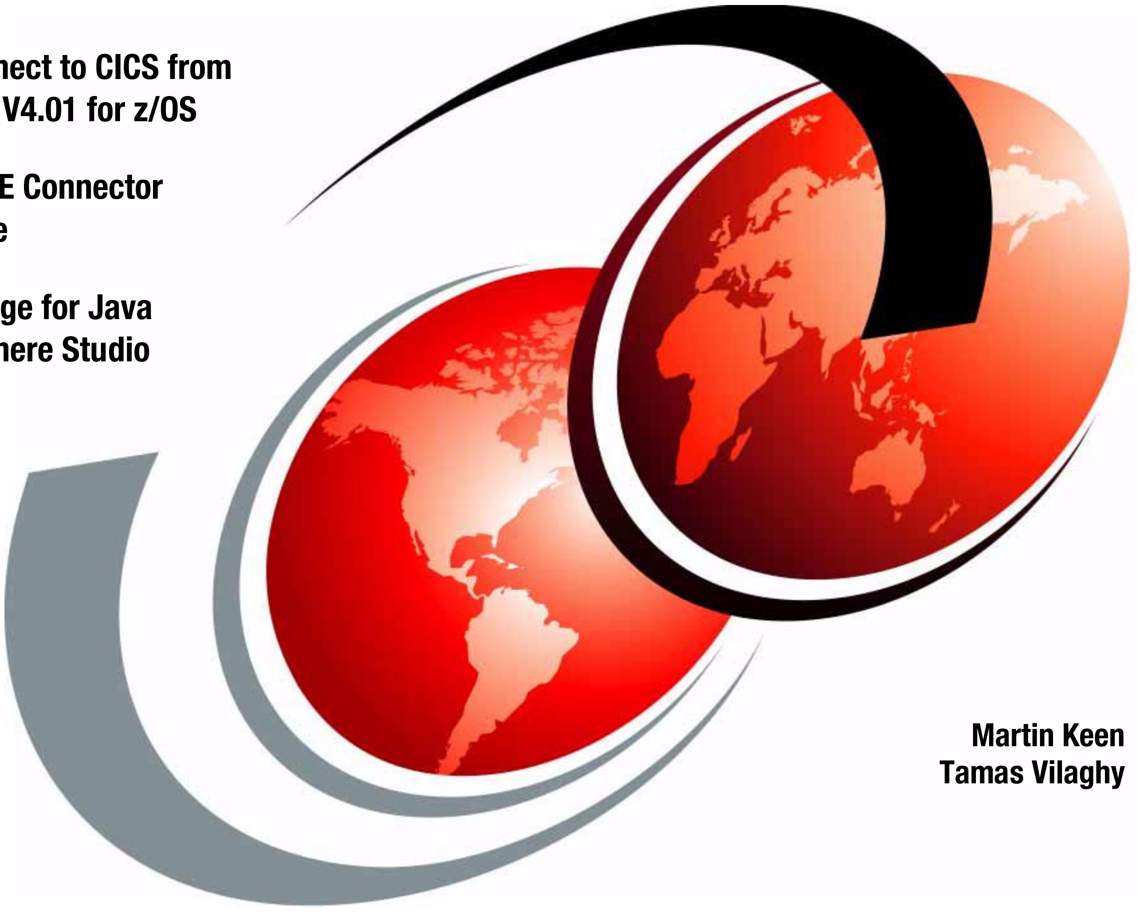
IBM

From code to deployment: Connecting to CICS from WebSphere V4.01 for z/OS

How to connect to CICS from WebSphere V4.01 for z/OS

Use the J2EE Connector Architecture

Use VisualAge for Java and WebSphere Studio



Martin Keen
Tamas Vilaghy

Redpaper

ibm.com/redbooks

International Technical Support Organization

From code to deployment: Connecting to CICS from WebSphere V4.01 for z/OS

May 2002

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

First Edition (May 2002)

This edition applies to Version 4.0.1 of WebSphere Application Server with V4.0.2 of CICS Transaction Gateway

Contents

Notices

Trademarks

Preface

The way you access CICS business logic from enterprise applications running in WebSphere Application Server has changed. In March 2002, WebSphere Application Server V4.0.1 for z/OS introduced support for the J2EE Connector Architecture, and with this support comes a whole new way of connecting to enterprise systems like CICS and IMS.

This Redpaper describes how to use the J2EE Connector Architecture support to create a managed connection between WebSphere for z/OS and CICS Transaction Server, using the CICS ECI resource adapter. The entire process is covered in this paper, from developing an enterprise application that makes calls to CICS, through to deploying the enterprise application to WebSphere for z/OS.

This Redpaper describes detailed instructions on how to prepare IBM's VisualAge for Java and the new WebSphere Studio configurations for use with the CICS ECI resource adapter, and then details how to build an enterprise application using these tools. This enterprise application is then deployed to a WebSphere J2EE server configured for the CICS ECI resource adapter.

Since this Redpaper was originally published, WebSphere Application Server for z/OS has introduced support for enterprise services generated in WebSphere Studio Application Developer Integration Edition, via Service Level 4. This is now the strategic way to develop applications that use the CICS ECI resource adapter. However, as a starting point you may wish to follow the approach documented here first, before moving on to the world of enterprise services.

The team that wrote this Redpaper

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

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

Tamas Vilaghy is a project leader at the International Technical Support Organization, Poughkeepsie Center. He leads redbook projects dealing with e-business on zSeries servers. Before joining the ITSO 6 months ago, he worked in System Sales Unit and Global Services departments of IBM Hungary. Tamas spent 2 years in Poughkeepsie from 1998 to 2000 dealing with zSeries marketing and competitive analysis. From 1991 to 1998 he held technical, marketing and sales positions dealing with zSeries.

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

Mike Cox
Washington Systems Center

Ken Muckenhaupt
IBM eTP Center, Poughkeepsie

Ivan Joslin
IBM WebSphere for z/OS development

Rich Conway
ITSO, Poughkeepsie

Phil Wakelin, Richard Johnson, and Daniel Cerecedo Diaz
The team of the *Java Connectors for CICS* redbook

Become a published author

Join us for a two- to six-week residency program! Help write an IBM Redbook dealing with specific products or solutions, while getting hands-on experience with leading-edge technologies. You'll team with IBM technical professionals, Business Partners and/or customers.

Your efforts will help increase product acceptance and customer satisfaction. As a bonus, you'll 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 papers to be as helpful as possible. Send us your comments about this Redpaper or other Redbooks in one of the following ways:

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

  ibm.com/redbooks

- Send your comments in an Internet note to:

  redbook@us.ibm.com

- Mail your comments to:

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

1 Overview

This chapter provides an overview of the new connector support in WebSphere Application Server V4.0.1 for z/OS. It introduces the J2EE Connector Architecture and CICS ECI resource adapter, then goes on to explain the choice of development tools you can use to develop enterprise applications that use this connector technology.

Finally it describes the sample used in this Redpaper, and lists a set of prerequisites.

This chapter contains the following sections:

- Connector support overview
- Development tool choices
- The example used in this Redpaper
- Before you begin

1.1 Connector support overview

So you want to access CICS from WebSphere?

The way you access CICS business logic from enterprise applications running in WebSphere Application Server has changed. In March 2002, WebSphere Application Server V4.0.1 for z/OS introduced support for the J2EE Connector Architecture, and with this support comes a whole new way of connecting to enterprise systems like CICS and IMS.

What is the J2EE Connector Architecture

It's a standard way for a Java component to connect to any enterprise system. It is now part of the J2EE standard, introduced in the J2EE V1.3 specification. WebSphere Application Server V4 is a J2EE V1.2 compliant application server, but has been extended to include this support.

Figure 1-1 shows how WebSphere connects to CICS from a session bean, using the J2EE Connector Architecture. This introduces the CICS ECI resource adapter.

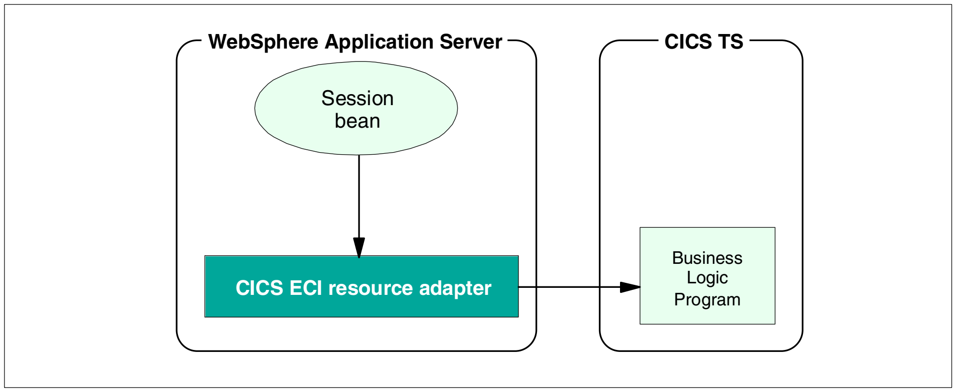


Figure 1-1 Overview of connecting to CICS

What is the CICS ECI resource adapter

First, lets introduce what a resource adapter is. The J2EE Connector Architecture states that the way an application server (like WebSphere) communicates with an enterprise system (like CICS) is through a resource adapter. This resource adapter is written specifically for the enterprise system it supports, but the same resource adapter can be plugged in to any application server.

The resource adapter exposes two components:

- Common Client Interface

  This is a common API amongst all resource adapters. A Java component wishing to use the resource adapter to access an enterprise system does so through this common API. Unlike connectors in the past, there is not a connector-specific API for the application programmer to learn, which significantly differs with each connector.

- System contracts

  This is how the application server communicates with the resource adapter. This is also common amongst resource adapters, allowing a resource adapter to plug in to any application server that supports the J2EE Connector Architecture.

The resource adapter that specifically provides access to CICS business logic is the CICS ECI resource adapter.

How does this affect my connections to CICS

The CICS ECI resource adapter allows WebSphere to take much tighter control in how connections to CICS are made. It allows WebSphere to *manage* the connections by using the system contracts of the resource adapter. This managed connection allows WebSphere to perform the following:

- Provide two phase commit processing using RRS
- Flow security credentials based on deployment information specified in the enterprise application. The userid you flow to CICS can come from:
  - The enterprise application
  - The userid of the J2EE server running the enterprise application
  - The userid of the caller of the enterprise application
  - A userid mapped to an EJB security role
- Provide sysplex-wide pooling of connections.

Do I still need the CICS Transaction Gateway

Yes! It is the CICS Transaction Gateway that actually ships the CICS ECI resource adapter.

The CICS ECI resource adapter internally uses the CICS Transaction Gateway in local mode to make the physical connection to CICS. It uses the same set of files as usual -- `ctgserver.jar` to provide the Java interface and `libCTGJNI.so` to make EXCI calls to CICS. See Figure 1-2.

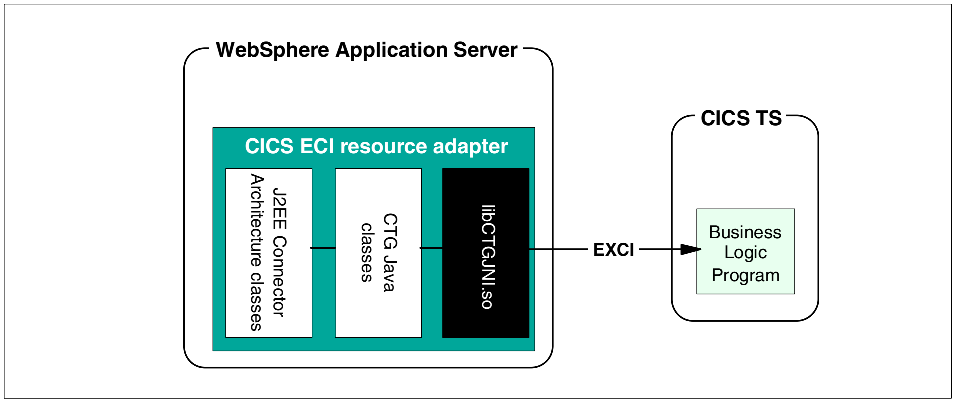

WebSphere Application Server
CICS ECI resource adapter
J2EE Connector Architecture classes
CTG Java classes
libCTGJNI.so
EXCI
CICS TS
Business Logic Program

Figure 1-2 Inside the CICS ECI resource adapter

Anything else I should know

Well, yes. The CICS Transaction Gateway ships *two* CICS ECI resource adapters:

- The original CICS ECI resource adapter fully complies with the J2EE Connector Architecture specification. It can be used in WebSphere Application Server V4 Advanced Edition on distributed platforms, or in any J2EE Connector Architecture compliant application server.

  This resource adapter has been shipped since CICS Transaction Gateway V4.0.1, and is available with the CTG on all platforms.

- A different CICS ECI resource adapter is required specifically for WebSphere on z/OS. This is because WebSphere on z/OS does not implement a fully J2EE Connector Architecture compliant environment. That's the bad news. The good news is because WebSphere on z/OS isn't constrained to the J2EE Connection Architecture specification, it can use RRS to provide two phase commit (this is not available on distributed platforms).

  This resource adapter is currently only available with the CICS Transaction Gateway V4.0.2 for z/OS.

Note: To an application programmer, the interfaces they use to work with either resource adapter are identical.

Note that WebSphere on z/OS and CICS must run within the same system.

To learn more about the J2EE Connector Architecture and the CICS ECI resource adapter, consult the IBM Redbook *Java Connectors for CICS,* SG24-6401 and *Assembling Java 2 Platform Enterprise Edition Applications,* SA22-7836.

1.2 Development tool choices

When developing an enterprise application which uses the CICS ECI resource adapter, there are two application programming tasks:

- Develop Common Client Interface code that uses the resource adapter to connector to CICS.
- Develop a session bean that uses this connector code.

IBM provides a set of development tools to help with this:

- VisualAge for Java Enterprise Edition

  This tool provides a component called the Enterprise Access Builder which can automatically generate Common Client Interface code. This code is stored within a command bean. The command bean is built using a set of wizards - no Java programming is required.
- WebSphere Studio

  Certain configurations of this tool contain a full J2EE development environment, allowing the development and test of session beans in a WebSphere environment.

This Redpaper proposes using a combination of the products to achieve the application programming tasks outlined above.

Note: Since this Redpaper was first published, WebSphere Application Server for z/OS has introduced support for enterprise services generated in WebSphere Studio Application Developer Integration Edition. This is now the strategic way to develop applications that use the CICS ECI resource adapter.

We generally recommend using this enterprise services approach ahead of the development approach described in this Redpaper. However, as a starting point, you may wish to follow the approach documented here first, before moving on to the world of enterprise services.

The application development approach described in this Redpaper is shown in Figure 1-3.

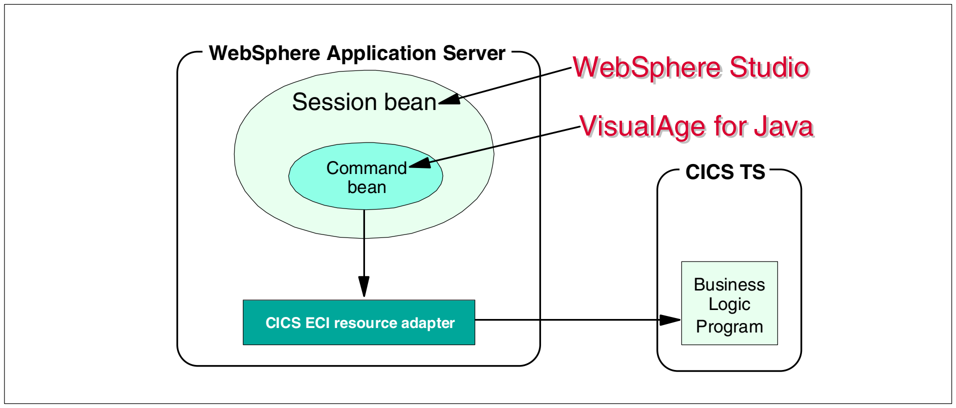


Figure 1-3 Recommended development tools

Why VisualAge for Java

If you did not use VisualAge for Java you would have to write the Common Client Interface code yourself. The Enterprise Access Builder automatically generates this code for you into a command bean. The user of the command bean can treat it as a black box; using the command bean's exposed methods to interact with the enterprise system without having to understand how the command bean internally interfaces with the CICS ECI resource adapter.

Why WebSphere Studio

IBM has significantly overhauled its development tooling, and introduced the new WebSphere Studio tool. This tool comes in several configurations, ranging from a simple Web page development configuration, to a full J2EE application development and test configuration. The following two configurations of WebSphere Studio can be used to develop J2EE enterprise applications:

- WebSphere Studio Application Developer
- WebSphere Studio Application Developer Integration Edition

Both of these configurations provide tools to generate session beans, and to test these beans in a powerful WebSphere Application Server test environment. For our purposes, the difference between these two configurations lies in the quality of the test environment:

- WebSphere Studio Application Developer

  This configuration does not contain a J2EE Connector Architecture compliant test environment. Therefore the CICS ECI resource adapter classes must be imported into the WebSphere test environment, and the connections to the resource adapter will not be managed by the test environment. This is a non-managed environment.

- WebSphere Studio Application Developer Integration Edition

  This configuration contains a J2EE Connector Architecture compliant test environment, and ships with the CICS ECI resource adapter (the non-z/OS version) already installed into the WebSphere test environment. Connections to the resource adapter are managed by the test environment. This is a managed environment.

What other options do I have

Before committing to the mix of development tools recommended by this Redpaper, you probably want to at least hear what alternatives are available. This section describes them.

The all-VisualAge for Java approach

In addition to providing the Enterprise Access Builder, VisualAge for Java provides a full EJB development and test environment. So why not use that to build session beans, and have the advantage of performing all of the application development tasks in a single tool? Here's why:

- VisualAge for Java supports the EJB 1.0 specification, whilst WebSphere supports the EJB 1.1 specification. Although the Application Assembly Tool can convert a session bean from the EJB 1.0 specification to 1.1, it is advisable to develop to the same level of specification as you intend to deploy. WebSphere Studio supports EJB 1.1.
- VisualAge for Java does not support the J2EE 1.2 packaging model of EAR and WAR files. Again this then becomes a job for the Application Assembly Tool. WebSphere Studio supports the packaging model.
- VisualAge for Java provides limited Web application support. It does not, for instance, contain an HTML editor. WebSphere Studio has dedicated Web application support.
- The VisualAge for Java EJB test environment provides limited functionality in comparison to WebSphere Studio.

The all-WebSphere Studio approach

WebSphere Studio is IBM's strategic development tool of choice, so why not use that for the entire development work? Here's why:

- Without VisualAge for Java's Enterprise Access Builder you have to write Common Client Interface code manually.
- Enterprise Access Builder provides a utility to take COBOL COMMAREAs and turn them in to J2EE Connector Architecture compliant Java records. The record classes automatically perform ASCII to EBCDIC and code page conversion. In WebSphere Studio you would have to write these records -- and the conversion routines -- yourself.

The enterprise services approach

Note: Since this Redpaper was originally published, WebSphere Application Server for z/OS has introduced support for enterprise services generated in WebSphere Studio Application Developer Integration Edition via Service Level 4 delivered in September 2002.

WebSphere Studio Application Developer Integration Edition contains an Enterprise Services component which can generate session beans or Web services that use the J2EE Connector Architecture to communicate with enterprise systems such as CICS. It also contains a Service Flow engine to group together multiple enterprise services, thereby grouping together a flow of communications to enterprise systems. This Service Flow engine is similar in concept to Navigators in the Enterprise Access Builder. WebSphere Application Server for z/OS, with Service Level 4, supports the deployment of simple enterprise services, but does not support Service Flows.

The enterprise services approach is the strategic way for generating application code for any J2EE resource adapter, including the CICS ECI resource adapter.

The Windows Notepad approach

Who needs a development environment at all? Why not just develop the entire application in a text editor? Well, where shall we start... how would you test, debug, maintain, and version this code? Hopefully this is one approach you can gleefully discount instantly!

1.3 The example used in this Redpaper

This Redpaper provides step-by-step instructions on how to:

- Prepare the development tools for use with the CICS ECI resource adapter.
- Develop and test an enterprise application that uses this resource adapter to communicate with CICS.
- Configure WebSphere on z/OS to use the CICS ECI resource adapter, and then deploy the enterprise application to WebSphere on z/OS.

To help describe this process these instructions describe how to build an example called Trader. This is a stock trading application that allows a user to buy and sell shares in numerous companies. Trader is a CICS business logic program, and this Redpaper describes how to build an enterprise application in WebSphere to drive this CICS program, using the CICS ECI resource adapter.

The completed Trader application components are shown in Figure 1-4.

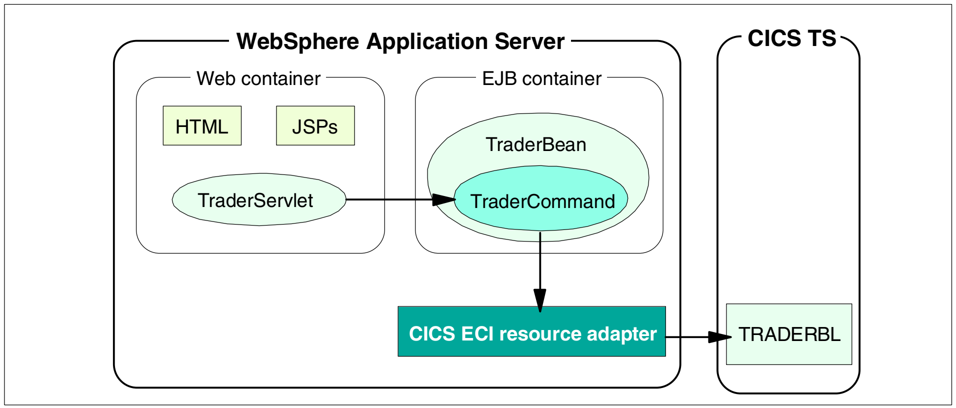


Figure 1-4 The Trader application

This Redpaper describes how to build the following components:

1. Use VisualAge for Java's Enterprise Access Builder to build a command bean (`TraderCommand`) which can link to the CICS program `TRADERBL`. This command bean uses the CICS ECI resource adapter to perform interactions with CICS.
2. Import the command bean in to WebSphere Studio and wrapper it in a session bean (`TraderBean`). Write some business methods in the session bean such as buy shares, sell shares, and get stock quotes. These business methods use the command bean to invoke business logic functions in the CICS Trader application.
3. Import a pre-built Web project into WebSphere Studio which contains a Web application to front end Trader, and a Java servlet (`TraderServlet`) that invokes the session bean.
4. Export the completed enterprise application so it can be deployed to WebSphere on z/OS.

1.4 Before you begin

If you intend to follow along with the instructions in this Redpaper, then the pre-requisites described in this section must be met.

CICS Transaction Server and CICS Transaction Gateway

- CICS Transaction Gateway V4.0.2 for z/OS must be installed and listening for TCP requests on a TCP/IP port. User authentication must be turned off.

Note: A CICS Transaction Gateway listener daemon is not used by WebSphere on z/OS, but is required to perform testing from VisualAge for Java and WebSphere Studio.

- CICS Transaction Server for z/OS V1.3 or above must be installed and running. CICS should be configured to process EXCI requests from the CICS Transaction Gateway. Security must be turned off.
- The Trader application must be installed within CICS. For instructions on how to do this, refer to Appendix A, “Trader CICS Application” on page 79.

WebSphere Application Server

- WebSphere Application Server V4.0.2 for z/OS must be installed and configured. The maintenance described in APAR PQ55873 must be applied to enable the connector support.
- A J2EE server must be configured, and must use the HTTP Listener. The connector support and Trader enterprise application will be deployed to this J2EE server.

Development tools

This Redpaper requires the use of the following development tools:

- VisualAge for Java Enterprise Edition V4.0
- A WebSphere Studio configuration. This Redpaper describes using:
  - WebSphere Studio Application Developer V4.0.3
  - WebSphere Studio Application Developer Integration Edition V4.1.1

Materials supplied with this Redpaper

To follow along with developing the Trader enterprise application you will need to obtain to obtain the additional material supplied with the Redpaper. See Appendix B, “Additional material” on page 87.

2

VisualAge for Java

This chapter describes how to prepare VisualAge for Java as a development environment for applications that use the CICS ECI resource adapter. It then provides step-by-step instructions on how to use VisualAge for Java to develop a command bean that uses the CICS ECI resource adapter.

This chapter contains the following sections:

- Overview
- Preparing VisualAge for Java
- Building a command and record bean

2.1 Overview

VisualAge for Java contains a component called the Enterprise Access Builder (EAB). It’s called the Enterprise Access Builder because it builds Java code that can access enterprise systems. The typical way to use the EAB is to have it generate a command bean. This command bean contains all of the logic necessary to communicate with an enterprise system. The user of the command bean can treat it as a black box; using the command bean’s exposed methods to interact with the enterprise system without having to understand how the command bean internally performs the enterprise access.

You can use the EAB to generate a command bean that connects to a CICS server. This command bean can link to a CICS program and exchange data, much like an EXEC CICS LINK. You can also choose which architecture the command bean internally uses to communicate with CICS:

- J2EE Connector Architecture
- Common Connector Framework

This paper focuses on using the J2EE Connector Architecture support. If the command bean internally uses this architecture it can run in a managed environment when it is deployed to WebSphere Application Server V4.0.1 for z/OS.

To build a command bean you complete a set of wizards which request properties such as the CICS server and program you wish to access. You pass data to and from the CICS program using record beans. A record bean is a Java implementation of a COMMAREA, and is typically embedded into a command bean.

If you build a record bean based on a COMMAREA with a *name* and *address* field, the record bean will provide setter methods (for example `setName()` and `setAddress()`) to populate the COMMAREA with values, and getter methods (for example `getName()` and `getAddress()`) to retrieve values from the COMMAREA. The record bean will automatically handle ASCII to EBCDIC and code page conversions internally.

This chapter describes how to install and configure VisualAge for Java to use the EAB, and then provides step-by-step instructions for building a command and record bean to use with the Trader example which we build in this Redpaper. If you follow these instructions you will build a command and record bean which you can then import into WebSphere Studio.

2.2 Preparing VisualAge for Java

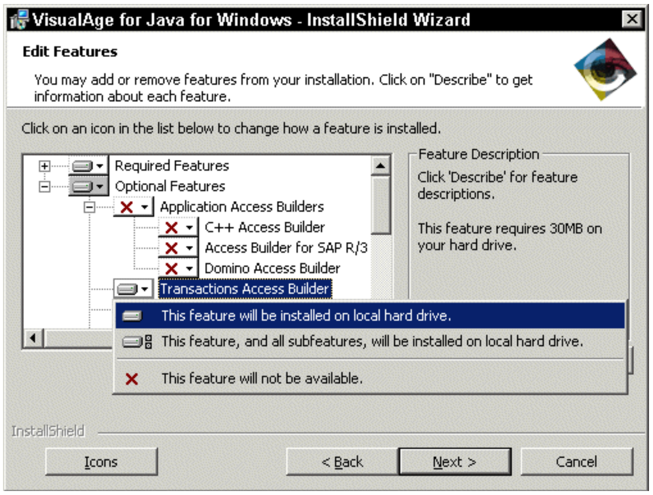
This section describes how to prepare VisualAge for Java Enterprise Edition V4.0 to enable enterprise access to CICS. Even if you have already installed VisualAge for Java it is necessary to make some changes to the base configuration to support the J2EE Connector Architecture 1.0 and CICS Transaction Gateway 4.0.2 levels of code, and an updated Enterprise Access Builder. These changes are described in this section.

2.2.1 Installing VisualAge for Java

You must install VisualAge for Java V4.0 Enterprise Edition, as this is the only version which contains the required support for the J2EE Connector Architecture. When installing VisualAge for Java, you must install the EAB support by choosing the *Transactions Access Builder* install option.

If you previously installed VisualAge for Java, but did not install the Transaction Access Builder, you can still add it by modifying your installation in the following way:

- Version any packages and projects in your workspace and then exit VisualAge for Java.
- Run `setup.exe` located in the root directory of the VisualAge for Java installation CD to start the Installer.
- Choose the **Modify** option to change your installation and click **Next**.
- Install the necessary extras by clicking on the option name and choosing **This feature will be installed on local hard drive** as shown in Figure 2-1.


VisualAge for Java for Windows - InstallShield Wizard

Edit Features

You may add or remove features from your installation. Click on "Describe" to get information about each feature.

Click on an icon in the list below to change how a feature is installed.

- Required Features
- Optional Features
  - Application Access Builders
    - C++ Access Builder
    - Access Builder for SAP R/3
    - Domino Access Builder
  - Transactions Access Builder

Feature Description

Click 'Describe' for feature descriptions.

This feature requires 30MB on your hard drive.

- This feature will be installed on local hard drive.
- This feature, and all subfeatures, will be installed on local hard drive.
- This feature will not be available.

Figure 2-1 Adding options to a VisualAge for Java installation

2.2.2 Configuring VisualAge for Java

You must configure VisualAge for Java to provide a development environment suitable for use with the CICS ECI resource adapter. The following configuration steps must be performed:

1. Update the Enterprise Access Builder (EAB) support

   By default the EAB can only generate code that adheres to the Common Connector Framework (CCF) standard. You must configure EAB to also support the J2EE Connector Architecture standard.

2. Update the level of J2EE Connector Architecture support

   Once configured, the EAB will use a beta level of the J2EE Connector Architecture classes. You must update these classes to the 1.0 specification.

3. Update the level of CICS Transaction Gateway support

   VisualAge for Java ships with CICS Transaction Gateway classes at the 3.1.2J level. This level of the CTG is no longer supported. You must update the CTG classes to the 4.0.2 level.

Step 1 - Update the Enterprise Access Builder

An updated version of the EAB that supports the J2EE Connector Architecture is supplied with the beta version of the J2EE Connectors, found on the VisualAge for Java install CD. Information about the J2EE Connector Architecture support is contained in the readme file on the install CD in the following subdirectory:

```
extras\BetaJ2EEConnectors\readme.eab
```

Note: Installing the Beta J2EE Connectors provides two components. First, the EAB tools are enhanced to support J2EE Connector Architecture compliant record and command beans. Second, classes are provided that implement the beta (proposed final draft version 2) of the J2EE Connector Architecture Specification 1.0.

The J2EE Connector Architecture Specification has been updated from proposed final draft to final version 1.0, and so these beta level classes must be updated after the EAB has been updated. CICS Transaction Gateway V4.0.2 supplies 1.0 specification classes. We show how to use these to update your workspace in the next section.

The following steps show how to install the required EAB enhancements.

1. Ensure that all of your projects or packages in the workspace are versioned.
2. Exit VisualAge for Java.
3. Run one of the following setup programs, as appropriate for your system, from the VisualAge for Java install CD and follow the installation steps.
   – `extras\BetaJ2EEConnectors\NT\Setup.exe`
   – `extras\BetaJ2EEConnectors\W2000\Setup.exe`
4. When the setup program has completed, start VisualAge for Java
5. Delete any of the following projects that may exist in your workbench. The new versions cannot be added to the workspace until the old versions have been removed.
   – Connector CICS
   – IBM Common Connector Framework
   – IBM Enterprise Access Builder Library
   – IBM Enterprise Access Builder Samples
   – IBM Enterprise Access Builder WebSphere Samples
   – IBM Java Record Library
6. Choose **File -> Quick Start -> Features -> Add Feature** and then add:
   – IBM Enterprise Access Builder Library 4.0
7. Your workbench now contains version 4.0 of the EAB, which supports the J2EE Connector Architecture.

Step 2 - Update the J2EE Connector Architecture classes

Now that you have successfully added support for the J2EE Connector Architecture to the EAB you must update the beta classes to the final 1.0 specification.

In your VisualAge for Java workbench ensure that the **Show Edition Names** button is turned on. Your workbench should contain the following project:

– J2EE Connector Architecture [Proposed Final Draft 2] 1.0

You need to replace the classes in this project with 1.0 specification classes, as described next.

Download JAR files from the CICS Transaction Gateway

The CICS Transaction Gateway V4.0.2 for z/OS ships a file called `connector.jar` which contains the J2EE Connector Architecture 1.0 specification classes. You can use this file to replace the classes in VisualAge for Java:

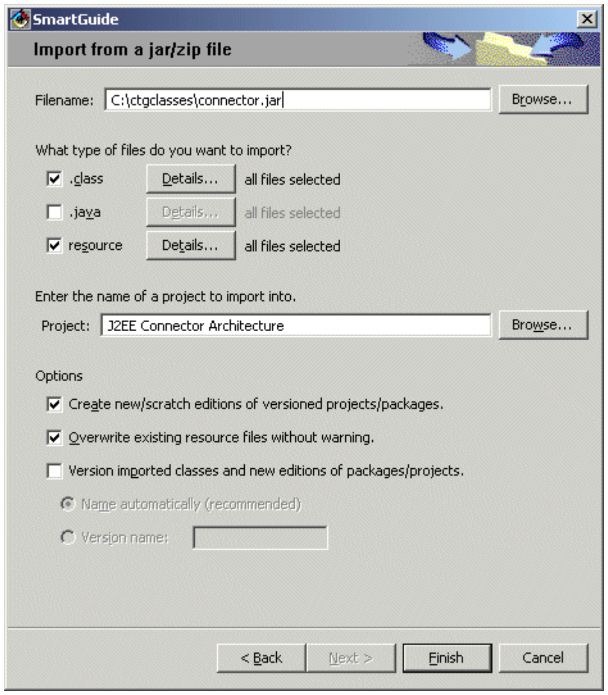
1. Use FTP to establish a connection to the z/OS system where the CICS Transaction Gateway V4.0.2 is installed.
2. Move to the `classes` directory of the CICS Transaction Gateway
   – For example: `cd /usr/lpp/ctg/classes`
3. Switch to binary mode (by typing `bin`) then download the following classes:
   – connector.jar
   – ctgclient.jar
   – ctgserver.jar
   – cicsj2ee.jar

Note: Although you only need connector.jar to update the J2EE Connector Architecture support, you will need the other JAR files in the next step.

Update the J2EE Connector Architecture project

Follow the steps below to update the J2EE Connector Architecture support:

1. Select the **J2EE Connector Architecture** project in the VisualAge for Java workbench.
2. Choose **File -> Import** from the VisualAge for Java workbench. Select **Jar file** as the import source.
3. Specify the full path to the file `connector.jar` you downloaded to your workstation, in the *Filename* box.
4. Ensure that the project to import in to is *J2EE Connector Architecture* and set the remaining import options as shown in Figure 2-2



Filename: C:\ctgclasses\connector.jar

What type of files do you want to import?

- [✓] .class — all files selected
- [ ] .java — all files selected
- [✓] resource — all files selected

Enter the name of a project to import into.

Project: J2EE Connector Architecture

Options

- [✓] Create new/scratch editions of versioned projects/packages.
- [✓] Overwrite existing resource files without warning.
- [ ] Version imported classes and new editions of packages/projects.
  - [● Name automatically (recommended)]
  - Version name:

Figure 2-2 Updating the J2EE Connector Architecture classes

5. Click **Finish** to import the new classes.
6. Version the project and its classes.
   - Right click on the **J2EE Connector Architecture** project and select **Manage -> Create Open Edition**.
   - Right click on the project again and select **Manage -> Version** Click **One Name** from the resulting dialog box, and type **1.0** into the textbox. Click **OK**. Your workspace should now contain the project *J2EE Connector Architecture* with a version number of *1.0*

Step 3 - Update the CICS Transaction Gateway classes

VisualAge for Java ships with the CICS Transaction Gateway V3.1.2J. This release is no longer supported by IBM. You should replace the V3.1.2J classes in your workbench with the CICS Transaction Gateway V4.0.2 classes, in order that your development environment matches your z/OS environment.

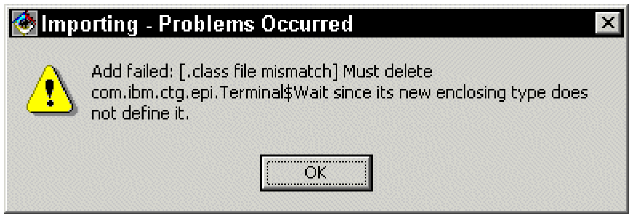
Note: There have been no API changes between CICS Transaction Gateway V3.1.2J and V4.0.2 for ECI CCI code. Therefore if you do not plan to run your code within the VisualAge for Java environment, you do not need to complete this step. However we recommend you do complete this step so you are able to perform unit testing within VisualAge for Java.

Follow the instructions below to update the CICS Transaction Gateway classes:

1. Start the VisualAge for Java workbench.
2. Delete the following projects if they exist in your workspace:
   – Connector CICS
   – Connector CICS Beta

   Note: If you do not remove the old Connector CICS classes from the workspace, you will receive an error while importing the new CTG V4.0.2 classes. If you see an error message like the one shown in Figure 2-3 then your workspace contained old CTG classes before you tried to import the new ones. If this happens, check to see if you removed the two projects from the workspace and ensure that there are no packages beginning with `com.ibm.ctg` elsewhere in the workspace.

   You can find these packages by right clicking on the projects view of your workspace and selecting **Go To -> Packages**. Selecting the package name will take you to its location in the workspace. Once you have removed these packages, start this step again and you will be able to successfully import the new classes.



```
Add failed: [.class file mismatch] Must delete
com.ibm.ctg.epi.Terminal$Wait since its new enclosing type does
not define it.
```

Figure 2-3 Error message importing CICS Transaction Gateway

3. Select **File -> Import** from the VisualAge for Java workbench. Select **Jar file** as the import source.
4. Specify the full path to the file `ctgclient.jar` (which you downloaded in the previous step) in the *Filename* box.
5. Enter the project name as **Connector CICS.**

6. Click **Finish** to import the new classes.
7. During the import a *Modify Palette* window will open. This allows you to store EPI beans in the JavaBean palette. The CICS Transaction Gateway for z/OS does not support EPI so press **Cancel** to skip this step.
8. Now import `cicsj2ee.jar` into the *Connector CICS* project by following the same procedure used to import `ctgclient.jar`. Again, select **Cancel** when the *Modify Palette* window opens.
9. Also import `ctgserver.jar` into the same *Connector CICS* project.
10. Now version the project and its classes. Right click on the **Connector CICS** project and choose **Manage -> Version**. Select **One Name** from the resulting dialog box and type `4.0.2` into the text box. Click **OK**. Your workspace should now contain project *Connector CICS* with a version number *4.0.2*

Note: You may notice some problems were generated during the imports. In the VisualAge for Java workbench, select the **All Problems** tab to see them. The problems relating to the `com.ibm.connector2.screen` package are for the EPI resource adapter, which is not supported by the CICS Transaction Gateway for z/OS. If you need to use the EPI resource adapter in VisualAge for Java, you should import `screenable.jar` into the *Connector CICS* project. The CICS Transaction Gateway V4.0.2 for distributed platforms ships this file.

You may also see some problems relating to the `javax.servlet.http` package. These problems only need to be fixed if you intend to run the CICS Transaction Gateway sample servlets in VisualAge for Java. To fix these problems import the *IBM WebSphere Test Environment* project into your workspace.

Your VisualAge for Java is now configured correctly, so you can begin building applications that use the CICS ECI resource adapter. Your workspace should contain the following projects and version numbers:

- IBM Enterprise Access Builder Library 4.0
- J2EE Connector Architecture 1.0
- Connector CICS 4.0.2

2.3 Building a command and record bean

Now VisualAge for Java has been configured, you can use it to develop record and command beans that will internally use the CICS ECI resource adapter to interact with CICS. You must generate a record bean for each COMMAREA you wish to map, and a command bean to represent each different interaction with CICS.

This section describes how to build a record and command bean to use with the Trader sample. It consists of the following steps:

1. Building the record bean
2. Building the command bean
3. Testing the command bean
4. Configuring the command bean for JNDI

2.3.1 Building the record bean

You need to generate an EAB record to provide a Java representation of the CICS COMMAREA used by the Trader application. This record bean will be used by a command bean when making interactions with CICS.

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

To generate a record bean you must locate the COBOL copybook that defines the COMMAREA you wish to map. The EAB provides a wizard to store the COMMAREA metadata in an element called a record type. A second wizard generates a record bean from this metadata (see Figure 2-4).

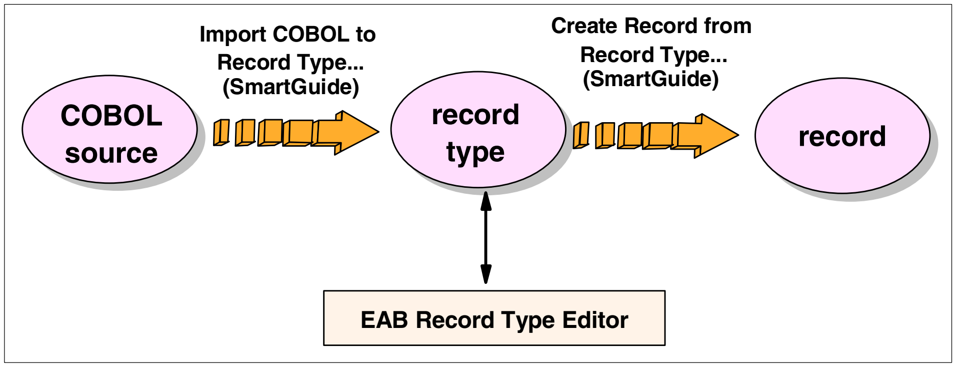

COBOL source
Import COBOL to Record Type... (SmartGuide)
record type
Create Record from Record Type... (SmartGuide)
record
EAB Record Type Editor

Figure 2-4 Generating records in the EAB

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

Create a record type

To build a record type to use with the Trader application, perform the following:

1. In VisualAge for Java, select **Workspace -> Tools -> Enterprise Access Builder -> Import COBOL to Record Type**. The *Import COBOL to Record Type* SmartGuide will open to guide you through the process.
2. Provide the name of the COBOL file containing the COMMAREA declaration. Use the file `commarea.txt` which is supplied with the additional materials accompanying this Redpaper. Select **A CICS Transaction** as the code to be imported. Then click **Next**.
3. Highlight **COMMAREA-BUFFER** in the *Available level 01 commareas* pane then press the **>** button to move it to the *selected commareas* pane. Then click **Next**.
4. To name and store the record type enter the following information:

Project:	**Trader Sample**
Package:	**itso.cics.eci.j2ee.trader**
Class name:	**TraderRecordType**

   Ensure **Continue working with newly created record type** is checked, and **Create record from record type** is selected. Click **Finish** to generate the record type. The *Create Record from Record Type* SmartGuide will be started.

Create a record from the record type

Perform the following:

1. The *Create Record from Record Type* SmartGuide should now be open. The project and package text boxes are already filled in. Enter the following information:

   a. You must provide a name for your record class. Name it **TraderRecord**.

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

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

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

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

   f. Be sure to select **Generate as javax.resource.cci.Record interface**. This ensures the EAB builds a record that internally uses the J2EE Connector Architecture. When finished, click **Next.**

2. Configure how data conversion is to be performed with the following steps:

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

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

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

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

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

A record bean is now generated. In your workbench, you should see the following classes in the `itso.cics.eci.j2ee.trader` package of the *Trader Sample* project:

– TraderRecord
– TraderRecordBeanInfo
– TraderRecordType

The *TraderRecord* class is the record bean. The *TraderRecordType* class contains the metadata used to create the record, but is not required at runtime. The *TraderRecordBeanInfo* class is only required in a JavaBeans environment.

2.3.2 Building the command bean

You need to generate an EAB command bean to manage the interaction with CICS. The command bean specifies how to connect to CICS, and what to do once the connection is made. It exposes methods of embedded record beans to set and get data to send to CICS.

To generate a command bean to use with the Trader sample, perform the following:

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

2. Set the following in the SmartGuide:

   a. Select the **Trader Sample** project and the **itso.cics.eci.j2ee.trader** package as the place to create the new command bean. Name the command bean **TraderCommand.**

b. In the *Connection Information* section select **Browse**. From the list of connection factories select **ConnectionFactoryConfiguration** from the **`com.ibm.ivj.eab.command`** package then press **OK**. This class will be used to configure the connection to the CICS server.

> **Attention:** The value you provide for the connection information determines the architecture the command bean will internally use. The *ConnectionFactoryConfiguration* class tells the command bean to use the J2EE Connection Architecture. The *CICSConnectionSpec* class tells the command bean to use the Common Connector Framework.

c. In the *InteractionSpec* section select Browse, highlight **ECIInteractionSpec** from the **`com.ibm.connector2.cics`** package, then press **OK**.

> **Note:** The only interaction spec classes listed are those from the J2EE Connector Architecture support. If you had selected different connection information then you would be presented with a different list of classes.

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

3. In the *Add Input/Output Beans* window, note the *Implements java.resource.cci.Record* checkbox at the top is checked, indicating that you are building a J2EE Connector Architecture compliant command bean. You now need to tell the command bean which record beans it should use when interacting with CICS:

   a. First, click the **Browse** button next to the text box labeled *Class name* and select the **TraderRecord** class from the **`itso.cics.eci.j2ee.trader`** package. This is the record bean generated in the previous section.

   b. The format of the COMMAREA used by the Trader application is identical for both input and output, therefore the same record bean can be used for both operations. Select **Use input bean type as output bean type**.

   c. Click **Finish** to generate the command bean and invoke the EAB *Command Editor.*

4. At this stage, the command bean has been generated. You now set the system specific properties of the command bean. First set the connection information fields in the Command Editor:

Note: You will initially hard code connection information into the command bean. This is for the purposes of unit test. Before exporting the command bean from VisualAge for Java you will need to configure it to lookup this connection information from JNDI instead, as described in “Configuring the command bean for JNDI” on page 26.

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

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

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

   i. Set the *connectionURL* to point to the gateway daemon. We used `tcp://wtsc58oe.itso.ibm.com` for our CTG. If the protocol is omitted, `tcp` is assumed.

   ii. Set the *portNumber* if you are not using the default of 2006 for the tcp protocol.

   iii. Set the *serverName* to the CICS server name where the Trader application will run. We used SC58CIC2.

5. Set the interaction spec properties. To do this click on **com.ibm.connector2.cics.ECIInteractionSpec** in the top right pane. We want to call the CICS Trader application, so set the *functionName* property to **TRADERBL**.

6. You must expose any record fields that you wish the user of the command bean to be able to get or set. This is performed by promoting record properties for both the input and output records:

   a. In the top left pane select **Input** and in the top right pane select i**tso.cics.eci.j2ee.trader.TraderRecord**. This gives a list of all the properties in the input record used by the command bean. To promote a specific property right click on it and select **Promote Property**. The Trader enterprise application requires the following input record properties to be promoted:

   - company__Name
   - no__Of__Shares__Dec
   - request__Type
   - update__Buy__Sell

- userid

b. Next promote properties on the output record. In the top left pane select **Output** and in the top right pane select i**tso.cics.eci.j2ee.trader.TraderRecord**. The Trader enterprise application requires the following output record properties be promoted:

  - commission__Cost__Buy
  - commission__Cost__Sell
  - company__Name__Tab
  - no__Of__Shares
  - total__Share__Value
  - unit__Share__Price
  - unit__Value__1__Days through unit__Value__7__Days

7. Save the command bean by selecting **Command -> Save**. Your command bean is now built and ready to test.

2.3.3 Testing the command bean

The EAB Test Client is a utility which allows you to test a command bean without having to write any code to invoke it. This section assumes your CICS server and CICS Transaction Gateway are running, and the Trader application has been installed in CICS. Follow the steps below to test TraderCommand:

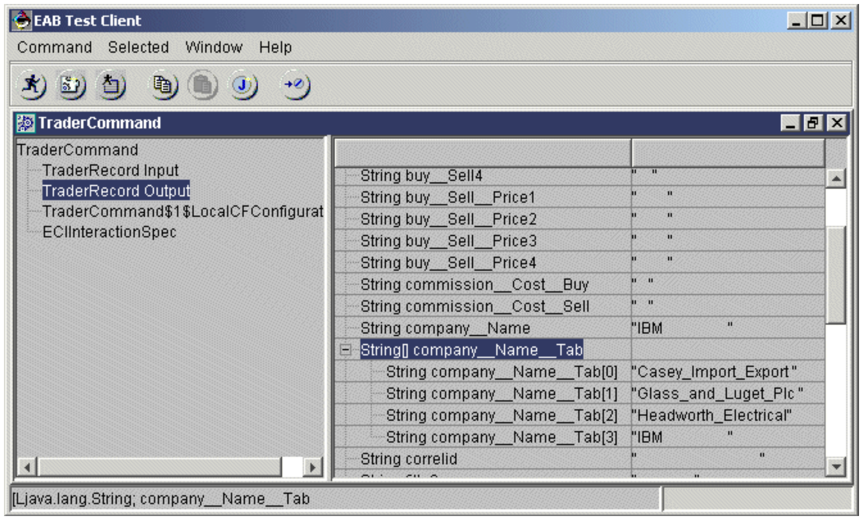
1. The EAB Test Client can be launched in two ways:
   - If you still have the Command Editor open select **Command -> Run Test Client**.
   - Alternatively select **Workspace -> Tools -> Enterprise Access Builder -> Launch Test Client**.
2. Depending upon how the EAB Test Client was started, a new instance of TraderCommand may be displayed in the EAB Test Client window. If it is not, then select **Command -> Create new instance**. Choose class **TraderCommand** from the `itso.cics.eci.j2ee.trader` package then click **OK**.
3. Highlight **TraderCommand Input** in the left pane. This displays the properties of the input record used by the command bean. You can manually set the values of these fields here, and these values will be placed in the COMMAREA sent to CICS. To test TraderCommand:

   a. Locate the *request__Type* property and click to the right of it to enter edit mode. Enter a value of **Get_Company**.

   b. Click on the **request__Type** property itself to come back out of edit mode and save the change made.

c. Click on the running man icon in the top left of the EAB Test Client window to invoke the command. The command will now attempt to invoke the Trader application in CICS, and populate the output record with the values returned in the CICS COMMAREA.

d. When control is returned, the EAB Test Client will automatically switch to the *TraderCommand Output* view. You can check the contents of the COMMAREA returned by CICS from here. Locate the *company__Name__Dec* property and expand it by clicking on **+** to the left of the property name. This property should contain a list of four companies as shown in Figure 2-5.



```
TraderCommand
  TraderRecord Input
  TraderRecord Output
  TraderCommand$1$LocalCFConfigurat
  ECIInteractionSpec
```

```
String buy__Sell4                  "  "
String buy__Sell__Price1           "     "
String buy__Sell__Price2           "     "
String buy__Sell__Price3           "     "
String buy__Sell__Price4           "     "
String commission__Cost__Buy       " "
String commission__Cost__Sell      " "
String company__Name               "IBM           "
String[] company__Name__Tab
   String company__Name__Tab[0]    "Casey_Import_Export"
   String company__Name__Tab[1]    "Glass_and_Luget_Plc"
   String company__Name__Tab[2]    "Headworth_Electrical"
   String company__Name__Tab[3]    "IBM           "
String correlid                    "                "
```

```
[Ljava.lang.String; company__Name__Tab
```

Figure 2-5 A successful invocation using the EAB Test Client

2.3.4 Configuring the command bean for JNDI

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

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

The command bean you have generated contains code that implements the former. However, the J2EE Connector Architecture specification requires that all applications deployed to a managed environment (such as WebSphere Application Server) must use JNDI.

This section describes how to modify the command bean to use JNDI to lookup a connection factory and details an optional step to test the command bean with JNDI.

Modifying the command bean

Before exporting your command bean you must change the connection information to use JNDI to lookup the connection factory by performing the following steps:

1. If the Command Editor is not already open, right click on the **TraderCommand** class and select **Tools -> Enterprise Access Builder -> Edit Command** to open it.
2. Highlight **Connector** in the left pane, and **ConnectionFactoryConfiguration** in the right pane. Set the following properties:
   a. As we are using JNDI we do not need a `ManagedConnectionFactory` object. Right click on *managedConnectionFactory* and select **Reset Property**. This sets the value of this property to null.
   b. Set *contextFactoryName* to **com.ibm.websphere.naming.WsnInitialContextFactory**. This is the JNDI context factory used by WebSphere Application Server.
   c. Set *res_ref_name* to **eis/CICS_ECI**. This is where the connection factory will be bound in the JNDI name space. The name server in WebSphere Application Server requires a subcontext, such as **eis/**.
   d. Set *res_type* to **com.ibm.connection2.cics.ECIConnectionFactory**. Click on the *res_type* property itself to come out of edit mode and ensure your change is saved.
3. Save the modified command bean by selecting **Command -> Save**.

Testing the command bean with JNDI (optional)

You can test the JNDI lookup functionality of the command bean in VisualAge for Java. Note however, that VisualAge for Java uses a different JNDI implementation to WebSphere Application Server so requires changes to the command bean JNDI settings.

Note: This section is optional. If you wish to skip it, move on to “Export the command and record bean” on page 30.

Binding a connection factory

Before testing the JNDI lookup in your command bean you must first create and bind a `ConnectionFactory` to the JNDI namespace, so that the command bean can look it up.

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

1. Add the *IBM WebSphere Test Environment* project to the workbench by selecting **File -> Quick Start**. Then **Features -> Add Feature**. Select **IBM WebSphere Test Environment** from the list and press **OK**. This project, and some related projects the test environment requires, will be imported into the workbench.
2. You also need to add the *IBM Enterprise Access Builder WebSphere Samples* project to the workbench. To do this:
   a. Select **Window -> Repository Explorer**.

   b. Select the **Projects** tab and locate and click on the **IBM Enterprise Access Builder WebSphere Samples** project.

   c. In the *Editions* pane right click on **3.5.3.5** and select **Add to Workspace**.

> **Note:** You cannot import the *IBM Enterprise Access Builder WebSphere Samples* project by adding this feature using the Quick Start menu. The Add Feature wizard will attempt to also import the *Connector CICS Beta* project, and this import will fail with the error:
>
> ```
> Add failed: The package com.ibm.connector.cics cannot be
> specified in both Connector CICS and Connector CICS Beta
> ```

3. In the *IBM Enterprise Access Builder WebSphere Samples* project, open the `com.ibm.ivj.eab.sample.ws.j2ee.servlet.LookupDeployerHelper` class. This class is used to both bind and lookup references in the namespace.
4. Examine the `createBinding()` method of `LookupDeployerHelper`. This method allows you to specify values for the `ECIManagedConnectionFactory` (which is used to create a `ConnectionFactory` object) and the `DefaultConnectionPoolProperties`, which is used to configure connection pooling. Use the setter methods of these classes to set deployment values, including:
   a. `setServerName()`, which sets the name of the CICS server to call

   b. `setConnectionURL()`, which specifies the connection URL of the CTG

   c. `setPortNumber()` if not using port 2006 with protocol tcp.

Additionally, you can customize the resource reference name by modifying the `rebind()` method of the `ctx` object. The supplied code sets this to CICSECI_A. Unlike the JNDI implementation in WebSphere Application Server, the resource reference name cannot contain a slash (for example `eis/CICS_ECI` is not permitted).

Save the changes made to the `LookupDeployerHelper` class.

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

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

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

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

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

   – A stack trace indicates an exception was thrown and the operation was not successful.

   – A message similar to the one shown in Example 2-1 indicates that the resource reference name was successfully bound to the namespace.

Example 2-1 Output from the JNDIDeployer servlet

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

Looking up a connection factory

To test if the command bean can lookup the connection factory, perform the following:

1. Start the EAB Test Client again, and create a new instance of TraderCommand.
2. Highlight **ConnectionFactoryConfiguration** to display the connection information parameters. It is necessary to change the following two parameters in order to lookup an object using the VisualAge for Java JNDI implementation:
   a. Change *contextFactoryName* to **com.ibm.ejs.ns.jndi.CNInitialContextFactory.**
   b. Change *res_ref_name* to **CICSECI_A**.

   This will only change these values for the lifetime of this TraderCommand instance. It will not change the TraderCommand properties permanently.
3. Highlight TraderRecordInput and set *request__Type* to **Get_Company** as before. Remember to come out of edit mode so your change is saved.
4. Invoke the command bean (by clicking on the running man) and check the output record for a list of companies in the *company__Name__Dec* property. If you see a list of companies as before then the JNDI lookup worked.

You should now stop the Persistent Name Server and Servlet Engine.

Export the command and record bean

In this chapter you have built a command and record bean which internally uses the J2EE Connector Architecture to communicate with CICS. You need to make this command and record bean accessible to WebSphere Studio, so you must export them. It is also a good idea to version your current project.

Perform the following:

1. In the VisualAge for Java workbench right click on the **Trader Sample** project and select **Manage -> Version**. The *Versioning Selected Items* window opens. Go with the default of *One name 1.0* and press **OK**. The Trader Sample project, and all classes within it, and now versioned.
2. Right click on the **Trader Sample** project again and select **Export**. This opens the Export Smartguide.
   a. Select **Directory** then press **Next**.
   b. Specify a directory to export to (we used `c:\vajoutput`). Under the *what do you want to export* section ensure only **java** is selected, and press the **Details** button next to *java*. Make sure only **TraderCommand** and

TraderRecord are selected (you will not need the other classes in WebSphere Studio). Press **OK** to return, then press **Finish**.

c. The source code for TraderCommand and TraderRecord will be exported to the directory you specified. Remember where this directory is, as you will need it in the next chapter.

You have now completed the VisualAge for Java chapter. Move on to the next chapter to use WebSphere Studio to build a session bean which uses the command and record bean you have generated.

3

WebSphere Studio

This chapter describes how to use WebSphere Studio to prepare an environment suitable to develop and test applications that use the CICS ECI resource adapter. It then describes how to import a command bean built in VisualAge for Java, and how to build an enterprise application that wrappers this command bean. The enterprise application is then exported, in preparation for deployment to WebSphere for z/OS.

Two configurations of WebSphere Studio are discussed in this chapter:

- WebSphere Studio Application Developer
- WebSphere Studio Application Developer Integration Edition

This chapter contains the following sections:

- Overview
- Building the Trader enterprise application

3.1 Overview

This chapter describes how to use WebSphere Studio to build a Trader enterprise application that interacts with CICS. This enterprise application contains a session bean that uses the command and record beans generated in the previous chapter to perform the communication with the Trader application in CICS.

WebSphere Studio has several configurations. This chapter describes using the following configurations:

- WebSphere Studio Application Developer V4.0.3
- WebSphere Studio Application Developer Integration Edition V4.1.1

For the most part, the procedure to develop the Trader enterprise application is identical for both configurations. Where there are differences, these are explained and separate steps are given for each configuration.

Note: The use of WebSphere Studio Application Developer Integration Edition does not imply we are developing enterprise services to communicate with CICS.

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

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

3.2 Building the Trader enterprise application

This section provides step-by-step instructions for building and testing the Trader enterprise application. It consists of the following steps:

1. Preparing WebSphere Studio
2. Creating a session bean
3. Testing the session bean
4. Completing the enterprise application

3.2.1 Preparing WebSphere Studio

This section contains the following steps:

1. Creating an enterprise application project for the Trader application
2. Setting the Java build path for the EJB project

   Note: This step differs depending upon the configuration of WebSphere Studio you are using. Separate instructions are given for Application Developer and Application Developer Integration Edition.

3. Importing the command and record into the EJB project
4. Importing data bean classes required by the Trader Web application

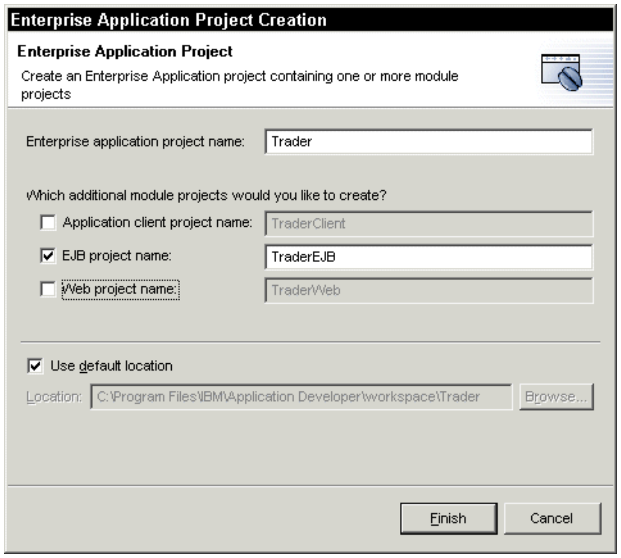
Once you have completed this section you will be ready to develop a session bean for the Trader application that interacts with CICS.

Creating an enterprise application project

An enterprise application project is used to associate multiple projects with a single enterprise application. An enterprise application project will typically be associated with an EJB project (which contains enterprise beans) and a Web project (which contains servlets, JSPs, and static content).

The following steps describe how to create an enterprise application project for Trader:

1. Start WebSphere Studio. Switch to the J2EE perspective by choosing **Perspective -> Open -> J2EE** from the menus.
2. Choose **File -> New -> Enterprise Application Project**. This opens the *Enterprise Application Project Creation* dialog box. At this point you can specify which sub-projects the enterprise application project will contain. Fill in the dialog as shown in Figure 3-1. Click **Finish** to create the project in your workspace.


Enterprise Application Project Creation

Enterprise Application Project

Create an Enterprise Application project containing one or more module projects

Enterprise application project name: Trader

Which additional module projects would you like to create?

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

- [x] Use default location

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

Figure 3-1 Creating an enterprise application project

Note: This enterprise application project is initially created containing only an EJB project for the session bean. Once the session bean is complete, you will later add a Web project to the enterprise application by importing the pre-existing Web components from a WAR file.

Setting the Java build path

Now that the projects have been created, it is necessary to add some JAR files to the Java build path, which contains the classes that your project will be using. You need to include support for the following components:

- J2EE Connector Architecture
- CICS ECI resource adapter
- Enterprise Access Builder

The method you use to do this depends on the configuration of WebSphere Studio you are using:

- WebSphere Studio Application Developer does not contain runtime support for any of these components, so JAR files must be imported from external sources.
- WebSphere Studio Application Developer Integration Edition contains runtime support for all of these components, so ships with all of the required JAR files.

Follow the relevant instructions for the WebSphere Studio configuration you are using.

WebSphere Studio Application Developer

Important: If you are using WebSphere Studio Application Developer Integration Edition you should not complete this section.

You need to download the following classes to your workstation. Use FTP to download these files in binary mode.

- To obtain the necessary J2EE Connector Architecture and CICS ECI resource adapter classes, download the following JAR files from your CICS Transaction Gateway installation (typically `/usr/lpp/ctg`):
  - `classes/connector.jar`
  - `classes/cicsj2ee.jar`
- To obtain the necessary Enterprise Access Builder classes, download the following JAR file from the WebSphere Application Server installation (typically `/usr/lpp/WebSphere`):
  - `lib/bboaxrt.jar`

  Note: Most of the Enterprise Access Builder classes can alternatively be found in `eablib.jar` and `recjava.jar` from the VisualAge for Java installation. However, `bboaxrt.jar` also contains classes belonging to the `com.ibm.connector` package which are also required on the Java build path, hence this JAR file is used instead.

The following steps describe how to add these JAR files to the TraderEJB project:

1. Open the *Navigator* view from the J2EE Perspective page by choosing **Perspective -> Show View -> Navigator** from the menus.
2. Right click on the **TraderEJB** project in the *Navigator* view and select **Properties**.
3. Choose **Java Build Path** from the options list on the left of the dialog box, and then select the **Libraries** tab from the right hand panel.
4. Click the **Add External JARs** button. Locate the `connector.jar`, `cicsj2ee.jar`, and `bboaxrt.jar` files you downloaded earlier, select each of them, and press **Open**.
5. When all three JAR files have been added to the Java build path, press **OK**.

WebSphere Studio Application Developer Integration Edition

Important: If you are using WebSphere Studio Application Developer you should not complete this section.

The following steps describe how to add the necessary JAR files to the TraderEJB project:

1. Open the *Navigator* view from the J2EE Perspective page by choosing **Perspective -> Show View -> Navigator** from the menus.
2. Right click on the **TraderEJB** project in the *Navigator* view and select **Properties**.
3. Choose **Java Build Path** from the options list on the left of the dialog box, and then select the **Libraries** tab from the right hand panel.
4. Click the **Add Variable** button. Perform the following:
   - Next to the *Variable Name* field press **Browse**.
   - From the *Variable Selection* window highlight **WAS_PLUGIN** and press **OK**. This environment variable points to the directory where the WebSphere Test Environment runtime classes are stored.
   - Next to the *Path Extension* field press **Browse**, move to the `lib` directory, select `eablib.jar`, and press **Open**. This adds `eablib.jar` to the Java build path. Press **OK**.
5. Complete the same process for the following JAR files, with each one using the WAS_PLUGIN variable as before:
   - `lib/recjava.jar`
   - `lib/jca.jar`
   - `lib/ccf.jar`
6. We also need to add some CICS ECI resource adapter classes. To do this click on **Add JARs**. Expand **Installed Resource Adapters -> CICS ECI**. Highlight `cicseci.jar` and `cicsframe.jar` and press **OK**.
7. Now all of the JAR files have been added to the Java build path, press **OK**.

Importing the command and record into the EJB project

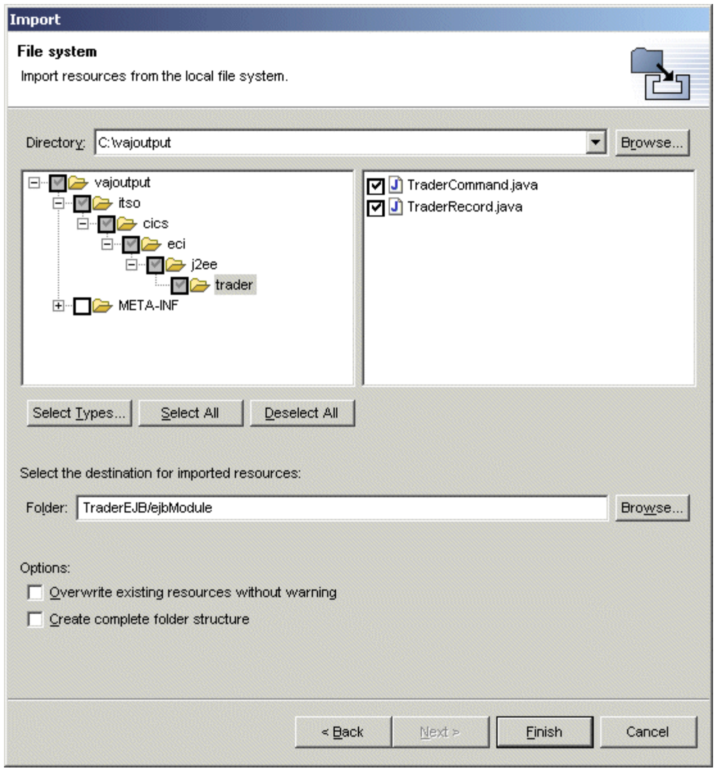
Important: Complete this, and all further, sections regardless of the configuration of WebSphere Studio you are using.

Now the TraderEJB project Java build path has been configured, you can import the command and record bean you built in VisualAge for Java into the TraderEJB project. Follow the steps below:

1. From the *J2EE* perspective *Navigator* view, click on the **TraderEJB** project. Choose **File -> Import** from the menu system.
2. Select **File system** as the import source and click **Next**.
3. Click **Browse** to locate the directory where you exported `TraderCommand.java` and `TraderRecord.java` as described in the previous chapter. Once you have located the directory, click **OK**.
4. Expand the directory name on the left side of the panel, and expand `itso -> cics -> eci -> trader`. Click on the `trader` folder, and then check the `TraderCommand.java` and `TraderRecord.java` boxes on the right side of the panel. Set the folder name to:

   ```
   TraderEJB/ejbModule
   ```

   Set all of the other remaining options as shown in Figure 3-2, then click **Finish** to complete the import.


Import

File system

Import resources from the local file system.

Directory: C:\vajoutput

- ☑ vajoutput
  - ☑ itso
    - ☑ cics
      - ☑ eci
        - ☑ j2ee
          - ☑ trader
  - ☐ META-INF

☑ TraderCommand.java
☑ TraderRecord.java

Select the destination for imported resources:

Folder: TraderEJB/ejbModule

Options:

☐ Overwrite existing resources without warning
☐ Create complete folder structure

Figure 3-2 Import TraderCommand and TraderRecord

Importing data bean classes

Two data bean classes are used by the session bean you will write. They are used to store data which is used by the Trader Web application. Follow the same procedure as above to import these extra classes into the `itso.cics.eci.j2ee.trader` package. These files are provided with the sample material shipped with this Redpaper:

- `CompaniesBean.java`
- `QuotesBean.java`

3.2.2 Creating a session bean

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

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

You need to follow these steps to create this session bean:

1. Add a new session bean to the TraderEJB project.
2. Add the business methods that use the command bean to connect to CICS.
3. Edit the EJB deployment descriptor
4. Generate bean deployment code

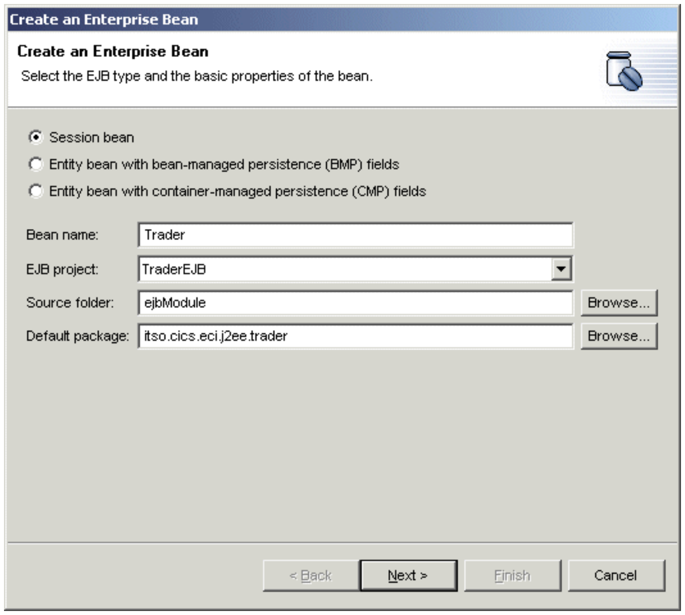
Adding a new session bean

Follow the steps below to create the basic framework for the session bean in WebSphere Studio:

1. From the J2EE perspective *Navigator* view, right click on the **TraderEJB** project and choose **New -> Enterprise Bean**. This opens the *Create EJB* dialog:
   a. Select **Session bean** as the enterprise bean type
   b. Enter the bean name as **Trader**

c. Enter a default package of **itso.cics.eci.j2ee.trader**. The completed wizard should look the same as Figure 3-3.

d. Click **Next** then **Finish** to create the session bean.


Create an Enterprise Bean

Create an Enterprise Bean
Select the EJB type and the basic properties of the bean.

(•) Session bean
( ) Entity bean with bean-managed persistence (BMP) fields
( ) Entity bean with container-managed persistence (CMP) fields

Bean name: Trader
EJB project: TraderEJB
Source folder: ejbModule
Default package: itso.cics.eci.j2ee.trader

Figure 3-3 Create Enterprise Bean wizard

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

Adding code to the session bean

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

Note: This section describes how to create a stateful session bean. You could alternatively create a stateless session bean, and modify `TraderServlet` to store state instead. The state is this instance is the userid.

Adding session bean fields

Add the following fields at the top of your session bean source code, as shown in Figure 3-4.

```
public class TraderBean implements javax.ejb.SessionBean {

   private TraderCommand traderCommand = null;
   private static final int NUM_OF_COMPANIES=4;
   private String ivUserid;
   private String ivPassword;
```

Figure 3-4 Session bean fields

Notice this defines an instance of TraderCommand called *traderCommand*.

Modifying the ejbCreate() method

When the session bean is started we want to instantiate the TraderCommand object, so it can be used later by the numerous business methods. If we did not instantiate TraderCommand in the `ejbCreate()` method we would potentially have to instantiate it every time a business method was called. Enter the line shown in Figure 3-5 in the ejbCreate() method.

```
public void ejbCreate() throws javax.ejb.CreateException {
   traderCommand = new TraderCommand();
}
```

Figure 3-5 The ejbCreate() method

Note: Instantiating a command bean does not result in a connection to CICS.

Adding the logon() and logoff() business methods

The first of the business methods to add are `logon()` and `logoff()`. Neither of these methods require an interaction with CICS, so we do not need to use the TraderCommand command bean. Enter the code shown in Figure 3-6.

```
public void logon(String userid, String password) {
   ivUserid = userid;
   ivPassword = password;
}

public void logoff() {
}
```

Figure 3-6 The logon() and logoff() methods

The `logon()` method stores the userid and password sent to it in the fields we defined earlier and the `logoff()` method has no function, but is required by the Trader Web application.

Adding the getCompanies() business method

This business method is used to retrieve a list of companies which can be used with the Trader application. It is of particular interest to us because it requires an interaction with the Trader application in CICS, hence it uses the TraderCommand command bean. Enter the code shown in Figure 3-7.

```
public CompaniesBean getCompanies() throws Exception {
   1  CompaniesBean companies = new CompaniesBean();
      try{
      2  traderCommand.setRequest__Type("Get_Company");
      3  traderCommand.execute();
      } catch (Throwable t) {
         t.printStackTrace();
         throw new Exception("Error getting companies: " + t.getMessage());
      }
      for (int i=0; i < NUM_OF_COMPANIES; i++) {
      4  companies.addCompany(traderCommand.getCompany__Name__Tab(i));
      }
      return companies;
}
```

Figure 3-7 The getCompanies() method

- 1 CompaniesBean is used to store the returned list of companies for a given company. It is used by the Trader Web application to display these values.
- 2 The instance of TraderCommand is used here. Before sending a call to CICS this line of code sets the *request__Type* property of the input record. The `setRequest__Type()` method is exposed by the command bean because we explicitly promoted the *request__Type* property of the input record in the EAB Command Editor.

Note: The `setRequest__Type()` method contains a double underscore.

- **3** The `execute()` method of the TraderCommand object makes the interaction with CICS, passing the values specified in the input record in a COMMAREA. This is a synchronous call, so control is not returned to the session bean until CICS has responded.
- **4** The output record in the TraderCommand object contains the values the Trader application populated in the output COMMAREA. The `getCompany__Name__Tab()` method retrieves a specific company name from an array. The getCompany__Name__Tab() method is exposed by the command bean because we explicitly promoted the *company__Name__Tab* property of the output record in the EAB Command Editor.

Adding the getQuotes() business method

This business method is used to retrieve stock information for a specific company. It uses the TraderCommand object in much the same way. It sets the request type, company name, and userid in the input record, and then after passing these to CICS, retrieves the returned stock values from the output record. Enter the code shown in Figure 3-8, noting the use of the TraderCommand object.

```
public QuotesBean getQuotes(String company) throws Exception {
   QuotesBean quotes = new QuotesBean();
   try{
      traderCommand.setRequest__Type("Share_Value");
      traderCommand.setCompany__Name(company);
      traderCommand.setUserid(ivUserid);
      traderCommand.execute();
   } catch (Throwable t) {
      t.printStackTrace();
      throw new Exception("Error getting quotes: " + t.getMessage());
   }
   quotes.setUnitSharePrice(traderCommand.getUnit__Share__Price());
   quotes.setUnitValue1Days(traderCommand.getUnit__Value__1__Days());
   quotes.setUnitValue2Days(traderCommand.getUnit__Value__2__Days());
   quotes.setUnitValue3Days(traderCommand.getUnit__Value__3__Days());
   quotes.setUnitValue4Days(traderCommand.getUnit__Value__4__Days());
   quotes.setUnitValue5Days(traderCommand.getUnit__Value__5__Days());
   quotes.setUnitValue6Days(traderCommand.getUnit__Value__6__Days());
   quotes.setUnitValue7Days(traderCommand.getUnit__Value__7__Days());
   quotes.setCommissionCostBuy(traderCommand.getCommission__Cost__Buy());
   quotes.setCommissionCostSell(traderCommand.getCommission__Cost__Sell());
   quotes.setNumberOfShares(traderCommand.getNo__Of__Shares());
   quotes.setTotalShareValue(traderCommand.getTotal__Share__Value());
   return quotes;
}
```

Figure 3-8 The getQuotes() method

Adding the buy() and sell() business methods

These business methods are used to buy and sell shares in a specific company. They both require an interaction with CICS, but this interaction is nearly identical for both methods. Therefore to reduce the amount of code required, both the `buy()` and `sell()` methods use a private method called `trade()` to perform the interaction with CICS. Enter the code shown in Figure 3-9.

```
public void buy(String company, int numberOfShares) throws Exception {
   trade(company, numberOfShares, true);
}

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

private void trade(String company, int numberOfShares, boolean buy) throws
Exception {
   try{
      traderCommand.setRequest__Type("Buy_Sell");
      traderCommand.setCompany__Name(company);
      traderCommand.setUserid(ivUserid);
      traderCommand.setNo__Of__Shares__Dec((short)numberOfShares);
      if (buy)
         traderCommand.setUpdate__Buy__Sell("1");
      else
         traderCommand.setUpdate__Buy__Sell("2");
      traderCommand.execute();
   } catch (Throwable t) {
      t.printStackTrace();
      throw new Exception("Error performing trade: " + t.getMessage());
   }
}
```

Figure 3-9 The buy() and sell() methods

Your session bean now contains all of the code it requires. Select **File -> Save** to save your changes.

Promoting the business methods

In order to make the business methods externally visible to the user of the session bean, each business method must be promoted to the remote interface. Add the `buy()`, `getCompanies()`, `getQuotes()`, `logoff()`, `logon()`, and `sell()` business methods to the remote interface by right click on each method name in the *Outline* view panel, and choose **Enterprise Bean -> Promote to Remote Interface**.

Editing the EJB deployment descriptor

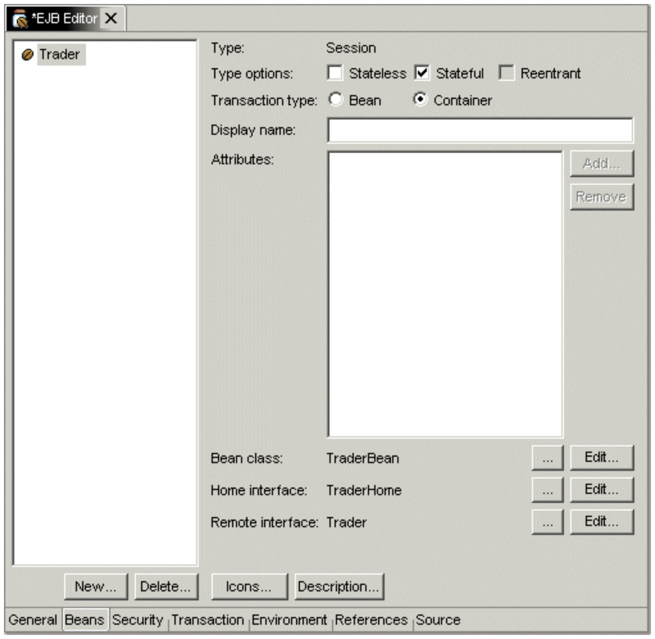
Now that the session bean is complete, you must make some changes to its deployment descriptor. The following steps explain how to make these changes:

Note: Deployment information can also be set and changed in the Application Assembly Tool, which is used before deploying to WebSphere Application Server for z/OS.

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

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

2. Select **Open With -> EJB Editor** from the popup menu.
3. This launches the deployment descriptor editor. Click the **Beans** tab at the bottom of the editor.
4. Now click on the **Trader** bean in the left panel of the editor to display the bean properties. You have created a stateful session bean, so change the type option to **Stateful** by clicking the checkbox to ensure that all the remaining values are as shown in Figure 3-10.



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

Figure 3-10 Setting the bean properties

5. Now click on the **References** tab and select the **Resource references** radio button. Click on the **Trader** entry in Resource Name column in the table. Click the **Add** button to the right of the table.
6. Complete each of the columns with the values shown below:
   - Resource name: `eis/CICS_ECI`
   - Type: `com.ibm.connector2.cics.ECIConnectionFactory`
   - Authentication: `Application`
   - Description: `CICS ECI resource adapter`

   Important: The Resource Name value must match the *res_ref_name* property which was set in the connection information of the TraderCommand command bean.

7. Press **Ctrl-S** to save the changes to the `ejb-jar.xml` file and close the editor.
8. Now right click on the **ejb-jar.xml** file again and this time select **Open With -> EJB Extension Editor**. This will launch the *EJB Extension Editor*, which allows you to add extra deployment information to the EJB JAR that is not contained in the standard deployment descriptor.
9. Click on the **Bindings** tab at the bottom of the editor and expand the tree under *TraderEJB* to display the *Trader* session bean. Select the **Trader** session bean and enter a JNDI name in the right hand panel of the editor. We used the name *TraderHome*, although you may wish to include a subcontext, or leave the JNDI name as the default of ejb/itso/cics/eci/j2ee/trader/TraderHome. These instructions assume you enter a name of **TraderHome** as shown in Figure 3-11.

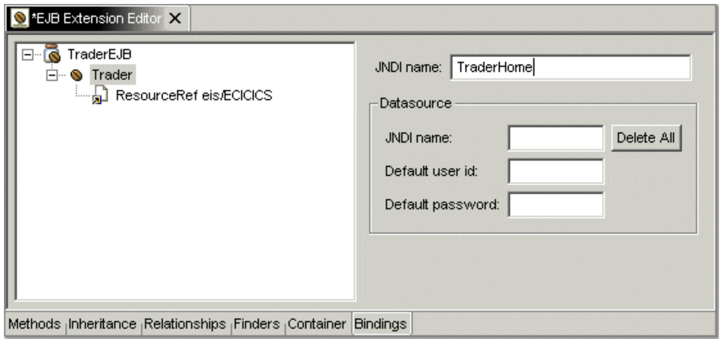
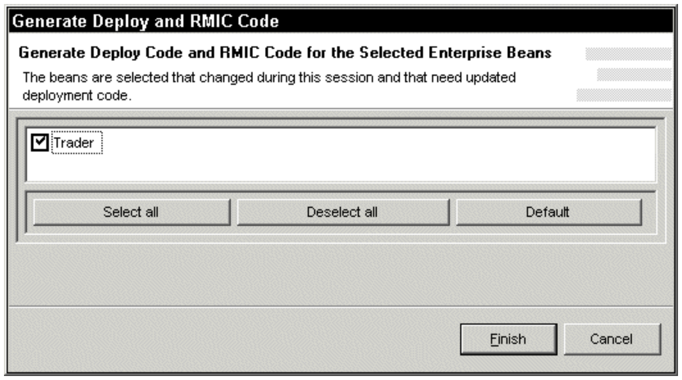

JNDI name: TraderHome

Figure 3-11 Adding the JNDI name of the session bean

10. Press **Ctrl-S** to save the file and close the EJB Extensions Editor.

Generating bean deployment code

Before the session bean can be run in an application server, its deployment code must be generated along with the stubs and ties. From the J2EE perspective *Navigator* view, right click on the **TraderEJB** project and select **Generate -> Deploy and RMIC code**. Select the checkbox next to the Trader enterprise bean as shown in Figure 3-12, and click **Finish** to generate the code. This will create all of the necessary deployment classes. You must perform this step whenever you have made changes to the bean code.


Generate Deploy and RMIC Code

Generate Deploy Code and RMIC Code for the Selected Enterprise Beans

The beans are selected that changed during this session and that need updated deployment code.

☑ Trader

Figure 3-12 Generating bean deployment code

3.2.3 Testing the session bean

Now the session bean is built, it is time to test it. This section describes how to test the Trader session bean within the WebSphere Test Environment of WebSphere Studio. The test environment differs depending on the configuration of WebSphere Studio you are using:

- WebSphere Studio Application Developer contains no connector support. If you perform the following tasks you can test in a non-managed environment:
  - Manually update the runtime classpath to support the J2EE Connector Architecture, the CICS ECI resource adapter, and Enterprise Access Builder.
  - Manually publish an `ECIConnectionFactory` instance to the JNDI namespace for the command bean to lookup.
- WebSphere Studio Application Developer Integration Edition supports the CICS ECI resource adapter, and allows you to test in a managed environment. It contains the following features:
  - No changes necessary to the runtime classpath.

– Provides a utility to automatically publish an `ECIConnectionFactory` instance to the JNDI namespace.

This section contains the following steps:

1. Creating a WebSphere Test Environment server
2. Modifying the runtime classpath of the test environment
3. Publishing an `ECIConnectionFactory` instance to the JNDI namespace
4. Using the EJB test client

Creating a test environment

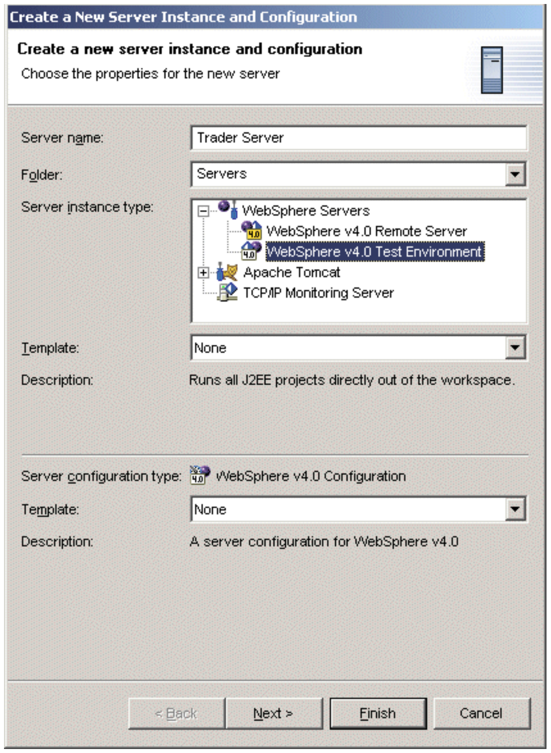
This section describes how to configure a test environment in WebSphere Studio.

Note: If you have been using WebSphere Studio outside the scope of this Redpaper you may already have a WebSphere Test Environment configured. This section recommends creating a new one exclusively for this enterprise application so the runtime classpath can be closely controlled.

The following steps describe how to configure a test environment:

1. Open the Server perspective. Select **Perspective -> Open -> Server**.
2. Select **File -> New -> Server Instance and Configuration**. The *Create a New Server Instance and Configuration* window opens. Enter the following:
   a. Specify a *Server name* of **Trader Server**
   b. Specify a *Folder* of **Servers**
   c. In *Server instance type* expand **WebSphere Servers** and select **WebSphere v4.0 Test Environment**

   The completed screen should match Figure 3-13. When done, press **Finish**.


Create a New Server Instance and Configuration

Create a new server instance and configuration

Choose the properties for the new server

Server name: Trader Server

Folder: Servers

Server instance type:
- WebSphere Servers
  - WebSphere v4.0 Remote Server
  - WebSphere v4.0 Test Environment
- Apache Tomcat
- TCP/IP Monitoring Server

Template: None

Description: Runs all J2EE projects directly out of the workspace.

Server configuration type: WebSphere v4.0 Configuration

Template: None

Description: A server configuration for WebSphere v4.0

Figure 3-13 Create a new server instance

3. In the Server Configuration panel there will now be a server instance and server configuration, both of which are called *Trader Server*. In the Server Configurations list, right click on **Trader Server** and select **Add Project -> Trader**. The Trader enterprise application will now be associated with the Trader Server test environment.

Modifying the runtime classpath of the test environment

Note: Only complete this step if you are using WebSphere Studio Application Developer. In contrast, WebSphere Studio Application Developer Integration Edition needs no changes to the runtime classpath of the test environment.

You need to add the following JAR files to the runtime classpath of the Trader Server test environment:

- CICS ECI resource adapter classes
  - `connector.jar`
  - `ctgclient.jar`

  - `ctgserver.jar`
  - `cicsj2ee.jar`

  You should have downloaded these files from the CICS Transaction Gateway to your workstation in “Step 2 - Update the J2EE Connector Architecture classes” on page 15.

- Enterprise Access Builder classes
  - `bboaxrt.jar`

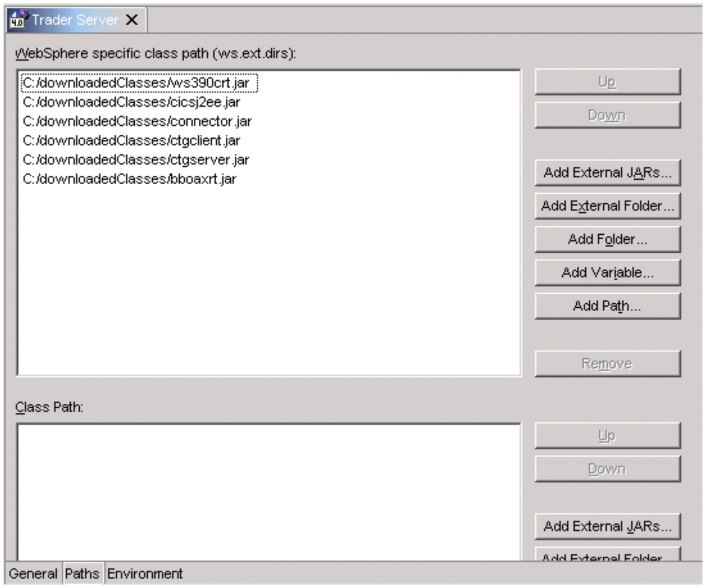
  You should have downloaded this file in “Setting the Java build path” on page 36.

- Java security classes (required for the JNDI publish)
  - `ws390crt.jar`

  This JAR file is found in the WebSphere Application Server `lib` directory (typically `/usr/lpp/WebSphere/lib`). Download this file to your workstation, and store it in the same directory as the others.

Once you have these JAR files on your workstation, perform the following to add them to the runtime classpath:

1. You should still be in the Server perspective. From here, in the Server Configuration panel, double click on the **Trader Server** server instance.
2. This will open up the properties for Trader Server. Select the **Paths** tab, then click the **Add External JARs** button in the *WebSphere specific class path* section. From here you can the six JAR files listed above. After adding these JARs the classpath should appear as shown in Figure 3-14. Save these changes with **Ctrl-S**.



WebSphere specific class path (ws.ext.dirs):

```
C:/downloadedClasses/ws390crt.jar
C:/downloadedClasses/cicsj2ee.jar
C:/downloadedClasses/connector.jar
C:/downloadedClasses/ctgclient.jar
C:/downloadedClasses/ctgserver.jar
C:/downloadedClasses/bboaxrt.jar
```

Figure 3-14 Runtime classpath of Trader Server

Publishing an ECIConnectionFactory instance

The TraderCommand command bean will attempt to retrieve connection information by performing a JNDI lookup at resource reference `eis/CICS_ECI` for an `ECIConnectionFactory` object. You must ensure that the WebSphere Test Environment JNDI namespace contains this resource reference.

The method used to publish the required resource to the JNDI namespace depends on the configuration of WebSphere Studio you are using. This section contains separate instructions for both Application Developer and Application Developer Integration Edition.

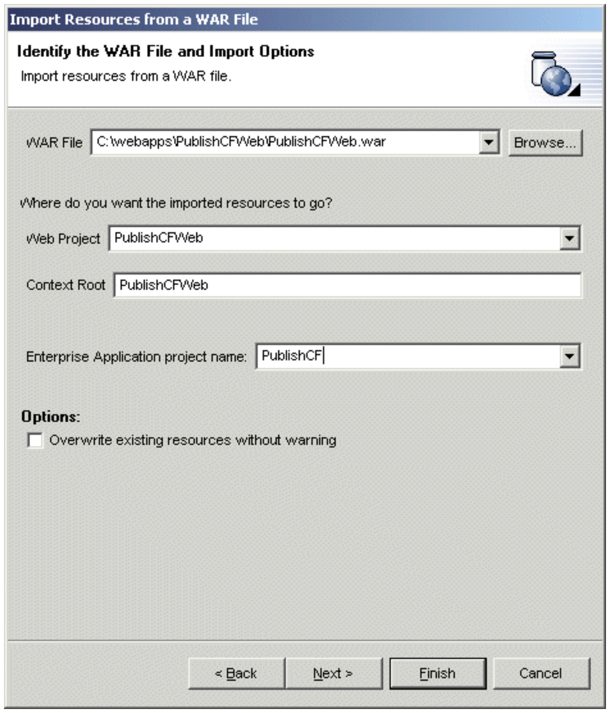
WebSphere Studio Application Developer

Important: If you are using WebSphere Studio Application Developer Integration Edition you should not complete this section.

WebSphere Studio Application Developer does not provide a utility to publish an `ECIConnectionFactory` object to the JNDI namespace. Therefore you must develop an application to perform this operation manually. A Web application called *PublishCFWeb* is included with the sample materials of this Redpaper, which performs this function.

To import PublishCFWeb into WebSphere Studio perform the following:

1. From the J2EE perspective, choose **File -> Import** from the menu system.
2. Select **WAR file** as the import source and click **Next**.
3. Click the **Browse** button and locate the supplied file `PublishCFWeb.war`. Select the file and click **Open**. Complete the remainder of the dialog box as shown in Figure 3-15.


Import Resources from a WAR File

Identify the WAR File and Import Options

Import resources from a WAR file.

WAR File: C:\webapps\PublishCFWeb\PublishCFWeb.war

Where do you want the imported resources to go?

Web Project: PublishCFWeb

Context Root: PublishCFWeb

Enterprise Application project name: PublishCF

Options:

☐ Overwrite existing resources without warning

Figure 3-15 Importing PublishCFWeb

4. Click **Next** to display the *Module Dependencies* dialog. No changes are needed here so click **Next** to move to the *Define Java Build Settings* dialog. Select the **Libraries** tab, click on **Add External JARs**, and add the following:
   - `connector.jar`
   - `cicsj2ee.jar`

   Then click **Finish**. The PublishCFWeb project should now be imported.
5. Add the PublishCF application to the Trader Server test environment. From the Server perspective go to the Server Configurations panel, right click on the **Trader Server** server configuration and select **Add Project -> PublishCF**.

You need to make some changes to this Web application before you can use it:

1. Expand the **PublishCFWeb** project to display the source code for PublishCFServlet:

   ```
   source/itso/cics/eci/j2ee/jndi/PublishCFServlet.java
   ```

2. You must ensure that the resource name you use matches what the TraderCommand command bean will attempt to look up. Change the code shown in Figure 3-16 if necessary.

```
public class PublishCFServlet extends HttpServlet {

   //*** Change subcontext and resource_name if necessary
   String subcontext = "eis";
   String resource_name = "eis/CICS_ECI";
```

Figure 3-16 Resource name settings in PublishCFServlet

3. You will need to change the connection information as shown in Figure 3-17 to represent your system setup. Uncomment any lines if you need to set these parameters.

```
      //*** Change connection information here
      ECIManagedConnectionFactory mcf = new ECIManagedConnectionFactory();
      mcf.setConnectionURL("tcp://wtsc58oe.itso.ibm.com");
      mcf.setServerName("SC58CIC2");
      //mcf.setUserName("user");
      //mcf.setPassword("password");
      //mcf.setPort();
```

Figure 3-17 Connection information in PublishCFServlet

4. Press **Ctrl-S** to save the changes made to this servlet.

When the command bean retrieves the JNDI reference it will need to use the CFNamingHelper class to rebuild the `ECIConnectionFactory` object. Therefore you must complete the following steps:

1. Right click on `CFNamingHelper.java` which is also part of the `itso.cics.eci.j2ee.jndi` package. Select **Copy**.
2. The *Folder Selection* dialog will open. In the *Select the destination* field enter:

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

   Press **OK** and the CFNamingHelper class will be copied to the TraderEJB project.

You are now ready to publish an ECIConnectionFactory object to the JNDI namespace. To do this, follow the steps below:

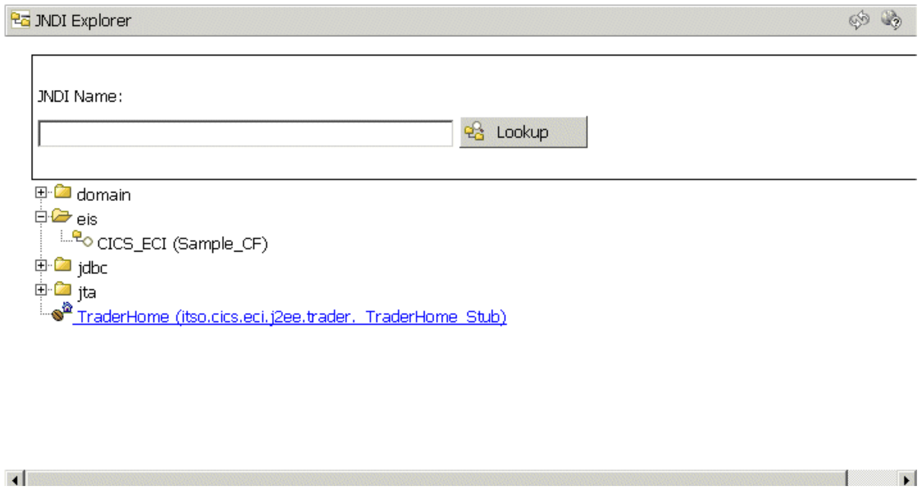
1. From the Server perspective select the **Servers** tab.
2. Right click on **Trader Server** and select **Start**. This will first publish the server configuration, then start the server.
3. Once started, open a Web browser within WebSphere Studio if one is not already open. You do this by pressing the *Open Web Browser* button which has an icon of a globe. Then enter the following URL:

   ```
   http://localhost:8080/PublishCFWeb/PublishCFServlet
   ```

   This will attempt to perform the publish. If it is successful you should see the message:

   ```
   Publish of eis/CICS_ECI successful
   ```

4. To can confirm the publish worked by using the JNDI Explorer. This application displays all of the references published in the JNDI name space. To use it select the **Servers** tab, right click on **Trader Servlet**, and select **Test Client**. When the Test Client starts up, select **JNDI Explorer** to display the contents of the namespace. If your publish was successful you will see a screen similar to Figure 3-18 showing the eis/CICS_ECI resource. Notice there is also a resource for TraderHome.



```
domain
eis
   CICS_ECI (Sample_CF)
jdbc
jta
TraderHome (itso.cics.eci.j2ee.trader._TraderHome_Stub)
```

Figure 3-18 The JNDI Explorer in Application Developer

Note: Every time you restart Trader Server you will need to re-run PublishCFServlet to publish to the JNDI namespace as the publish is not persistent between server restarts.

You are now ready to test the Trader session bean.

WebSphere Studio Application Developer Integration Edition

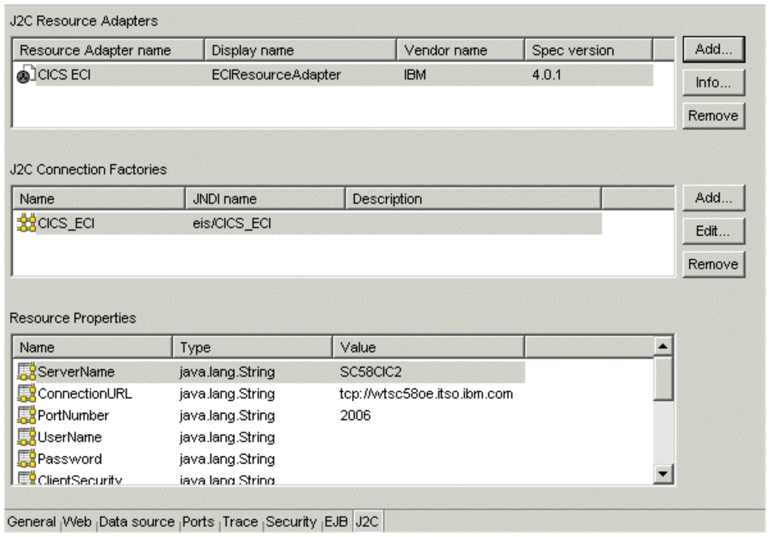
Important: If you are using WebSphere Studio Application Developer you should not complete this section.

WebSphere Studio Application Developer Integration Edition can automatically publish `ECIConnectionFactory` references into the JNDI namespace. This is all part of the J2EE Connector Architecture and CICS ECI resource adapter runtime support provided by this configuration. It allows you to test in a managed environment.

To configure the managed environment for CICS, perform the following:

1. From the Server perspective locate the *Server Configuration* panel and double click on the **Trader Server** server configuration. In the panel that opens, select the **J2C** tab.
2. You can now install and configure the CICS ECI resource adapter:
   a. In the *J2C Resource Adapters* section press **Add**. In the *Resource Adapter Name* field highlight **CICS ECI** then press **OK**.
   b. In the *J2C Connection Factories* section press **Add**. Enter the following values:
      - Name: **CICS_ECI**
      - JNDI name: **eis/CICS_ECI**

      The other properties are used for connection pooling. Press **OK**.
   c. The *Resource Properties* section will now contain a list of properties. Change the relevant properties for your installation. We entered:
      - ServerName: **SC58CIC2**
      - ConnectionURL: **tcp://wtsc58oe.itso.ibm.com**
   d. The completed screen should resemble Figure 3-19 (although your resource properties will be different). Press **Ctrl-S** to save.


J2C Resource Adapters

Resource Adapter name	Display name	Vendor name	Spec version
CICS ECI	ECIResourceAdapter	IBM	4.0.1

J2C Connection Factories

Name	JNDI name	Description
CICS_ECI	eis/CICS_ECI	

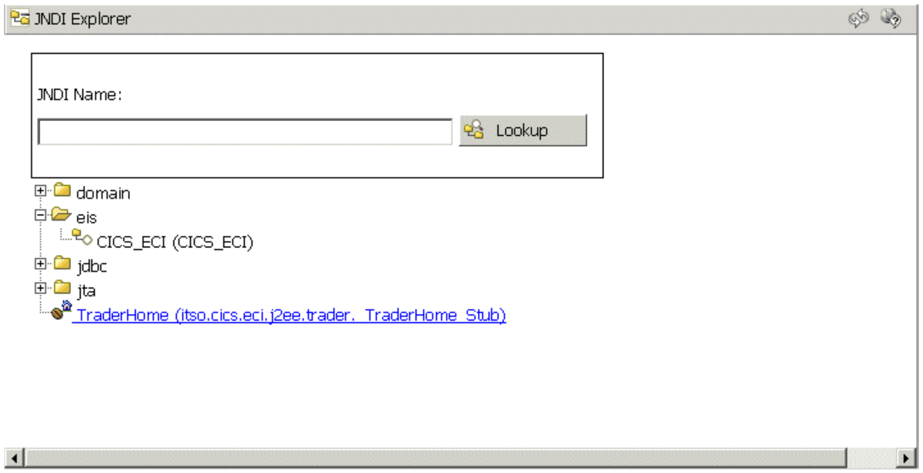
Resource Properties

Name	Type	Value
ServerName	java.lang.String	SC58CIC2
ConnectionURL	java.lang.String	tcp://wtsc58oe.itso.ibm.com
PortNumber	java.lang.String	2006
UserName	java.lang.String	
Password	java.lang.String	
ClientSecurity	java.lang.String	

Figure 3-19 J2C configuration in Application Developer Integration Edition

Now whenever Trader Server is started, this resource will be published to the JNDI server automatically. You can see what is stored in the JNDI namespace by using the JNDI Explorer application. To do this, follow the steps below:

1. From the Server perspective select the **Servers** tab.
2. Right click on **Trader Server** and select **Start**. This will first publish the server configuration, then start the server.
3. Once the server has started select the **Servers** tab, right click on **Trader Servlet**, and select **Test Client**. When the Test Client starts up, select **JNDI Explorer** to display the contents of the namespace. If your publish was successful you will see a screen similar to Figure 3-20 showing the eis/CICS_ECI resource. Notice there is also a resource for TraderHome.


JNDI Explorer

JNDI Name:

Lookup

```
domain
eis
  CICS_ECI (CICS_ECI)
jdbc
jta
TraderHome (itso.cics.eci.j2ee.trader._TraderHome_Stub)
```

Figure 3-20   The JNDI Explorer in Application Developer Integration Edition

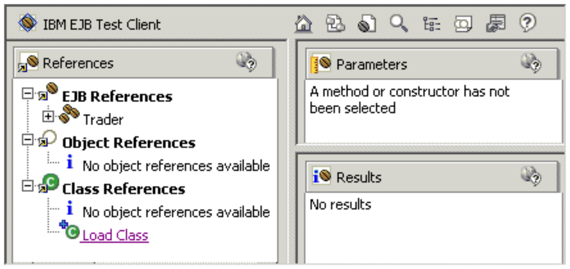
You are now ready to test the Trader session bean.

Using the EJB Test Client

Attention: Complete this, and all other, sections regardless of the WebSphere Studio configuration you are using

This section assumes the Trader Server is started and an `ECIConnectionFactory` object has been published to the JNDI name server. To test the Trader session bean perform the following:

1. From the IBM EJB Test Client refresh the JNDI explorer. Click on the **TraderHome** reference. This is the home interface of the Trader session bean. The EJB page of the Test Client opens as shown inFigure 3-21.


IBM EJB Test Client

References

```
EJB References
  Trader
Object References
  No object references available
Class References
  No object references available
  Load Class
```

Parameters

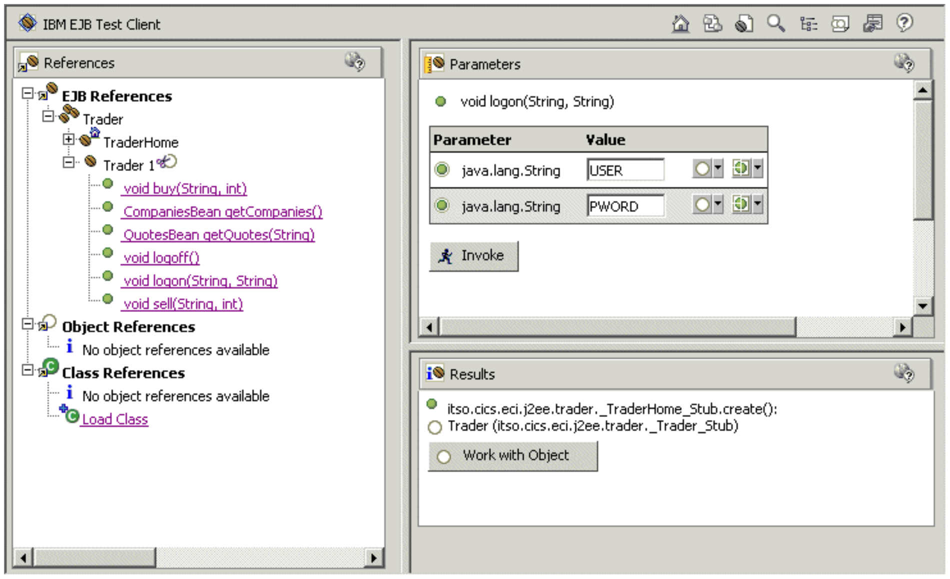
A method or constructor has not been selected

Results

No results

Figure 3-21   EJB page of the EJB Test Client

2. In the *References* panel, fully expand the tree of the Trader *EJB References*, and then click on the home interface's **create()** method.
3. This will add the `create()` method to the *Parameters* panel. This panel is used to supply any required values to a method before invoking it. As the `create()` method takes no input, just click the **Invoke** button to create an instance of the session bean. The *Results* panel will display the new session bean object. Click the **Work with Object** button that also appears in the *Results* panel at the bottom.
4. This will add an instance of the `Trader` bean named `Trader1` to the *EJB References* section of the *References* panel. Expand the tree on this reference to see the available methods from the bean's remote interface.
5. At this point, you can choose a method to invoke. Clicking on our `logon()` method causes the method to be displayed in the *Parameters* panel, as shown in Figure 3-22.



void logon(String, String)

Parameter	Value
java.lang.String	USER
java.lang.String	PWORD

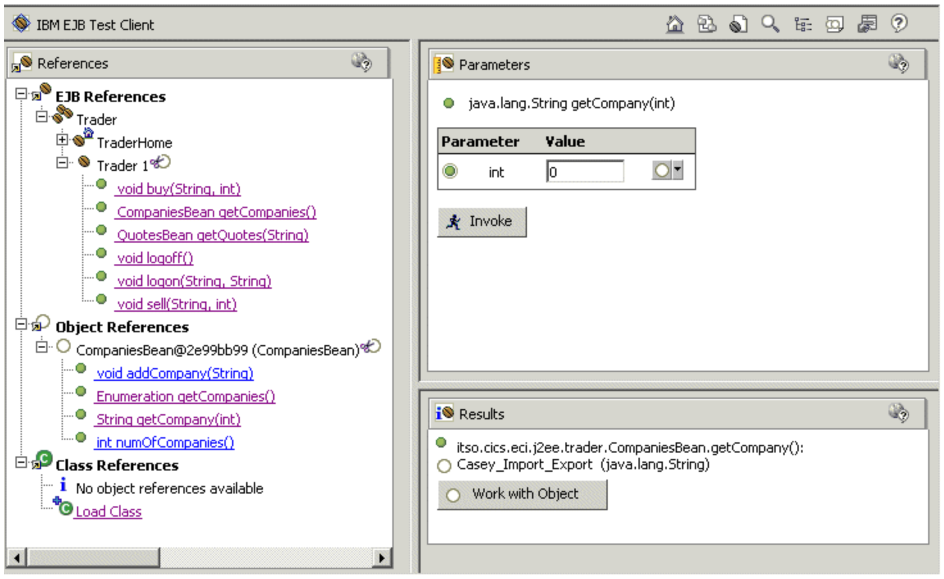
itso.cics.eci.j2ee.trader._TraderHome_Stub.create():
Trader (itso.cics.eci.j2ee.trader._Trader_Stub)

Figure 3-22 Invoking methods on the bean's remote interface

6. This method takes two strings as input. Enter a userid into the first box and a password into the second, and click **Invoke,** which should successfully run this method, as reported in the *Results* panel.
7. Next, try the `getCompanies()` method, which actually makes a connection to CICS. Click on the method name and then click the **Invoke** button in the same way as the `logon()` method. The `getCompanies()` method returns an object of

type `CompaniesBean`, which is one of the data bean classes that was used to hold information about the companies available for trading.

8. The `CompaniesBean` object now appears in the *Results* panel. Click on the **Work with Object** button so that you can look at the data contained by this bean. This adds a CompaniesBean instance to the Object References tree in the *References* panel. By expanding the tree on this object, you can choose one of its available public methods in the same way as you chose a remote interface method on the session bean. Click on the **getCompany()** method.
9. Set the input `int` parameter in the *Parameters* panel to **0** and click **Invoke**. This returns the name of the first company, Casey_Import_Export, as shown in Figure 3-23.


IBM EJB Test Client

References

EJB References
- Trader
  - TraderHome
  - Trader 1
    - void buy(String, int)
    - CompaniesBean getCompanies()
    - QuotesBean getQuotes(String)
    - void logoff()
    - void logon(String, String)
    - void sell(String, int)

Object References
- CompaniesBean@2e99bb99 (CompaniesBean)
  - void addCompany(String)
  - Enumeration getCompanies()
  - String getCompany(int)
  - int numOfCompanies()

Class References
- No object references available
- Load Class

Parameters

java.lang.String getCompany(int)

Parameter	Value
int	0

Invoke

Results

itso.cics.eci.j2ee.trader.CompaniesBean.getCompany():
Casey_Import_Export (java.lang.String)

Work with Object

Figure 3-23 Testing the getCompany() method

10. By using the same techniques, you will be able to use the test client to test each of the remaining methods in your session bean.

3.2.4 Completing the enterprise application

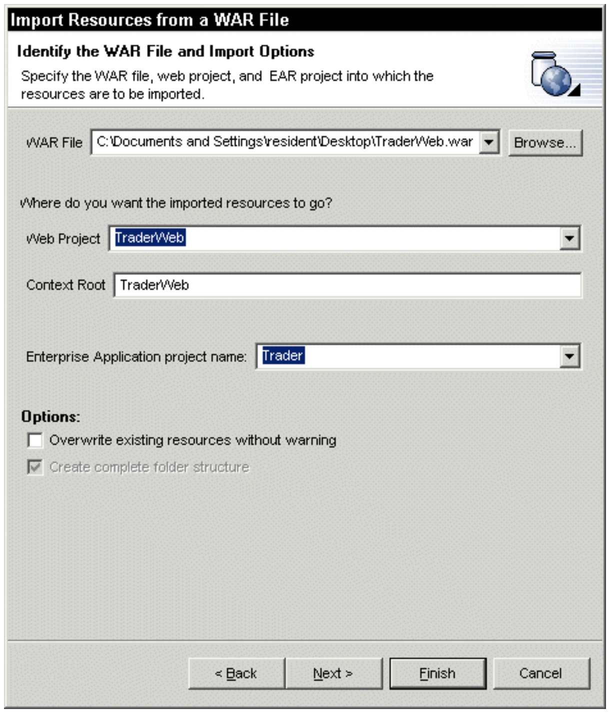
There are two final steps to complete the enterprise application. They are:

1. Import the Web component of the enterprise application
2. Export the entire enterprise application as an EAR file

Importing the Web components

Now that the session bean is completed and successfully working, you can add the Web components of the application to the workspace, and then test the completed Web application. The following steps explain how to do this:

1. Choose **File -> Import** from the menu system.
2. Select **WAR file** as the import source and click **Next**.
3. Click the **Browse** button and locate the supplied file `TraderWeb.war`. This contains the servlet and JSPs for our application, as well as additional data bean classes. Select the file and click **Open**. Complete the remainder of the dialog box as shown in Figure 3-24.


Import Resources from a WAR File
Identify the WAR File and Import Options
Browse...
< Back
Next >
Finish
Cancel

Specify the WAR file, web project, and EAR project into which the resources are to be imported.

WAR File: C:\Documents and Settings\resident\Desktop\TraderWeb.war

Where do you want the imported resources to go?

Web Project: TraderWeb

Context Root: TraderWeb

Enterprise Application project name: Trader

Options:

- [ ] Overwrite existing resources without warning
- [x] Create complete folder structure

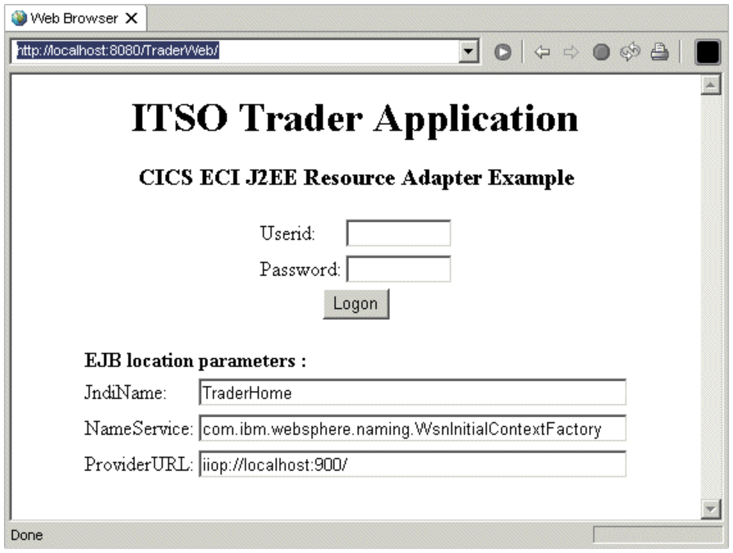
Figure 3-24 Importing the TraderWeb WAR file

4. Click **Next** to display the *Module Dependencies* dialog.
5. Click the checkbox for `TraderEJB.jar` to add the session bean to the servlet's build and runtime classpaths. Click **Next** to display the *Define Java Build Settings* dialog.

6. The *TraderEJB* project has already been added to the build class path in the previous step, so click **Finish** as there is no need to add any further resources.
7. The workspace will now contain the completed enterprise application project *Trader*, and its two module projects *TraderEJB* and *TraderWeb*. You can now test the application. Restart the test environment like before by switching to the *Servers* view; right clicking on the server instance, and choosing **Restart**.
8. Check the *Console* view to ensure that the server started without any errors.

Note: Remember if you are using WebSphere Studio Application Developer you must run PublishCFServlet before attempting to use the Trader enterprise application.

9. This time, right click on the **TraderWeb** project in the *Navigator* view and choose **Run on Server**. This will open the welcome page, `Logon.html`, of the Web application in a *Web Browser* view. Your screen should look similar to that shown in Figure 3-25.



ITSO Trader Application

CICS ECI J2EE Resource Adapter Example

Userid:

Password:

EJB location parameters :

JndiName: TraderHome

NameService: com.ibm.websphere.naming.WsnInitialContextFactory

ProviderURL: iiop://localhost:900/

Figure 3-25 Testing the enterprise application

10. Enter a userid and password, and click **Logon** to begin testing the completed application.

Note: The EJB location parameters tell the servlet that processes the request where it can find the session bean in a JNDI namespace. If you used a different name than `TraderHome` for the session bean's JNDI binding, you must change the *JndiName* value in this form to match the name you chose. You also have the ability to change the *NameService* and *ProviderURL* values, but the defaults are correct for the scenario that is described.

Exporting the enterprise application

Once you are happy that the completed enterprise application is working, you should export it so it can ultimately be deployed to WebSphere Application Server for z/OS. Follow these steps to export the application as an EAR file. The EAR file will contain two components, an EJB JAR file and a WAR file, which represents the EJB module project and the Web module project respectively.

1. Choose **File -> Export** from the menu system, and choose **EAR file** as the export destination.
2. Select the enterprise application project **Trader** from the drop down box as the resource to be exported. Click **Browse** to choose where to save the file. The examples saves it as `Trader.ear`.
3. Click **Finish** to generate the EAR file.

You have now completed the WebSphere Studio chapter. The next chapter describes how to deploy your enterprise application into WebSphere Application Server for z/OS.

4

WebSphere Application Server for z/OS

This chapter describes how to install and configure the CICS ECI resource adapter into WebSphere for z/OS. It then describes the steps necessary to take an enterprise application exported from WebSphere Studio and deploy it to WebSphere for z/OS.

This chapter contains the following sections:

- Enabling CICS connector support
- Deploying to WebSphere Application Server
- Advanced tasks to try

4.1 Enabling CICS connector support

The steps required to configure the CICS ECI resource adapter are fully described in the *Installation and Customization* manual in *“Chapter 5: Performing Advanced Tasks”* in the *“Configuring the WebSphere for z/OS-supported connectors”* sub-section. Ensure you obtain at least the Fifth Edition of this manual, which was published on April 2nd 2002. The instructions in the section summarize the instructions given in this manual.

4.1.1 Preparing the CICS ECI resource adapter

This section describes the steps necessary to prepare for the use of the CICS ECI resource adapter with WebSphere Application Server. It consists of the following steps:

- CICS Transaction Gateway considerations
- CICS Transaction Server considerations
- Installation of the CICS ECI resource adapters

CICS Transaction Gateway considerations

This Redpaper assumes you have the CICS Transaction Gateway V4.0.2 installed.

To configure the CICS ECI resource adapter you need to determine if you will use a generic or specific EXCI pipe to communicate with CICS. The default is to use a generic pipe. If you wish to use a specific pipe you must define the DFHJVPIPE environment variable in the J2EE server that will be used for the CICS connector support. This value must match the NETNAME parameter of an installed CONNECTION definition in CICS.

CICS Transaction Server considerations

This Redpaper assumes you have a CICS Transaction Server V1.3 region, or higher, configured and running.

You need to perform the following tasks:

1. Set the RRMS SIT parameter
2. Configure EXCI in CICS
3. Define an EXCI library to your J2EE server
4. Define security profiles

Setting the RRMS SIT parameter

For each CICS region that you wish to access through the CICS ECI resource adapter you must set the following System Initialization Table parameter:

– `RRMS=YES`

Configuring EXCI in CICS

Ensure that EXCI is configured in each CICS region you wish to access using the CICS ECI resource adapter. This involves installing a SESSIONS and CONNECTION resource definition. Samples are provided in the CICS supplied group DFH$EXCI. Also ensure IRC is set to Open.

Defining an EXCI library to your J2EE server

The J2EE server that will use the CICS ECI resource adapter needs to access and load the CICS module DFHXCSTB from the SDFHEXCI library supplied with CICS. This library is typically CICSTS13.CICS.SDFHEXCI or CICSTS22.CICS.SDFHEXCI depending on the CICS release and naming conventions you use. Complete one of the following steps:

- Add module DFHXCSTB to the LPA
- Add the SDFHEXCI library to either the system linklist concatenation, or to the STEPLIB concatenation of the J2EE server.

Defining security profiles

If your CICS region uses SURROGAT checking for CICS regions (as defined in the DFHXCOPT table), you must authorize the userid associated with the J2EE server region to act as a surrogate for the clients that invoke J2EE applications running in the J2EE server. Complete the following:

1. For all potential clients of J2EE application components that will use the CICS ECI resource adapter, define a RACF CICS DFHEXCI SURROGAT profile definition. You may define one profile for each user, one profile for each group of users, or a single surrogate profile for all users.

   ```
   RDEFINE SURROGAT *.DFHEXCI UACC(NONE) OWNER(profile-owner-userid)
   ```

2. Authorize the userid of the J2EE server region to be the CICS DFHEXCI surrogate for the specific userid or set of userids that you just identified through the profile in the RACF SURROGAT class.

   ```
   PERMIT *.DFHEXCI CLASS(SURROGAT) ID(server-userid) ACCESS(READ)
   ```

Installation of the CICS ECI resource adapter

Resource adapters are packaged in Resource Adapter Archive (RAR) files. The CICS ECI resource adapter for z/OS is provided in the file `cicseciRRS.rar` in the `deployable` directory of the CICS Transaction Gateway. Note the following:

- This `cicseciRRS.rar` resource adapter archive is designed exclusively for z/OS. It makes use of RRS to provide global transaction support. All non-z/OS platforms use the `cicseci.rar` file instead.
- A RAR file is much like an EAR file. It is an archive made up of JAR files and a deployment descriptor.
- Unlike many application servers, WebSphere Application Server for z/OS cannot work with RAR files directly, so you must extract the contents of the RAR file into a directory, and point the WebSphere classpath to each JAR file in this directory.

To install the CICS ECI resource adapter, perform the following:

1. Create a new directory in Unix System Services to extract the contents of the RAR file to. For example:
   - `/usr/lpp/connectors`

   Ensure your userid has write access to this directory, and grant read and execute permissions to everyone else. We recommend you also mount an HFS against this directory.
2. Locate the `cicseciRRS.rar` file and copy it to the connectors directory you just created. By default `cicseciRRS.rar` will be in the following directory:
   - `/usr/lpp/ctg/deployable`
3. In the connectors directory extract the contents of `cicseciRRS.rar`. To do this:
   a. Add the `bin` directory of your Java installation to the PATH environment variable if it is not already defined. For example:
      - `export PATH=/usr/lpp/java/IBM/J1.3/bin:$PATH`

   b. Enter the following command from the connectors directory:
      - `jar -xvf cicseciRRS.rar`

      This will extract the following directory and files:
      - `META-INF`
      - `ccf2.jar`
      - `cicseciRRS.jar`
      - `cicsframe.jar`
      - `cctgclient.jar`
      - `ctgserver.jar`
      - `libCTGJNI.so`
      - `libCTGJNI_g.so`
4. Use the following commands to change the permissions of the `libCTGJNI.so` and `libCTGJNI_g.so` files:
   - `chmod ugo+x libCTGJNI.so`

– `chmod ugo+x libCTGJNI_g.so`

4.1.2 Defining connection information

You are now ready to define connection information to the J2EE server. Perform the following

1. From the Systems Management User Interface, create a new conversation. This conversation will be used to add the CICS connector support.
2. Turn on connection management in the sysplex. To do this right click on the sysplex name in the conversation and select **Modify**. In the *Configuration Extensions* section check the box labeled **Connection Management**. Click on **Selected -> Save** to save your changes.
3. Expand the list of J2EE servers and locate the J2EE server that you wish to use with the CICS ECI resource adapter. Right click on the J2EE server and select **Modify**. Make the following changes in the *environment variable list*:

   a. Add the following classes from the connector directory you created earlier to the CLASSPATH variable:

      - `cicseciRRS.jar`
      - `cicsframe.jar`
      - `ctgserver.jar`
      - `ctgclient.jar`

      Note: Separate each entry with a colon (**:**)

   b. Add the connectors directory to the LIBPATH. For example:

      - `/usr/lpp/connectors`

   c. If you are using a specific EXCI pipe, you must set the DFHJVPIPE environment variable here

   d. We recommend that you initially turn on JNDI tracing so you can see the connections being made to CICS. Set the CTG_JNI_TRACE environment variable to a file, for example:

      - `CTG_JNI_TRACE = /tmp/jniTrace.txt`

   Press **Selected -> Save** to save your changes.
4. To define a CICS ECIConnectionFactory as a new J2EE resource perform the following:

   a. Locate the *J2EE Resources* folder under your sysplex, right click on it and select **Add**.

   b. In the properties window enter the following:

i. Set the *J2EE resource name*. We used **CICS_ECI**.

ii. Set the *J2EE resource type* to **CICS_ECIConnectionFactory**.

Note: If you are not presented with the CICS_ECIConnectionFactory option in the J2EE resource type list, then APAR PQ55873 may not have been applied. If the APAR has been applied and you still do not have a CICS_ECIConnectionFactory resource type, move to the directory:

```
/usr/lpp/WebSphere/samples/
```

Look for the following two files:

```
CICS_ECIConnectionFactory.properties
CICS_ECIConnectionFactory.xml
```

If these files exist, then copy them to the following directory (replacing `SYSPLEX` with the sysplex name you are using):

```
/WebSphere390/CB390/SYSPLEX/resources/templates
```

This is the directory the SM EUI uses to locate J2EE resource types by default. Restart the SM EUI to pick up the change.

iii. Press **Selected -> Save** to save your changes. Look for the following message in the status bar to confirm the operation completed successfully:

```
BBON0515I J2EE Resources name was added.
```

5. You can now create a J2EE resource instance which defines the connection information. To do this, perform the following:

a. Expand the J2EE resource you have just created. This displays a *J2EE resource instances* folder. Right click on it and select **Add**.

b. In the properties window enter the following:

i. Set the *CICS_ECIConnectionFactory instance name*. We used **CIC2**.

ii. Set the *System name* with which this J2EE resource instance is to be associated.

iii. Set the *Server name* to the APPLID of the CICS server you wish to call. We used **SC58CIC2**.

c. Click on **Selected -> Save** to save your changes. Look for the following message in the status bar to confirm the operation completed successfully:

```
BBON0515I J2EE resource Instance [name] was added.
```

The J2EE server region is now configured to use the CICS ECI resource adapter. Do not commit the conversation yet because you still need to deploy an application to make use of this support, and this is described in the next section.

4.2 Deploying to WebSphere Application Server

This section describes how to take the Trader EAR file generated by WebSphere Studio and deploy it into WebSphere Application Server for z/OS.

4.2.1 Application Assembly Tool

EAR files generated in WebSphere Studio cannot be directly deployed to WebSphere Application Server for z/OS. You must use the Application Assembly Tool (AAT) to generate an EAR file that WebSphere can use. The AAT also gives you the opportunity to change some deployment information.

Note: In this Redpaper we used the Application Assembly Tool V4.00.027. If you are using an earlier release of AAT we recommend you upgrade. You can download the latest version of the AAT from:

```
https://www6.software.ibm.com/dl/websphere20/zosos390-p
```

Complete the following to generate a EAR file suitable for WebSphere from the `Trader.ear` file exported from WebSphere Studio:

1. Before starting the AAT you need to set the AAT user classpath. When the AAT is importing files it sometimes needs access to classes that are not in the EAR file being imported. In the case of Trader.ear, the TraderCommand class contained within this EAR file makes a reference to the Enterprise Access Builder class CommunicationCommand. This class is not packaged in the EAR file -- instead it is added to the WebSphere runtime classpath as it is a class that is commonly required by many applications.

   The CommunicationCommand class is contained in the `bboaxrt.jar` file. To add this class to the AAT classpath set the AAT_USER_PATH environment variable to point to this file. To do this in Windows 2000:

   a. Click **Start -> Settings -> Control Panel.**

   b. From the Control Panel double click on **System**, select the **Advanced** tab, then click on the **Environment Variables** button.

   c. In the *user variables* section press the **New** button and add the environment variable:

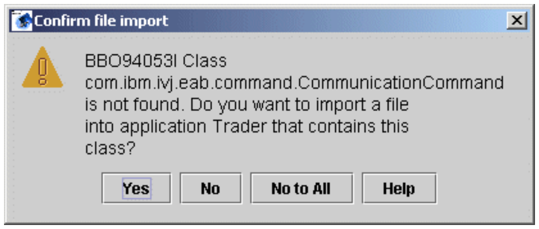
      i. Set the *Variable Name* to **AAT_USER_PATH**

ii. Set the *Variable Value* to point to `bboaxrt.jar` (you have downloaded this in the previous chapter). For example: `c:\downloadedclasses\bboaxrt.jar`

d. Click **OK** in each of the three windows to save the change.

2. Start the AAT, or restart it if it was already open. In the left pane, right click on **Applications** and select **Import**. Specify the location of the `Trader.ear` file you exported from WebSphere Studio then press **OK**. The file should import successfully.

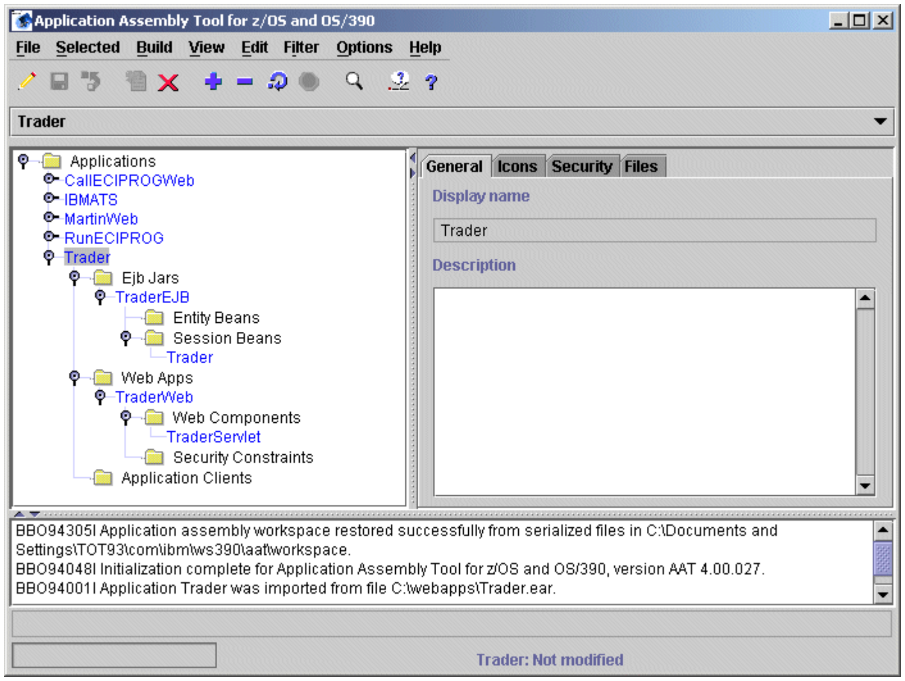
   If you see the message shown in Figure 4-1 then you did not correctly set the AAT_USER_PATH environment variable. These messages inform you of which classes you will need to add to the AAT_USER_PATH. There does not seem to be a good way to predict in advance what is needed, as not all referenced classes in the WebSphere runtime are required.



BBO94053I Class com.ibm.ivj.eab.command.CommunicationCommand is not found. Do you want to import a file into application Trader that contains this class?

Figure 4-1 AAT warning message

3. Under the applications folder there should now be a Trader application. Fully expand it to show all the components contained within the application, as shown in Figure 4-2.



```
BBO94305I Application assembly workspace restored successfully from serialized files in C:\Documents and
Settings\TOT93\com\ibm\ws390\aat\workspace.
BBO94048I Initialization complete for Application Assembly Tool for z/OS and OS/390, version AAT 4.00.027.
BBO94001I Application Trader was imported from file C:\webapps\Trader.ear.
```

Figure 4-2 The contents of the Trader application

4. Click on **Trader -> Ejb Jars -> TraderEJB -> Session Beans -> Trader**. From here you can set deployment descriptor information for the session bean. For the purposes of this test we will let most things default, or leave them to keep the same values as we set in WebSphere Studio. However we do need to make one change:
   a. Right click on **Trader** and select **Modify**.
   b. Select the **Transactions** tab. From here we can define the transaction attribute for each method in the remote interface.
   c. We will not be using two phase commit with Trader so select **Modify all**, and change the *Transaction attribute* to **NotSupported**. Click **OK**.
   d. Click on **Selected -> Save** to save your changes.
5. You are now ready to create the deployment code for Trader. Right click on the **Trader** application and select **Deploy**. Look for the following message to confirm deployment has successfully completed:

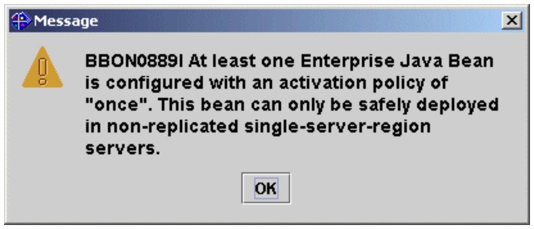
   ```
   BBO94009I Application Trader was deployed.
   ```
6. Finally export the modified Trader application from the AAT to the file system. Right click on the **Trader** application and select **Export**. Enter a path to store

the EAR file, and be sure that the *WebSphere for z/OS Version 4.0 compatible* option is checked. Press **OK**.

4.2.2 Systems Management User Interface

The final task is to take the EAR file generated in the AAT, and deploy it to a J2EE server in WebSphere. Complete the following steps:

1. Return to the conversation you were previously editing in the Systems Management User Interface. Right click on the J2EE server where you installed the CICS ECI resource adapter and select **Install J2EE Application**.
2. In the *EAR Filename* field, enter the name of the EAR file you exported from the AAT, then press **OK**. You will see the message shown in Figure 4-3.



```
BBON0889I At least one Enterprise Java Bean
is configured with an activation policy of
"once". This bean can only be safely deployed
in non-replicated single-server-region
servers.
```

Figure 4-3 Activation policy message

This message implies that the Trader session bean specifies an activation policy of *once*, and therefore can only be deployed to a single server region. You can change the activation policy in the AAT by selecting the Trader session bean and clicking on the Policy tab. For the purposes of our test we will leave this activation policy set to once, and will define our J2EE server to be a single server region. Click **OK**.

Note: We do not recommend using single server J2EE regions in production, for the reasons of scalability. The single server approach is used in this Redpaper for simplicity of testing the Trader application.

3. The *Reference and Resource Resolution* window will open. We need to set and resolve some references before the Trader enterprise application can be deployed.
   a. Expand the **Trader EJB** folder and highlight the **Trader** session bean. There are several properties we need to set in here.
      i. Ensure the **EJB** tab is selected. From here you can specify the full JNDI name to give the Trader session bean. You can set this value to

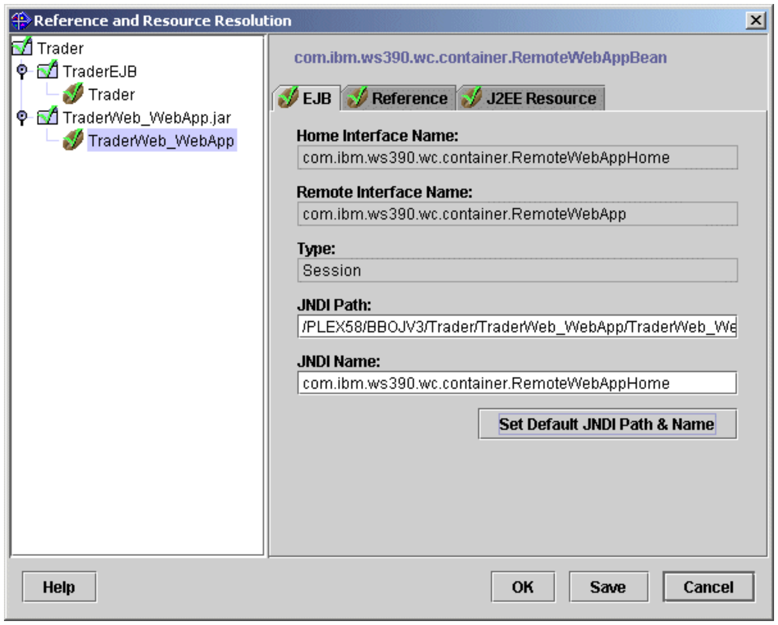
what you want, as the Trader Web application does not hard code the JNDI name of the session bean. We let the *JNDI Path* default, and set the *JNDI Name* to **TraderHome**.

ii. Click on the **EJB Resource** tab. Here you associate the resource reference used by the command bean with an `ECIConnectionFactory` object. Click in the *J2EE resource* field, and select the **CICS_ECI** resource you created earlier.

b. Expand the **TraderWeb_WebApp.jar** folder and select the **TraderWeb_WebApp** bean. You are required to provide a JNDI name for the Web application. This is used internally by WebSphere, and does not affect our application so press the **Set Default JNDI Path & Name** button.

c. All required references should have now been set, and the OK button will become active, as shown in Figure 4-4. Press **OK** to deploy the enterprise application. If the operation completes successfully a message similar to the one below will display:

```
BBON0470I EAR file Trader_resolved.ear has been successfully
installed on server BBOJV3.
```


Reference and Resource Resolution
Trader
TraderEJB
Trader
TraderWeb_WebApp.jar
TraderWeb_WebApp
com.ibm.ws390.wc.container.RemoteWebAppBean
EJB
Reference
J2EE Resource
Set Default JNDI Path & Name
Help
OK
Save
Cancel

Home Interface Name: com.ibm.ws390.wc.container.RemoteWebAppHome

Remote Interface Name: com.ibm.ws390.wc.container.RemoteWebApp

Type: Session

JNDI Path: /PLEX58/BBOJV3/Trader/TraderWeb_WebApp/TraderWeb_We

JNDI Name: com.ibm.ws390.wc.container.RemoteWebAppHome

Figure 4-4 Reference and Resource Resolution window

   d. As the earlier message implied, we need to set this J2EE server to be a single server instance. Right click on the J2EE server name and select **Modify**. Add the following environment variable to the list:

      - `MAX_SRS = 1`

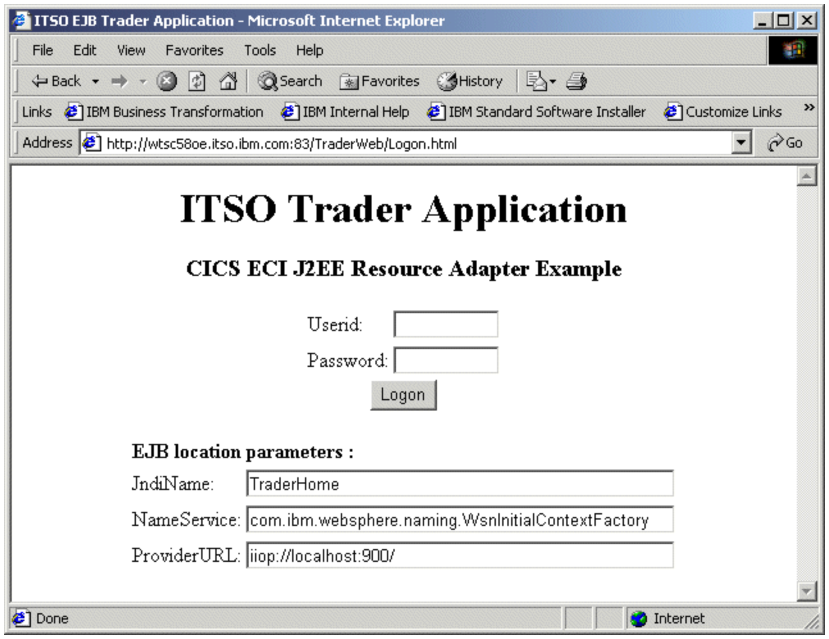
   e. You are not ready to activate the conversation. Right click the conversation name and complete the following steps in turn:

      i. **Validate**
      ii. **Commit**
      iii. **Complete -> All tasks**
      iv. **Activate**

4. Once the conversation is activated your are ready to test the Trader enterprise application. Open a Web browser and specify the host name and port of your J2EE server followed by the `/TraderWeb/Logon.html` URI. For example:

   `http://wtsc58oe.itso.ibm.com:83/TraderWeb/Logon.html`

   The logon screen should display as shown in Figure 4-5,


ITSO EJB Trader Application - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Links IBM Business Transformation IBM Internal Help IBM Standard Software Installer Customize Links
Address http://wtsc58oe.itso.ibm.com:83/TraderWeb/Logon.html Go

ITSO Trader Application

CICS ECI J2EE Resource Adapter Example

Userid:
Password:
Logon

EJB location parameters :

JndiName: TraderHome
NameService: com.ibm.websphere.naming.WsnInitialContextFactory
ProviderURL: iiop://localhost:900/

Figure 4-5 The completed Trader Web application

5. You need to specify the JNDI name of the Trader session bean on this screen. To confirm the value to set it to, look at the active conversation in the Systems Management User Interface. Expand your J2EE server to show a list of J2EE applications installed within it. From here expand **Trader -> J2EEModules ->**

TraderEJB.jar -> J2EE Components -> Trader. Highlight the **Trader** J2EE component and copy the value in the *Home JNDI name* field. Paste this value into the *JndiName* field in the Web browser.

6. Enter a userid and password in the Web browser form then press **Logon**. You can begin testing the Trader enterprise application.

Congratulations! You have built an enterprise application that uses the CICS ECI resource adapter in WebSphere for z/OS!

4.3 Advanced tasks to try

Once you have a working example using the CICS ECI resource adapter, you can experiment with the managed environment qualities of service offered by WebSphere for z/OS. For example you could try:

- Modifying security credentials flowed by the resource adapter. Changing security related settings alters the userid used to run the Trader application in CICS.
- Turning on two-phase commit processing.
- Experimenting with connection management options.

These tasks are described in Chapter 4 of *Assembling Java 2 Platform Enterprise Edition Applications,* SA22-7836, in section “ A closer look at the J2EE server in the Connectors”.

A

Trader CICS Application

This appendix describes the 3270 COBOL Trader application used as the basis of the enterprise application described in this Redpaper.

The Trader CICS Application

We start by providing a description of how the original 3270 based version of the Trader application functions. We then provide a summary of what definitions are required when installing the Trader application in your CICS system.

To obtain the sample COBOL Trader application and accompanying JCL, refer to Appendix B, "Additional material" on page 87.

The 3270 Trader COBOL application

Trader, written in COBOL, uses the VSAM access method for file access and the CICS 3270 BMS programming interface. It is a pseudo-conversational application, meaning that a chain of related non-conversational CICS transactions is used to convey the impression of a conversation to the users as they go through a sequence of screens that constitute a business transaction. The application consists of two modules: TRADERPL, which contains the 3270 presentation logic; and TRADERBL, which contains the business logic. TRADERPL invokes TRADERBL using an EXEC CICS LINK and passing a COMMAREA structure for input and output. TRADERBL contains logic to query and write to the persistent VSAM data, stored in two files the *company file* and the *customer file*.

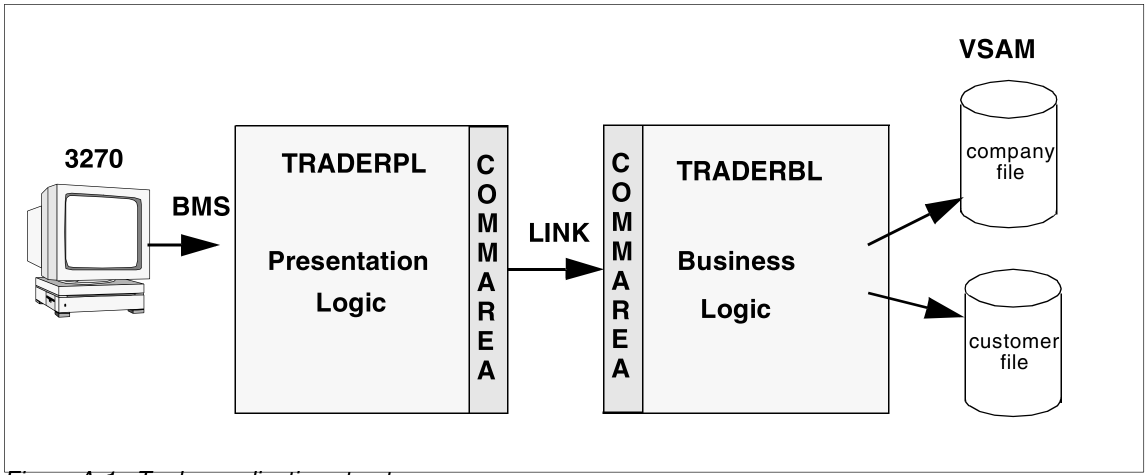


Figure A-1 Trader application structure

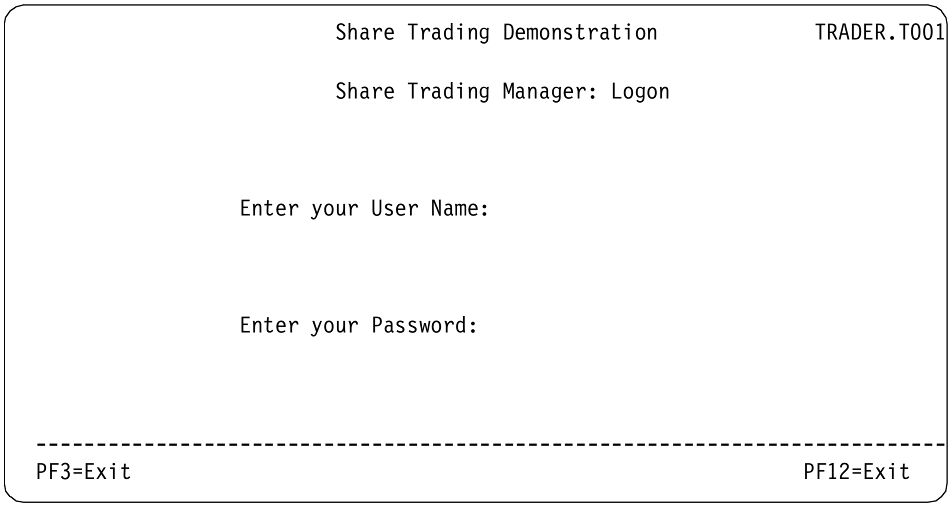
At each step, the application presents a set of options. The user makes a choice, then presses the required key in order to send their selections back to the application. The application performs the necessary actions based on the user's choice and presents the results together with any possible new options. The application has a strict hierarchical menu structure which allows the user to return to the previous step by using the PF3 key.

3270 application flows

In this section, we describe a typical business transaction when using the 3270 Trader application:

1. The program TRADERPL is invoked on a 3270 capable terminal by entering the initial CICS transaction identifier (TRAD). TRADERPL calls TRADERBL, passing an inter-program COMMAREA of 400 bytes. TRADERBL expects the COMMAREA to contain a request type and associated data. There are three request types: *Get_Company* to return a company list, *Share_Value* to return a list of share values, or *Buy_Sell* to buy or sell shares. In this step the request type is *Get_Company*.

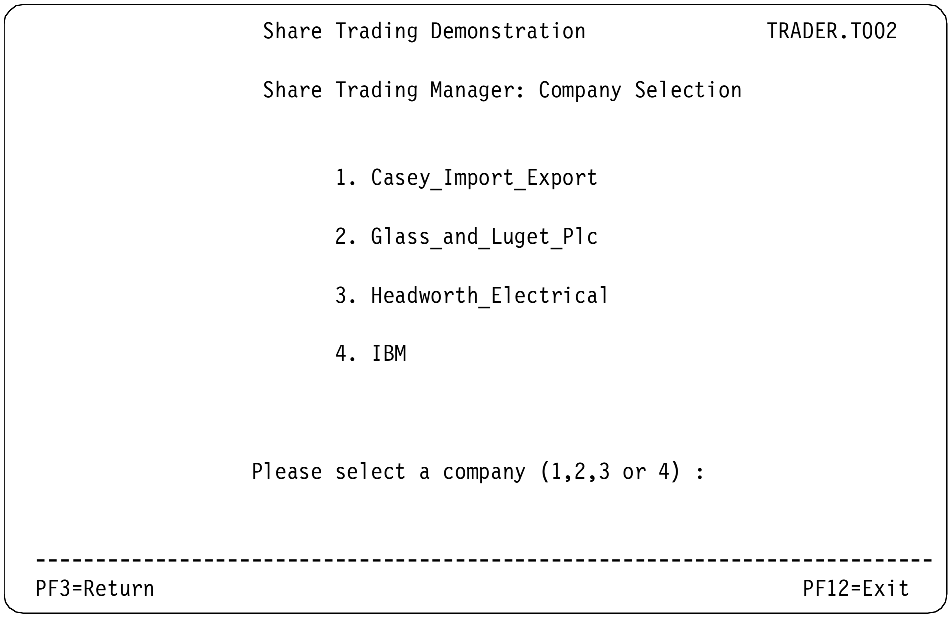
   When TRADERBL receives a *Get_Company* request, it browses the company file and returns the first four entries to TRADERPL. At this point the user has not entered any request, but the application assumes that a Get_Company request will be following. TRADERPL then sends the *signon display* (T001 shown in Figure A-2), which prompts for a user ID and password. The list of companies is stored in the COMMAREA associated with the terminal when the TRAD transaction ends, so that it will be available at the next task in the pseudo-conversational sequence.



```
                    Share Trading Demonstration          TRADER.T001

                    Share Trading Manager: Logon



              Enter your User Name:



              Enter your Password:



 ----------------------------------------------------------------------------
 PF3=Exit                                                          PF12=Exit
```

Figure A-2 Trader signon display

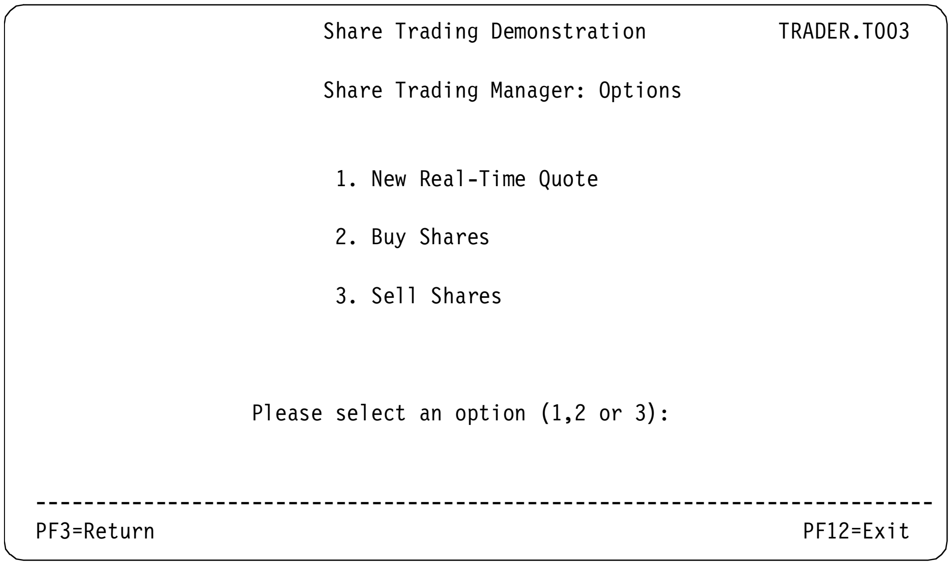
2. The next transaction invokes TRADERPL, which receives the *signon display* (T001) and the saved COMMAREA from step 1. If security is active in the CICS region the user ID and password entered will be used to signon to CICS. Then the using the company data acquired in step 1, TRADERPL sends the *company selection display* (T002), the format of which is shown in Figure A-3. TRADERPL then returns, specifying the next transaction to run and the associated COMMAREA.



```
                    Share Trading Demonstration              TRADER.T002

                    Share Trading Manager: Company Selection


                          1. Casey_Import_Export

                          2. Glass_and_Luget_Plc

                          3. Headworth_Electrical

                          4. IBM


                   Please select a company (1,2,3 or 4) :


 ---------------------------------------------------------------------------
 PF3=Return                                                        PF12=Exit
```

Figure A-3 Company selection display

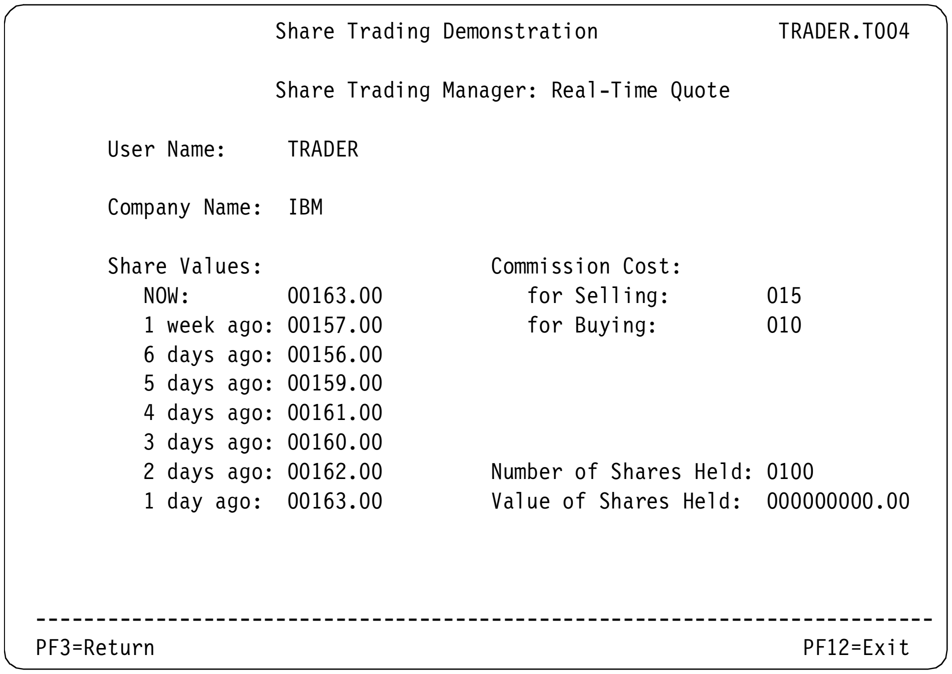
3. The user selects the company to trade from the *Company Selection* display, and presses Enter. The program TRADERPL is invoked and sends the *Options display* (T003, shown in Figure A-4) to the terminal. The user can now decide whether to buy, sell, or get a new real-time quote. TRADERPL returns, specifying the next transaction to run and the associated COMMAREA.



```
                        Share Trading Demonstration          TRADER.T003

                        Share Trading Manager: Options


                         1. New Real-Time Quote

                         2. Buy Shares

                         3. Sell Shares


                  Please select an option (1,2 or 3):


---------------------------------------------------------------------------
PF3=Return                                                       PF12=Exit
```

Figure A-4   Options menu display

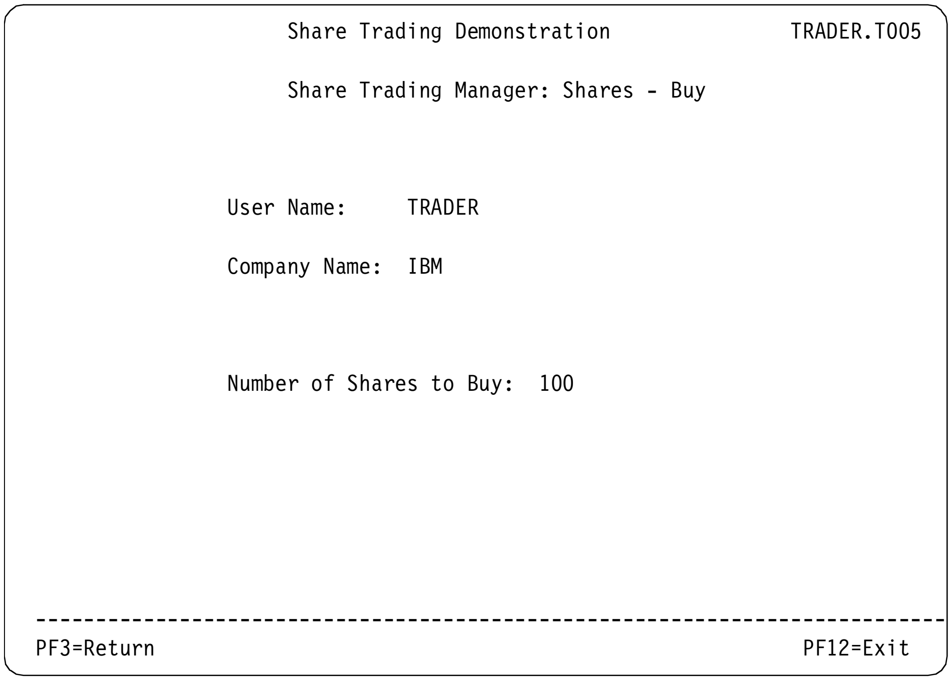
4. The user then selects Option **1** and presses the Enter key. TRADERPL is invoked and determines that the user's request is a *Share_Value* request type. TRADERPL calls TRADERBL, passing the request type and the company selected earlier. TRADERBL reads the customer file to determine how many shares are held, then reads the company file to determine the price history, and returns the information to TRADERPL. TRADERPL uses this data to build a *Real-Time Quote* display (T004) as illustrated in Figure A-5. This display shows the recent history of share values for the company chosen, the number of shares held with this company, and the total value of these shares. TRADERPL returns, specifying the next transaction to run and the associated COMMAREA data.



```
                 Share Trading Demonstration              TRADER.T004

                 Share Trading Manager: Real-Time Quote

      User Name:     TRADER

      Company Name:  IBM

      Share Values:                    Commission Cost:
         NOW:         00163.00            for Selling:        015
         1 week ago: 00157.00            for Buying:         010
         6 days ago: 00156.00
         5 days ago: 00159.00
         4 days ago: 00161.00
         3 days ago: 00160.00
         2 days ago: 00162.00          Number of Shares Held: 0100
         1 day ago:  00163.00          Value of Shares Held:  000000000.00



 ---------------------------------------------------------------------------
 PF3=Return                                                      PF12=Exit
```

Figure A-5 Real-time quote display

5. The user now presses **PF3** to go back to the *Options Menu*. TRADERPL is invoked and sends the *Options display* (T003) to the terminal (repeating the actions of Step 3), and returns, specifying the next transaction to run and the associated COMMAREA data.
6. The user now requires to purchase shares, so selects Option **2** and presses the Enter key. Program TRADERPL receives map T003 and determines that the user wants to buy shares, and sends the *Shares-Buy* display (T005) shown in Figure A-6. TRADERPL returns, specifying the next transaction to run and the associated COMMAREA.



```
                    Share Trading Demonstration            TRADER.T005

                    Share Trading Manager: Shares - Buy



               User Name:     TRADER

               Company Name:  IBM



               Number of Shares to Buy:  100








--------------------------------------------------------------------------
PF3=Return                                                       PF12=Exit
```

Figure A-6 Shares - Buy display

7. Program TRADERPL receives the T005 screen and builds a *Buy_Sell* request COMMAREA which is passed to program TRADERBL. TRADERBL reads the company file and then performs a READ for UPDATE and REWRITE to the customer file to update the customers share holdings. The success of the request is returned to TRADERPL in the COMMAREA, and TRADERPL sends the *Options display* (T003) reporting the successful buy to the user. TRADERPL returns, specifying the next transaction to run and the associated COMMAREA.
8. Next the user checks his share holdings by repeating Step 4.
9. The user returns to the options screen by repeating Step 5.
10. The business transaction is completed by the user pressing PF12, which performs an EXEC CICS SEND TEXT to write a message to the terminal and reports that the session is complete. TRADERPL then executes the final EXEC CICS RETURN command. No COMMAREA is specified because the pseudo-conversation is over, and there is no conversation state data to retain.

CICS resource definitions

To install the COBOL Trader application the following CICS resources need to be created:

- **Files**

  Trader uses the following two VSAM files:

  – COMPFILE

    This file is used to store the list of companies and associated share prices. It can be created using the supplied JCL `TRADERCOCJL.TXT` which requires as input the file `TRADERCODATA.TXT`

  – CUSTFILE

    This file is used to store the list of users and share holdings. It can be created using the supplied JCL `TRADERCUJCL.TXT`

- **Transactions**

  The 3270 version of Trader requires just one transaction **TRAD**, which should specify the program TRADERPL

- **Programs**

  CICS program definitions are only required if program autoinstall is not active. The 3270 trader application uses two COBOL programs which will need to compiled and placed in a dataset in your CICS region DFHRPL concatenation. The COBOL source code for these applications is available with the sample source code with this book. For details on how to download this refer to Appendix B, “Additional material” on page 87.

  – TRADERPL

    This contains the 3270 presentation logic and is invoked by transaction TRAD.

  – TRADERBL

    This contains the business logic and is invoked by program TRADERPL, or it can be linked to from another application that contains its own presentation logic, such as a Java servlet.

- **Mapset**

  Trader uses the Mapset **NEWTRAD** which comprises the maps T001, T002, T003, T004, T005 and T006. The Mapset is supplied in the file `NEWTRAD.TXT` and will need to be assembled and link-edited, and the load module placed in a dataset in your CICS region DFHRPL concatenation.

For further information on creating the resource definitions for Trader, refer to the supplied file `TRADERRDO.TXT`, which contains the output of a CSD extract for the Trader application.

B

Additional material

This Redpaper refers to additional material that can be downloaded from the Internet as described below.

Locating the Web material

The Web material associated with this Redpaper is available in softcopy on the Internet from the IBM Redbooks Web server. Point your Web browser to:

`ftp://www.redbooks.ibm.com/redbooks/SG24????`

Alternatively, you can go to the IBM Redbooks Web site at:

ibm.com/redbooks

Select the **Additional materials** and open the directory that corresponds with the redbook form number, SG24????.

Using the Web material

The additional Web material that accompanies this Redpaper includes the following files:

File name	*Description*
Connector_code.zip	Sample code required by this redpaper.

System requirements for downloading the Web material

The following system configuration is recommended:

Hard disk space:	700MB minimum
Operating System:	Windows 98 / Windows NT or higher
Processor:	Pentium(R) II 300 processor or higher recommended
Memory:	256MB

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. The following directories will be created:

VisualAge for Java	Source code required by the VisualAge for Java chapter.
WebSphere Studio	Source code required by the WebSphere Studio chapter.
Trader-Cobol	The COBOL source code for the Trader CICS application.
Completed Trader	The completed Trader enterprise application.

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this Redpaper.

IBM Redbooks

For information on ordering these publications, see “How to get IBM Redbooks” on page 89.

- *Java Connectors for CICS,* SG24-6401

Other resources

These publications are also relevant as further information sources:

- *WebSphere Application Server V4.0.1 for z/OS Installation and Customization,* GA22-7834
- *WebSphere Application Server V4.0.1 for z/OS Assembling Java 2 Platform, Enterprise Edition (J2EE) Applications,* SA22-7836

How to get IBM Redbooks

You can order hardcopy Redbooks, as well as view, download, or search for Redbooks at the following Web site:

`ibm.com/redbooks`

You can also download additional materials (code samples or diskette/CD-ROM images) from that site.

IBM Redbooks collections

Redbooks are also available on CD-ROMs. Click the CD-ROMs button on the Redbooks Web site for information about all the CD-ROMs offered, as well as updates and formats.

Index

From code to deployment: Connecting to CICS from WebSphere V4.01 for z/OS

Redpaper

How to connect to CICS from WebSphere V4.01 for z/OS

What to use for development

Use VisualAge for Java and WebSphere Studio

In March 2002, WebSphere Application Server V4.0.1 for z/OS introduced support for the J2EE Connector Architecture, and with this support comes a whole new way of connecting to enterprise systems like CICS and IMS.

This Redpaper shows how to use the J2EE Connector Architecture support to create a managed connection between WebSphere for z/OS and CICS Transaction Server, using the CICS ECI resource adapter. The entire process is covered in this paper, from developing an enterprise application that makes calls to CICS, through to deploying the enterprise application to WebSphere for z/OS.

This Redpaper describes detailed instructions on how to prepare IBM's VisualAge for Java and the new WebSphere Studio configurations for use with the CICS ECI resource adapter, and then details how to build an enterprise application using these tools. This enterprise application is then deployed to a WebSphere J2EE server configured for the CICS ECI resource adapter.