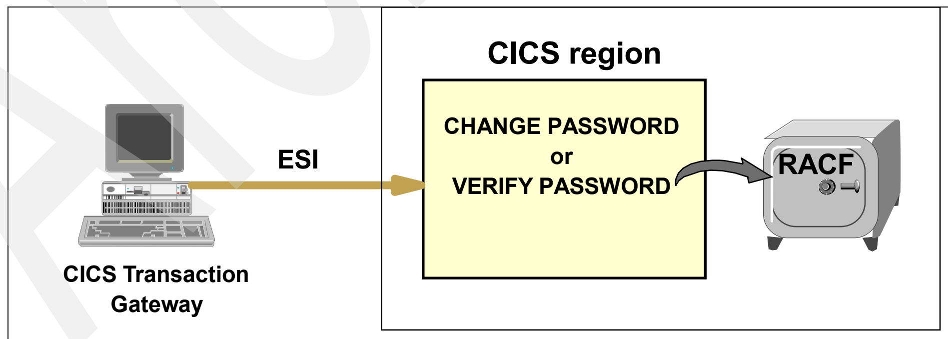


Figure 2-3 External Security Interface

Archived

2.2 Programming languages

In this section we provide a summary of the six principal programming languages environments supported by the CICS TG. We consider each language in term and consider how a simple ECI application can be implemented for each environment.

- Java
- J2EE
- C
- C++
- COBOL
- .NET

2.2.1 Java applications

The CICS TG provides a set of Java base classes that can be used in a variety of Java application environments (including standalone applications or J2EE components such as servlets). This includes 64-bit application environments, due to the byte code portability offered by the Java architecture. The four key classes shown in Table 2-1 are used for making ECI, EPI, and ESI calls. They are contained in the com.ibm.ctg.client package and supplied in the ctgclient.jar file.

Table 2-1 Java base classes

Class	Description
JavaGateway	Controls connection to the CICS TG and the flowing of requests
ECIRequest	Represents an ECI request
EPIRequest	Represents an EPI request
ESIRequest	Represents an ESI request

The EPI support classes (Table 2-2) are recommended as an alternative to the EPIRequest class, due to the simpler more object orientated model.

Table 2-2 Java EPI support classes

Class	Description
Terminal	Represents the EPI virtual terminal
Screen	Represents the instance of a screen on a terminal
Field	Provides access to fields on a screen

Archived

Sample code

The ECI Java program shown in Example 2-1 uses a JavaGateway object to flow a simple ECI request to CICS by executing the following steps:

1. The program instantiates a JavaGateway object. The parameterized JavaGateway constructor simplifies the instantiation of the JavaGateway by first setting the relevant properties and then implicitly calling the open() method.
2. The program creates an instance of an ECIRequest object and sets the call parameters.
3. The application flows the request to CICS using the flow() method of the JavaGateway object.
4. The program checks for a successful return code from the flow operation.
5. The program closes the JavaGateway object using the close() method.

Example 2-1 Sample ECI Java program

```
try {
1 JavaGateway jg = new JavaGateway("tcp://wtsc59.itso.ibm.com",2006);
  byte commarea[]=("---------------------------").getBytes("IBM037");

2 ECIRequest req = new ECIRequest(ECIRequest.ECI_SYNC, //sync or async
    "SC59CIC1", //CICS region name
    null, null, //userid & password
    "EC01", //program name
    null, // Mirror transaction ID
    commarea, commarea.length, //commarea data & length
    ECIRequest.ECI_NO_EXTEND, //extended mode
    ECIRequest.ECI_LUW_NEW); //LUW token

3 jg.flow(req);
  System.out.println( "Commarea output: "
    + new String(req.Commarea, "IBM037"));
4 System.out.println("Rc: " + req.getRc());
5 jg.close();

}catch (IOException ioe) {
  System.out.println("Handled exception: " + ioe.toString());
}
```


Archived

Qualities of service

When using the Java base classes, the following features and qualities of service are available:

- Connection management

  Connection management is the responsibility of the application developer. Connections are managed through the use of the JavaGateway class that represents the connection from the Java client to a Gateway daemon. Each thread of execution needs to either manage its own dedicated JavaGateway object or share access to a set of pooled objects. Local and remote modes of operation are supported, allowing a locally or remotely installed CICS TG to be used.

- Transport security

  The TLS and SSL security protocols are supported for the encryption of connections from the Java client to the Gateway daemon. In addition, in local mode it is also possible to use SSL connections directly into CICS TS V3.2 when using a direct IPIC connection into CICS. Both client and server authentication options are supported, and fine control of the encryption ciphers is available.

- Transactional support

  Synconreturn and one-phase commit transactions are supported when using the ECIRequest Java base class. To use a synconreturn request, set the extend mode field in the ECIRequest constructor to ECI_NO_EXTEND. To chain together several requests into an extended logical unit-of-work (LUW), set the extend mode field to ECI_EXTENDED. The CICS TG then generates the LUW identifier, which is returned to identify the unit-of-work in CICS. This LUW identifier must be input to all subsequent calls for the same unit-of-work by the user application. To terminate the LUW, set the eci_extend_mode parameter to ECI_NO_EXTEND in the final call or set the eci_extend_mode parameter to ECI_COMMIT or ECI_BACKOUT to explicitly commit or back out changes.

- Data transfer

  For Java applications using the ECI, data can be transferred using either a COMMAREA or a channels interface. When using a COMMAREA, the data record should be a class that implements the javax.resource.cci.Record interface. When using containers and channels, the data record should be a class which implements the javax.resource.cci.MappedRecord interface.

- 64-bit application support

  Due to the portability of the Java architecture, the Java base classes are supported in 32- and 64-bit runtime environments when used in remote mode. In addition, in local mode 64-bit applications are supported on z/OS in conjunction with WebSphere Application Server V6.1 for z/OS or later.

- Further information

  Both synchronous and non-synchronous call types are supported for use with the Java base classes. Most applications use synchronous calls due to the lower overheads and simpler programming model. Synchronous calls are supported using either a callback or a get reply mode.

  The programming reference for the Java base classes is provided in Javadoc™ format, which is available in the CICS TG Information Center at the following Web page:

  `http://publib.boulder.ibm.com/infocenter/cicstgmp/v7r2/index.jsp?topic=/com.ibm.cics.tg.doc/javadoc/index.html`

  Detailed information about developing an ECI application using the Java base classes is provided in Chapter 4, “Java-based Sample Application” on page 65. For information about using the EPI support classes, using the asynchronous calls, or developing applications using the ESI refer to the IBM Redbooks publication, *The XML Files: Using XML for Business-to-Business and Business-to-Consumer Applications*, SG24-6104.

2.2.2 JCA applications

The CICS TG provides a set of resource adapters for use in J2EE application server environments (such as WebSphere Application Server). The resource adapters compile the necessary runtime JAR files into a single archive, which is used for both application development and deployment. The following three resource adapters are provided:

- cicseci.rar: CICS ECI resource adapter
- cicseciXA.rar: CICS ECI XA resource adapter
- cicsepi.rar: CICS EPI resource adapter

The CICS ECI XA resource adapter and CICS ECI resource adapter provide the same programming interfaces. However, the XA resource adapter allows integration with the XA transaction support in the J2EE application server. This provides the potential for two-phase commit global transaction support from WebSphere Application Server to CICS. For more information, see the “Transactional support” bullet item on page 33.

The JCA specification defines a standard set of programming interfaces called the Common Connector Interface (CCI) for communicating with Enterprise Information Servers (EIS). The resource adapters each provide a set of classes extending the CCI, to provide connections and interaction classes for programmatic access to CICS.

JCA applications can be developed in one of two modes:

- Managed

  Managed applications are the recommended approach because this allows integration with the Application Server transaction, connection management, and security support.

- Unmanaged

  Unmanaged applications can be run in any environment, and rely on the manual creation and management of connections. They do not benefit from any of the runtime qualities of service offered by the J2EE application server.

The classes listed in Table 2-3 are the key CCI objects used for making CCI calls to an EIS such as CICS. They are provided in the javax.resource.cci package in the resource adapters.

Table 2-3 CCI classes

Class	Description
InitialContext	Used to make interactions with the JNDI namespace
ConnectionFactory	Used to create a connection
Connection	The applications handle to a managed connection in the pool
Interaction	Represents the request to CICS
ResourceException	Parent class for all exceptions generated

The classes listed in Table 2-4 on page 31 are the principal ECI and EPI objects used for JCA-managed applications. They are contained in the com.ibm.connector2.cics, com.ibm.connector2.screen, and com.ibm.connector2.cci packages supplied in the relevant resource adapter.

Archived

Table 2-4 CICS CCI classes

Class	Description
EciInteractionSpec	Defines the details of an individual ECI call
ECIChannelRecord	Record representing the collection of containers in a channel
CCILocalTransaction	Controls the transaction state for a one-phase commit transaction
EPIConnectionSpec	Defines the details of an individual EPI call
AIDKey	Represents the 3270 AID keys for EPI calls
LogonLogoff	Controls terminal logon and logoff for EPI calls
CICSTxnAbendException	A resource exception type for a CICS transaction abend

Sample code

The JCA program shown in Example 2-2 on page 32 runs in the managed environment and sends an ECI request to CICS by executing the following steps:

1. The program instantiates an initial context and uses this to look up the resource reference for the connection factory in the JNDI namespace.
2. A handle to a connection is created by invoking the getConnection() method on the connection factory object. This returns a connection from the connection pool.
3. An interaction is created from the connection representing an instance of an ECI call.
4. A record is created using the custom GenericRecord class to represent the record passed to and from CICS.
5. The call parameters are set on the interaction using the ECIInteractionSpec. This includes the call type as synchronous, the name of the CICS program, and the length of the COMMAREA record.
6. The call is invoked using the execute() method on the interaction.
7. The interaction and connection are closed, releasing the connection back into the pool.

Archived

Example 2-2 Sample CCI application

```
try {
1 Context ic = new InitialContext();
  ConnectionFactory cf = (ConnectionFactory)
  ic.lookup("java:comp/env/eis/ResRef");

2 Connection cxn = cf.getConnection;
3 Interaction ixn= cxn.createInteraction();

4 GenericRecord record = new GenericRecord(
             ("--------------------------").getBytes("IBM037"))

5 ECIInteractionSpec ixnSpec= new ECIInteractionSpec();
  ixnSpec.setInteractionVerb(ixnSpec.SYNC_SEND_RECEIVE);
  ixnSpec.setFunctionName("EC01");
  ixnSpec.setCommareaLength(record.length);

  System.out.println("Comm in: "
                  +new String(record.getCommarea(),"IBM037"));

6 ixn.execute(ixnSpec, record, record);
}catch (ResourceException re) {re.printStackTrace();}

try{
7 ixn.close();
  cxn.close();
}catch(Exception e){e.printStackTrace();}
```

Qualities of service

When using the CCI a rich set of qualities of service are available, as discussed in the following sections.

- Connection management

  For managed applications, connections are managed by the J2EE application server. The pool manager component of the J2EE application server is responsible for managing the state of all connections and integrates the pooling support with the security and transaction support. Local and remote modes of operation are supported, allowing a locally or remotely installed CICS TG to be used.

Archived

- Transport level security

  The TLS and SSL security protocols are supported for the encryption of connections from the Java client to the Gateway daemon. In addition, in local mode it is also possible to use SSL connections directly into CICS TS V3.2 or later when using a direct IPIC connection into CICS. Both client and server authentication options are supported, and fine detailed control of the encryption ciphers is available.

- Transactional support

  Synconreturn, one-phase commit, and two-phase commit transactions are supported when using the CICS ECI XA resource adapter. Two-phase commit transaction control is though the XA interface and control can either be explicit or implicit. Explicit transaction control functions through the use of the UserTransaction interface, and is referred to as bean-managed. Implicit control, or container-managed, functions through the use of the EJB deployment descriptor and is delegated to the EJB container. For further information about transaction management with JCA applications refer to the IBM whitepaper *Exploiting the J2EE Connector Architecture, Integrating CICS and WebSphere Application Server using XA global transactions* available at the following Web page:

  http://www.ibm.com/developerworks/websphere/techjournal/0607_wakelin/0607_wakelin.html

- Data transfer

  For JCA-based applications using the ECI, data can be transferred using either a COMMAREA or a channel interface. When using a COMMAREA, the data record should be a class that implements the javax.resource.cci.Record interface. When using containers and channels data record should be a class that implements the and javax.resource.cci.MappedRecord interfaces.

  CICS TG provides APIs for creating channels and containers containing either binary (BIT) or character-based (CHAR) data types. More information about the usage of channels and containers with JCA applications is contained in Chapter 3, “Channels and Containers” on page 49.

- 64-bit

  Due to the portability of the Java architecture, the JCA classes are supported in 32- and 64-bit runtime environments when used in remote mode. In addition, in local mode 64-bit applications are supported on z/OS in conjunction with WebSphere Application Server for z/OS V6.1 or later.

- Further information

  Both synchronous and non-synchronous call types are supported for use with the CCI. Most applications use synchronous calls due to the lower overheads and simpler programming model.

Tooling integration is provided through the Rational Application Developer J2C tooling, which provides a set of wizards for importing COBOL or PL/I copybooks and for creating J2C beans to encapsulate the entire call to CICS. This is described further in Chapter 5, “J2C Application Development With Rational Application Developer” on page 105.

The programming reference for the CCI is provided in Javadoc format, which is available with the product, and also in the CICS TG Information Center at the following Web page:

http://publib.boulder.ibm.com/infocenter/cicstgmp/v7r2/index.jsp?topic=/com.ibm.cics.tg.doc/javadoc/index.html

For further information about JCA programming refer to our samples discussed in 4.2.2, “Sample COMMAREA-based JCA client development” on page 74 and 4.3.2, “Sample channel-based JCA client development” on page 93.

2.2.3 C applications - Client API

The CICS TG provides a set of C functions, known as the *Client API*, for making ECI, EPI, or ESI calls to CICS. These are the same functions also provided with the CICS Universal Client and are only available in local mode, when the CICS TG is installed on the same machine as the client application. A full range of functions is supported, including both synchronous and asynchronous ECI, EPI, and ESI calls, all of which are defined in the header files cics_eci.h, cics_epi.h, and cics_esi.h. These can be found in the CICS TG /include sub-directory. Applications have to compiled using the relevant header file and need to be linked against the correct library, which for Windows is the cclwin32.lib file. Support is provided by CICS TG for multiplatforms and requires a locally installed CICS TG on the client machine.

The following key functions are provided for using for each of the ECI, EPI, and ESI interfaces as shown in Table 2-5, Table 2-6 on page 35, and Table 2-7 on page 35).

Table 2-5 ECI functions

Function	Description
CICS_ExternalCall	ECI call
CICS_EciListSystems	List systems call, returning available CICS servers

Table 2-6 EPI functions

Function	Description
CICS_EpiInitialize	Initialization of the EPI
CICS_EpiAddTerminal	Install a terminal in CICS
CICS_EpiAddExTerminal	Install an extended terminal in CICS
CICS_EpiStartTran	Start transaction on a terminal
CICS_EpiReply	Send data to a CICS transaction
CICS_EpiDelTerminal	Delete installed terminal
CICS_EpiPurgeTerminal	Immediate deletion of installed terminal
CICS_EpiListSystems	List system call, returning available CICS servers
CICS_EpiTerminate	Termination of the EPI

Table 2-7 ESI functions

Function	Description
CICS_VerifyPassword	Verify user ID and password
CICS_ChangePassword	Change password
CICS_SetDefaultSecurity	Set default user ID to be used for requests

Attention: Usage of these C interfaces requires a locally installed CICS TG on the same machine as the client application

Qualities of service

When using the C APIs the following features and qualities of service are available:

- Connection management

  The C interfaces are only supported in local mode, so there are no network connections to be managed into the Gateway.

- Transport level security

  SSL and TLS security options are not supported with C applications.

- Transactional support

  Synconreturn and one-phase commit (extended LUWs) transactions are supported when using the ECI. To start an extended LUW, set the eci_extend_mode parameter to ECI_EXTENDED and set the eci_luw_token

parameter to zero when making a program link call. The CICS TG then generates an LUW identifier, which is returned in the eci_luw_token field of the parameter block. This identifier must be input to all subsequent calls for the same unit of work by the user application. To terminate the LUW, set the eci_extend_mode parameter to ECI_NO_EXTEND in the final call or set the eci_extend_mode parameter to ECI_COMMIT or ECI_BACKOUT to commit or back out changes.

- Data transfer

  For C client applications using the ECI, data can be transferred using only a COMMAREA, which is passed to and from CICS as an encoded byte array.

- 64-bit application support

  Only 32-bit runtime application support is provided.

- Further information

  Both synchronous and non-synchronous call types are supported for use with the Client API. Asynchronous calls are supported using either a callback or a get reply model.

  The programming reference for the C APIs is provided in the CICS TG Programming Guide, which is available in the CICS TG Information Center at the following Web page:

  http://publib.boulder.ibm.com/infocenter/cicstgmp/v7r2/index.jsp

2.2.4 C applications: ECI v2

The CICS TG V7.2 provides a new API for making ECI calls to CICS. This is similar in function to the original ECI but provides the ability to use a remote Gateway daemon rather than a locally installed CICS TG. Only synchronous ECI calls are supported. The functions are defined in the header files ctgclient_eci.h and ctgclient.h which can be found in the CICSTG /include sub-directory. Applications then need to be linked against the correct library, which for Windows is the ctgclient.lib file. Support is provided by both CICS TG for z/OS and CICS TG for Multiplatforms. There is no client runtime support on z/OS for ECIv2.

The main functions listed in Table 2-8 on page 37 are provided for using for using the ECI v2 interfaces in C.

Archived

Table 2-8 ECI v2 functions

Function	Description
CTG_openRemoteGatewayConnection	Open connection to Gateway daemon
CTG_closeGatewayConnection	Close connection to Gateway daemon
CTG_closeAllGatewayConnections	Close all connections
CTG_listSystems	List defined CICS systems
CTG_ECI_Execute	Execute ECI call

Sample code

The client application shown in Example 2-3 sends an ECI request to CICS by executing the following steps:

1. Declare the data types for the program and server name.
2. Build the COMMAREA and initialize to nulls.
3. Build the ECI parameter block and initialize.
4. Execute the ECI request passing in the ECI parameter block.
5. Retrieve the COMMAREA data returned from CICS.

Example 2-3 Sample C ECI application

```
#include <cics_eci.h>
int main(){
1  char               program[8 + 1]  = "EC01";
   char               Server[8 +1] = "SC59CIC1";
   short              rc;

2  int calen = 18;
   char               CommArea[calen];
   memset(CommArea, '\0', calen);

3  ECI_PARMS          EciParms;
   memset(&EciParms, 0, sizeof(ECI_PARMS));
   EciParms.eci_version                              = ECI_VERSION_1A;
   EciParms.eci_call_type                            = ECI_SYNC;
   memcpy(&EciParms.eci_program_name, program, 8);
   memcpy(&EciParms.eci_system_name,  Server, 8);
   EciParms.eci_commarea                             = CommArea;
   EciParms.eci_commarea_length                      = calen;
   EciParms.eci_extend_mode                          = ECI_NO_EXTEND;
   EciParms.eci_luw_token                            = ECI_LUW_NEW;

4  rc = CICS_ExternalCall (&EciParms);
```

```
      if (rc == ECI_NO_ERROR) {
5       printf("Call returned with : %s\n", CommArea);
      } else {
        printf("Call failed with error: %d\n", rc);
      }
}
```

Qualities of service

When using the ECI v2 the following qualities of service are available.

- Connection management

  The C interfaces are only supported in remote mode, so the network connection from the Client application to the Gateway daemon must be managed by the client program.

- Transport level security

  SSL and TLS security options are not supported with ECI v2 C applications.

- Transactional support

  Synconreturn and one-phase commit (extended LUWs) transactions are supported when using the ECI using the same model as used in the Client API C interfaces (see 2.2.3, “C applications - Client API” on page 34).

- Data transfer

  For C client applications using the ECI, data can be transferred using only a COMMAREA, which is passed to and from CICS as an encoded byte array.

- 64-bit application support

  Only 32-bit runtime application support is provided.

- Further information

  Only synchronous call types are supported for use with the ECI v2 API. The programming reference for the ECIv2 is provided in doxygen format in the ctgclientdoc.zip file, and is also available in the CICS TG Information Center at the following Web page:

  http://publib.boulder.ibm.com/infocenter/cicstgzo/v7r2/index.jsp?topic=/com.ibm.cics.tg.zos.doc/remotecapi/index.html

Sample code

The C program shown in Example 2-4 sends an ECI request to CICS by executing the following steps:

1. Declare the data types for the Gateway connection and the program and server name.
2. Build the COMMAREA and initialize to nulls.
3. Build the ECI parameter block and initialize.
4. Open the Gateway connection using the host name and port of the Gateway daemon.
5. Execute the ECI request passing in the gateway token from step 5. and the ECI parameter block.
6. Retrieve the COMMAREA data returned from CICS.

Example 2-4 Sample C ECIv2 application

```
#include <ctgclient_eci.h>
int main(){
1  short rc;
   int port = 2006;
   char*  hostname = "wtsc59.itso.ibm.com";
   CTG_ConnToken_t gwyTok;
   char             program[8 + 1]  = "EC01";
   char             Server[8 +1] = "SC59CIC1";
2  int calen = 53;
   char             CommArea[calen];
   memset(CommArea, '\0', calen);

3  CTG_ECI_PARMS        EciParms;
   memset(&EciParms, 0, sizeof(CTG_ECI_PARMS));
   EciParms.eci_version                           = ECI_VERSION_2;
   EciParms.eci_call_type                         = ECI_SYNC;
   memcpy(&EciParms.eci_program_name, program,8);
   memcpy(&EciParms.eci_system_name,  Server, 8);
   EciParms.eci_commarea                          = CommArea;
   EciParms.eci_commarea_length                   = calen;
   EciParms.eci_extend_mode                       = ECI_NO_EXTEND;
   EciParms.eci_luw_token                         = ECI_LUW_NEW;

4  rc=CTG_openRemoteGatewayConnection(hostname, port, &gwyTok, 0);
   if (rc == CTG_OK) {
     printf("Connection to Gateway opened\n");

5    rc = CTG_ECI_Execute(gwyTok, &EciParms);
```

Archived

```
        if (rc == ECI_NO_ERROR) {
6           printf("Call returned with data: %s\n", CommArea);
        } else {
            printf("Call returned with error: %d\n", rc);
        }
    } else {
        printf("Connection to Gwy failed with error: %d\n", rc);
    }
}
```

2.2.5 C++ applications

The CICS TG provides a set of classes for C++ programming to the ECI, EPI, and ESI interfaces. These can only be used in local mode on the same machine as the CICS TG is installed.

The classes shown in Table 2-9 are provided for the C++ interface when using ECI.

Table 2-9 ECI C++ classes

Class	Function
CclConn	Connection between a client and a named CICS server
CclBuf	Buffer to hold the COMMAREA data
CclFlow	Created for each client-server interaction
CclUOW	For managing a transaction scope in an extended LUW
CclECI	Access to server information

The classes in Table 2-10 are provided for the C++ interface when using ESI.

Table 2-10 ESI C++ classes

Class	Function
CclConn	Provides verifyPassword() and a changePassword() functions for managing the credentials for a CICS user ID.

Archived

The classes in Table 2-11 are provided for the C++ interface when using EPI.

Table 2-11 EPI C++ classes

Class	Function
CclEPI	Initializes and terminates the CICS TG EPI function
CclEPI	Represents a 3270 terminal
CclEPI	Controls the flow of data to and from CICS within a single 3270 session
CclScreen	Maintains data on the 3270 virtual screen and provides access to this data. A single CclScreen object is created by the CclTerminal object; use the screen method on the CclTerminal object to obtain it. The CclScreen object is updated by the CclTerminal object when 3270 data is received from CICS.
CclField	CclField objects are created and deleted when 3270 data from the CICS region is processed by a CclScreen object. Methods in this class allow field text and attributes to be read and updated. Modified fields are sent to CICS on the next send.

Sample code

The C++ program shown in Example 2-5 flows an ECI request to CICS by executing the following steps:

1. Create the objects CclConn, CclFlow and CclBuf and set the CICS server name, call type and COMMAREA data.
2. Flow the request by calling the link method on the CclConn object.
3. Catch any exceptions else and process the reply using CclBuf method dataArea().

Example 2-5 Sample C++ application

```
1 CclConn server("SC59CIC1");
  CclFlow sflow(Ccl::sync);
  CclBuf commarea("Data to be sent");

2 server.link(sflow, "EC01", &commarea);
3 cout << commarea.dataArea();

public:
void handleReply(CclBuf* receivedCommarea){
   }
};
```

Archived

Qualities of Service

When using the C++ API the following qualities of service are available.

- Connection management

  The C++ interfaces are only supported in local mode. There are no network connections to be managed into the Gateway.

- Transport level security

  SSL and TLS security options are not supported with C++ applications.

- Transactional support

  Synconreturn and one-phase commit (extended LUWs) transactions are supported when using the ECI using the same model as used in the Client API C interfaces (see 2.2.3, “C applications - Client API” on page 34).

- Data transfer

  For C++ client applications using the ECI, data can be transferred using only a COMMAREA, which is passed to and from CICS as an encoded byte array.

- 64-bit application support

  Only 32-bit runtime application support is provided.

- Further information

  Both synchronous and non-synchronous call types are supported for use with the Client API. Asynchronous calls are supported using either a callback or a get reply model. For more information refer to the Programming Guide, which is available in the CICS TG Information Center at the following Web page:

  http://publib.boulder.ibm.com/infocenter/cicstgmp/v7r2/index.jsp

2.2.6 COBOL applications

The CICS TG provide COBOL copybooks that allow the C libraries described in 2.2.3, “C applications - Client API” on page 34 to be directly linked from COBOL applications. Copybooks can be found in the <install directory>/copybook directory and are separated into copybooks for the ECI, EPI, and ESI interfaces. Once the COBOL application has been compiled, the output should be linked against the shared library (cclwin32.lib on Windows) for the platform on which the application runs.

Archived

Sample code

The client application shown in Example 2-6 sends an ECI request to CICS by executing the following steps:

1. Initialize the data areas for the server name and COMMAREA.
2. Initialize COMMAREA to nulls.
3. Build the ECI parameter block and initialize.
4. Execute the ECI request passing in the ECI parameter block.
5. Retrieve the COMMAREA data returned from CICS.

Example 2-6 Sample COBOL application

```
1       01 WS-AREA.
          02 COMMAREA              PIC X(18) VALUE LOW-VALUES.
          02 SERVER                PIC X(8)  VALUE 'SC59CIC1' .

2         SET  ECI-COMMAREA        TO ADDRESS OF COMMAREA.
3         MOVE LOW-VALUES TO ECI-PARMS.
          SET  ECI-SYNC            TO TRUE.
          MOVE 'EC01'              TO ECI-PROGRAM-NAME.
          MOVE SERVER              TO ECI-SYSTEM-NAME.
          MOVE LENGTH OF COMMAREA  TO ECI-COMMAREA-LENGTH.
          SET  ECI-NO-EXTEND       TO TRUE.
          SET  ECI-VERSION-1A      TO TRUE.
4         CALL 'CICSEXTERNALCALL'
                  USING BY REFERENCE ECI-PARMS
                  RETURNING ECI-ERROR-ID.
          IF ECI-NO-ERROR
5          DISPLAY 'Program returned with CommArea:' COMMAREA
          END-IF.
```

Qualities of service

When using the COBOL API the following features and qualities of service are available:

- Connection management

  The supplied COBOL interfaces are only available for use with ECI, EPI and ESI interfaces in local mode and so there are no network connections to be managed into the Gateway.

> **Tip:** COBOL copybooks and linkage scripts are not supplied for use with the ECI v2 C API. They can be created by hand for use with COBOL programs in the same way as supplied for the ECI Client API.

Archived

- Transport level security

  SSL and TLS security options are not supported with applications in local mode.

- Transactional support

  Synconreturn and one-phase commit (extended LUWs) transactions are supported when using the ECI using the same model as used in the Client API interfaces (see 2.2.3, “C applications - Client API” on page 34).

- Data transfer

  For COBOL client applications using the ECI, data can be transferred using only a COMMAREA, which is passed to and from CICS as an encoded byte array.

- 64-bit application support

  Only 32-bit runtime application support is provided.

- Further information

  Both synchronous and non-synchronous call types are supported. Asynchronous calls are supported using either a callback or a get reply model. For more information refer to the Programming Guide, which is available in the CICS TG Information Center at the following Web page:

  http://publib.boulder.ibm.com/infocenter/cicstgmp/v7r2/index.jsp

2.2.7 .NET client applications

This CICS TG V7.2 provides an API that allows .NET applications to access CICS servers using the ECI API. This is built on top of the ECIv2 API and can be used in remote mode with the CICS TG on any platform. The function is provided as a supported product extension in SupportPac CA73 CICS TG V7.2 .NET application support available at the following Web page:

http://www.ibm.com/support/docview.wss?rs=1083&uid=swg27007241

The API provides a mixed-mode DLL assembly (Figure 2-4 on page 45) that exposes the functions of the ECI API in a form that will be familiar to developers working in C# or VB.NET. This API allows .NET applications, such as ASP.NET applications running in Microsoft® IIS, to invoke COMMAREA-based programs in CICS using the facilities of a remote CICS Transaction Gateway. The SupportPac includes sample ECI applications written in C# and VB.NET and documentation for using the API.

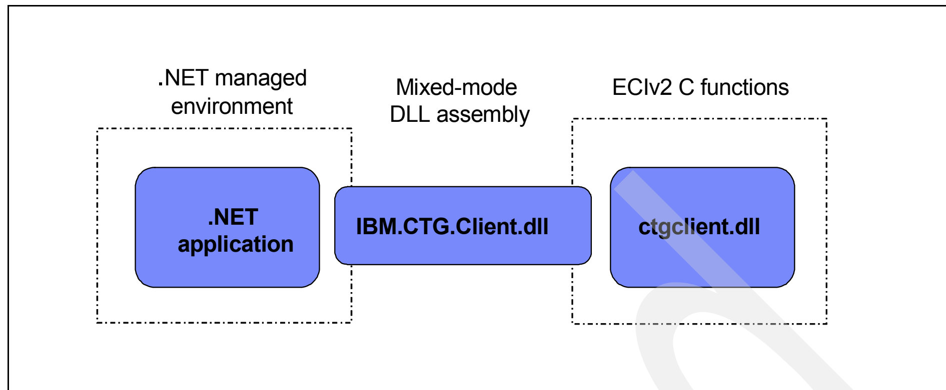


Figure 2-4 .CICS TG .NET mixed mode DLL assembly

If support for EPI or the ESI interfaces is required in a .NET environment, there are 2 other options provided by the CICS TG.

1. Create a mixed mode DLL wrapper for the EPI or ESI C interfaces. A tutorial demonstrating how to do this is available in the SupportPac CICS TG: Developing .NET components for CICS connectivity , CA72.
2. Use the COM classes provided by the CICS TG. These are supported for use with the Visual Basic® and VBScript languages. These are also only available in local mode.

The classes shown in Table 2-12 are provided for the .NET interface when using ECI.

Table 2-12 ECI .NET classes

Class	Description
EciRequest	Represents an ECI request
GatewayConnection	Represents a connection to a remote Gateway
GatewayException	Represents an exception that is thrown when an error occurs within a Gateway daemon
GatewayRequest	Base class for requests that can be sent to a Gateway
Trace	Provides methods for controlling trace

Sample code

The .NET program shown in Example 2-7 sends an ECI request to CICS by executing the following steps:

1. Set the host name and port values.
2. Open the Gateway connection. This will be automatically closed when the object is cleaned up.
3. Initialize the ECI request and set the server name, program name, extend mode, and COMMAREA size.
4. Execute the ECI request using the Flow method.
5. Check the ECI return code and retrieve the COMMAREA data returned.
6. Otherwise return the ECI error code.

Example 2-7 Sample .NET C# ECI application

```
1string host = "wtsc59.itso.ibm.com";
 int port = 2006;

2 using (GatewayConnection gwyC = new GatewayConnection(host, port)) {
3       EciRequest eciReq;
        eciReq = new EciRequest();
        eciReq.ServerName = "SC59CIC1";
        eciReq.Program = "EC01";
        eciReq.ExtendMode = EciExtendMode.EciNoExtend;
        eciReq.SetCommareaData(new byte[18]);
4       gwyC.Flow(eciReq);

        switch (eciReq.EciReturnCode) {

5        case EciReturnCode.EciNoError:
            byte[] commarea = eciReq.GetCommareaData();
            string commareaStr = Encoding.ASCII.GetString(commarea);
            Console.WriteLine("ASCII: {0}", commareaStr);
            break;

        default:
6            Console.WriteLine("ECI return code: {0} ({1})",
            eciReq.EciReturnCode.ToString(), (int) eciReq.EciReturnCode);
            break;
        }
}
```

Qualities of service

When using the .NET API the following features and qualities of service are available:

- Connection management

  The .NET API is built on ECI v2 for use in remote mode. The network connection from the Client application to the Gateway daemon must be managed by the client program.

- Transport level security

  SSL and TLS security options are not supported with ECI v2 applications.

- Transactional support

  Synconreturn and one-phase commit (extended LUWs) transactions are supported when using the ECI using the same model as used in the Client API C interfaces (see 2.2.3, “C applications - Client API” on page 34).

- Data transfer

  For .NET client applications using the ECI, data can be transferred using only a COMMAREA, which is passed to and from CICS as an encoded byte array.

- 64-bit application support

  Only 32-bit runtime application support is provided.

- Further information

  Only synchronous ECI call types are supported. Documentation for using the .NET API is provided in doxygen format in the SupportPac.

Archived

2.3 Summary

The following table summarizes the capabilities of the different APIs discussed in this chapter

Table 2-13 Language summary table

Language	Local or remote mode	Transport level security (SSL)	Global transaction support	Interfaces	Channels and containers	64-bit
Java base classes	Local and remote mode	Yes	synconreturn 1pc (extended LUWs)	ECI, EPI, ESI	Yes	Yes
JCA (CCI)	Local and remote mode	Yes	synconreturn 1pc (LocalTransaction) 2pc (XA)	ECI, EPI	Yes	Yes
C	Local and remote mode	No	synconreturn 1pc (extended LUWs)	ECI, EPI, ESI, ECIv2	No	No
C++	Local mode	No	synconreturn 1pc (extended LUWs)	ECI, EPI, ESI	No	No
COBOL	Local mode	No	synconreturn 1pc (extended LUWs)	ECI, EPI, ESI	No	No
.NET	Remote mode	No	synconreturn 1pc (extended LUWs)	ECIv2	No	No

Archived

3

Channels and Containers

In this chapter, the following topics are described:

- Channels and containers background
- IPIC communications
- Character conversion background
- Channels and containers Data Conversion
- Using the CICS TG channels and containers API
- Use of channels and containers in a COBOL program
- Conversion Issues

Archived

© Copyright IBM Corp. 2009. All rights reserved.

3.1 Overview

One of the powerful capabilities introduced with the CICS Transaction Gateway V7.1 (CICS TG) is the ability to use channels and containers when communicating with the CICS Transaction Server. The channels and containers interface is supported when communicating from a Java client using Java ConnectorArchitecture (J2C) Common Client Interface (CCI) Classes, or using the original CICS TG base classes to a CICS TS V3.2 and later. This chapter provides background on the basic concepts and also provides coding examples.

The ability to use channels and containers means that you are no longer limited to 32 KB transfers between your Java program and CICS. In addition to providing the ability to send more data, you can also more logically organize and control the data. Channels and containers place data conversion in the control of the application programmer, not the systems programmer. To use these abilities, connect the Gateway daemon to CICS using an IPIC connection introduced for CICS TS V3.2.

3.2 Channels and containers background

New for CICS TS V3 is a feature called channels and containers. Prior to CICS TS V3, program-to-program communications inside CICS was limited to a COMMAREA. This single communications area had a maximum size of 32 KB. The COMMAREA data either had to be converted as appropriate on the client side, or you could have CICS perform the conversion. If CICS performs the conversion, you must inform CICS how to perform the data conversion through the DFHCNV conversion mechanism. The DFHCNV conversion mechanism provides assembler macros, compiled into the DFHCNV table, to tell CICS on a field-by field basis how to convert the data. Although this provides a fine level of control, it takes quite a bit of processing time and is tedious when changes are made to the data layout. Additionally, because it is the system programmer's responsibility to make changes to the DFHCNV table, it inserts the system programmer into the application development life cycle.

With CICS TS V3, for program-to-program communications, you can place large amounts of data into a container associated with a channel. You can place multiple containers in a channel, then pass the channel (with all of its containers) to the target CICS program. When placing data into a container, you assign a name to the container. CICS doesn't place any size restrictions on the amount of data placed in the container. But because there is a finite amount of storage available in a CICS region, the application programmer should pass only the required data areas. A channel is also given a name.

Archived

3.2.1 Channel and container data formats

Character data eligible for conversion is placed in a CHAR container, whereas bit-oriented data is placed into a BIT container. Data is placed into a container using the **EXEC CICS PUT CONTAINER() CHANNEL()** command. The **GET CONTAINER() CHANNEL()** command is used to access data. Character data in a CHAR container can be converted to a different code page when you place data into or take data out of the CHAR container. You could, for example, place data encoded in one character set (for example, US EBCDIC) into a container and get the data in a different character set (such as UTF-8). BIT containers are for bit-oriented data or characters that should not undergo conversion. Typical BIT data would be a graphic (for example, a picture for a drivers license), or binary-oriented numerical data.

3.2.2 Channel and container data flows

Only modified containers are returned by CICS on an ECI or DPL call. To take advantage of this optimization, the application programmer should return data in one or more separate containers and return an error container when requests cannot be processed successfully. This approach to passing data is quite different from the COMMAREA approach where you pass a single communications area large enough to hold both the request data and the response data. When passing COMMAREAs from a CICS TG Java client to CICS, trailing nulls are not transmitted, but when using channels and containers, unmodified containers are not transmitted at all. COMMAREAs are a fixed size that must be agreed to by the client and the CICS server program. The COMMAREA sent from the client can be larger than expected by the CICS server program, but if smaller, will usually cause problems. Containers can be variable in size, with the client and server working with the data as appropriate.

Use of channels and containers allows you to better organize the data passed between the client and the CICS server program. Better data organization will result in a quicker understanding of the data structures being passed, allowing faster problem resolution and lower maintenance costs. For example, in an insurance application where you need to return the various types of insurance policies owned by a specified customer, instead of sending arrays of various sized segments in a single COMMAREA, or possibly making multiple server application invocations for each insurance policy, the server application could place the customer's boat insurance information in one container, their car insurance information in another container, their homeowner insurance information in another container, and then return all containers at once. The client, instead of working with variable length arrays (or making multiple invocations to the server) would simply place the customer identifier in a container associated with a channel and pass the channel to the server. When

the data was passed back to the client with the various insurance policy information, the client would access the various policy information by accessing returned containers in the channel.

IPIC communications

The ability to communicate between CICS regions, or between a Gateway daemon and a CICS region through TCP/IP, was introduced with CICS TS V3.2. This initial TCP/IP-based implementation is for DPL between CICS regions and ECI using the CICS Transaction Gateway.

For CICS region-to-region communication, the target CICS region listens for incoming IPIC communications using a TCPIPService definition specified with a protocol of IPIC. Configuration of both the CICS region and the CICS TG for communications using IPIC is documented in the CICS Transaction Gateway manuals and InfoCenter.

3.3 Character Conversion background

As computers were developed, different techniques were used to represent the various characters. The series of bits (binary digits), or bit configuration used to represent a set of characters is referred to as a *code page*. One of the original representations, called ASCII (American Standard Code for Information Interchange™), used 7 bits (binary digits) to represent 128 characters. This 7-bit code actually used 8 bits, with the 8th bit being used for parity during data transmissions. Later code pages used the 8th bit as part of the character representation, which allowed for up to 256 characters. Because various languages have different character sets, different 8-bit code pages are needed for various languages. As computer vendors used various code pages, standards organizations like ANSI (American National Standards Institute) and ISO (International Standards Organization) stepped in to help create a standard set of code pages to enhance interoperability between platforms and operating systems. Some of the more common code pages are EBCDIC (Extended Binary Coded Decimal Interchange Code), 437 (the original IBM PC code page), ISO 8859-1 (the default code page for HTML), and 1252 (Microsoft Windows default code). Additional code pages exist for various languages (such as Greek, Turkish, Portuguese, Icelandic, and so forth).

3.3.1 Double byte character representation

For languages that need more than 256 character representations in their code page, the DBCS (or Double Byte Character Sets) were defined. By using 2 bytes, with each byte consisting of 8 bits (for a total of 16 bits), 65,000 characters can

Archived

be represented. Each DBCS has an accompanying SBCS (Single Byte Character Set). This SBCS uses 8 bits and often contains the most frequently used characters for a given character sets. To optimize transmission, a shift-in and shift-out character is used to signal the switch between the DBCS and its accompanying SBCS. Although a DBCS contains many possible bit configurations, there are different DBCSs for different languages.

3.3.2 Unicode character representation

An attempt at providing a single representation for all characters is Unicode. Developed at the same time as the UCS (Universal Character Set) standard, Unicode contains about 100,000 characters. It would seem like the problem should stop with Unicode, but unfortunately there are several Unicode representations. One of the more popular versions of Unicode is UTF-8.

3.3.3 UTF-8 character representation

UTF-8 is a variable length code page representation where one, two, three, or four bytes are used to represent the various characters. Because of the way bit configurations are organized in relation to other code pages, conversion between various ASCII code pages and UTF-8 is usually efficient. The speed of conversion, plus using only the number of bytes needed to represent any one character provides insight as to why UTF-8 is popular.

3.3.4 Character representation using Java

Java internally uses a Unicode representation for its “String” object. Having a single character representation that includes all characters used in any language helps Java with its promise of “Write Once Run Anywhere.” When using characters in Java, one almost always uses the Java String object. There is also a need when using Java to represent data with series of octets (for example, a series of 8-bit bytes). For this data representation, Java offers a byte array. A Java byte array is just a series of bytes. Because the bytes are in an array, an individual byte can be referenced by indexing into the array. A byte array is similar to a PIC X field in COBOL, where the X indicates that the field can contain any bit configuration. Because the CICS TG Java client is commonly running in Java, but the CICS server program is running in a different code page, it is easy to see that some data conversion needs to take place. Data conversion using the channel and container API is described in 3.4, “Channel and container data conversion” on page 54. It provides the CICS server program the flexibility of obtaining character data from the client in any code page it prefers. Based on the details provided in this section on code pages, hopefully it is clear why the CICS TG developers, for the normal case, chose to take Java character data (a String)

from the Java client program and convert it to a byte array containing UTF-8 (a compact Unicode) before the data is transmitted to the CICS server program where it can be accessed in any code page. Conversely, character data returned from the CICS server program is converted to UTF-8 before being transmitted back to the Java client where it is then placed into a String object.

3.3.5 Coded Character Set Identifier (CCSID)

One last term in this section is CCSID (Coded Character Set Identifier). Although a CCSID and code page are often used synonymously, they are actually quite different. A code page is a mapping between characters and bit configurations. Each character is assigned a code point (bit configuration). A CCSID (represented by a 16-bit number) contains the necessary information to preserve characters while processing and exchanging characters.

3.4 Channel and container data conversion

A diagram depicting the data conversion performed when using the CICS TG is shown in Figure 3-1.

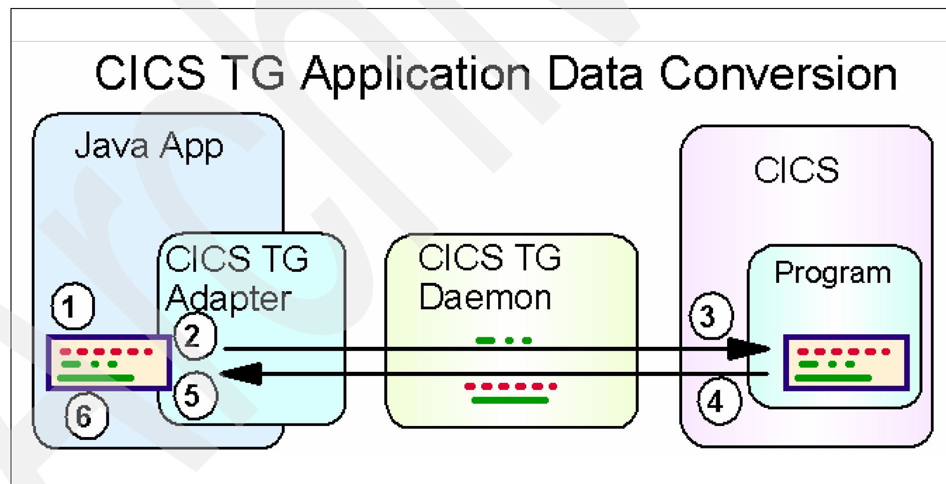


Figure 3-1 Application Data Conversion

Archived

The following steps are a description of the flows and conversion depicted in Figure 3-1 on page 54:

1. The Java client application creates a channel, and places data in BIT or CHAR containers as appropriate. The containers in the channel represent both CHAR containers (longer line) and BIT (shorter line) containers. In this example, the client places only the smaller CHAR container into the channel and invokes the CICS server program.
2. The CICS ECI resource adapter, by default, places String data in the CHAR container into byte arrays using the default JVM™ encoding. In the diagram only the smaller CHAR container placed on the channel will be transmitted to the CICS server program.

> **Note:** An alternative API to the JCA CCI is the CICS TG Java base classes. These classes are the original CICS TG Java API and are discussed in 4.2.1, "Sample COMMAREA-based Java client development" on page 67. When using base classes, the encoding for the container data can be set to any required code page. For information about when to use this function, refer to the developerWorks® article *Exploiting CICS Channels and Containers from Java clients*, available at the following Web page:
>
> `http://www.ibm.com/developerworks/websphere/library/techarticles/0810_wakelin/0810_wakelin.html`

3. When the data in the channel is received by the CICS server program, it accesses the data in the container with an **EXEC CICS GET CONTAINER()** command. The data returned from the **GET CONTAINER** will either be in the region's home code page (specified in the LOCALCCSID SIT parameter), or in the code page requested by **GET CONTAINER** command using the INTOCCSID option.
4. After the CICS server program processes the request, it will add one or more CHAR or BIT containers to the channel to be returned to the client program. In Figure 3-1 on page 54, the CICS server program has placed a CHAR container (longer line) and a BIT container (shorter line) on the channel. The code page of the CHAR container data is either the regions home code page or the code page specified using the FROMCCSID of the **PUT CONTAINER()** command that placed the data in the CHAR container. Before CICS sends the modified (new) containers back to the client, CICS converts the CHAR container data (just the longer green line in this case) to UTF-8 (or the code page specified for the channel by client). In this example, only the two modified (new) containers are sent back to the client.
5. The CICS ECI resource adapter converts the CHAR container data (in this case the longer green line) to a String.

6. The Java application programmer accesses the String contents of the CHAR container and uses it as appropriate.

3.5 Using the CICS TG channel and container API

This section discusses how to create channels and containers in your Java client program, how to send the channel/containers to a CICS TS V3.2 region, and how to access returned containers. Because there are differences between the CCI and the CICS TG base classes, we discuss each of them separately.

The design of channels and containers is elegant, efficient, and should greatly simplify the development of your Java client application. As previously indicated in 3.2.1, “Channel and container data formats” on page 51, there are two types of containers: CHAR and BIT. If you pass a String to the methods that create a container, you will create a CHAR container. If you pass a byte array to the methods that create a container, you will create a BIT container.

In your CICS TS V3.2 application program, when you use a **`GET CONTAINER`** on a CHAR container, the container will be converted into the specified INTOCCSID or INTOCODEPAGE. If INTOCCSID or INTOCODEPAGE are not specified, the value for conversion defaults to the CCSID of the region (which is specified on the LOCALCCSID SIT parameter). This means that for CHAR containers, conversion is automatic and you do not have to specify anything in your client or server programs. In your CICS TS V3.2 application program, when you use a **`GET CONTAINER`** on a BIT container, no conversion will be performed on the data. If the client sends a BIT container and you request conversion on the BIT container data, your **`GET CONTAINER API`** command will receive an invalid request response code. It therefore makes sense, and is suggested, to place character data that is to be converted into CHAR containers and data that should not be converted into a BIT container. This is different from the previous COMMAREA approach to sending data between your Java client program and CICS. Wizards are provided in RAD and RDz to generate Java classes that perform conversion of your data on the client side when using the COMMAREA approach. There are also DFHCNV macros that allow you to specify conversion on the server side.

Because number data (for example, an account balance) needs some kind of a conversion between your Java client and your CICS TS V3.2-based program, most applications pass numbers as characters (for example, alphanumeric data, a display numeric field in COBOL (PIC 999 USAGE IS DISPLAY)).

A straight forward approach to handle data when using channels and containers in your client Java program is to create a Java data object. This is an object with properties that correspond to your data structure with getter and setter methods

for each property in your object. Your data object would have a toString() method that gets the values of all properties as a single String (which you could use to create a CHAR container), and your data object would have a fromString() method that would separate a single String received in a CHAR container into separate properties. Although you could return hundreds of individual containers with individual bits of data (for example, customer first name, customer last name, customer address line 1, and so forth), you will find that grouping related data into a single container is the easiest approach to managing the data.

3.5.1 Using channels and containers in a Java client

The channel and container API is different between Java client programs written to use the CCI (Java Connect Architecture Common Client Interface), and Java client programs using the CICS TG base classes. If you are using the CICS TG base classes, go to 3.5.3, “Using channels and containers in Java clients using the CICS TG base classes” on page 60.

Consider the following section of code shown in Example 3-1.

Example 3-1 Sample CCI-based Java client

```
public static String REQUEST_INFO_CONTAINER = "REQUESTCONTAINER";
public static String CHANNELNAME = "MYCHANNEL";
public static String ERROR_CONTAINER = "ERROR";
public static String DATA_CONTAINER = "DATACONTAINER";
.
.
.
try {
1 ConnectionFactory cf = createAConnectionFactory();
2 Connection conn = cf.getConnection();
3 Interaction interaction = conn.createInteraction();
// Create the channel and place the REQUEST_INFO_CONTAINER in the
channel
4 ECIChannelRecord myChannel = new ECIChannelRecord(CHANNEL_NAME);
String dataToBeSent = "Some data to be sent";
5 myChannel.put(REQUEST_INFO_CONTAINER, dataToBeSent);
// create an ECIInteractionSpec
ECIInteractionSpec ixnSpec= new ECIInteraction
6 Spec(SYNC_SEND_RECEIVE,"CICSPROG");
// invoke the CICS TS V3.2 program
7 interaction.execute(ixnSpec, myChannel, myChannel);
// we returned, check if we received an error container, best practice
is to
```


Archived

```
// return an error container when something goes wrong on the server
side
8 if (myChannel.containsKey(ERROR_CONTAINER)) {
// an error container was received from the CICS-based program
// process the error as appropriate and return
9 throw new Exception(myChannel.get(ERROR_CONTAINER));
}
// use the returned data
10 String str = myChannel.get(DATA_CONTAINER);
System.out.println("The information returned from CICS was: "+str);
// shut it down
interaction.close();
conn.close();
} catch (ResourceException e) {
System.out.println("Unexpected ResourceException:"+e);
e.printStackTrace();
}
```

Explanation of Example 3-1 on page 57:

- 1 This statement creates a connection factory. It will be different depending on whether you want a managed or non-managed connection. See Chapter 2, “Developing CICS TG Applications” on page 23 for further information about the use of managed versus non-managed JCA applications.
- 2 This statement gets a connection.
- 3 An interaction is created from the connection.
- 4 Create a channel.
- 5 Add a container. In this case, because we are adding a String, it is a CHAR container. If we had added a byte array, we would have added a BIT container.
- 6 Create an interactionSpec.
- 7 Invoke the CICS program by invoking the execute method of the interaction, passing it the interactionSpec, the input channel, and the output channel.
- 8 This is how you test for the existence of a container. It is suggested that when things to wrong in the server program, to return a container with the error information.
- 9 In this case, when an error is returned, we are writing out the String that was returned in the CHAR error container.
- 10 This is the command to get the contents of a CHAR container. If the container is a BIT container, the contents are placed in a byte array.

3.5.2 Browsing containers in a Java clients using the CCI

In order to browse a series of returned containers using the CCI interface, use the following sample code shown in Example 3-2.

Example 3-2 Browse containers sample CCI-based Java code

```
Object o = null;
String containerName = null;
String contValueString = null;
byte[] contValueByteArray = null;
1 Iterator it = myChannel.keySet().iterator();
2 while (it.hasNext()) {
3 containerName = (String)it.next();
4 o = myChannel.get(containerName);
5 if (o instanceof byte[]) {
System.out.println("The "+containerName+" container is a BIT
container");
6 contValueByteArray = (byte[])myChannel.get(containerName);
// work with the data as a byte[]
7 } else {
8 contValueString = (String)myChannel.get(containerName);
System.out.println("The "+containerName+
" CHAR container has a value of:"+contValueString);
}
```

Explanation of Example 3-2:

- 1 The myChannel.keySet() returns a set of container names.
- 2 Iterate through the set.
- 3 Get next container name.
- 4 Get the container value – the get() method on a channel returns an Object.
- 5 Check if the Object is an instance of a byte array.
- 6 If it is a byte array, work with it. Note that although we know it is of type byte array, we have to type cast it as a byte[ ] because the get() method returns an Object.
- 7 If it isn't a byte[ ] (a BIT container), it must be a String (a CHAR container).
- 8 If it is a String, work with it. Note that although we know it is of type String, we have to type cast it as a String because the get() method returns an Object.

Archived

3.5.3 Using channels and containers in Java clients using the CICS TG base classes

This section details the use of the channel and container API when using the CICS TG base classes. If you are using the CCI classes, proceed to 3.5.1, “Using channels and containers in a Java client” on page 57.

Note: the code below contain a minimal amount of error checking. This level of error checking would be unacceptable in a production environment.

Consider the section of code in Example 3-3.

Example 3-3 Sample CICS TG base class Java client

```
public static String REQUEST_INFO_CONTAINER = "REQUESTCONTAINER";
public static String CHANNELNAME = "MYCHANNEL";
public static String ERROR_CONTAINER = "ERROR";
public static String DATA_CONTAINER = "DATACONTAINER";
public static String theConnectionURL = “tcp://localhost”;
.
.
.
try {
1 JavaGateway javaGatewayObject = new JavaGateway(theConnectionURL,
2006);
// create a channel and container
2 Channel reqChannel = new Channel(CHANNEL_NAME);
String dataToBeSent = "Some data to be sent";
3 reqChannel.createContainer(REQUEST_INFO_CONTAINER, dataToBeSent);
// specify the details of the where and how you want to go
4 ECIRequest eciRequestObject = new ECIRequest(
ECIRequest.ECI_SYNC, //ECI call type
theServerName, //CICS server
null, //CICS username
null, //CICS password
theFunctionName, //Program to run
null, //Transaction to run
reqChannel, //Channel
ECIRequest.ECI_NO_EXTEND, //ECI extend mode
0 //ECI LUW token
);
5 int iRc = javaGatewayObject.flow(eciRequestObject);
// do gateway flow error checking here
// close the gateway object
```

```
6 Channel respChan = eciRequestObject.getChannel();
Container cont = null;
try {
7 cont = respChan.getContainer(ERROR_CONTAINER);
// if we are here, we received an error container – process it
return;
8 } catch (ContainerNotFoundException e) {
// if we are here, we didn’t receive an error container, fall through
9 ) catch (Throwable t) {
throw new Exception(“Unexpected Throwable: “+t.getMessage());
}
// all ok, process output containers
10 cont = respChan.getContainer(DATA_CONTAINER);
11 String str = cont.getCHARData();
12 System.out.println("The information returned from CICS was: "+str);
) catch (Throwable t) {
System.out.println(“Unexpected exception: “+t.getMessage();
}
```

Explanation of Example 3-3 on page 60:

- 1 Create a JavaGateway object.
- 2 Create a Channel with the specified name.
- 3 Create a CHAR container.
- 4 Create an ECI request.
- 5 Flow the Request.
- 6 Create an interactionSpec.
- 7 Check if we were returned an error container.
- 8 If the error container was not found, we will get a ContainerNotFoundException.
- 9 Catch other exceptions.
- 10 Get the returned container.
- 11Access the container contents as CHAR data.
- 12 Work with the returned CHAR data.

Archived

3.5.4 Browsing containers in a Java clients using the CICS TG base classes

To browse a series of returned containers using the CICS TG base classes you could use code as shown in Example 3-4. Note that you would need to add some error checking.

Example 3-4 Browse containers sample using CICS TG base classes Java code

```
// we have a returned channel - lets start processing it
1 Channel respChan = eciRequestObject.getChannel();
Container cont = null;
String contValueString = null;
byte[] contValueByteArray = null;
String containerName = null;
2 Iterator it = respChan.getContainerNames().iterator();
3 while (it.hasNext()) {
4 containerName = (String)it.next();
5 cont = respChan.getContainer(containerName);
6 if (cont.getType() == Container.ContainerType.BIT) {
7 contValueByteArray = cont.getBITData();
// work with the byte array
} else {
8 contValueString = cont.getCHARData());
// work with the String
}
}
```

Explanation of Example 3-4:

- 1 Access the returned channel.
- 2 Get a iterator for the set of container names.
- 3 Iterate through the container names.
- 4 Get a container name.
- 5 Get the container.
- 6 Check if it is a BIT container.
- 7 Get the data in the BIT container.
- 8 Get the data in the CHAR container.

Archived

3.6 Use of channels and containers in a CICS COBOL Program

Below are snippets from a COBOL program that illustrate the use of channels and containers in a CICS server program. Use of the channel and container API is fully documented (including many code samples) in Redbooks publication *CICS Transaction Server V3R1 Channels and Containers Revealed*, SG24-7227. The Redbooks publication also contains application design and implementation, system management and configuration, a sample application, and frequently asked questions.

Example 3-5 shows the code to determine if a COMMAREA was received or if a channel was received.

Example 3-5 Determine if COMMAREA or channel was received

```
.
.
PROCEDURE DIVISION.
* Check if passed a COMMAREA, if yes, ABEND
If EIBCALEN NOT = 0 THEN
EXEC CICS ABEND ABCODE('ABR0') END-EXEC
End-If.
* Check if passed a channel, if not, ABEND
EXEC CICS ASSIGN CHANNEL(Channel-Name) END-EXEC.
If Channel-Name = Space then
EXEC CICS ABEND ABCODE('ABR2') END-EXEC
End-If.
```

The code to access the data in a container is shown in Example 3-6. If the container is a CHAR container, because no INTOCCSID or INTOCODEPAGE is specified, the data will be converted to the regions home code page, specified in the LOCALCCSID SIT parm. If the container is a BIT container, no conversion would take place.

Example 3-6 Access container data

```
EXEC CICS GET CONTAINER('BROWSE-INFO')
INTO(Request-Information)
END-EXEC.
```

Example 3-7 shows the code to place data into a container.

Note: the keyword CHAR is not specified so the container specified will be a BIT container. This means that no conversion will take place when this data is placed into or read from the container. If the data needs to be converted, the CHAR keyword should be specified and both CICS and the CICS TG will convert the data as necessary when you place data into or take data out of the CHAR container.

Example 3-7 Store data into container

```
EXEC CICS PUT
CONTAINER('NUM-RECORDS')
FROM(Record-Counter)
END-EXEC.
```

3.7 Conversion problems

Install the following fixes to address specific issues with channels and containers:

- Correct data conversion when using channels and containers from non-EBCDIC systems (such as the CICS Transaction Gateway) requires the PTF for CICS APARs PK49490 and PK49021.
- Correct data conversion for non US-ASCII Java clients requires the PTF for APAR PK62925, available in CICS Transaction Gateway V7.1.0.2
- Review additional software requirements in the CICS Transaction Gateway support page:

  http://www-01.ibm.com/support/docview.wss?uid=swg21239203

Archived

4

Java-based Sample Application

The Java APIs provided by the CICS TG assist the developer in creating client applications for communication with server-side CICS applications. In this chapter we present four Java client applications that communicate with the CICS server program. Two of the sample application use the COMMAREA as the information exchange medium and the other two use channels and containers for the information exchange.

Archived

© Copyright IBM Corp. 2009. All rights reserved.

4.1 Java application setup

The following pre-requisite software installation and configuration settings are necessary to execute the sample Java Client applications.

4.1.1 Software pre-requisites

The software pre-requisites installed to run the sample applications are as follows:

- CICS TG v7.2 for z/OS
- Java1.5 SR7
- CICS TS 3.2

> **Note:** Instructions on how to obtain the complete source code for the applications described in this chapter are found in Appendix E, “Additional material” on page 253.

4.1.2 Configuring the CICS TG

Perform the following steps to configure the CICS TG:

1. Install the CICS TG v7.2 and Java 5 SR7 on z/OS platform.
2. Run the **`ctgctg`** command to start the configuration tool.
3. Make a CICS server connection (CICSIPIC) using the IPIC protocol.

Archived

4.2 COMMAREA-based applications

In the following sections we demonstrate the development of COMMAREA-based sample Java and J2C client applications.

4.2.1 Sample COMMAREA-based Java client development

Our sample Java client program CustProgGetSingleBaseClasses reads, deletes, adds, and updates customer records in a VSAM file by calling the CUSTPROG back-end CICS sample program. The source code is shown in Appendix B, “Java Sample Code” on page 191

The sample client program uses the base ECI APIs to call the sample CICS CUSTPROG program. This sample Java client program uses the COMMAREA for passing data to and from the CICS CUSTPROG program.

The following Java files are used by the client application:

- com.ibm.itso.eci.CustProgGetSingleBaseClasses.java

  This is the main Java file which makes the ECI calls to CICS
- com.ibm.itso.model.CustProgCommarea.java

  This is the Java file used to send and receive the customer data from the CICS server program

Compiling and Executing the sample Java client

For instructions on compiling and executing the Java client program, refer to Appendix D, “CICS samples” on page 243.

The usage instructions are shown in Example 4-1.

Example 4-1 Usage of COMMAREA-based sample java program

```
---------- CustProgGetSingleBaseClasses ----------
This program can be used to invoke the CUSTPROG CICS-based COBOL
program to retrieve a single record through the specified CICS TG.
Three positional parguments are required.  The CICS TG Base Classes are
used in this program.

Usage: CustProgGetSingleBaseClasses <ServerConnectionName> <Port>
<Gateway> <custid:xxxxxxxx>

where <ServerConnectionName> is the CICS Server Connection Name.
      <Port> is the CICS TG port configured using ctgcfg tool.
```

Archived

```
     <Gateway> is the the system where Gateway daemon is running.
     In <custid:xxxxxxxx>,  xxxxxxxx is the customer id whose data you
want to retrieve and the Customer Id length is 8 characters.

Example: CustProgGetSingleBaseClasses CIC1IPIC 2006
tcp://wtsc59.itso.ibm.com custid:00000001
The above example will try to get the customer record for Customer Id
"00000001"
```

Analyzing the sample Java client code

We now examine in detail how the sample Java client makes calls to the CICS CUSTPROG sample program to retrieve records.

First, examine the data structures used in the CustProgCommarea.java file shown in Example 4-2. This program contains a java class with the same data structures as in the sample CICS COBOL program COMMAREA CUSTIORQ copybook described in Example 4-1 on page 67. Objects of this CustProgCommarea class can be used to store the COMMAREA data retrieved from the CICS CUSTPROG sample program, or it can be used to send the COMMAREA data to the CICS CUSTPROG sample program. This class acts as the data binding class, which maps the COBOL copybook structure to its Java equivalent. The mapping is achieved on similar lines as explained in “Create container-Java mapping” on page 96.

Example 4-2 Data structure definitions in CustProgCommarea.java file

```
public class CustProgCommarea {

   private static String BLANKS50 = "
";
   private static String BLANKS30 = "                              ";
   private static String BLANKS20 = "                    ";
   private static String BLANKS8  = "        ";

   private String requestType = " ";
   private String returnCode = "  ";
   private String customerId = BLANKS8;
   private String customerLastName = BLANKS20;
   private String customerFirstName = BLANKS20;
   private String customerCompany = BLANKS30;
   private String customerAddr1 = BLANKS30;
   private String customerAddr2 = BLANKS30;
   private String customerCity = BLANKS20;
   private String customerState = BLANKS20;
   private String customerCountry = BLANKS30;
```

Archived

```
   private String customerMailCode = BLANKS20;
   private String customerPhone = BLANKS20;
   private String customerLastUpdateDate = BLANKS8;
   private String returnComment = BLANKS50;

// code continues here ...
```

The CustProgGetSingleBaseClasses.java file communicates with the CICS CUSTPROG sample program. The data structures in this class are shown in Example 4-3.

Example 4-3 Data structures defined in CustProgGetSingleBaseClasses.java

```
package com.ibm.itso.eci;

import java.io.IOException;
import java.io.UnsupportedEncodingException;
import com.ibm.ctg.client.*;
import com.ibm.itso.model.CustProgCommarea;

public class CustProgGetSingleBaseClasses {

1  private static JavaGateway javaGatewayObject;
2  private String theServerName = "";
3  private int iPortNumber = 0;
4  private String theConnectionURL = "";

5  private String theCustomerId = "00000001";
6  private String theFunctionName = "CUSTPROG";

// code continues here ...
```

- **1** JavaGateway object variable javaGatewayObject is used for making a connection to the Gateway daemon, which is listening on a port.
- **2** ServerName is the name of the CICS server defined in the ctg.ini configuration file.
- **3** iPortNumber is an int variable used for supplying the port number while creating a JavaGateway object.
- **4** ConnectionURL string variable holds the URL of the server name or the IP Address of the system where the Gateway daemon is running. It can be either tcp://<host or IP> or ssl://<host or IP> . If you do not specify a protocol, the default is tcp.

Archived

- 5 CustomerId is a String variable to hold the customer ID supplied by the Java client user for which the Java program is going to retrieve the details.
- 6 FunctionName is a String that holds the CICS server program name to which the client will be calling.

Next, we look at the ECIRequest object and how it is being used by the sample program, as shown in Example 4-4. The program creates the ECIRequest object using the extended constructor by passing the optional parameters.

Example 4-4 Use of the ECIRequest object in the sample Java client

```
1     eciRequestObject = new ECIRequest(ECIRequest.ECI_SYNC, // ECI
call type
2           theServerName, // CICS server
3           null, // CICS userid
4           null, // CICS password
5           theFunctionName, // CICS program to be run
6           null, // CICS transid to be run
7           abytCommarea, // Byte array containing the COMMAREA
8           cpd.length(), // COMMAREA length
9           ECIRequest.ECI_NO_EXTEND, // ECI extend mode
10          0); // ECI LUW token
```

Now we will examine the usage of each parameter:

- 1 The sample program uses the synchronous program link call. This is specified by the first parameter ECIRequest.ECI_SYNC passed to the ECIRequest class constructor.
- 2 The second parameter passed is theServerName, which contains the CICS server connection name. We will be passing this information from the command line.
- 3 The third parameter is used to provide a user ID to the CICS program. It is set to null because security was not enabled in our CICS region.
- 4 The fourth parameter provides the password for the user ID. It is also set to null because a user ID was not provided.
- 5 The fifth parameter, theFunctionName, is the CICS server program name to which the Java client program is going to call.
- 6 The sixth parameter is set to null because we are running the program as a default CICS transaction.
- 7 The seventh parameter, the abytCommarea, is a byte array of the CustProgCommarea object which holds the value for the customer ID in the customerId field.

Archived

- 8 The eighth parameter is the COMMAREA length specified by invoking the cpd.length() method to obtain the length of the data structure for holding the CustProgCommarea object.
- 9 The ninth parameter, ECIRequest.ECI_NO_EXTEND, specifies that the sample program is using non-extended ECI.
- 10 The last parameter applies to the LUW and is set to 0 as this program does not use extended ECI calls.

Next, we look at how an ECIRequest is sent to the CICS TG and how the response from CICS TG is received. The code shown in Example 4-5 is for flowing the ECIRequest to the JavaGateway flow method and receiving the response.

Example 4-5 Flowing the ECIRequest to the CICS TG

```
...........................
...................
      int iRc = 0;
      try {
1        iRc = javaGatewayObject.flow(eciRequestObject);
      } catch (IOException e3) {
         System.out.println("IOException encountered trying to invoke 
the CUSTPROG program. Exception was:"+ e3.getMessage());
         return;
      }

2     switch (eciRequestObject.getCicsRc()) {
      case ECIRequest.ECI_NO_ERROR:
         if (iRc == 0) {
            break;
         } else {
            System.out.println("\nError from Gateway ("+ 
eciRequestObject.getRcString() + ").");
            if (javaGatewayObject.isOpen() == true) {
               try { javaGatewayObject.close(); } catch (Throwable t2) 
{ ; }
            }
            return;
         }
      case ECIRequest.ECI_ERR_SECURITY_ERROR:
         System.out.print("\n\nSecurity problem communicating with the 
CICS TG. ");
         System.out.println("\nYou are not authorised to run this 
transaction.");
```


Archived

```
         System.out.println("\nECI returned: "+
eciRequestObject.getCicsRcString());
         if (javaGatewayObject.isOpen() == true) {
            try { javaGatewayObject.close(); } catch (Throwable t2) { ;
}
         }
         return;
      case ECIRequest.ECI_ERR_TRANSACTION_ABEND:
         System.out.println("\nTransaction ABEND indicator received.");
         System.out.println("\nECI returned: "+
eciRequestObject.getCicsRcString());
         System.out.println("Abend code was " +
eciRequestObject.Abend_Code+ "\n");
         if (javaGatewayObject.isOpen() == true) {
            try { javaGatewayObject.close(); } catch (Throwable t2) { ;
}
         }
         System.exit(1);
      default:
         System.out.println("\nProblem invoking Program.");
         System.out.println("\nECI returned: "+
eciRequestObject.getCicsRcString());
         System.out.println("Abend code was " +
eciRequestObject.Abend_Code+ "\n");
         if (javaGatewayObject.isOpen() == true) {
            try { javaGatewayObject.close(); } catch (Throwable t2) { ;
}
         }
         System.exit(1);
      }

      if (javaGatewayObject.isOpen() == true) {
3        try { javaGatewayObject.close(); } catch (Throwable t1) { ; }
      }
..........................
..................
```

- 1 The flow method of the JavaGateway object takes the eciRequestObject previously created in Example 4-4 on page 70. This method both flows the request and retrieves the response. The flow method returns an integer as a return code. Because this is non-extended synchronous call, the flow method will block until the server program completes the request.
- 2 The code next determines if the CICS return code is normal (ECI_NO_ERROR), and if so, whether an error was returned from the

JavaGateway (such as a thread timeout). If the CICS return code is not normal it checks first for a security error (ECI_ERR_SECURITY_ERROR) or a CICS transaction abend (ECI_ERR_TRANSACTION_ABEND). If neither of these two errors occurred, it outputs the value of the return code received.

- **3** Once the flow method responds, the JavaGateway object is closed because it has successfully received the COMMAREA data.

Next, we examine how the COMMAREA data sent by CICS server program is retrieved. Example 4-6 below shows how the program retrieves the COMMAREA data from the CustProgCommarea object.

Example 4-6 Retrieving COMMAREA data returned by the CICS CUSTPROG sample application

```
...........................
...................
try {
1         cpd.setBytes(abytCommarea, "037");
      } catch (UnsupportedEncodingException e) {
         System.out.println("Unsupported code page:037 found when
trying to convert the returned COMMAREA to the local code
page"+e.getMessage());
         e.printStackTrace();
      }
      if (cpd.getReturnCode().equals("00")) {
2        System.out.println("Customer id: "+cpd.getCustomerId()+",
name: "+cpd.getCustomerFirstName().trim()+"
"+cpd.getCustomerLastName().trim()+", last updated:
"+cpd.getCustomerLastUpdateDate());
      } else {
         System.out.println("The CICS program was accessed but a
problem was indicated:  the program returned
\""+cpd.getReturnComment().trim()+"\".");
      }
...........................
...................
```

- **1** The COMMAREA data sent by the CICS program will be in EBCDIC format. Therefore, the code converts the COMMAREA data from EBCDIC to Unicode before reading.
- **2** Once the data is converted, the data structure is read using the getter methods of the CustProgCommarea object.

4.2.2 Sample COMMAREA-based JCA client development

This section describes the sample Java client application `CustProgGetSingleJ2C` which uses the Java Connector Architecture (JCA), also known as J2C APIs, to communicate with our sample back-end CUSTPROG program. The client uses a COMMAREA as the exchange medium to transfer data between the client and the CICS server application. The source code is shown in “CustProgGetSingleJ2C” on page 197.

Client application task analysis

An analysis of the various tasks the client application must perform is shown in Figure 4-1.

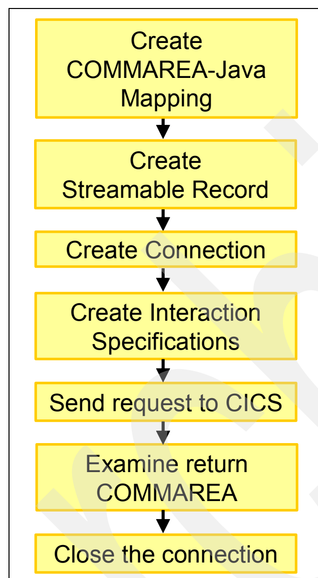


Figure 4-1 Tasks required of the application

Archived

We will develop three Java classes to perform the functions required for this application:

- CustProgCommarea

  Performs the COBOL Structure to Java mapping.

- JavaStringRecord

  Functions as a carrier of the COMMAREA to the J2C Connections.

- CustProgGetSingleJ2C

  Contains the main method and methods to perform all the tasks shown in Figure 4-1 on page 74.

Our program will use Java APIs from these external packages:

- javax.resource.cci

  A collection of CCI classes, laid down as part of JCA specifications.

- javax.resource

  To import the ResourceException class.

- com.ibm.connector2.cics

  Classes specific to the CICS resource adapters which implement the CCI specifications.

- java.io

  To import the IOException class. IOException may be thrown during streaming operations.

Use import statements to reference the necessary classes from the above packages.

Create the COMMAREA to Java mappings

Our CICS CUSTPROG sample program is written in COBOL and it expects a COMMAREA on input. The COMMAREA is the binary equivalent of a COBOL structure. In this case, the COMMAREA should represent a byte buffer that is equivalent to the COBOL structure shown in Example 4-1 on page 67. The Java class CustProgCommArea enables us to perform this function. Additional information about this class is available in “Analyzing the sample Java client code” on page 68

In order for the sample application to use the *CustProgCommArea* class, the following steps must be followed:

1. Create an object of the CustProgCommArea class.
2. Set the required properties to change the state.
3. Use the toString method to convert the object to its COMMAREA equivalent.

Archived

Create a streamable record

In the JCA specification, the javax.resource.cci.Interaction class enables a component to execute the CICS EIS functions. The key method for this interface is the execute method, which takes as input the javax.resource.cci.Record and the javax.resource.cci.InteractionSpec. Therefore, we must first instantiate an object of type javax.resource.cci.Record, which encapsulates the COMMAREA and provides the streaming and serialization functions.

In our case, we are using the class JavaStringRecord which is included with the CICS TG provided samples. To use this class in our sample Java client we copied this class into the com.ibm.itso.model package.

1. Create an instance of the class
2. Set the encoding, because the COMMAREA is character-based.
3. Encapsulate the COMMAREA byte buffer using the setText method.
4. Use the getText method to read the data from the record.

Before proceeding, we should review the steps covered so far. The main CustProgGetSingleJ2C class contains the invokeCustProg method, which performs the majority of the sample Java client tasks. We will focus our analysis on this method.

Import statements we used are shown in Example 4-7.

Example 4-7 Import statements

```
package com.ibm.itso.jca;

import javax.resource.ResourceException;
import javax.resource.cci.Connection;
import javax.resource.cci.ConnectionFactory;
import javax.resource.cci.Interaction;
import com.ibm.connector2.cics.ECIInteractionSpec;
import com.ibm.connector2.cics.ECIManagedConnectionFactory;
import com.ibm.itso.model.CustProgCommarea;
import com.ibm.itso.model.JavaStringRecord;
```

The code for the invokeCustProg method, which supports the functions described so far, is shown in Example 4-8 on page 77.

Archived

Example 4-8 invokeCustProg method

```
public void invokeCustProg(String[] args) {
try {
1 CustProgCommarea cpca = new CustProgCommarea();
2 cpca.setRequestType("R");
3 cpca.setCustomerId(theCustomerId);
4 JavaStringRecord jsr = new JavaStringRecord();
5 jsr.setEncoding("IBM037");
6 jsr.setText(cpca.toString());
7 ConnectionFactory cf = createAConnectionFactory();
8 Connection conn = cf.getConnection();
9 is = getMyECIInteractionSpec();
10 Interaction interaction = conn.createInteraction();
11 interaction.execute(is, jsr, jsr);
12 String returnedData = jsr.getText();
13 cpca.setText(returnedData);
14 if (cpca.getReturnCode().equals("00")) {
       System.out.println("Customer id: "+cpca.getCustomerId()+
              ", name: "+cpca.getCustomerFirstName().trim()+
              " "+cpca.getCustomerLastName().trim()+
              ", last updated: "+cpca.getCustomerLastUpdateDate());
   }
15 else {
       System.out.println("The CICS program was accessed but "+
              " a problem was indicated:  the program returned \""+
              cpca.getReturnComment().trim()+
              "\". Return code :: "+cpca.getReturnCode());
   }
```

The logic of the code in Example 4-8 is as follows:

- 1 Create an object of CustProgCommArea.
- 2 Set the request type as R, indicating a read of data from the VSAM file.
- 3 Set the customer’s ID. This variable is set explicitly elsewhere in the application. A sample value can be 00000001.
- 4 Create an instance of streamable record.
- 5 Tell the string record class which encoding scheme we are using.
- 6 Encapsulate the CommArea buffer into the string record.
- 7 Call the private createAConnectionFactory() method to create the Connection factory. See “Create a connection”.
- 8 Invoke the getConnection method on the ConnectionFactory object to obtain an instance of connection, which is configured according to the values set for the CICSManagedConnectionFactory.

Archived

Create a connection

To create the connection in our non-managed application we first need to build our ConnectionFactory. In this sample Java client application we have defined a private method named createAConnectionFactory (Example 4-9). This method creates and returns a ConnectionFactory instance. We use this ConnectionFactory within the invokeCustProg method to create a connection.

> **Note:** This sample is a non-managed JCA application that requires the manual definition and creation of the ConnectionFactory. This is useful for testing purposes. However, for deployment into a J2EE application server such as WebSphere Application Server, we would suggest that the ConnectionFactory be defined within the WebSphere Administration console, and referenced using a resource reference. This provides extra flexibility and exploits the JCA qualities of service for transaction, connection, and security management.

Example 4-9 Creating a ConnectionFactory

```
private ConnectionFactory createAConnectionFactory()
                                    throws ResourceException {
1 if (eciMgdCf == null) {
2 eciMgdCf = new ECIManagedConnectionFactory();
3 eciMgdCf.setConnectionURL(theConnectionURL);
4 eciMgdCf.setPortNumber(thePortNumber);
5 eciMgdCf.setServerName(theServerName);
6 eciMgdCf.setLogWriter(new java.io.PrintWriter(System.err));
7 if (!thePassword.equals("")) {
              eciMgdCf.setPassword(thePassword);
          }
8 if (!theUserName.equals("")) {
              eciMgdCf.setUserName(theUserName);
          }
9 if (debug) {
              int ri = ECIManagedConnectionFactory.RAS_TRACE_INTERNAL;
              eciMgdCf.setTraceLevel(new Integer(ri));
          }
      }
10 return (ConnectionFactory) eciMgdCf.createConnectionFactory();
  }
```

Archived

The code for the `createAConnectionFactory` method performs the following:

- **1** The variable eciMgdCf is declared at the class level, so that the ConnectionFactory is accessible across all methods of this class. First we check if the ConnectionFactory is already created and if not it will create one.
- **2** Instantiate the ECIManagedConnectionFactory class and configure the object. In our example this class is directly instantiated. This class configures the main attributes of the non-managed connection between the Java client application and the CICS TG.
- **3** Set the URL for the CTG gateway daemon using the setConnectionURL method. Our sample application used the URL WTSC59.itso.ibm.com.
- **4** Set the port number on which the gateway daemon is listening using the setPortNumber method. Our sample application used 2006 as the port number.
- **5** Set the CICS server name (as known to the CTG) using the setServerName method. Our sample application used CIC1IPIC as the server name.
- **6** Redirect the log messages to the Error console.
- **7** and **8**, set the security attributes using the setPassword and setUserName methods.
- **9** `If` debug is set on, then start the trace.
- **10** Instantiate and return the javax.resource.cci.ConnectionFactory object. Use the ECIManagedConnectionFactory object previously created to instantiate a class that implements the ConnectionFactory interface. We invoked the createConnectionFactory method of the ECIManagedConnectionFactory object.

Create Interaction Specifications

Instantiate and configure the specific object for the type of interactions that are required. In this case, we used 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. The configuration is performed by invoking its setter methods to populate the attributes of the object.

For our sample Java client application, we created a private method getMyECIInteractionSpec to create the Interaction specifications object. The code for this method is shown in Example 4-10 on page 80.

Example 4-10 Creating Interaction Specification

```
1 private ECIInteractionSpec getMyECIInteractionSpec()
                                throws ResourceException {
2    if (is == null) {
         is = new ECIInteractionSpec();
3        is.setCommareaLength(theCommareaLength);
4        is.setExecuteTimeout(theExecuteTimeoutMilliseconds);
5        is.setFunctionName(theFunctionName);
6        is.setInteractionVerb(ECIInteractionSpec.SYNC_SEND_RECEIVE);
7        is.setReplyLength(theReplyLength);
8        if (!theTPNName.equals("")) {
            is.setTPNName(theTPNName);
         }
9        else if (!theTranName.equals("")) {
            is.setTranName(theTranName);
         }
     }
10   return is;
  }
```

The execution flow is explained below.

- **1** Declare the getMyECIInteractionSpec method.
- **2** The ECIInteractionSpec object is declared at the class level so that it is accessible from the various methods. Create a new ECIInteractionSpec object if it does not already exist.
- **3** Set the COMMAREA length property using the setCommareaLength method. For this sample application the value was calculated to be 309.
- **4** Set the execution time out value. We have set it conservativly to 5000 milliseconds which is 5 seconds.
- **5** Set the function name of the server side program. In our case the name is CUSTPROG.
- **6** Set the Interaction verb to SYNC_SEND_RECEIVE to send data and then wait for the response.
- **7** Set the length of the expected returned COMMAREA data if this is shorter than the defined COMMAREA length.

> **Note:** setReplyLength() should only be used if the COMMAREA data is not null-truncated as this disables the dynamic null truncation of transmitted data by the Gateway daemon.

Archived

- **8** and **9** Set the mirror transaction name or transaction name, if either exists.Both the mirror TPN name and transaction name cannot exist together. Only one of them can be set. TPN overrided tranName.
- **10** Return the ECIInteractionSpec object.

We will use this InteractionSpec object in the next step.

Send the request to CICS TG

To send a request to the CICS TG to be executed in our CUSTPROG sample back-end program requires an Interaction object. This is obtained from the Connection object previously created. Once we have the Interaction object, we need to invoke the execute method within it. When the execute method is invoked on the Interaction object, it opens the socket connection to the gateway daemon and flows the ECI request to CICS over this connection. The parameters that must be supplied to the execute method are as follows:

- An ECIInteractionSpec object
- A JavaStringRecord object as both an input and output parameter.

The section of code that invokes the execute method is shown in Example 4-8 on page 77.

- **9** Get the InteractionSpec object from the private method created in the previous step.
- **10** Create an Interaction object from the previously created connection.
- **11** Invoke the execute method passing the InteractionSpec object and the JavaStringRecord object for for both the input and the output.

The CUSTPROG sample program running in the CICS environment on the server side receives the Read request. It returns the result to the sample Java client in the COMMAREA.

Examining the result

The COMMAREA returned from the CICS CUSTPROG sample application is written to the Record object which in our case is the JavaStringRecord object. We can access it and convert it back to the CustProgCommarea object for further processing. The code for handling the returned data is shown in Example 4-8 on page 77.

Archived

- **12** Get the returned COMMAREA as a String object from the JavaStringRecord object.
- **13** Convert this String object representing the COMMAREA into a CustProgCommarea object by invoking the setText method. The setText method will parse the returned byte buffer and set the properties of the CustProgCommarea object. On completion of the setText method call, the CustProgCommarea object will represent the returned COMMAREA structure.
- **14** A return code of 00 indicates there were no errors while retrieving the data therefore the information returned is displayed on the console.
- **15** If the return code is not 00, it indicates an error during the retrieval of data. In that case, print the return comment and the return code.

Close the connection

Finally, close both the Interaction and the Connection objects. The close method on the Connection object is the final statement in the program and in doing so the underlying socket connection to the gateway daemon is closed. It must be executed after the close method on the Interaction object as shown in Example 4-11

Example 4-11 Closing the connection

```
interaction.close();
conn.close();
```

The significant code elements of our sample Java client CustProgGetSingleJ2C have been analyzed. Additionally, two more methods are used in this class.

- getCommandLineArguments

  The getCommandLineArguments method processes the command line arguments. This sample application expects the server name, port number, and connection URL to be supplied as command line arguments. This method parses the arguments and sets the class variables accordingly. If there is an error with the arguments, it prints the correct usage of the command line arguments to the console.

- printUsageStatement

  The printUsageStatement method prints the correct usage of the command line arguments to the console.

Compiling and Executing

For instructions on compiling and executing this sample, refer to Appendix D, “CICS samples” on page 243.

Archived

4.3 Channel-based applications

In the following sections we demonstrate the development of channel-based sample Java and J2C client applications.

Appendix E explains the corresponding sample CICS server programs used, and how to compile and install it into the back-end CICS TS environment.

4.3.1 Sample channel-based Java client development

This section explains how our sample Java client program accesses customer records in VSAM by calling the CICS server program CUSTBRWS using channels and containers. The source code is shown in Appendix B, “Java Sample Code” on page 191.

The sample Java client program uses base ECI API to call the CICS server program using the channels and containers mechanism to send data to and receive data from the CUSTBRWS CICS server program.

The following Java files are used by our channels and containers-based sample client application:

- com.ibm.itso.eci.CustProgMultiRecBaseClasses.java

  This file is the main program which performs processing of the data and communications with CICS.

- com.ibm.itso.model.CustProgRequestInfo.java

  This file is used to construct the input container being sent to the CICS CUSTBRWS program.

- com.ibm.itso.model.CustProgRecord.java

  This file is used to store the customer information details contained in the output container returned by the CICS CUSTBRWS program.

- com.ibm.itso.model.CustProgErrorInfo.java

  This file is used for error processing.

Compiling the sample Java client program

For instructions on compiling and executing the Java client program, refer to Appendix D, “CICS samples” on page 243.

The output is shown in Example 4-12 on page 84.

Example 4-12 Usage of the channels-based sample java program

```
--- CustProgMultiRecBaseClasses - Using Channels and Containers ---
This program can be used to invoke the CUSTBRWS CICS-based COBOL
program to retrieve multiple records through the specified CICS TG.
Three positional arguments are required.  The CICS TG Base Classes are
used in this program.

Usage: CustProgMultiRecBaseClasses <ServerConnectionName> <Port>
<Gateway>

    Example: CustProgMultiRecBaseClasses CIC1IPIC 2006
tcp://wtsc59.itso.ibm.com

   startid:xxxxxxxx  - The program will try to get account information
starting with customer id "00000000".  You can override this by adding
startid:xxxxxxxx after the 3 positional parameters, where xxxxxxxx is
the first customer id you would like returned.  The Customer Id length
is 8 characters.  If less than 8 characters are specified, they will be
padded out to 8 with blanks

   return:nn  - The program will try to return 5 customer accounts, but
you can override this by adding return:nn after the 3 positional
parameters.  The max number of records that can be returned is
currently 200. (where nn is the number of accounts you want returned.)

   direction:B  - The program will try to browse forward on the VSAM
file, but you can override this by adding direction:B after the 3
positional parameters.  (B is for Backword).

   keylen:n  - The program will assume that the keylength is 8, but you
request the browse start with the first record on the file by
specifying keylen:0 after the 3 positional parameters.  You can also
specify a GENERIC browse by specifying keylen:n, where n is the number
of characters in the generic key.  If greater than 8 is specified, 8 is
assume d.
   Specifying keylen:0 takes precidence over startid:xxxxxxxx.
```

How the sample Java client program works

Following are the steps on how the sample Java client make the calls to the CICS CUSTBRWS sample program to retrieve multiple records using a channels and containers interface.

1. The CustProgMultiRecBaseClasses.java is used for the main processing and flowing of the requests. Data definitions used are shown in Example 4-13 on page 85.

Example 4-13 Data definitions used by CustProgMultiRecBaseClasses.java

```
private static JavaGateway javaGatewayObject;
private static String CHANNEL_NAME = "MYCHANNEL";
private static String REQUEST_INFO_CONTAINER = "BROWSE-INFO";
private static String ERROR_CONTAINER = "ERROR";
private static String NUM_RECS_CONTAINER = "NUM-RECORDS";

// request settings
private int maxRecordsToReturn = 5;
private String direction = "F";
private String startingKey = "00000000";
private int startingKeyLength = 8;

// location settings:
private String theConnectionURL = "";
private String thePortNumber = "";
private String theServerName = "";
private String theFunctionName = "CUSTBRWS";
```

   – The javaGatewayObject is used for communicating with Gateway daemon. Initially we need three containers:

     - BROWSE-INFO is used for the input container,
     - ERROR is used for the error container,
     - NUM-RECORDS is used to determine the number of records fetched by the server program.

   – The input container variables are declared with default values.

   – The theConnectionURL, thePortNumber, and theServerName values are obtained from the command line arguments. The theFunctionName is provided as CUSTBRWS, which is the name of the CICS program performing the server side logic.

2. The CustProgRequestInfo.java program constructs the input containers passed to the CICS CUSTBRWS program. The source for this program is available in Appendix E, “Additional material” on page 253. Data definitions are shown in Example 4-14.

Archived

Example 4-14 Java input container for CICS CUSTBRWS program

```
   private static String BLANKS8  = "        ";
   private static String ZEROS8  = "00000000";
   private static String DEFAULT_MAX_RECORDS  = "0001";
   private static String DEFAULT_DIRECTION  = "F";
   private static String DEFAULT_START_KEY_LENGTH  = "08";

1  private String maxRecords = DEFAULT_MAX_RECORDS;
2  private String direction = DEFAULT_DIRECTION;
3  private String startKey = BLANKS8;
4  private String startKeyLength = DEFAULT_START_KEY_LENGTH;
```

- **1** The maximum number of records to be returned is specified in the maxRecords variable.
- **2** The direction of the browse is specified in the direction variable (B=backward, F=forward).
- **3** Specify the starting key in the startKey variable.
- **4** Specify the length of the key in the startKeyLength variable.

3. Look at the data structure within the CustProgRecord.java file shown in Example 4-15. The source for this program is available in Appendix E, "Additional material" on page 253. This structure holds the customer information returned by the CICS CUSTBRWS server program.

Example 4-15 Customer information record container

```
   private static String BLANKS30 = "                              ";
   private static String BLANKS20 = "                    ";
   private static String BLANKS8  = "        ";

   private String customerId = BLANKS8;
   private String customerLastName = BLANKS20;
   private String customerFirstName = BLANKS20;
   private String customerCompany = BLANKS30;
   private String customerAddr1 = BLANKS30;
   private String customerAddr2 = BLANKS30;
   private String customerCity = BLANKS20;
   private String customerState = BLANKS20;
   private String customerCountry = BLANKS30;
   private String customerMailCode = BLANKS20;
   private String customerPhone = BLANKS20;
   private String customerLastUpdateDate = BLANKS8;
```

Archived

4. The CustProgErrorInfo.java data structure used for storing the error code and related comments is shown in Example 4-16 below. The source for this program is available in Appendix E, “Additional material” on page 253.

Example 4-16 Error code and error comments container

```
private static String BLANKS50 = " ";
private static String BLANKS2  = "  ";

private String requestReturnCode = BLANKS2;
private String requestReturnComment = BLANKS50;
```

5. A JavaGateway object is created for communicating with the Gateway daemon as shown in Example 4-17.

Example 4-17 Creating JavaGateway object in CustProgMultiRecBaseClasses

```
try {
javaGatewayObject = new JavaGateway(theConnectionURL,
Integer.parseInt(thePortNumber));
catch (IOException e1) {
System.out.println("IOException trying to communicate to the CICS TG at
"+theConnectionURL+", Port "+thePortNumber+":"+e1.getMessage());
return;
}
```

6. Populate the CustProgRequestInfo object data structure as shown in Example 4-18.

Example 4-18 input container data in CustProgMultiRecBaseClasses

```
CustProgRequestInfo cri = new CustProgRequestInfo();
cri.setMaxRecords(maxRecordsToReturn);
cri.setDirection(direction);
cri.setStartKey(startingKey);
cri.setStartKeyLength(startingKeyLength);
```

7. A channel must be created to send containers to the CICS program. Create an input container and add the container to the channel just created, as shown in Example 4-19.

Example 4-19 Creating channel and container objects in CustProgMultiRecBaseClasses

```
//  create a channel
        Channel reqChannel = null;
      try {
         reqChannel = new Channel(CHANNEL_NAME);
      } catch (ChannelException e1) {
```

```
        System.out.println("ChannelException while trying to create a
channel: "+e1.getMessage());
        System.out.println("Terminating!");
        System.exit(1);
      }

      // Place the request data in a REQUEST_INFO_CONTAINER in
CHANNEL_NAME
        try {
        reqChannel.createContainer(REQUEST_INFO_CONTAINER,
cri.toString());
      } catch (UnsupportedEncodingException e1) {
        System.out.println("UnsupportedEncodingException while trying
to add the REQUEST_INFO_CONTAINER to the channel: "+e1.getMessage());
        System.out.println("Terminating!");
        System.exit(1);
      } catch (ContainerException e1) {
        System.out.println("ContainerException while trying to add the
REQUEST_INFO_CONTAINER to the channel: "+e1.getMessage());
        System.out.println("Terminating!");
        System.exit(1);
      }
```

8. Create the ECIRequest object as shown in Example 4-20. The ECIRequest constructor is passed channel as an input parameter instead of a COMMAREA.

Example 4-20 ECIRequest object with channel parameter within CustProgMultiRecBaseClasses

```
// specify the details of the where and how you want to connect
eciRequestObject = new ECIRequest(
1   ECIRequest.ECI_SYNC,        //ECI call type
2   theServerName,              //CICS server
3   null,                       //CICS username
4   null,                       //CICS password
5   theFunctionName,            //Program to run
6   null,                       //Transaction to run
7   reqChannel,                 //Channel
8   ECIRequest.ECI_NO_EXTEND,   //ECI extend mode
9   0                           //ECI LUW token
);
```

- 1 Specifies this is a synchronous ECI request.
- 3 and 4 The user ID and passwords are provided as null because security is not enabled for our CICS region.
- 5 The theFunctionName parameter is the CICS server program name this sample Java client will call.
- 7 The seventh parameter is set to channel.
- 8 This specifies this to be a non-extended ECI request.
- 9 The LUW parameter is supplied as 0 because we are not making any extended calls.

9. The ECIRequest object must be sent to the CICS TG. This is accomplished by the javaGatewayObject flow method which takes the eciRequestObject as a parameter as shown in Example 4-21

Example 4-21 CustProgMultiRecBaseClasses flowing the ECIRequest object to the CICS TG

```
// make the request
        int iRc = 0;
      try {
1        iRc = javaGatewayObject.flow(eciRequestObject);
      } catch (IOException e3) {
         System.out.println("IOException encountered trying to invoke
the "+theFunctionName+" program. Exception was:"+e3.getMessage());
         return;
      }

      //  if we had an error - try to tell the user what happened
2 switch (eciRequestObject.getCicsRc()) {
        case ECIRequest.ECI_NO_ERROR:
        if (iRc == 0) {
         //System.out.println("No error from ECI request");
         break;
        } else {
         System.out.println("\nError from Gateway
("+eciRequestObject.getRcString()+").");
         if (javaGatewayObject.isOpen() == true) { try {
javaGatewayObject.close(); } catch (Throwable t2) { ; } }
               return;
            }
         case ECIRequest.ECI_ERR_SECURITY_ERROR:
            System.out.print("\n\nSecurity problem communicating with
the CICS TG. ");
            return;
```

```
        case ECIRequest.ECI_ERR_NO_CICS:
           System.out.println("\n\nError no CICS with the specified
name.");
           return;
        case ECIRequest.ECI_ERR_TRANSACTION_ABEND:
           System.out.println("\nA transaction ABEND was reported.");
           System.out.println("\nECI returned: " +
eciRequestObject.getCicsRcString());
           System.out.println("Abend code was " +
eciRequestObject.Abend_Code + "\n");
           if (javaGatewayObject.isOpen() == true) { try {
javaGatewayObject.close(); } catch (Throwable t2) { ; } }
           System.exit(1);
        default:
           System.out.println("\nA transaction ABEND was reported.");
           System.out.println("\nECI returned: " +
eciRequestObject.getCicsRcString());
           System.out.println("Abend code was " +
eciRequestObject.Abend_Code + "\n");
           if (javaGatewayObject.isOpen() == true) { try {
javaGatewayObject.close(); } catch (Throwable t2) { ; } }
           System.exit(1);
     }

       //System.out.println("ECI - CicsRc
is:"+eciRequestObject.getCicsRc()+"="+eciRequestObject.getCicsRcString(
));

       // all is fine with the communication, close the gateway object
3 if (javaGatewayObject.isOpen() == true) { try {
javaGatewayObject.close(); } catch (Throwable t1) { ; } }

       // if the call didn't get a channel back, tell someone and quit
       if ( ! eciRequestObject.hasChannel()) {
       System.out.println("\nThe call to "+theFunctionName+" did not
return a channel, terminating!");
        System.exit(1);
       }
```

- **1** Because this is a synchronous request, the flow method blocks until the server program CUSTBRWS completes the request.
- **2** The flow method returns an integer containing the status of the request. The code checks for a successful return code 0(ECI_NO_ERROR). If the request was not successful, it checks for a security error, an invalid CICS destination, or a transaction abend error. If none of these errors are returned, it displays the error code along with the error description.
- **3** It closes the gateway connection by calling the close method of the JavaGateway object.

10. The eciRequestObject returned the channel from the CUSTBRWS CICS sample program. We will demonstrate how it is processed for the various types of containers. The code for processing returned containers is shown in Example 4-22.

Example 4-22 CustProgMultiRecBaseClasses—Process output containers received from the CUSTBRWS CICS sample program

```
// we have a returned channel - lets start processing it
Channel respChan = eciRequestObject.getChannel();
// Display the returned content
Container cont = null;
try {
1 cont = respChan.getContainer(ERROR_CONTAINER);
CustProgErrorInfo cei = null;
if (cont.getType() == Container.ContainerType.BIT) {
cei = new CustProgErrorInfo(cont.getBITData(), "037");
} else {
cei = new CustProgErrorInfo(cont.getCHARData());
}
System.out.println("Response from "+theFunctionName+" was
"+cei.getRequestReturnCode()+" - "+cei.getRequestReturnComment());
   System.exit(1);
2 } catch (ContainerNotFoundException e) {
        // this is the 'normal' case - don't do anything
// Other execption handling code follows here ...

// get the number of data records returned
int numRecsReturned = 0;
try {
3 cont = respChan.getContainer(NUM_RECS_CONTAINER);
if (cont.getType() == Container.ContainerType.BIT) {
byte[] bNum = cont.getBITData();
try {
String tempStr = new String(bNum, "037");
```

```
numRecsReturned = Integer.parseInt(tempStr);
} catch (UnsupportedEncodingException e) {}

// Other Exception handling code continue here ...

String strContNum = "";
4 for (int i = 0; i < numRecsReturned; i++) {
strContNum = ("0000"+(i+1)).substring(("0000"+(i+1)).length()-4);
try {
cont = respChan.getContainer(strContNum);
CustProgRecord cpr = null;
if (cont.getType() == Container.ContainerType.BIT) {
cpr = new CustProgRecord(cont.getBITData(), "037");
} else {
cpr = new CustProgRecord(cont.getCHARData());
}
System.out.println("Customer id: "+cpr.getCustomerId()+", name:
"+cpr.getCustomerFirstName().trim()+"
"+cpr.getCustomerLastName().trim()+", last updated:
"+cpr.getCustomerLastUpdateDate());
        }catch (ContainerNotFoundException e) {}

// Other Exception handling code continues here ...
```

- **1** The code first checks for an ERROR container returned by the server program if there were problems encountered while handling the request.
- **2** If the ERROR container does not exist the getContainer method throws a ContainerNotFoundException which is expected because normally we would not receive an ERROR container.
- **3** The number of records returned by the server program is available in the container NUM_RECS_CONTAINER.
- **4** Once you determine the number of records and the names of the containers you can retrieve them by calling the getContainer method of the channel object. Remember the CUSTBRWS CICS server program uses the record container names 0001, 0002, etc.

> **Note:** For each container received, the program checks whether it is BIT or CHAR container, If the returned container is of type BIT, the program converts the character data from EBCDIC to Unicode by passing 037 as the encoding value.

Archived

4.3.2 Sample channel-based JCA client development

This section describes a sample Java client program CustProgMultiRecJ2C, which uses JCA, also known as J2C APIs, to interact with the back-end CICS CUSTBRWS sample program. The sample Java client uses channels and containers as the exchange medium between the client and server. The source code is shown in Appendix B, “Java Sample Code” on page 191.

How our sample application works

The functions performed by this sample application are better understood by referring to Figure 4-2 on page 93, which shows the different objects and their roles in the application.

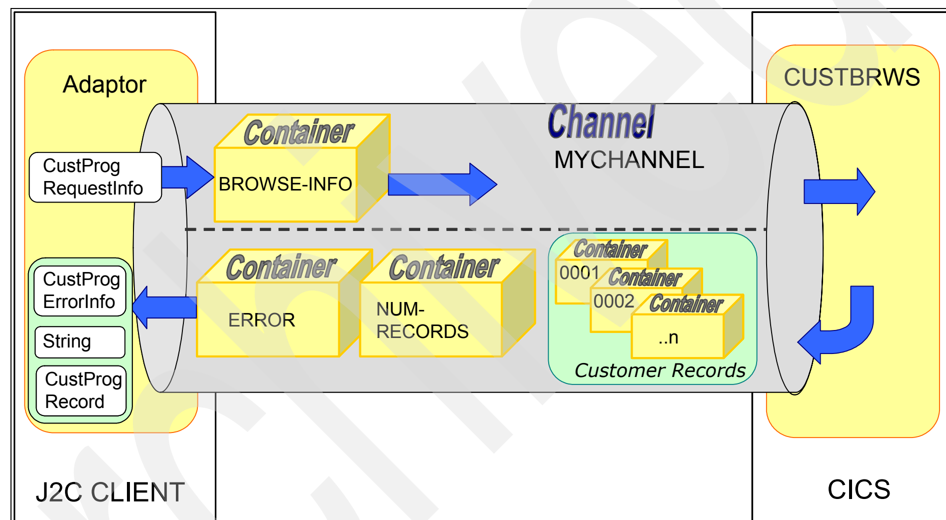


Figure 4-2 CustProgMultiRecJ2C application working

A channel is created between the J2C client and server side CICS. This channel is identified by the name MYCHANNEL. From the client, an outbound container named BROWSE-INFO is sent to the server through this channel. A Java object, CustProgRequestInfo, maps this container to the client Java environment. At the server side, CUSTBRWS program receives this container and processes it. The result of this processing is bundled in multiple containers named ERROR, NUM-RECORDS, and a series of containers with dynamic names starting from 0001. The ERROR container is optional and will be sent only when there is an error detected by the server side while processing the request. These containers are routed back to the J2C client, through the same channel.

The client receives the containers sent from the server and maps them to the Java environment through instances of CustProgErrorInfo, String, and CustProgRecord classes respectively. The J2C client program can then process the data encapsulated within these objects.

Application tasks analysis

The various tasks the CustProgMultiRecJ2C sample Java client must perform are shown in Figure 4-3 on page 94.

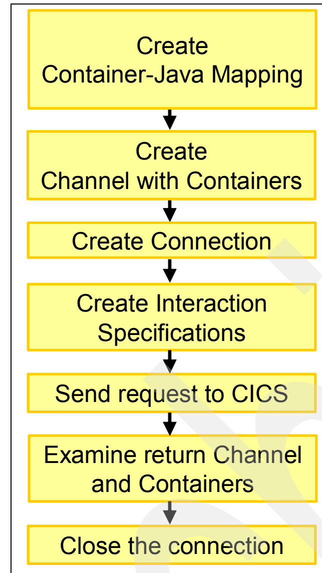


Figure 4-3 Tasks required of the CustProgMultiRecJ2C Java client application

We will create four Java classes for the client application:

- CustProgMultiRecJ2C

  Contains the main method and methods to perform all the tasks shown in Figure 4-3. This class will use instances of other classes.

- CustProgRequestInfo

  Java representation of a container named BROWSE-INFO.

- CustProgRecord

  Representation of one container, which encapsulates one fetched record from VSAM file in back-end CICS.

- CustProgErrorInfo

  Java representation of a container named ERROR.

Archived

Our Java client application also uses Java APIs from these external packages:

- javax.resource.cci

  This package is a collection of CCI classes, laid down as part of JCA specifications.

- javax.resource

  This package imports the ResourceException class.

- com.ibm.connector2.cics

  This package is classes specific to the CICS resource adapters which implement the channels and containers APIs.

- java.io

  This package contains the IOException class. IOException may be thrown during streaming operations.

The sample Java client application is started by invoking the main method in the CustProgMultiRecJ2C class. In the CustProgMultiRecJ2C class we defined a private method invokeCustBrws, which performs the tasks listed in Figure 4-3 on page 94. We will therefore focus on this invokeCustBrws method. We touch upon the functions of other classes during the analysis of the code in the CustProgMultiRecJ2C class.

The import statements used in our sample CustProgMultiRecJ2C Java client are shown in Example 4-23.

Example 4-23 CustProgMultiRecJ2C import statements

```
package com.ibm.itso.jca;

import java.io.UnsupportedEncodingException;
import javax.resource.ResourceException;
import javax.resource.cci.Connection;
import javax.resource.cci.ConnectionFactory;
import javax.resource.cci.Interaction;
import com.ibm.connector2.cics.ECIChannelRecord;
import com.ibm.connector2.cics.ECIInteractionSpec;
import com.ibm.connector2.cics.ECIManagedConnectionFactory;
import com.ibm.itso.model.CustProgErrorInfo;
import com.ibm.itso.model.CustProgRecord;
import com.ibm.itso.model.CustProgRequestInfo;
```

Create container-Java mapping

The back-end CICS sample program CUSTBRWS is written in COBOL and expects a channel to be passed as the input. The channel is named MYCHANNEL and includes a container named BROWSE-INFO. The BROWSE-INFO container is the binary equivalent of the COBOL structure shown in Appendix D, “CICS samples” on page 243. Because we are sending this data from a Java client program, we need a Java class that logically represent the container's structure in Java and converts its data into a byte buffer, which is the equivalent of the BROWSE-INFO COBOL structure. We invoked the Java class CustProgRequestInfo to perform this task. We will show how this mapping is achieved.

The BROWSE-INFO structure shown in Appendix D, “CICS samples” on page 243 defines four variables. Each of these occupies a fixed number of bytes, as shown in Table 4-1.

Table 4-1 Field names, types, and number of bytes

Field Name	Type	Number of bytes
Max-Records	PIC 9999	4
Direction	PIC X	1
Start-With-This-Key	PIC X(8)	8
Start-Key-Length	PIC 99	2

The byte buffer, which represents this container, can be visualized as shown in Figure 4-4.

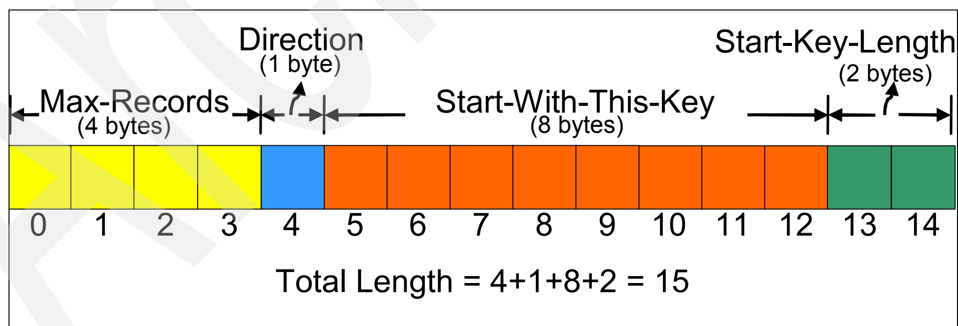


Figure 4-4 Byte buffer for BROWSE-INFO container

Archived

One of the mechanisms for creating this byte buffer is to use Java String objects. To do this, perform the following steps:

1. Create individual String objects for each field. The length of the String object should be exactly the same as the length of the field in bytes.
2. Create an aggregate String object, by concatenating the individual String objects created in step 1. The length of this aggregate String will be 15.
3. Encode the String object created in step 2 to EBCDIC. Because Java does not have direct support for EBCDIC, use IBM-037 as the encoding scheme.
4. Convert the String to byte array using getBytes() method in String object.

Referring to Example 4-14 on page 86, we see that the fields of CustProgRequestInfo are initialized towards achieving this objective. The setter methods make sure that the String objects are always of fixed byte length.

Conversely, when a String object, which represents the byte buffer is supplied to the CustProgRequestInfo object, the respective fields need to be extracted from it based on the byte length of individual fields. The setText method shown in Example 4-24 accomplishes this.

Example 4-24 Extracting container fields

```
public void setText(String totData) {
1    setMaxRecords(totData.substring(0, 4));
2    setDirection(totData.substring(4, 5));
3    setStartKey(totData.substring(5, 13));
4    setStartKeyLength(totData.substring(13, 15));
}
```

- 1 From the aggregate String object, extract the first 4 bytes, which corresponds to Max-Records
- 2 Extract the fifth byte, which represents the Direction field
- 3 Extract the next 8 bytes, that is, bytes starting from index number 5 till 13. This corresponds to Start-With-This-Key field.
- 4 Extract the last two bytes which represents the Start-Key-Length filed.

The complete source for this program is available in Appendix E, “Additional material” on page 253.

Archived

Note: Classes such as CustProgRequestInfo, which bridge the gap between non-Java data structures and Java classes are known as data binding classes. For large and complex non-Java structures, the creation of data binding classes can become tedious. Development tools, such as Rational Application Developer provide simple interfaces for rapid generation of data binding classes from non-Java data structures. See Chapter 5, “J2C Application Development With Rational Application Developer” on page 105

In our sample application, creating the channel with its containers is accomplished in three steps:

1. Create a channel object using the ECIChannelRecord.
2. Create the container.
3. Add the container to the channel.

The code snippet is shown in Example 4-25.

Example 4-25 Creating channel and container within CustProgMultiRecJ2C.

```
1 ECIChannelRecord myChannel = new ECIChannelRecord(CHANNEL_NAME);
2 CustProgRequestInfo cri = new CustProgRequestInfo();
  cri.setMaxRecords(maxRecordsToReturn);
  cri.setDirection(direction);
  cri.setStartKey(startingKey);
  cri.setStartKeyLength(startingKeyLength);
3 if (startingKeyLength == 0) {
     System.out.println("Requesting a maximum of "+maxRecordsToReturn+
         " records starting with the first customer id on the file.");
  }
  else if (startingKeyLength > 7) {
     System.out.println("Requesting a maximum of "+maxRecordsToReturn+
             " records starting with customer ids that begin with \""+
             cri.getStartKey() + "\".");
  }
  else {
     System.out.println("Requesting a maximum of "+maxRecordsToReturn+
             " records starting with customer ids that begin with \""
             +cri.getStartKey().substring(0,startingKeyLength)+ "\".");
  }
  if (direction.equalsIgnoreCase("b")) {
     System.out.println("The sequence will be descending.");
  }
  else {
     System.out.println("The sequence will be ascending.");
  }
4 myChannel.put(REQUEST_INFO_CONTAINER, cri.toString());
```

Archived

The program logic flow is:

- **1** Create an Instance of channel using ECIChannelRecord.
- **2** Create the container instance and set the properties. Using the respective setter methods we set the maximum records to be returned from the server, the sorting direction, the starting key and length of the starting key.
- **3** Print some useful information onto the console for reference.
- **4** Use the toString method of the CustProgRequestInfo object to convert the object to its container equivalent and add the container to the channel just created. The REQUEST_INFO_CONTAINER variable is defined at the class level as a constant and its value is set to BROWSE-INFO, which is the name of our container.

Flow the request to CICS

The channel and its container must be sent to our sample back-end COBOL program running in CICS. The code that performs this within the CustProgMultiRecJ2C application is shown in Example 4-26 on page 99. This is a continuation of the code shown in Example 4-25 on page 98.

Example 4-26 Sending the ECI request to CICS

```
5 ConnectionFactory cf = createAConnectionFactory();
  Connection conn = cf.getConnection();
6 ECIInteractionSpec is = getMyECIInteractionSpec();
7 Interaction interaction = conn.createInteraction();
8 interaction.execute(is, myChannel, myChannel);
```

- **5** To create the Connection, we require a ConnectionFactory. In our sample application, we defined a private method named createAConnectionFactory. This method creates and returns a ConnectionFactory. We use this ConnectionFactory within the invokeCustBrws method to create a Connection. The code and detailed explantion of the createAConnectionFactory method is described in "Create a connection" on page 78. We use the createAConnectionFactory method in this sample Java client as well.
- **6** Instantiate and configure the specific object for the type of interaction 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. The configuration is performed by invoking its setter methods to populate the attributes of the object. In this sample client application, we created a private method getMyECIInteractionSpec to create the Interaction specifications

object. The code and explanation of the getMyECIInteractionSpec method is described in “Create Interaction Specifications” on page 79. We will use the getMyECIInteractionSpec method in this application as well.

- **7** Create an Interaction object from the previously created connection.
- **8** To send the request to CICS to be executed by the back-end Cobol program CUSTBRWS, we need an Interaction object. This is obtained from the Connection object that we previously created. Once we have the Interaction object, we need to invoke the execute method within it. The execute method invoked on the Interaction object opens the socket connection to the Gateway daemon and flows the request to CICS. The execute method takes three parameters:
  - An InteractionSpec, created previously.
  - A channel object as both an input and output parameter. In this case, an instance of the ECIChannelRecord, which was created earlier.

The sample CUSTBRWS COBOL program running in the CICS environment on the server side receives the request. After executing the request it returns the result to the client using the channel. The channel may contain one or more containers within it.

Examining the results received.

The channel returned from the CUSTBRWS back-end CICS program to our Java client contains one or more of the following containers:

- A container named ERROR, which if present, contains information about any errors that occurred during data retrieval at the server side CICS program.
- A container named NUM-RECORDS, which contains information about the number of records retrieved from the VSAM file at the server side.
- A variable number of containers whose names are derived based on the number of records fetched. The server side CICS program in our sample application creates a container for each record fetched from the VSAM file.

Containers within the channel are searched and fetched based on the container name. We examine the returned channel by searching for an ERROR container, which if non-existent, confirms the successful reading of records by the server side CICS program. Code depicting this check is shown in Example 4-27 on page 101. This code is a continuation of Example 4-26 on page 99.

Archived

Example 4-27 Checking the ERROR container

```
9 if (myChannel.containsKey(ERROR_CONTAINER)) {
      CustProgErrorInfo cei = null;
10    containerObj = myChannel.get(ERROR_CONTAINER);
11    if (containerObj instanceof byte[]) {
          try {
            cei = new CustProgErrorInfo((byte[]) containerObj, "037");
          }
          catch (UnsupportedEncodingException e) {
              System.out.println("UnsupportedEncodingException "+
                "converting byte array to specified code page:"
                + e.getMessage());
              System.out.println("Terminating!");
              System.exit(1);
          }
      }
12    else if (containerObj instanceof String) {
          cei = new CustProgErrorInfo((String) containerObj);
      }
13 System.out.println("Response from " +
                          theFunctionName + " was " +
                          cei.getRequestReturnCode()
                          + " - " + cei.getRequestReturnComment());
      System.exit(1);
  }
```

The program logic flow is:

- **9** In the returned channel search for the existence of any container with the name ERROR using the containsKey method. The variable ERROR_CONTAINER is defined at the class level as a static constant with a value of ERROR.
- **10** If an ERROR container exists, fetch the container from the channel by passing the container name to the get method. The container type is generally not known at the time of fetching, so it is assigned to a variable of type java.lang.Object rather than to a specific object type. Therefore, the variable containerObj is of type java.lang.Object.
- **11** Check if the container is a bit container. If this is the case, attempt to decode it using code page 037, and create the container's Java equivalent object CustProgErrorInfo. If the conversion and creation of the CustProgErrorInfo object fails, display the error information on the console.

- **12** Check if the container is a String container, that is, a character container. If this is the case, type `cast the container to String` and create the container's Java equivalent object CustProgErrorInfo.
- **13** Display the error information on the console.

When the channel does not have a container named ERROR, it means that the request operation sent to the server side CICS was successful. In that case we need to determine the number of records returned. The information about the number of records fetched is embedded in the container named NUM-RECORDS. The code snippet for determining this number is shown in Example 4-28. This is a continuation of the code shown in Example 4-27 on page 101.

Example 4-28 Determine number of records returned

```
else {
    int numRecsReturned = 0;
14  containerObj = myChannel.get(NUM_RECS_CONTAINER);
15  if (containerObj instanceof byte[]) {
        System.out.println("The object returned from the " +
                NUM_RECS_CONTAINER + " was a byte[]");
        byte[] bNum = (byte[]) containerObj;
16      try {
            String tempStr = new String(bNum, "037");
            numRecsReturned = Integer.parseInt(tempStr);
        }
17      catch (UnsupportedEncodingException e) {
            System.out.println("UnsupportedEncodingException"+
            " while converting NUM_RECS byte array to a String" +
             e.getMessage());
            System.out.println("Terminating!");
            System.exit(1);
        }
    }
18  else if (containerObj instanceof String) {
        System.out.println("The object returned from the " +
                NUM_RECS_CONTAINER + " was a String");
        numRecsReturned = Integer.parseInt((String) containerObj);
    }
19  else {
        System.out.println("Unknown object type returned from the "
                + NUM_RECS_CONTAINER + ".");
        System.out.println("Terminating!");
        System.exit(1);
    }
20  System.out.println("The number of records returned was: " +
                       numRecsReturned);
```


Archived

Logic for this section of the code is as follows.

- **14** From the channel, fetch the container named NUM-RECORDS.
- **15** Check whether the container is a raw byte array, for example, a bit container.
- **16** If it is a byte array, try to convert it into a string, using 037 as the encoding scheme. If the conversion is successful, convert the string to an integer. This will be the number of records returned by the server side CICS program.
- **17** If the decoding of the string fails, display the error information on the console and exit the program.
- **18** If the container is not a raw byte array, check if it is of type String, for example, a character container. If this is the case, convert the String into an integer. The value of this integer is the number of records returned by the server side CICS program.
- **19** If the container is neither a bit container nor a character container, an error may have occurred. Output an error message and exit the program.
- **20** All the steps have succeeded therefore the number of records returned by the server side CICS program are displayed on the console.

Each record returned is embedded in a separate container. Once we know the number of records returned, we can generate the name of the container dynamically and retrieve each of these containers from the channel, as shown in Example 4-29.

Example 4-29 Retrieving the records

```
String strContNum = "";
for (int i = 0; i < numRecsReturned; i++) {
    String rc = "0000" + (i + 1);
21  strContNum = (rc).substring(rc.length() - 4);
    try {
22      containerObj = myChannel.get(strContNum);
        CustProgRecord cpr = null;
23      if (containerObj instanceof byte[]) {
            cpr = new CustProgRecord((byte[]) containerObj, "037");
        }
        else {
            cpr = new CustProgRecord((String) containerObj);
        }
24      System.out.println("Customer id: " + cpr.getCustomerId() +
                ", name: " + cpr.getCustomerFirstName().trim() +
                " " + cpr.getCustomerLastName().trim()
                + ", last updated: " +
                cpr.getCustomerLastUpdateDate());
```

Archived

```
    }
25  catch (UnsupportedEncodingException e) {
        System.out.println("UnsupportedEncodingException while"+
                " converting BIT container " +
                strContNum + " to a String:" + e.getMessage());
    }
}
```

Let’s examine this code:

- **21** Based on the total number of records returned, create a container name dynamically. For example, the container name for the first and ninety ninth records will be 0001 and 0099 respectively.
- **22** Get the container with this name from the channel
- **23** Check if this is a bit container or a character container. Based on its contents, convert the container into a Java equivalent class either using encoding or directly.
- **24** Display details of the record.
- **25** If the program throws an exception, display error details on the console.

Close the connection

Close both the Interaction and the Connection objects. The close method on the Connection object is the final statement in the program flow. It closes the underlying socket connection to the gateway daemon. It must be executed after the close method on the Interaction object, as shown in Example 4-30.

Example 4-30 Closing the connection

```
interaction.close();
conn.close();
```

Archived

5

J2C Application Development With Rational Application Developer

With common standard architectures like the ones laid down as part of J2EE specifications, it is the support from the tools that makes the difference when it comes to efficient application development. It is the tool that catalyzes the transformation of the architecture into a product.

IBM Rational Application Developer is an integrated development environment (IDE) that provides a seamless environment to design, build, and deploy J2EE applications. This chapter focuses on leveraging the power of this IDE in developing J2EE applications for interacting with CICS. We guide you through the different steps in the development process and conclude with the deployment of the application to a WebSphere Application Server.

© Copyright IBM Corp. 2009. All rights reserved.

5.1 Rational Application Developer

A typical product development cycle involves the development of smaller components independently and integrating them to produce the application. When an enterprise's product marketing approach may change from day to day, it becomes imperative to shrink the time to market for that product. Application developers, in support of this dynamic environment, require tools that can make their application development process faster and easier. Creating enterprise class applications that are state of the art, flexible, and scalable, requires the right set of tools. This is where rapid application development tools and an IDE become a necessity.

An IDE is a set of tools that support application development. It enables software development teams to complete large projects faster and with a high level of quality.

Rational Application Developer (RAD) is an industry-leading IDE for developing Java and J2EE-based applications. It is built on the Eclipse platform and extends the base functionality through the use of plug-ins. The easy-to-use wizards and templates generate high-quality, resilient, scalable applications that are optimized to run on WebSphere Application Server and IBM WebSphere Portal Server products.

The samples in this chapter were developed using Rational Application Developer V7.5. Rational Application Developer V7.5 includes enhancements for developing applications based on the Connector (J2C) architecture. We demonstrate how a J2EE application developer can use this IDE to generate a J2C bean that allows access to functions provided by the back-end CICS system.

5.2 Getting Started

To repeat the development steps in this chapter requires Rational Application Developer V7.5 with WebSphere Application Server Test Environment V7.0 and J2C tools be installed. Although the installation of the WebSphere Application Server test environment is not mandatory, it simplifies the deployment and testing of the application. Because the RAD V7.5 installer does not install the J2C tools by default, we needed to explicitly select this feature as shown in Figure 5-1 on page 107.

Archived

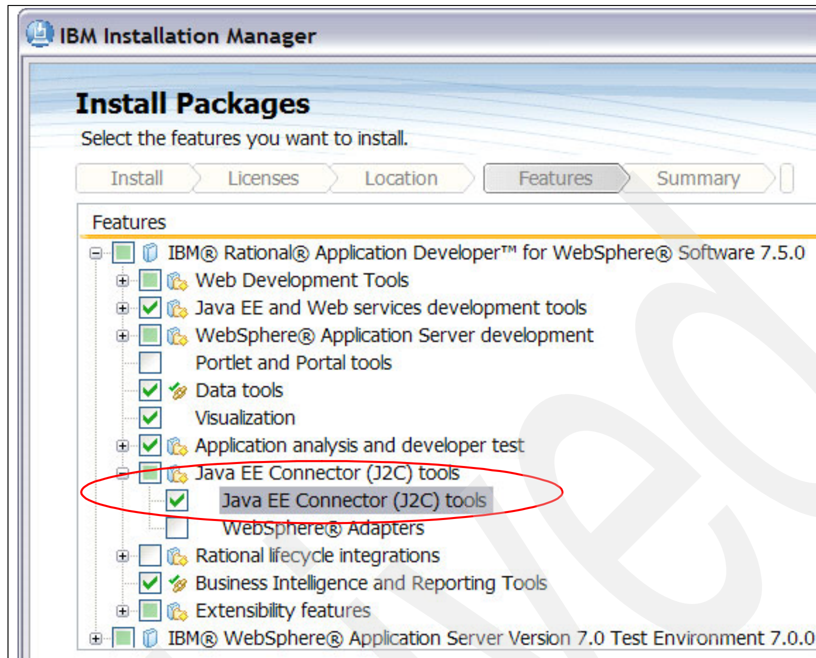


Figure 5-1 J2C feature selection during install

If RAD was previously installed on the workstation, insure that the J2C feature was also installed. To verify the installed features after starting RAD, click **Help** → **About Rational Application Developer for WebSphere** → **Plug-in Details**.

You will see a list of installed plug-ins for J2C tools, as shown in Figure 5-2 on page 108. If they are not present, install the J2C feature using the IBM Installation Manager.

Archived

S	Provider	Plug-in Name
	IBM	J2C Code Generation Plugin
	IBM	J2C Common Deployment Plug-in
	IBM	J2C Java Bean Cheat Sheet Actions ...
	IBM	J2C Java Bean Cheat Sheet Conten...
	Internati...	J2C Java Bean Samples
	IBM	J2C JSF Deployment Plug-in
	IBM	J2C Metadata
	IBM	J2C Migration Plug-in
	IBM	J2C plugin for RAR operations
	IBM	J2C Product Plug-in
	IBM	J2C Record Writer for PLI Plug-in
	IBM	J2C Record Writer for PLI Plug-in
	IBM	J2C Record Writer Plug-in

Figure 5-2 Plug-ins for J2C Tools

5.3 Sample application

The J2C application development process is demonstrated by our sample application. The components of our sample application are shown in Figure 5-3.

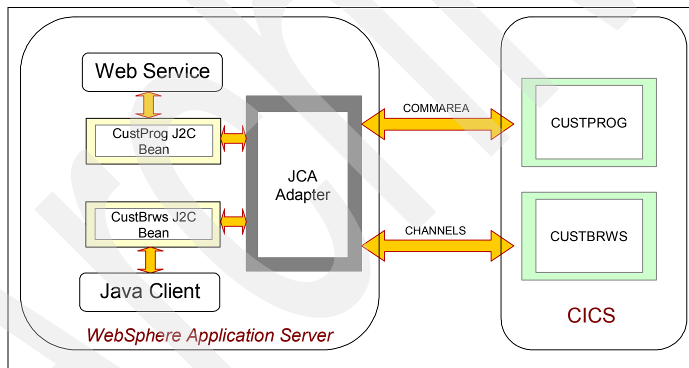


Figure 5-3 Sample Application Block Diagram

Two COBOL programs named CUSTPROG and CUSTBRWS are running in the CICS back-end. Both CICS programs perform operations on a VSAM file. These two programs are described in Chapter 4, “Java-based Sample Application” on page 65. The CUSTPROG program uses the COMMAREA for input and output whereas the CUSTBRWS program uses channels for the information exchange with the client.

Archived

Two J2C beans namely, CustProg and CustBrws are used to access the functions within the CUSTPROG and CUSTBRWS programs. The J2C beans execute in the WebSphere Application Server Java environment, which does not have direct access to programs running in CICS. The JCA adapter provided by the CICS Transaction Gateway (CTG) acts as the bridge between the Java beans and back-end CICS programs.

The complete sample application was built in four steps using Rational Application Developer:

1. Generate data binding classes corresponding to the COBOL data structures
2. Generate a J2C bean to access CICS
3. Add a J2C bean method to invoke the functionality provided by the CUSTPROG back-end programs.
4. (Optional) Create a Web service to expose this functionality.

At the completion of Step 3, we have created J2C beans, which can be used in other Java application components which run on the server. The sequence of steps required is depicted in Figure 5-4.

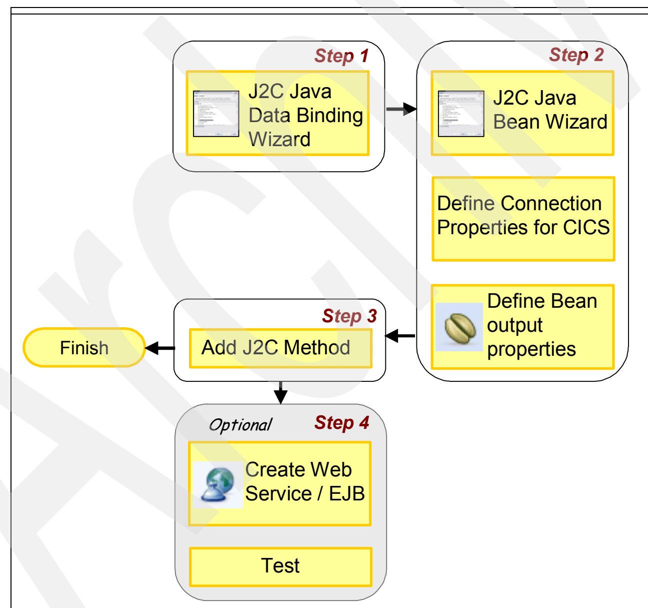


Figure 5-4 Sequence of operations

We cover each of these steps in the following sections. Before proceeding, a Java project named itso needs to be created in the RAD workspace.

5.3.1 Generate data binding classes

Data binding classes are Java classes encapsulating COBOL, PL/I, or C language structures. They can be quickly and easily generated in RAD using a simple importer wizard, and used as input or output types for interactions with CICS either as standalone classes or together with the generated J2C beans. The CUSTPROG program running in our CICS system is a CICS COBOL-based application that uses a COBOL record structure to describe the layout of the communications area. RAD provides options to both generate binding classes and marshal the input and output data dependant upon the target platform. In our sample, the requirement is on Java to COBOL. We performed the following steps to generate data binding classes:

Note: New in RAD 7.5 is support for CICS channel and container-based applications. This support is discussed further in 5.4, "Developing with channels and containers" on page 140

1. Click **File** → **New** → **Other** → **J2C** → **CICS/IMS Java Data Binding**, as shown in Figure 5-5. This launches the CICS Java Data Binding Wizard.

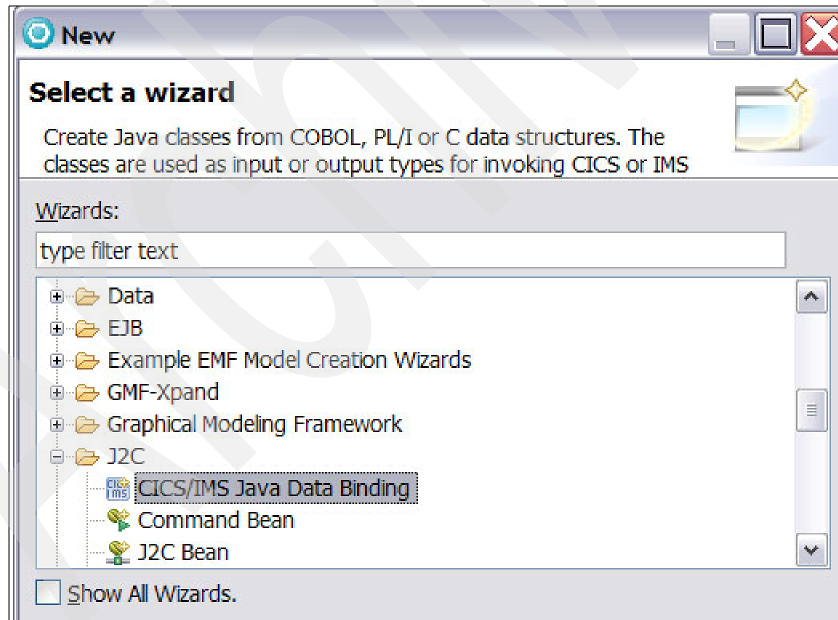


Figure 5-5 J2C CICS Java Data Binding Wizard Launch

2. Click **Next** to open the Data Import window (Figure 5-6).

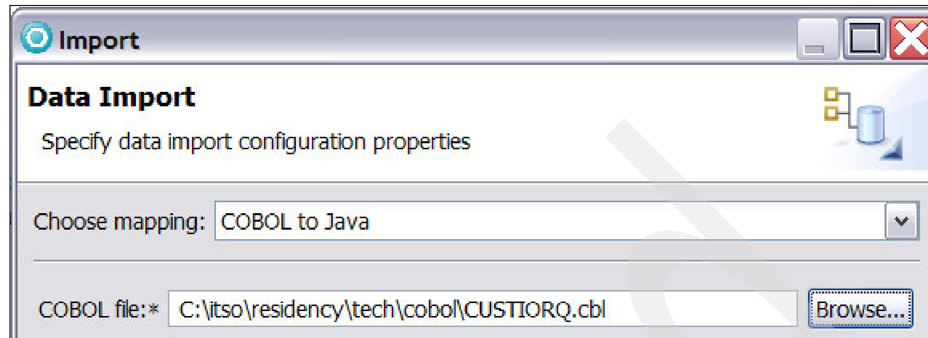


Figure 5-6 J2C Binding Wizard Data Import

The RAD tool supports the mapping of various language structures. We require the conversion of COBOL data structures to their Java equivalent. Therefore, in the "Choose mapping" drop-down list, select "COBOL to Java".

To create the Java equivalent of a COBOL data structure, the wizard needs to know how the COBOL data structure is defined. Typically, this structure is available within the COBOL program file or in a separate COBOL copy book. In the "COBOL file" text box, specify the COBOL program or COBOL copy book where these definitions are available.

By default, RAD assumes any file with the .cbl extension is a complete COBOL program, and any file with the .cpy extension contains only the data structure definitions.

3. These assumptions can be altered by changing the preferences for the COBOL file importer by clicking **Window** → **Preferences**.

Archived

4. In the Preferences panel (Figure 5-7), expand **Importer**, and select **COBOL**. The General tab provides options to select the target platform, endianess, and so forth. In this example we select the target platform as z/OS.

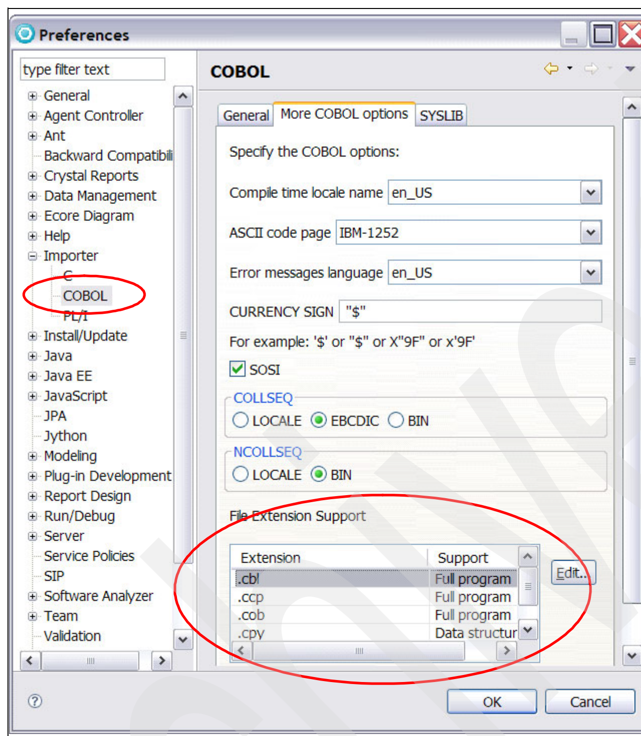


Figure 5-7 Changing COBOL import preferences

Specify all the system specific parameters in the Preferences window before proceeding to the next step. The RAD tool expects a syntactically complete and correct COBOL structure. If the structure contains compiler-specific key words or missing sections may cause errors when attempting to parse the structure

5. Click **Next** to choose the appropriate data structure, as shown in Figure 5-8 on page 113.

   The default selections presented for Platform and Code Page are based on the choices made in the previous step. Change these selections if necessary.

   > **Note:** When the RAD generated application code is executing, all data sent to CICS will be converted from the selected code page into a Java String in the Unicode format. For more information about different options for handling data conversion, refer to Appendix A, “Data conversion techniques” on page 181.

   In this example, the COBOL file CUSTIORQ.cbl we imported contained only one data structure identified by the CUSTPROG-COMMAREA (Figure 5-8). This structure needs to be converted to its Java equivalent. Select this and proceed to the next step.

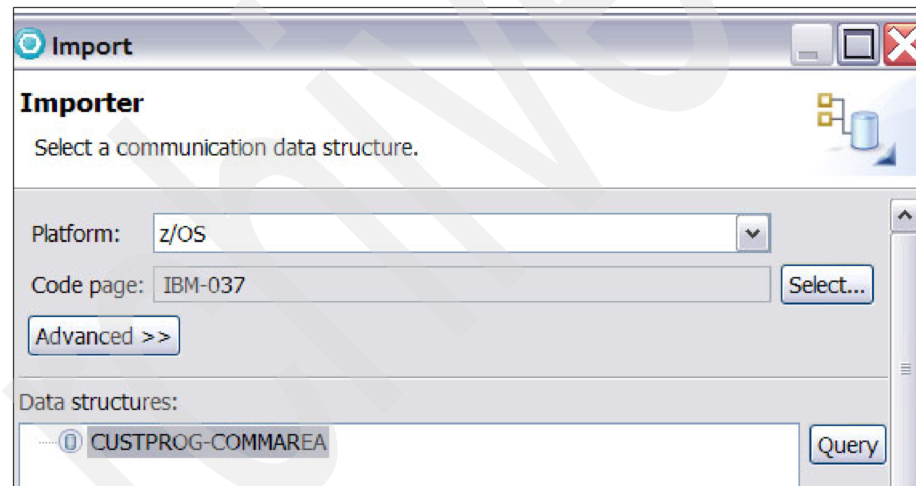


Figure 5-8 Selecting the COBOL data structure to be bound to Java

Archived

6. Click **Next** to modify the Saving Properties (Figure 5-9).

   By default, the binding Java class will be given a name that resembles the data structure identifier. In this example, the default Java class name given by the wizard is CUSTPROGCOMMAREA. We modified it to CustProgCommArea to adhere to the Java conventions. We add this binding class to the com.ibm.itso package, which belongs to the project itso.

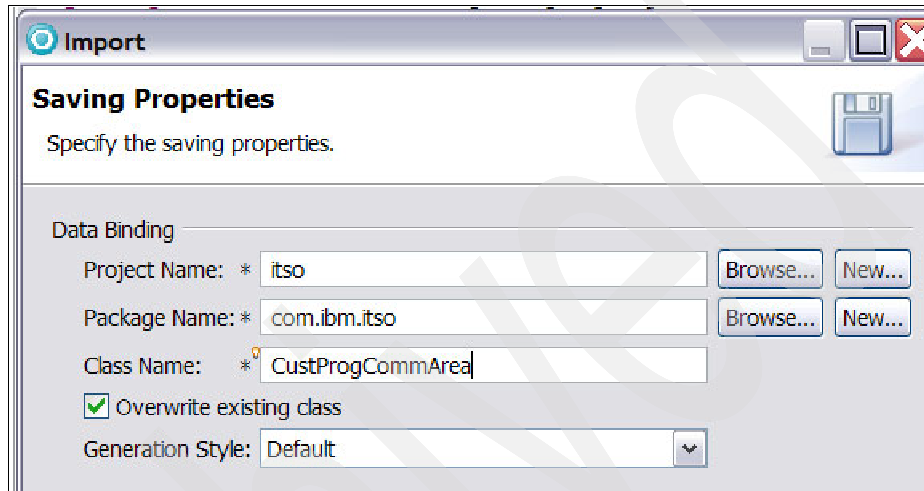


Figure 5-9 Saving Properties options

7. Click **Finish.** The wizard generates the Java classes. View this binding class in the package explorer view of RAD (Figure 5-10).

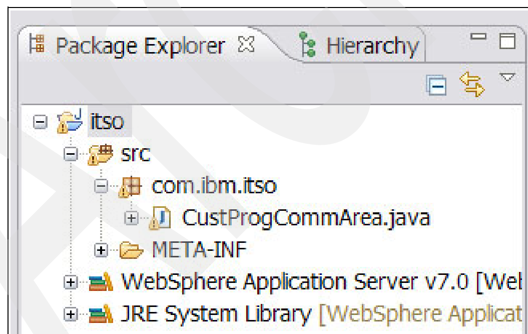


Figure 5-10 Wizard-generated Java binding class

The Java class CustProgCommArea represents the input COMMAREA structure, which the server side CICS COBOL program CUSTPROG expects. This class is available in the project workspace. This Java bean acts as a carrier of COMMAREA whenever the Java program exchanges information with the CICS COBOL program CUSTPROG.

Note: The contents of Java class CustProgCommArea is different from the Java class with the same name in Chapter 4, “Java-based Sample Application” on page 65, though the purpose of both the classes are same. All Java classes referred to in this chapter contain code generated by Rational Application Developer wizards. The Java classes discussed in Chapter 4, “Java-based Sample Application” on page 65 are user written code.

5.3.2 Generate a J2C bean to access CICS

With the binding classes at our disposal, we can proceed to create a J2C bean that will connect to the back-end CICS. To transfer data to the back-end CICS COBOL program, this J2C bean will use the binding class we created in the previous section. We will use the J2C bean wizard to create an empty J2C bean by performing the following steps:

1. Click **File** → **New** → **Other** → **J2C** → **J2C Bean**.

   This will bring up the J2C bean wizard shown in Figure 5-11.

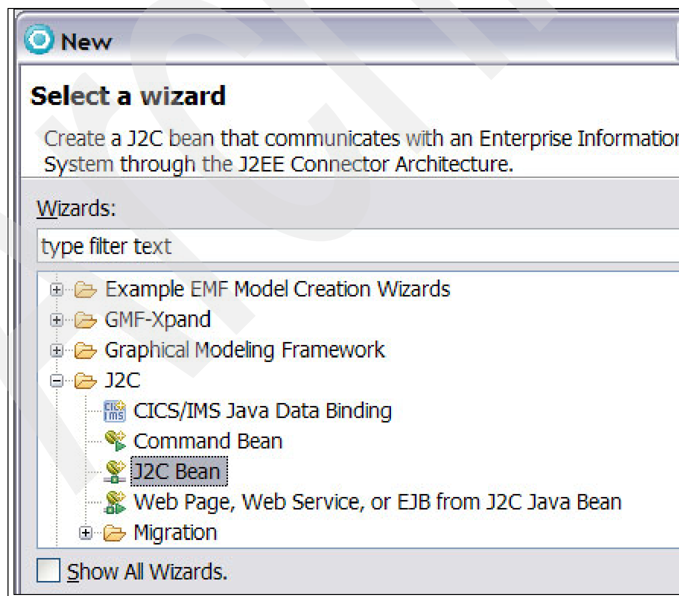


Figure 5-11 Launching the J2C bean wizard

A resource adapter is a system-level software driver that a Java application uses to connect to an enterprise information system (EIS), which in our case is CICS. The resource adapter plugs into an application server and provides connectivity between CICS, the application server, and the enterprise application. At this stage we need to inform the wizard about the adapter we are going to use to connect to the back-end CICS system. The selection of the adapter is significant during both the development stage and the deployment process. During the deployment process the same or a compatible adapter as used during development should be chosen.

RAD V7.5 bundles four different versions of CICS resource adapters. These adapters are provided by the CICS Transaction Gateway (CTG) product. For this example we will use the ECIResourceAdapter 7.1.0.2 for CICS, as shown in Figure 5-12. This CICS ECI resource adapter is based on Version 1.5 of the J2EE Connector Architecture (JCA 1.5) specification. This version of the CICS ECI resource adapter for Java runs with WebSphere Application Server Version 6.1 and later.

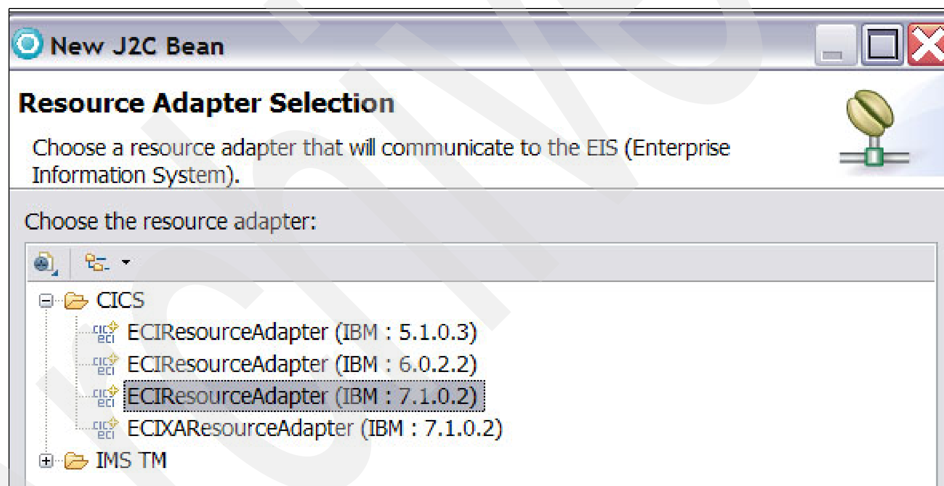


Figure 5-12 Resource Adapter Selection window

2. Select the **ECIResourceAdapter (IBM: 7.1.0.2)** resource adapter and click **Next**.
3. Import the resource adapter into the workspace. Resource adapter projects are stored in Resource Adapter Archive (RAR) files. If the resource adapter selected in the previous step is not available in the project's workspace, it will be imported into the current workspace at this stage. cicseci7102.rar is the resource adapter RAR file. This is imported into our current workspace from the file system. See Figure 5-13 on page 117.

Archived

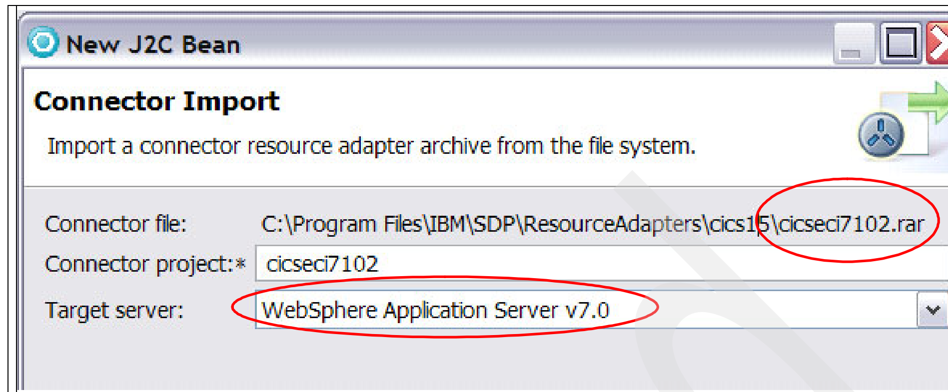


Figure 5-13 Connector Import window

4. In the Target Server drop-down list, select the application server to target for development. This selection affects the run-time settings by modifying the class path entries for the project. We selected WebSphere Application Server 7.0 as the target because we intend to deploy the final application on this version of WebSphere Application Server. See Figure 5-13.
5. Clicking **Next** imports the adapter into the current workspace and brings up the Connection Properties section of the wizard.
6. If the adapter was imported earlier into project's workspace and is available, it will be listed by the wizard in the resource adapter selection (Figure 5-14). Select this and click **Next**. In this case the wizard will skip the *connector import* section.

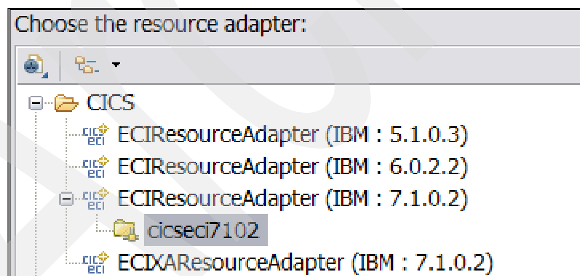


Figure 5-14 Resource adapters available in the workspace

7. Set the connection properties. When an application component like an EJB wants to interchange information with back-end CICS, it does not interact directly with resource adapters. Instead, it interacts through Connection objects. Connections from the Java bean to the CICS can be either managed or non-managed. Managed connections are obtained through the JNDI and

are managed by the application server. Non-managed connections are obtained directly through the resource adapter for two-tiered applications. If both connection types are selected, a managed connection is attempted first, followed by a non-managed connection.

This part of the wizard allows you to specify the connection type. If **Managed Connection** is selected, the corresponding JNDI lookup name must also be specified. The application uses this JNDI name to search and create the connection using connection factory. A managed connection is the recommended connection type because the management of connections, transactions, and security can be delegated to the application server, exploiting the capabilities of application server. Connections created by the application server share in a common pool of connections, which permits the re-use of the connection factories by other applications.

The RAD J2C tool provides a convenient way to define a new managed connection factory on the server or to select an existing managed connection from a WebSphere server instance.

In this example we chose the **Managed Connection** option. We will create a new managed connection and map it to the JNDI name ItsoCICSConnection.

8. Enter the JNDI name ItsoCICSConnection in the JNDI Name text box, as shown in Figure 5-15 and click **New.**

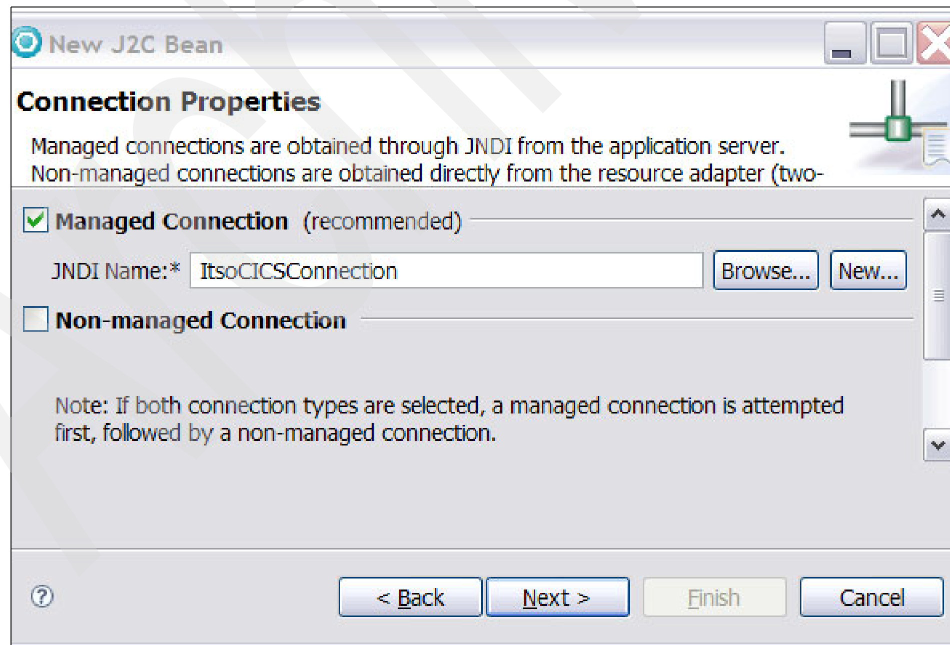


Figure 5-15 Selecting the connection type

This pops up a new wizard window as shown in Figure 5-16.

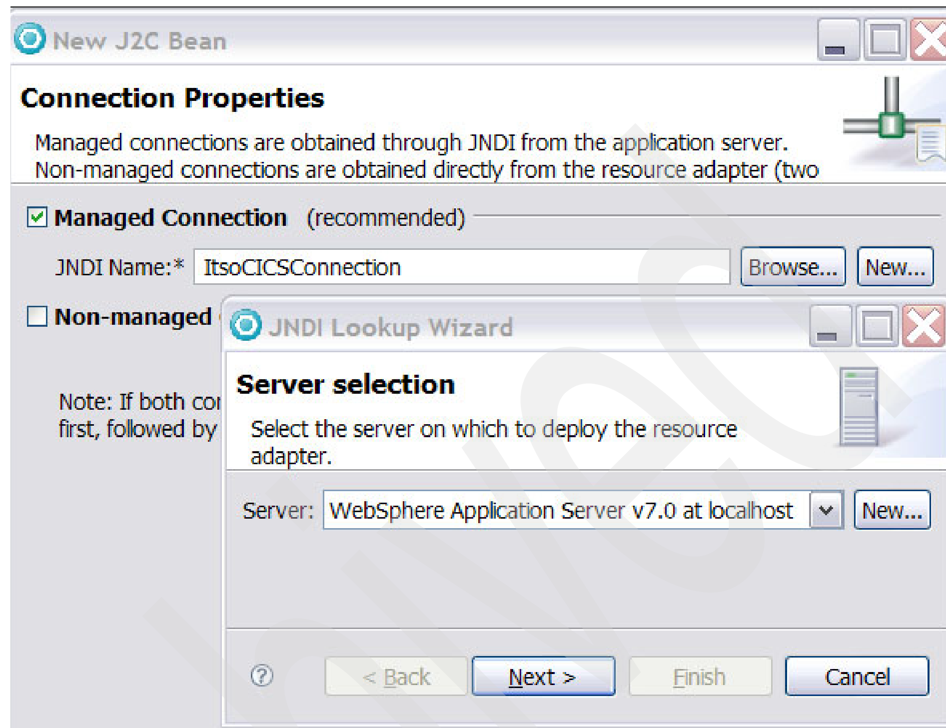


Figure 5-16 Selecting WebSphere Application Server instance

When we selected a managed connection, we presumed that the adapter was installed in the target application server in a stand-alone mode. This implies that the adapter is visible to all the Enterprise Application Archives (EARs) installed on the application server. If the resource adapter is not installed on the server instance specified, the wizard will install a resource adapter on the server in the process of creating the managed connection factory. Therefore, we inform the wizard about the WebSphere Application Server instance on which we expect the resource adapter to be installed and the connection factory that is being created.

Our target server instance is the instance created by the test environment of our WebSphere Application Server v7.0, which is embedded in RAD v7.5. Because it is embedded within the RAD runtime environment, the application server will run as a localhost. If the target server instance is not a localhost, select the appropriate instance from the drop-down list. If the target server instance is not available in the drop-down list, add it to the list of servers registered with RAD by clicking **New**.

Once the server instance is selected, specify the CICS-specific properties that are required for creating a successful connection to CICS.

9. Click **Next** in the pop-up window. This brings up the connection factory settings window of the J2C wizard.

   Managed connections are maintained by the application server, in this case the WebSphere Application Server v7.0. Typically the application server maintains a pool of connections and the maintenance of this connection pool will be delegated to ConnectionFactory, which acts as a connection pool manager running on the application server. Application components, like an EJB, should not create the connection objects directly. Instead, it should request the ConnectionFactory for a connection. The ConnectionFactory responds to this request in the most efficient manner, either by creating a new connection or obtaining an existing connection from the pool.

   The connection properties for creating the connection factory are specific to CICS. In this example the connection to our back-end CICS system is achieved through the CICS Transaction Gateway (CTG).

10. Enter the properties listed in Table 5-1 to create the connection factory.

Table 5-1 Important Connection Factory Properties

Field	Value
Connection URL	The connection URL of the CICS TG in the form protocol://hostname.
Server Name	Name of the CICS server as known to the CICS TG.
Port Number	Port on which the Gateway daemon listens for client requests.
User Name	If security is enabled, enter the user ID that you use to connect to CICS.
Password	If security is enabled, enter the password for the defined user ID.

Archived

The value of other parameters depends on the CICS environment settings for the application. After the entries are made, the J2C Connection Factory properties should appear as shown in Figure 5-17.

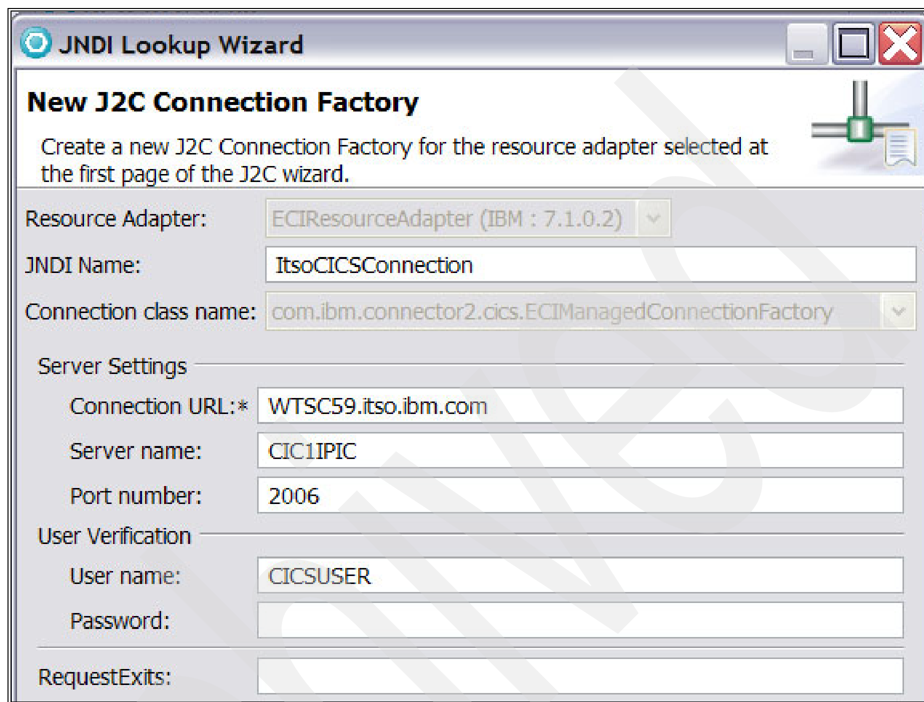


Figure 5-17 Properties for Connection Factory

11.Click **Finish**. The pop-up window will close.

Archived

12. On the Connection Properties window, click **Next.** This brings us to the J2C Java beans output properties panel shown in Figure 5-18.

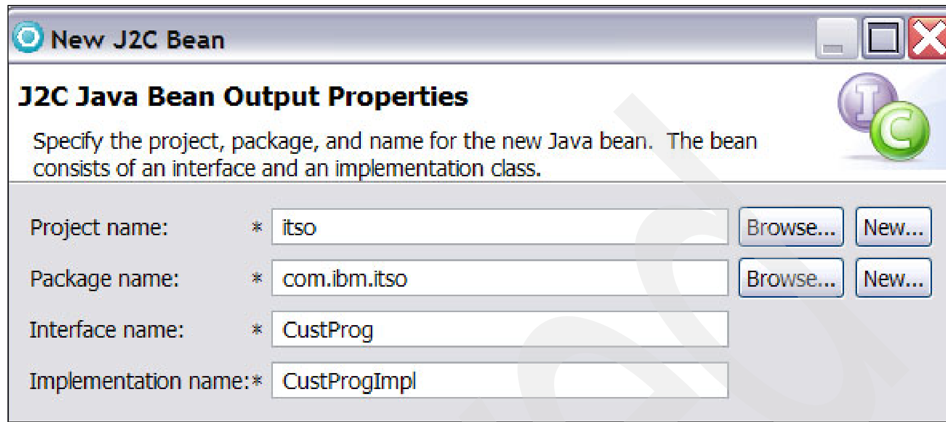


Figure 5-18 J2C Bean Properties

Defining J2C bean output properties.

Define the output properties for the J2C Java bean. To do this, perform the following steps:

1. In the Project name field, ensure that the correct name of your project appears. This is the project to which the J2C bean will be associated. If the name does not appear by default, use the **Browse** button to select the project from the available list of projects in the current RAD workspace. Otherwise, a new project can be created by clicking **New**. Here, we associate the J2C bean with *itso* project.
2. In the Package name field, enter the name of the J2C bean package. The bean wizard will create the Java classes under this package. Add this to package com.ibm.itso.
3. In the Interface name field, enter the name of the J2C bean interface class. The application components access the methods of the J2C bean through this interface. The purpose of our J2C bean is to access functionalities of the CICS back-end program CUSTPROG. Therefore, we named the interface CustProg.
4. In the Implementation name field, enter the name of the implementation class for this J2C bean. By default, the name appears as < interface name>Impl. It will appear as CustProgImpl. This class will contain the implementation of the interface specified in the Interface Name field.

   Once the entries are completed, the J2C Bean Properties will appear as shown in Figure 5-18.

Archived

5. Click **Finish** to create a simple J2C Java bean to which we will later add the data binding classes.

   The J2C bean wizard now creates the specified managed connection factory on the server instance. If the selected WebSphere v7.0 instance is running, the managed connection factory will be created immediately. If the server is not yet started, the creation of the managed connection factory will be postponed until the server is started, in which case the server republishes the configuration information and the user need not re-start the server instance.

   After the creation of the connection factory on the target server, the J2C wizard generates code that is capable of performing a connection to the back-end CICS system. If the bean creation is successful, the Java interface CustProg and its implementation CustProgImpl will be visible under the project for the package specified. The RAD package view will show the content as in Figure 5-19.

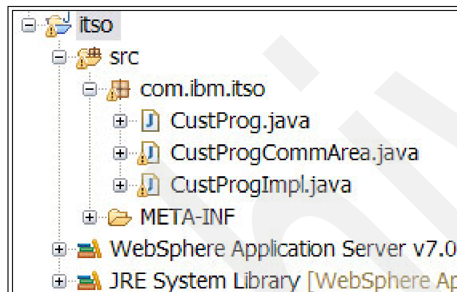


Figure 5-19 Java Bean Interface and Implementation generated by the wizard

   A quick look at the generated code shows the presence of few doclet tags in the code. Even after the J2C bean code has been generated, we can still modify the generated code as needed by changing the values of the doclet tags. As an example, if you want to change the JNDI name, change the value of the jndi-name attribute of the @j2c.connectionFactory doclet tag as shown in Figure 5-20 on page 124. The body of the J2C bean is generated by RAD when the changes are saved.

```
package com.ibm.itso;

import javax.resource.ResourceException;
import javax.resource.cci.Connection;
import javax.resource.cci.Interaction;
import javax.resource.cci.ConnectionFactory;
import javax.resource.cci.ConnectionSpec;
import javax.resource.cci.InteractionSpec;
import javax.resource.cci.Record;
import javax.resource.cci.ResourceAdapterMetaData;


/**
 * @j2c.connectionFactory jndi-name="ItsoCICSConnection"
 * @j2c.connectionSpec class="com.ibm.connector2.cics.ECIConnectionSpec"
 * @generated
 */
public class CustProgImpl implements com.ibm.itso.CustProg {

    private ConnectionSpec typeLevelConnectionSpec;
    private InteractionSpec invokedInteractionSpec;
    private InteractionSpec interactionSpec;
    private ConnectionSpec connectionSpec;
    private Connection connection;
    private ConnectionFactory connectionFactory;
```

Figure 5-20 Doclet tags in generated code

6. Verify the installation of the resource adaptor and the connection factory by using the WebSphere Application Server administration console. To do this, perform the following steps
   a. Start the WebSphere Application Server environment from within RAD and start the WebSphere Application Server administration console.
   b. On the leftside navigation panel, expand **Resource Adapters**. The entries should appear as shown in Figure 5-21 on page 125.

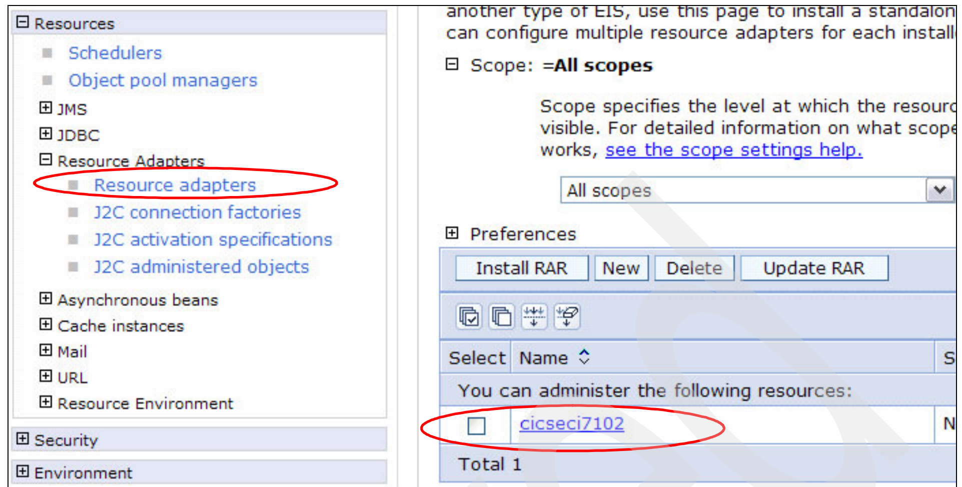


Figure 5-21 Resource Adapter deployed in WebSphere Application Server

c. Click **J2C Connection Factories**. The entries should appear as shown in Figure 5-22

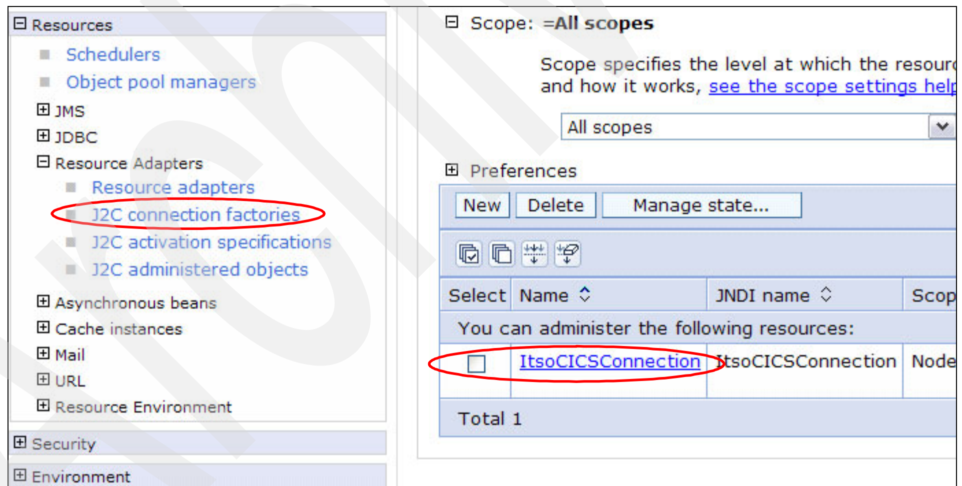


Figure 5-22 Connection Factory created in WebSphere Application Server

We have now created two main artifacts:

- A J2C bean that knows how to connect to the back-end CICS system.
- The Data Binding classes that represent the data required to pass to the CUSTPROG CICS application.

These two artifacts exists as two separate and independent pieces. We do not yet have a method which uses the binding and the connection mechanism to interact with the CUSTPROG CICS application. Our next task will be to add a new method to the J2C bean which will be responsible for invoking the CustProg operation on the back-end CICS system.

5.3.3 Adding a J2C bean method

Perform the following steps to add a J2C bean method:

1. On the Package Explorer view of the RAD workspace, scroll to the CustProgImpl J2C bean implementation class. Right-click and a menu will display. From the menu, click **Source** → **Add/Edit J2C bean method**.

   The “New Java Method” wizard window displays (Figure 5-23). Click **Add**.

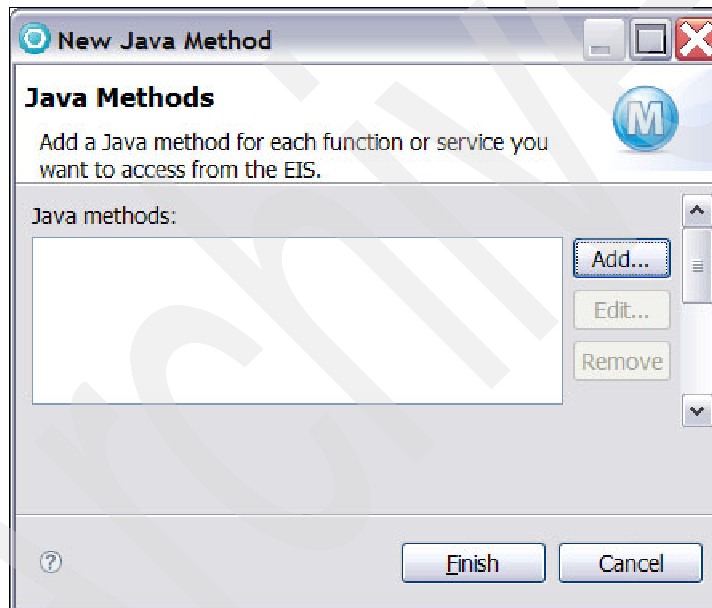


Figure 5-23 Add a Java method

2. The Add Java Method window displays. In the Name field, enter the name of the Java method to be added to the J2C bean. We chose to call the method invokeCustProg.

Archived

3. Specify the input type for the method. These are the data binding types that you created in earlier steps. The Input type field cannot be edited directly. Use **Browse** to select these classes. If the binding classes were not created earlier, invoke the binding classes wizard from this window by clicking **New**.

   In this case, we have created the binding types already and must use the **Browse** button to select the type.

4. For our application, we pass the COMMAREA structure and receive the same COMMAREA structure in the response from the back-end CICS program. This means the return type of this method is the same as the input type. Select the **Use the input type for output** check box.

   Once the selection is complete the Java Method panel (Figure 5-24) displays.

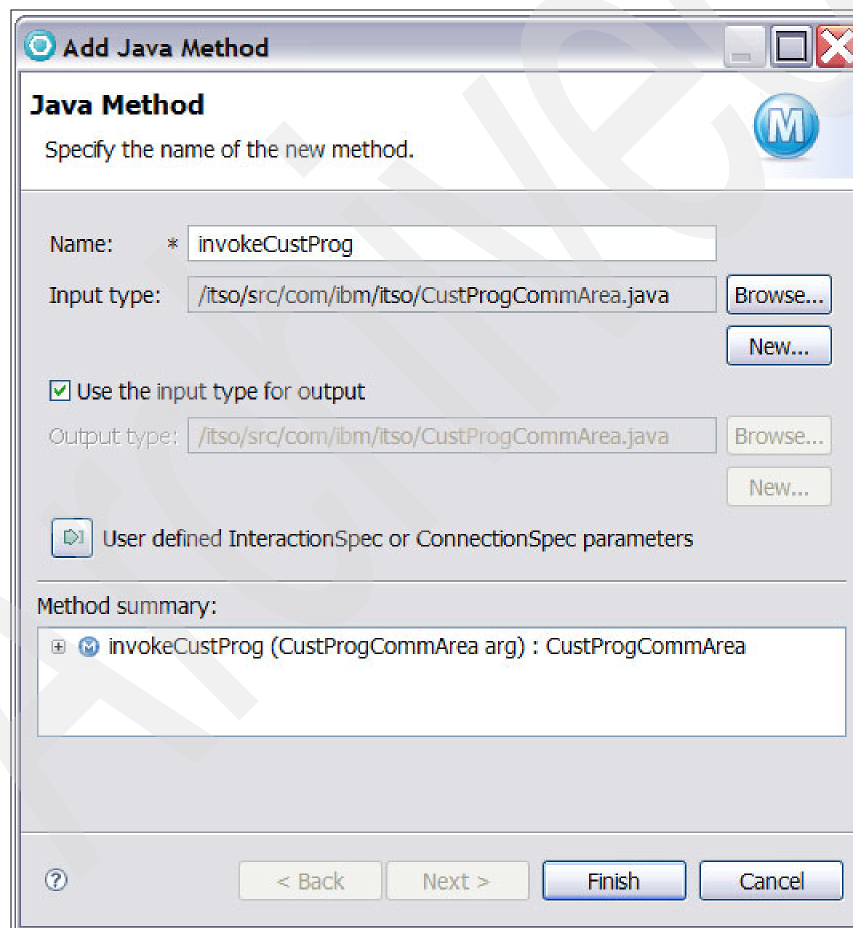


Figure 5-24 Values for the Java method

5. Click **User defined InteractionSpec or ConnectionSpec parameters** (Figure 5-24 on page 127). A window displays, where we specify the interaction and connection specification properties associated with the execution of the back-end CICS function that we want to expose. These become the input arguments to the business methods in the J2C bean. We selected the interaction specification properties shown in Figure 5-25. Notice that when we select a property, the corresponding method signature on the invokeCustProg changes.

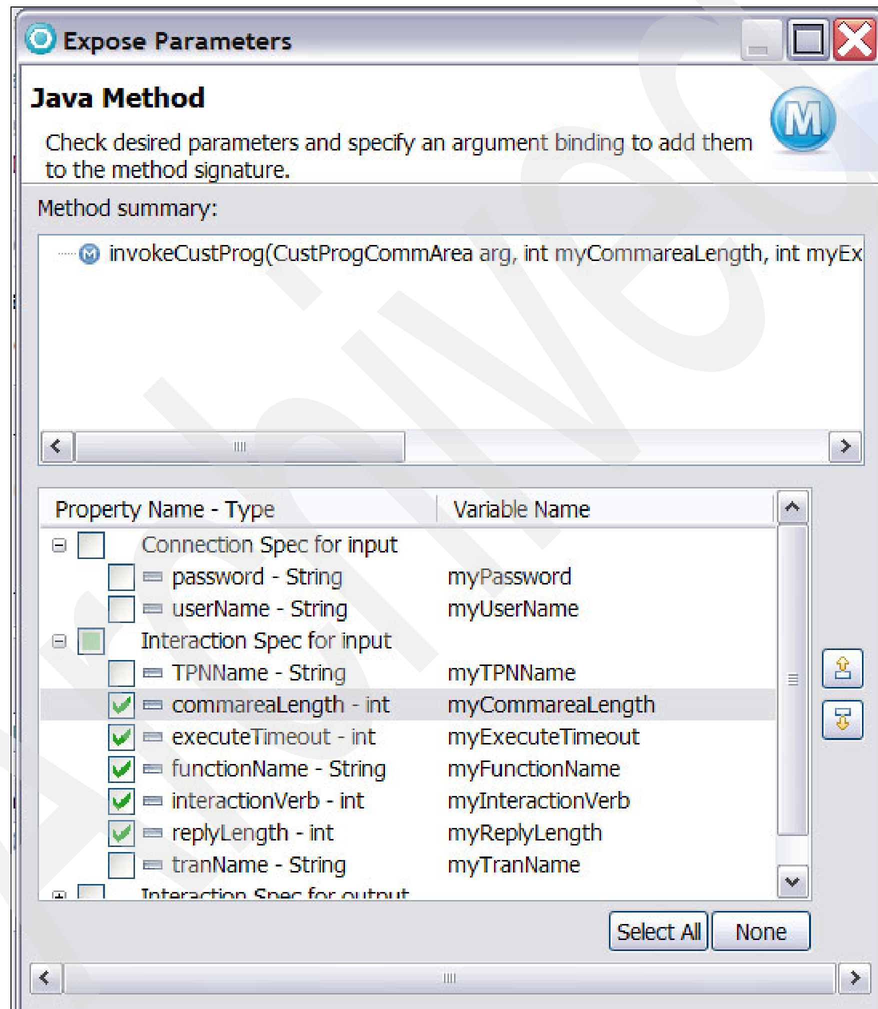


Figure 5-25 Parameters for InteractionSpec and ConnectionSpec

6. Click **Finish** on the “Expose Parameters” panel and click **Finish** on the “Add Java Method” window.

Archived

7. We are now back to the “New Java Method” window. Add as many methods as you have services on the back-end CICS program. We are only exposing one function, so we click **Finish**.

The wizard now generates the new method invokeCustProg within the CustProgImpl class. The generated code is shown in Figure 5-26. The method signature matches what was displayed in the New Method window in Figure 5-25 on page 128.

```
public com.ibm.itso.CustProgCommArea invokeCustProg(
        com.ibm.itso.CustProgCommArea arg,
        int myCommareaLength,
        int myExecuteTimeout,
        java.lang.String myFunctionName,
        int myInteractionVerb, int myReplyLength)
        throws javax.resource.ResourceException {
    ConnectionSpec cs = getConnectionSpec();
    InteractionSpec is = interactionSpec;
    try {
        if (cs == null) {
            cs = getTypeLevelConnectionSpec();
        }
        if (is == null) {
        is = new com.ibm.connector2.cics.ECIInteractionSpec();
        }
        ((com.ibm.connector2.cics.ECIInteractionSpec) is)
                .setCommareaLength(myCommareaLength);
        ((com.ibm.connector2.cics.ECIInteractionSpec) is)
                .setExecuteTimeout(myExecuteTimeout);
        ((com.ibm.connector2.cics.ECIInteractionSpec) is)
                .setFunctionName(myFunctionName);
        ((com.ibm.connector2.cics.ECIInteractionSpec) is)
                .setInteractionVerb(myInteractionVerb);
        ((com.ibm.connector2.cics.ECIInteractionSpec) is)
                .setReplyLength(myReplyLength);
    } catch (Exception e) {
        throw new ResourceException(e.getMessage());
    }
    com.ibm.itso.CustProgCommArea output =
        new com.ibm.itso.CustProgCommArea();
    invoke(cs, is, arg, output);
    return output;
}
```

Figure 5-26 Wizard generated method

We now have a functional J2C bean that is able to access the back-end CICS and invoke the functions provided by the CUSTPROG application. This J2C bean can be reused in other Java stand-alone applications and other Java projects.

Next, using the WebServices creation wizard, we will expose this functionality as a WebService.

5.3.4 Creating a WebService from the J2C bean

Perform the following steps to create a WebService from the J2C bean.

1. Invoke the WebServices creation wizard, by clicking **File** → **New** → **Other** → **J2C** → **Web Page,Web Service or EJB from J2C Bean** (Figure 5-27).

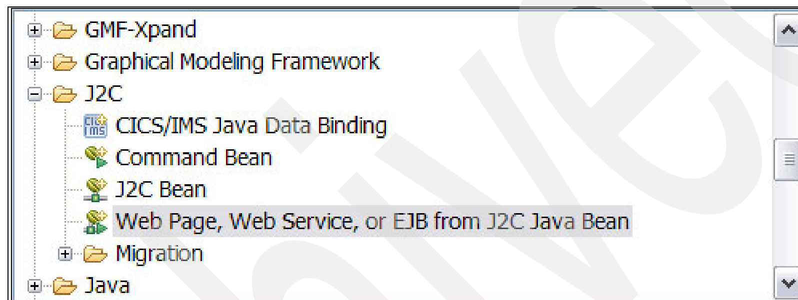


Figure 5-27 Invoking WebServices creation wizard for J2C Bean

Click **Next**. This brings up the J2C Java bean selection window (Figure 5-28).

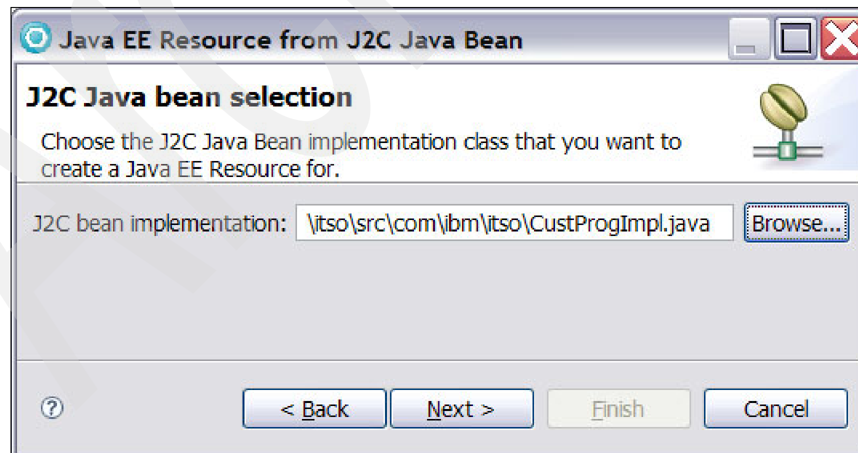


Figure 5-28 Selecting J2C bean for creating WebService

Archived

2. Use **Browse** to select the CustProgImpl J2C bean created previously. Click **Next**. The Java Resource Type window (Figure 5-29) displays.

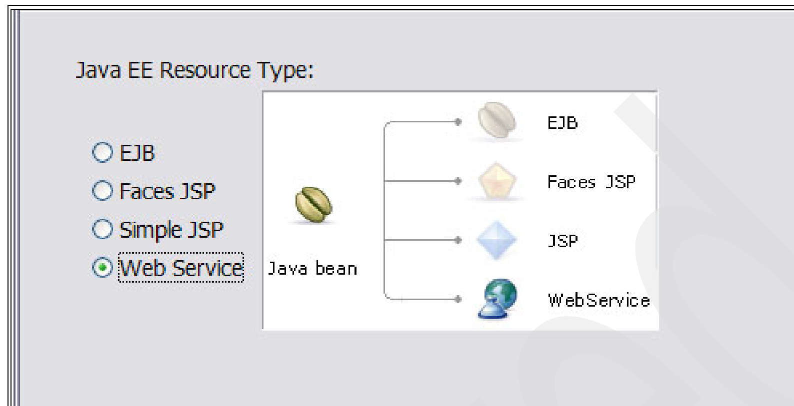


Figure 5-29 Selecting Web Service for J2C Bean

3. Select the **Web Service** radio button. Click **Next**. The Web Service Creation window displays.
4. Specify the dynamic web project for which this web service will be generated. Currently, we have not created any web project in the RAD workspace. Create a new one by clicking **New**. On the window (Figure 5-30 on page 132) that displays, enter the project name as itsoWeb. Click **Finish**.

Archived

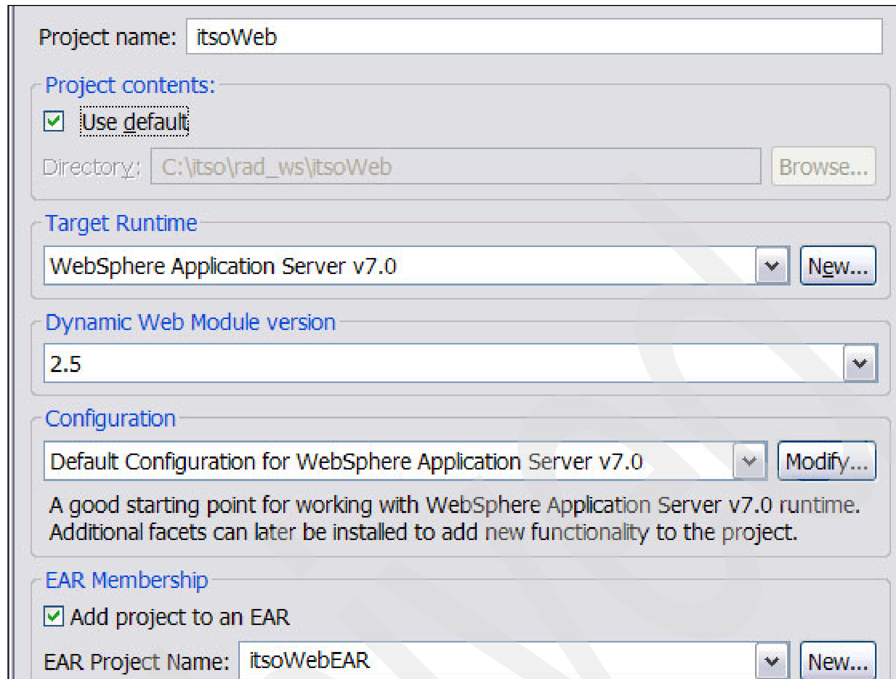


Figure 5-30 Creating a Web Project

The wizard now creates a new Web project and a new EAR project. The WebService Creation window displays.

5. Click **Browse**. Locate this new project as the Web Project to which this web service will be added.
6. Start the embedded WebSphere Application Server so it can look up JNDI names, if required.

   WebSphere Application Server requires your code to reference application server resources (such as data sources or J2C connection factories) through logical names, rather than access the resources directly in the Java Naming and Directory Interface (JNDI) name space. These logical names are referred to as resource references. By using the resource reference you avoid having to hard code the real JNDI name of your resource in your application code. The resource reference is defined in the deployment descriptor of the application and can be changed to point to a different runtime resource depending on the target application server.

   We specified the resource name for the web service as itsoWSRef.

   Once these entries have been made, the panel should appear as shown in Figure 5-31 on page 133.

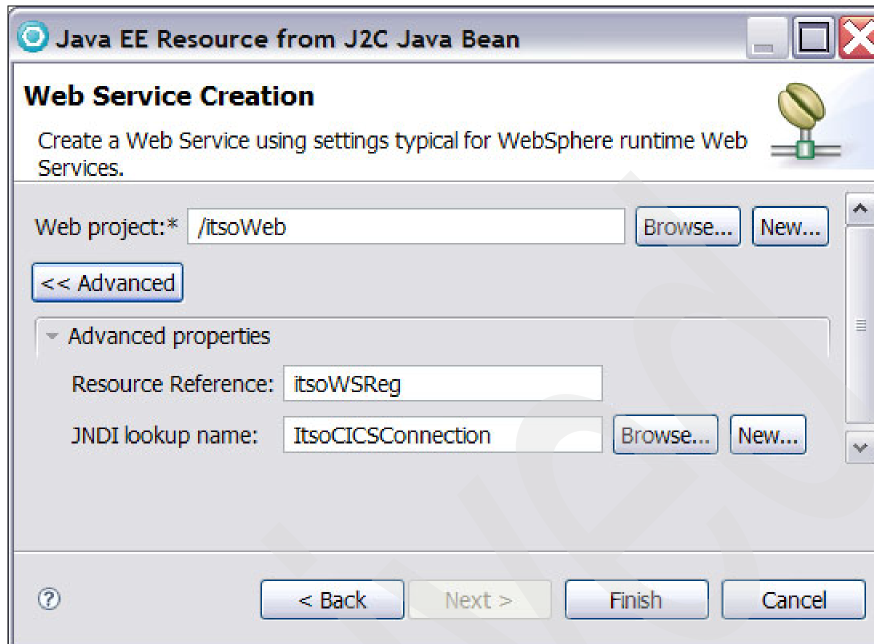


Figure 5-31 Web Service Creation

Archived

7. Click **Finish.** The wizard generates all the artifacts necessary for exposing the CUSTPROG functionality as a Web Service. These artifacts can be viewed in detail by using the Package Explorer view in RAD, as shown in Figure 5-32.

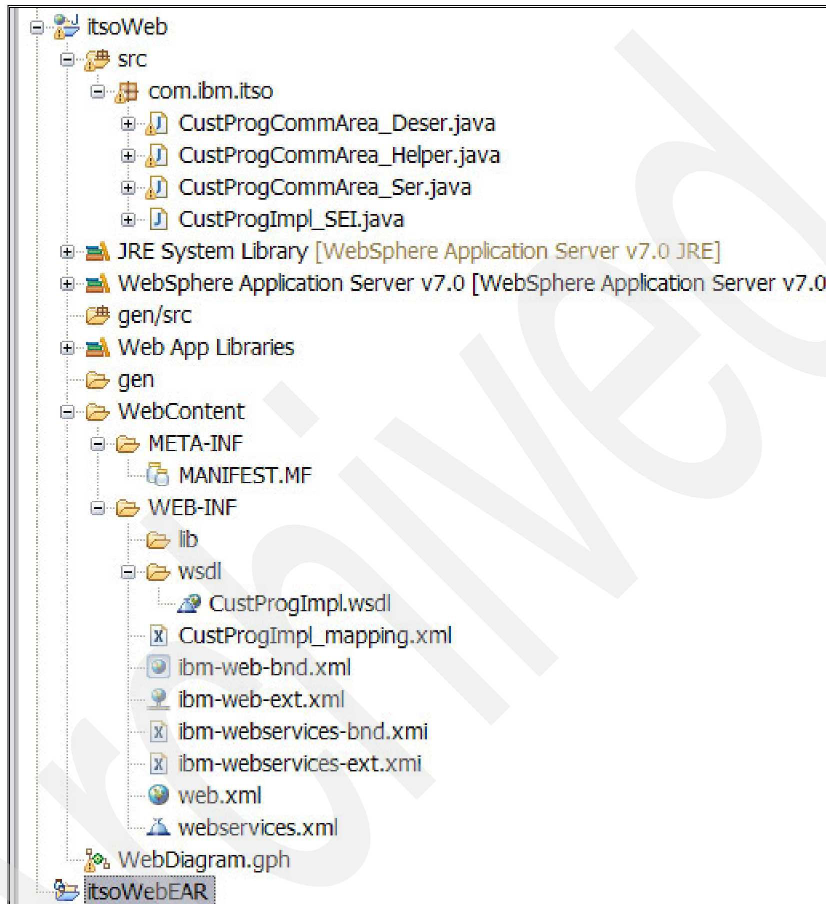


Figure 5-32 Artifacts Generated for our Web Service

We are now able to test the Web Service we created.

Archived

Testing the Web Service

The EAR project that contains the Web Service has to be added to the target WebSphere Application Server. Perform the following steps to test the Web Service.

1. Navigate to the servers view in RAD and right-click the target server to which the WebService is being deployed. Click **Add and Remove Projects** as shown in Figure 5-33. This brings up a panel where we can add the project to the target server.

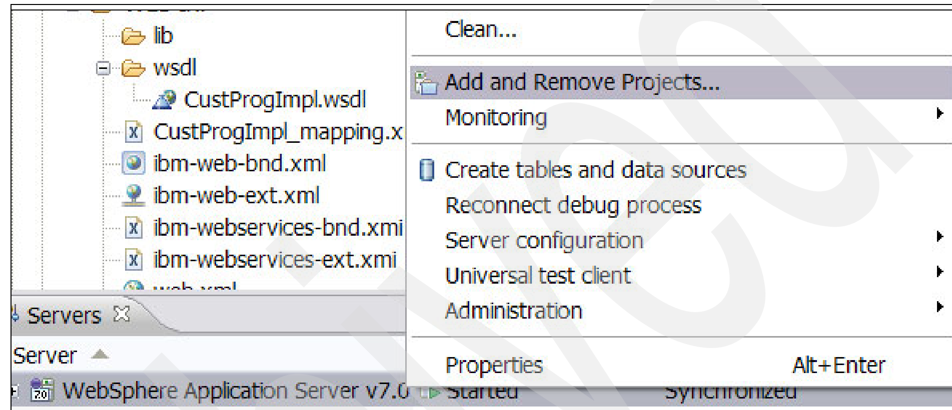


Figure 5-33 Adding the project to the server

Archived

2. Add the project itsoWebEAR to the embedded WebSphere Application Server as shown in Figure 5-34.

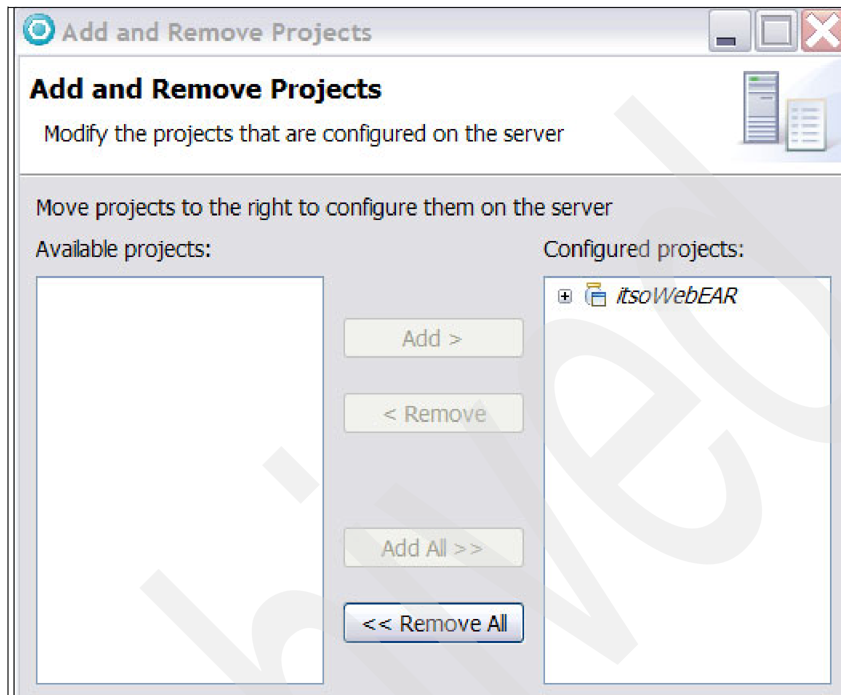


Figure 5-34 Adding project to Application Server

Click **Finish.** The Add Remove Projects deploys the application to the target server. If the WebSphere Application Server is active it publishes the application immediately without the server needing to be restarted.

The status of the new project can be verified using the WebSphere Administrative console on the target server as shown in Figure 5-35.

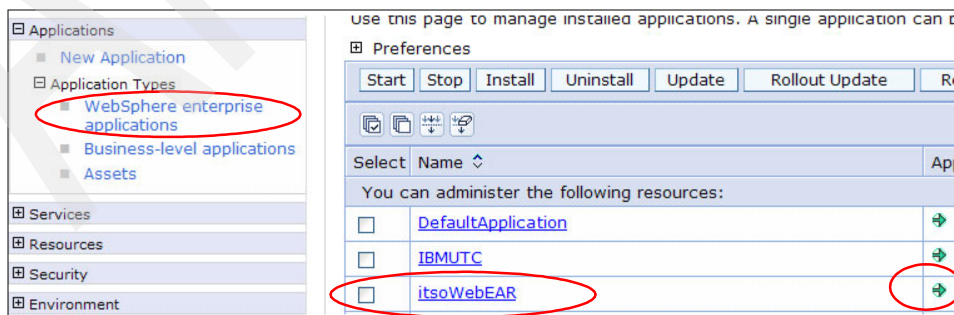


Figure 5-35 Project status on WebSphere Application Server

3. The Web Services Definitions are located in a file named CustProgImpl.wsdl. In the package explorer view, locate this file and right-click **Web Services** → **Test with Web Service Explorer**, as shown in Figure 5-36.

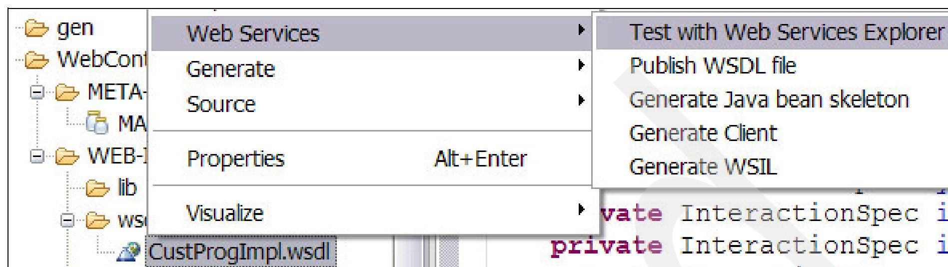


Figure 5-36 Select Web Services Explorer

The Web Services explorer starts. All the Operations defined within the CustProgImpl.wsdl file are listed on the right pane. In our example we defined only one operation named invokeCustProg, which is shown in Figure 5-37.

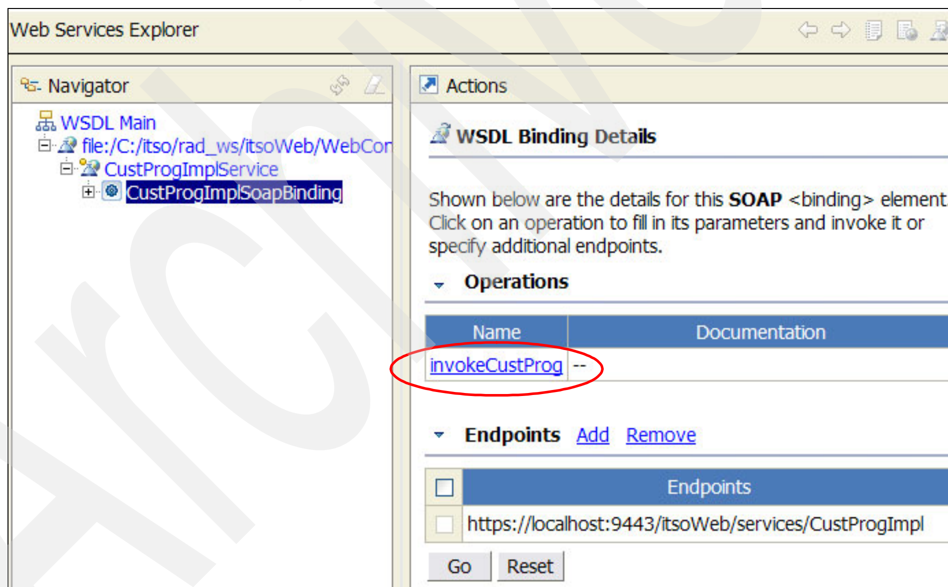


Figure 5-37 Web Service Explorer view of defined operations

Archived

4. Click the invokeCustProg link to invoke this operation. On the resultant window, enter the parameter values. We chose to pass commArea Length, execution timeout, function name, interaction verb, and reply length as parameters. Therefore, we need to enter all these values, as shown in Figure 5-38.

myCommareaLength int

309

myExecuteTimeout int

5000

myFunctionName string ☐ nil?

CUSTPROG

myInteractionVerb int

1

myReplyLength int

309

Figure 5-38 Parameter values for testing

For simplicity, these values are entered manually. In a complex application these values would be passed dynamically as parameters to the J2C bean. The value of commAreaLength is calculated based on the COBOL data structure we are passing to the back-end CICS program.

For the remaining data fields we need to enter only the request type and customer ID. We will set the request type to 'R', and the customer ID to '00000001', indicating that we want to read the details of the customer known by this ID. After the entries are made, the window will appear as shown in Figure 5-39 on page 139.

arg nil?

recordName string nil?

recordShortDescription string nil?

bytes base64Binary

request__type string nil?

R

ret__code string nil?

customerId string nil?

00000001

Figure 5-39 Entering the parameters for reading the customer details

Archived

5. Click **Go**. The Web Service will be invoked and the result will be displayed in the console window, as shown in Figure 5-40.

Body

invokeCustProgResponse

invokeCustProgReturn

recordName (string): com.ibm.itso.CustProgCommArea

recordShortDescription (string): com.ibm.itso.CustProgCommArea

bytes (base64Binary): UjAwMDAwMDAwMDFKZXRzb24gICAgICAgICAgICAgIEdlb3JnZS

request__type (string): R

ret__code (string): 00

customerId (string): 00000001

customerLastName (string): Jetson

customerFirstName (string): George

customerCompany (string): IBM

customerAddr1 (string): 1 Hursley Park

customerAddr2 (string):

customerCity (string): Hursley

customerState (string): Hants

customerCountry (string): England

customerMailCode (string): 123-123-123

customerPhone (string): 0-11-44-962-123456

customerLastUpdateDate (string): 11/17/08

return__comment (string): Customer information retrieved

Figure 5-40 Response from back-end CICS Program

5.4 Developing with channels and containers

So far we have demonstrated a CICS program that uses a COMMAREA as the mechanism for interchanging information between the client and the CICS server application. Now we will demonstrate how to interchange information between the client and the CICS server application using channels and containers as the communication medium. A complete description of channels and containers is presented in Chapter 3, “Channels and Containers” on page 49.

Archived

Application development using channels and containers is similar to the COMMAREA-based application development explained previously. The major difference is in the creation of the data binding classes. This would be expected, because the structure of COMMAREA and channels are quite different.

1. To generate the Data Binding classes, navigate to **File** → **New** → **Other** → **J2C** → **CICS/IMS Java Data Binding**.

   This will launch the CICS Java Data Binding Wizard previously shown in Figure 5-5 on page 110.

2. Click **Next** to start the Data Import Wizard.

3. In the Choose Mapping drop-down list select **COBOL CICS Channels to Java**, as shown in Figure 5-41.

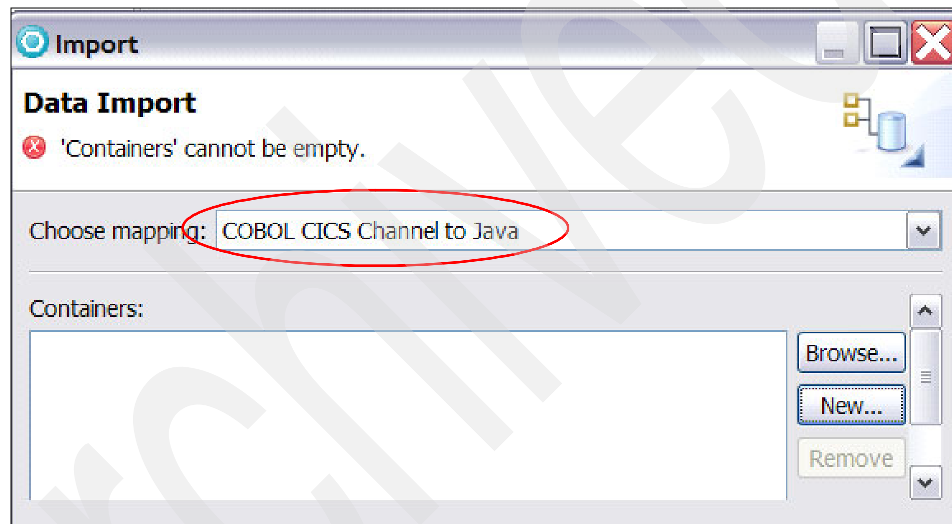


Figure 5-41 Selecting Mapping type for Channels

   RAD supports converting COBOL programs and C programs written for channels and containers into Java equivalent binding classes. Our sample application is a COBOL program, so we selected the option shown in Figure 5-41.

4. Once mapping is selected, we must specify a container. Because we have not yet created any binding classes for channels, we clicked **New.** A Data Import window is displayed where we specify the COBOL program or Copybook where the container structures are defined, as shown in Figure 5-42.

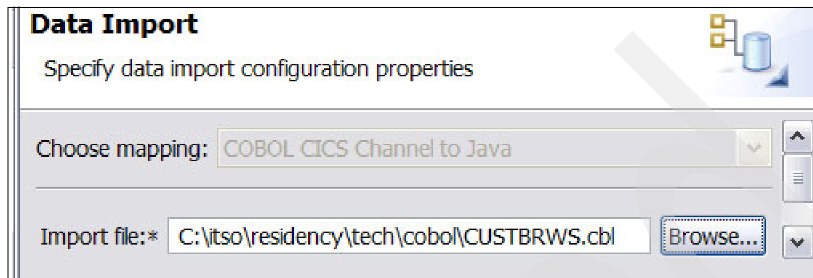


Figure 5-42 Selecting COBOL file for importing Channels

5. Click **Next** to select the container structures. The platform selected is z/OS. All of the COBOL structures available in the file are displayed within the Data structures section. We need to select each of the data structures required. We are only interested in the Request-Information data structure, so select that option, as shown in Figure 5-43.

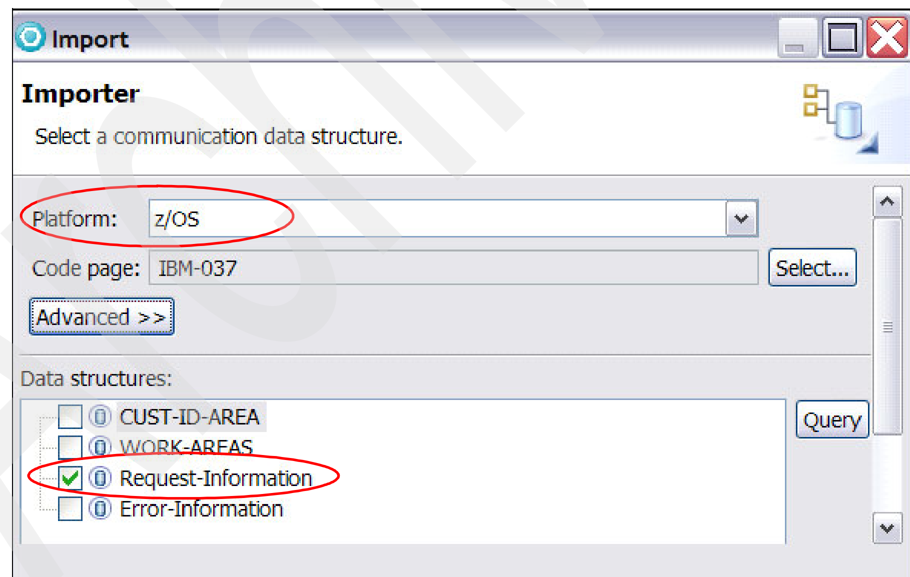


Figure 5-43 Selecting the container structures

6. Click **Finish.** This closes the pop-up window and returns us to the Data Import window. We can see that the container Request-Information is now added into the containers section, as shown in Figure 5-44.

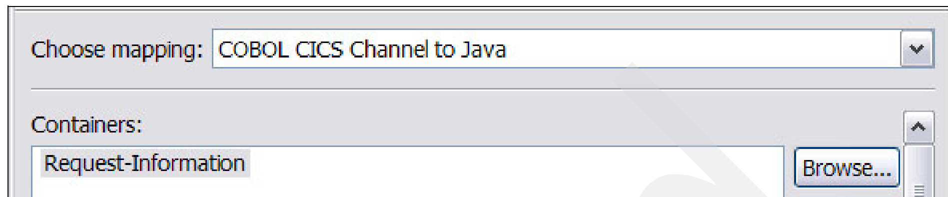


Figure 5-44 Adding the container

A single channel can encapsulate many containers. The data structure definitions of each of these containers may not be defined by a single COBOL or C file. This means we may have to import the file multiple times to obtain the definitions for of all the containers. If the Java data mapping is already created for a container, use the **Browse** button to find and select the Java binding class.

In the example, the CUSTBRWS COBOL program expects a channel which contains a single container. The structure of this container is defined in file CUSTBRWS.cbl under Request-Information.

7. Click **Next** on the Data Import window to save the binding properties.

Archived

8. Click the CICS Channel text on the left side to display the property window for this channel. Select the project name and package name where the generated data binding classes will be stored. Enter the Channel's data binding class and the name of the channel, as shown in Figure 5-45.

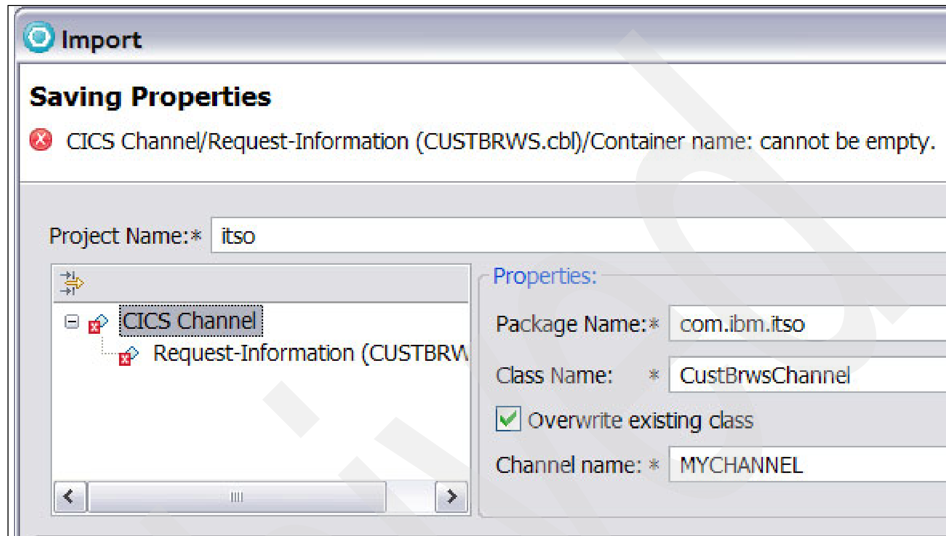


Figure 5-45 Entering Channel Properties

The Saving Properties panel is displayed.

9. On the left pane, select **Request-Information**. This will display the container properties section on the right side. Enter the Container name and select the container type, as shown in Figure 5-46.

   In this example the container is a CHAR type, signifying that it contains character data that will be converted appropriately by CICS.

   > **Note:** Because our container is a CHAR container, data conversion support is provided by CICS. However, if we were using a BIT container the RAD data conversion support would be used to convert data to and from EBCDIC using the chosen encoding. See Figure 5-43 on page 142.

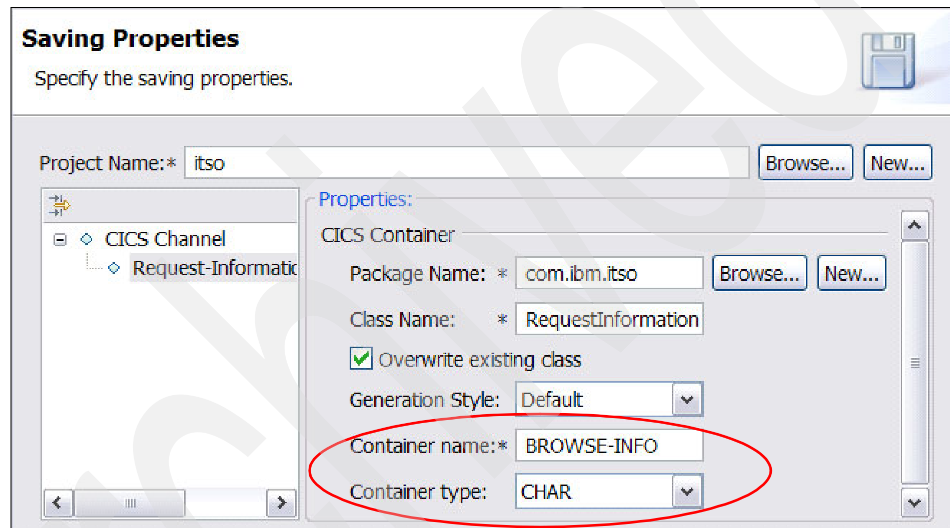


Figure 5-46 Entering container properties

10. Click **Finish** to generate the binding classes for the channels and containers.

    Examining the project using the package explorer, we see that the wizard created two Java files, as shown in Figure 5-47: one for the channel and one for the container.

Figure 5-47 Classes generated for channels and containers

Archived

The channel class is inherited from com.ibm.connector2.cics.ECIChannelRecord and the container is encapsulated within the channel, as shown in Figure 5-48.

```
public class CustBrwsChannel extends com.ibm.connector2.cics.ECIChannelRecor
    private static final long serialVersionUID = -8141597004983786720L;
    private com.ibm.itso.RequestInformation requestInformation = null;

    public CustBrwsChannel() throws javax.resource.ResourceException {
        super("MYCHANNEL");
        requestInformation = new com.ibm.itso.RequestInformation();
    }
```

Figure 5-48 Generated Code for Channels and Containers

We have now generated the binding classes necessary to transfer the data from the client application to the back-end CICS program.

We will generate the J2C bean in the same way as we generated it for the COMMAREA-based application. In this example we have created a J2C bean with the name CustBrws. When the J2C bean wizard finishes generating the J2C bean, two additional classes will be created, as shown in Figure 5-49.

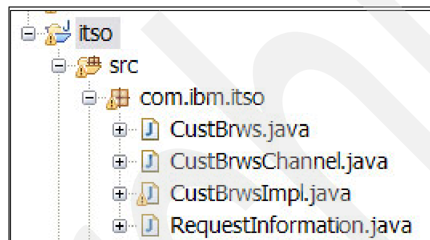


Figure 5-49 Classes Generated by J2C Bean Wizard

We need to add a new bean method to this wizard generated implementation class in the same way as we did for the COMMAREA classes. The Add Java Method window of the wizard displays, as shown in Figure 5-50 on page 147.

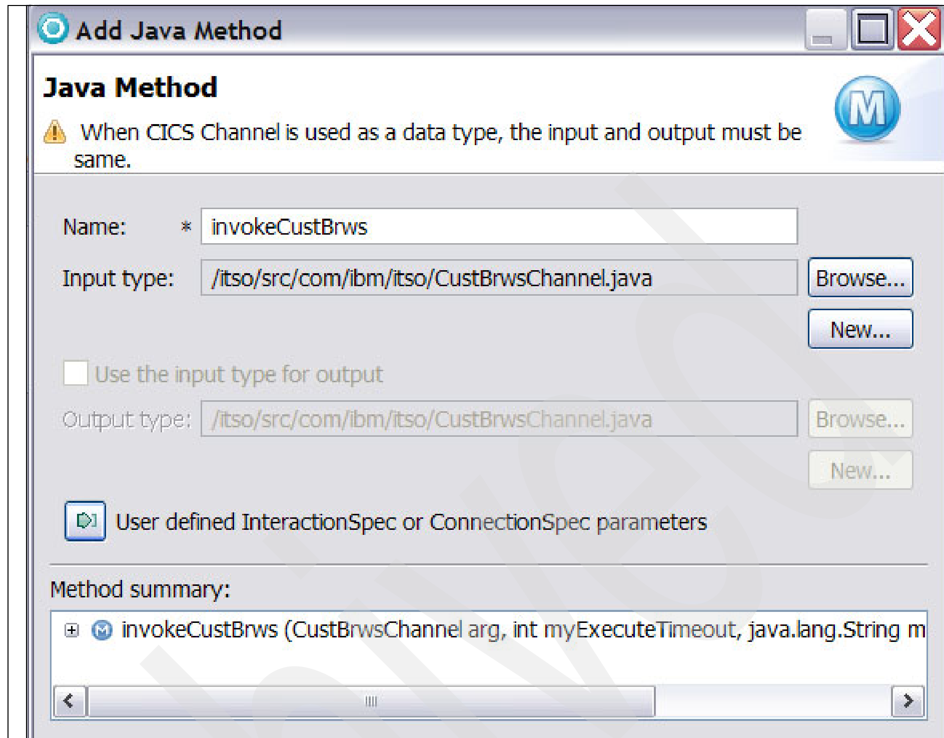


Figure 5-50 Adding a Java method to the J2C bean

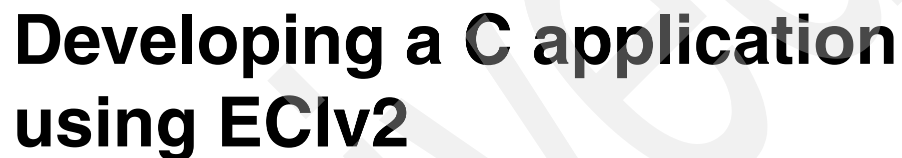
When channels are used as a data type, the return type of the added Java method must be the same as the input type. We therefore must send a channel and receive a channel when we invoke this J2C method. In this example, for the user defined Interaction spec, we selected only the executionTime out and the function name.

11. Click **Finish** to add the method.

The J2C bean for handling channel type requests is now complete. Like other Java beans, this J2C bean can be reused in other applications deployed on the application server.

Archived

6



Developing a C application using ECIv2

This chapter describes the C APIs available for interacting with a CICS application. It presents a sample C client application that uses the ECIv2 API to invoke a back-end CICS application using the CICS Transaction Gateway.

Archived

© Copyright IBM Corp. 2009. All rights reserved.

6.1 Introduction to the ECIv2 API

The ECIv2 API is introduced as part of the CICS TG V7.2 release and enables a client application to communicate with a CICS TG running on a remote machine in much the same manner as the Java APIs.

An application developer using this API is able to call COMMAREA-based CICS applications. Applications may make use of the transaction support in CICS either by having each program run in its own transaction or by multiple calls within an extended logical unit of work (LUW).

Applications written using this API can be developed for the Windows, Linux, or UNIX platforms that are supported by the CICS TG.

This function is similar to that provided by the existing Client APIs in the CICS TG. However an application written using ECIv2 is not required to run on the same machine as a CICS TG installation.

6.1.1 Current restrictions using ECIv2 API

When writing applications using the version of ECIv2 shipped with CICS TG v7.2, the following restrictions apply:

- Applications can only communicate with a CICS TG using TCP/IP. The use of SSL and the local protocol are not supported.
- Two-phase commit transactions using the XA protocol are not available.
- Communication with CICS servers is only available when using ECI with a COMMAREA. Support for channels and containers is not provided.

6.1.2 Characteristics of the ECIv2 API

A topology that involves ECIv2 applications may possess some of the following characteristics:

- Multiple applications on separate machines connecting to a single Gateway daemon.
- Client applications may be deployed on a machine that is different than the system on which the CICS TG is running.
- C applications can make use of the IPIC protocol for connecting to CICS.

Archived

6.1.3 Documentation

The documentation for the API is provided as a set of html files which are contained in a compressed file called ctgclientdoc.zip in the <install location>/docs directory, or they may be obtained from the Information Center at the following Web page:

http://publib.boulder.ibm.com/infocenter/cicstgzo/v7r2//index.jsp?fdoc=aimctg

6.1.4 Overview of an ECIv2 application

Each ECIv2-based client application should implement a similar logic flow in order to communicate with a program running in CICS. This flow is as follows:

1. Connect to the CICS TG using a known host name and port number. This step must be performed for the following steps to complete successfully.
2. (Optional) Obtain a list of the available CICS servers that the CICS TG is configured to communicate with.
3. Flow one or more ECI requests to a CICS program and receive a response back in a COMMAREA.
4. Close the connection to the CICS TG.

6.2 Sample CICS C application overview

The sample *custbrowse.c* client application shown in Appendix C, “Sample ECIv2 client” on page 227 interacts with the back-end CICS *CUSTPROG* sample application. The client application performs one of the following functions:

- Create a new customer record.
- Retrieve a customer record.
- Update an existing customer record.
- Delete a customer record.

Data is passed between the client application and the back-end CICS application using a COMMAREA.

Following is the description of the COMMAREA-based CUSTPROG CICS COBOL program and its installation into the back-end CICS TS environment.

The CICS sample COBOL server program has the following components:

- CUSTPROG.cbl

  COBOL sample program that performs Create, Read, Update, Delete of records in a VSAM file.

- CUSTREC.cbl

  Copybook that describes the layout of the VSAM file

- CUSTIORQ.cbl

  Copybook for the COMMAREA layout

- CUSTLOAD.cbl

  COBOL utility program - LINK to this using CECI to load records in the VSAM file

- CUSTVSAM.jcl

  IDCAMS statements to define the VSAM file

Execute the COMMAREA-based server program

To create, read, update, or delete from the VSAM file, place data in the COMMAREA as appropriate and LINK (or the equivalent) to CUSTPROG.

6.2.1 The COMMAREA structure

The COMMAREA used by our CICS application stores all information sequentially as character data. Table 6-1 details each of the fields and their sizes.

Table 6-1 COMMAREA structure

Field	Size (bytes)
Request type	1
Return code	2
ID	8
Last name	20
First name	20
Company	30
Address line 1	30
Address line 2	30
City	20

Archived

Field	Size (bytes)
State	20
Country	30
Mail code	20
Phone number	20
Date of last update	8
Return comment from application	50

The Request type field can take one of four values:

- C: Create a customer record
- R: Retrieve a customer record
- U: Update a customer record
- D: Delete a customer record

Specifying a request type other than these four causes the program to abend with abend code CUS2.

The total size of the expected COMMAREA is 309 bytes. If a COMMAREA that is larger or smaller than 309 bytes is supplied to the program, the program abends with an abend code of CUS1.

Example 6-1 shows the sample CUSTPROG read logic. The code expects a customer ID to be passed using the COMMAREA and the customer information related to that customer ID is returned to the client in the COMMAREA. In addition to reading information, the CUSTPROG sample program contains business logic for adding, deleting, and updating the VSAM records.

Example 6-1 The CUSTPROG read logic

```
2000-DO-READ.
           EXEC CICS READ FILE('CUSTFILE')
                  RIDFLD(CustomerId)
                  INTO(CUST-FILE-LAYOUT)
                  RESP(RESP-FLD)
                  END-EXEC.
           EVALUATE RESP-FLD
               WHEN DFHRESP(NORMAL)
                   PERFORM 5200-MOVE-RECORD-TO-COMMAREA
                   MOVE '00' TO RET-CODE
                   MOVE 'Customer information retrieved'
                         TO RETURN-COMMENT
                   PERFORM 7100-RETURN-ERROR-MESSAGE
```


Archived

```
            WHEN DFHRESP(NOTFND)
                MOVE '04' TO RET-CODE
                MOVE 'Fund not found'
                      TO RETURN-COMMENT
                PERFORM 7100-RETURN-ERROR-MESSAGE
            WHEN OTHER
                MOVE '04' TO RET-CODE
                MOVE 'Error reading customer file'
                        TO RETURN-COMMENT
                PERFORM 7100-RETURN-ERROR-MESSAGE
        END-EVALUATE.
```

Installing the COMMAREA-based sample program on CICS

Perform the following steps to install the COMMAREA-based sample program on CICS.

1. Compile both the CUSTPROG and CUSTLOAD programs. They need access to the CUSTREC and CUSTIORQ copybooks.
2. Define the VSAM file using IDCAMS.
3. Define a PROGRAM definition for the above programs.
4. Define a FILE definition named CUSTDATA, fixed length, updatable, readable, writeable, and so forth.

LINK to the CUSTLOAD program using CECI to install sample data into the VSAM file. You can do a quick edit on the CUSTLOAD.cbl file if you wish to load specific names in the records in the VSAM file.

6.2.2 Data conversion

The back-end CICS application executes on the z/OS platform in an EBCDIC environment where as the C client application executes in a UNICODE environment. To ensure that the COMMAREA data passed between the two programs is properly converted we created a DFHCNV table entry for the CUSTPROG program with CLINTCP=437,SRVERCP=037. For further details about using DFHCNV, refer to the *CICS Transaction Server on z/OS V3.2 Internet Guide*, SC34-6831.

6.3 Developing the ECIv2 sample application

In this section we describe the development of a C client application that implements the ECIv2 API calls to communicate with the back-end CICS CUSTPROG sample application.

6.3.1 Sample client application overview

We developed a client application in C that interacts with the CICS application to create, retrieve, update, and delete customer data as requested. The client application is started from the command line and provides a text-based interface for performing the required actions.

Starting the application

The application can take five startup arguments. Two are required and three are optional. These arguments are as follows:

- -h

  The host name or IP address of the Gateway daemon that we want the application to connect to. This is a required parameter and so we specified wtsc59.itso.ibm.com.

- -n

  The port that the Gateway daemon's TCPHandler is listening on. This is a required parameter and so we specified 2006.

- -s

  The CICS server that is running the program that we want to use. This parameter is optional, and if one isn't specified then the application will prompt the user to choose one from the list of available CICS servers the Gateway daemon knows about.

- -u

  The user ID that should be used to validate the user against a secure connection. This is an optional parameter.

- -p

  The password for the user ID specified using the -u flag. This is an optional parameter.

Archived

Example 6-2 shows the command we used to run the application.

Example 6-2 Starting the C sample application

```
custprog -h wtsc59.itso.ibm.com -n 2006
```

Running the sample client application

Because we did not specify a CICS server on the command line, we are prompted to choose one from those available to the Gateway daemon, as shown in Example 6-3.

Example 6-3 Choosing a CICS server

```
Please choose the CICS server to use:
1) CIC1IPIC
2) SC59CIC1
Select server:
```

The sample application continues and presents a menu of options from which to select. See Example 6-4.

Example 6-4 Sample application main menu

```
Customer Control Program
Choose option:
1) Create a customer record
2) Retrieve a customer record
3) Delete a customer record
4) Update a customer record
5) Exit
Choice:
```

Once an option is selected from the menu, the sample application prompts for any information specific to that option. Retrieving or deleting a customer record only requires a customer number. Creating or updating a customer record prompts you for all of the customer record information.

The information provided is stored in the corresponding fields in the COMMAREA structure and the back-end CICS program is called. Upon return from the program call the information from that operation is displayed. When retrieving a customer record, the customer record information is displayed. However, all other operations return a message, obtained from the COMMAREA, that indicate the success or failure of the operation.

Archived

If an error occurs at any time, whether the application is unable to communicate with the Gateway daemon or there is a problem calling the CICS program, an error is displayed.

6.3.2 The application development environment

We chose to develop our sample client application on a Linux workstation using the GNU C Compiler (gcc) to compile and link the application.

In order to gain access to the API functions, the sample application must include the header files ctgclient.h and ctgclient_eci.h. These header files are stored in the include sub-directory for the product install.

We defined the symbol CICS_LNX in the compilation step by specifying the -D flag to indicate to the API functions that the application was running on a Linux platform. Each platform has its own symbol which will need to be defined and these are shown in Table 6-2.

Table 6-2 Symbol for compilation by platform

Platform	Symbol
Linux	CICS_LNX
Windows	CICS_W32
AIX	CICS_AIX
Solaris	CICS_SOL
HP-UX	CICS_HPUX

When linking the sample application the API functions are contained in the libctgclient.so shared library. Without this library the application will not compile or run, so it must be available. This library is located in the <install location>/lib directory and symlinked into /usr/lib on Linux and UNIX platforms.

Archived

The shared libraries for supported platforms are shown in Table 6-3.

Table 6-3 Shared library by platform for ECIv2

Platform	Shared library
Linux	libctgclient.so
Windows	ctgclient.dll
AIX	libctgclient.a
Solaris	libctgclient.so
HP-UX	libctgclient.sl
HP Itanium	libctgclient.so

The header and shared library files

If the CICS TG is not installed on the platform where you will develop the application, then the header and shared library files needed are available as part of the CICS TG installation. They are contained in a compressed file named ctgredist.zip, located in the <install directory>/deployable directory. This compressed file can be copied from the CICS TG deployable directory to the platform where the application will be developed. Once copied, extract the files into the appropriate local directories.

Interpreting return codes

The interpretation of return codes from the API calls is greatly improved with the addition of the CTG_getRCString function. A client application can use this function to obtain a string representation of a return code that provides more meaningful error information to a user or application developer.

The CTG_getRCString function takes as input the return code received along with a pointer to a character array owned by the application into which the string representation will be stored. Example 6-5 shows the use of this function in our application.

Example 6-5 Using the CTG_getRcString function

```
char rcString[CTG_MAX_RCSTRING + 1];

CTG_getRcString(rc, rcString);
printf(“Return code: %s\n”, rcString);
```

Archived

6.3.3 Connecting to the CICS Transaction Gateway

To connect to a CICS TG, we must provide the sample application with the host name or IP address for that system, and the port number on which the TCPHandler is listening. For our sample application the host name wtsc59.itso.ibm.com and port 2006 were specified as command line arguments.

Using this information, the CTG_openRemoteGatewayConnection function establishes the connection to the CICS TG. In addition to the host and port information, a pointer to a CTG_ConnToken_t and a timeout value for the connection attempt is passed.

For our application the timeout value was set to zero as we were prepared to wait as long as necessary to establish the connection. See Example 6-6.

Example 6-6 Connecting to the Gateway daemon

```
char *hostname = "wtsc59.itso.ibm.com";
int port = 2006;
CTG_ConnToken_t gatewayToken;
int rc=0;

rc = CTG_openRemoteGatewayConnection(hostname, port, &gatewayToken, 0);
```

Note: The local: protocol for local mode operation is not available when using ECIv2.

The function call returns CTG_OK if the connection was successful, or an error return code indicating one of the possible failures.

CTG_ConnToken_t

This type represents a connection to a CICS TG and is required for all subsequent calls to that Gateway daemon. The CTG_ConnToken_t can only be used on the thread on which it was created, and a separate token is required for each connection to the Gateway daemon.

The connection token represents our connection to a CICS TG and is used by all subsequent API calls that interact with the Gateway daemon. The token is restricted to the thread that created it, so a multi-threaded application must create a separate connection for each thread.

Archived

Checking the connection

Our Gateway daemon was configured to have connection logging turned on by having the connectionlogging=on setting in the configuration file. When the application connects, a message similar to the one in Example 6-7 is written to the information log.

Example 6-7 Log message showing our ECIv2 client has connected to the Gateway

```
11/25/08 15:07:21:326 [0] CTG6506I Client connected:
[ConnectionManager-0] -
tcp@Socket[addr=/9.57.138.115,port=60243,localport=2006]
```

Possible error codes

Some of the more common error codes are detailed here along with the suggested corrective actions.

- CTG_ERR_BADHOST

  This return code indicates the value in the host address parameter is not a valid host name, IPv4, or IPv6 address. Check that the host address parameter for the address of the CICS TG is correct.

- CTG_ERR_BADPORT

  We receive this return code if we specify a port number that is less than one or greater than 65535. Here we should check the port parameter to ensure the value is in the correct range.

- CTG_ERR_NULLADDRESS

  This return code indicates that the address parameter which we provided pointed to NULL. When we receive this return code we need to check that the pointer we have for the address is pointing at the value for the address.

- CTG_ERR_CONNECTTIMEOUT

  The connection took longer to establish than the time allowed value specified in the open call.

- CTG_ERR_CONNECTFAILED

  The application was unable to establish a connection with the Gateway daemon. If the address and port supplied as parameters are correct then it may be necessary to investigate what caused the connection failure.

Working in a multi-threaded environment

When writing an application that uses multiple threads to call CICS programs, care needs to be taken that the connection token is only used on the thread that opened the connection.

If an API call detects that a connection token is being used on a different thread to the one that created it, it will return the return code CTG_ERR_TIDMISMATCH and the request will fail.

6.3.4 Communicating with the CICS servers

A client application can only connect to CICS TSs that the CICS TG has been configured to communicate with. The CTG_listSystems call is provided to supply an application with a list of the available CICS TS.

Example 6-8 demonstrates the use of the CTG_listSystems call.

Example 6-8 Sample code to retrieve a list of CICS systems

```
#define SERVER_NUM 10

CTG_listSystem_t servers[SERVER_NUM];
int serverCount = SERVER_NUM;
int rc = 0;

rc = CTG_listSystems(gatewayToken, &serverCount, servers);
```

The gatewayToken parameter is the same token that was created by the CTG_openRemoteGatewayConnection function.

The serverCount parameter is both an input and output parameter. On input it specifies the number of entries in the servers array. On output it specifies the number of CICS systems that are available.

The servers parameter is an array of CTG_listSystem_t structs that are populated with the CICS server information. Each CTG_listSystem_t structure contains the name and description of an available CICS server.

Return code values

Following are some of the return code values received from the CTG_openRemoteGatewayConnection call.

- CTG_OK

  Returned on a successful call.

- CTG_ERR_MORE_SYSTEMS

  Indicates more CICS servers are defined to the CICS TG than entries available in the List parameter array. You may choose to work with the limited list of CICS servers returned in the array or reissue the request after creating

Archived

a larger array list using the value in Systems. This can be achieved by using malloc() to create the initial array and then using realloc() to create a large one using the returned size as input.

- CTG_ERR_NO_SYSTEMS

  Indicates there are no CICS servers defined to the CICS TG. This situation requires changes to the CICS TG configuration and should be reported to the CICS TG administrator.

6.3.5 Calling the CICS back-end program

Once we have the name of a CICS server, either explicitly defined within the application or obtained dynamically using the list systems API call, we can now issue calls to a program running on the CICS server.

Using the COMMAREA interface

All input and output communication between the sample client application and the CICS sample CUSTPROG program is done using the COMMAREA, which has a defined structure stored in a copybook. We define a structure in our C application, which mirrors the COMMAREA definition as shown in Example 6-9.

Example 6-9 Structure representing COMMAREA within the CUSTIORQ.cbl copybook

```
struct Commarea {
   char requestType;
   char returnCode[2];
   char id[8];
   char lastName[20];
   char firstName[20];
   char company[30];
   char address1[30];
   char address2[30];
   char city[20];
   char state[20];
   char country[30];
   char mailCode[20];
   char phone[20];
   char lastUpdateDate[8];
   char comment[50];
};
```

Note: The string data stored in the structure is not guaranteed to be NULL terminated, so care needs to be taken when using these fields to avoid segmentation errors.

Archived

In our sample application a COMMAREA struct was created for each request type and populated with the necessary data. Once the request is complete the COMMAREA struct is discarded.

Setting the ECI parameters

A number of ECI parameter values must be set in order for a C application using the ECIv2 API to connect to a CICS program. Additionally, there are several optional parameters that may be set.

To encapsulate all of these parameter values, the ECIv2 API provides an CTG_ECI_PARMS structure. The fields and their values are as follows:

- Version (eci_version)

  This field is used to store the version of ECI that is being used. For our ECIv2 API application we set it to ECI_VERSION_2.

- Call type (eci_call_type)

  We set this field to ECI_SYNC as this is the only call type supported by the ECIv2 API.

- Program name (eci_program_name)

  We set this to 'CUSTPROG' which is the name of the program defined in our CICS region.

- CICS server (eci_system_name)

  We set this to the server name specified on the command line at startup or the name of the CICS server chosen from the list systems command. Optionally this parameter can be left blank to use the default CICS server for the Gateway daemon to which we are connecting.

- User ID (eci_userid)

  Because we enabled security in our CICS region, this field contains a RACF user ID.

- Password (eci_password)

  This field contains the RACF password for the user ID we previously specified.

- Commarea (eci_commarea)

  We set this field to be a pointer to the COMMAREA struct that contains the information for this call.

- COMMAREA length (eci_commrea_length)

  The length of the COMMAREA needs to be set to the largest amount of data a CICS application could be expected to send or receive. In our case, this is 309 bytes.

Archived

- Extend mode (eci_extend_mode)

  We set this to ECI_NO_EXTEND, which informs CICS that it should SYNCPOINT when the response from the CICS program is sent back to our client application.

Additional parameters

There are other parameters available in the CTG_ECI_PARMS structure that our sample application did not use. These are as follows:

- CICS transaction identifier (eci_transid)

  The value set for this parameter is will be reflected in the EIB block as the EIBTRNID field.

- ECI timeout (eci_timeout)

  This parameter controls how long the CICS TG should wait for a response from CICS to the program call. If the time taken to for the CICS program to execute the request is longer than this value a return code of ECI_ERR_REQUEST_TIMEOUT is received.

- Mirror transaction (eci_tpn)

  Setting this parameter controls the mirror transaction started in CICS to initiate the program. If this value is not set then, the mirror transaction will be CSMI if a CICS TG for Multiplatform is used, or CPMI if using CICS TG for z/OS.

- LUW Token (eci_luw_token)

  If multiple calls to CICS programs need to be made within a single transaction then this field will contain the token used to identify the transaction. The first request in a transaction will set this to ECI_LUW_NEW and after the request has completed will contain the token to use on all subsequent requests.

Before setting any of the parameters, the contents of the ECI_PARMS structure should first be set to NULL using the **memset** command. Then the individual fields should be assigned. An example of how this is done is shown in Example 6-10.

Example 6-10 Setting up the contents of the CTG_ECI_PARMS structure

```
CTG_ECI_PARMS eciParms;
memset(&eciParms, 0, sizeof(CTG_ECI_PARMS));

/*Set the necessary ECI parameters*/
eciParms.eci_version = ECI_VERSION_2;
eciParms.eci_call_type = 2;
memcpy(eciParms.eci_program_name, program, strlen(program));
memcpy(eciParms.eci_system_name, server, strlen(server));
memcpy(eciParms.eci_userid, userid, strlen(userid));
```

Archived

```
memcpy(eciParms.eci_password, password, strlen(password));
eciParms.eci_commarea = commarea;
eciParms.eci_commarea_length = COMMAREA_LENGTH;
eciParms.eci_extend_mode = ECI_NO_EXTEND;
```

Executing the ECI call

Once the COMMAREA and CTG_ECI_PARMS are created and initialized appropriately, we are ready to make a request to the CICS back-end program and process the response.

We issue the CTG_ECI_Execute call, which takes a CTG_ConnToken_t and a pointer to our CTG_ECI_PARMS structure, as shown in Example 6-11. The ECI_PARMS structure, and our COMMAREA struct, are updated as a result of issuing this call.

Example 6-11 Calling the CICS program

```
int rc;
rc = CTG_ECI_Execute(gatewayToken, &eciParms);
```

The return value from this call indicates whether the call was successful or not. If successful, this value will be ECI_NO_ERROR. Otherwise, the return value contains an indication of a specific error.

Problems encountered with the API call or communicating with the Gateway daemon result in an error code that starts with the prefix CTG_ . Problems running the CICS program or the ECI parameters specified will result in an error code that starts with the prefix ECI_.

Information about the return codes can be found in the html documentation included with the product or in the Information Center. Alternatively, the ctgclient.h header file contains a description of all the possible return codes.

Processing the ECI call return codes

We first check for any CTG_ type return codes followed by a check for any ECI_ type return codes.

- CTG_ERR_GATEWAY_CLOSED

  This return code indicates that the Gateway daemon has closed the connection. This could be caused by the systems administrator shutting down the Gateway daemon or the connection remained idle longer than the idle time out value set in TCPHandler.

Archived

- CTG_ERR_WORK_WAS_REFUSED

  Two situations can occur that generate this return code.

  – One is caused when the Gateway daemon is unable to allocate a Worker thread for our request to run on. In this situation we can retry the request in the event that a Worker thread has become available. If the request continues to fail, the Gateway administrator may need to adjust the settings to allow the work through.

  – The other situation occurs when the Gateway daemon is in the process of shutting down while it is still waiting on outstanding work to complete. In this situation we have to wait for the Gateway daemon to complete its shutdown processing and be restarted before we are able to do issue any more requests.

- ECI_ERR_TRANSACTION_ABEND

  We receive this return code if the CICS program we call abends. An abend code is also stored in the eci_abend_code field of the CTG_ECI_PARMS struct and should be inspected to determine whether it was a program or system abend that caused the problem. For our application, receiving a CUS1 or CUS2 abend would indicate a problem with the COMMAREA input.

- ECI_ERR_NO_CICS

  Receiving this return code indicates that the CICS server that we attempted to communicate with is currently unavailable. To diagnose the problem it may be necessary to examine diagnostic information provided by the Gateway daemon. Further details can be found in *Exploring Systems Monitoring for CICS Transaction Gateway V7.1 for z/OS*, SG24-7562.

- ECI_ERR_UNKNOWN_SERVER

  If a CICS server is specified that is not defined to the CICS TG we are connected to then any requests sent will receive this return code. To avoid this problem, use the CTG_listSystems function, as described in 6.3.4, “Communicating with the CICS servers” on page 161 to ensure you connect to a known CICS server. If communicating with a CICS TG on z/OS this error will not be returned as the Gateway cannot distinguish between an EXCI connection that doesn’t exist and one that is currently unavailable. In this situation the return code ECI_ERR_NO_CICS will always be returned.

Archived

6.3.6 Closing the ECI connection

When the application completes its work it closes the connection to the Gateway daemon as shown in Example 6-12. This releases resources held by the Gateway daemon and invalidates the CTG_ConnToken_t. Any subsequent calls issued that attempt to use this token will receive an CTG_ERR_GATEWAY_CLOSED return code.

Example 6-12 Closing our connection to the Gateway daemon

```
int rc;
rc = CTG_closeGatewayConnection(&gatewayToken);
```

Upon successful completion, the function receives a CTG_OK return code.

The message recorded in the Gateway daemon log when the client successfully disconnects is shown in Example 6-13.

Example 6-13 Gateway daemon log output

```
11/25/08 15:07:34:034 [0] CTG6507I Client disconnected:
[ConnectionManager-0] -
tcp@Socket[addr=/9.57.138.115,port=60243,localport=2006], reason =
ECIv2 client application closed the connection
```

If the application gets a return code other than CTG_OK, this could indicate that the connection is already closed or there has been an error in the API. It may be possible to retry the failing call or the application should consider closing all its connections and terminating.

Close all ECI connections

In the situation where the references to one or more CTG_ConnToken_t have been lost, it is still possible to close the connections in a controlled manner by using the CTG_closeAllGatewayConnections function.

This function is intended for use in an application error handler when the application is in an unrecoverable state. Following a call to the CTG_closeAllGatewayConnections function it is recommended that the application be terminated.

Archived

6.4 Problem determination

If a problem occurs during the running of a program there are a several places we can look to try and determine the problem.

6.4.1 CICS server reported errors

If we receive an ECI error, such as a transaction abend, then more details are available in the CICS log.

If the return code is CICS region is unavailable (ECI_ERR_NO_CICS) or unknown (ECI_ERR_UNKNOWN_SERVER), check the Gateway daemon logs and configuration settings to see if the server is known to the CICS TG and if there are problems communicating with it.

6.4.2 Application errors

For error codes that start with the prefix CTG_ we can use the API trace to help determine the cause of the problem.

Controlling trace through API functions

The API trace is controlled through the function CTG_setAPITraceLevel(), which takes as input an int value indicating the level of trace desired. Table 6-4 shows the available trace levels along with the amount of trace details provided.

Table 6-4 API trace levels and their meaning

Trace level	Output detail
CTG_TRACE_LEVEL0	No trace output
CTG_TRACE_LEVEL1	Exceptions only
CTG_TRACE_LEVEL2	Events
CTG_TRACE_LEVEL3	Function entry and exit
CTG_TRACE_LEVEL4	Debug information

The trace levels are cumulative, meaning that CTG_TRACE_LEVEL2 contains all information made available by CTG_TRACE_LEVEL1 and so on.

By default, the trace information is written to the standard error stream (stderr). However, we can direct the API to write the trace information to a file by invoking the CTG_setAPITraceFile() function.

Archived

Controlling trace through environment variables

APAR PK76778 adds functions that allow for trace to be turned on and sent to a file without changing the code of the application or having to recompile it.

Turning trace on can be done by setting the environment variable CTG_CLIENT_TRACE_LEVEL to a value of 0, 1, 2, 3, or 4. Controlling the trace destination can be done by setting the environment variable CTG_CLIENT_TRACE_FILE to the name of the file to which to write the trace information. These environment varaiables need to be set before the application is started.

Using trace to analyze a problem

As an example, when our Gateway daemon was unavailable our sample client applications CTG_openRemoteGatewayConnection function recived the CTG_ERR_CONNECTFAILED return code.

We set the API trace level to CTG_TRACE_LEVEL1 and again attempted to connect to a Gateway daemon that wasn't there. We observed the trace output written to stderr shown in Example 6-14.

Example 6-14 Trace output for an unavailable Gateway daemon

```
**** IBM CICS Transaction Gateway CTGClient DLL Version 7.2.0.0
Diagnostic trace - build(c720-20081025) ****
25/11/08 11:59:56:083 [0x00000DBF,0xB7F5A6C0] [3006] *EXC*
clapi_gwycon.connectToGatewayUnable to connect to Gateway: Connection
refused
25/11/08 11:59:56:083 [0x00000DBF,0xB7F5A6C0] [2004] *EXC*
clapi_ctg_gwytok.openClientSocket() Failed to connect to Gateway :2007
25/11/08 11:59:56:083 [0x00000DBF,0xB7F5A6C0] [3228] *EXC*
clapi_socketcomms.flow Unable to write data to socket rc=-1
25/11/08 11:59:56:083 [0x00000DBF,0xB7F5A6C0] [3229] *EXC*
clapi_socketcomms.flow Unable to send data to Gateway: errno=32
```

The line of trace output containing the phrase 'Connection refused' indicates that the Gateway daemon is not listening on the IP address and port number specified in our client application. The next line of output reports the IP address and port number that the CTG_openRemoteGatewayConnection was using. This should match the IP address and port where the Gateway daemon is running. If it does, an administrator will need to check that the Gateway daemon is running correctly. Otherwise, the application should be altered to use the correct IP address and port number.

Archived

Migrating a Client application to ECIv2

This chapter describes the steps required to migrate an existing ECI application from the Client API to the new ECIv2 API.

Archived

© Copyright IBM Corp. 2009. All rights reserved.

7.1 Introduction to application migration

Existing C applications which use the ECI Client API in order to interact with a CICS server program require changes to use the ECIv2 API functionality. In this chapter we discuss what changes are required to migrate an application.

There are four steps required to migrate an existing application that uses the Client API to use the new ECIv2 API:

1. Change the included header files.
2. Add code to connect to a Gateway daemon.
3. Change any list systems and ECI calls to use the new functions.
4. Compile the updated source code against the new shared library.

> **Note:** It is not possible to migrate EPI and ESI applications to use the new API. An application cannot contain both Client API and ECIv2 API calls

7.2 Changing the header files

There are two C header files, ctgclient.h and ctgclient_eci.h, which contain the function prototypes and type definitions required to use the ECIv2 API.

The include statement for cics_eci.h used by a Client API application is shown in Example 7-1.

Example 7-1 Header file included for Client API implementation

```
/*Client API header file*/
#include <cics_eci.h>
```

The two ECIv2 header files that replace the cics_eci.h header file for a ECIv2-based application are shown in Example 7-2.

Example 7-2 Header files to include for ECIv2-based on API implemented

```
/*ECIv2 API header files*/
#include <ctgclient.h>
#include <ctgclient_eci.h>
```

Archived

7.3 Managing the Gateway daemon connection

In order to connect the client application using the ECIv2 API to a remote CICS TG it must know the host name or IP address of the system where the Gateway daemon is running, and the port number on which its TCPHandler is listening. In our setup the host name is wtsc59.itso.ibm.com and the Gateway daemon is listening on port 2006.

Additionally, you may need to know the name of the CICS server to which the client requests should be sent.

7.3.1 Opening connections to the Gateway daemon

Before the ECIv2 client application can call a CICS program, it must establish a connection with the Gateway daemon using the CTG_openGatewayConnection function and the information provided as noted in 7.3, “Managing the Gateway daemon connection” on page 173. This call must be made prior to any calls that require the connection token, as this call initializes the token.

If this call fails, no other ECIv2 API functions will succeed due to not having a valid connection token. Further details about using this function can be found in 6.3.3, “Connecting to the CICS Transaction Gateway” on page 159.

7.3.2 Working with a connection token

The type CTG_ConnToken_t is defined by the API to represent a connection token. This token is used or referenced in every API function that communicates with the Gateway daemon.

When opening or closing a connection, a reference to the connection token needs to be passed as a parameter.

When calling a CICS program or getting a list of known CICS systems, the connection token must be passed by value to the CTG_ECI_Execute or CTG_listSystems functions.

Once the connection token is valid, the connection remains open until we close the connection.

In the sample application described in Chapter 6, “Developing a C application using ECIv2” on page 149, we stored the token in a local variable in the main function, which was then passed as a parameter into all the functions for which it was required.

Archived

Example 7-3 shows the code required to create a connection token variable and call the CTG_openGatewayConnection function in order to establish the connection.

Example 7-3 Opening the Gateway connection

```
int rc = 0;
CTG_ConnToken_t connectionToken;
rc = CTG_openGatewayConnection("wtsc59.itso.ibm.com",
                                2006,
                                &connectionToken, 0);
```

Working in a multi-threaded application

If a client application uses multiple threads to perform concurrent CICS calls, each thread will need its own connection token, as they cannot be shared between threads. An application that attempts to use a token on a thread other than the one that opened the connection will receive an error.

7.3.3 Closing connections to the Gateway daemon

Before an application terminates it should close all the connections it had previously established. This should be done when no more calls to the CICS TG are going to be required, as this invalidates the connection token. Example 7-4 shows a connection being closed using the CTG_closeGatewayConnection function.

Example 7-4 Closing the Gateway connection

```
rc = CTG_closeGatewayConnection(&connectionToken);
```

This call is unlikely to fail unless the connection has previously been closed. Further details about using this function can be found in 6.3.6, "Closing the ECI connection" on page 167.

In the event of some error conditions, an application could lose the references to one or more connection tokens.The application can use the function CTG_closeAllGatewayConnections to close all outstanding Gateway daemon connections.

7.4 Migrating Client API function calls

In this section we explain how Client API functions have changed with the new ECIv2 API, and what application changes are needed to use the ECIv2 API version of these functions.

7.4.1 Listing available systems

The list systems function has a number of changes that we need to adapt the program to use. To list available systems you would change from the CICS_EciListSystems Client API to use the CTG_listSystems ECIv2 API.

```
CICS_EciListSystems(cics_char_t *namespace, cics_ushort_t *Systems,
CICS_EciSystem_t *List)
```

to

```
CTG_listSystems(CTG_ConnToken_t gwTok, unsigned short *Systems,
CTG_listSystem_t *List)
```

The namespace parameter, which was always set to NULL, has been removed and replaced with the CTG_ConnToken_t parameter, which we established by after calling CTG_openRemoteGatewayConnection.

The Systems parameter has changed type from the previous CICS TG defined type to a standard C type but retains the same use. On input, it is the size of the List parameter. On output, it contains the number of CICS servers the CICS TG knows about.

The List parameter has changed type as well, but like the Systems parameter, it retains the same functionality. The array we define to hold the results has changed from being of type CICS_EciSystem_t to CTG_listSystem_t. The new type still contains the name of the server and the description.

If the application needs to store the chosen server in a variable, the length of the character array should use the CTG_LIST_SYSTEM_LENGTH definition to ensure there is enough memory allocated to store the name.

Example 7-5 shows the list systems using the old Client API.

Example 7-5 List Systems using the Client API list systems call

```
/*Client API List Systems*/
cics_ushort_t serverCount = 10;
cics_sshort_t rc = 0;
CICS_ECIListSystems_t servers[serverCount];
char chosenServer[CICS_ECI_SYSTEM_MAX+1];

rc = CICS_EciListSystems(NULL, &serverCount, servers);
if(rc == ECI_NO_ERROR){
   memcpy(chosenServer, servers[0].SystemName, CICS_ECI_SYSTEM_MAX);
}
```

Example 7-6 shows how to list systems using the new ECIv2 API list systems call.

Example 7-6 List Systems using the ECIv2 API list systems call

```
/*ECIv2 API List Systems*/
unsigned short serverCount = 10;
int rc = 0;
CTG_listSystems_t servers[serverCount];
char chosenServer[CTG_LIST_SYSTEM_LENGTH+1];

rc = CTG_listSystems(gwTok, &serverCount, servers);
if(rc == CTG_OK){
   memcpy(chosenServer, servers[0].SystemName, CTG_LIST_SYSTEM_LENGTH);
}
```

7.4.2 ECI parameters

The ECI_PARMS type has been replaced by the new CTG_ECI_PARMS type. However, after the initial parameters are set up, the passing of the remaining parameter values is unchanged. This means that you do not need to change the code that fills in the parameter structure once the initial setup instructions have been changed. The migration steps needed are as follows:

1. Change all code that references the ECI_PARMS type to use the new CTG_ECI_PARMS type.
2. The value previously set in the eci_version field must be updated from either ECI_VERSION_1 or ECI_VERSION_1A to ECI_VERSION_2.

Archived

3. All ECI_ASYNC calls need to be converted to ECI_SYNC calls.
4. All ECI_STATE calls, whether synchronous or asynchronous, must be removed.

Example 7-7 shows sample code setting up the old ECI_PARMS type structure.

Example 7-7 ECI_PARMS type structure

```
/*Client API parms structure*/
ECI_PARMS eciParms;
memset(&eciParms, 0, sizeof(ECI_PARMS));
eciParms.eci_version = ECI_VERSION_1A;
eciParms.eci_call_type = ECI_SYNC;
```

Example 7-8 shows sample code for setting up the new CTG_ECI_PARMS type structure.

Example 7-8 CTG_ECI_PARMS type structure

```
/*ECIv2 API parms structure*/
CTG_ECI_PARMS eciParms;
memset(&eciParms, 0, sizeof(CTG_ECI_PARMS));
eciParms.eci_version = ECI_VERSION_2;
eciParms.eci_call_type = ECI_SYNC;
/*Setting the rest of the parameters is the same as the old Client */
/*API*/
```

7.4.3 Calling the CICS program

The Client API function used to call the CICS back-end program, CICS_ExternalCall(ECI_PARMS *EciParms), is replaced with CTG_ECI_Execute(CTG_ConnToken_t gwTok, CTG_ECI_PARMS *EciParms)

The gwTok parameter is the connection token we have stored after initializing it in the CTG_openRemoteGatewayConnection function.

The EciParms parameter has changed from a pointer to an ECI_PARMS struct to a CTG_ECI_PARMS, which we described in 7.4.2, “ECI parameters” on page 176.

Example 7-9 on page 178 shows sample code calling the CICS back-end program using the old Client API.

Example 7-9 Calling the CICS back-end program using the Client API

```
/*Client API*/
cics_sshort_t rc = 0;
ECI_PARMS eciParms;
char *program = "CUSTBRWS";
/*Initialization of the parameters structure*/
memcpy(eciParms.eci_program_name, program, 8);
...
/*Make the call*/
rc = CICS_ExternalCall(&eciParms);
```

Example 7-10 shows sample code calling the CICS back-end program using the new ECIv2 API.

Example 7-10 Calling the CICS back-end program using the new ECIv2 API

```
/*ECIv2*/
int rc = 0;
CTG_ECI_PARMS eciParms;
char *program = "CUSTBRWS";
/*Initialization of the parameters structure*/
memcpy(eciParms.eci_program_name, program, 8);
...
/*Make the call*/
rc = CTG_ECI_Execute(gwTok, &eciParms);
```

Further details about calling CICS programs can be found in 6.3.5, "Calling the CICS back-end program" on page 162.

7.5 Compiling the application

The CICS Transaction Gateway installation includes C header files and a shared library for the ECIv2 API. The header files and shared library are used during application development to compile and link the ECIv2 application program. The shared library is also required at runtime.

The header files and shared libraries for all supported platforms are also packaged in the file ctgredist.zip. This allows ECIv2 applications to be built and deployed on any supported platform.

Table 7-1 on page 179 shows which ECIv2 library should be used based on the Client API library currently being used on each platform.

Archived

Table 7-1 Shared libraries for each platform and API type

Platform	Client API library	ECIv2 library
Windows	cclwin.dll	ctgclient.dll
Linux Intel	libccllnx.so	libctgclient.so
AIX	libcclaix.a	libctgclient.a
HP-UX	libcclhpux.sl	libctgclient.sl
HP Itanium	libcclhpux.so	libctgclient.so
Solaris	libcclsol.so	libctgclient.so

Note: When developing the application on a Windows platform, the ctgclient.lib and the ctgclient.dll files are required for the compilation of the application.

The CICS TG installation also contains sample code and makefiles for compiling and linking an application that can be used as the basis of the makefile for building a new application.

Archived

Archived

A

Data conversion techniques

When writing Java applications to invoke CICS programs, data conversion is a key issue because CICS, which runs on IBM System z processors, was developed in an EBCDIC world, whereas Java is based on Unicode, which is derived from ASCII character sets used in the UNIX and PC environments. In this appendix, we provide information about two different strategies you can use to perform data conversions:

- Conversion within Java
- Conversion by CICS: DFHCNV templates

Each of these options has its own advantages and drawbacks. We will discuss each of these in the following sections.

© Copyright IBM Corp. 2009. All rights reserved.

Conversion within Java

If you are an avid Java programmer, the thought of writing your 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 of benefit.

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 Unicode string. Each time you convert from a string to a byte array you convert each character from Unicode to a single byte character. 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 Example A-1 we specify the EBCDIC code page IBM037.

Example: A-1 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 shown in Example A-2.

Example: A-2 Correct encoding of the data on the string constructor

```
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 on return, data is converted from EBCDIC to Unicode. No data conversion is required within CICS. See Figure A-1 on page 183.

Archived

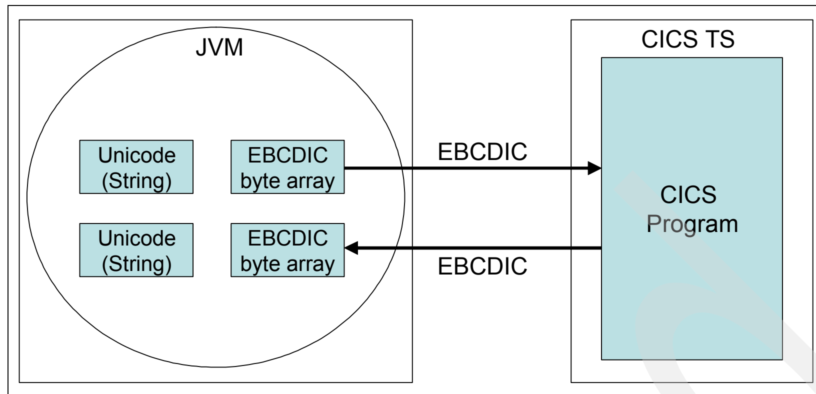


Figure A-1 Code page aware Java bean: EBCDIC input to CICS

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, as shown in Figure A-2. This is not as inefficient as it sounds. 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 cost of EBCDIC conversion can be transferred to CICS, potentially improving performance within the JVM.

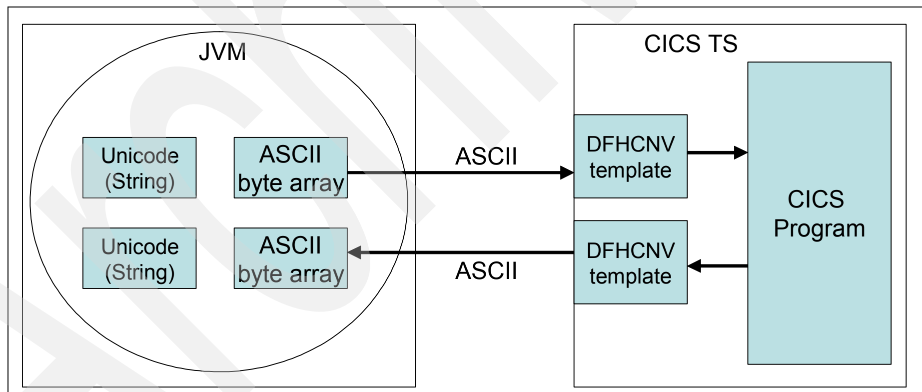


Figure A-2 Code page-aware Java bean ASCII input to CICS

In this case you would use an ASCII code page such as 8859_1 when creating the byte array as shown in Example A-3.

Example: A-3 Creating a byte array

```
byte abCommarea[] = new byte[27];
abCommarea = "abcd".getBytes("8859_1");
```

Having received the byte array back from CICS, convert to a string as shown in Example A-4.

Example: A-4 Convert to a string

```
String strCommarea = new String(abCommarea,"8859_1");
```

Attention: Specifying an encoding is important if you port your code from Intel to System z. I you do not specify an encoding, the default platform encoding is 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, you may think that this is a relatively simple affair, because both Java and z/OS store integers in big-endian format. Consider that all data passed to CICS must flow as a byte array. It is necessary to convert all integer values into a byte array before they can be passed to CICS. The code in Example A-5 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, the length is assumed to be 4 bytes, and 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.

Example: A-5 Method getByteData

```
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;
}
```

The logic in Example A-5 on page 184 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 Example A-6 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.

Example: A-6 Method getIntData()

```
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;
}
```

The logic in Example A-6 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. 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.

Archived

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.

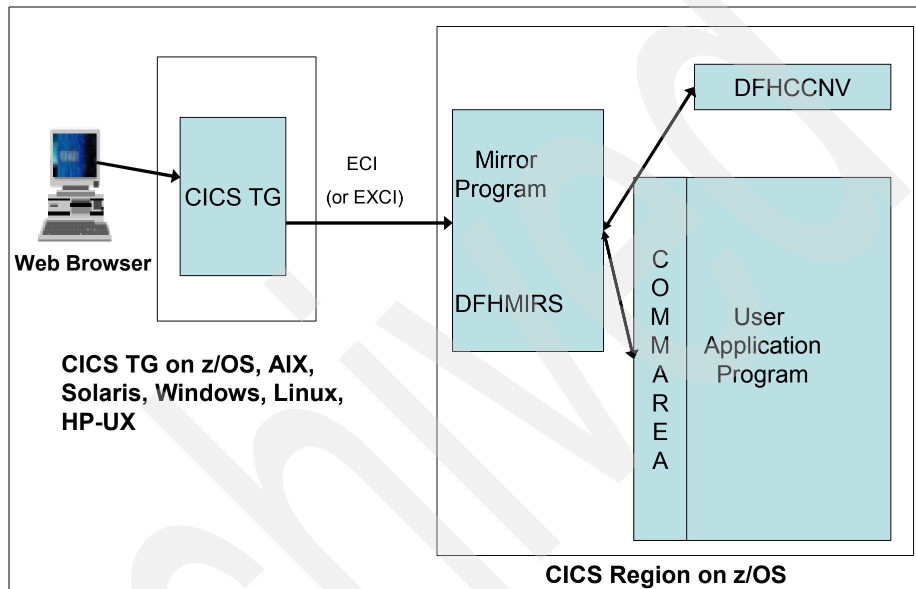


Figure A-3 CICS Transaction Gateway: ECI data conversion

Only if DFHCCNV finds a conversion template in the DFHCNV table that 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 Example A-7 on page 187.

Example: A-7 Sample DFHCNV character conversion template

```
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
```

- 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. The USEOEMCP parameter signifies that the code page currently in use by the platform 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 Example A-8 on page 188 details use of the CicsCpRequest object.

Archived

Example: A-8 Sample code for CicsCpRequest

```
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());
}
```

The logic in Example A-8 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 be used as the ASCII code page when creating an ASCII byte array to flow to CICS. This ASCII byte array is then converted to EBCDIC in CICS using a DFHCNV template.

The code in Example A-9 illustrates how to use the CicsCpRequest object to query the CTG code page and subsequently use this information to create an ASCII byte array to flow to CICS in an ECI request.

Example: A-9 Using the CicsCpRequest code page

```
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("CIC1IPIC", // CICS Server
            "null", // userid
            "null", // password
            "CUSTPROG", // 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());
}
```

The logic in Example A-9 on page 188 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. System z 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*, where 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 application to CICS.

Archived

DFHCNV supports the conversion of 2- and 4-byte numeric data from little-endian format to big-endian. This is achieved using DATATYP=NUMERIC, as shown in Example A-10. A DATATYP=BINARY entry signifies the data is already in big-endian format and does not need conversion. To convert 8-byte numeric values, packed decimal, or floating point values, it is necessary to use a different technique, such as the CICS user-replaceable conversion program DFHUCNV.

Example: A-10 Sample DFHCNV numeric conversion template

```
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
```

The template in Example A-10 is similar to the character template shown in Example A-7 on page 187, 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 is still converted from little-endian to big-endian because CICS overrides the TYPE-BINARY field. It assumes the data originated from a Windows platform and therefore treats it as though it were little-endian data.

Archived

B

Java Sample Code

Following are the four source files containing the main methods for the sample Java clients used in this Redbooks publication. They are:

- CustProgGetSingleBaseClasses.java
- CustProgGetSingleJ2C.java
- CustProgMultiRecBaseClasses.java
- CustProgMultiRecJ2C.java

A supporting source file,CustProgCommarea.java, is also included.

The following three source files for the Java Sample Client application described earlier in the Redbook are not printed out here, but can be found by following the instructions in Appendix E, “Additional material” on page 253:

- CustProgErrorInfo.java
- CustProgRecord.java
- CustProgRequestInfo.java

Archived

© Copyright IBM Corp. 2009. All rights reserved.

CustProgGetSingleBaseClasses

> **Note:** Highlighted lines are analyzed in detail in 4.2.1, “Sample COMMAREA-based Java client development” on page 67

Example: B-1 CustProgGetSingleBaseClasses.java

```
package com.ibm.itso.eci;

import java.io.IOException;
import java.io.UnsupportedEncodingException;
import com.ibm.ctg.client.*;

public class CustProgGetSingleBaseClasses {

   private static JavaGateway javaGatewayObject;


   private String theServerName = ""; // we can get this from the first command line
argument
   private String thePortNumber = "";  // we get this from the second command line
argument
   private int iPortNumber = 0;
   private String theConnectionURL = "";// we get this from the third command line
argument

   private String theCustomerId = "00000001";
   private String theFunctionName = "CUSTPROG";

   boolean debug = false;
   public static void main(String[] args) {
      System.out.println("---------- CustProgGetSingleBaseClasses ----------");
      new CustProgGetSingleBaseClasses(args);
   }

   public CustProgGetSingleBaseClasses(String[] args) {

      getCommandLineArguments(args);

      System.out.println("Will attempt to retrieve customer Id "+theCustomerId+" via
program "+theFunctionName);
      System.out.println("Server "+theServerName+" will be accessed at
"+theConnectionURL+" listening on port "+iPortNumber);
```

```
      ECIRequest eciRequestObject = null;

      // create a commarea object and populate it
      CustProgCommarea cpd = new CustProgCommarea();
      cpd.setRequestType("R");
      cpd.setCustomerId(theCustomerId);
      System.out.println("Getting information for customer: "+cpd.getCustomerId());

      try {
         javaGatewayObject = new JavaGateway(theConnectionURL, iPortNumber);
      } catch (IOException e1) {
         System.out.println("IOException trying to communicate to the specified CICS 
TG."+e1.getMessage());
         return;
      }

      byte[] abytCommarea = null;
      try {
         abytCommarea = cpd.getBytes("037");
      } catch (Throwable t) {
         System.out.println("Problem converting COMMAREA to codepage 037.  Cannot 
complete program. "+t.getMessage());
         return;
      }

      // Use the extended constructor to set the parameters on the ECIRequest
      // object
      eciRequestObject = new ECIRequest(ECIRequest.ECI_SYNC, // ECI call type
           theServerName, // CICS server
           null, // CICS userid
           null, // CICS password
           theFunctionName, // CICS program to be run
           null, // CICS transid to be run
           abytCommarea, // Byte array containing the COMMAREA
           cpd.length(), // COMMAREA length
           ECIRequest.ECI_NO_EXTEND, // ECI extend mode
           0); // ECI LUW token

      int iRc = 0;

      try {
         iRc = javaGatewayObject.flow(eciRequestObject);
      } catch (IOException e3) {
```

```
         System.out.println("IOException encountered trying to invoke the CUSTPROG 
program. Exception was:"+ e3.getMessage());
         return;
      }

      switch (eciRequestObject.getCicsRc()) {
      case ECIRequest.ECI_NO_ERROR:
         if (iRc == 0) {
            break;
         } else {
            System.out.println("\nError from Gateway ("+ 
eciRequestObject.getRcString() + ").");
            if (javaGatewayObject.isOpen() == true) {
               try { javaGatewayObject.close(); } catch (Throwable t2) { ; }
            }
            return;
         }
      case ECIRequest.ECI_ERR_SECURITY_ERROR:
         System.out.print("\n\nSecurity problem communicating with the CICS TG. ");
         System.out.println("\nYou are not authorised to run this transaction.");
         System.out.println("\nECI returned: "+ eciRequestObject.getCicsRcString());
         if (javaGatewayObject.isOpen() == true) {
            try { javaGatewayObject.close(); } catch (Throwable t2) { ; }
         }
         return;
      case ECIRequest.ECI_ERR_TRANSACTION_ABEND:
         System.out.println("\nTransaction ABEND indicator received.");
         System.out.println("\nECI returned: "+ eciRequestObject.getCicsRcString());
         System.out.println("Abend code was " + eciRequestObject.Abend_Code+ "\n");
         if (javaGatewayObject.isOpen() == true) {
            try { javaGatewayObject.close(); } catch (Throwable t2) { ; }
         }
         System.exit(1);
      default:
         System.out.println("\nProblem invoking Program.");
         System.out.println("\nECI returned: "+ eciRequestObject.getCicsRcString());
         System.out.println("Abend code was " + eciRequestObject.Abend_Code+ "\n");
         if (javaGatewayObject.isOpen() == true) {
            try { javaGatewayObject.close(); } catch (Throwable t2) { ; }
         }
         System.exit(1);
      }

      if (javaGatewayObject.isOpen() == true) {
         try { javaGatewayObject.close(); } catch (Throwable t1) { ; }
```

```
      }

      try {
         cpd.setBytes(abytCommarea, "037");
      } catch (UnsupportedEncodingException e) {
         System.out.println("Unsupported code page:037 found when trying to convert 
the returned COMMAREA to the local code page"+e.getMessage());
         e.printStackTrace();
      }
      if (cpd.getReturnCode().equals("00")) {
         System.out.println("Customer id: "+cpd.getCustomerId()+", name: 
"+cpd.getCustomerFirstName().trim()+" "+cpd.getCustomerLastName().trim()+", last 
updated: "+cpd.getCustomerLastUpdateDate());
      } else {
         System.out.println("The CICS program was accessed but a problem was 
indicated:  the program returned \""+cpd.getReturnComment().trim()+"\".");
      }
   }

   private void getCommandLineArguments(String[] args) {
      boolean gotArg = false;

      // parse the parameters:
      for (int i=0; i<args.length; i++) {
         if (i == 0) {theServerName = args[0]; continue; }
         if (i == 1) {
            thePortNumber = args[1];
            try {
               iPortNumber = Integer.parseInt(thePortNumber);
            } catch (NumberFormatException e) {
               System.out.println("*** ERROR *** - the second argument - the port 
number - must be numeric.");
               System.exit(1);
            }
            continue;
         }
         if (i == 2) {theConnectionURL = args[2]; continue; }

         gotArg=false;
         if (args[i].equalsIgnoreCase("DEBUG:TRUE")) { debug=true; gotArg=true; }
         if (args[i].equalsIgnoreCase("DEBUG:FALSE")) { debug=false; gotArg=true; }

         if (args[i].length() > 7) {
            if ((args[i].toUpperCase()).startsWith("CUSTID:")) {
               theCustomerId=args[i].substring(7);
```

```
               gotArg=true;
            }
         }

         if (!gotArg) {
            System.out.println("*** ERROR *** - \""+args[i]+"\" is an unknown
argument...\n");
            printUsageStatement();
            System.exit(1);
         }

      }
      if (theServerName.equals("") || thePortNumber.equals("") ||
theConnectionURL.equals("")) {
         printUsageStatement();
         System.exit(1);
      }

   }


   private void printUsageStatement() {
      System.out.println("This program can be used to invoke the CUSTPROG CICS-based
COBOL program");
      System.out.println("to retrieve a single record through the specified CICS TG.
Three positional");
      System.out.println("arguments are required.  The CICS TG Base Classes are used
in this program.");
      System.out.println(" ");
      System.out.println("Usage: CustProgGetSingleBaseClasses <ServerConnectionName>
<Port> <Gateway> <custid:xxxxxxxx>");
      System.out.println(" ");
      System.out.println("where <ServerConnectionName> is the CICS Server Connection
Name.");
      System.out.println("      <Port> is the CICS TG port configured using ctgcfg
tool.");
      System.out.println("      <Gateway> is the the system where Gateway daemon is
running.");
      System.out.println("      In <custid:xxxxxxxx>,  xxxxxxxx is the customer id
whose data you want to retrieve and the Customer Id length is 8 characters.");
      System.out.println("     ");
      System.out.println("Example: CustProgGetSingleBaseClasses CICWKS99 2006
tcp://wtsc59.itso.ibm.com custid:00000001");
      System.out.println("The above example will try to get the customer record for
Customer Id \"00000001\". ");
```

```
      System.out.println("     ");
      System.out.println(".");

   }
}
```

CustProgGetSingleJ2C

> **Note:** Highlighted lines are analyzed in detail in 4.2.2, “Sample COMMAREA-based JCA client development” on page 74.

Example: B-2   CustProgGetSingleJ2C.java

```
package com.ibm.itso.jca;

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

public class CustProgGetSingleJ2C {

   private ECIManagedConnectionFactory eciMgdCf = null;
   private ECIInteractionSpec is = null;

   // Settings:
   private boolean debug = false;

   private String theServerName = "";  // we get this from the first command line
argument
   private String thePortNumber = "";  // we get this from the second command line
argument
   private String theConnectionURL = "";// we get this from the third command line
argument

   private String theCustomerId = "00000001";
   private String theFunctionName = "CUSTPROG";
   private String theUserName = "";
   private String thePassword = "";
```

Archived

```
   private String theTPNName = "";
   private String theTranName = "";
   private int theExecuteTimeoutMilliseconds = 5000;
   private int theCommareaLength = 309;
   private int theReplyLength = 309;

   public static void main(String[] args) {
      System.out.println("---------- CustProgGetSingleJ2C ----------");
      new CustProgGetSingleJ2C(args);
   }

   public CustProgGetSingleJ2C(String[] args) {
      getCommandLineArguments(args);
        System.out.println("Will attempt to retrieve customer Id "+theCustomerId+"
via program "+theFunctionName);
        System.out.println("Server "+theServerName+" will be accessed at
"+theConnectionURL+" listening on port "+thePortNumber);
      invokeCustProg(args);
   }

    public void invokeCustProg(String[] args) {
        try {
            CustProgCommarea cpca = new CustProgCommarea();
            cpca.setRequestType("R");
            cpca.setCustomerId(theCustomerId);
            JavaStringRecord jsr = new JavaStringRecord();
            // jsr.setEncoding("IBM037");
            jsr.setText(cpca.toString());

            ConnectionFactory cf = createAConnectionFactory();
            Connection conn = cf.getConnection();
            is = getMyECIInteractionSpec();
            Interaction interaction = conn.createInteraction();
            interaction.execute(is, jsr, jsr);

            String returnedData = jsr.getText();
            cpca.setText(returnedData);
            if (cpca.getReturnCode().equals("00")) {
                System.out.println("Customer id: " + cpca.getCustomerId() + ", name:
" + cpca.getCustomerFirstName().trim() + " " + cpca.getCustomerLastName().trim()
                        + ", last updated: " + cpca.getCustomerLastUpdateDate());
            }
            else {
                System.out.println("The CICS program was accessed but " + " a problem
was indicated:  the program returned \"" + cpca.getReturnComment().trim()
```

```
                        + "\". Return code :: " + cpca.getReturnCode());
            }

            // shut it down
            interaction.close();
            conn.close();
        }
        catch (ResourceException e) {
            System.out.println("Unexpected ResourceException:" + e);
            e.printStackTrace();
        }

    }
private ConnectionFactory createAConnectionFactory()
                           throws ResourceException {
   if (eciMgdCf == null) {
       eciMgdCf = new ECIManagedConnectionFactory();
       eciMgdCf.setConnectionURL(theConnectionURL);
       eciMgdCf.setPortNumber(thePortNumber);
       eciMgdCf.setServerName(theServerName);
       eciMgdCf.setLogWriter(new java.io.PrintWriter(System.err));
       if (!thePassword.equals("")) {
           eciMgdCf.setPassword(thePassword);
        }
       if (!theUserName.equals("")) {
           eciMgdCf.setUserName(theUserName);
      }
      if (debug) {
           int ri = ECIManagedConnectionFactory.RAS_TRACE_INTERNAL;
           eciMgdCf.setTraceLevel(new Integer(ri));
        }
    }
   return (ConnectionFactory) eciMgdCf.createConnectionFactory();
 }

  private ECIInteractionSpec getMyECIInteractionSpec() throws ResourceException {
       if (is == null) {
           is = new ECIInteractionSpec();
           is.setCommareaLength(theCommareaLength);
           is.setExecuteTimeout(theExecuteTimeoutMilliseconds);
           is.setFunctionName(theFunctionName);
           is.setInteractionVerb(ECIInteractionSpec.SYNC_SEND_RECEIVE);
           is.setReplyLength(theReplyLength);
           if (!theTPNName.equals("")) {
               is.setTPNName(theTPNName);
```

```
            }
            if (!theTranName.equals("")) {
                is.setTranName(theTranName);
            }
        }
        return is;
    }


   private void getCommandLineArguments(String[] args) {
      boolean gotArg = false;

      // parse the parameters:
      for (int i=0; i<args.length; i++) {
         if (i == 0) {theServerName = args[0]; continue; }
         if (i == 1) {thePortNumber = args[1]; continue; }
         if (i == 2) {theConnectionURL = args[2]; continue; }

         gotArg=false;
         if (args[i].equalsIgnoreCase("DEBUG:TRUE")) { debug=true; gotArg=true; }
         if (args[i].equalsIgnoreCase("DEBUG:FALSE")) { debug=false; gotArg=true; }

         if (args[i].length() > 7) {
            if ((args[i].toUpperCase()).startsWith("CUSTID:")) {
               theCustomerId=args[i].substring(7);
               gotArg=true;
            }
         }

         if (!gotArg) {
            System.out.println("*** ERROR *** - \""+args[i]+"\" is an unknown
argument...\n");
            printUsageStatement();
            System.exit(1);
         }

      }
      if (theServerName.equals("") || thePortNumber.equals("") ||
theConnectionURL.equals("")) {
         printUsageStatement();
         System.exit(1);
      }

   }
```

```
   private void printUsageStatement() {
      System.out.println("This program can be used to invoke the CUSTPROG CICS-based
COBOL program");
      System.out.println("to retrieve a single record through the specified CICS TG.
Three positional");
      System.out.println("arguments are required.  The J2C classes are used in this
program.");
      System.out.println(" ");
      System.out.println("Usage: CustProgGetSingleJ2C <CICS_Applid> <CICS_TG_Port>
<CTGLocationAndMode>");
      System.out.println("       ");
      System.out.println("    Example: CustProg_GetSingle_J2C CICWKS99 2006
tcp://localhost");
      System.out.println("       ");
      System.out.println("  custid:xxxxxxxx  - The program will try to get Customer
Id \"00000001\", ");
      System.out.println("     but you can override this by adding custid:xxxxxxxx
after the 3 positional");
      System.out.println("     parameters.  The Customer Id length is 8
characters.");
      System.out.println("     (where xxxxxxxx is the customer id whoose data you
want to retrieve.)");

   }


}
```

CustProgMultiRecBaseClasses

Note: Highlighted lines are analyzed in detail in 4.3.1, “Sample channel-based Java client development” on page 83.

Example: B-3 CustProgMultiRecBaseClasses.java

```
package com.ibm.itso.eci;

import java.io.IOException;
import java.io.UnsupportedEncodingException;
//import java.util.Iterator;

import com.ibm.ctg.client.*;
import com.ibm.ctg.client.exceptions.ChannelException;
import com.ibm.ctg.client.exceptions.ContainerException;
import com.ibm.ctg.client.exceptions.ContainerNotFoundException;

public class CustProgMultiRecBaseClasses {

   // ---------------------------------------------------------------
   public static String product_name = "CustProgMultiRecBaseClasses; Sample Program
for CICS Transaction Gateway";

   public static String program_version = "V0.0.1";

   public static String copyright = "Licensed Materials - Property of IBM\n" +
                            " \n" +
                            "© Copyright IBM Corporation 2008\n" +
                            "All Rights Reserved\n" +
                            " \n" +
                            "US Government Users Restricted Rights - Use, \n" +
                            "duplication, or disclosure restricted by GSA ADP \n" +
                            "Schedule Contract with IBM Corporation";

   //  ---------------------------------------------------------------

   private static JavaGateway javaGatewayObject;
   private static String CHANNEL_NAME = "MYCHANNEL";
   private static String REQUEST_INFO_CONTAINER = "BROWSE-INFO";
   private static String ERROR_CONTAINER = "ERROR";
   private static String NUM_RECS_CONTAINER = "NUM-RECORDS";
```


Archived

```
   private boolean debug = false;

   // request settings
   private int maxRecordsToReturn = 5;
   private String direction = "F";
   private String startingKey = "00000000";
   private int startingKeyLength = 8;

   // location settings:
   private String theConnectionURL = "";
   private String thePortNumber = "";
   private String theServerName = "";
   private String theFunctionName = "CUSTBRWS";

   // Main method
   public static void main(String[] args) {
      System.out.println("---------- CustProgMultiRecBaseClasses - Using Channels and 
Containers ----------");
      new CustProgMultiRecBaseClasses(args);
   }

   public CustProgMultiRecBaseClasses(String[] args) {

      ECIRequest eciRequestObject = null;

      getCommandLineArguments(args);

      System.out.println("Using the CICS Transaction Gateway Base Classes.  Invoking 
program "+theFunctionName+".");
      System.out.println("The server "+theServerName+" will be accessed at 
"+theConnectionURL+" listening on port "+thePortNumber+".");

      try {
         javaGatewayObject = new JavaGateway(theConnectionURL, 
Integer.parseInt(thePortNumber));
      } catch (IOException e1) {
         System.out.println("IOException trying to communicate to the CICS TG at 
"+theConnectionURL+", Port "+thePortNumber+":"+e1.getMessage());
         return;
      }

      //  the CustProg_RequestInfo object corresponds to the container that is sent 
to
      //       describe what should be returned from the CICS TS program
      CustProgRequestInfo cri = new CustProgRequestInfo();
```

```
      cri.setMaxRecords(maxRecordsToReturn);
      cri.setDirection(direction);
      cri.setStartKey(startingKey);
      cri.setStartKeyLength(startingKeyLength);
      if (startingKeyLength == 0) {
         System.out.println("Requesting a maximum of "+maxRecordsToReturn+" records 
starting with the first customer id on the file.");
      } else if (startingKeyLength > 7){
         System.out.println("Requesting a maximum of "+maxRecordsToReturn+" records 
starting with customer ids that begin with \""+cri.getStartKey()+"\".");
      } else {
         System.out.println("Requesting a maximum of "+maxRecordsToReturn+" records 
starting with customer ids that begin with \""+cri.getStartKey().substring(0, 
startingKeyLength)+"\".");
      }

      if (direction.equalsIgnoreCase("b")) {
         System.out.println("The sequence will be descending.");
      } else {
         System.out.println("The sequence will be ascending.");
      }

      //  create a channel
        Channel reqChannel = null;
      try {
         reqChannel = new Channel(CHANNEL_NAME);
      } catch (ChannelException e1) {
         System.out.println("ChannelException while trying to create a channel: 
"+e1.getMessage());
         System.out.println("Terminating!");
         System.exit(1);
      }

      // Place the request data in a REQUEST_INFO_CONTAINER in CHANNEL_NAME
        try {
         reqChannel.createContainer(REQUEST_INFO_CONTAINER, cri.toString());
      } catch (UnsupportedEncodingException e1) {
         System.out.println("UnsupportedEncodingException while trying to add the 
REQUEST_INFO_CONTAINER to the channel: "+e1.getMessage());
         System.out.println("Terminating!");
         System.exit(1);
      } catch (ContainerException e1) {
         System.out.println("ContainerException while trying to add the 
REQUEST_INFO_CONTAINER to the channel: "+e1.getMessage());
         System.out.println("Terminating!");
```


Archived

```
         System.exit(1);
      }

        // specify the details of the where and how you want to go
        eciRequestObject = new ECIRequest(
                ECIRequest.ECI_SYNC,        //ECI call type
                theServerName,//CICS server
                null,                       //CICS username
                null,                       //CICS password
                theFunctionName,               //Program to run
                null,                       //Transaction to run
                reqChannel,                 //Channel
                ECIRequest.ECI_NO_EXTEND,   //ECI extend mode
                0                           //ECI LUW token
             );

        // make the request
        int iRc = 0;
      try {
         iRc = javaGatewayObject.flow(eciRequestObject);
      } catch (IOException e3) {
         System.out.println("IOException encountered trying to invoke the
"+theFunctionName+" program. Exception was:"+e3.getMessage());
         return;
      }

      //  if we had an error - try to tell the user what happened
        switch (eciRequestObject.getCicsRc()) {
        case ECIRequest.ECI_NO_ERROR:
        if (iRc == 0) {
         break;
        } else {
         System.out.println("\nError from Gateway
("+eciRequestObject.getRcString()+").");
         if (javaGatewayObject.isOpen() == true) { try { javaGatewayObject.close(); }
catch (Throwable t2) { ; } }
               return;
           }
         case ECIRequest.ECI_ERR_SECURITY_ERROR:
            System.out.print("\n\nSecurity problem communicating with the CICS TG. ");
            return;
         case ECIRequest.ECI_ERR_NO_CICS:
            System.out.println("\n\nError no CICS with the specified name.");
            return;
         case ECIRequest.ECI_ERR_TRANSACTION_ABEND:
```

```
            System.out.println("\nA transaction ABEND was reported.");
            System.out.println("\nECI returned: " +
eciRequestObject.getCicsRcString());
            System.out.println("Abend code was " + eciRequestObject.Abend_Code +
"\n");
            if (javaGatewayObject.isOpen() == true) { try { javaGatewayObject.close();
} catch (Throwable t2) { ; } }
            System.exit(1);
         default:
            System.out.println("\nA transaction ABEND was reported.");
            System.out.println("\nECI returned: " +
eciRequestObject.getCicsRcString());
            System.out.println("Abend code was " + eciRequestObject.Abend_Code +
"\n");
            if (javaGatewayObject.isOpen() == true) { try { javaGatewayObject.close();
} catch (Throwable t2) { ; } }
            System.exit(1);
      }


        // all is fine with the communication, close the gateway object
        if (javaGatewayObject.isOpen() == true) { try { javaGatewayObject.close(); }
catch (Throwable t1) { ; } }

        // if the call didn't get a channel back, tell someone and quit
        if ( ! eciRequestObject.hasChannel()) {
        System.out.println("\nThe call to "+theFunctionName+" did not return a
channel, terminating!");
         System.exit(1);
        }

        // we have a returned channel - lets start processing it
        Channel respChan = eciRequestObject.getChannel();

        // Display the returned content
        Container cont = null;

        // check for errors returned by the CICS program
        try {
         cont = respChan.getContainer(ERROR_CONTAINER);
         CustProgErrorInfo cei = null;
         if (cont.getType() == Container.ContainerType.BIT) {
            cei = new CustProgErrorInfo(cont.getBITData(), "037");  // Java converts
from EBCDIC
         } else {
```

```
            cei = new CustProgErrorInfo(cont.getCHARData());
         }
         System.out.println("Response from "+theFunctionName+" was 
"+cei.getRequestReturnCode()+
               " - "+cei.getRequestReturnComment());
         System.exit(1);
      } catch (ContainerNotFoundException e) {
         // this is the 'normal' case - don't do anything
      } catch (UnsupportedEncodingException e) {
         System.out.println("UnsupportedEncodingException while trying to get the 
"+ERROR_CONTAINER+" container: "+e.getMessage());
         System.out.println("Terminating!");
         System.exit(1);
      } catch (ContainerException e) {
         System.out.println("ContainerException while trying to get the 
"+ERROR_CONTAINER+" container: "+e.getMessage());
         System.out.println("Terminating!");
         System.exit(1);
      }

      // get the number of data records returned
      int numRecsReturned = 0;
      try {
         cont = respChan.getContainer(NUM_RECS_CONTAINER);
         if (cont.getType() == Container.ContainerType.BIT) {
            System.out.println("The NUM-RECORDS container is a BIT container");
            byte[] bNum = cont.getBITData();
            try {
               String tempStr = new String(bNum, "037");
               numRecsReturned = Integer.parseInt(tempStr);
            } catch (UnsupportedEncodingException e) {
               System.out.println("UnsupportedEncodingException while converting 
NUM_RECS byte array to a String"+e.getMessage());
               System.out.println("Terminating!");
               System.exit(1);
            }
         } else {
            System.out.println("The NUM-RECORDS container is a CHAR container");
            System.out.println("Value of NUM-RECORDS is:"+cont.getCHARData());
            numRecsReturned = Integer.parseInt(cont.getCHARData());
         }
         System.out.println("The number of records returned was: "+numRecsReturned);
      } catch (NumberFormatException e) {
         System.out.println("NumberFormatException while trying to turn the number of 
returned records from a String to an int: "+e.getMessage());
```

```
      } catch (ContainerNotFoundException e) {
         System.out.println("ContainerNotFoundException while trying to get the
"+NUM_RECS_CONTAINER+" container: "+e.getMessage());
      } catch (UnsupportedEncodingException e) {
         System.out.println("UnsupportedEncodingException while trying to get the
"+NUM_RECS_CONTAINER+" container: "+e.getMessage());
      } catch (ContainerException e) {
         System.out.println("ContainerException while trying to get the
"+NUM_RECS_CONTAINER+" container: "+e.getMessage());
      }


      String strContNum = "";
      for (int i = 0; i < numRecsReturned; i++) {
         strContNum = ("0000"+(i+1)).substring(("0000"+(i+1)).length()-4);
         try {
            cont = respChan.getContainer(strContNum);
            CustProgRecord cpr = null;
            if (cont.getType() == Container.ContainerType.BIT) {
               cpr = new CustProgRecord(cont.getBITData(), "037");
            } else {
               cpr = new CustProgRecord(cont.getCHARData());
            }
            System.out.println("Customer id: "+cpr.getCustomerId()+", name:
"+cpr.getCustomerFirstName().trim()+" "+cpr.getCustomerLastName().trim()+", last
updated: "+cpr.getCustomerLastUpdateDate());
         } catch (ContainerNotFoundException e) {
            System.out.println("ContainerNotFoundException while requesting container
"+strContNum+":"+e.getMessage());
         } catch (UnsupportedEncodingException e) {
            System.out.println("UnsupportedEncodingException while converting BIT
container "+strContNum+" to a String:"+e.getMessage());
         } catch (ContainerException e) {
            System.out.println("ContainerException while requesting container
"+strContNum+":"+e.getMessage());
         }
      }
   }

   private void getCommandLineArguments(String[] args) {
      boolean gotArg = false;

      // parse the parameters:
      for (int i=0; i<args.length; i++) {
         if (i == 0) {theServerName = args[0]; continue; }
```

```
         if (i == 1) {thePortNumber = args[1]; continue; }
         if (i == 2) {theConnectionURL = args[2]; continue; }

         gotArg=false;
         if (args[i].equalsIgnoreCase("DEBUG:TRUE")) { debug=true; gotArg=true; }
         if (args[i].equalsIgnoreCase("DEBUG:FALSE")) { debug=false; gotArg=true; }

         if (args[i].equalsIgnoreCase("DIRECTION:B")) { direction="B"; gotArg=true; }

         if (args[i].length() > 8) {
            if ((args[i].toUpperCase()).startsWith("STARTID:")) {
               startingKey=args[i].substring(8);
               gotArg=true;
            }
         }

         if (args[i].length() > 7) {
            if ((args[i].toUpperCase()).startsWith("RETURN:")) {
               String s =args[i].substring(7);
               try {
                  maxRecordsToReturn=Integer.parseInt(s);
               } catch (NumberFormatException e) {
                  System.out.println("*** ERROR *** - the number of records to be 
returned must be numeric.");
                  System.exit(1);
               }
               gotArg=true;
            }
         }

         if (args[i].length() > 7) {
            if ((args[i].toUpperCase()).startsWith("KEYLEN:")) {
               String s =args[i].substring(7);
               try {
                  startingKeyLength=Integer.parseInt(s);
               } catch (NumberFormatException e) {
                  System.out.println("*** ERROR *** - the keylength for a GENERIC 
browse must be numeric.");
                  System.exit(1);
               }
               gotArg=true;
            }
         }

         if (!gotArg) {
```

Archived

```
            System.out.println("*** ERROR *** - \""+args[i]+"\" is an unknown
argument...\n");
            printUsageStatement();
            System.exit(1);
         }

      }
      if (theServerName.equals("") || thePortNumber.equals("") ||
theConnectionURL.equals("")) {
         printUsageStatement();
         System.exit(1);
      }

   }


   private void printUsageStatement() {
      System.out.println("This program can be used to invoke the CUSTBRWS CICS-based
COBOL program");
      System.out.println("to retrieve multiple records through the specified CICS TG.
Three positional");
      System.out.println("arguments are required.  The CICS TG Base Classes are used
in this program.");
      System.out.println(" ");
      System.out.println("Usage: CustProgMultiRecBaseClasses <ServerConnectionName>
<Port> <Gateway>");
      System.out.println("        ");
      System.out.println("    Example: CustProgMultiRecBaseClasses CICWKS99 2006
tcp://localhost");
      System.out.println("        ");
      System.out.println("   startid:xxxxxxxx  - The program will try to get account
information starting");
      System.out.println("     with customer id \"00000000\".  You can override this
by adding ");
      System.out.println("     startid:xxxxxxxx after the 3 positional parameters,
where xxxxxxxx is the");
      System.out.println("     first customer id you would like returned.  The
Customer Id length");
      System.out.println("     is 8 characters.  If less than 8 characters are
specified, they will be ");
      System.out.println("     padded out to 8 with blanks");
      System.out.println("   return:nn  - The program will try to return 5 customer
accounts, ");
      System.out.println("     but you can override this by adding return:nn after
the 3 positional");
```

```
      System.out.println("     parameters.  The max number of records that can be
returned is currently 200.");
      System.out.println("     (where nn is the number of accounts you want
returned.)");
      System.out.println("   direction:B  - The program will try to browse forward on
the VSAM file, ");
      System.out.println("     but you can override this by adding direction:B after
the 3 positional");
      System.out.println("     parameters.  (B is for Backword)");
      System.out.println("   keylen:n  - The program will assume that the keylength
is 8, ");
      System.out.println("     but you request the browse start with the first record
on the file");
      System.out.println("     by specifying keylen:0 after the 3 positional
parameters.  You can also ");
      System.out.println("     specify a GENERIC browse by specifying keylen:n, where
n is the number of ");
      System.out.println("     characters in the generic key.  If greater than 8 is
specified, 8 is assumed.");
      System.out.println("     Specifying keylen:0 takes precidence over
startid:xxxxxxxx.");

   }

}
```

Archived

CustProgMultiRecJ2C

> **Note:** Highlighted lines are analyzed in detail in 4.3.2, "Sample channel-based JCA client development" on page 93

Example: B-4 CustProgMultiRecJ2C.java

```
package com.ibm.itso.jca;

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

public class CustProgMultiRecJ2C {

   // ---------------------------------------------------------------
   public static String product_name = "CustProg_MultiRec_J2C; Sample Program for
CICS Transaction Gateway";

   public static String program_version = "V0.0.1";

   public static String copyright = "Licensed Materials - Property of IBM\n" +
                            " \n" +
                            "© Copyright IBM Corporation 2008\n" +
                            "All Rights Reserved\n" +
                            " \n" +
                            "US Government Users Restricted Rights - Use, \n" +
                            "duplication, or disclosure restricted by GSA ADP \n" +
                            "Schedule Contract with IBM Corporation";

   //  ---------------------------------------------------------------

   // Channel and Container Names
   private static String CHANNEL_NAME = "MYCHANNEL";
   private static String REQUEST_INFO_CONTAINER = "BROWSE-INFO";
   private static String ERROR_CONTAINER = "ERROR";
   private static String NUM_RECS_CONTAINER = "NUM-RECORDS";
```

```
   private boolean debug=false;

   // Settings for the request to the CICS-based program
   private int maxRecordsToReturn = 5;
   private String direction = "F";
   private String startingKey = "00000000";
   private int startingKeyLength = 8;   // zero will start with the first record on
the file


   // Connection and Interaction related settings
   private String theConnectionURL = "";
   private String thePortNumber = "";
   private String theServerName = "";
   private String theUserName = "";
   private String thePassword = "";
   private String theFunctionName = "CUSTBRWS";
   private String theTPNName = "";
   private String theTranName = "";
   private int theExecuteTimeoutMilliseconds = 5000;

   // Factory and Specification objects
   private ECIManagedConnectionFactory eciMgdCf = null;
   private ECIInteractionSpec is = null;

   // ------------------------------------------------------------------------
   public static void main(String[] args) {
      System.out.println("---------- CustProgMultiRecJ2C - Using Channels and
Containers ----------");
      new CustProgMultiRecJ2C(args);
   }

   public CustProgMultiRecJ2C(String[] args) {
        getCommandLineArguments(args);
        invokeCustBrws(args);
    }

   private void invokeCustBrws(String[] args) {
        Object containerObj = null;
        System.out.println("Using the CICS Transaction Gateway J2C Classes.  Invoking
program " + theFunctionName + ".");
        System.out.println("The server " + theServerName + " will be accessed at " +
theConnectionURL + " listening on port " + thePortNumber + ".");

        try {
```

```
            ECIChannelRecord myChannel = new ECIChannelRecord(CHANNEL_NAME);
            CustProgRequestInfo cri = new CustProgRequestInfo();
            cri.setMaxRecords(maxRecordsToReturn);
            cri.setDirection(direction);
            cri.setStartKey(startingKey);
            cri.setStartKeyLength(startingKeyLength);
            if (startingKeyLength == 0) {
                System.out.println("Requesting a maximum of " + maxRecordsToReturn +
" records starting with the first customer id on the file.");
            }
            else if (startingKeyLength > 7) {
                System.out.println("Requesting a maximum of " + maxRecordsToReturn +
" records starting with customer ids that begin with \"" + cri.getStartKey() +
"\".");
            }
            else {
                System.out.println("Requesting a maximum of " + maxRecordsToReturn +
" records starting with customer ids that begin with \""
                        + cri.getStartKey().substring(0, startingKeyLength) + "\".");
            }
            if (direction.equalsIgnoreCase("b")) {
                System.out.println("The sequence will be descending.");
            }
            else {
                System.out.println("The sequence will be ascending.");
            }
            myChannel.put(REQUEST_INFO_CONTAINER, cri.toString());

            ConnectionFactory cf = createAConnectionFactory();
            Connection conn = cf.getConnection();
            Interaction interaction = conn.createInteraction();
            ECIInteractionSpec is = getMyECIInteractionSpec();
            interaction.execute(is, myChannel, myChannel);

            // we returned, display the returned content
            // CustProg_ErrorInfo corresponds to the returned ERROR container
            if (myChannel.containsKey(ERROR_CONTAINER)) {
                CustProgErrorInfo cei = null;
                containerObj = myChannel.get(ERROR_CONTAINER);
                if (containerObj instanceof byte[]) {
                    try {
                        cei = new CustProgErrorInfo((byte[]) containerObj, "037");
                    }
                    catch (UnsupportedEncodingException e) {
```

```
                        System.out.println("UnsupportedEncodingException " +
"converting byte array to specified code page:" + e.getMessage());
                        System.out.println("Terminating!");
                        System.exit(1);
                    }
                }
                else if (containerObj instanceof String) {
                    cei = new CustProgErrorInfo((String) containerObj);
                }
                System.out.println("Response from " + theFunctionName + " was " +
cei.getRequestReturnCode() + " - " + cei.getRequestReturnComment());
                System.exit(1);
            }
            else {
                int numRecsReturned = 0;
                containerObj = myChannel.get(NUM_RECS_CONTAINER);
                if (containerObj instanceof byte[]) {
                    System.out.println("The object returned from the " +
NUM_RECS_CONTAINER + " was a byte[]");
                    byte[] bNum = (byte[]) containerObj;
                    try {
                        String tempStr = new String(bNum, "037");
                        numRecsReturned = Integer.parseInt(tempStr);
                    }
                    catch (UnsupportedEncodingException e) {
                        System.out.println("UnsupportedEncodingException" + " while
converting NUM_RECS byte array to a String" + e.getMessage());
                        System.out.println("Terminating!");
                        System.exit(1);
                    }
                }
                else if (containerObj instanceof String) {
                    System.out.println("The object returned from the " +
NUM_RECS_CONTAINER + " was a String");
                    numRecsReturned = Integer.parseInt((String) containerObj);
                }
                else {
                    System.out.println("Unknown object type returned from the " +
NUM_RECS_CONTAINER + ".");
                    System.out.println("Terminating!");
                    System.exit(1);
                }
                System.out.println("The number of records returned was: " +
numRecsReturned);
```

Archived

```
                // Display the information returned from the CICS-based program
                // the CICS program returns a container for each record returned
                String strContNum = "";
                for (int i = 0; i < numRecsReturned; i++) {
                    strContNum = ("0000" + (i + 1)).substring(("0000" + (i +
1)).length() - 4);
                    try {
                        containerObj = myChannel.get(strContNum);
                        CustProgRecord cpr = null;
                        if (containerObj instanceof byte[]) {
                            cpr = new CustProgRecord((byte[]) containerObj, "037");
                        }
                        else {
                            cpr = new CustProgRecord((String) containerObj);
                        }
                        System.out.println("Customer id: " + cpr.getCustomerId() + ",
name: " + cpr.getCustomerFirstName().trim() + " " + cpr.getCustomerLastName().trim()
                                + ", last updated: " +
cpr.getCustomerLastUpdateDate());
                    }
                    catch (UnsupportedEncodingException e) {
                        System.out.println("UnsupportedEncodingException while
converting BIT container " + strContNum + " to a String:" + e.getMessage());
                    }
                }
            }

            // shut it down
            interaction.close();
            conn.close();
        }
        catch (ResourceException e) {
            System.out.println("Unexpected ResourceException:" + e);
            e.printStackTrace();
        }

    }
   private ConnectionFactory createAConnectionFactory() throws ResourceException {
      if (eciMgdCf == null) {
         eciMgdCf = new ECIManagedConnectionFactory();
         eciMgdCf.setConnectionURL(theConnectionURL);
         eciMgdCf.setPortNumber(thePortNumber);
         eciMgdCf.setServerName(theServerName);
         eciMgdCf.setLogWriter(new java.io.PrintWriter(System.err));
         if ( ! thePassword.equals("")) { eciMgdCf.setPassword(thePassword); }
```

```
        if ( ! theUserName.equals("")) { eciMgdCf.setUserName(theUserName); }
        if (debug) {
           eciMgdCf.setTraceLevel(new
Integer(ECIManagedConnectionFactory.RAS_TRACE_INTERNAL));
        }
     }
     return (ConnectionFactory)eciMgdCf.createConnectionFactory();
  }

  private ECIInteractionSpec getMyECIInteractionSpec() throws ResourceException {
     if (is == null) {
        is = new ECIInteractionSpec();
        is.setExecuteTimeout(theExecuteTimeoutMilliseconds);
        is.setFunctionName(theFunctionName);
        is.setInteractionVerb(ECIInteractionSpec.SYNC_SEND_RECEIVE);
        is.setTPNName(theTPNName);
        is.setTranName(theTranName);
     }
     return is;
  }

  private void getCommandLineArguments(String[] args) {
     boolean gotArg = false;

     // parse the parameters:
     for (int i=0; i<args.length; i++) {
        if (i == 0) {theServerName = args[0]; continue; }
        if (i == 1) {thePortNumber = args[1]; continue; }
        if (i == 2) {theConnectionURL = args[2]; continue; }

        gotArg=false;
        if (args[i].equalsIgnoreCase("DEBUG:TRUE")) { debug=true; gotArg=true; }
        if (args[i].equalsIgnoreCase("DEBUG:FALSE")) { debug=false; gotArg=true; }

        if (args[i].equalsIgnoreCase("DIRECTION:B")) { direction="B"; gotArg=true; }

        if (args[i].length() > 8) {
           if ((args[i].toUpperCase()).startsWith("STARTID:")) {
              startingKey=args[i].substring(8);
              gotArg=true;
           }
        }

        if (args[i].length() > 7) {
           if ((args[i].toUpperCase()).startsWith("RETURN:")) {
```

```
               String s =args[i].substring(7);
               try {
                  maxRecordsToReturn=Integer.parseInt(s);
               } catch (NumberFormatException e) {
                  System.out.println("*** ERROR *** - the number of records to be
returned must be numeric.");
                  System.exit(1);
               }
               gotArg=true;
            }
         }

         if (args[i].length() > 7) {
            if ((args[i].toUpperCase()).startsWith("KEYLEN:")) {
               String s =args[i].substring(7);
               try {
                  startingKeyLength=Integer.parseInt(s);
               } catch (NumberFormatException e) {
                  System.out.println("*** ERROR *** - the keylength for a GENERIC
browse must be numeric.");
                  System.exit(1);
               }
               gotArg=true;
            }
         }

         if (!gotArg) {
            System.out.println("*** ERROR *** - \""+args[i]+"\" is an unknown
argument...\n");
            printUsageStatement();
            System.exit(1);
         }

      }
      if (theServerName.equals("") || thePortNumber.equals("") ||
theConnectionURL.equals("")) {
         printUsageStatement();
         System.exit(1);
      }

   }


   private void printUsageStatement() {
```


Archived

```
      System.out.println("This program can be used to invoke the CUSTBRWS CICS-based 
COBOL program");
      System.out.println("to retrieve multiple records through the specified CICS TG. 
Three positional");
      System.out.println("arguments are required.  The CICS TG Base Classes are used 
in this program.");
      System.out.println(" ");
      System.out.println("Usage: CustProg_MultiRec_J2C <CICS_Applid> <CICS_TG_Port> 
<CTGLocationAndMode>");
      System.out.println("       ");
      System.out.println("    Example: CustProg_MultiRec_J2C CICWKS99 2006 
tcp://localhost");
      System.out.println("       ");
      System.out.println("   startid:xxxxxxxx  - The program will try to get account 
information starting");
      System.out.println("     with customer id \"00000000\".  You can override this 
by adding ");
      System.out.println("     startid:xxxxxxxx after the 3 positional parameters, 
where xxxxxxxx is the");
      System.out.println("     first customer id you would like returned.  The 
Customer Id length");
      System.out.println("     is 8 characters.  If less than 8 characters are 
specified, they will be ");
      System.out.println("     padded out to 8 with blanks");
      System.out.println("   return:nn  - The program will try to return 5 customer 
accounts, ");
      System.out.println("     but you can override this by adding return:nn after 
the 3 positional");
      System.out.println("     parameters.  The max number of records that can be 
returned is currently 200.");
      System.out.println("     (where nn is the number of accounts you want 
returned.)");
      System.out.println("   direction:B  - The program will try to browse forward on 
the VSAM file, ");
      System.out.println("     but you can override this by adding direction:B after 
the 3 positional");
      System.out.println("     parameters.  (B is for Backword)");
      System.out.println("   keylen:n  - The program will assume that the keylength 
is 8, ");
      System.out.println("     but you request the browse start with the first record 
on the file");
      System.out.println("     by specifying keylen:0 after the 3 positional 
parameters.  You can also ");
      System.out.println("     specify a GENERIC browse by specifying keylen:n, where 
n is the number of ");
```

```
      System.out.println("     characters in the generic key.  If greater than 8 is
specified, 8 is assumed.");
      System.out.println("     Specifying keylen:0 takes precidence over
startid:xxxxxxxx.");

   }


}
```

B.1 CustProgCommarea

Example: B-5 CustProgCommarea.java

```
package com.ibm.itso.model;

import java.io.UnsupportedEncodingException;

public class CustProgCommarea {

   private static String BLANKS50 = "
";
   private static String BLANKS30 = "                              ";
   private static String BLANKS20 = "                    ";
   private static String BLANKS8  = "        ";

   private String requestType = " ";
   private String returnCode = "  ";
   private String customerId = BLANKS8;
   private String customerLastName = BLANKS20;
   private String customerFirstName = BLANKS20;
   private String customerCompany = BLANKS30;
   private String customerAddr1 = BLANKS30;
   private String customerAddr2 = BLANKS30;
   private String customerCity = BLANKS20;
   private String customerState = BLANKS20;
   private String customerCountry = BLANKS30;
   private String customerMailCode = BLANKS20;
   private String customerPhone = BLANKS20;
   private String customerLastUpdateDate = BLANKS8;
   private String returnComment = BLANKS50;

   private int size = requestType.length()+returnCode.length()+customerId.length()+
```


Archived

```
customerLastName.length()+customerFirstName.length()+customerCompany.length()+
        customerAddr1.length()+customerAddr2.length()+customerCity.length()+
        customerState.length()+customerCountry.length()+customerMailCode.length()+

customerPhone.length()+customerLastUpdateDate.length()+returnComment.length();


   public CustProgCommarea() { }

   public CustProgCommarea(String totData) {
      setText(totData);
   }

   public void setBytes(byte[] b, String enc) throws UnsupportedEncodingException {
      //setText(new String(b, enc));
      setText(new String(b));

   }
   public byte[] getBytes(String enc) throws UnsupportedEncodingException {
      //return toString().getBytes("037");
      return toString().getBytes();

   }

   public void setText(String totData) {
      setRequestType(totData.substring(0,1));// 1
      setReturnCode(totData.substring(1,3));// 2
      setCustomerId(totData.substring(3,11));// 8
      setCustomerLastName(totData.substring(11,31));// 20
      setCustomerFirstName(totData.substring(31,51));// 20
      setCustomerCompany(totData.substring(51,81));// 30
      setCustomerAddr1(totData.substring(81,111));// 30
      setCustomerAddr2(totData.substring(111,141));// 30
      setCustomerCity(totData.substring(141,161));// 20
      setCustomerState(totData.substring(161,181));// 20
      setCustomerCountry(totData.substring(181,211));// 30
      setCustomerMailCode(totData.substring(211,231));// 20
      setCustomerPhone(totData.substring(231,251));// 20
      setCustomerLastUpdateDate(totData.substring(251,259));// 8
      setReturnComment(totData.substring(259,309));// 50
   }

   public String toString() {
      return requestType+returnCode+customerId+
```

```
      customerLastName+customerFirstName+customerCompany+
      customerAddr1+customerAddr2+customerCity+
      customerState+customerCountry+customerMailCode+
      customerPhone+customerLastUpdateDate+returnComment;
   }

   public int length() { return size; }

   public String getCustomerAddr1() { return customerAddr1; }
   public void setCustomerAddr1(String theCustomerAddr1) {
      if (theCustomerAddr1 == null) {
         customerAddr1 = BLANKS30;
      } else {
         customerAddr1 = (theCustomerAddr1+BLANKS30).substring(0, 30);
      }
   }

   public String getCustomerAddr2() { return customerAddr2; }
   public void setCustomerAddr2(String theCustomerAddr2) {
      if (theCustomerAddr2 == null) {
         customerAddr2 = BLANKS30;
      } else {
         customerAddr2 = (theCustomerAddr2+BLANKS30).substring(0, 30);
      }
   }

   public String getCustomerCity() { return customerCity; }
   public void setCustomerCity(String theCustomerCity) {
      if (theCustomerCity == null) {
         customerCity = BLANKS20;
      } else {
         customerCity = (theCustomerCity+BLANKS20).substring(0, 20);
      }
   }

   public String getCustomerCompany() { return customerCompany; }
   public void setCustomerCompany(String theCustomerCompany) {
      if (theCustomerCompany == null) {
         customerCompany = BLANKS30;
      } else {
         customerCompany = (theCustomerCompany+BLANKS30).substring(0, 30);
      }
   }

   public String getCustomerCountry() { return customerCountry; }
```

Archived

```
   public void setCustomerCountry(String theCustomerCountry) {
      if (theCustomerCountry == null) {
         customerCountry = BLANKS30;
      } else {
         customerCountry = (theCustomerCountry+BLANKS30).substring(0, 30);
      }
   }

   public String getCustomerFirstName() { return customerFirstName; }
   public void setCustomerFirstName(String theCustomerFirstName) {
      if (theCustomerFirstName == null) {
         customerFirstName = BLANKS20;
      } else {
         customerFirstName = (theCustomerFirstName+BLANKS20).substring(0, 20);
      }
   }

   public String getCustomerId() { return customerId; }
   public void setCustomerId(String theCustomerId) {
      if (theCustomerId == null) {
         customerId = BLANKS8;
      } else {
         customerId = (theCustomerId+BLANKS8).substring(0, 8);
      }
   }

   public String getCustomerLastName() { return customerLastName; }
   public void setCustomerLastName(String theCustomerLastName) {
      if (theCustomerLastName == null) {
         customerLastName = BLANKS20;
      } else {
         customerLastName = (theCustomerLastName+BLANKS20).substring(0, 20);
      }
   }

   public String getCustomerLastUpdateDate() { return customerLastUpdateDate; }
   public void setCustomerLastUpdateDate(String theCustomerLastUpdateDate) {
      if (theCustomerLastUpdateDate == null) {
         customerLastUpdateDate = BLANKS8;
      } else {
         customerLastUpdateDate = (theCustomerLastUpdateDate+BLANKS8).substring(0,
8);
      }
   }
```

```
public String getCustomerMailCode() { return customerMailCode; }
public void setCustomerMailCode(String theCustomerMailCode) {
   if (theCustomerMailCode == null) {
      customerMailCode = BLANKS20;
   } else {
      customerMailCode = (theCustomerMailCode+BLANKS20).substring(0, 20);
   }
}

public String getCustomerPhone() { return customerPhone; }
public void setCustomerPhone(String theCustomerPhone) {
   if (theCustomerPhone == null) {
      customerPhone = BLANKS20;
   } else {
      customerPhone = (theCustomerPhone+BLANKS20).substring(0, 20);
   }
}

public String getCustomerState() { return customerState; }
public void setCustomerState(String theCustomerState) {
   if (theCustomerState == null) {
      customerState = BLANKS20;
   } else {
      customerState = (theCustomerState+BLANKS20).substring(0, 20);
   }
}

public String getRequestType() { return requestType; }
public void setRequestType(String theRequestType) {
   if (theRequestType == null) {
      requestType = " ";
   } else {
      requestType = (theRequestType+" ").substring(0, 1);
   }
}

public String getReturnCode() { return returnCode; }
public void setReturnCode(String theReturnCode) {
   if (theReturnCode == null) {
      returnCode = "  ";
   } else {
      returnCode = (theReturnCode+"  ").substring(0, 2);
   }
}
```

Archived

```
   public String getReturnComment() { return returnComment; }
   public void setReturnComment(String theReturnComment) {
      if (theReturnComment == null) {
         returnComment = BLANKS50;
      } else {
         returnComment = (theReturnComment+BLANKS50).substring(0, 50);
      }
   }

}
```

Archived

Archived

C

Sample ECIv2 client

Following is the source for the sample ECIv2 client.

Archived

© Copyright IBM Corp. 2009. All rights reserved.

custbrowse.c

Example: C-1 custbrowse sample client application

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#include <ctgclient.h>
#include <ctgclient_eci.h>

#define COMMAREA_LENGTH 309
struct Commarea {
   char requestType;
   char returnCode[2];
   char id[8];
   char lastName[20];
   char firstName[20];
   char company[30];
   char address1[30];
   char address2[30];
   char city[20];
   char state[20];
   char country[30];
   char mailCode[20];
   char phone[20];
   char lastUpdateDate[8];
   char comment[50];
};

/* The name of the CICS program we are going to call*/
#define PROGRAM "CUSTPROG"
/* The number of CICS servers we are going to request
 * on the list systems call */
#define SERVER_NUM 10

char *userid;
char *password;

/* Method prototypes */
int chooseCICSServer(CTG_ConnToken_t, char*);
int makeCICSCall(CTG_ConnToken_t, char*, struct Commarea*);
void processResponseCode(int);
void displayCustomerDetails(struct Commarea);
```

Archived

```
int menuChoice(void);
void readCustomerRecord(CTG_ConnToken_t, char*);
void deleteCustomerRecord(CTG_ConnToken_t, char*);
void createCustomerRecord(CTG_ConnToken_t, char*);
void updateCustomerRecord(CTG_ConnToken_t, char*);
void getCustomerDetails(struct Commarea*);

/* Entry point for the program which does the following steps:
 * 1. Parse the input parameters
 * 2. Connect to the specified Gateway daemon
 * 3. Present a list of available CICS servers (if required)
 * 4. Display the main menu for the application
 */
int main(int argc, char **argv) {
   CTG_ConnToken_t gwTok;
   int rc = 0;
   int choice = 0;
   int currentOption = 0;
   /*Variables to store the hostname and password*/
   char* hostname = NULL;
   int port = 0;
   /*Variable to hold the chosen CICS server*/
   char server[CTG_LIST_SYSTEM_LENGTH + 1] = {'\0'};

   while((currentOption = getopt(argc,argv,"h:n:s:u:p:"))!=EOF){
      switch(currentOption){
      case 'h':
         hostname = optarg;
         break;
      case 'n':
         port = atoi(optarg);
         break;
      case 's':
         strncpy(server, optarg, CTG_LIST_SYSTEM_LENGTH);
         break;
      case 'u':
         userid = optarg;
         break;
      case 'p':
         password = optarg;
         break;
      }
   }

   if(hostname == NULL || port == 0){
```

Archived

```
      printf("Error: The hostname and port must be specified and valid\n");
      return -1;
   }

   rc = CTG_openRemoteGatewayConnection(hostname, port, &gwTok, 0);

   /* Check whether the connection has been successful */
   if(rc != CTG_OK){
      processResponseCode(rc);
   } else {
      if(server == NULL){
         /*Provide a choice of CICS server to use*/
         rc = chooseCICSServer(gwTok, server);
      }
      if (rc == CTG_OK) {
         /* Start of processing loop */
         while(choice != 5){
            /* Display the menu and get the user's choice */
            choice = menuChoice();
            switch(choice){
            case 1:
               /* Create a record */
               createCustomerRecord(gwTok, server);
               break;
            case 2:
               /* Read a record */
               readCustomerRecord(gwTok, server);
               break;
            case 3:
               /* Delete a record */
               deleteCustomerRecord(gwTok, server);
               break;
            case 4:
               /* Update an existing record */
               updateCustomerRecord(gwTok, server);
               break;
            }

         }
      }
   }
   return rc;
}

/* Displays the menu and prompts the user to make a choice
```

Archived

```
 * of what they want to do.
 *
 * Returns the menu option the user chose.
 */
int menuChoice(){
   /*Buffer to read the menu choice from the keyboard*/
   char choice[128];
   /*Option value to return to the caller*/
   int option;
   /*Flag to indicate whether the user has made a valid choice*/
   int valid = 0;

   /*Display the menu*/
   printf("\nCustomer Control Program\n");
   printf("Choose option:\n");
   printf("1) Create a customer record\n");
   printf("2) Retrieve a customer record\n");
   printf("3) Delete a customer record\n");
   printf("4) Update a customer record\n");
   printf("5) Exit\n");
   printf("Choice: ");

   /*Loop until the user enters a valid choice*/
   while(valid == 0){
      /*Read input from the keyboard*/
      fgets(choice, 3, stdin);
      /*Convert the entered characters to an int*/
      option = atoi(choice);
      /*If this is a valid choice*/
      if(option > 0 && option < 6){
         /*Set the flag to 1 to exit the loop*/
         valid = 1;
      } else {
         /*Tell the user the choice was invalid and prompt again*/
         printf("Invalid choice\n");
         printf("Choice: ");
      }
   }
   /*Return the selected option*/
   return option;
}

/*
 * Retrieves a list of available CICS servers from the
 * specified Gateway daemon. If the information is
```

Archived

```
 * successfully then the user is prompted to choose
 * one from those available.
 *
 * Input:
 * gwTok - A Gateway token representing the Gateway connection
 * server - A pointer to a char[] to store the chosen server name
 *
 * Output:
 * The return code from the CTG_listSystems call.
 */
int chooseCICSServer(CTG_ConnToken_t gwTok, char *server) {
   /*An array of CTG_listSystem_t which is populated by
    *the CTG_listSystems call*/
   CTG_listSystem_t servers[SERVER_NUM];
   /*The size of the servers array*/
   unsigned short serverCount = SERVER_NUM;
   /*Flag to indicate whether we should loop*/
   int loop = 0;
   /*Buffer for user input*/
   char temp[128];
   /*Variable to store the users choice*/
   int choice = 0;
   /*Loop counter*/
   int i = 0;
   /*Variable to store return code*/
   int rc = 0;

   /*Call the CICS TG to get the available CICS servers*/
   rc = CTG_listSystems(gwTok, &serverCount, servers);
   if (rc == CTG_OK || rc == CTG_ERR_MORE_SYSTEMS) {
      /*If the call returned some servers*/
      if (rc == CTG_ERR_MORE_SYSTEMS) {
         /*Display a warning if there were more defined*/
         /*servers than we had space for*/
         printf("Warning: There are more systems defined than could be displayed\n");
         rc = ECI_NO_ERROR;
      }
      /*Display the server information to the user*/
      printf("Please choose the CICS server to use:\n");
      for (i = 0; i < serverCount; i++) {
         printf("%d) %s\n", i + 1, servers[i].SystemName);
      }
      do {
         /*Prompt the user to choose a server*/
         printf("Select server: ");
```

```
         fgets(temp, 3, stdin);
         choice = atoi(temp);
         if (choice > 0 && choice <= serverCount) {
            strcpy(server, servers[choice - 1].SystemName);
            loop = 0;
         } else {
            printf("Invalid choice (%d)\n", choice);
            loop = 1;
         }
      } while (loop == 1);
   }
   return rc;
}

/*
 * Function to retrieve a customer record and display it
 * to the user.
 *
 * Input:
 * gwTok - A Gateway token representing the Gateway connection
 * server - A pointer to a char[] containing the CICS server name
 */
/*
 * Function to retrieve a customer record and display it
 * to the user.
 *
 * Input:
 * gwTok - A Gateway token representing the Gateway connection
 * server - A pointer to a char[] containing the CICS server name
 */
void readCustomerRecord(CTG_ConnToken_t gwTok, char *server){
   struct Commarea commarea;
   char recordNumber[10] = {0,0,0,0,0,0,0,0,0,0};
   char rcString[3];
   char tempStr[51];
   int rc = 0;

   printf("\nRead a customer record\n");
   printf("Enter customer number: ");
   fgets(recordNumber, 10, stdin);

   memset(&commarea, 0, sizeof(commarea));
   commarea.requestType = 'R';
   memcpy(commarea.id, recordNumber, sizeof(commarea.id));
```

Archived

```
   rc = makeCICSCall(gwTok, server, &commarea);
   if(rc == ECI_NO_ERROR){
      strncpy(rcString, commarea.returnCode, sizeof(commarea.returnCode));
      rc = atoi(rcString);
      if(rc == 0){
         displayCustomerDetails(commarea);
      } else {
         strcpy(tempStr, commarea.comment);
         printf("Error: %s\n", tempStr);
      }
   }
}

/*
 * Function to delete a customer record.
 *
 * Input:
 * gwTok - A Gateway token representing the Gateway connection
 * server - A pointer to a char[] containing the CICS server name
 */
void deleteCustomerRecord(CTG_ConnToken_t gwTok, char *server) {
   struct Commarea commarea;
   char recordNumber[10] = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };
   char rcString[3] = {0,0,0};
   char tempStr[51];
   int rc = 0;

   printf("\nDelete customer record\n");
   printf("Enter customer number: ");
   fgets(recordNumber, 10, stdin);

   memset(&commarea, 0, sizeof(commarea));
   commarea.requestType = 'D';
   memcpy(commarea.id, recordNumber, sizeof(commarea.id));

   rc = makeCICSCall(gwTok, server, &commarea);
   if (rc == ECI_NO_ERROR) {
      strncpy(rcString, commarea.returnCode, sizeof(commarea.returnCode));
      strcpy(tempStr, commarea.comment);
      rc = atoi(rcString);
      if (rc == 0) {
         printf("%s\n", tempStr);
      } else {
         printf("Error: %s\n", tempStr);
      }
```

```
   }
}

/*
 * Function to create a customer record.
 *
 * Input:
 * gwTok - A Gateway token representing the Gateway connection
 * server - A pointer to a char[] containing the CICS server name
 */
void createCustomerRecord(CTG_ConnToken_t gwTok, char *server){
   struct Commarea commarea;
   char rcString[3] = {0,0,0};
   char tempStr[51];
   int rc = 0;

   memset(&commarea, 0, sizeof(commarea));
   commarea.requestType = 'C';
   getCustomerDetails(&commarea);
   rc = makeCICSCall(gwTok, server, &commarea);
   if (rc == ECI_NO_ERROR) {
      strncpy(rcString, commarea.returnCode, sizeof(commarea.returnCode));
      strcpy(tempStr, commarea.comment);
      rc = atoi(rcString);
      if (rc == 0) {
         printf("%s\n", tempStr);
      } else {
         printf("Error: %s\n", tempStr);
      }
   }
}

/*
 * Function to update a customer record.
 *
 * Input:
 * gwTok - A Gateway token representing the Gateway connection
 * server - A pointer to a char[] containing the CICS server name
 */
void updateCustomerRecord(CTG_ConnToken_t gwTok, char *server){
   struct Commarea commarea;
   char rcString[3] = {0,0,0};
   char tempStr[51];
   int rc = 0;
```

Archived

```
   memset(&commarea, 0, sizeof(commarea));
   commarea.requestType = 'U';
   getCustomerDetails(&commarea);
   rc = makeCICSCall(gwTok, server, &commarea);
   if (rc == ECI_NO_ERROR) {
      strncpy(rcString, commarea.returnCode, sizeof(commarea.returnCode));
      strcpy(tempStr, commarea.comment);
      rc = atoi(rcString);
      if (rc == 0) {
         printf("%s\n", tempStr);
      } else {
         printf("Error: %s\n", tempStr);
      }
   }
}

/**
 * Prompts the user for each of the fields that make up
 * a customer record. Each field is then stored in the
 * supplied Commarea structure ready for use with the
 * CICS program.
 *
 * Input:
 * commarea - A pointer to a Commarea structure
 */
void getCustomerDetails(struct Commarea *commarea) {
   char tempStr[128];
   char *tmpPtr;

   memset(tempStr, 0, sizeof(tempStr));
   printf("Id (8):");
   fgets(tempStr, 10, stdin);
   memcpy(commarea->id, tempStr, sizeof(commarea->id));

   memset(tempStr, 0, sizeof(tempStr));
   printf("Last name (20):");
   fgets(tempStr, 22, stdin);
   tmpPtr = strchr(tempStr, '\n');
   if(tmpPtr != NULL){
      *tmpPtr = '\0';
   }
   memcpy(commarea->lastName, tempStr, sizeof(commarea->lastName));

   memset(tempStr, 0, sizeof(tempStr));
   printf("First name (20):");
```


Archived

```
fgets(tempStr, 22, stdin);
tmpPtr = strchr(tempStr, '\n');
if (tmpPtr != NULL) {
   *tmpPtr = '\0';
}
memcpy(commarea->firstName, tempStr, sizeof(commarea->firstName));

memset(tempStr, 0, sizeof(tempStr));
printf("Company (30):");
fgets(tempStr, 32, stdin);
tmpPtr = strchr(tempStr, '\n');
if (tmpPtr != NULL) {
   *tmpPtr = '\0';
}
memcpy(commarea->company, tempStr, sizeof(commarea->company));

memset(tempStr, 0, sizeof(tempStr));
printf("Address line 1 (30):");
fgets(tempStr, 32, stdin);
tmpPtr = strchr(tempStr, '\n');
if (tmpPtr != NULL) {
   *tmpPtr = '\0';
}
memcpy(commarea->address1, tempStr, sizeof(commarea->address1));

memset(tempStr, 0, sizeof(tempStr));
printf("Address line 2 (30):");
fgets(tempStr, 32, stdin);
tmpPtr = strchr(tempStr, '\n');
if (tmpPtr != NULL) {
   *tmpPtr = '\0';
}
memcpy(commarea->address2, tempStr, sizeof(commarea->address2));

memset(tempStr, 0, sizeof(tempStr));
printf("City (20):");
fgets(tempStr, 22, stdin);
tmpPtr = strchr(tempStr, '\n');
if (tmpPtr != NULL) {
   *tmpPtr = '\0';
}
memcpy(commarea->city, tempStr, sizeof(commarea->city));

memset(tempStr, 0, sizeof(tempStr));
printf("State (20):");
```

Archived

```
   fgets(tempStr, 22, stdin);
   tmpPtr = strchr(tempStr, '\n');
   if (tmpPtr != NULL) {
      *tmpPtr = '\0';
   }
   memcpy(commarea->state, tempStr, sizeof(commarea->state));

   memset(tempStr, 0, sizeof(tempStr));
   printf("Country (30):");
   fgets(tempStr, 32, stdin);
   tmpPtr = strchr(tempStr, '\n');
   if (tmpPtr != NULL) {
      *tmpPtr = '\0';
   }
   memcpy(commarea->country, tempStr, sizeof(commarea->country));

   memset(tempStr, 0, sizeof(tempStr));
   printf("Mail code (20):");
   fgets(tempStr, 22, stdin);
   tmpPtr = strchr(tempStr, '\n');
   if (tmpPtr != NULL) {
      *tmpPtr = '\0';
   }
   memcpy(commarea->mailCode, tempStr, sizeof(commarea->mailCode));

   memset(tempStr, 0, sizeof(tempStr));
   printf("Phone number (20):");
   fgets(tempStr, 22, stdin);
   tmpPtr = strchr(tempStr, '\n');
   if (tmpPtr != NULL) {
      *tmpPtr = '\0';
   }
   memcpy(commarea->phone, tempStr, sizeof(commarea->phone));
}

/*
 * Function to call the CICS program and wait for the response.
 *
 * Input:
 * gwTok - A Gateway token representing the Gateway connection
 * server - A pointer to a char[] containing the CICS server name
 * commarea - A pointer to a commarea structure containing the input to
 *            the CICS program.
 */
int makeCICSCall(CTG_ConnToken_t gwTok, char *server, struct Commarea *commarea) {
```

Archived

```
   /*ECI Parameters structure*/
   CTG_ECI_PARMS eciParms;
   /*Variable to store the return code*/
   int rc = 0;
   /*The name of the CICS program*/
   char *program = PROGRAM;
   /*Userid to run the CICS program under*/
   char *userid = "ANDREWS";
   /*Password for the specified userid*/
   char *password = "b4ckst4b";

   /*Initialize the parameter structure to zeros*/
   memset(&eciParms, 0, sizeof(CTG_ECI_PARMS));

   /*Set the necessary ECI parameters*/
   eciParms.eci_version = ECI_VERSION_2;
   eciParms.eci_call_type = 2;
   memcpy(eciParms.eci_program_name, program, strlen(program));
   memcpy(eciParms.eci_system_name, server, strlen(server));
   memcpy(eciParms.eci_userid, userid, strlen(userid));
   memcpy(eciParms.eci_password, password, strlen(password));
   eciParms.eci_commarea = commarea;
   eciParms.eci_commarea_length = COMMAREA_LENGTH;
   eciParms.eci_extend_mode = ECI_NO_EXTEND;

   /*Call the Gateway to run the CICS program*/
   rc = CTG_ECI_Execute(gwTok, &eciParms);
   /*Check the return code*/
   if (rc != ECI_NO_ERROR) {
      /*Display the return code value*/
      processResponseCode(rc);
      if (rc == ECI_ERR_TRANSACTION_ABEND) {
         /*If the program abended then display the abend code*/
         printf("Abend code %4.4s\n", eciParms.eci_abend_code);
      }
   }

   return rc;
}

/*
 * Displays the record fields from the supplied commarea
 * to the user.
 *
 * Input:
```


Archived

```
 * commarea - The Commarea containing the record information.
 */
void displayCustomerDetails(struct Commarea commarea) {
   char tempStr[31];
   memset(tempStr, 0, sizeof(tempStr));
   strncpy(tempStr, commarea.id, sizeof(commarea.id));
   printf("Id:\t\t%s\n", tempStr);
   memset(tempStr, 0, sizeof(tempStr));
   strncpy(tempStr, commarea.lastName, sizeof(commarea.lastName));
   printf("Lastname:\t%s\n", tempStr);
   memset(tempStr, 0, sizeof(tempStr));
   strncpy(tempStr, commarea.firstName, sizeof(commarea.firstName));
   printf("Firstname:\t%s\n", tempStr);
   memset(tempStr, 0, sizeof(tempStr));
   strncpy(tempStr, commarea.company, sizeof(commarea.company));
   printf("Company:\t%s\n", tempStr);
   memset(tempStr, 0, sizeof(tempStr));
   strncpy(tempStr, commarea.address1, sizeof(commarea.address1));
   printf("Address1:\t%s\n", tempStr);
   memset(tempStr, 0, sizeof(tempStr));
   strncpy(tempStr, commarea.address2, sizeof(commarea.address2));
   printf("Address2:\t%s\n", tempStr);
   memset(tempStr, 0, sizeof(tempStr));
   strncpy(tempStr, commarea.city, sizeof(commarea.city));
   printf("City:\t\t%s\n", tempStr);
   memset(tempStr, 0, sizeof(tempStr));
   strncpy(tempStr, commarea.state, sizeof(commarea.state));
   printf("State:\t\t%s\n", tempStr);
   memset(tempStr, 0, sizeof(tempStr));
   strncpy(tempStr, commarea.country, sizeof(commarea.country));
   printf("Country:\t%s\n", tempStr);
   memset(tempStr, 0, sizeof(tempStr));
   strncpy(tempStr, commarea.mailCode, sizeof(commarea.mailCode));
   printf("Mailcode:\t%s\n", tempStr);
   memset(tempStr, 0, sizeof(tempStr));
   strncpy(tempStr, commarea.phone, sizeof(commarea.phone));
   printf("Phone:\t\t%s\n", tempStr);
   memset(tempStr, 0, sizeof(tempStr));
   strncpy(tempStr, commarea.lastUpdateDate, sizeof(commarea.lastUpdateDate));
   printf("Lastupdate:\t%s\n", tempStr);
}

/*
 * Displays the name of the supplied return code to the
 * user.
```

```
 *
 * Input:
 * rc - The return code to display.
 */
void processResponseCode(int rc) {
   char rcString[CTG_MAX_RCSTRING + 1];

   CTG_getRcString(rc, rcString);

   printf("Return code: %s\n", rcString);
}
```

Archived

Archived

D

CICS samples

The following information is the description of the COMMAREA and channel-based CICS COBOL programs and its usage and installation into the back-end CICS TS environment.

© Copyright IBM Corp. 2009. All rights reserved.

CICS COBOL server program components

- CUSTBRWS.cbl

  COBOL program to 'browse' the VSAM file and return a series of records
- CUSTPROG.cbl

  COBOL sample program that performs Create, Read, Update, Delete of records in a VSAM file.
- CUSTREC.cbl

  Copybook that describes the layout of the VSAM file
- CUSTIORQ.cbl

  Copybook for the COMMAREA layout
- CUSTLOAD.cbl

  COBOL utility program: LINK with CECI to load records in the VSAM file
- CUSTVSAM.jcl

  IDCAMS statements to define the VSAM file

Executing the COMMAREA-based server program

To create, read, update, or delete from the VSAM file, place data in the COMMAREA as appropriate and LINK (or the equivalent) to CUSTPROG. The COMMAREA copybook CUSTIORQ layout is shown in Example D-1.

Example: D-1 CUSTIORQ.cbl copybook

```
01 CUSTPROG-COMMAREA.
   03  REQUEST-TYPE                PIC X.
   03  RET-CODE                    PIC XX.
   03  CustomerId                  PIC X(8).
   03  CustomerLastName            PIC X(20).
   03  CustomerFirstName           PIC X(20).
   03  CustomerCompany             PIC X(30).
   03  CustomerAddr1               PIC X(30).
   03  CustomerAddr2               PIC X(30).
   03  CustomerCity                PIC X(20).
   03  CustomerState               PIC X(20).
   03  CustomerCountry             PIC X(30).
   03  CustomerMailCode            PIC X(20).
   03  CustomerPhone               PIC X(20).
   03  CustomerLastUpdateDate      PIC X(8).
   03  RETURN-COMMENT              PIC X(50).
```

Archived

Error situations

When the CUSTPROG program is unable to fulfill a request, it places an indicator in the RET-CODE field along with a verbose explanation of the error in the RETURN-COMMENT field.

To Read

1. Enter `R` in the request-type field,
2. Enter the customer ID in the CustomerId field,
3. LINK to CUSTPROG to read customer information of that particular customer ID.

To Delete

1. Enter `D` in the request-type field,
2. Enter the customer ID in the CustomerId field,
3. LINK to CUSTPROG to delete the record from VSAM.

To Add

1. Enter a `C` in the request-type field,
2. Enter the customer ID in the CustomerId field,
3. Enter the fields to be added in the rest of the fields
4. LINK to CUSTPROG to add a new record to VSAM.

To Update

1. Enter a `U` in the request-type field,
2. Enter the customer ID in the CustomerID field,
3. Enter the contents to replace the record fields in the appropriate fields,
4. LINK to CUSTPROG. NOTE that this is a record replace, the program is not sophisticated enough to just replace an individual field

Example D-2 on page 246 shows the reading of customer information by passing the customer ID using the COMMAREA to the CUSTPROG sample program. In addition to reading information the CUSTPROG sample program contains the business logic for adding, deleting and updating the VSAM records.

Archived

Example: D-2   The CUSTPROG read logic

```
2000-DO-READ.
           EXEC CICS READ FILE('CUSTFILE')
                  RIDFLD(CustomerId)
                  INTO(CUST-FILE-LAYOUT)
                  RESP(RESP-FLD)
                  END-EXEC.
           EVALUATE RESP-FLD
               WHEN DFHRESP(NORMAL)
                   PERFORM 5200-MOVE-RECORD-TO-COMMAREA
                   MOVE '00' TO RET-CODE
                   MOVE 'Customer information retrieved'
                         TO RETURN-COMMENT
                   PERFORM 7100-RETURN-ERROR-MESSAGE
               WHEN DFHRESP(NOTFND)
                   MOVE '04' TO RET-CODE
                   MOVE 'Fund not found'
                         TO RETURN-COMMENT
                   PERFORM 7100-RETURN-ERROR-MESSAGE
               WHEN OTHER
                   MOVE '04' TO RET-CODE
                   MOVE 'Error reading customer file'
                           TO RETURN-COMMENT
                   PERFORM 7100-RETURN-ERROR-MESSAGE
           END-EVALUATE.
```

Installing the COMMAREA-based sample program on CICS

Perform the following steps to install the COMMAREA-based sample program on CICS.

1. Compile both the CUSTPROG and CUSTLOAD programs. They need access to the CUSTREC and CUSTIORQ copybooks.
2. Define the VSAM file using IDCAMS.
3. Define a PROGRAM definition for the above programs.
4. Define a FILE definition named CUSTDATA and set its properties including fixed length, updatable, readable, and writeable.
5. LINK to the CUSTLOAD program using CECI to install sample data into the VSAM file. You can do a quick edit on the CUSTLOAD.cbl file if you want to load specific names in the records in the VSAM file.

Archived

Executing the Channel-based server program

The sample CUSTBRWS COBOL program supports a channel and container interface. To instruct the CUSTBRWS program to return a series of records, place the information in Example D-3 in a container named BROWSE-INFO using the layout shown in Example D-3.

Example: D-3 Input container record information

```
01 Request-Information.
           05  Max-Records                 PIC 9999.
           05  Direction                   PIC X.
               88  Browse-Forward          VALUE 'F'.
               88  Browse-Backward         VALUE 'B'.
           05  Start-With-This-Key         PIC X(8).
           05  Start-Key-Length            PIC 99.
```

- Specify the number of records you want returned in the Max-Records field
- Specify the direction of the browse in the Direction field (B=backward, F=forward)
- Specify the starting key in the Start-With-This-Key field.
- Put the length of the key in the Start-Key-Length field (if you specify anything other than a 0 or 8 the program performs the browse using a generic key)

Link to the CUSTBRWS program.

Any error information is stored in a container. If error information exists the program returns a container named 'ERROR' in the format shown in Example D-4.

Example: D-4 Error container information

```
01  Error-Information.
         05  Request-Return-Code        PIC XX.
           05  Request-Return-Comment       PIC X(50).
```

- The Request-Return-Code contains a numeric value.
- The Request-Return-Comment contains a verbose explanation of the return code.

When successful, the number of requested records is returned in a container named 'NUM-RECORDS' as a 4 position display numeric field (e.g. 0005). The records are returned in a series of containers named '0001', '0002', '0003', etc. The contents of the containers is the record layout in the CUSTREC copybook as shown in Example D-5 on page 248.

Archived

Example: D-5 Output container information

```
03  CustFileId                   PIC X(8).
         03  CustFileLastName          PIC X(20).
         03  CustFileFirstName         PIC X(20).
         03  CustFileCompany           PIC X(30).
         03  CustFileAddr1             PIC X(30).
         03  CustFileAddr2             PIC X(30).
         03  CustFileCity              PIC X(20).
         03  CustFileState             PIC X(20).
         03  CustFileCountry           PIC X(30).
         03  CustFileMailCode          PIC X(20).
         03  CustFilePhone             PIC X(20).
           03  CustFileLastUpdateDate       PIC X(8).
```

Note: 0001, 0002, and so forth, are used as container names to enable the client program to display them in any order. When browsing containers the order is not guaranteed however you might want to display the records in a certain order. In this case the order is controlled by using the container name as the order. This allows you to start a loop in the client program and obtain first container 0001, then 0002, and so forth.

Installing the Channel-based sample application on CICS

Perform the following steps:

1. Compile CUSTPROG, CUSTBRWS, and CUSTLOAD . They need access to the CUSTREC and CUSTIORQ copybooks.
2. Define the VSAM file using IDCAMS.
3. Define a PROGRAM definition for the above programs.
4. Define a FILE definition named CUSTDATA, fixed length, updatable, readable, writeable, and so forth.

Next LINK to the CUSTLOAD program using CECI to load a few records into the VSAM file (you can do a quick edit on the CUSTLOAD.cbl file if you want specific names in the records in the VSAM file)

Compiling and Executing Java Sample Programs

The Sample Java programs are organized in three different packages.

- com.ibm.itso.eci

  Contains Java samples which interacts the server side CICS program through base classes.

- com.ibm.itso.jca

  Contains Java samples which interacts the server side CICS program through J2C adaptor.

- com.ibm.itso.model

  Common data model classes which will be used across programs.

All the programs are available as additional material. See Appendix E, “Additional material” on page 253.

Compilation of Java programs are done using Ant build tool. The directory antrules contains two files namely build.xml and buld.properties. Former contains the build rules for compilation of classes and the latter contains the environment variable required to run the build. To compile the sample programs, perform the following steps.

1. Install Ant build tool

   Ant is an open source build tool for Java programs. Download it from http://ant.apache.org/, and install it in your development machine.

2. Set build environment variables

   Environment variables required to compile the Java samples are available in *build.properties* file. There are two variable available in this file, which need to be entered. First variable CTG_LIB points to the location where CTG related class libraries are available. Second variable J2EE_HOME points to the J2EE installation directory. An example entry is shown below.

   – CTG_LIB

     C:/Program Files/IBM/CICS Transaction Gateway/classes.

   – J2EE_HOME

     C:/Java5

3. Compile the sample programs

   Run *ant* from *antrules* directory. This will compile all the Java files and put the compiled class files under a new directory name *build.* Apart from this, it will create a *itso.jar* file, which contains all the compiled classes as a single *jar* file.

4. Execute the program.

   Executing the program has to be done from the command prompt. Include the `ctgclient.jar`, `connector.jar`, `ccf2.jar`, `cicsj2ee.jar`, and the `ctgserver.jar` from the CTG product installation and the `j2ee.jar` from the J2EE installation into your CLASSPATH. Also include either `itso.jar` or *build* directory in the CLASSPATH.

5. Invoke the respective sample program from the command prompt. For example, invoke the *CustProgMultiRecJ2C* application using java com.ibm.itso.jca.CustProgMultiRecJ2C

Output of a typical compilation and execution sequence is shown in Example D-6.

Example: D-6 Typical compilation and execution of sample application

```
1 C:\itso\src\antrules>ant clean
Buildfile: build.xml

clean:
   [delete] Deleting directory C:\itso\src\build

BUILD SUCCESSFUL
Total time: 0 seconds
2 C:\itso\src\antrules>ant
Buildfile: build.xml

init:
    [mkdir] Created dir: C:\itso\src\build

compile:
    [javac] Compiling 9 source files to C:\itso\src\build

makejar:
      [jar] Building jar: C:\itso\src\build\itso.jar

all:

BUILD SUCCESSFUL
Total time: 1 second
3 C:\itso\src\antrules>SET CTG_LIB=C:\Program Files\IBM\CICS
Transaction Gateway\classes
4 C:\itso\src\antrules>SET J2EE_HOME=C:\Java5
5 C:\itso\src\antrules>SET
CLASSPATH=%CTG_LIB%\ctgclient.jar;%CTG_LIB%\connector.jar;%CTG_LIB%\ccf
```


Archived

```
2.jar;%CTG_LIB%\cicsj2ee.jar;%CTG_LIB%\ctgserver.jar;%J2EE_HOME%\lib\j2
ee.jar;..\build;%CLASSPATH%

6 C:\itso\src\antrules>java com.ibm.itso.jca.CustProgMultiRecJ2C
CIC1IPIC 2006 WTSC59.itso.ibm.com
---------- CustProgMultiRecJ2C - Using Channels and Containers
----------
Using the CICS Transaction Gateway J2C Classes.  Invoking program
CUSTBRWS.
The server CIC1IPIC will be accessed at WTSC59.itso.ibm.com listening
on port
2006.
Requesting a maximum of 5 records starting with customer ids that begin
with "
00000000".
The sequence will be ascending.
The object returned from the NUM-RECORDS was a String
The number of records returned was: 5
Customer id: 00000001, name: George Jetson, last updated: 11/17/08
Customer id: 00000002, name: Wilma FlintStone, last updated: 11/17/08
Customer id: 10000001, name: Fred FlintStone, last updated: 11/17/08
Customer id: 10000002, name: Bugs Bunny, last updated: 11/17/08
Customer id: 20000002, name: Oscar Rabit, last updated: 11/17/08

C:\itso\src\antrules>
```

- 1 Use ant clean to remove any previously created class files. This command deletes the build directory.
- 2 Compile the source file by running ant.
- 3 4 5 Set the CLASSPATH variable for executing the Java sample program.
- 6 Execute the Java program with necessary arguments.

Archived

Archived

E

Additional material

This Redbooks publication refers to additional material that can be downloaded from the Internet as described below.

Locating the Web material

The Web material associated with this book is available in softcopy on the Internet from the IBM Redbooks Web server. Point your Web browser at:

ftp://www.redbooks.ibm.com/redbooks/SG247714

Alternatively, you can go to the IBM Redbooks Web page at:

ibm.com/redbooks

Select the **Additional materials** and open the directory that corresponds with the IBM Redbooks form number, SG247714.

Archived

© Copyright IBM Corp. 2009. All rights reserved.

Using the Web material

The additional Web material that accompanies this book includes the following files:

File name	*Description*
SG24-7714.zip	Zipped Code Samples, readme file

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. Open the readme file. This file contains the description of the accompanying files and the instructions for their use.

Archived

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this book.

IBM Redbooks

For information about ordering these publications, see "How to get Redbooks" on page 256. Note that some of the documents referenced here may be available in softcopy only.

- *CICS Transaction Server V3R1 Channels and Containers Revealed, SG24-7227*
- *Java Connectors for CICS: Featuring the J2EE Connector Architecture*, SG24-6401

Other publications

These publications are also relevant as further information sources:

- *CICS Transaction Gateway for z/OS V7.2 Administration* SC34-6961
- *CICS Transaction Gateway V7.2 Programming Guide* SC34-6965
- *CICS Transaction Gateway V7.2 Programming Reference* SC34-6966

Online resources

These Web sites are also relevant as further information sources:

- *CICS Transaction Gateway for z/OS Version 7 Release 2 Information Center*

  http://publib.boulder.ibm.com/infocenter/cicstgzo/v7r2//index.jsp?fdoc=aimctg
- *CICS Transaction Gateway for Multiplatforms Version 7 Release 2 Information Center*

  http://publib.boulder.ibm.com/infocenter/cicstgmp/v7r2//index.jsp?fdoc=aimctg

© Copyright IBM Corp. 2009. All rights reserved.

- *CA72: CICS TG: Developing .NET components for CICS connectivity*

  http://www.ibm.com/support/docview.wss?rs=1083&uid=swg24018409
- CA73: CICS TG V7.2 .NET application support

  http://www-01.ibm.com/support/docview.wss?rs=1083&uid=swg24021950&fdoc=aimctg
- Integrating WebSphere Application Server and CICS using CICS Transaction Gateway

  http://www.ibm.com/developerworks/websphere/techjournal/0607_wakelin/0607_wakelin.html
- Improving the efficiency of TCPIP through null stripping

  http://www-01.ibm.com/support/docview.wss?uid=swg21066715
- Integrating CICS and WebSphere Application Server using XA global transactions

  http://www.ibm.com/developerworks/websphere/techjournal/0607_wakelin/0607_wakelin.html
- Transactional integration of WebSphere Application Server and CICS with the J2EE Connector Architecture

  http://www-01.ibm.com/support/docview.wss?rs=1083&uid=swg24018409

How to get Redbooks

You can search for, view, or download Redbooks, Redpapers, Technotes, draft publications and Additional materials, and order hardcopy Redbooks, at this Web page:

ibm.com/redbooks

Help from IBM

IBM Support and downloads

ibm.com/support

IBM Global Services

ibm.com/services

Archived

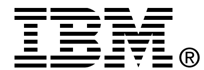
Developing Connector Applications for CICS

(0.5" spine)
0.475"<->0.875"
250 <-> 459 pages

Archived

Archived

Archived



Developing Connector Applications for CICS

Java, J2C and C ECIv2 interfaces revealed

Tooling to build and deploy J2C based clients

Sample client programs

This IBM Redbooks publication focuses on application development using the CICS Transaction Gateway V7 programming interfaces for access to CICS, including Java, JCA, C, and .NET client development. This Redbooks publication also includes the exploitation of the new CICS TS V3 Channels and Containers programming model for exchange of large payloads, support for the J2C beans provided by Rational Application Developer, and the invocation of CICS programs from the .NET environment.

Archived

INTERNATIONAL TECHNICAL SUPPORT ORGANIZATION

BUILDING TECHNICAL INFORMATION BASED ON PRACTICAL EXPERIENCE

IBM Redbooks are developed by the IBM International Technical Support Organization. Experts from IBM, Customers and Partners from around the world create timely technical information based on realistic scenarios. Specific recommendations are provided to help you implement IT solutions more effectively in your environment.

For more information:
ibm.com/redbooks

SG24-7714-00 0738432814