WebSphere. software

IBM

# WebSphere on z/OS - Optimized Local Adapters

- Exploit high-performance bi-directional communications between WebSphere and CICS applications
- Installation and configuration examples
- Sample application scenarios

G. Michael Connolly
Mitch Johnson
Edward McCarthy
Joerg-Ulrich Veser



ibm.com/redbooks

Redpaper

International Technical Support Organization

# WebSphere on z/OS - Optimized Local Adapters

August 2009



REDP-4550-00

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



**First Edition (August 2009)**

This edition applies to WebSphere Application Server Version 7, Fix Pack 4

# Contact an IBM Software Services Sales Specialist

Start SMALL, Start BIG, ... **JUST START**

architectural knowledge, skills, research and development . . .

**that's IBM Software Services for WebSphere.**

Our highly skilled consultants make it easy for you to design, build, test and deploy solutions, helping you build a smarter and more efficient business. **Our worldwide network of services specialists wants you to have it all!** Implementation, migration, architecture and design services: IBM Software Services has the right fit for you. We also deliver just-in-time, customized workshops and education tailored for your business needs. You have the knowledge, now reach out to the experts who can help you extend and realize the value.

For a WebSphere services solution that fits your needs, contact an IBM Software Services Sales Specialist:
**ibm.com**/developerworks/websphere/services/contacts.html

# Contents



# Notices

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

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

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

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

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

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

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

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

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

COPYRIGHT LICENSE:

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

# Trademarks

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

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

CICS®
DB2®
IBM®
IMS™
RACF®
Rational®
Redbooks®
Redpaper™
Redbooks (logo) ®
System z®
WebSphere®
z/OS®
zSeries®

The following terms are trademarks of other companies:

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

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

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

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

# Preface

This IBM® Redpaper™ publication describes the steps involved in the installation, configuration and implementation of the new Optimized Local Adapters (OLA) support available with WebSphere® Application Server.

A step-by-step approach is used to guide you through the OLA installation and configuration process.

The OLA bi-directional communications functions are presented in detail through the development, deployment and execution phases using three sample application scenarios:

- Modify the existing IBM benchmark application Trade6 to enable it to receive OLA inbound calls from an external application written in C.
- A CICS® Cobol program modified to invoke an EJB™ within the WebSphere Application Server on z/OS®.
- An EJB in the WebSphere Application Server on z/OS invoking a Cobol program in CICS.

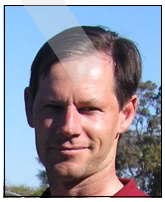
## The team who wrote this paper

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

The terms WebSphere Optimized Local Adapters and Optimized Local Adapters (OLA) are identical in meaning and are used interchangeably throughout this publication.

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

**Mitch Johnson** is a Senior Software Engineer in IBM Software Services for WebSphere (ISSW) at IBM Research Triangle Laboratory. His areas of expertise include enterprise connectivity and installation, configuration, and administration of WebSphere for z/OS and IMS™, as well as CICS, TXSeries, and DB2® on various platforms.



**Edward McCarthy** currently works in the e-business Enablement Services team for IBM Global Services Australia. The team is responsible for all aspects of WebSphere Application Server and WebSphere MQ across all platforms. For the last five years, he has specialized in supporting the WebSphere range of products. He worked as a senior CICS and WebSphere MQ systems programmer for over eight years with a large IBM client.

**Joerg-Ulrich Veser** is an IT Specialist working since 2006 in the pre-sales support for WebSphere on z/OS in Germany. His areas of expertise include infrastructure architecture design, implementation, problem determination, high availability and security on WebSphere products for z/OS. He holds a degree in Computer Science from the University of Cooperative Education in Mannheim, Germany.

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

Rich Conway - International Technical Support Organization, Poughkeepsie, NY
IBM USA

James Mulvey - Websphere Application Server for z/OS development
IBM USA

Timothy Kaczynski - Websphere Application Server for z/OS development
IBM USA

Colette Manoni - WebSphere Application Server for z/OS architecture
IBM USA

Don Bagwell - Advanced Technical Support for Z/OS, IBM Washington Systems Center
IBM USA

Dennis Behm - IT Specialist - Field Technical Professional for Rational/WebSphere on System z®
IBM Germany

## Become a published author

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

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

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

ibm.com/redbooks/residencies.html

## Comments welcome

Your comments are important to us!

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

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

  ibm.com/redbooks

- Send your comments in an e-mail to:

  redbooks@us.ibm.com

- Mail your comments to:

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

1

# Introduction to OLA

Optimized Local Adapters (OLA) is a new function provided with WebSphere Application Server for z/OS maintenance level 7.0.0.4, released May 19th, 2009.

This chapter briefly covers the following OLA features:

- Efficient cross-memory transfer
- Bi-directional capability
- Security propagation
- Transaction propagation

## 1.1 OLA overview

Optimized Local Adapters is a new method of cross-memory local communications between WebSphere Application Server for z/OS and external address spaces such as CICS, batch programs, and Unix Systems Services (USS) programs.

The terminology, WebSphere Optimized Local Adapters and Optimized Local Adapters (OLA), is identical in meaning and is used interchangeably throughout this publication.

The new function is bi-directional—from WebSphere Application Server for z/OS to the external address space, or from the external address space into WebSphere Application Server for z/OS.

The key advantages of this new function can be summarized as follows:

- Very efficient cross-memory transfer from WebSphere Application Server to the external address space, or from the external address space into WebSphere Application Server.
- Bi-directional capability – You can leverage WebSphere Application Server EJB assets as local services from external address spaces such as CICS or batch programs.
- Security propagation – From WebSphere Application Server you can flow the user ID, the servant ID, or the EJB role ID into the external address space; or from the external address space you can flow the client ID or, in the case of CICS, the CICS region ID or the CICS task userid.
- Transaction propagation – When operating from CICS into WebSphere Application Server, the latter can participate in a CICS unit of work for two-phase commit processing. However, in the initial release of WOLA in Version 7.0.0.4, transaction is not supported for flows from WebSphere Application Server into CICS, otherwise known as *outbound* from WebSphere Application Server.

In summary, the Optimized Local Adapter represents a way to link WebSphere Application Server for z/OS and external address spaces in an optimized, high-speed, cross-memory, bi-directional manner.

An excellent overview that covers the OLA functions and their applicable environments is available for download at:

`http://www-03.ibm.com/support/techdocs/atsmastr.nsf/WebIndex/WP101490`

## 1.2 OLA Redpaper

The following Redpaper chapters present our experiences in the installation, configuration and bi-directional application exploitation of the OLA functions.

Chapter 2, “WebSphere Optimized Local Adapters - Installation and configuration” on page 5 covers the steps we took to install and configure the WOLA feature within our ITSO WebSphere Application Server environment. The configuration steps are described using both the wsadmin shell script method and the Administrative Console method. Additionally, we utilized the OLACC01 sample program shipped with the WOLA feature as the mechanism for performing our WOLA installation verification test. We document the steps we took in setting up and running the OLACC01 sample.

Chapter 3, “WebSphere Optimized Local Adapters-enabled Trade application” on page 23 shows how to modify the existing IBM Trade6 application to receive inbound calls from an application written in C. We show how we programmed the C application to make the WOLA function calls into the WebSphere Trade6 application.

Chapter 4, “OLA - CICS to EJB in WebSphere” on page 37 demonstrates how a COBOL program running in CICS can use the WOLA support to invoke an EJB located in the WebSphere Application Server. The use of Rational® Application Developer for the development of the target EJB is explained as well as the modifications made to our sample COBOL program to use the WOLA APIs to invoke the business logic contained in the EJB.

Chapter 5, “WebSphere Optimized Local Adapter - Outbound to CICS scenario” on page 77 demonstrates the development of an EJB using RDz, which implements the WOLA classes accessing CICS applications that utilize both the CICS COMMAREA and container interfaces.

2

# WebSphere Optimized Local Adapters - Installation and configuration

This chapter covers the installation and configuration of the WebSphere for z/OS Optimized Local Adapters.

It covers the following topics:

- Installation of the WOLA load libraries
- Configuration of the WOLA support using either the olaRar.py script or the Administrative Console
- Validating the WOLA-enabled environment

## 2.1 Prerequisites

In order to benefit from this new feature, the following requirements need to be complied with:

- WebSphere Application Server for z/OS V7.0.0.4

  The WebSphere Optimized Local Adapters will be provided with the service stream of WebSphere Application Server for z/OS V7 Fix pack 4. No additional feature pack is required. All necessary resources are provided with the SMP/E HFS of WebSphere Application Server.

- The EJB application, which makes use of WebSphere Optimized Local Adapter inbound or outbound connections, need to be deployed to an application server, which must run in 64-bit mode.

  If you have migrated from previous versions of WebSphere Application Server for z/OS, the target application server will still be configured for 31-bit mode unless you have made use of 64-bit mode before.

  In the case of an application server with 31-bit mode, you can easily switch it to 64-bit by checking "**Run in 64 bit JVM™ mode**" in the application server's **<appserver_name>** panel of the admin console without the need to modify the JCL start procedures.

  With WebSphere Application Server V7.0, the default mode is 64-bit for the application server.

In WebSphere Application Server V7.0.0.4 the default configuration setting for the WebSphere Optimized Local Adapter is *inactive*. It can be activated on a node-by-node basis. To activate the WebSphere Optimized Local Adapter, the following tasks are performed:

- Create the WebSphere Optimized Local Adapter load library and symbolic links in the WebSphere configuration HFS by running the olainstall.sh shell script.
- Change the node configuration by running the olaRAR.py Jython script. Alternatively, these configuration changes can be manually entered via the admin console.
- Set RACF® CBIND Permissions for the external address spaces.
- Perform an Installation Verification Test (IVT) for the WebSphere Optimized Local Adapter environment.

Figure 2-1 on page 7 diagrams the WebSphere Optimized Local Adapter installation and configuration steps.

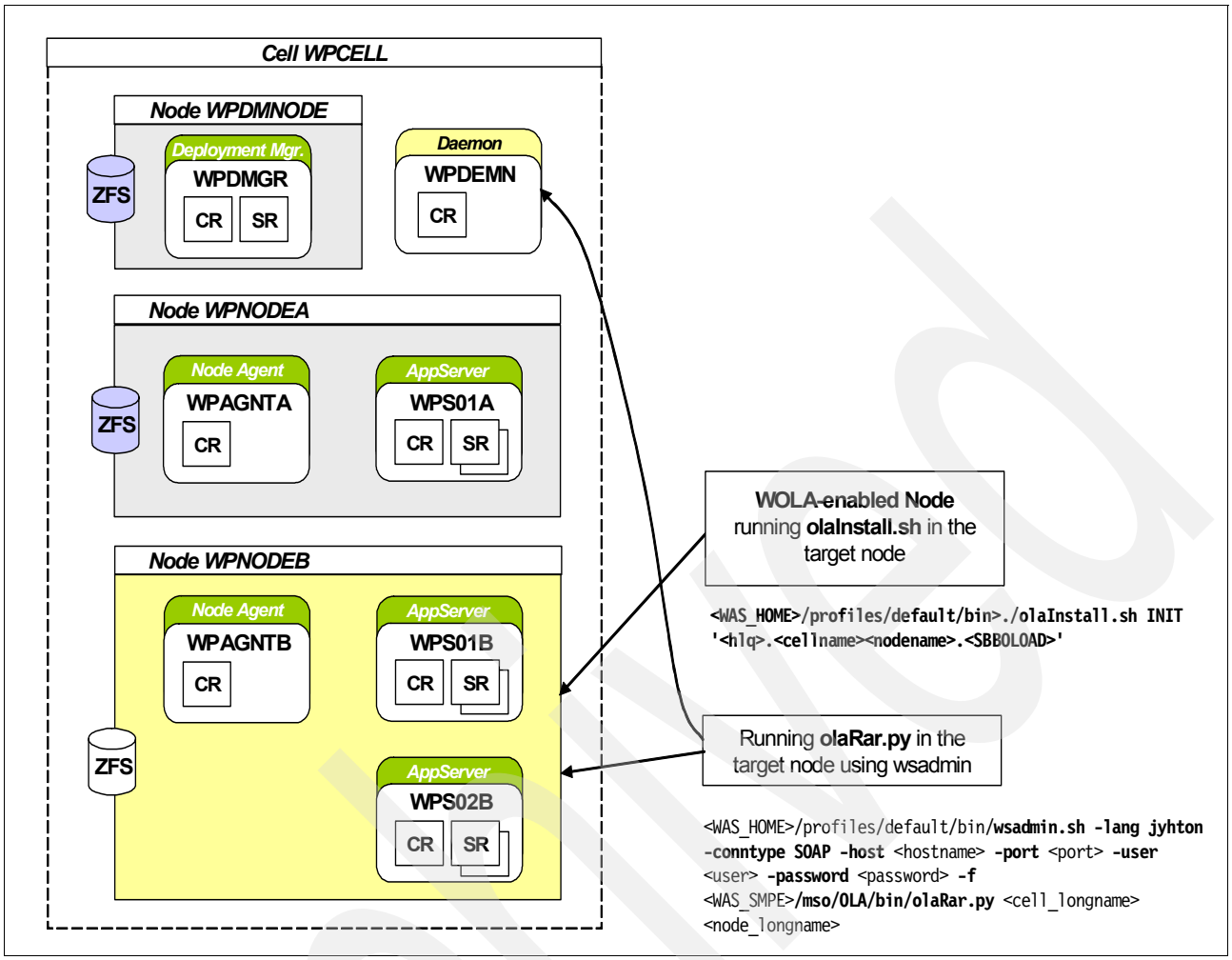



*Figure 2-1 Overview of WOLA installation and configuration*

## 2.2 WebSphere Optimized Local Adapter load libraries

To install and configure the WebSphere Optimized Local Adapter, additional WebSphere Optimized Local Adapter load libraries and additional symbolic links in the WebSphere configuration HFS need to be set up. One of the tasks of the olaInstall.sh script is to create additional symbolic links in the WebSphere configuration HFS, in order to link to the ola modules and plug-in jar files contained in the product HFS. Moreover, the olaInstall.sh script copies the ola load modules from the product HFS into a PDS data set. This PDS data set containing the load modules is available for external address spaces to utilize the WebSphere Optimized Local Adapter functions in order to connect to a WebSphere Application Server. Therefore, this data set needs to be in the STEPLIB of the external address space or in the LNKLST concatenation. However, it is not necessary to include this data set in the STEPLIB of the WebSphere Application Server start procedures (JCL).

Introduced with V7.0, the load libraries for WebSphere Application Server are now included in the product HFS instead of separate MVS data sets. Consequently the load module libraries do not have to be added to the STEPLIB of the WebSphere address spaces.

The olaInstall.sh shell script is located in the <WAS_HOME>/profiles/default/bin/ directory of each node. It needs to be executed for each node that requires WebSphere Optimized Local Adapter support.

**Attention:** Although not a requirement, we recommend that you run the olaInstall.sh on the deployment manager node, if you make use of EJB deploy at install time. This is the case for instance, if you create an EJB and do not generate deploy code with Rational Application Developer (RAD) or Rational Developer for System z (RDz). The alternative would be to provide the class path to ola_apis.jar in the SMP/E HFS during EJB deployment as shown in “Specify class path during EJB deployment” on page 9.

Prior to the execution of the olaInstall.sh script, a new PDS data set needs to be allocated, into which the ola load modules will be copied. Example 2-1 shows the values we used for allocating the OLA SBBOLOAD data set.

*Example 2-1 OLA SBBOLOAD data set*

```
General Data                            Current Allocation
 Management class . . : **None**         Allocated cylinders : 2
 Storage class  . . . : **None**         Allocated extents . : 1
  Volume serial . . . : WAS04B           Maximum dir. blocks : NOLIMIT
  Device type . . . . : 3390
 Data class . . . . . : **None**
  Organization  . . . : PO              Current Utilization
  Record format . . . : U                Used pages  . . . . : 203
  Record length . . . : 0                % Utilized  . . . . : 56
  Block size  . . . . : 32760            Number of members . : 24
  1st extent cylinders: 2
  Secondary cylinders : 1
  Data set name type  : LIBRARY
```

The olaInstall.sh script can be invoked using the following command:

```
<WAS_HOME>/profiles/default/bin>./olaInstall.sh INIT
'<hlq>.<cellname><nodename>.<SBBOLOAD>'
```

Our invokation of the olaInstall.sh script and the resulting output messages are shown in Example 2-2.

*Example 2-2 Output of olaInstall.sh*

```
JVESER @ SC04:/wasconfig/wpcell/wpnodeb/AppServer/profiles/default/bin>./olaInstall.sh INIT
'BBMC7004.WPCELL.WPNODEB.SBBOLOAD'
processing ...
function: INIT
WAS_HOME = /wasconfig/wpcell/wpnodeb/AppServer
WAS_CELL = WPCell
WAS_SMPE_ROOT = /wasconfig/wpcell/wpnodeb/wassmpe
File system owner: WPADMIN
File system group: WPCFG

Creating directory: /wasconfig/wpcell/wpnodeb/AppServer/lib/olaModules

set owner and group for /wasconfig/wpcell/wpnodeb/AppServer/lib/olaModules
```

```
link /wasconfig/wpcell/wpnodeb/AppServer/lib/olaModules to
/wasconfig/wpcell/wpnodeb/wassmpe/mso/OLA/lib/olaModules/*

link /wasconfig/wpcell/wpnodeb/AppServer/plugins/com.ibm.ws390.ola.jar to
/wasconfig/wpcell/wpnodeb/wassmpe/mso/OLA/plugins/com.ibm.ws390.ola.jar

link /wasconfig/wpcell/wpnodeb/AppServer/installableApps/ola.rar to
/wasconfig/wpcell/wpnodeb/wassmpe/mso/OLA/installableApps/ola.rar

link /wasconfig/wpcell/wpnodeb/AppServer/lib/modules/bbgadapt to
/wasconfig/wpcell/wpnodeb/wassmpe/mso/OLA/lib/modules/bbgadapt

link /wasconfig/wpcell/wpnodeb/AppServer/lib/bbgadapt to libraries

link /wasconfig/wpcell/wpnodeb/AppServer/lib/libbbgadapt.so to libraries

add external link for bbgadapt to /wasconfig/wpcell/wpnodeb/AppServer/lib/links

add external link for libbbgadapt.so to /wasconfig/wpcell/wpnodeb/AppServer/lib/links

Run osgiCfgInit
OSGi cache successfully cleaned for /wasconfig/wpcell/wpnodeb/AppServer/profiles/default.

Copying /wasconfig/wpcell/wpnodeb/AppServer/lib/olaModules/* to
BBMC7004.WPCELL.WPNODEB.SBBOLOAD...
/wasconfig/wpcell/wpnodeb/AppServer/lib/olaModules/bboa1cng ->
//'BBMC7004.WPCELL.WPNODEB.SBBOLOAD(bboa1cng)': executable

/wasconfig/wpcell/wpnodeb/AppServer/lib/olaModules/bboa1cnr ->
//'BBMC7004.WPCELL.WPNODEB.SBBOLOAD(bboa1cnr)': executable

/wasconfig/wpcell/wpnodeb/AppServer/lib/olaModules/bboa1get ->
//'BBMC7004.WPCELL.WPNODEB.SBBOLOAD(bboa1get)': executable

...
```

### 2.2.1 Specify class path during EJB deployment

If you create an EJB within an enterprise application and do not generate deploy code with Rational tooling prior to deployment, EJB deploy will be executed automatically during install time. Consequently the deployment manager needs to be able to access the WebSphere Optimized Local Adapter APIs during deployment. Either the olaInstall.sh shell script needs to be run against the deployment manager node, or the location of ola_apis.jar needs to be specified in the class path during EJB deployment. The ola_apis.jar is located in `<SMPE_HFS>/mso/OLA/lib/ola_apis.jar`.

In order to specify this class path, select **Deploy enterprise beans** in the install options during the deployment; see Figure 2-2 on page 10.

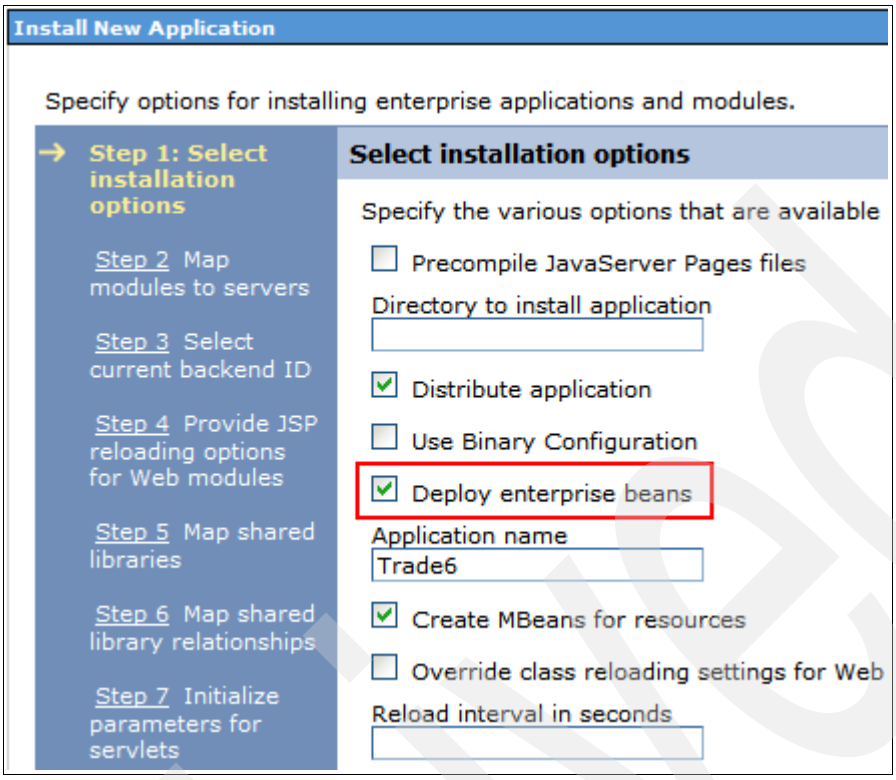



*Figure 2-2 Check "Deploy enterprise beans"*

In the step "Provide options to perform the EJB deploy" the class path to ola_apis.jar can be specified, as shown in Figure 2-3 on page 11.

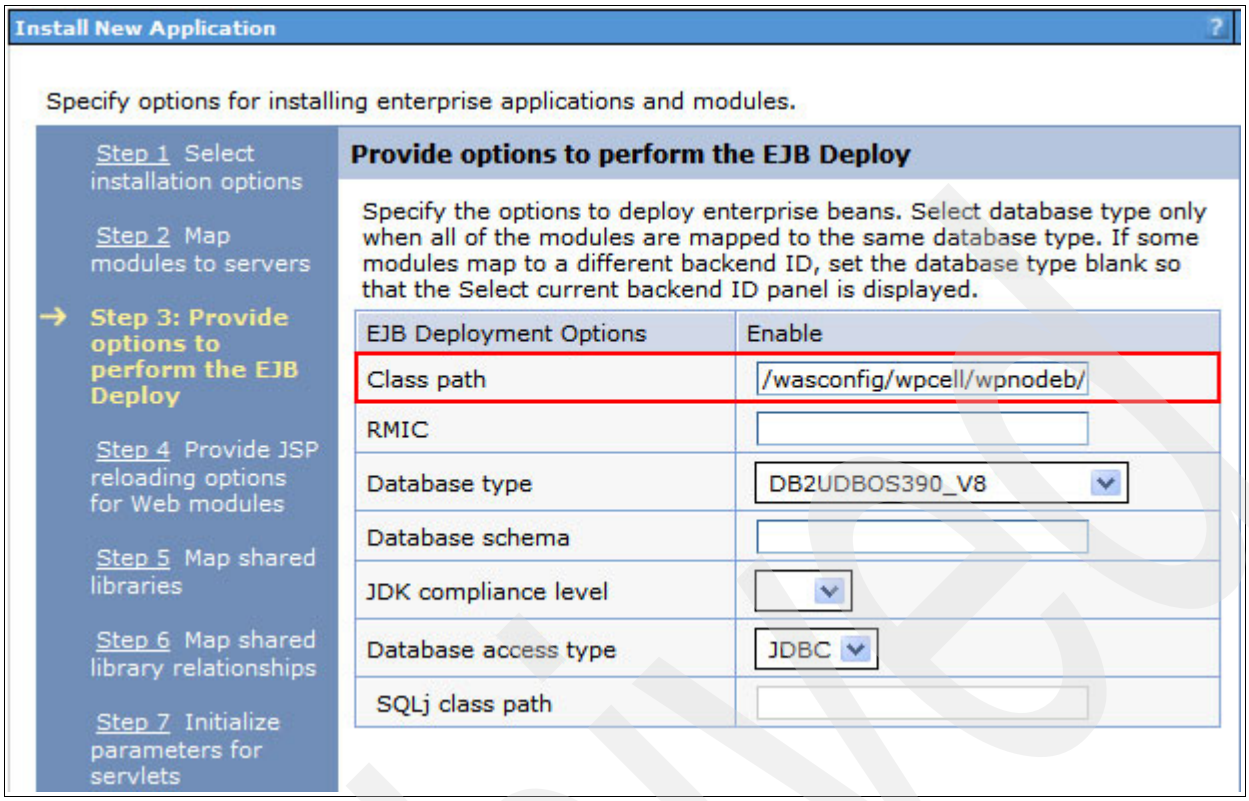



*Figure 2-3 Specify class path for EJB Deploy*

## 2.3 Configuration

So far the olaInstall.sh shell script has set up the WebSphere Optimized Local Adapter load libraries and symbolic links for the WebSphere configuration HFS. To complete the WebSphere Optimized Local Adapter installation and configuration, the olaRar.py Jython script needs to be run.

With WebSphere Application Server for z/OS V7.0 Fix pack 4 a dedicated Jython script is provided for the installation of the WebSphere Optimized Local Adapter JCA adapter. This Jython script is named olaRar.py and is located in the `/<smpe_root>/mso/OLA/bin/` directory of the SMP/E HFS.

The olaRar.py script sets the environment variable `WAS_DAEMON_ONLY_enable_adapter = true` for the cell scope and creates the resource adapter along with a corresponding J2C connection factory.

The following example shows how to invoke the olaRar.py Jython script using the wsadmin interface. If administrative security is being used then a user name and password must be provided. In addition, the cell long name and node long name need to be specified as parameters for the Jython script. Both parameters build the scope in which the WebSphere Optimized Local Adapter JCA will be installed.

```
<WAS_HOME>/profiles/default/bin/wsadmin.sh -lang jyhton -conntype SOAP -host
<hostname> -port <port> -user <user> -password <password> -f
<WAS_SMPE>/mso/OLA/bin/olaRar.py <cell_longname> <node_longname>
```

The messages shown in Example 2-3 indicate a successful execution of the olaRar.py script.

*Example 2-3 Sample output of olaRar.py*

```
JVESER @ SC04:/wasconfig/wpcell/wpnodeb/AppServer/profiles/default/bin>wsadmin.sh
-lang Jython -conntype SOAP -host wtsc04.itso.ibm.com -port 12002 -user jveser
-password jveser -f /SC04/zWebSphereMC/mso/OLA/bin/olaRar.py WPCell WPNodeB

WASX7209I: Connected to process "dmgr" on node WPDmNode using SOAP connector;  The
type of process is: DeploymentManager
WASX7303I: The following options are passed to the scripting environment and are
available as arguments that are stored in the argv variable: "[WPCell, WPNodeB]"

setting WAS_DAEMON_ONLY_enable_adapter = true

Installing resource adapter

create J2C connection factory

Validating configuration...
Saving configuration changes...
```

**Tip:** In order to avoid SSL problems caused by missing signer certificates in the corresponding SAF Keyring, the wsadmin.sh should be executed by the wsadmin user, which has been defined during the product customizing. The user can be changed with the OMVS command **su wsadmin**.

After successful execution of both installation scripts, a restart of the daemon address space is requireded for activating the WebSphere Optimized Local Adapter support. In the job output of the restarted daemon, the following message should appear:

```
BBOM0001I          enable_adapter: 1.
```

### 2.3.1 Manually change settings in the admin console

**Note:** Perform these steps using the Admin console only if the olaRar.py script was not utilized during installation.

First, using the admin console, the environment variable WAS_DAEMON_ONLY_enable_adapter = true needs to be defined at the cell scope. It enables the WebSphere Optimized Local Adapter JCA and should appear in the daemon job log during startup.

- Go to **Environment** → **WebSphere variables**.
- Select the cell scope and click on**New**.
- Insert `WAS_DAEMON_ONLY_enable_adapter` as **Name** and `true` as **Value**.
- Click **OK** to create the environment variable.
- Save the changes and synchronize with nodes.

The following tasks need to be performed in order to install the WebSphere Optimized Local Adapter JCA and the corresponding connection factory:

- Go to **Resources** → **Resource adapters**.
- Select the node as scope, where your WebSphere Optimized Local Adapter target server belongs and click **Install RAR**.

  Choose the Remote file system and specify the full path to the ola.rar file, which is located in `<WAS_HOME>/installableApps/ola.rar` and click **Next**, as shown in Figure 2-4.

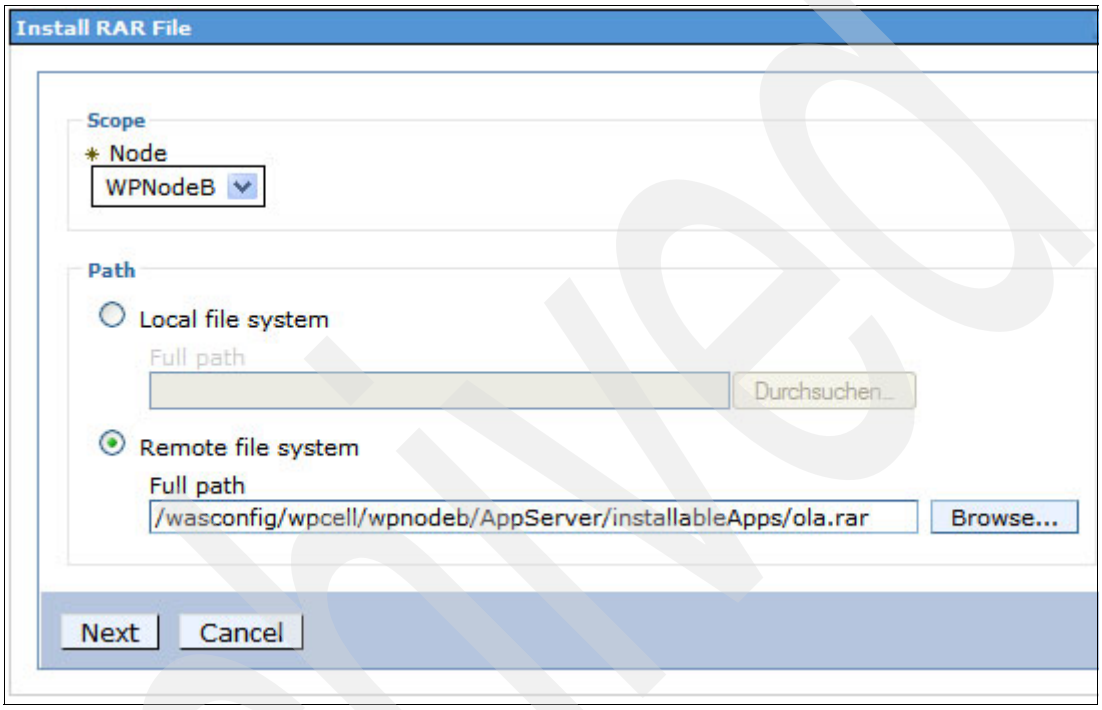



*Figure 2-4 Specify full path to the ola.rar file*

- You can take the defaults for configuring the resource adapter and click **OK**, as shown in Figure 2-5 on page 14.

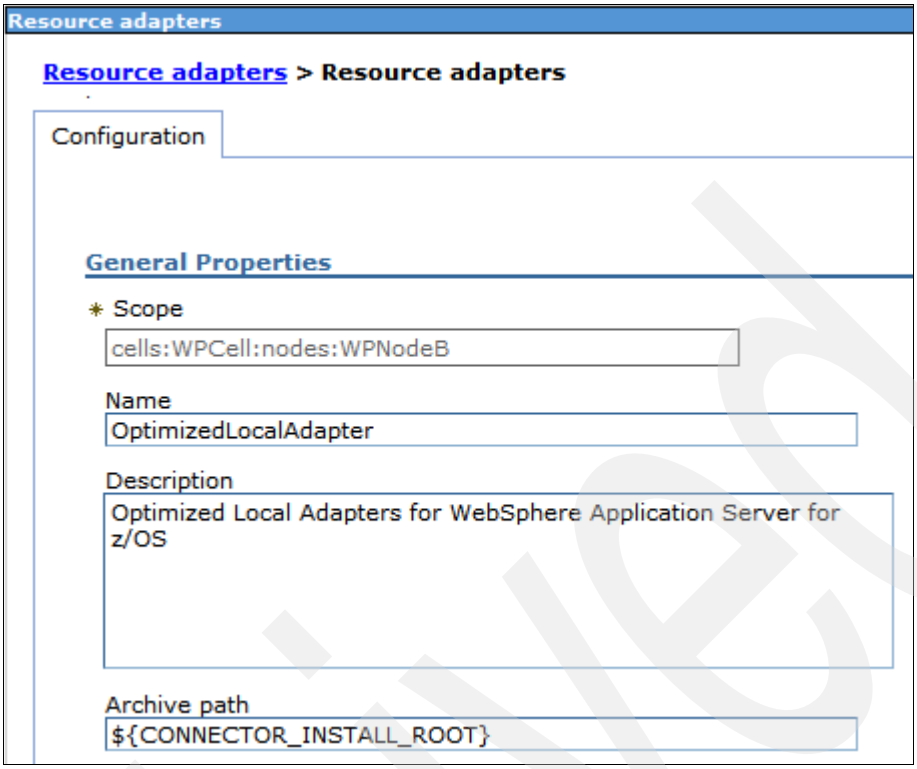



*Figure 2-5 Specify resource adapters*

- Save changes and synchronize with nodes.
- All resource adapters belonging to the selected node will be displayed in this overview.

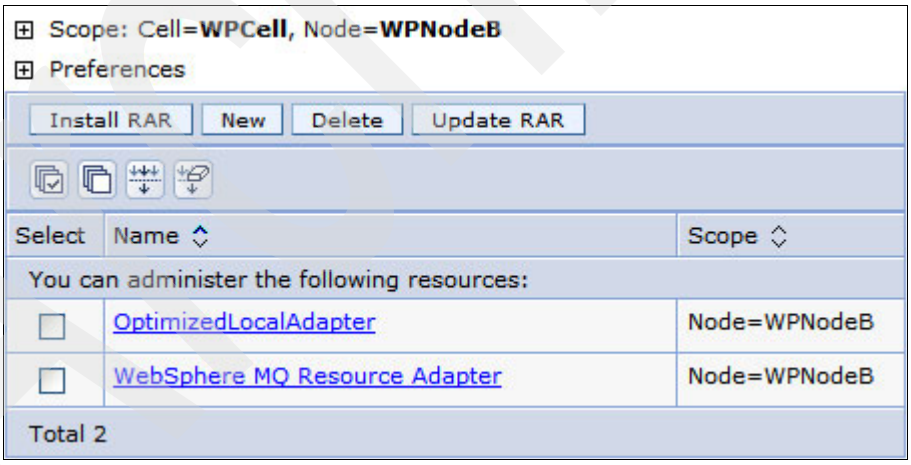



*Figure 2-6 Resource adapter overview*

- Click **OptimizedLocalAdapter** and select **J2C connection factories** in the right navigation bar.
- Click **New**.
- In our example, shown in Figure 2-7 on page 15, we specified `ola` for the Name and `eis/ola` as the JNDI name and clicked **Ok** to create a corresponding connection factory. This JNDI name is also recognized by the sample applications which we describe later.

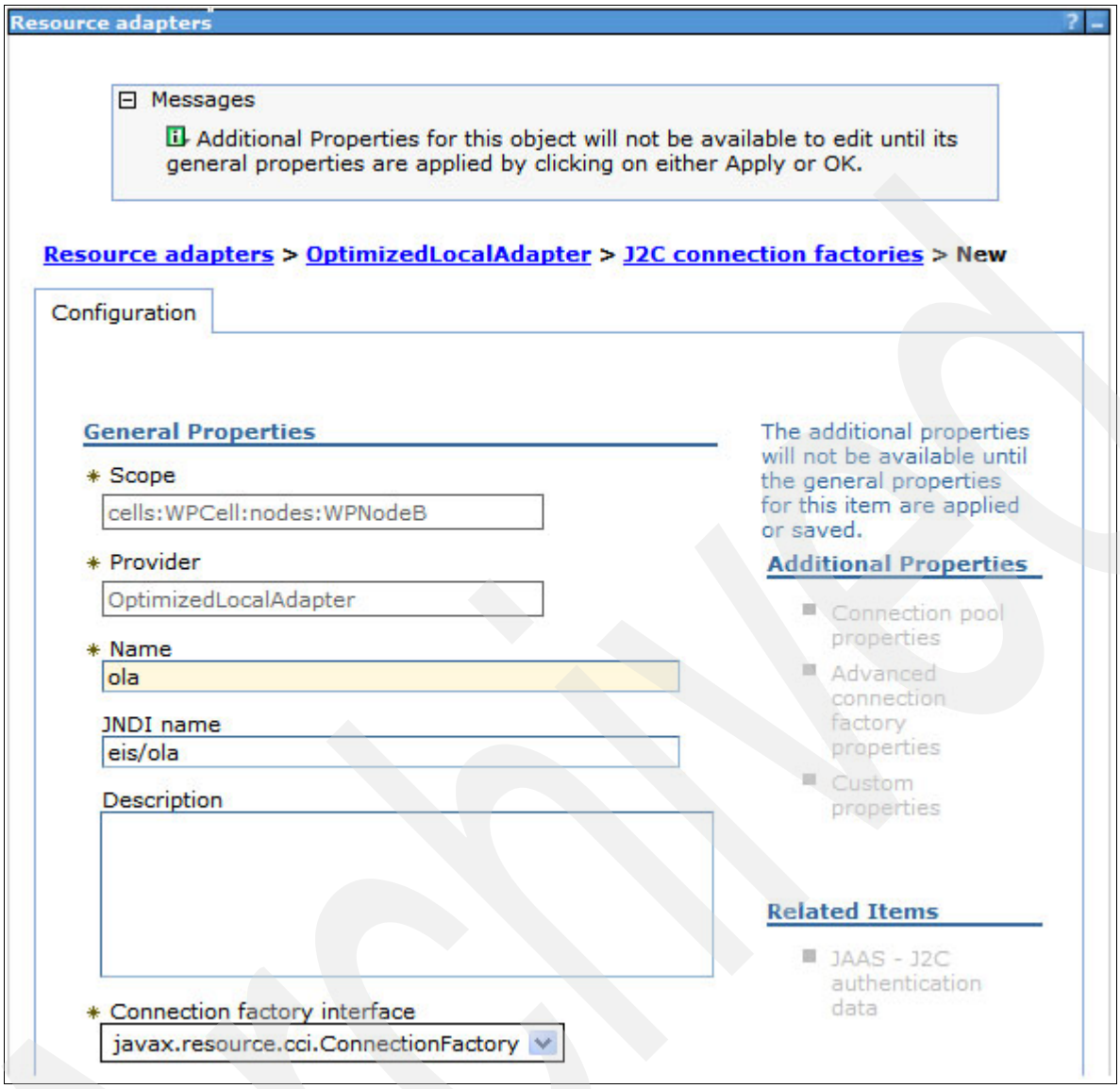



*Figure 2-7 Define a Connection factory*

- Save changes and synchronize with nodes.

After successful execution of both installation scripts, a restart of the daemon address space is requireded for activating the WebSphere Optimized Local Adapter support. In the job output of the restarted daemon, the following message should appear:

```
BBOM0001I          enable_adapter: 1.
```

## 2.4 Installation Verification Test (IVT) for WebSphere Optimized Local Adapter

With WebSphere Application Server for z/OS Fix pack 4 a set of sample applications are provided in the SMP/E HFS. These demonstrate different WebSphere Optimized Local Adapter scenarios, such as inbound calls from C/Batch or CICS to WebSphere or outbound calls from WebSphere to C/Batch or CICS.

The simplest way to validate the functionality of WebSphere Optimized Local Adapter is to use the provided sample program OLACC01. This C program runs in a batch job and calls the EJB application OLASample1 in the application server using WebSphere Optimized Local Adapter. To successfully execute the Installation Verification Test (IVT), the following tasks need to be performed:

- Allocate a partitioned data set extended (PDSE) for the WebSphere Optimized Local Adapter sample JCLs, which include the source code, for instance a data set named *hlq*.WOLA.CNTL with the following properties: RECFM=FB, DSORG=PO, LRECL=80, BLKSIZE=9040, TRKS=40.

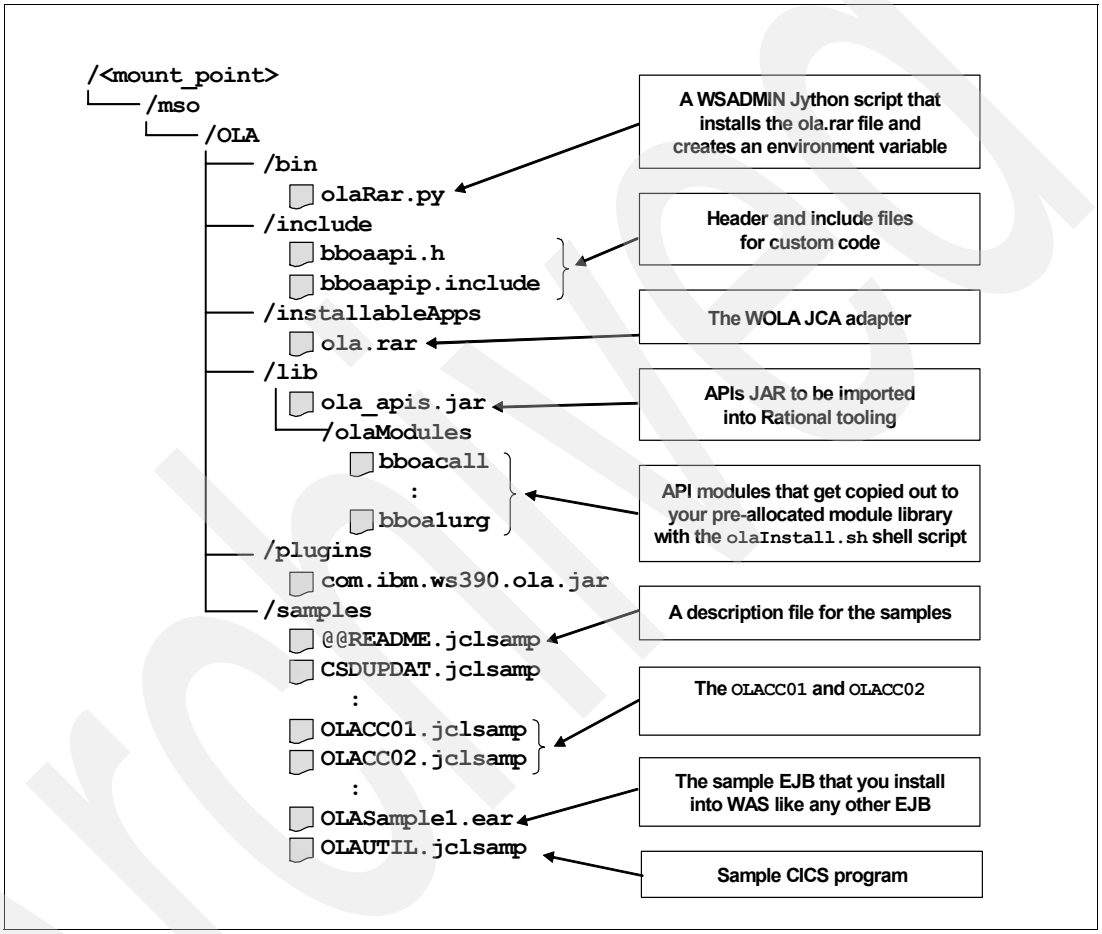



*Figure 2-8 Structure of the SMP/E HFS with regard to WebSphere Optimized Local Adapter*

- Copy all sample JCLs, which are located in the path /<WAS_SMPE>/mso/OLA/samples/* into the allocated sample CNTL data set using the TSO OGET command as shown in the following example:

  ```
  OGET '/<WAS_SMPE>/mso/OLA/samples/OLACC01.jclsample'
  'hlq.OLA.SAMPLES.CNTL(OLACC01)'
  ```

  This command needs to be issued for each sample program.

- The C header file bboaapi.h needs to be copied to the sample data set as well:

  ```
  OGET '/<WAS_SMPE>/mso/OLA/include/bboaapi.h' 'hlq.OLA.SAMPLES.CNTL(BBOAAPI)
  ```

- Allocate a second data set for the compiled load modules. This data set must be allocated as record format (RECFM) U - Undefined. For instance, the data set can be named *hlq*.WOLA.LOAD.

- Deploy sample EJB application OLASample1 on the target application server using the admin console. This sample application is located in the following USS path: `<WAS_SMPE>/mso/OLA/samples/OLASample1.ear`. You can accept the defaults for the installation. This EAR file also contains the corresponding Java™ sources, which can easily be used as a basis for developing your own WebSphere Optimized Local Adapter EJB applications.
- Adjust the OLACC01 program according to your C compiler environment, especially the outfile (*hlq*.OLA.SAMPLE.LOAD(OLACC01)) and the DD statements for the compile and binding step. The SBBOLOAD data set listed in the bind DD statements has been created for the `olaInstall.sh` shell script during the installation of WebSphere Optimized Local Adapter and provides the WebSphere Optimized Local Adapter APIs for external address spaces.
- Specify the target application server in the OLACC01 program. In this case the short name of the cell, node, and server is required.

  ```
  GETARG(daemonGroupName, 1, 8, 0x00, "WPCELL");
  GETARG(nodeName, 2, 8, ' ', "WPNODEB");
  GETARG(serverName, 3, 8, ' ', "WPS01B");
  ```

  The short names are always part of the MVS START command, for example:

  ```
  START WPACRB,JOBNAME=WPS01B,ENV=WPCELL.WPNODEB.WPS01B
  ```
- Submit the OLACC01 in order to compile and bind the C program. The job should return a RC=0.
- Adjust OLABATCH in order to execute the compiled C program OLACC01, which is located in the LOAD data set.
- By submitting the modified OLABATCH, the C program will call the sample EJB application OLASAMPLE1. Success is indicated by RC=0 along with the following message in the job output of OLABATCH:

  ```
  Invoking service "ejb/com/ibm/ola/olasample1_echoHome"
  Invoke response length matches expected: 61
  Invoke response data matches expected
  ```

  The corresponding message in the SYSPRINT of the application server servant region should be as follows:

  ```
  olasample1_echo method execute() running! Returning passed data:  ¢
   @            @M         ]@
  ```

If the WebSphere Optimized Local Adapter was not able to invoke the EJB, the OLABATCH job output will provide a return code and reason code such as: **Invoke error! rc: 8 rsn: 34 rv: 0**. A complete list of possible return codes, along with their descriptions, is provided at the Infocenter:

http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/index.jsp?topic=/com.ibm.websphere.zseries.doc/info/zseries/ae/cdat_olaapis.html

In total, the sample C program OLACC01 makes three API calls. First of all the program registers with the daemon group using the corresponding API BBOA1REG. In the second step it uses a BBOA1INV call to invoke the target EJB and passes data pointers for the request and response data to the EJB method. In this sample OLLCC01 calls the EJB olasample1_echoBean, which is part of the OLASample1 application, which we deployed in an earlier step. In the following row of the OLACC01 program the JNDI name for the target EJB is defined. In a later step this JNDI name will be passed through the BBOA1INV API.

```
#define serviceJNDIname "ejb/com/ibm/ola/olasample1_echoHome"
```

The WebSphere Optimized Local Adapter is always looking for an “execute method” in the target EJB, which receives a byte array and returns a byte array. Example 2-4 shows the execute method of olasample1_echoHome. This method just converts the byte array into a string, prints this string to the job log and returns the byte array to the C program OLACC01.

*Example 2-4 Extract of olasample1_echoHome*

```
      public byte[] execute(byte[] arg0) {
            String list = new String(arg0);
            System.out.println("olasample1_echo method execute()
running! Returning passed data: "+list);
            return arg0;
   }
```

Finally, the BBOA1URG API is called in order to unregister from the daemon group as shown in Figure 2-9. During the register and unregister call the invoke API can be called multiple times, for instance using a loop.

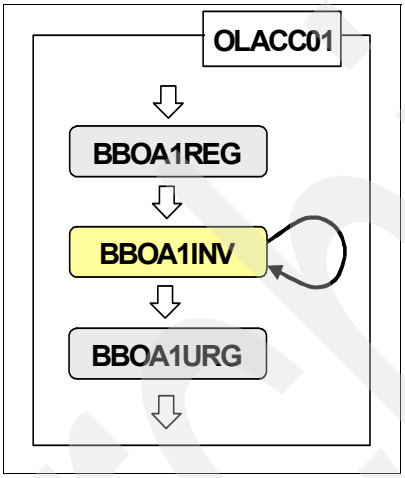



*Figure 2-9 Flow of WebSphere Optimized Local Adapter APIs in OLACC01*

More samples are provided with WebSphere. The corresponding documentation is provided in the Infocenter:
http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/index.jsp?topic=/com.ibm.websphere.zseries.doc/info/zseries/ae/cdat_olasamples.html

## 2.5 Sample application - OLACC02

In contrast to OLACC01, this sample is much more complex, because it demonstrates a combination of WebSphere Optimized Local Adapter inbound and outbound calls.

The C program OLACC02 (see Figure 2-10 on page 19) invokes a round-trip EJB and passes data using an inbound call. The data gets processed by the round-trip EJB, initiates a WebSphere Optimized Local Adapter outbound call using the J2C adapter passing data, and invokes SampleService, which is a host service provided by the C program OLAACC02. The C program passes back the received data to the round-trip EJB.

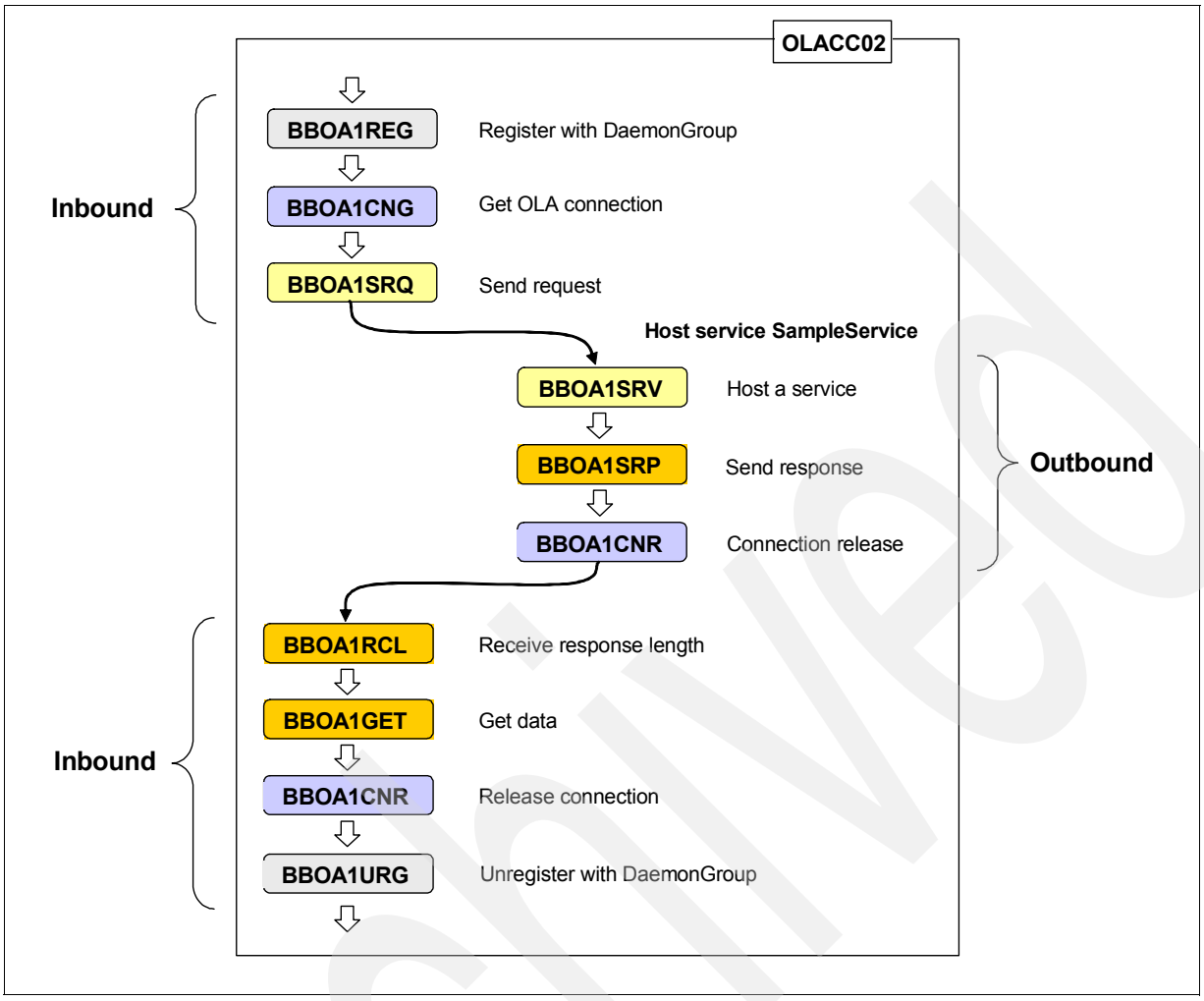



*Figure 2-10 The OLABATCH program*

Expected job output of OLABATCH:

```
Calling Send Request to ejb/com/ibm/ola/olasample1_roundtripHome
Calling BBOA1SRV to host a service for 'SampleService'
Host service length matches expected: 61
Host service data matches expected
Calling Send Response, sending 59 bytes...
RCL response length matches expected, 59
Get data matches expected
```

Application Server Servant region SYSPRINT:

```
olasample1_roundtripBean.execute() entered, received 61 bytes
olasample1_roundtripBean.execute() exiting, returning 59 bytes
```

## 2.6 Monitoring

The activity of the WebSphere Optimized Local Adapter adapters can be monitored by using MVS modify commands. For instance, all current WebSphere Optimized Local Adapter registrations regarding one application server can be displayed with the following command:

```
F WPS01B,DISPLAY,ADAPTER,REGISTRATIONS
BBOA0006I: SHOWING REGISTRATIONS FOR SERVER:
BBOA0000I: TYPE: s JOBNAME: WPS01B   NAME: *WASCTL*
BBOA0001I:  JOBNUM: 0 ACTIVE: true ACTIVE-CONNECTIONS: 0
BBOA0002I: MIN-CONN: 0 MAX-CONN: 0 STATE: 00 TRACELEVEL: 00
BBOA0023I: THIS REGISTRATION DOES NOT HAVE ANY CONNECTION HANDLES
BBOA0026I:
BBOA0003I: Name         Jobname  SWT TL Min  Max  Act  State
BBOA0004I: OLACC01      OLABATCH 000 02 0001 0005 0001 00
BBOA0004I: OLACC02      OLABATC2 000 02 0001 0005 0001 00
BBO00188I END OF OUTPUT FOR COMMAND DISPLAY,ADAPTER,REGISTRATIONS
```

All external address spaces will be listed which called the OLABATCH register API BBOA1REG, but did not call the unregister API BBOA1URG so far. There are a lot of possibilities to filter the results.

The following MVS modify command provides the status of the adapter showing the current WebSphere Optimized Local Adapter version, the maximum number of connections, and the daemon group that this server belongs to.

```
F WPS01B,DISPLAY,ADAPTER,STATUS
BBOA0005I: STATUS: VER:1 MAX-CONN:100 DAEMON-GRP-NAME:WPCELL
BBO00188I END OF OUTPUT FOR COMMAND DISPLAY,ADAPTER,STATUS
```

More information about MVS modify commands for WebSphere Optimized Local Adapter are provided in the Infocenter:

http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/index.jsp?topic=/com.ibm.websphere.zseries.doc/info/zseries/ae/tdat_mvsmodifyola.html

## 2.7 Troubleshooting

Under some circumstances it might be necessary to turn on tracing to identify a WebSphere Optimized Local Adapter-specific problem.

- Change the trace option through the administrative console:
  - Click **Troubleshooting** → **Log and Trace** → ***server_name*** → **Change Log Detail Levels**.
  - Change the trace level in the Configuration tab.
- Change the trace level in the Runtime tab.

  Figure 2-11 shows trace options and a trace level example.

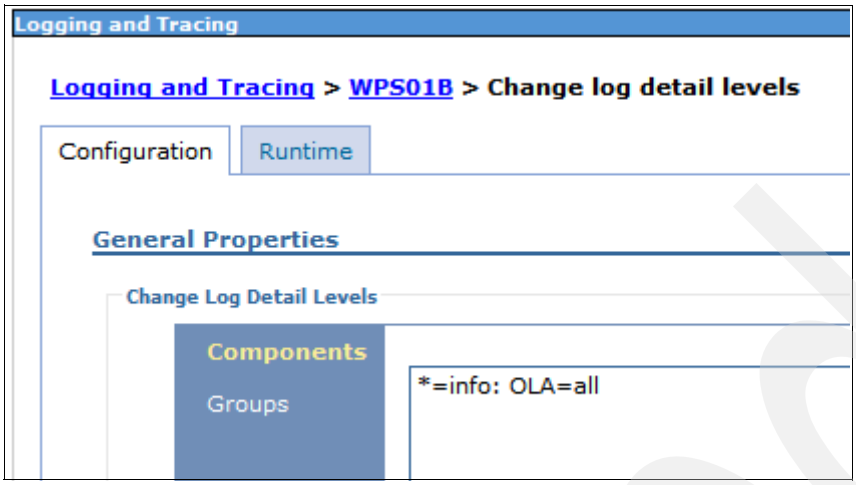



*Figure 2-11 Trace options and levels*

- Change trace options/level using the MVS modify commands:

  WebSphere Optimized Local Adapter Java trace:

  ```
  F CR_short_name,tracejava=OLA=all
  ```

  This is the corresponding modify command to the trace level change in the admin console shown in Figure 2-11. This trace will appear in the job log of the application server control region.

- Moreover, it is possible to enable a WebSphere Optimized Local Adapter native trace with the following MVS modify command:

  ```
  F CR_short_name,TRACEDETAIL=G
  ```

- Once the traces are captured, the following modify command can be used to reset to the initial trace settings:

  ```
  F CR_short_name,traceinit
  ```

  With the use of MVS modify commands the trace overhead can be reduced to a minimum.

  For more information about the available modify command options, refer to the InfoCenter at:

  http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/index.jsp?topic=/com.ibm.websphere.zseries.doc/info/zseries/ae/rxml_mvsmodify.html

**Note:** The runtime trace tab and the MVS modify command change trace options dynamically,\ whereas the configuration trace tab sets trace options permanently and requires a restart of the server for activation.

3

# WebSphere Optimized Local Adapters-enabled Trade application

In this chapter we modify the IBM benchmark application Trade6 to enable it to receive WebSphere Optimized Local Adapters inbound calls from a C program. We also create the C program OLACTRA to implement the WebSphere Optimized Local Adapters functions needed to make calls to the Trade6 application.

## 3.1 Overview

Trade6 simulates an online stock trading system which enables clients, using a browser-based Graphical User Interface, to view their portfolio, lookup stock quotes as well as buy or sell shares of stock. Trade6 is a J2EE™ application implemented through stateless Enterprise Java Beans (EJBs) as well as Java Server Pages (JSPs).

The WebSphere Optimized Local Adapters-enabled Trade6 application, along with the corresponding OLACTRA C program is available for download. See Appendix A, “Additional material” on page 103.

## 3.2 Prerequisites

The WebSphere Optimized Local Adapters must first be installed and configured following the steps as described in Chapter 2, “WebSphere Optimized Local Adapters - Installation and configuration” on page 5.

The IBM Benchmark application Trade6 is the basis for our WebSphere Optimized Local Adapters sample implementation. The unmodified Trade6 can be downloaded from the Internet using the following Link:

https://www14.software.ibm.com/webapp/iwm/web/preLogin.do?source=trade6

In order to modify the existing Trade6 application, an application development tool is required; such as Rational Application Developer (RAD) or Rational Developer for System z (RDz). For the purposes of this example we use the RAD Tooling.

## 3.3 Trade6 WebSphere Optimized Local Adapters modifications

The original Trade6 application is designed to be accessed by clients using a standard browser interface. This interface enables clients to perform all of the actions required to maintain their simulated accounts (see client browser interface Figure 3-1 on page 25). In our example we will modify the original Trade6 application to include a WebSphere Optimized Local Adapters inbound call interface. We will then perform client functions from a C program.

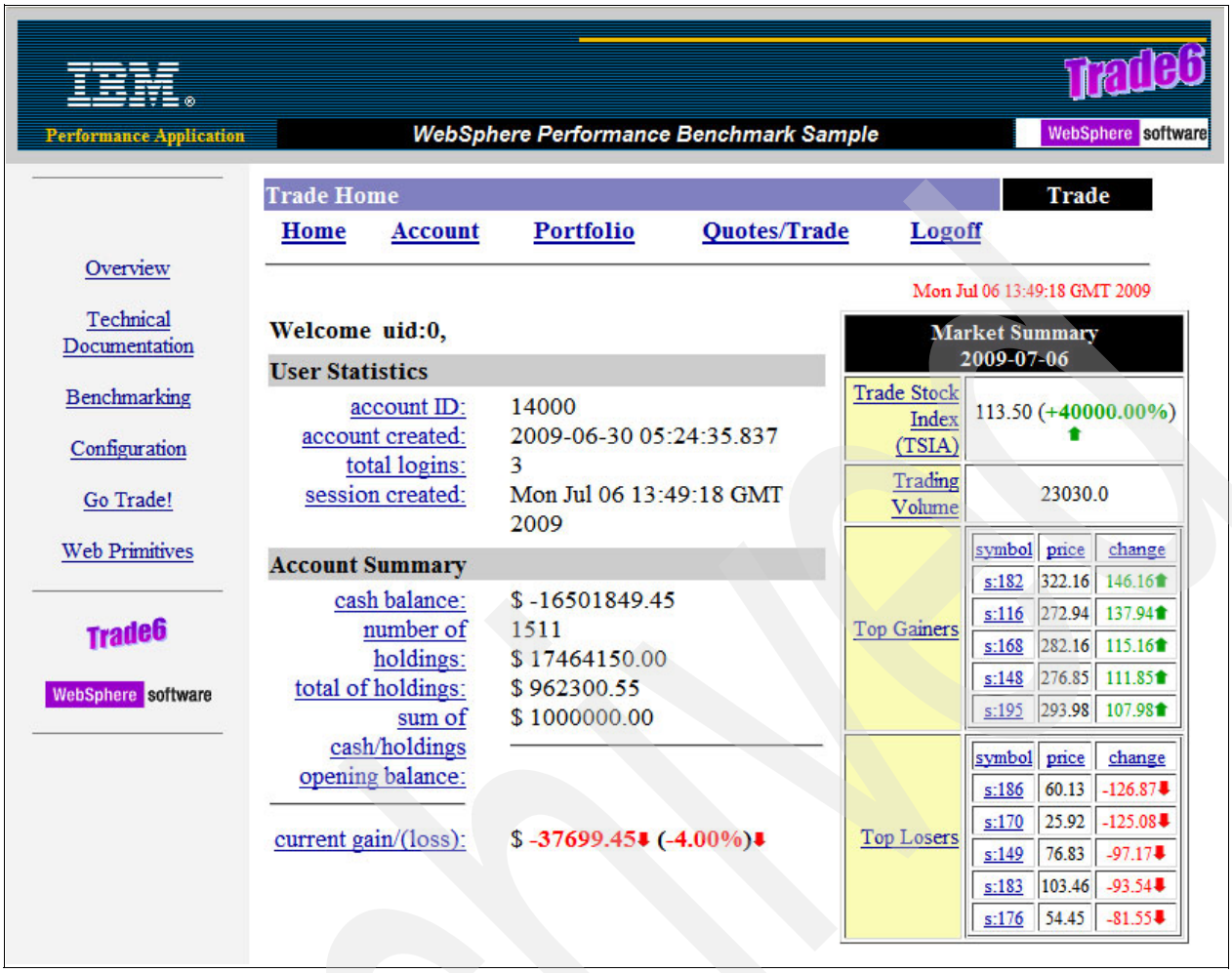



*Figure 3-1 Trade6 application - home panel*

## 3.4 Creating the WebSphere Optimized Local Adapters-enabled Trade6 sample application

The original Trade6 application must first be imported into the Rational Application Developer (RAD) tool. The Trade6 application is available from:

`https://www14.software.ibm.com/webapp/iwm/web/preLogin.do?source=trade6`

The Trade6 application is also available as trade_original.ear with the additional materials supplied with this Redpaper (see Appendix A, “Additional material” on page 103).

### 3.4.1 Add the ola_apis.jar to the Build Path

- First download the `ola_apis.jar` file from the host to the local workstation, for instance with FTP. This file is located in the following path of the SMP/E HFS:

  `<WAS_SMPE>/mso/OLA/lib/ola_apis.jar`

- The `ola_apis.jar` file, which contains the WebSphere Optimized Local Adapters APIs, is added to the Build Path by right-clicking the tradeEJB tab and then selecting **Build Path → Configure Build Path**.
- Select the Libraries tab and click **Add External JARs**.

The location of the `ola_apis.jar` file now needs to be specified:

Click **Open** to add it to the Build Path.

An additional library for the server runtime must also be added.

- Click **Add Library** and choose **Server Runtime**. Depending on your target server, select either the WebSphere Application Server V6.0 stub or V7.0 stub and then click **Finish**.
- Select the Order and Export tab and enable the server runtime library that was just added.
- Click **OK**.

## 3.4.2 Create the WolaEJB Session Bean

We now create an EJB that implements the WebSphere Optimized Local Adapters interfaces to enable the calling of the Trade6 application from our external C program.

- Right-click **tradeEJB** and select **New → Enterprise Bean (1.x-2.x)**.

  The Create an Enterprise Bean panel as shown in Figure 3-2 is displayed.

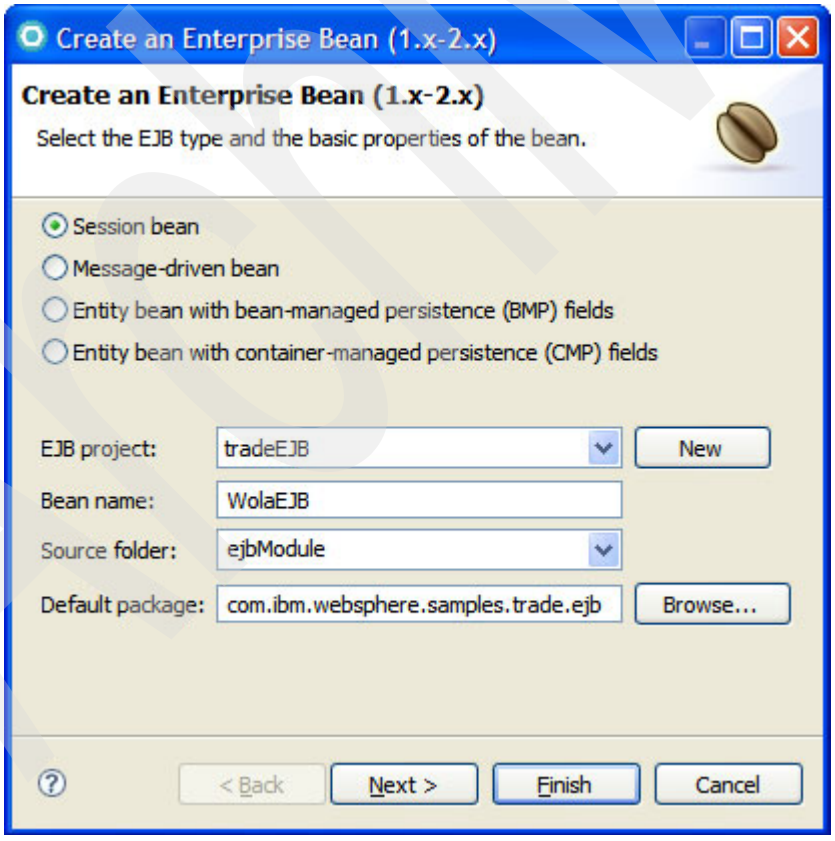



*Figure 3-2 Create a Session Bean*

- Select the type of Enterprise Bean as **Session Bean**.
- In this sample we named our WebSphere Optimized Local Adapters Bean **WolaEJB**.

- We added the WolaEJB bean to the same default package as the existing TradeEJB (com.ibm.websphere.samples.trade.ejb).
- Click **Next** to continue.

  The Enterprise Bean Details panel appears, as shown in Figure 3-3.

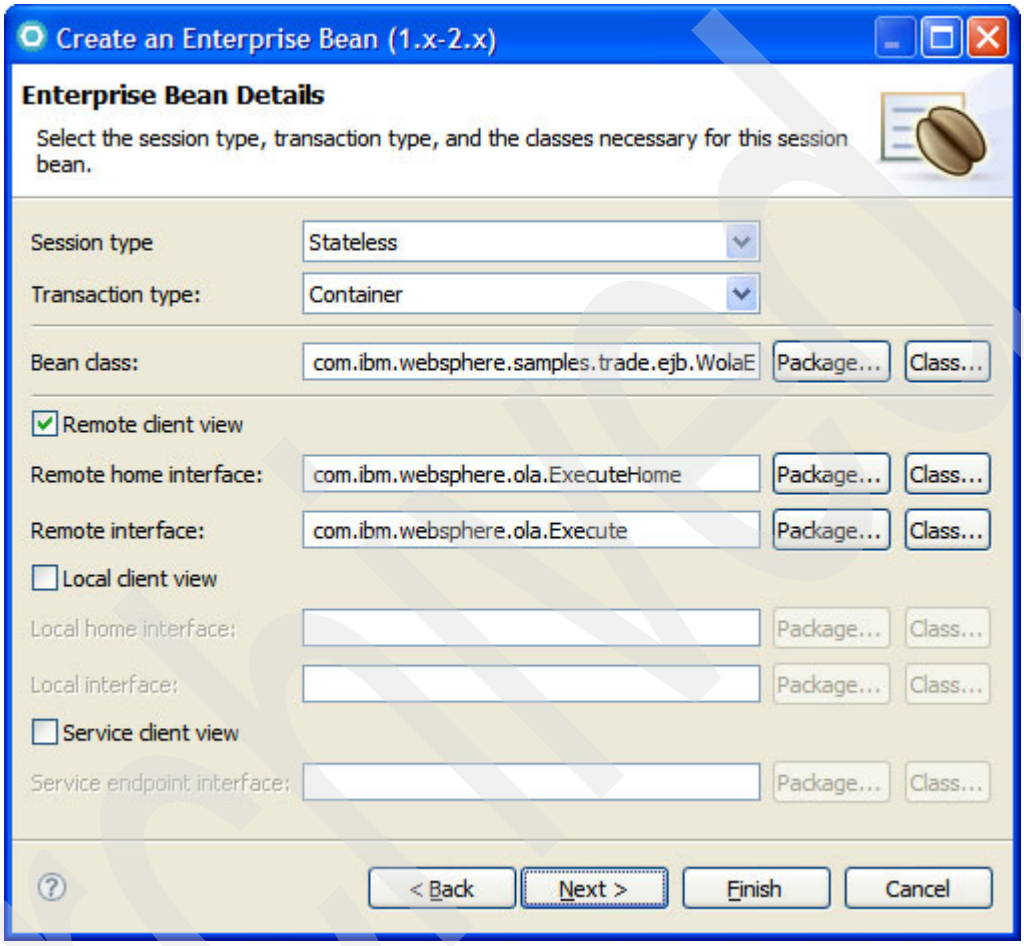



*Figure 3-3* Enterprise Bean Details panel

- Specify the Remote home interface and the Remote interface as illustrated in Figure 3-3.

  Remote Home interface: `com.ibm.websphere.ola.ExecuteHome`

  Remote interface: `com.ibm.websphere.ola.Execute`
- Click **Finish** to generate the new Session Bean WolaEJB.

The Session Bean WolaEJB, including the WolaEJBBean, Execute and ExecuteHome should now be added to the tradeEJB project as shown in Figure 3-4 on page 28.

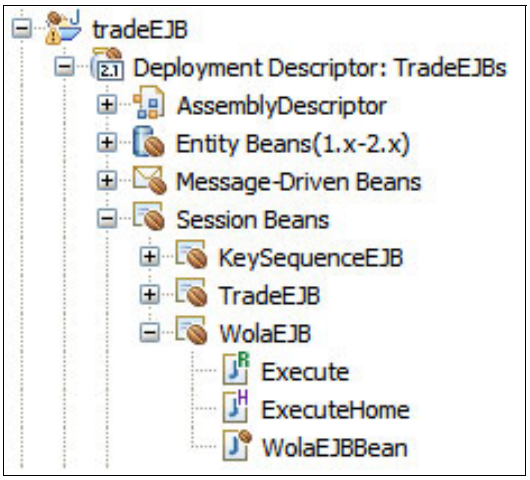



*Figure 3-4 Hierarchy of the tradeEJB project*

## Coding the WolaEJB bean

In this section the new WolaEJB bean is developed in a step-by-step approach. For completeness we provide the final version of our modified for WebSphere Optimized Local Adapters Trade6 application as a RAD 7.5 project interchange file (`WolaEnabledTrade6_RAD75ProjectInterchange.zip`) as well as the corresponding EAR file (`trade_WolaEnabled.ear`) as Additional Material for this Redpaper.

The following imports must first be defined and then the TradeHome object of the WolaEJB class must be instantiated and initialized; Example 3-1.

*Example 3-1 Create the TradeHome object*

```
package com.ibm.websphere.samples.trade.ejb;

import java.io.UnsupportedEncodingException;

import javax.naming.InitialContext;
import javax.naming.NamingException;

import com.ibm.websphere.samples.trade.OrderDataBean;
import com.ibm.websphere.samples.trade.ejb.Trade;
import com.ibm.websphere.samples.trade.ejb.TradeHome;

/**
 * Bean implementation class for Enterprise Bean: WolaEJB
 */
public class WolaEJBBean implements javax.ejb.SessionBean {

      static final long serialVersionUID = 3206093459760846163L;

      private TradeHome tradeHome = null;
```

```
  private javax.ejb.SessionContext mySessionCtx;
```

Next the generated ejbCreate method must be specified. A JNDI lookup for TradeEJB is required because the WolaEJB bean invokes the *buy()* method within TradeEJB for the execution of a buy order.

*Example 3-2 TradeEJB JNDI lookup*

```
public void ejbCreate() throws javax.ejb.CreateException {
      InitialContext ic;
      try {
            ic = new InitialContext();
            tradeHome = (TradeHome)
            ic.lookup("java:comp/env/ejb/Trade"); }
      catch (NamingException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
      }
}
```

In order for the WebSphere Optimized Local Adapter to access the WolaEJB, an *execute()* method is required. The *execute()* method is invoked by the WebSphere Optimized Local Adapter when the WolaEJB is called by the OLACTRA C program using the corresponding JNDI name. The *execute()* method expects to receive a byte array as shown in Example 3-3.

*Example 3-3 WolaEJB execute method*

```
      public byte[] execute(byte[] arg0) {

  }
```

It is important to understand that the request data from the external address space will be passed to this execute method as a byte array and the response data will be delivered back to the external address space as a byte array. All request and response data needs to be parsed to one-byte strings. For parsing you can either work with a fixed length format for all passed variables or a separator such as a semicolon. In this case we decided to use a fixed length format for simplicity reasons.

Within the *execute()* method the byte array received is converted into a unicode string. Next three fixed length substrings are constructed which contain the user ID, the stock identifier (symbol) and the quantity. These three parameters, passed to the WolaEJB from the calling OLACTRA program, are printed out to the job log of the servant region. See Example 3-4.

*Example 3-4 execute method - Converting the byte array*

```
System.out.println("WolaEJB Input");


String olaByteString_unicode = new String(arg0, "Cp1140");
```

```
System.out.println("Trade olaByteString:" + olaByteString_unicode);

if (olaByteString_unicode.length() > 25) {
      // Cut the String to the Attributes
      String userID = olaByteString_unicode.substring(0, 8).trim();
      String symbol = olaByteString_unicode.substring(8, 13).trim();
      double quantity = Double.parseDouble(olaByteString_unicode.
            substring(13, 24));
      int orderProcessingMode = 0;

      System.out.println(" ");
      System.out.println("Trade Application - WolaEJB");
      System.out.println(" ");
      System.out.println("Trade SYSOUT userID:    " + userID );
      System.out.println("Trade SYSOUT stoack:    " + symbol );
      System.out.println("Trade SYSOUT quantity: " + quantity );
      System.out.println(" ");
```

In the following steps the *buy* method within the Trade6 application is called with the parameters `userID`, `symbol`, and `quantity`. The *buy* method executes the buy stock order and the corresponding result values, total price, order fee and order status are printed to the servant region job log. See Example 3-5.

*Example 3-5 execute method - order execution*

```
      // Execute Order
      OrderDataBean olaOrder;
      Trade trade = getTrade();
      olaOrder = trade.buy(userID, symbol, quantity,
      orderProcessingMode);

      // Get result values from order
      String orderID = new String (olaOrder.
            getOrderID().toString());
      String orderConfirmedSymbol = new String(olaOrder.
            getSymbol());
      String orderConfirmedQuantity = new String(Double.
            toString(olaOrder.getQuantity()).toString());
      String orderTotalPrice = new String(olaOrder.
            getPrice().toString());
      String orderOrderFee = new String(olaOrder.
            getOrderFee().toString());
      String orderStatus = new String (olaOrder.
            getOrderStatus().toString());

      // Create a confirmation message String olaOrderConfirmation;
```

```
        System.out.println("TradeEJB Confirmation");
        System.out.println("Trade SYSOUT orderID:     " + orderID);
        System.out.println("Trade SYSOUT stock:       "
              + orderConfirmedSymbol);
        System.out.println("Trade SYSOUT quantity:    "
        + orderConfirmedQuantity);
        System.out.println("Trade SYSOUT stock price: "
              + orderTotalPrice);
        System.out.println("Trade SYSOUT orderfee:    "
              + orderOrderFee);
       System.out.println("Trade SYSOUT orderstatus: "
              + orderStatus);
```

The return variables associated with the buy stock order are placed into the byte array olaByteReturn, which is returned to the OLACTRA program as shown in Example 3-6. As previously mentioned, we defined a fixed length for each variable returned to the OLACTRA program. Therefore, if necessary the result variables are padded with blanks. This padding is performed by the *stringPadding* method, which is shown in Example 3-7 on page 32. Before the *return* statement is executed, the byte array is converted to EBCDIC.

*Example 3-6 Construct the byte array*

```
        // Construct byte[] for Return value:
        String olaReturn = "";
        olaReturn = olaReturn + stringPadding(orderID,8);
        olaReturn = olaReturn + stringPadding(orderTotalPrice,9);
        olaReturn = olaReturn + stringPadding(orderOrderFee,9);
        olaReturn = olaReturn + stringPadding(orderStatus,12);
        olaReturn = olaReturn + stringPadding(
              orderConfirmedQuantity,11);
        olaReturn = olaReturn + stringPadding(orderConfirmedSymbol,5);

        byte[] olaByteReturn = olaReturn.getBytes("Cp1047");
        return olaByteReturn;
}

else {
        System.out.println("Warning: WOLA input byteString must have
              a size greater than 25 characters.");
        String errorEncoded2 = new String("The order could not be
              processed.");
        byte[] errorreturn2 =  errorEncoded2.getBytes("Cp1047");
        return errorreturn2;
}
```

Example 3-7 shows the *stringPadding* method, which pads the fixed length fields with blanks as needed.

*Example 3-7 String padding*

```
public String stringPadding(String input,int maxchars){
      if(input.length() > maxchars)
      { input = input.substring(0, maxchars).trim(); }

      String output = input;
      int addchars = maxchars - input.length();

      for(int i=0; i<addchars; i++)
      { output = output + " "; }

      return output;
}
```

The *getTrade* method shown in Example 3-8 creates an instance of the TradeEJB bean for the execution of the stock buy order.

*Example 3-8 getTrade method*

```
      private Trade getTrade() throws Exception {
            Trade tradeBean;

            tradeBean = tradeHome.create();
            return tradeBean;
   }
```

## Prepare for deployment

To generate all necessary files for deployment, right-click the **tradeEJB** project and choose **Java EE** → **Prepare for deployment**.

## Add EJB references and JNDI names to the deployment descriptor

The EJB references can be modified in the deployment descriptor of the tradeEJB project as follows:

- Navigate to the reference tab and choose **WolaEJB**.
- Click **Add** to add a new reference.
- Choose the preselected **EJB reference**.
- Choose **TradeEJBs** → **TradeEJB** and change the Name **ejb/TradeEJB** to **ejb/Trade**.
- Make sure the reference type **Remote** is selected.
- Click **Finish**.

The ibm-ejb-jar-bnd.xml file must be edited to change the default JNDI name ejb/com/ibm/websphere/ola/ExecuteHome to ejb/WolaEJB. This JNDI name will be utilized by the OLACTRA program during the BBO1INV call.

- Navigate to the ibm-ejb-jar-bnd.xml file using the following path: **tradeEJB** → **ejbModule** → **META-INF** → **ibm-ejb-jar-bnd.xmi**.
- Search for the string ejb/com/ibm/websphere/ola/ExecuteHome and substitute it with ejb/WolaEJB as shown in Example 3-9.

*Example 3-9 Specify jndiName*

```
<ejbBindings xmi:id="EnterpriseBeanBinding_1247083205500" jndiName="ejb/WolaEJB">
    <enterpriseBean xmi:type="ejb:Session"
href="META-INF/ejb-jar.xml#WolaEJB"></enterpriseBean>
    <ejbRefBindings xmi:id="EjbRefBinding_1247087843593" jndiName="ejb/TradeEJB">
      <bindingEjbRef href="META-INF/ejb-jar.xml#EjbRef_1247087843593"/>
    </ejbRefBindings>
  </ejbBindings>
```

### Deploying the trade.ear

The simplest means to deploy the Trade6 application is to execute the trade.jacl script provided with the Trade6 installation package. It performs the WebSphere configuration setting needed for Trade6 including data sources and JMS definitions. If you wish to execute the trade.jacl script as is, the olainstall.sh shell script must first be run against the Deployment Manager as described in Chapter 2, "WebSphere Optimized Local Adapters - Installation and configuration" on page 5. Otherwise the deployment using the trade.jacl script will fail.

If you choose to deploy the trade.ear manually, the complete path name to the ola_apis.jar must be added to the classpath during EJB deploy.

## 3.4.3 Modifying the OLACC01 sample application

For this sample we created the C program OLACTRA, which is based on the OLACC01 sample program that is shipped with the OLA product. The OLACTRA program is provided as Additional Materials for this Redpaper. See Appendix A, "Additional material" on page 103.

Compared to OLACC01, we have changed the JNDI name to ejb/WolaEJB as shown in Example 3-10.

*Example 3-10 Edit JNDI name*

```
   #define serviceJNDIname "ejb/WolaEJB"
```

Moreover we specified the target server in our WebSphere environment as shown in Example 3-11.

*Example 3-11 Specify target server*

```
GETARG(daemonGroupName, 1, 8, 0x00, "WPCELL");
GETARG(nodeName, 2, 8, ' ', "WPNODEB");
GETARG(serverName, 3, 8, ' ', "WPSO1B");
```

Finally we modified the data2send variable, which contains the request data userID, stock name, and quantity.

### 3.4.4 Testing the WebSphere Optimized Local Adapter-enabled Trade6 application

To verify the WebSphere Optimized Local Adapter-enabled Trade6 application, the OLACTRA program must first be compiled and linked by submitting the corresponding job. Ensure that the target application server is up and running and the Trade6 application is started.

The compiled C program can then be executed with the supplied OLABATCH job. A successful execution is indicated by RC=0.

In the job output of the target application server servant region the messages shown in Example 3-12 should appear. The output contains the data received by WolaEJB from the OLACTRA program as well as the TradeEJB order confirmation, which contains the orderID, stock name, quantity, stock price, order fee, and the order status. This data is returned to the OLATTRA program as a byte array.

*Example 3-12 Servant region Job log*

```
Trade Application - WolaEJB

Trade SYSOUT userID:   uid:0
Trade SYSOUT stoack:   s:134
Trade SYSOUT quantity: 10000.0

TradeEJB Confirmation
Trade SYSOUT orderID:     100929
Trade SYSOUT stock:       s:134
Trade SYSOUT quantity:    10000.0
Trade SYSOUT stock price: 8.50
Trade SYSOUT orderfee:    24.95
Trade SYSOUT orderstatus: closed
```

In the job output of the OLACTRA program the target application server, as well as the invoking EJB, are printed displayed. The userID, stock name and quantity, sent to the WolaEJB, are also listed. If the order was placed successfully, the OLACTRA application program prints out a notification along with the response data from the WolaEJB. Ensure that the orderID displayed by the OLACTRA program, as shown in Example 3-13, is the same orderID displayed in the servant region job log shown in Example 3-12.

*Example 3-13 Job output from the OLACTRA program*

```
Registrate with DaemonGroup:WPCELL
                Node       :WPNODEB WPSO1B  OLACCO1
                Server     :WPSO1B  OLACCO1

Invoking enterprise bean: "ejb/WolaEJB"

Placing Trade order:
        userID:   uid:0
        stock:    s:134
        quantity: 10000.0

Trade order has been placed sucessfully.

               orderID price    orderFee status      quantity   stock
```

```
Response Data: 100929  8.50     24.95     closed      10000.0     s:134
```

As a further verification, the results of the Trade6 transaction can be viewed using the Trade6 browser interface.

- In a Web browser set the target URL to the address of the application server where the Trade application is deployed along with the default context root of `/trade/`.
- Select **Go Trade** in the navigation bar.
- For the login take the provided default user name uid:0 and the corresponding password.
- Click **Login**.

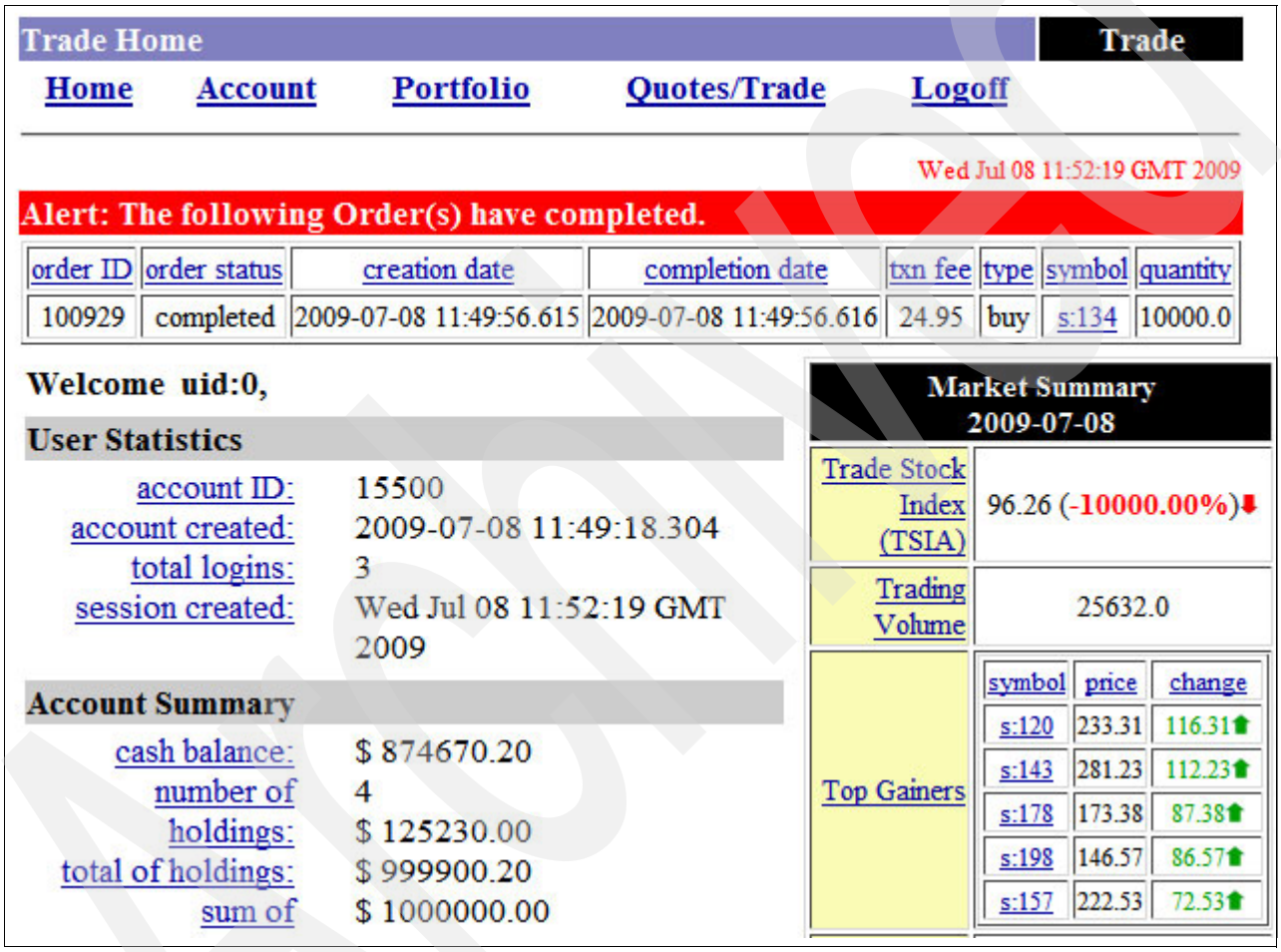



*Figure 3-5 Trade initial login screen for user uid:0*

An `Alert: The following Order(s) have completed` message should appear indicating that a new order has been placed. Verify that the orderID, along with the symbol name and quantity, are equal to the values displayed in the job output logs. By clicking the **Portfolio** tab a new entry should appear in the Portfolio as shown in Figure 3-6 on page 36.

Trade Portfolio | Trade

Home | Account | Portfolio | Quotes/Trade | Logoff

Wed Jul 08 11:54:14 GMT 2009

**Portfolio** — **Number of Holdings: 4**

| holding ID | purchase date | symbol | quantity | purchase price | current price | purchase basis | market value | gain/(loss) | trade |
|---|---|---|---|---|---|---|---|---|---|
| 94441 | 2009-07-08 11:49:18.312 | s:676 | 188.0 | 81.00 | 69.63 | 15228.00 | 13090.44 | -2137.56⬇ | sell |
| 94442 | 2009-07-08 11:49:18.327 | s:29 | 85.0 | 136.00 | 42.23 | 11560.00 | 3589.55 | -7970.45⬇ | sell |
| 94443 | 2009-07-08 11:49:18.34 | s:82 | 143.0 | 94.00 | 158.23 | 13442.00 | 22626.89 | 9184.89⬆ | sell |
| 96930 | 2009-07-08 11:53:59.775 | s:134 | 10000.0 | 8.42 | 8.42 | 84200.00 | 84200.00 | 0.00⬆ | sell |
| | | | | | Total | $ 124430.00 | $ 123506.88 | $ -923.12⬇ (-1.00%)⬇ | |

*Figure 3-6 A new holding has been added to the portfolio*

4

# OLA - CICS to EJB in WebSphere

This chapter demonstrates how to use WebSphere for z/OS Optimized Local Adapters to invoke an EJB from a COBOL program in CICS.

It covers the following topics:

- Describes a sample COBOL application that currently runs within CICS.
- Describes how to use Rational Application Developer to develop an EJB to perform the same business logic functions as the existing CICS COBOL sample program.
- How to modify the main COBOL program to call an EJB using the WebSphere Optimized Local Adapter APIs.
- Testing of the sample programs.

## 4.1 Reasons for CICS calling EJBs

Perhaps the first question that needs to be asked is: why would a program in CICS want to call an EJB?

### 4.1.1 Integration

Organizations today have diversified runtime environments to run their business applications. Many would have COBOL-based applications running in CICS on z/OS as well as applications written in Java running in WebSphere Application Server.

Where once applications were largely independent, organizations are now looking to integrate applications into business processes. This leads to applications running in different runtimes needing to be able to interact with each other.

The organization may then require that an application running in CICS be able to invoke business functionality that has been implemented in EJBs running in WebSphere Application Server.

One approach would be to rewrite the business logic in the EJB in COBOL and run it locally in the CICS region. This is clearly a less than desirable solution because it would be costly in time and effort, plus it also means that the organization is now supporting two sets of application code that perform the same function.

This is where WebSphere for z/OS Optimized Local Adapters comes into play, as it provides a straightforward way to achieve integration between CICS and WebSphere Application Server. Since this feature is included with WebSphere Application Server for z/OS V7, there is little cost beyond the time involved in coding the COBOL program that calls the EJB.

The above is predicated on the assumption that WebSphere Application Server is running on z/OS, because the WebSphere for z/OS Optimized Local Adapters functions can only be used between CICS and WebSphere Application Server when both are running in the same z/OS LPAR.

A Java program running in CICS can use the standard approach for calling an EJB in WebSphere where the Java program first performs an initial context look-up, and so on. This approach allows for the fact that the WebSphere Application Server being accessed could be located anywhere—it does not have to be located within the same z/OS LPAR. No samples demonstrating this technique are supplied with the CICS product. Additionally, the announcement letter for CICS Transaction Server V4.1 states that a future release of CICS would remove session bean support. When that support is removed, the aforementioned Java technique for calling remote EJBs will no longer work.

WebSphere for z/OS Optimized Local Adapters is focused on enhancing the value proposition of running WebSphere Application Server V7 on z/OS. Organizations already running WebSphere Application Server on distributed and CICS on z/OS may consider the benefits of simpler integration and higher performance that WebSphere for z/OS Optimized Local Adapters provides as part of a business case for moving some of their WebSphere Application Server environments from distributed to z/OS.

### 4.1.2 Migration

Another reason organizations may make use of WebSphere for z/OS Optimized Local Adapters is to move some business functionality currently located in CICS to the WebSphere Application Server.

An organization may have some business logic currently implemented in programs running in CICS. The organization may be developing new applications to run in WebSphere Application Server that require some part of business logic currently running in CICS. The organization may decide to redevelop the business logic currently running in CICS to run in EJBs. But they still have applications in CICS requiring access to the business logic in the new EJBs.

WebSphere for z/OS Optimized Local Adapters again suits this type of scenario. The programs in CICS that used to call other programs in CICS that implemented the business logic, can be relatively easily modified to call EJBs that now implement the business logic.

It is this type of scenario that we will demonstrate in this chapter, because it also effectively covers the first reason described above.

## 4.2 The scenario

First we explain the sample application running in CICS and then we show how we moved the business logic from this CICS application in an EJB, and finally how we modified the COBOL program running in CICS to call the EJB.

### 4.2.1 The existing CICS application

Our sample CICS application consists of the following components:

- EPSL01 - COBOL program that handles the presentation logic
- EPSL02 - COBOL program that performs the business logic, in this case some simple date calculations
- EPSM01 - BMS Map
- EPSL - CICS Transaction definition, which calls the EPSL01 program

In CICS the EPSL transaction calls the EPSL01 program. This displays the BMS map from EPSM01. The user then enters a date, and EPSL01 passes this date to the EPSL02 program via a COMMAREA. EPSL02 calculates the number of days from that date to the current date and what day of the week that date was and returns the results to EPSL01, which then displays them to the user.

### 4.2.2 How to invoke

To run the existing CICS sample application, we logged onto a CICS region and then typed `EPSL`, which presented the display shown in Figure 4-1 on page 40.

```
                EPS TOOLS BIRTHDAY/RETIREMENT EXAMPLE


     PLEASE ENTER BIRTHDATE: YYYYMMDD

              1    ENTER A 1 TO SEE YOUR BIRTHDAY
                   ENTER A 2 TO CALCULATE YOUR RETIREMENT









 F3/F12 TO TERMINATE, ENTER TO PROCESS, CLEAR TO START OVER
```

*Figure 4-1 Initial display of existing CICS transaction*

A date is entered in place of YYYYMMDD on the panel. Entering a 1 results in the number of days since that being displayed. Entering a 2 displays how many days until you retire assuming that will be when you turn 65. Also returned for both options is the date in formatted form, as shown in Figure 4-2.

```
                EPS TOOLS BIRTHDAY/RETIREMENT EXAMPLE


     PLEASE ENTER BIRTHDATE: 19940621

              1    ENTER A 1 TO SEE YOUR BIRTHDAY
                   ENTER A 2 TO CALCULATE YOUR RETIREMENT

     HERE IS YOUR BIRTHDATE AND NO DAYS FROM CURR

  YOUR BIRTHDATE AND DAY:      Tuesday 21 June 1994

  HOW LONG AGO WAS THIS?:      000005471 DAYS




 F3/F12 TO TERMINATE, ENTER TO PROCESS, CLEAR TO START OVER
```

*Figure 4-2 Result of entering a date and selecting option 1*

Entering a 2 will cause the EPSL01 program to call the EPSL03 COBOL program; we did not change this part.

## 4.3 Building the EJB

We used IBM Rational Application Developer for WebSphere to develop the EJB. This section describes the steps involved.

### 4.3.1 Create a project

In RAD select **File** → **New** → **Enterprise Application Project** as shown in Figure 4-3.

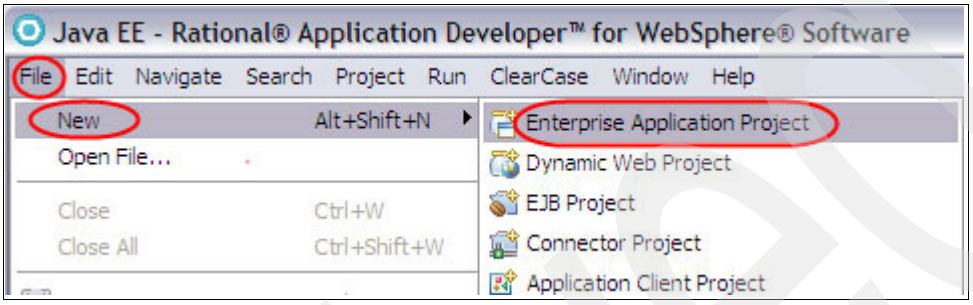



*Figure 4-3 Create new project*

On the next panel, shown in Figure 4-4 on page 42, we set the project name to WOLA-CicsToWasSand. We also set the target runtime to be WebSphere Application Server V6.1 so that RAD would generate a type 2 EJB.

We then clicked **Next**.

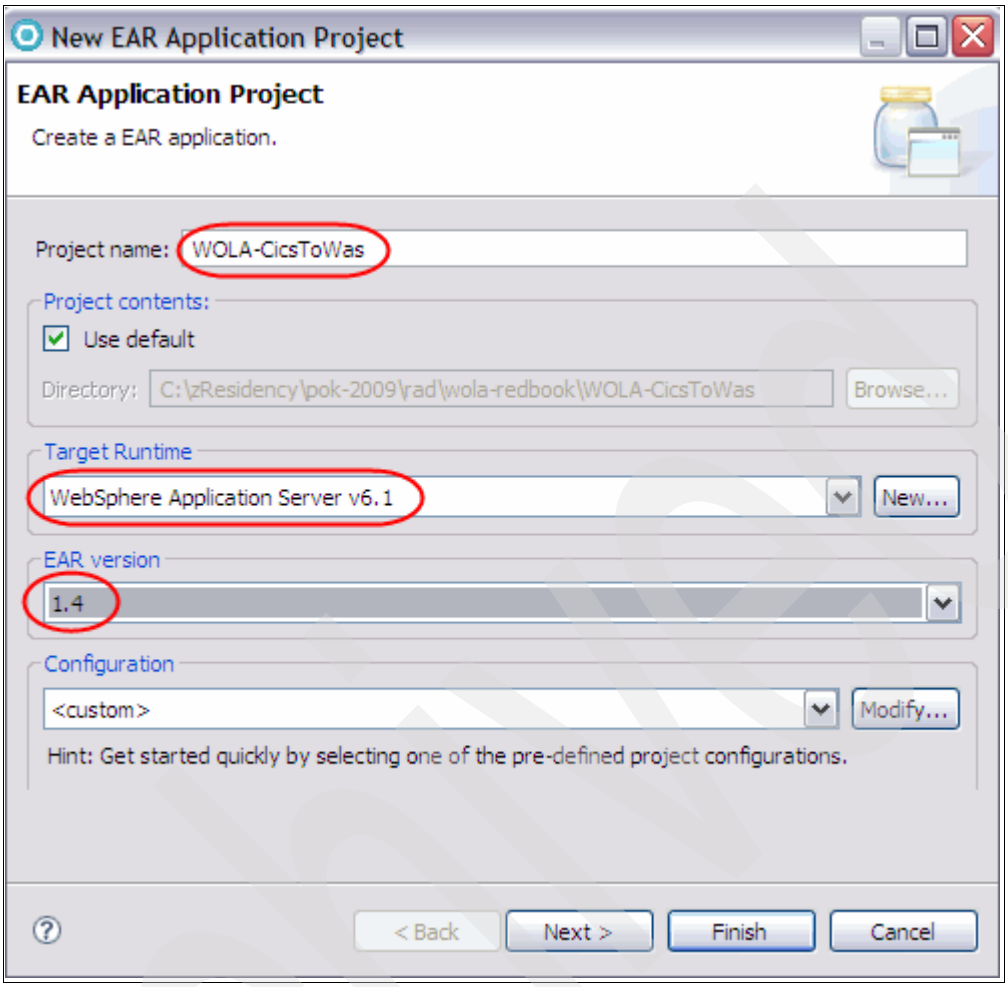



*Figure 4-4 Name and select project target*

On the next panel, Figure 4-5, click **New Module**, which will display another panel used to select which modules will be used in the application. Select only the EJB Module and click **Finish**.

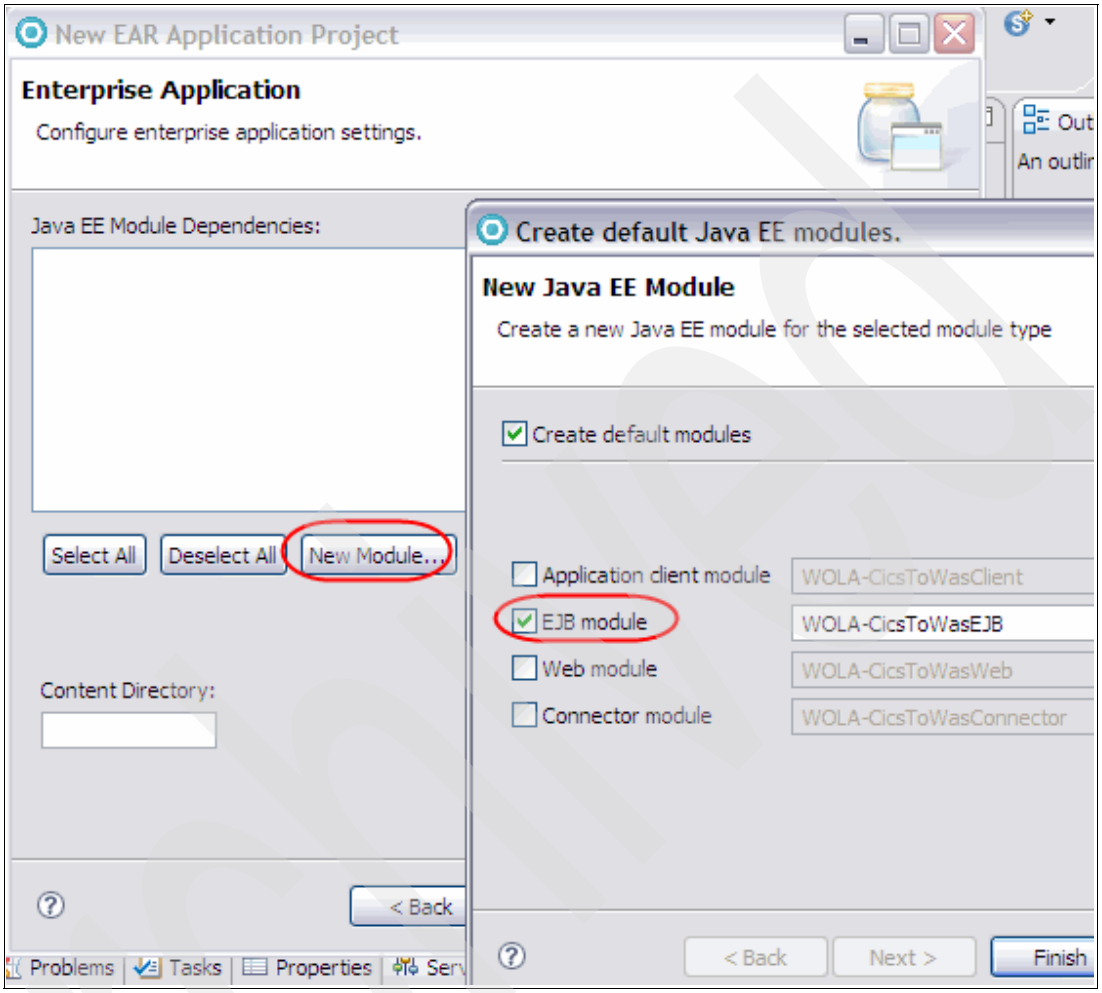



*Figure 4-5 Specify EJB module to be created*

RAD will then display the panel shown in Figure 4-6. Click **Finish**.

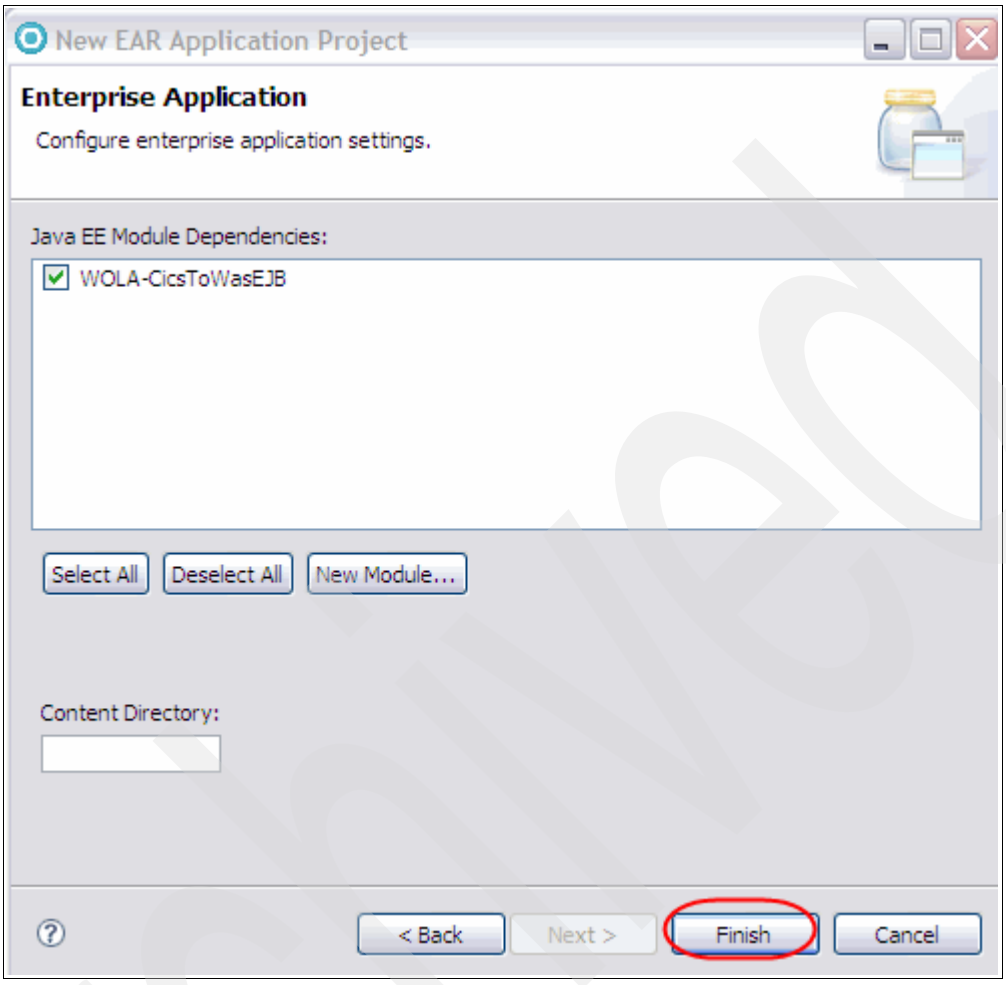



*Figure 4-6 Project with EJB module*

RAD will then create the project and the display will then be similar to that shown in Figure 4-7.

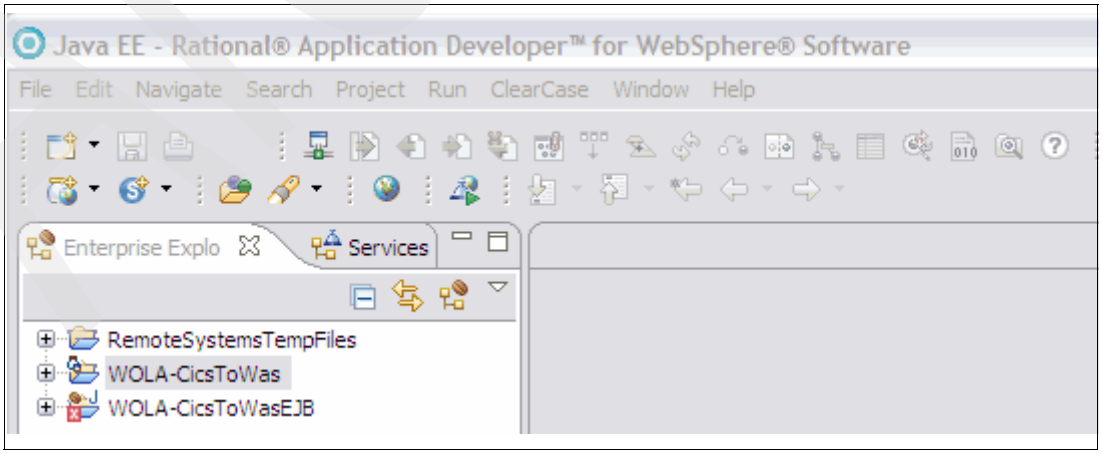



*Figure 4-7 Project generated*

The small red box with a cross in it means that at this stage there is no EJB defined in the project, which is the next step.

## 4.3.2 Add WebSphere Optimized Local Adapter jar file to build path

The EJB to be called by a program in CICS needs to have specific class interface names set on the remote interface. Adding the WebSphere Optimized Local Adapter-supplied jar file to the build path simplifies the process of setting the correct remote interface values and the creation of the EJB.

It is possible to set up the EJB without modifying the build path, and we show this approach in this chapter as well. It is recommended, however, to use the approach of adding the WebSphere Optimized Local Adapter jar file to the build path.

### Download the WebSphere Optimized Local Adapter jar file

Download the WebSphere Optimized Local Adapter jar file to your workstation from the lib subdirectory of the product directory on the z/OS LPAR. The required file is called `ola_apis.jar`. On our system it was located in the directory at:

/usr/lpp/zWebSphereMC/V7R0/mso/OLA/lib

### Update the build path

To update the build path select the EJB part of the project, then select **Build Path** → **Configure BuildPath** as shown in Figure 4-8.


*Figure 4-8 Start process to modify build path*

On the panel displayed, select the **Libraries** tab, and then click **Add External JARs**, as shown in Figure 4-9.

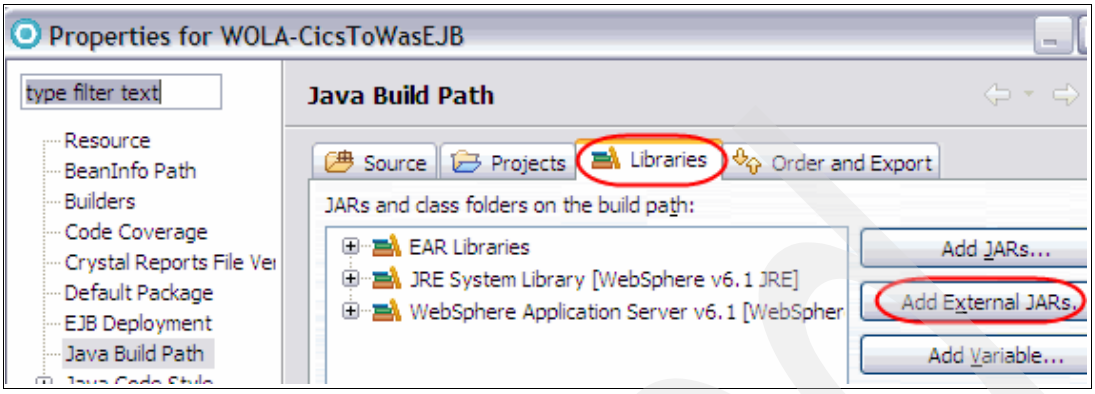



*Figure 4-9 Adding External JAR to the build path*

A file dialog box will open. Use it to locate the `ola_apis.jar` file you downloaded previously and select it. The display will be updated to show that it has been added, as shown in Figure 4-10.

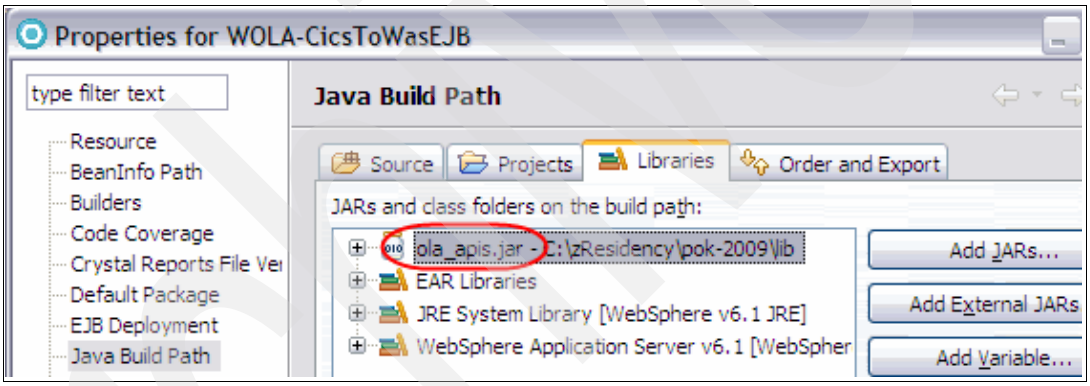



*Figure 4-10 ols_apis.jar added to build path*

Click **OK** to save this change to the build path.

## 4.3.3 Generate EJB skeleton code

The next step is to generate a stateless session EJB. Right-click **CICSToWASEJB**, then select **New** → **Session Bean**, as shown in Figure 4-11 on page 47.

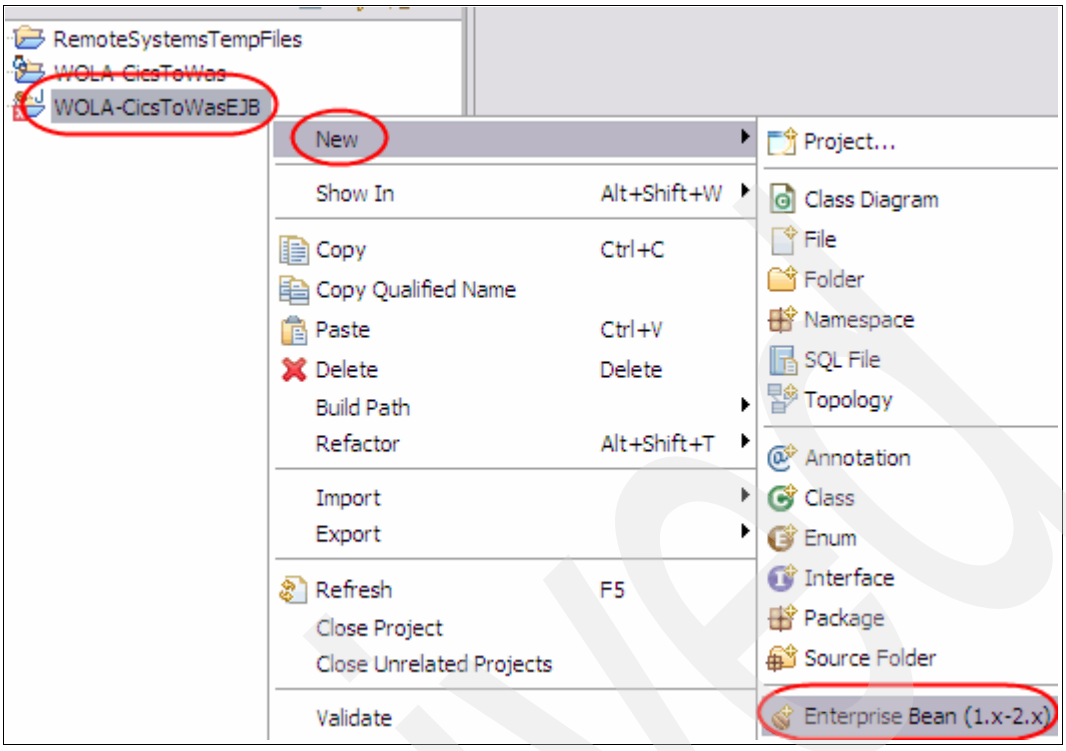



*Figure 4-11 Start the process to create a new EJB*

Set the Java package to `com.ibm.wola.c2w` and the Bean name to `DateSupportRoutines`, then click **Next** as shown in Figure 4-12.

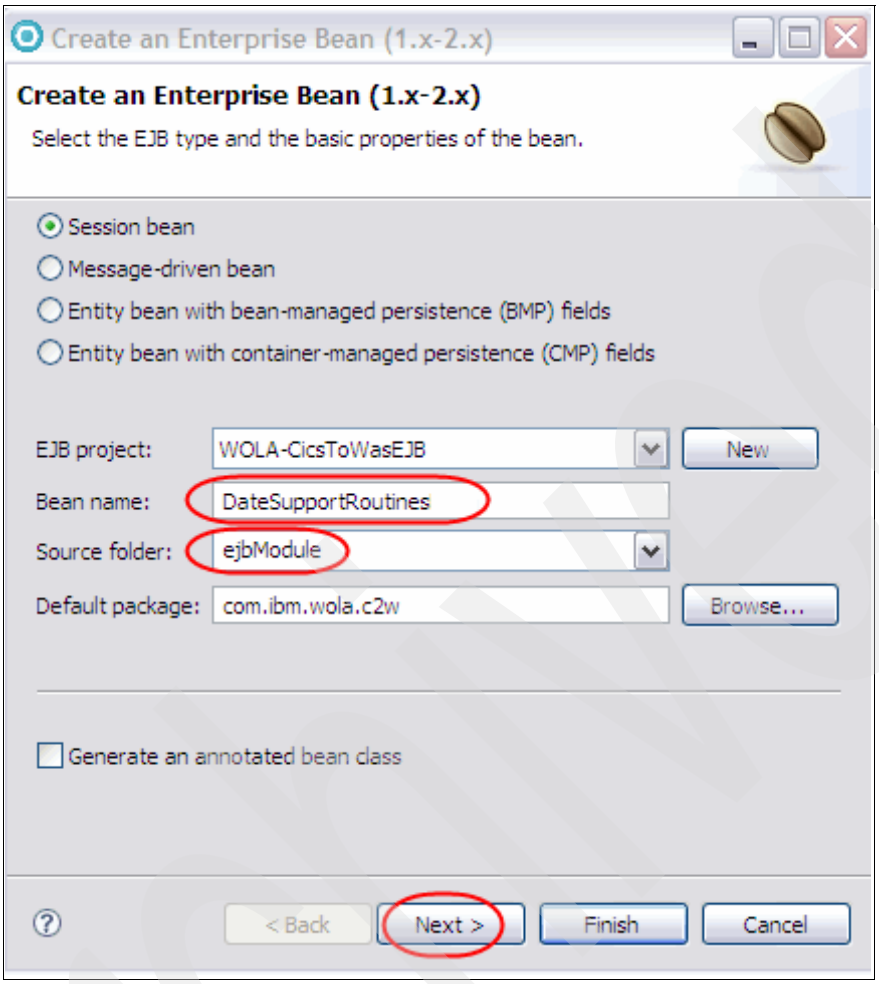



*Figure 4-12 Specify Bean name and package*

### Setting remote interface when ola_apis.jar is present in the build path

Figure 4-13 shows the panel displayed.

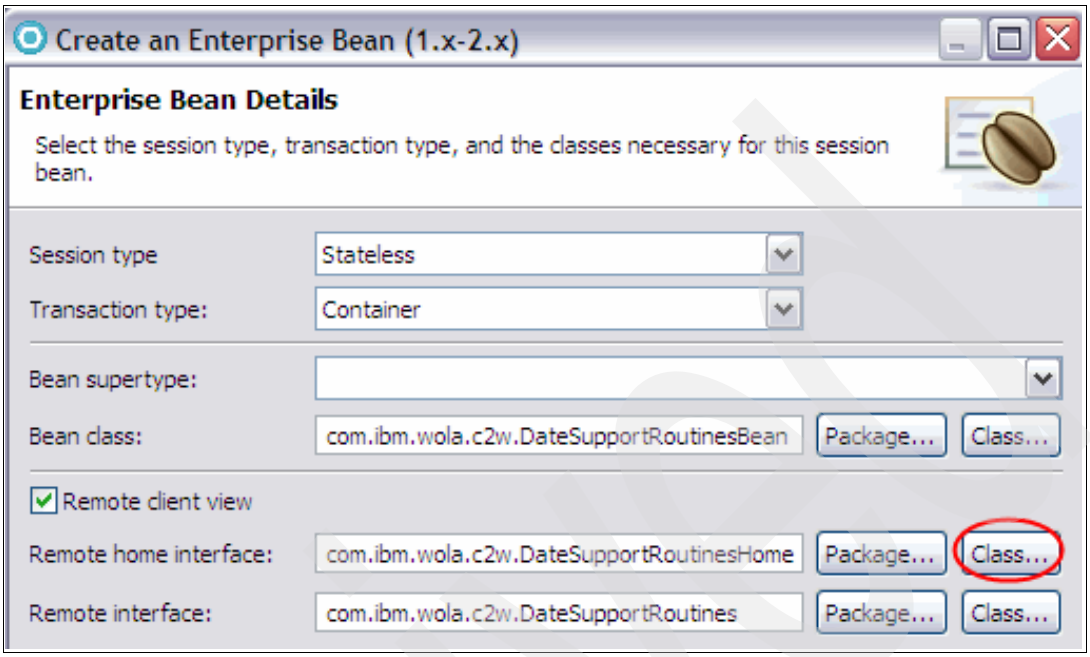



*Figure 4-13 Locating remote interface class when creating EJB*

Because `ola_apis.jar` is on the build path, click **Class** for the Remote Home Interface, which will bring up the display shown in Figure 4-14.

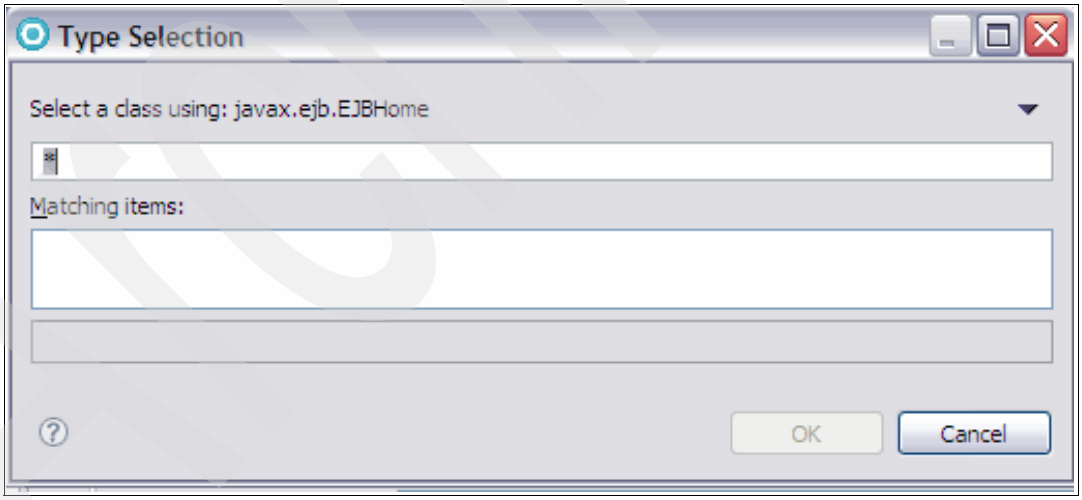



*Figure 4-14 Panel to locate interface class*

In the top box type the string:

```
com.ibm.websphere.ola.
```

Make sure to type in the end full stop after `ola`. Once you type this full stop RAD will search the build path and display a list of classes that can be selected. In this case there will be only one, as shown in Figure 4-15 on page 50.

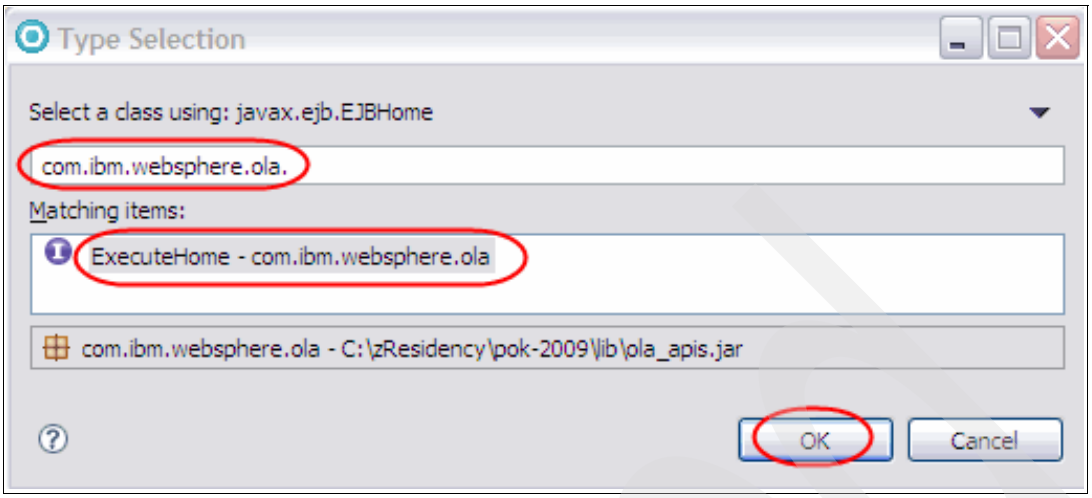



*Figure 4-15 The WebSphere Optimized Local Adapter ExecuteHome class located*

Select the ExecuteHome interface and click **OK**. Perform the same process for the Remote interface, selecting the Execute class. Once this has been done the display will look as shown in Figure 4-16.

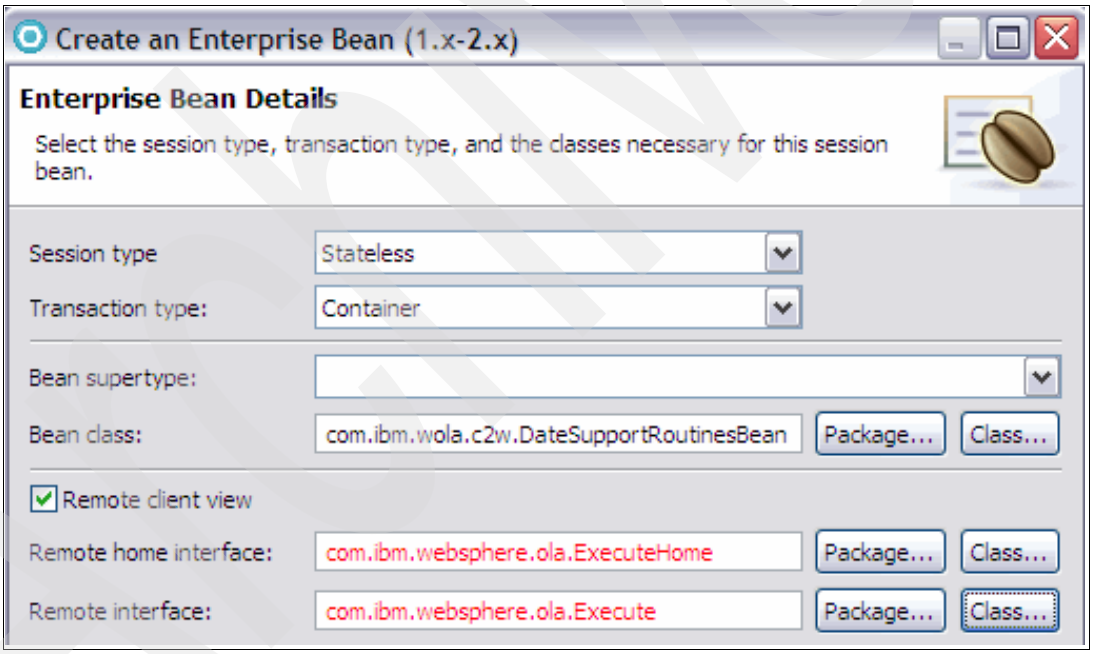



*Figure 4-16 Both remote interfaces set to WebSphere Optimized Local Adapter classes*

Click **Finish** and RAD will generate the EJB skeleton structure.

### Setting remote interface when ola_apis.jar is *not* in the build path

When `ola_apis.jar` has not been added to the build path, you need to type in the required interface names.

On the panel displayed after clicking **Next** on Figure 4-12 on page 48, set the Remote home interface to `com.ibm.websphere.ola.ExecuteHome` and the Remote interface to `com.ibm.websphere.ola.Execute`. This is done by just typing these values into the input boxes. Then click **Finish**, as shown in Figure 4-17 on page 51.

**Important:** The values set for the Remote home interface and Remote interface must be set to the values shown. If they are not set to these values, the call will not succeed.

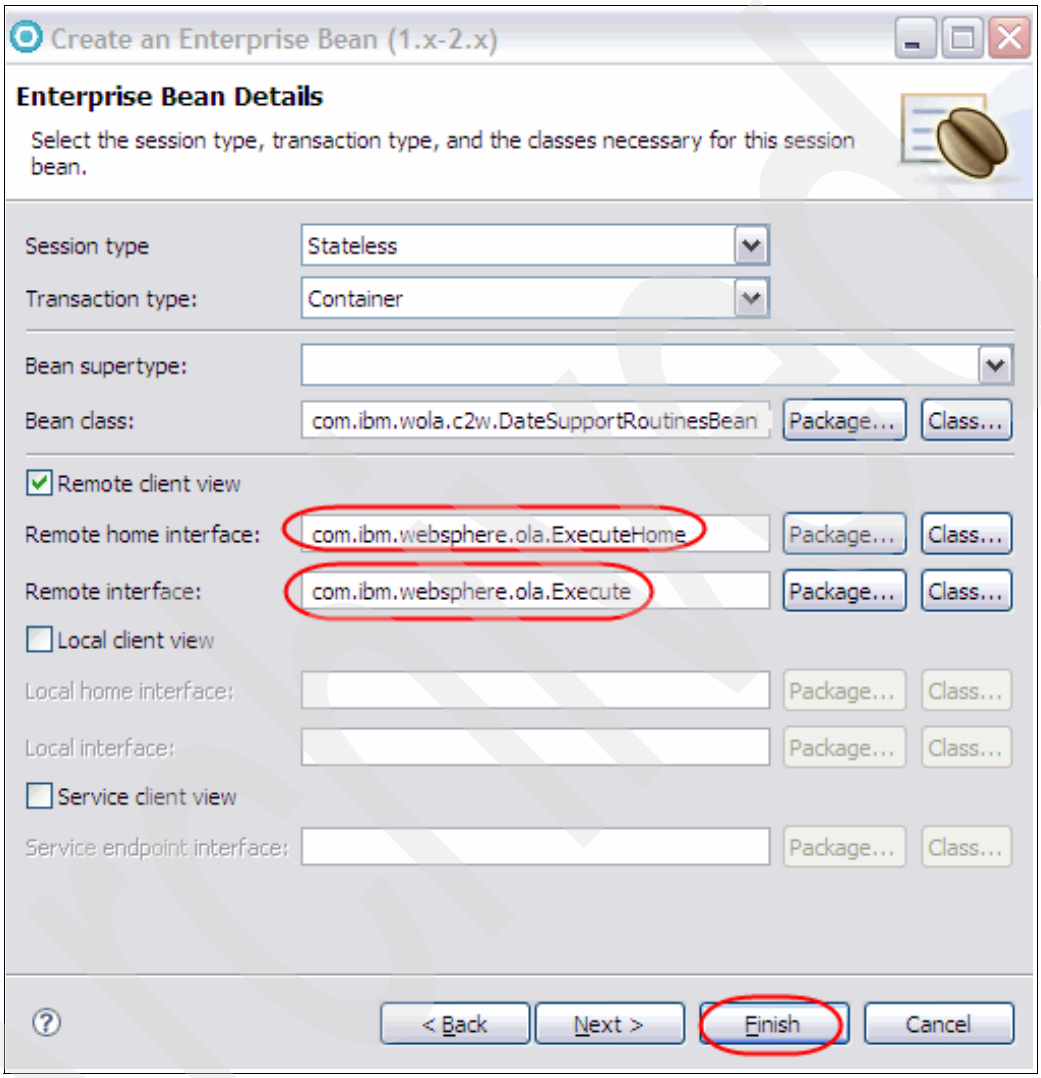



*Figure 4-17 Setting remote interface values*

Click **Finish** and RAD will generate the EJB skeleton structure.

### Confirm enablement panels

One or two panels similar to that shown in Figure 4-18 asking to confirm enablement may be displayed; if so, click **OK**.

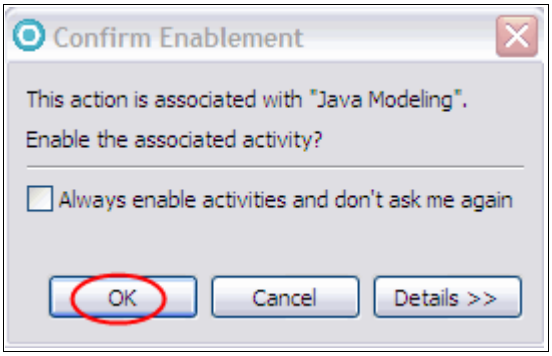



*Figure 4-18 Confirm enablement prompt*

## Result with WebSphere Optimized Local Adapter jar file on the build path

When the WebSphere Optimized Local Adapter `ola_apis.jar` file has been added to the build path, the result will be as shown in Figure 4-19.

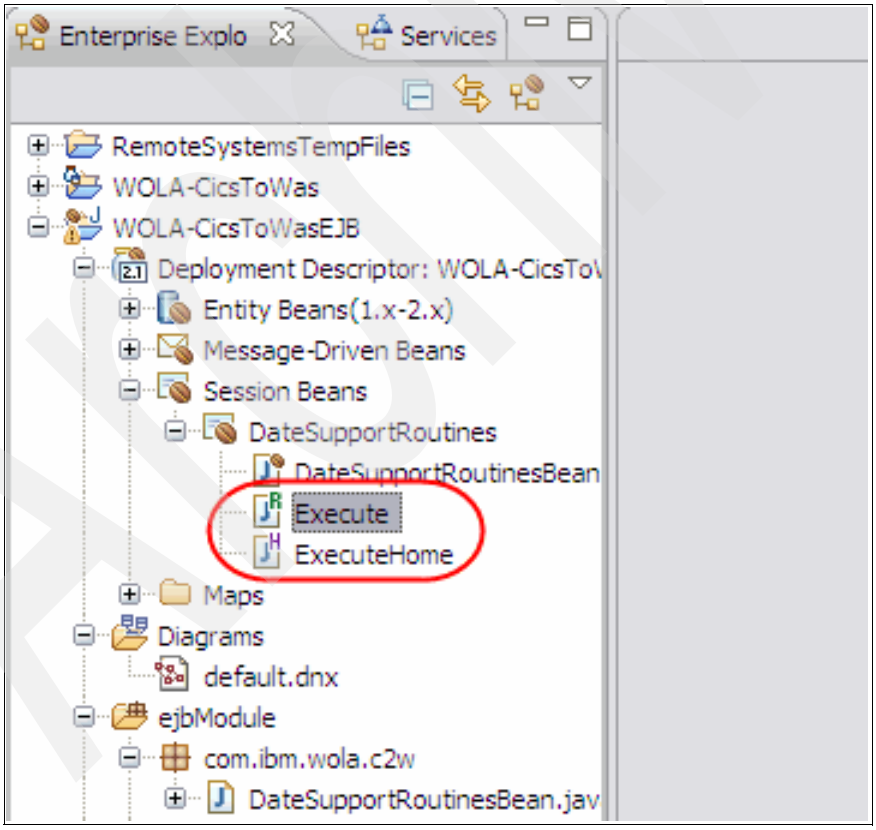



*Figure 4-19 Generated remote interfaces*

### Result when WebSphere Optimized Local Adapter jar file has *not* been added to the build path

When the WebSphere Optimized Local Adapter jar file has not been added to the build path, in the main project view, expanding the project contents will provide a display similar to that shown in Figure 4-20.

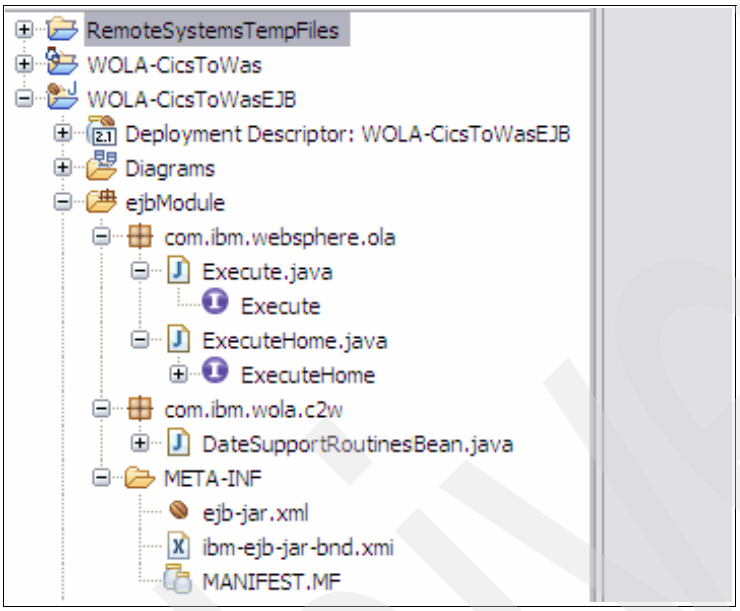



*Figure 4-20 View of the project contents after creating EJB*

## 4.3.4 Generate CommArea helper class

CICS passes and receives data via a CICS CommArea when it calls the method in the EJB. The EJB method needs to extract data from the received CommArea and set data in the CommArea to be returned.

One way to do this would be to develop your own code. However, this is a non-trivial task. The code would need to work out the size in bytes for a data item in the CommArea, calculate the position in a data buffer where data is located, and so on.

RAD provides tooling that can generate code to handle manipulation of the CommArea object, with getter and setter methods for each field in the structure. The generated class can then be used by application code in the EJB method to manipulate the CommArea as required.

Right-click **WOLA-CicsToWas**, select **New** → **Other**, which then displays a new panel, and then select **CICS/IMS Java Data Binding**, as shown in Figure 4-21.

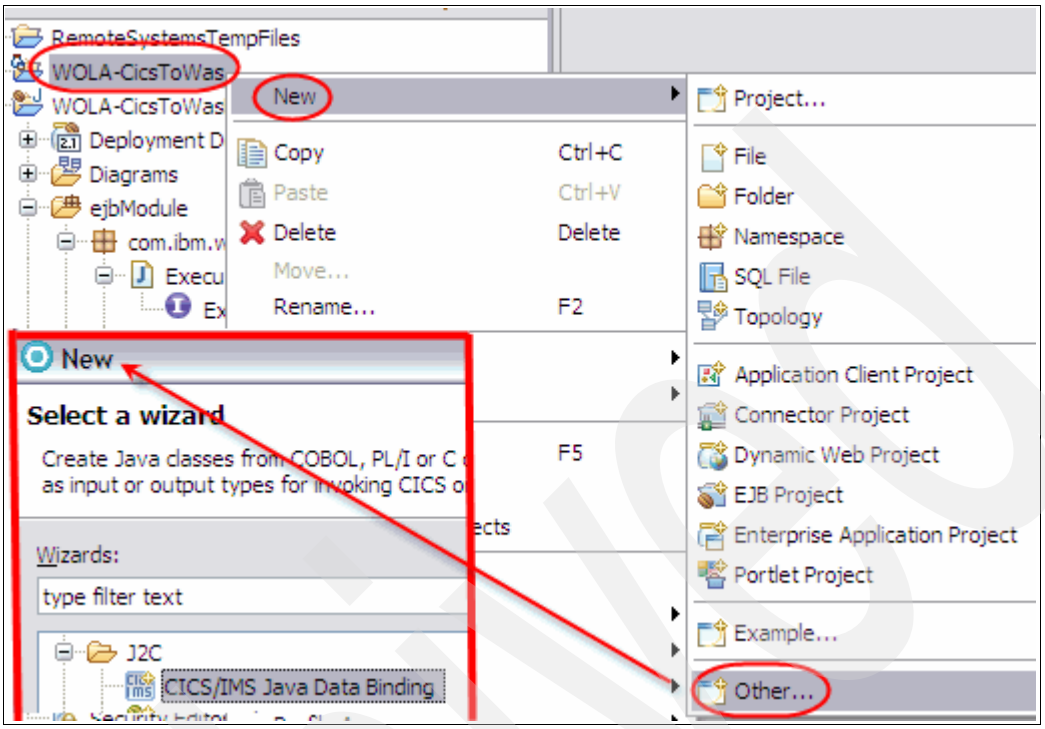



*Figure 4-21 Selecting a wizard to help generate Java class to map CommArea*

On the panel shown in Figure 4-22, click **Browse** and then locate the `EPSL02.cbl` file and click **Open**.

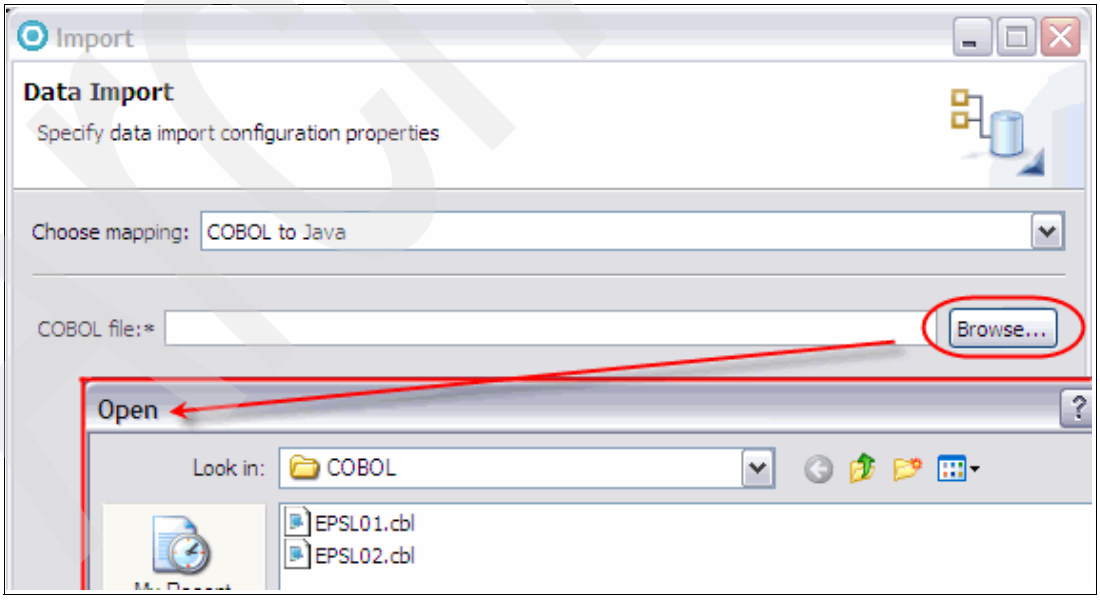



*Figure 4-22 Locating the COBOL source code*

The path and name of the selected file will be set in the COBOL file field, then click **Next**.

On the next panel, set the Platform as z/OS, then click **Query**. This causes RAD to read the COBOL source code and display a list of any data structures it finds. Select the **DFHCOMMAREA** structure and click **Next**, as shown in Figure 4-23.

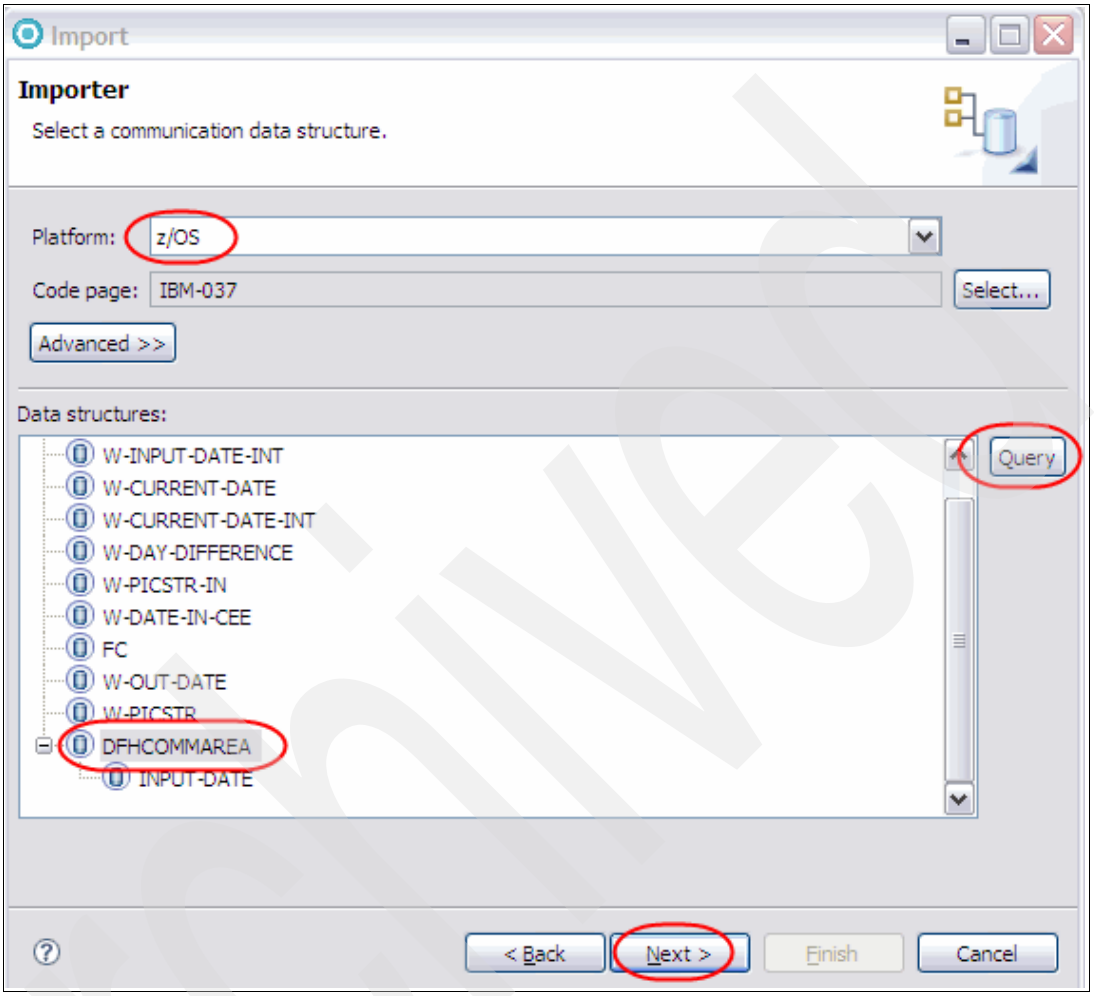



*Figure 4-23 Setting platform type and selecting the COBOL data structure to map*

On the next panel specify the package and class name for the Java class being generated, then click **Finish** as shown in Figure 4-24.

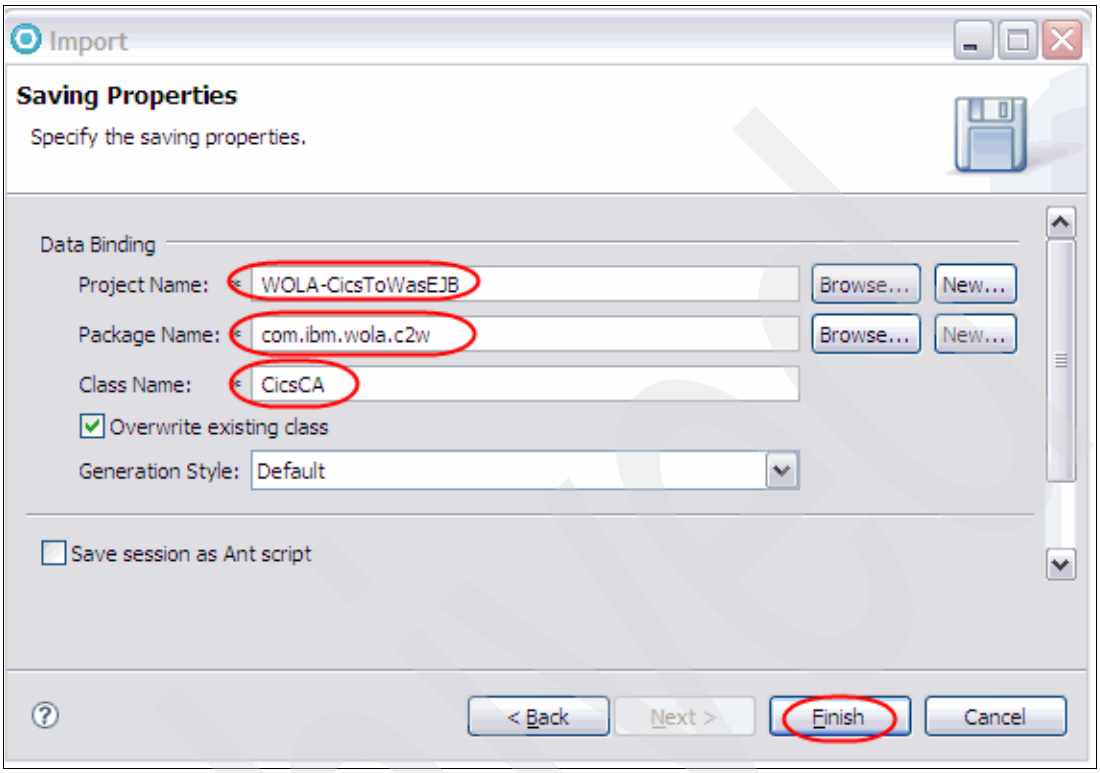



*Figure 4-24 Setting package and class name*

RAD then generates the Java code and opens this code for perusal. In the frame labelled Outline you can see the various getter and setter methods.

When we first started developing the EJB we used the EPSL02 COBOL program as input to the building of this Java class. In “Additional fields in the CommArea” on page 64, we explain how we added some extra fields to the data being passed to the EJB. When we did this, we deleted the CicsCA Java code in RAD, and redid the process using the updated data structure. We copied just the new data structure to a separate file, FTP’ed this down to our PC and then used it in the same way that EPSL02.CBL was used.

### 4.3.5 Code the business logic in a method

The next step is to create a method containing the business logic currently performed in the EPSL02 COBOL program.

**Important:** The method name in the EJB must be *execute*. A method named *execute* must exist for the call to the EJB to succeed.

Only an EJB JNDI name can be specified by the WebSphere Optimized Local Adapter APIs when an external process calls an EJB in WebSphere Application Server.

In the DateSupportRoutines EJB we created a method called *execute*. The method was added to the DateSupportRoutinesBean.java program. The method expects a byte array as input and returns a byte array. This byte array consists of the COMMAREA passed between

the DateSupportRoutines EJB and CICS. We will use the CicsCA Java class to manipulate the byte array.

We then added code to perform same date calculation and date formatting that EPSL02 does.

The data in the received byte array object is stored in the CicsCA Java class by these lines:

```
CicsCA inputCommArea = new CicsCA();
inputCommArea.setBytes(inputFromCics);
```

Once the received data is in the inputCommArea object, the various getter methods in the class can be used to extract data. For example, this line extracts the date passed from CICS:

```
String inputDateDD = inputCommArea.getInput__date__dd();
```

Note that if a COBOL program in CICS needed to pass data greater then 32K in size to the EJB, then the same process described here would be used. The method in the EJB has no intrinsic knowledge of what a CICS CommArea is; to the method it is just an object. The only difference would be that the program in CICS is using Containers rather than CommAreas.

## 4.3.6 Generate deployment code

The next step is to generate deployment code. Right-click **WOLA-CicsToWasEJB**, then **Java EE** → **Prepare for Deployment** as shown in Figure 4-25.

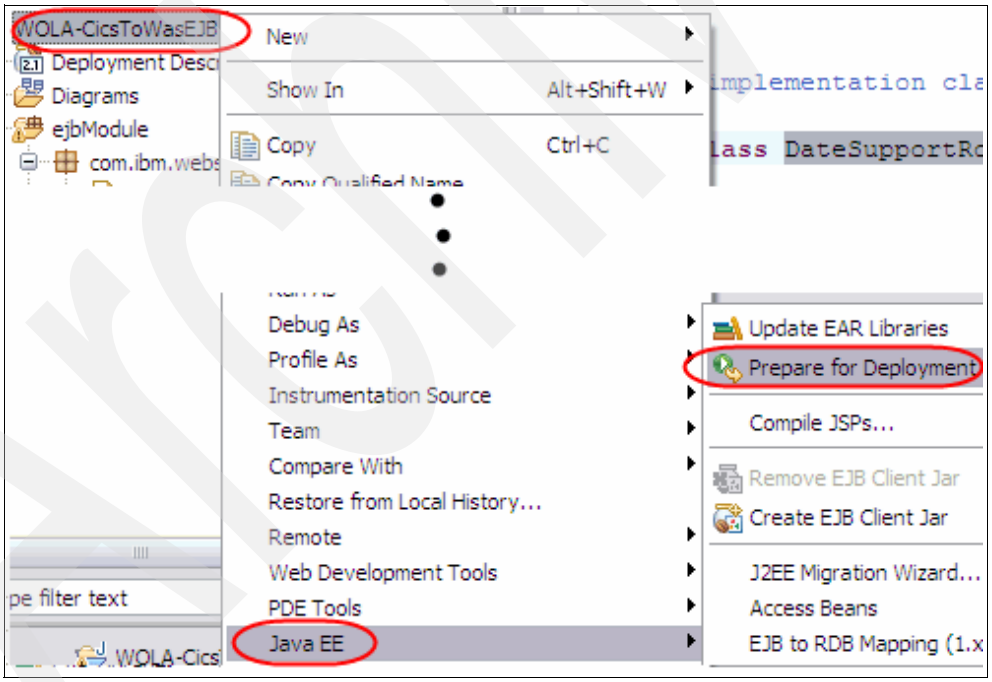



*Figure 4-25 Generating the deployment code*

A number of classes are generated similar to those shown in Figure 4-26 on page 58.

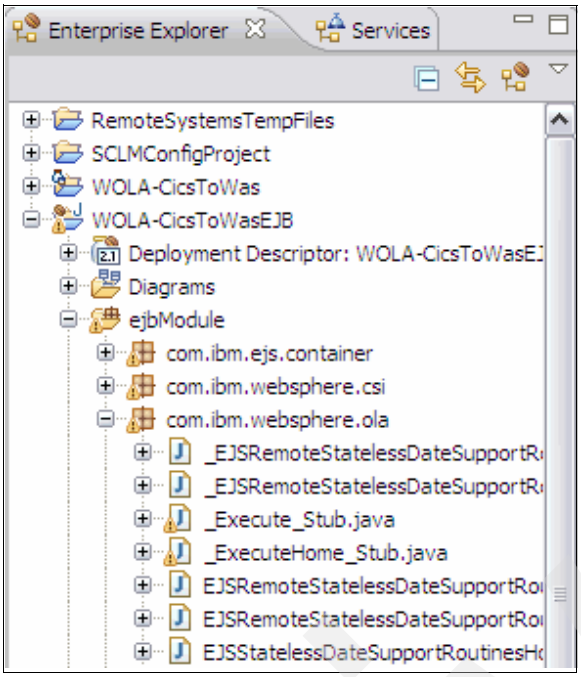



*Figure 4-26 EJB generated classes*

### 4.3.7 Promote the execute method to EJB remote interface

If you did not use the approach described in “Setting remote interface when ola_apis.jar is present in the build path” on page 49, then you need to promote the execute method to the EJB remote interface, so that it can be called from CICS using the WebSphere Optimized Local Adapter APIs.

Expand DateSupportRoutineBean.java, then expand the class object listed under that. Right-click the execute method and select **Enterprise Bean (1.x-2.x)** → **Promote to Remote Interface** as shown in Figure 4-27 on page 59.

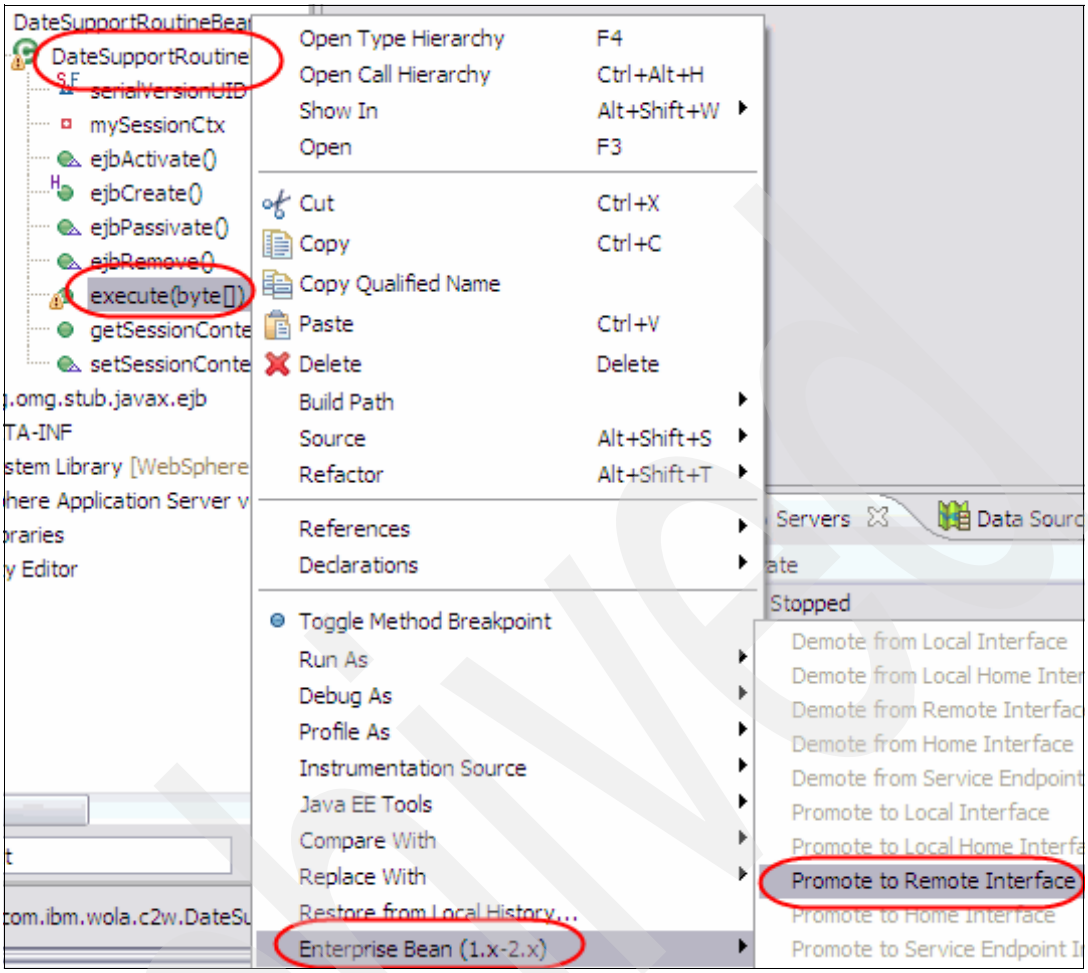



*Figure 4-27 Promoting the execute method to remote interface*

Some of the classes will then have small red boxes with a cross next to them. To clear these, use the **Prepare for Deployment** option as was done in Figure 4-25 on page 57.

## 4.3.8 Update the EJB JNDI name

When a CICS application needs to invoke an EJB using the WebSphere Optimized Local Adapter APIs, it specifies the JNDI name of the EJB. However, RAD will generate a default JNDI name for the called EJB, as shown in Example 4-1:

*Example 4-1 RAD default JNDI name*

```
ejb/com/ibm/websphere/ola/ExecuteHome
```

This value could be used if there is only to be one EJB in the WebSphere server. It is recommended that the default name be replaced with a JNDI name that indicates the name of the EJB being called. This will avoid multiple EJBs in the same WebSphere server having the same default JNDI name. Additionally, when looking at the COBOL source code, the JNDI used in the WebSphere Optimized Local Adapter API call will by implication help identify the EJB being called.

To locate the area where the JNDI name can be modified, locate the ejb-jar.xml entry and double-click it. RAD will open the file. Click the **Bean** tab and then select the

**DateSupportRoutine** entry. Scroll down under the Programming Model Extensions and you will find the JNDI name field, as shown in Figure 4-28.

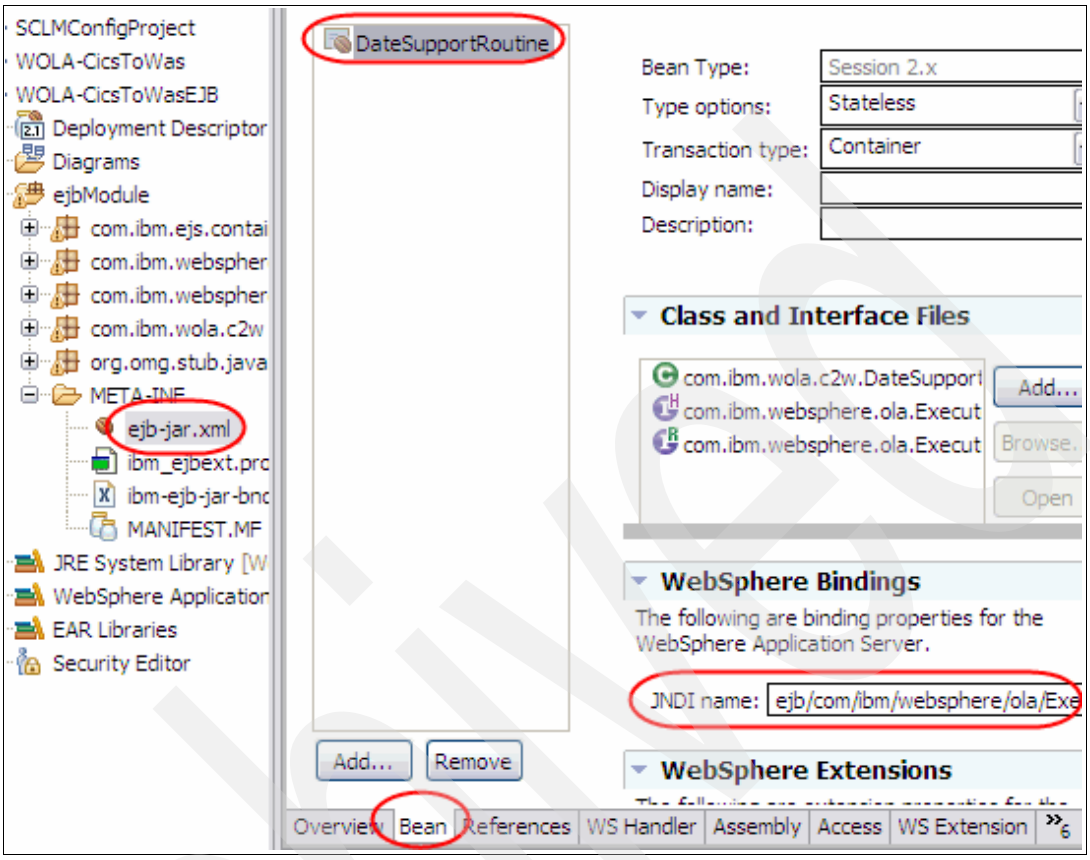



*Figure 4-28 Locating the JNDI name field*

Change the JNDI name to:

```
ejb/wola-c2w/dateSupportRoutineBean
```

and then close the `ejb-jar.xml` file to save the change.

## 4.3.9 Export to an ear file

Next export the application to an ear file, by selecting **WOLA-CicsToWas project** → **Export** → **EAR file** as shown in Figure 4-29 on page 61. In the SaveAs dialog box displayed, provide a local directory on your workstation in which to save the ear file, setting the name of the ear file to `WOLA-CicsToWas.ear` and then click **Save**.

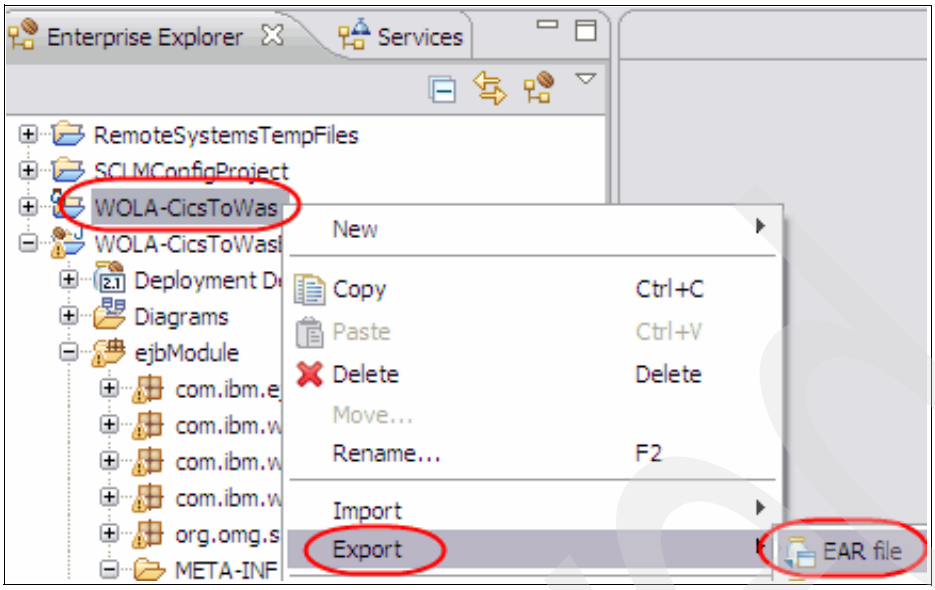



*Figure 4-29 Exporting the application to an ear file*

### 4.3.10 Deploy the application into WebSphere server

We then deployed the `WOLA-CicsToWas.ear` file into a server in a WebSphere Application Server V7 cell. This was done using the WebSphere Administrative GUI via the standard process.

We then started the deployed application.

## 4.4 Change the COBOL program to call EJB

The next step is to change the EPSL01 COBOL program to call the EJB. We made two copies of EPSL01 and called them EPSLW1 and EPSLE1. This is to create both a simple and a complex example for demonstrating the use of the WebSphere Optimized Local Adapter APIs.

### 4.4.1 CICS to WebSphere overview

WebSphere Optimized Local Adapter provides a number of APIs used to allow external processes to invoke an EJB in WebSphere and for an EJB to invoke a function running in an external process. In this chapter we describe the WebSphere Optimized Local Adapter APIs used by an external process, which in our case is CICS, to call an EJB in WebSphere.

The steps required to call an EJB from CICS are:

- The external process registers to the WebSphere cell.
- A connection from the WebSphere Optimized Local Adapter-managed pool is obtained.
- The EJB is called.
- The connection is returned to the WebSphere Optimized Local Adapter-managed pool.
- The process deregisters from the WebSphere cell.

This link from the WebSphere V7 infocenter

http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/index.jsp?topic=/com.ibm.websphere.zseries.doc/info/zseries/ae/cdat_olaapis.html

has this to say about the connections from an external process to WebSphere:

> "When the term local connection is used in this API documentation, we are referring to a cross-memory link that is created for communication between an external address space on the z/OS system and the WebSphere Application Server on the same z/OS system. The WebSphere Application Server and exploiter address space have to be running on the same z/OS image. The adapter API manages these local connections in pools that are associated with each uniquely registered caller. The 12-character registration name can only be used for one set of connection pools per address space. There is no limit to the number of unique registrations in a single address space. It is limited only by the amount of available storage."

## 4.4.2 COBOL samples

The purpose of our sample is to show how we went about modifying the existing EPSL01 program, so that it called the EJB rather than EPSL02.

A good starting place is to review the code in the COBOL samples shipped with the WebSphere Optimized Local Adapter feature. On our system they were located in `/usr/lpp/zWebSphereMC/V7R0/mso/OLA/samples`.

The OLACB06.jclsamp sample shows how to use just three WebSphere Optimized Local Adapter APIs to call the EJB. These are:

- BBOA1REG - registers to WebSphere.
- BBOA1INV - invokes the execute method in the EJB and gets the result.
- BBOA1URG - deregisters from WebSphere.

We decided to change the EPSLE1 program to use the above three APIs. This was to develop a working sample that used the minimum number of APIs to call the EJB.

The OLACB05.jclsamp sample uses six of the WebSphere Optimized Local Adapter APIs to call the EJB. These are:

- BBOA1REG - registers to WebSphere.
- BBOA1CNG - gets a connection from the pool to WebSphere.
- BBOA1SRQ - invokes the execute method in the EJB.
- BBOA1GET - gets the data returned by the EJB.
- BBOA1CNR - returns the connection to the pool.
- BBOA1URG - deregisters from WebSphere.

We decided to change the EPSLW1 program to use the above six APIs in order to develop a working sample that used more of the APIs.

## 4.4.3 The code to be replaced

The original EPSL01 program contained the code shown in Example 4-2 on page 63 to call the EPLS02 program. The CommArea passed contains the data entered on the 3270 panel.

This is the code we need to replace with WebSphere Optimized Local Adapter APIs to call the EJB in the WebSphere Application Server.

*Example 4-2 EPSL01 sample code*

```
A600-CALL-DAY-DIFFERENCE.
    MOVE 'EPSL02' TO W-CALL-PROGRAM
    MOVE 0        TO W-DAY-DIFFERENCE
    MOVE 0        TO W-COMM-PROGRAM-RETCODE

    EXEC CICS LINK PROGRAM( W-CALL-PROGRAM )
                   COMMAREA( DFHCOMMAREA )
    END-EXEC
```

### 4.4.4 The replacement code

In the EPSLE1 program we replaced the lines shown in Example 4-2 with the code in the OLACB06.jclsamp sample that issued the BBOA1REG, BBOA1INV, and BBOA1URG APIs. We also changed the references to the original transaction name of EPSL to EPSE and replaced references to the BMS map name of EPSM01 with EPSMW1.

In the EPSLW1 program we replaced the lines shown in Example 4-2 with the code in the OLACB05.jclsamp sample that issued the BBOA1REG, BBOA1CNG, BBOA1SRQ, BBOA1GET, BBOA1CNR, and BBOA1URG APIs. We also changed the references to the original transaction name of EPSL to EPSW and the replaced references to the BMS map name of EPSM01 with EPSMW1.

We also copied from the EPSLE1 and EPSLW1 samples all of the 01 level variables listed under the comment:

```
* WOLA VARIABLES - START
```

### 4.4.5 Changes to the copied code

The variables copied from the EPSLE1 and EPSLW1 shipped samples contain two variables that specify the name of the WebSphere Application Server on which the EJB is deployed, plus the name of the node that server is located in. These variables as they were copied are:

```
01 nodename                  PIC X(8) VALUE 'SY1     '.
01 servername                PIC X(8) VALUE 'BBOS001 '.
```

We changed these to the values specific to our WebSphere configuration. The *nodename* was set to a value of WPNODEA and the *servername* to WPS01A.

The following line also requires modification:

```
MOVE 'SY1' TO daemonname.
```

The daemonname variable is used on the BBOA1REG call. It specifies the name of the WebSphere Application Server for z/OS daemon group to be joined. To verify the correct value to use, we looked in the job log of the daemon STC of the cell and found this line:

```
BBOM0001I       daemon_group_name: WPCELL.
```

We thus changed the line to:

```
MOVE 'WPCELL' TO daemonname
```

We added the following lines so that the DFHCOMMAREA data structure was passed to the EJB, and the data returned stored in the DFHCOMMAREA:

```
SET rqst-area-addr TO ADDRESS OF DFHCOMMAREA.
MOVE LENGTH OF DFHCOMMAREA TO rqst-len.
SET resp-area-addr TO ADDRESS OF DFHCOMMAREA.
MOVE LENGTH OF DFHCOMMAREA TO resp-len.
```

**Note:** Although this example shows a CICS CommArea being passed, there is no 32 K size limit on the amount of data passed. If a program required that a CICS container be passed to a second program, you could pass the contents of the CICS container to the EJB.

### Trace messages

We also added EXEC CICS WRITE OPERATOR statements to the code in each sample program to write messages to the CICS job log following each WebSphere Optimized Local Adapter API call. These messages identify which WebSphere Optimized Local Adapter API was called along with the subsequent return code and reason code.

## 4.4.6 Additional fields in the CommArea

This sample also provides a good opportunity to examine how security is implemented—in particular, which userid does the EJB run under in WebSphere.

We added the additional fields shown in Example 4-3 to the DFHCOMMAREA data structure.

*Example 4-3 Additional CommArea fields*

```
10  USERID-IN-EJB                PIC X(20).
10  SERVER-NAME                  PIC X(8).
10  SERVER-STC-ID                PIC X(8).
```

The EJB stores the WebSphere principal under which the EJB is running in the USERID-IN-EJB field. The name and started task ID of the WebSphere server are also returned. We added similar fields to the data structure W-COMMUNICATION-AREA because this was used in the transaction processing.

## 4.4.7 Propagation of CICS userid

We wanted to demonstrate how the userid under which the CICS transaction is running can be propagated to the EJB running in WebSphere.

This capability is controlled by settings in the program performing the call to the EJB as well as the setting of an environment variable in the WebSphere Application Server.

### Enabling CICS to support propagation to WebSphere

The BBOA1REG API has a parameter described in the WebSphere V7 infocenter as registerflags. This is a 4-byte field. The infocenter describes that for the userid under which the transaction is running to flow from CICS to WebSphere, bit 29, referred to as `reg_flag_C2Wprop`, must be set to 1. We modified the values for the `regopts` variable as shown in Example 4-4.

*Example 4-4 regopts variable values*

```
01 regopts.
```

```
   05 ro-bytes-1-2              PIC 9(1) comp value 0.
   05 ro-bytes-3-4              PIC 9(1) comp value 0.
```

We then added the following code to set the required bit to 1 if the user indicates to propagate the CICS userid by entering a Y in the new field of the BMS map:

```
IF EPPUIDI = "Y" THEN
   MOVE 4 to ro-bytes-3-4.
```

A value of 4 will set the last four bits of the *ro-bytes-3-4* to a value of '0100'. The 32 bits in the 4-byte `regopts` variable are numbered from 0 starting from the left, which means that bit 29 is the third to the last bit.

**Note:** This same process can be performed from any program running in any type of process calling an EJB in WebSphere that wants to propagate the userid to the EJB in WebSphere.

## Enabling WebSphere to support identity propagation from CICS

In WebSphere the environment variable `ola_cicsuser_identity_propagate` must be set to a value of 1. A detailed explanation of the environment variable is available from the infocenter using the following link:

http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/index.jsp?topic=/com.ibm.websphere.zseries.doc/info/zseries/ae/cdat_olacustprop.html

We defined this environment variable at the server level and then restarted the server.

## Understanding inbound propagation to WebSphere

The BBOA1REG API contains a parameter that can be set to indicate whether the userid under which the CICS transaction is running should be propagated to the WebSphere Application Server.

This means that once the BBOA1REG API call has been issued, this setting cannot be changed unless the program issues a BBOA1URG API call and then issues another BBOA1REG API call with the `registerflags` parameter containing the new settings.

**Note:** A CICS region can issue multiple BBOA1REG API calls with different registration names. Therefore, you could have one BBOA1REG API call with the `reg_flag_C2Wprop` set to 0 and another with `reg_flag_C2Wprop` set to 1.

If a calling program sets the `reg_flag_C2Wprop` parameter to 1 and the target WebSphere Application Server does not have the `ola_cicsuser_identity_propagate` environment variable set to 1, then the registration will fail with a return code of x'08' and a reason code of x'15'. A reason code of x'15' is described in the infocenter as:

- An error occurred while attempting to make contact with the local WebSphere Application Server.
- If you set the reg_flag_C2Wprop bit (bit 29) to 1, ensure that the WebSphere environment variable, `ola_cicsuser_identity_propagate`, is set to 1.

### 4.4.8 Changes to the BMS map

We first made a copy of EPSM01 and renamed it EPSMW1. We then modified this BMS map to include:

- An additional input field which if set to Y will test that the userid under which the transaction is running in CICS is propagated to the EJB running in the WebSphere Application Server.
- Additional output fields to display the userid under which the EJB is executing in the WebSphere Application Server, along with the WebSphere STC name and the job number.

Using the ISPF editor for modifying BMS maps is relatively straightforward and involves editing the source, compiling it, and then running it in CICS to view the effects of the design changes. This can be a painstaking process if you are not used to using ISPF to perform this type of task. Since we already had RDz V7.5 installed on our workstation, we utilized a capability it provides to speed this redesign process.

In our RDz V7.5 we opened the EPSMW1 source using the BMS Map Editor and clicked the **Design** tab. This provides a view in RDz V7.5 in which you can modify the BMS layout by using a *drag and drop* type approach. This made it much simpler and quicker to modify the BMS map for the changes we required.

## 4.5 Running the new COBOL programs to call the EJB

To run the new COBOL programs in our CICS region we needed to perform some actions to prepare the CICS region.

### 4.5.1 Installing the WebSphere Optimized Local Adapter CICS definitions

WebSphere Optimized Local Adapter provides sample CICS CSD definitions for use in CICS. On our system the sample was located at /usr/lpp/zWebSphereMC/V7R0/mso/OLA/samples/CSDUPDAT.jclsamp. We copied this sample, modified the JCL to specify the CSD used by our CICS region, and then ran it. This created two groups:

```
BBOACSD - contains the required definitions to use WOLA in a CICS region
BBOASAMP - definitions to run the supplied WOLA samples
```

We then installed the BBOACSD group into the CICS region using the CEDA transaction.

### 4.5.2 Make the WebSphere Optimized Local Adapter load modules available to CICS

For applications in CICS to issue the WebSphere Optimized Local Adapter APIs, the load modules that provide the WebSphere Optimized Local Adapter support must be available to the CICS region.

The following link to the WebSphere V7 infocenter describes how to execute the olaInstall.sh script. One of the functions performed when running this script is to copy the optimized local adapter load modules to the specified data set.

http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/index.jsp?topic=/com.ibm.websphere.zseries.doc/info/zseries/ae/tdat_enableconnector.html

We ran this script on our system using these commands:

```
cd /wasconfig/wpcell/wpnodea/AppServer/profiles/default/bin
./olaInstall.sh INIT EDMCAR.WOLA.PRODUCT.LOADLIB
```

The output from this command contained the following:

```
Copying /wasconfig/wpcell/wpnodea/AppServer/lib/olaModules/* to
EDMCAR.WOLA.PRODUCT.LOADLIB...
```

Upon completion the EDMCAR.WOLA.PRODUCT.LOADLIB data set contained the required load modules.

We could have added this data set to the DFHRPL DD card of the CICS JCL and then stopped and started the region to make the modules available. However, this was not necessary since CICS provides a means to dynamically add data sets to the DFHRPL via the LIBRARY definition.

In a group named WOLAREDB we defined a new LIBRARY definition named WOLAPROD and set the data set name field to the value EDMCAR.WOLA.PRODUCT.LOADLIB. We then installed this definition into the CICS region.

### 4.5.3 The TRUE exit

Calls to the WebSphere Application Server from CICS using the WebSphere Optimized Local Adapter APIs are implemented using a Task Related User Exit (TRUE).

#### Starting the TRUE exit

The TRUE exit must first be started in the CICS region for any WebSphere Optimized Local Adapter APIs to execute. We started the TRUE exit manually by logging onto the CICS region and entering:

```
BBOC START_TRUE
```

which resulted in the following message:

```
BBOA8001I Exit enabled Successfully.
```

In the CICS job log, in a DD statement named BBOOUT, the following messages appeared, as shown in Example 4-5.

*Example 4-5 TRUE exit start-up*

```
<==========  ADAPTERS CONTROL TASK START ===  Wed Jun 17 02:32:33   =====>
Trace Level defaults to 0. Use TRC=0|1|2 to control tracing.
Enabling BBOATRUE exit.
Return Code: 0 Reason Code: 0
Elapsed time: 0.000000 seconds
Elapsed CPU time: 0.000105 seconds
<==========  ADAPTERS CONTROL TASK END   ===  Wed Jun 17 02:32:33   =====>
```

#### Stopping the TRUE exit

The TRUE exit can be stopped manually by logging onto the CICS region and entering:

```
BBOC STOP_TRUE
```

which results in the following message:

```
BBOA8001I Exit disabled Successfully.
```

### 4.5.4 CICS definitions for our sample

To run our sample programs required the addition of several definitions to the CICS region. We created a group in the CSD called WOLAREDB in which we added the definitions shown in Table 4-1.

*Table 4-1 CICS definitions*

| Definition name | Type | Purpose |
|---|---|---|
| | | |
| EPSLE1 | Program | Modified copy of EPSL01 that uses three WebSphere Optimized Local Adapter APIs |
| EPSLW1 | Program | Modified copy of EPSL01 that uses six WebSphere Optimized Local Adapter APIs |
| EPSL01 | Program | Original initial COBOL program |
| EPSL02 | Program | Called by EPSL01 to perform date calculation |
| EPSMW1 | Program | Modified copy of EPSM01 |
| EPSM01 | Program | BMS Map used by EPSL01 |
| EPSE | Transaction | Invokes the EPSLE1 program |
| EPSL | Transaction | Invokes the EPSL01 program |
| EPSW | Transaction | Invokes the EPSLW1 program |
| WOLAPROD | Library | References EDMCAR.WOLA.PRODUCT.LOADLIB which contains the WOLA product code |
| WOLALOAD | Library | References EDMCAR.WOLA.LOADLIB which contains the COBOL programs |

The program definitions are required because the CICS region we used was not configured to use the CICS automatic program install capability.

We installed these definitions in the CICS region using CEDA.

### 4.5.5 Access to the CBIND SAF class

Access to WebSphere Application Server by an external process using WebSphere Optimized Local Adapter is controlled by the CBIND class. A detailed description of this can be found at:

http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/index.jsp?topic=/com.ibm.websphere.zseries.doc/info/zseries/ae/tdat_security_in.html

To verify which CBIND rule was being used to control access by the CICS region to the server in the WebSphere Application Server cell, we updated the CBIND RACF rule to prohibit access by the userid under which the CICS region was running. When we ran our

new transaction in CICS we received the error message in the system log shown in Example 4-6 on page 69.

*Example 4-6 Access authority error*

```
ICH408I USER(CICSTS  ) GROUP(CICS    ) NAME(CICS Test Region)
  CB.BIND.WP.WPC1A CL(CBIND   )
  INSUFFICIENT ACCESS AUTHORITY
  FROM CB.BIND.WP.** (G)
  ACCESS INTENT(READ   )  ACCESS ALLOWED(NONE   )
```

This message indicates that an SAF call to the CBIND class was made to determine whether the user CICSTS had READ access to a resource called CB.BIND.WP.WPC1A. WPC1A was the cluster transition name for the server. We did not have a specific rule for the server, which meant that RACF used the generic rule CB.BIND.WP.** to determine if access was permitted. As we had modified this rule so that the userid CICSTS had access of NONE, the attempt by CICS to connect to WebSphere failed as expected.

**Note:** It is the userid of the CICS region that is used to check whether access to the CBIND rule is permitted. This is the case even when we ran our test program to try and flow the userid under which the transaction was executing over to the EJB running in WebSphere. In our case the transaction in CICS was running under the userid of EDMCAR. This userid did not require access to the CBIND rule.

After having verified how the CBIND rule controlled access to the WebSphere Application Server, we updated the rule to allow the CICSTS userid access of READ. No restart of either the WebSphere Application Server or the CICS region was required to activate the changes made to the CBIND rule.

### 4.5.6 Running the sample programs

We next tested our WebSphere Optimized Local Adapter-enabled sample programs by logging into the CICS region with the userid EDMCAR, entered the transaction ID EPSE and then clicked Enter. This produced the modified display shown in Figure 4-30 on page 70.

```
                EPS TOOLS BIRTHDAY/RETIREMENT EXAMPLE


     PLEASE ENTER BIRTHDATE: YYYYMMDD

              1    ENTER A 1 TO SEE YOUR BIRTHDAY
                   ENTER A 2 TO CALCULATE YOUR RETIREMENT
                   Propagate userid:  N









WebSphere STC Name:           STC Id:           Uid in EJB:
F3/F12 TO TERMINATE, ENTER TO PROCESS, CLEAR TO START OVER
```

*Figure 4-30 Initial display using the modified BMS*

We then entered a date and set the Propagate userid field to Y and clicked Enter, which produced the result shown in Figure 4-31.

```
                EPS TOOLS BIRTHDAY/RETIREMENT EXAMPLE


     PLEASE ENTER BIRTHDATE: 19920621

              1    ENTER A 1 TO SEE YOUR BIRTHDAY
                   ENTER A 2 TO CALCULATE YOUR RETIREMENT
                   Propagate userid:  Y
     HERE IS YOUR BIRTHDATE AND NO DAYS FROM CURR

  YOUR BIRTHDATE AND DAY:      Sunday 21 June 1992

  HOW LONG AGO WAS THIS?:      000006206 DAYS




WebSphere STC Name: WPS01AS  STC Id: STC01870  Uid in EJB: EDMCAR
F3/F12 TO TERMINATE, ENTER TO PROCESS, CLEAR TO START OVER
```

*Figure 4-31 Test showing logged on CICS userid flowing to EJB in WebSphere*

The data displayed shows that the userid under which the EJB ran in WebSphere is the userid propagated from CICS, namely EDMCAR. The STC name and job number of the WebSphere server that the EJB ran in were also retrieved and displayed.

We next wanted to verify that we could run the EPSLE1 program such that the CICS region userid flowed to the EJB. We clicked **CLEAR**, typed in a date, left the Propagate userid field as N and clicked Enter, which produced the result shown in Figure 4-32 on page 71.

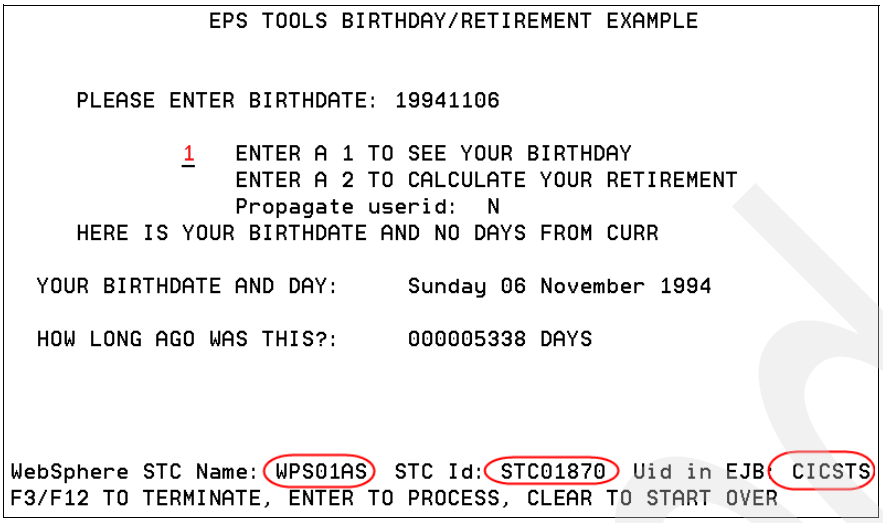



*Figure 4-32 Test showing CICS region userid flowing to EJB in WebSphere*

The data displayed shows that the userid under which the EJB ran in WebSphere is the CICS region userid, namely CICSTS. The STC name and job number of the WebSphere server that the EJB ran in were also retrieved and displayed.

We performed a similar sequence to test the EPSLW1 program by running the EPSW transaction.

### Trace messages

Example 4-7 shows trace messages written to the CICS job log as a result of the EXEC CICS WRITE OPERATOR statements we added to the program.

*Example 4-7* Sample trace messages

```
STC03508  +calling BBOA1REG              RC: 0000 RS: 0000
STC03508  +done    BBOA1REG              RC: 0000 RS: 0000
STC03508  +call    BBOA1CNG              RC: 0000 RS: 0000
STC03508  +done    BBOA1CNG              RC: 0000 RS: 0000
STC03508  +set up for EJB                RC: 0000 RS: 0000
STC03508  +call BBOA1SRQ                 RC: 0000 RS: 0000
STC03508  +done BBOA1SRQ                 RC: 0000 RS: 0000
STC03508  +call BBOA1GET                 RC: 0000 RS: 0000
```

## 4.5.7 Tracing WebSphere Optimized Local Adapter activity in WebSphere

There are two WebSphere Optimized Local Adapter-related trace settings that can be used in WebSphere although IBM does not publish detailed information about the trace data that is written as a result of these settings. However, the trace data may still prove useful for verifying the basic flows of your WebSphere Optimized Local Adapter calls.

### Tracedetail

To trace the flow of WebSphere Optimized Local Adapter-related activities in the WebSphere control region STC, issue the command:

```
f <wasServerName>,tracedetail=g
```

If you executed a program that calls an EJB in WebSphere and no trace messages are produced for the target control region, you can assume that something is wrong, most likely with the parameter settings in the WebSphere Optimized Local Adapter APIs of the calling program.

If trace messages appear but your program does not successfully call the EJB, then at least you know that the call from the external program is arriving at the target WebSphere Application server.

If you know what the data being passed into the EJB looks like, you can check the trace data to see if you can view it there. For example, when we traced our sample, we found a trace record that began with the following, as shown in Example 4-8.

*Example 4-8 Start of sample EJB data trace*

```
Trace: 2009/06/18 10:25:10.828 01 t=7B54C0 c=UNK key=S2
  Description: bbgarcb readComplete sync read RC
  Connection Ptr: 81073A3C0
  Return code: EC
  context and message length: 236
   context and message start: 9A9C91C78
  message:  data_address=00000009a9c91a80, data_length=7
  +-------------------------------------------------------
  |OSet| A=00000009a9c91a80  Length = 02e4 |     EBCDIC
  +----+-----------------------------------+------------
  |0000|C2C2D6C1 D4E2C740 00010000 000002E4|BBOAMSG ....
```

Example 4-9 shows what we found near the end of the trace record.

*Example 4-9 End of sample EJB data trace*

```
|02a0|F0F00000 00000000 00000000 00000000|00..............|
|02b0|E4F0F0F0 F1F7F8F8 F0F1F2F6 F0F4F4F8|U000178801260448|
|02c0|00000000 00000000 00000000 00000000|................|
|02d0|00000000 00000000 00000000 00000000|................|
```

Observe that the 17880126 data shown in Example 4-9 represents the date we entered on the CICS panel. This tells us that the data we entered in CICS was successfully passed to WebSphere.

Within the trace record we also found the lines shown in Example 4-10.

*Example 4-10 Trace of JNDI name*

```
Service name:  data_address=00000009a9c91abc, data_length=2
+-------------------------------------------------------
|OSet| A=00000009a9c91abc  Length = 0100 |     EBCDIC     |
+----+-----------------------------------+----------------+
|0000|85918261 A6969381 6083F2A6 618481A3|ejb/wola-c2w/dat|
|0010|85E2A497 979699A3 D996A4A3 899585C2|eSupportRoutineB|
|0020|85819500 00000000 00000000 00000000|ean.............|
|0030|00000000 00000000 00000000 00000000|................|
```

This is the JNDI name we set in our sample COBOL program. You can compare the values seen in the trace data with what you believe you coded in the calling program.

### Java trace

The WebSphere Optimized Local Adapter Java classes in WebSphere can be traced by specifying one of the following trace settings either in the server via the WebSphere Administration GUI:

com.ibm.ws390.ola.*=all

or by issuing the following operator command:

```
f <wasServerName>,tracejava="com.ibm.ws390.ola.*=all"
```

The WebSphere Optimized Local Adapter functions are for the most part implemented in non-Java code. Therefore, no Java trace output is produced for the processing of WebSphere Optimized Local Adapter calls into WebSphere from an external process. You may observe Java trace data in the WebSphere control region for the WebSphere Optimized Local Adapter Java classes by issuing a WebSphere Optimized Local Adapter-related DISPLAY command such as:

```
F WPSO1A,DISPLAY,ADAPTER,REGISTRATIONS
```

## 4.5.8 Tracing WebSphere Optimized Local Adapter activity in CICS

Tracing the WebSphere Optimized Local Adapter-based activity in CICS is initiated by starting the TRUE exit with additional trace parameters; for example:

```
BBOC START_TRUE XTR=Y TRC=2
```

When we ran our sample program we saw trace messages in the BBOOUT DD of the CICS region, shown in Example 4-11 on page 73.

*Example 4-11 TRUE exit trace data*

```
BBOATRUE: WAS-CICS TRUE entered.
BBOATRUE: R1 on entry            ----->>        15BF7EA8
BBOATRUE: TRUE Caller R1         ----->>        16C98FC0
BBOATRUE: TRUE Caller Stub EPA   ----->>        16D13630
BBOATRUE: Called from stub       ----->>        BBOA1CNG
BBOATRUE: Returned from PC.
BBOATRUE: WAS-CICS TRUE exiting.
```

### Dumping trace recorded in WebSphere

Starting the TRUE exit in CICS with the trace option signals to the WebSphere Optimized Local Adapter feature in WebSphere to record trace information in an internal trace buffer. The contents of this buffer can be dumped to the WebSphere server control region log by issuing a command similar to:

```
F <wasCRname>,DISPLAY,OLATRACE=<jobname>
```

To test this trace function we issued the following command:

```
F WPSO1A,DISPLAY,OLATRACE=SCSCPAZS
```

Example 4-12 shows two lines of the control region SYSPRINT output.

*Example 4-12 SYSPRINT output*

```
******** OLATRACE Modify command output start **
******** Trace records for Jobname: SCSCPAZS
```

These two lines were followed by a series of messages from the trace buffer. IBM does not publish information about how to interpret these trace messages.

### 4.5.9 Display registrations

A DISPLAY command issued against the WebSphere server will show information about registrations to that server. For example, we issued this command on our system:

```
F WPS01A,DISPLAY,ADAPTER,REGISTRATIONS
```

which produced the output shown in Example 4-13.

*Example 4-13 DISPLAY command output*

```
BBOA0006I: SHOWING REGISTRATIONS FOR SERVER:
BBOA0000I: TYPE: s JOBNAME: WPS01A   NAME: *WASCTL*
BBOA0001I:  JOBNUM: 0 ACTIVE: true ACTIVE-CONNECTIONS: 0
BBOA0002I: MIN-CONN: 0 MAX-CONN: 0 STATE: 00 TRACELEVEL: 00
BBOA0023I: THIS REGISTRATION DOES NOT HAVE ANY CONNECTION HANDLES
BBOA0026I:
BBOA0003I: Name         Jobname  SWT TL Min  Max  Act  State
BBOA0004I: WOLAJM01REG  SCSCPAZS 000 00 0001 0010 0008 02
BBOA0004I: YLACB05REG   SCSCPAZS 000 00 0001 0010 0001 02
BBOA0004I: KLACB05REG   SCSCPAZS 000 00 0001 0010 0001 00
BBO00188I END OF OUTPUT FOR COMMAND DISPLAY,ADAPTER,REGISTRATIONS
```

The value under the State column indicates the trace level that was set by the connecting process.

## 4.6 WebSphere Optimized Local Adapter and CICS for real world applications

The samples described in this chapter are not indicative of the way you would go about coding real world applications to use WebSphere Optimized Local Adapter APIs. The samples we developed are self contained, each doing the complete process of calling an EJB, namely registering to WebSphere, calling the EJB, and then un-registering from EJB.

This is a convenient method for developing simple applications to test the basic functionality of WebSphere Optimized Local Adapter-based applications in your environment. However, it is not a recommended way of coding applications for production environments.

For production applications in CICS, you would create a mechanism that executes during CICS startup to perform one or more registrations to one or more WebSphere servers. Subsequent transactions would then call EJBs as needed, most likely just using the BBOA1INV API.

Our sample EPSLW1 COBOL program implemented six of the WebSphere Optimized Local Adapter APIs. It is believed that advanced exploiters will make effective use of those WebSphere Optimized Local Adapter APIs, for example, to carefully manage connections to WebSphere.

## 4.7 Additional materials

The samples used in this chapter are available for download as additional material from the Web site location of this Redpaper.

The additional material for this chapter is supplied in a zip file called `wola-ch4-add-Materials.zip`. When unzipped, a directory called `wola-ch4-add-Materials` is created. Table 4-2 lists the contents.

*Table 4-2 Additional materials*

| Directory | Content |
|---|---|
| | |
| bms | Contains BMS Map source code |
| cobol | Contains COBOL source code |
| csd | Contains definitions to define resources in CICS CSD for the samples |
| ear | Contains ear file with the EJB |
| jcl | Contains the JCL used to compile COBOL and BMS Maps |
| RAD-pi | Contains RAD project interchange file, this can be imported into RAD to view Java source |
| xmitFiles | Contains XMIT files |

### 4.7.1 XMIT files

In the xmitFiles subdirectory are five files. The contents of each XMIT file is self evident from the name. These can be FTP'ed in binary to the z/OS LPAR. To extract the content of each file, issue a command of the form:

```
receive inds(WOLA.REDB.cobol)
```

You will be prompted to enter a data set name into which the files will be placed, or just press Enter to accept the default data set name.

## 4.8 Summary

In this chapter we took an existing CICS application where one COBOL program called another, and showed how with fairly minimal effort we were able to change the COBOL program to call an EJB in WebSphere.

5

# WebSphere Optimized Local Adapter - Outbound to CICS scenario

This chapter provides a brief introduction to J2EE Connectors and a description of how WebSphere Optimized Local Adapter implements the J2EE Connectors classes. Also covered are the steps required to implement WebSphere Optimized Local Adapter classes in a simple J2EE application that accesses a CICS application using both CICS COMMAREA and Container interfaces.

# 5.1 Introduction to J2EE Connector Architecture

Sun's J2EE Connector Architecture (J2CA) has standardized the Java classes needed by an application to access an Enterprise Information System (EIS) like CICS. This section provides a brief introduction to the principles and concepts of J2CA that apply to WebSphere Optimized Local Adapters.

For detailed information about the J2C, refer to:

```
http://java.sun.com/j2ee/connector/
```

**Note:** The acronyms JCA and J2EE Connector (J2C) sometimes appear to be used interchangeably. This chapter uses J2CA to refer to the architecture and J2C to refer to the implementation of the architecture. JCA was intentionally not used since it is reserved for use as an acronym for Java Cryptographic Architecture.

## 5.1.1 Connector components

Figure 5-1 illustrates the main concepts behind J2CA.

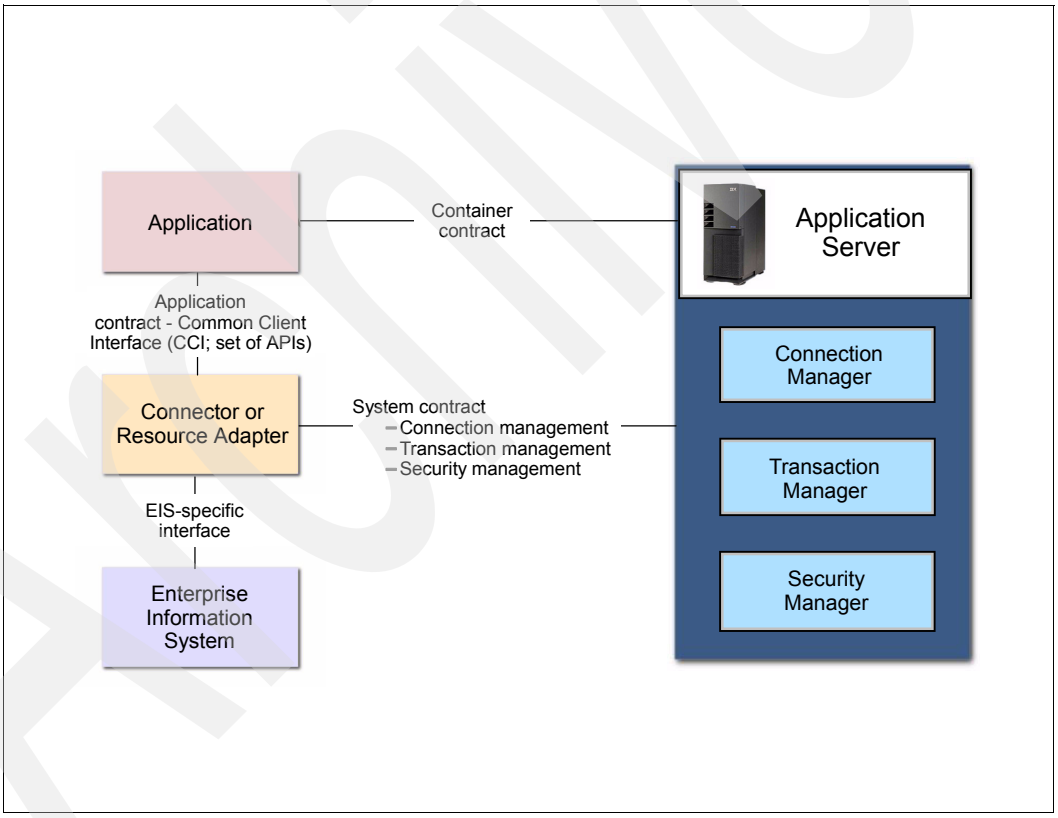



*Figure 5-1 J2EE Connector Architecture*

The *application server* is the runtime engine or EJB or Web Server runtime container. It provides such facilities as transaction support, security support, and persistence capabilities.

The connector or *resource adapter* (for example, WebSphere Optimized Local Adapter) provides the interface between the Java client application and the support provided by the application server.

The *application* represents the client application code which uses the client application programming interface (API) provided by the resource adapter. This API is called the Common Client Interface (CCI). It defines a common API, which a client program can use to access the EIS. The CCI is the connection between the client program and the resource adapter and is the main focus of this chapter.

The *Enterprise Information System* is the back-end system (CICS in our case) where the business logic processing occurs.

### 5.1.2 The Common Client Interface

CCI defines a unified remote function call interface, which focuses on executing functions in the EIS and retrieving the results. From a programming perspective, this means that programmers only have to use a single unified interface with which they can get data from the EIS (for example, from CICS). The EIS-supplied resource adapter handles abstracting the difference and provides a unified programming model to the programmer. This model is independent of actual EIS behavior and communication requirements.

### 5.1.3 Establishing a connection to a resource

The J2EE Connector Framework supports access to EISs from both managed and non-managed connections (for a WebSphere Optimized Local Adapter connection we are only concerned with managed connections). A managed connection means that the application server (that is, WebSphere Application Server) handles all aspects of the connection. The application server handles the Quality of Service (QoS). This includes, for example, providing a connection factory instance, establishing the connection to an EIS connection, and finally freeing the connection from a connection pool when it is no longer needed.

Figure 5-2 on page 80 illustrates the process of establishing a managed connection to a resource. Since the application server manages the connections and provides the QoS, the application starts the process with a request to the Java Naming and Directory Interface (JNDI) for a connection factory lookup (Step 1).

JNDI returns a Connection Factory object to the client application (Step 2). A factory object can create other objects, in this case connections.

To create a connection with specific WebSphere Optimized Local Adapter properties, the client application creates a Connection Specification object with the desired WebSphere Optimized Local Adapter properties. This object, along with the Connection Factory object, is sent to the Connection Manager requesting a Connection object with these properties (Step 3).

A Connection object is returned to the application with the QoS as specified by the client application (Step 4). This Connection object is used by the client application to create an Interaction object which is used to access WebSphere Optimized Local Adapter services (Step 5), which in turn accesses the target EIS system (Step 6).

The WebSphere Optimized Local Adapter component (the CICS link server) executing in the CICS region links to the requested CICS program using either a CICS COMMAREA or Container (Step 7). The CICS program is executed and returns the results in either a CICS COMMAREA or Container, and this response flows back to the client application executing in WebSphere Application Server.

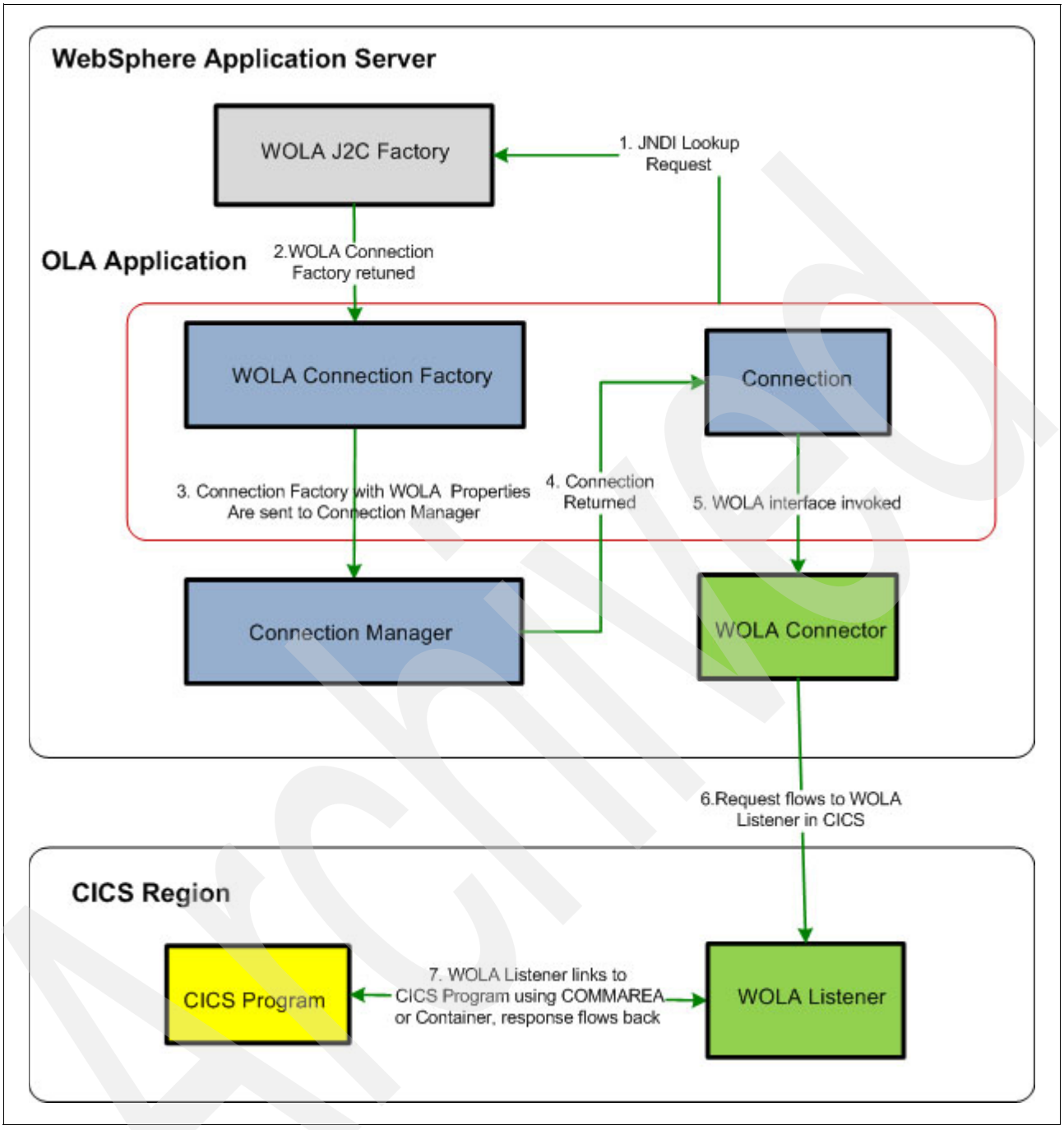



*Figure 5-2 Establishing a connection to CICS*

## 5.2 Exploring WebSphere Optimized Local Adapter implementation of CCI

Before starting any development of the client application, we wanted to explore the WebSphere Optimized Local Adapter extensions to the CCI classes. We went to the WebSphere Application Server for z/OS InfoCenter Web site and learned that there are three WebSphere Optimized Local Adapter CCI classes used by a client:

- com.ibm.websphere.ola.ConnectionSpecImpl
- com.ibm.websphere.ola.InteractionSpecImpl
- com.ibm.websphere.ola.IndexedRecordImpl

We explored these classes and determined the following.

### 5.2.1 Class com.ibm.websphere.ola.ConnectionSpecImpl

This class is used to pass WebSphere Optimized Local Adapter-specific data to the WebSphere Optimized Local Adapter resource adapter in order to customize the connection established with the target of the outbound request. This class has setter (as well as getter) methods which are used to provide the desired WebSphere Optimized Local Adapter connection properties.

| method | Usage |
|---|---|
| setConnectionWaitTimeout(int) | Set the number of seconds to wait for a connection. |
| setLinkTaskReqContID(String) | Set the name of the container used to pass the request message to a CICS program. |
| setLinkTaskReqContType(int) | Set the type of request container to use. Specify 0 to indicate a CHAR container, or 1 to indicate a BIT container. |
| setLinkTaskResContID(String) | Set the name of the container to pass the response message in from a CICS program. |
| setLinkTaskRspContType(int) | Set the type of response container to use. Specify 0 to indicate a CHAR container, or 1 to indicate a BIT container. |
| setLinkTaskTranID(String) | Set the CICS transaction name used to run the WebSphere Optimized Local Adapter Program Link invocation task. |
| setRegisterName(Sting) | Set the name of the register to which to connect. |
| setUseCICSContainer(Boolean) | Set to true to use containers when communicating with the WebSphere Optimized Local Adapter CICS Link Server, otherwise set to false. |

### 5.2.2 Class com.ibm.websphere.ola.InteractionSpecImpl

This class is used to pass the service name to the WebSphere Optimized Local Adapter resource adapter in order to identify the target application, which is the CICS program name. This class has a setter (as well as a getter) method that is used to provide the WebSphere Optimized Local Adapter properties.

| Method | Usage |
|---|---|
| setServiceName(String) | Set the name of the service to which to execute. |

### 5.2.3 Class com.ibm.websphere.ola.IndexedRecordImpl

This class is used for the request and response messages sent and received between the client application and WebSphere Optimized Local Adapter resource adapter. This class has several methods but for our purposes we are interested in the setter and getter methods that are used to add the client's request to the message provided to the WebSphere Optimized

Local Adapter resource adapter and to retrieve the response from the WebSphere Optimized Local Adapter resource adapter after the service has been executed.

| Method | Usage |
|---|---|
| add(Object) | Add a Java object to the list of elements. |
| get(Object) | Retrieve the first object from the list of elements. |

**Note:** For examples of these classes being used, see Example 5-3 on page 89.

## 5.3 Developing a WebSphere Optimized Local Adapter client to access a sample application

To demonstrate the development of a client that uses WebSphere Optimized Location Adapters we used two simple CICS programs, each of which provides the same function. Program CSCVINQ (written in the C programming language) is passed an employee number in a COMMAREA and uses this number to access the CICS IVP VSAM file (FILEA) to insert, update, delete, or retrieve an employee record using the employee number as the key. Program CSCVINC (written in COBOL) performs the same function but uses CICS containers to pass the request to the program and return the response.

Rational Application Developer V7.5 (with J2C support installed) was used to build the Java artifacts required to develop the OLA-enabled client. These artifacts include the Java classes that represent the COMMAREA and Containers. We started by developing these Java classes first.

### 5.3.1 Preparing the RDz workspace

To prepare the RDz workspace for developing the application, we created an Enterprise Application Project named *ItsoWolaEar* and a new EJB Project named *ItsoWolaEJB* with an EJB Client Jar module project named *ItsoWolaEJBClient.* Each project was added to the ItsoWolaEar Enterprise Application; see Figure 5-3 on page 83.

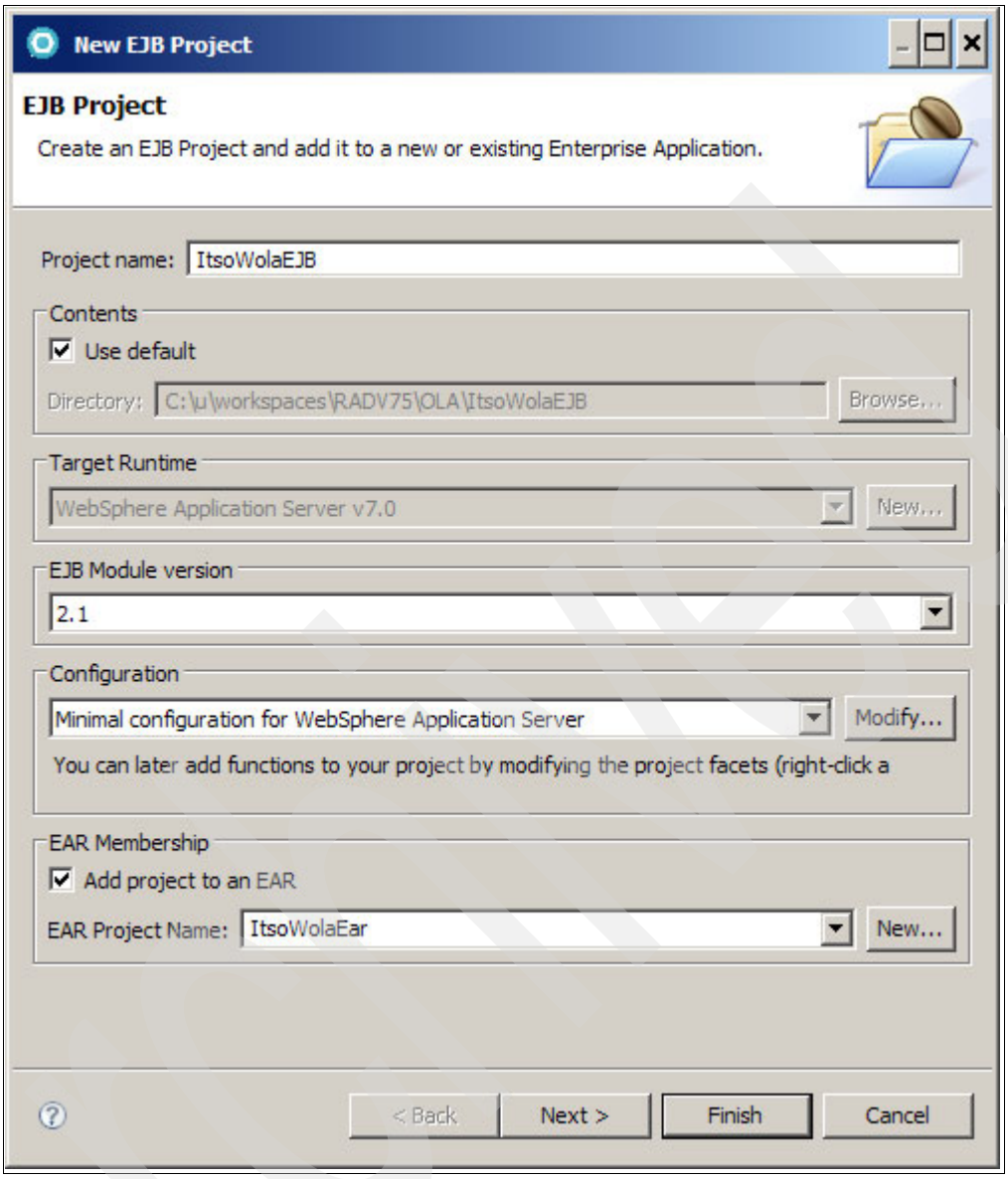



*Figure 5-3 New EJB project*

## 5.3.2 RDz and the CICS sample source

The RDz tooling or wizards that are used to create the Java representations of a COMMAREA or Container work with COBOL, C, and PL/1 source. But the RDz tooling is easier to use if, rather than working with the complete source of a program, we used a stripped-down or template version of the program that included just the COMMAREA or Container areas in COBOL instead. Since the RDz J2C tooling is not concerned with business logic, only with the layouts of the communication areas, creating a template version of any program for use in RDz works quite well.

Program CSCVINQ was written in the C programming language, but we created the COBOL program in Example 5-1 to represent the COMMAREA used by this program.

*Example 5-1 Program CSCVINQ template source*

```
IDENTIFICATION DIVISION.
```

```
       PROGRAM-ID. CSCVINQ.
       ENVIRONMENT DIVISION.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01  COMMAREA.

           03  ACTION         PIC X.
           03  CEIBRESP       PIC S9(8) COMP.
           03  CEIBRESP2      PIC S9(8) COMP.
           03  USERID         PIC X(8).
           03  STAT           PIC X.
           03  NUMB           PIC X(6).
           03  NAME           PIC X(20).
           03  ADDRX          PIC X(20).
           03  PHONE          PIC X(8).
           03  DATEX          PIC X(8).
           03  AMOUNT         PIC X(8).
           03  COMMENT        PIC X(9).

       LINKAGE SECTION.
       01  DFHCOMMAREA        PIC X(120).

       PROCEDURE DIVISION.

           EXEC CICS RETURN END-EXEC.
           GOBACK.
```

Note: We encountered issues when using the RDz wizards with the C source to create the J2C classes for this COMMAEA structure. Converting the C source to COBOL was the only reliable way to create the J2C record classes.

Program CSCVINC is written in COBOL but was simplified for RDz wizards by using the sample source in Example 5-2.

*Example 5-2 COBOL Container template for program CSCVINC*

```
IDENTIFICATION DIVISION.
       PROGRAM-ID. CSCVINC.

       ENVIRONMENT DIVISION.
       CONFIGURATION SECTION.
       DATA DIVISION.
       WORKING-STORAGE SECTION.

       01 Request-Message.
           03  ACTION         PIC X.
           03  USERID         PIC X(8).
           03  STAT           PIC X.
           03  NUMB           PIC X(6).
           03  NAME           PIC X(20).
           03  ADDRX          PIC X(20).
           03  PHONE          PIC X(8).
           03  DATEX          PIC X(8).
           03  AMOUNT         PIC X(8).
```

```
    03  COMMENT        PIC X(9).
01 Response-Message.
    03  ACTION         PIC X.
    03  CEIBRESP       PIC S9(8) COMP.
    03  CEIBRESP2      PIC S9(8) COMP.
    03  USERID         PIC X(8).
    03  STAT           PIC X.
    03  NUMB           PIC X(6).
    03  NAME           PIC X(20).
    03  ADDRX          PIC X(20).
    03  PHONE          PIC X(8).
    03  DATEX          PIC X(8).
    03  AMOUNT         PIC X(8).
    03  COMMENT        PIC X(9).

 PROCEDURE DIVISION.
MAIN-PROCESSING SECTION.
EXEC CICS RETURN END-EXEC.
EXIT.
```

### 5.3.3 Using the RDz tooling to create the Java classes

In the Project Explorer pane of the RDz workspace we selected **ItsoWolaEJBClient** and right-clicked it. We selected **New** and then **Other** and then expanded the J2C folder to view the J2C wizard options; see Figure 5-4 on page 86.

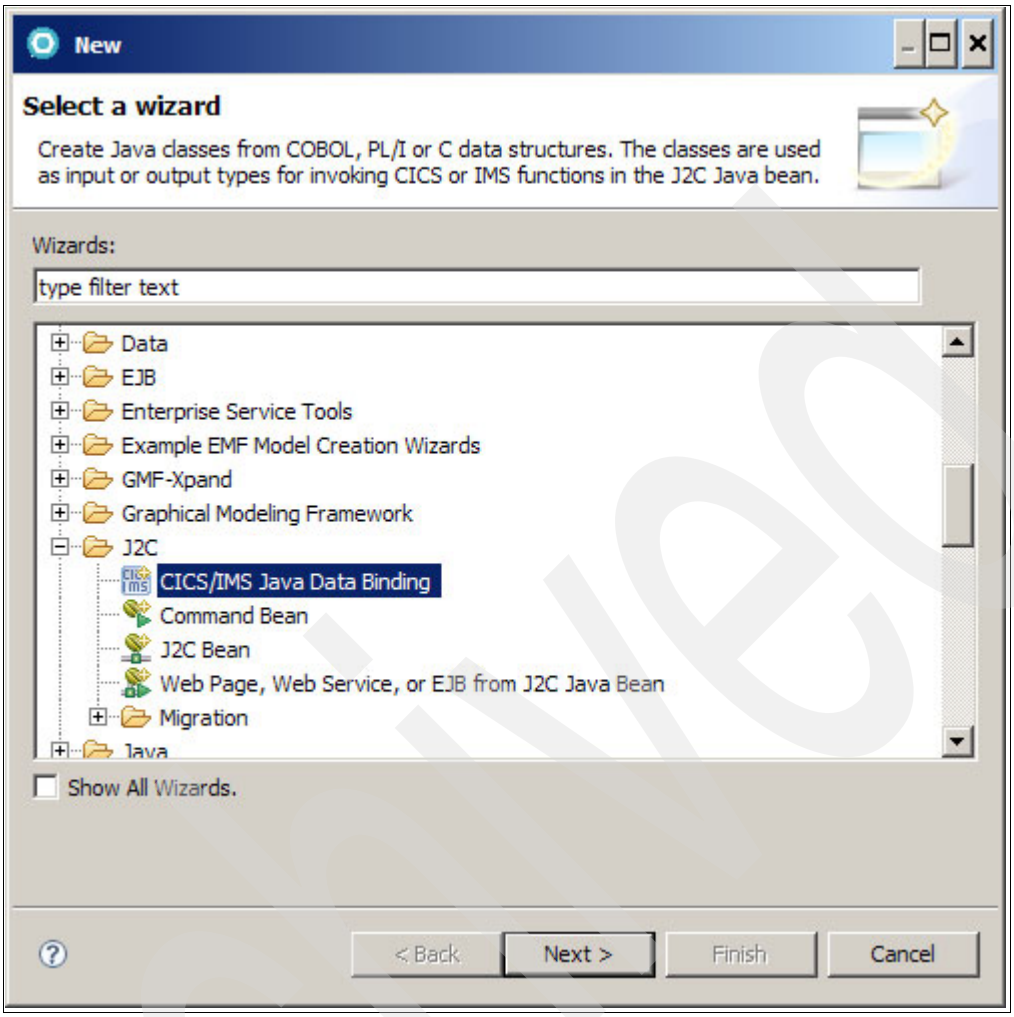



*Figure 5-4 Selecting the CICS/IMS Java Data Binding wizard*

We clicked **Next** to continue and on the Data Import panel we selected **COBOL to Java mapping** and used the Browse button to locate the COBOL source file (see Example 5-1 on page 83). We clicked **Next** to continue.

On the Import panel we selected **z/OS** for the Platform and used the Query button to locate and open the local COBOL source for program CSCVINQ.

**Note:** By selecting z/OS as the target platform the generated Java code would have included the code to convert ASCII to EBCDIC in WebSphere before being sent to CICS and the code to convert the EBCDIC to ASCII on the return. If a CICS conversion program (DFHCNV) and/or the containers were being converted from ASCII to EBCDIC by the CICS program by the GET CONTAINER request we could have specified Windows® as the target platform and the data would have been sent in ASCII to CICS for data conversion.

In general, because of Big and Little Endian conversion issues between EBCDIC and non-EBCDIC platforms, it is probably a good idea to let the Java client perform the data conversions.

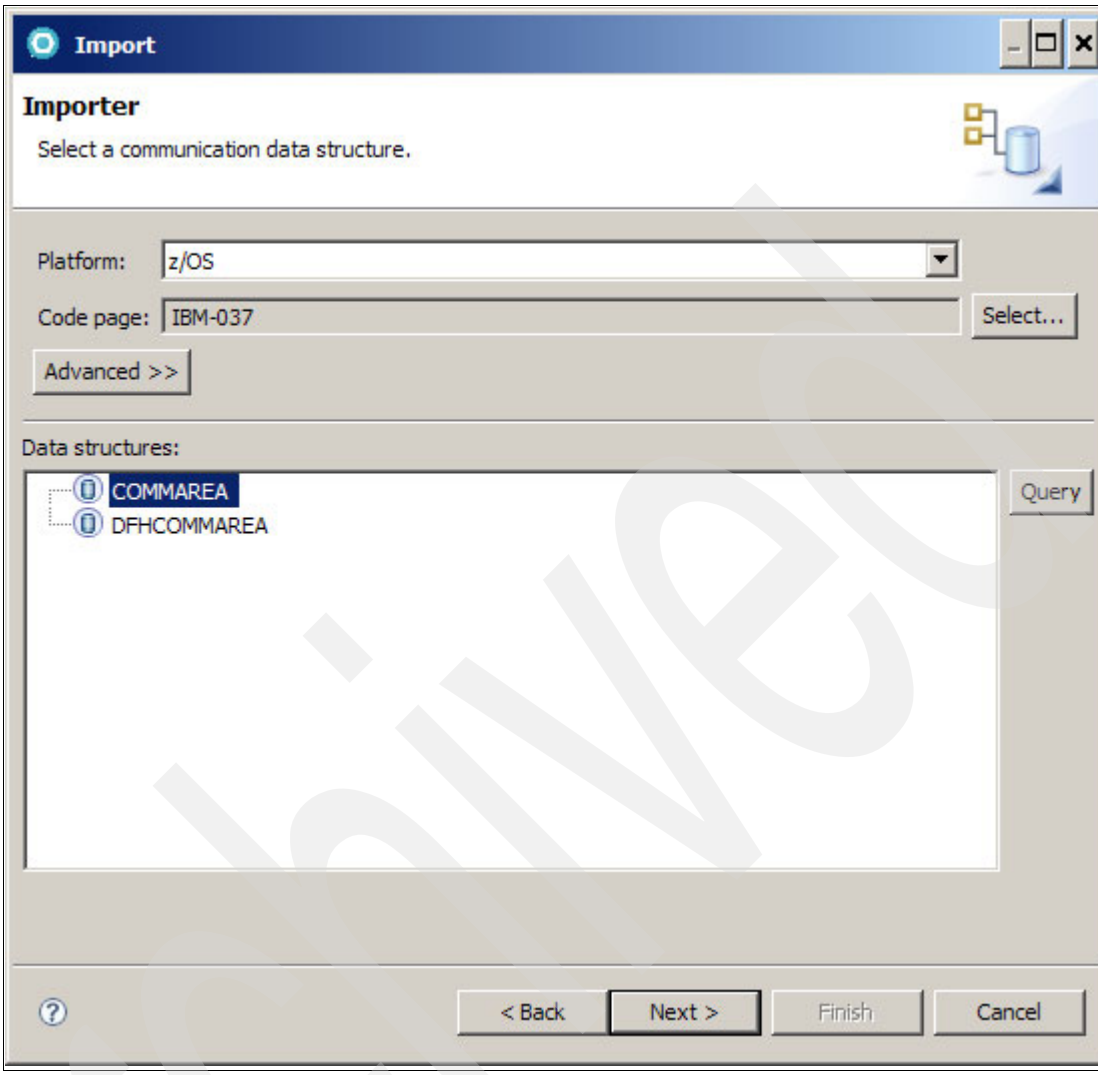



*Figure 5-5 Select COBOL structure*

We selected the **COMMAREA** (see Example 5-1 on page 83) structure and clicked **Next t**o continue.

Note: The DFHCOMMAREA structure in the original COBOL source is defined as a PIC X(120) field. Selecting this structure would not provide direct access to the individual fields in the COMMAREA and therefore no Java getter and setter methods would have been generated for these fields.

On the Saving Properties panel (see Figure 5-6 on page 88) we entered `com.itso.wola.j2c` for the Package Name and `CscvinqCOMMAREA` for the Class Name. We clicked **Finish** to complete the generation of the Java code representation of this COBOL structure.

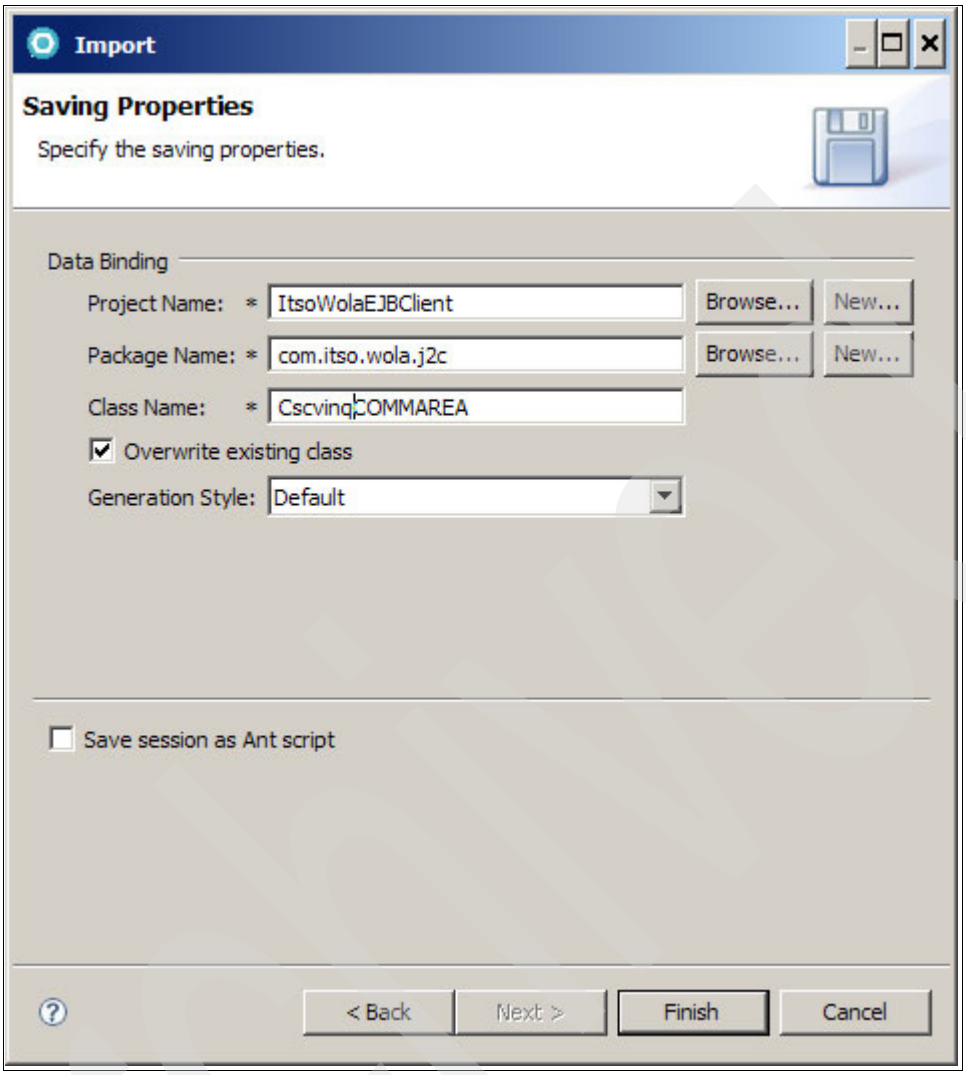



*Figure 5-6 Data Binding information for the Java class*

These steps were repeated to create the Java class CscvincRequestmessage for the structure Request-Message and CscvincResponseMessage for the structure Response-Message in the program CSCVINC (see Example 5-2 on page 84).

## 5.3.4 Developing the WebSphere Optimized Local Adapter CCI client code

Using what we have learned about the WebSphere Optimized Local Adapter CCI classes and the J2C classes created with the RAD J2C wizards we developed the sample code (see Example 5-3 on page 89) to demonstrate how these classes are used together to create a WebSphere Optimized Local Adapter client.

In this example we created an instance of our J2C class and set the employee number (1). We then performed a JNDI lookup to obtain a connection factory (2).

Next we obtained an instance of the WebSphere Optimized Local Adapter ConnectionSpecImpl and used its setter methods to specify the desired WebSphere Optimized Local Adapter connection properties (3). Using the connection factory and the connection specification we obtained a connection from the connection manager (4).

We created an instance of the WebSphere Optimized Local Adapter InteractionSpecImpl and used its setter method to identify the CICS application program (5). We then created a WebSphere Optimized Local Adapter IndexRecordImpl instance and added the J2C class for the COMMAREA as its first element (6).

Next the code (7) created an instance of an Interaction and then used this interaction to execute the WebSphere Optimized Local Adapter interface (8). The response was retrieved from the WebSphere Optimized Local Adapter record (9) and then the J2C class getter methods were used to retrieve the individual fields from the response record.

*Example 5-3 Sample OLA CCI code*

```
com.itso.wola.j2c.CscvinqCommarea commarea = new

com.itso.wola.j2c.CscvinqCommarea();   ---> 1

/* Use the COMMAREA 'setter' method to set the

employee number in the request message */

commarea.setNumb(“111111”);

/* Obtain a connection factory by using an indirect lookup

for the JNDI name */

javax.naming.InitialContext ctx = new javax.naming.InitialContext();

javax.resource.cci.ConnectionFactory connectionFactory =

(javax.resource.cci.ConnectionFactory)

ctx.lookup(“java:comp/env/OLA”);  ---> 2

/* Create an OLA ConnectionSpecImpl instance and

set the WOLA connection properties */

com.ibm.websphere.ola.ConnectionSpecImpl connectionSpec = new

com.ibm.websphere.ola.ConnectionSpecImpl();   ---> 3

connectionSpec.setRegisterName("OLAServer");

connectionSpec.setConnectionWaitTimeout(20);

connectionSpec.setLinkTaskTranID("BBO#");

connectionSpec.setLinkTaskReqContID("");

connectionSpec.setLinkTaskRspContID("");

connectionSpec.setLinkTaskReqContType(0);

connectionSpec.setLinkTaskRspContType(0);

connectionSpec.setUseCICSContainer(false);   /* No CICS containers*/

/* Obtain a CCI Connection using the obtained connection factory
```

```
and the requested WOLA properties */
javax.resource.cci.Connection connection =
connectionFactory.getConnection(connectionSpec); ---> 4
/* Create an WOLA InteractionSpecImpl instance and set the WOLA
service name (CICS program name) */
com.ibm.websphere.ola.InteractionSpecImpl interactionSpecImpl =
new com.ibm.websphere.ola.InteractionSpecImpl();
interactionSpecImpl.setServiceName("CSCVINQ");      ---> 5

/* Create an WOLA IndexRecordImpl instance and add the COMMAREA */
com.ibm.websphere.ola.IndexedRecordImpl indexRecordImpl =
new com.ibm.websphere.ola.IndexedRecordImpl();
indexRecordImpl.add(commarea); /*.getBytes());          ---> 6
/* Create a CCI Interaction instance using the connection */
javax.resource.cci.Interaction interaction =
connection.createInteraction();      ---> 7
/* Invoke the WOLA interface and retrieve results */
javax.resource.cci.Record outputRecord =          ---> 8
interaction.execute(interactionSpecImpl,indexRecordImpl);
/* Retrieve the response from the output record as a byte array */
commarea.setBytes((byte[])       ---> 9
(((com.ibm.websphere.ola.IndexedRecordImpl)
outputRecord).get(0)));
/* Use the COMMAREA's 'getter' methods to access the response */
System.out.println(commarea.getAction());         ---> 10
System.out.println(commarea.getCeibresp());
System.out.println(commarea.getCeibresp2());
System.out.println(commarea.getUserid());
System.out.println(commarea.getStat());
System.out.println(commarea.getNumb());
```

```
System.out.println(commarea.getName());

System.out.println(commarea.getAddrx());

System.out.println(commarea.getPhone());

System.out.println(commarea.getDatex());

System.out.println(commarea.getAmount());

System.out.println(commarea.getComment());
```

## 5.4 Creating a J2EE application

We used the J2C Java classes created earlier (CscvinqCommarea, CscvincRequestMessage, and CscvincResponseMessage) and code from Example 5-3 on page 89 to develop a simple J2EE application (see supplemental material) that consists of JSPs, servlets, and an EJB that acts as a client to the CICS sample programs.

### 5.4.1 J2EE application components

The initial JSP™ collects information and invokes a servlet. The servlet creates either an instance of a COMMAREA or a container J2C class and invokes a method of the EJB passing the WebSphere Optimized Local Adapter parameters in the method signature; see Example 5-4.

*Example 5-4 Snippet of code from the servlet*

```
Context ctx = null;

com.itso.wola.j2c.ContainerResponseMessage containerResponseMessage

= new com.itso.wola.j2c.ContainerResponseMessage();

com.itso.wola.j2c.ContainerRequestMessage containerRequestMessage

= new com.itso.wola.j2c.ContainerRequestMessage();

String regnameString = request.getParameter("reg_name");

String servnameString = request.getParameter("service_name");

String linktransidString = request.getParameter("link_transid");

String reqcontidString = request.getParameter("request_contid");

String reqconttypeString =

request.getParameter("request_conttype_bit");

int reqconttype = 0;

if (reqconttypeString.toUpperCase().equals("YES")) {

reqconttype = 1;}
```

```
String rspcontidString = request.getParameter("response_contid");
String rspconttypeString =
request.getParameter("response_conttype_bit");
int rspconttype = 0;
if (rspconttypeString.toUpperCase().equals("YES")) {
rspconttype = 1;}
boolean usecontainer = false;
if (servnameString.equals("CSCVINC")) {
usecontainer = true;}
String empNumb = request.getParameter("empNumb");
containerRequestMessage.setNumb(empNumb);
ctx = new InitialContext();
WolaHome beanHome = null;
Object o = ctx.lookup("java:comp/env/ejb/Wola");
beanHome = (WolaHome)javax.rmi.PortableRemoteObject.
narrow(o,WolaHome.class);
Wola beanRemote = beanHome.create();
byte [] input = containerRequestMessage.getBytes();
byte [] output = null;
output  = beanRemote.driveIntoCics2(input, regnameString,
servnameString, linktransidString,
reqcontidString, reqconttype, rspcontidString,
rspconttype, usecontainer);
containerResponseMessage.setBytes(output);
ServletContext sc = getServletContext();
request.setAttribute("rspMessage",containerResponseMessage );
RequestDispatcher rd = sc.getRequestDispatcher
("/ContainerResults.jsp");
rd.forward(request, response);
```

**Note:** This code is for reference only and is not complete. To see the complete code, see the original source in the EAR or project interchange file in the supplemental material.

The EJB uses these parameters (see Example 5-5) to create the CCI and WebSphere Optimized Local Adapter class instances and then execute the WebSphere Optimized Local Adapter interface to access the CICS programs and return the results back to the servlet and for displaying a resulting JSP.

*Example 5-5 Snippet of code from the sample EJB*

```
public byte[] driveIntoCics2(byte[] input, String registerName,
String serviceName, String CICSTranID,String ReqContID,
int ReqContType, String RspContID, int RspContType,
boolean UseCICSContainer) {
byte[] output = null;
Connection connection = null;
Interaction interaction = null;
/* Obtain connection factory by doing an indirect JNDI lookup */
InitialContext ctx = new InitialContext();
ConnectionFactory connectionFactory =
(ConnectionFactory)ctx.lookup("java:comp/env/OLA");
/* Create a connectionSpecImpl class instance and initialize with requested OLA properties*/
ConnectionSpecImpl connectionSpecImpl = new ConnectionSpecImpl();
connectionSpecImpl.setRegisterName(registerName);
connectionSpecImpl.setConnectionWaitTimeout(20);
connectionSpecImpl.setLinkTaskTranID(CICSTranID);
connectionSpecImpl.setLinkTaskReqContID(ReqContID);
connectionSpecImpl.setLinkTaskRspContID(RspContID);
connectionSpecImpl.setLinkTaskReqContType(ReqContType);
connectionSpecImpl.setLinkTaskRspContType(RspContType);
connectionSpecImpl.setUseCICSContainer(UseCICSContainer);
connection = connectionFactory.getConnection(connectionSpecImpl);
/* Create interactionSepcImpl and set the OLA service name */
InteractionSpecImpl interactionSpecImpl = new InteractionSpecImpl();
```

```
interactionSpecImpl.setServiceName(serviceName);

/* Create an interaction */

interaction = connection.createInteraction();

/* Create an IndexRecordImpl instance and add the OLA message */

IndexedRecordImpl indexRecordImpl = new IndexedRecordImpl();

indexRecordImpl.add(input);

/* Invoke the OLA interface */

Record outputRecord =

interaction.execute(interactionSpecImpl ,indexRecordImpl);

output = (byte[])(((IndexedRecordImpl)outputRecord).get(0));

}
```

**Note:** This code is for reference use only and is not complete. To see the complete code see the original source in the EAR or project interchange file in the supplemental material.

### 5.4.2 EJB Deployment Descriptor Resource Reference

Since we are doing an indirect lookup we added a Resource reference for the name (OLA) used in the indirect lookup to the EJB Deployment Descriptor (see Figure 5-7) to match the java:coomp lookup done in the EJB; see Example 5-5 on page 93.

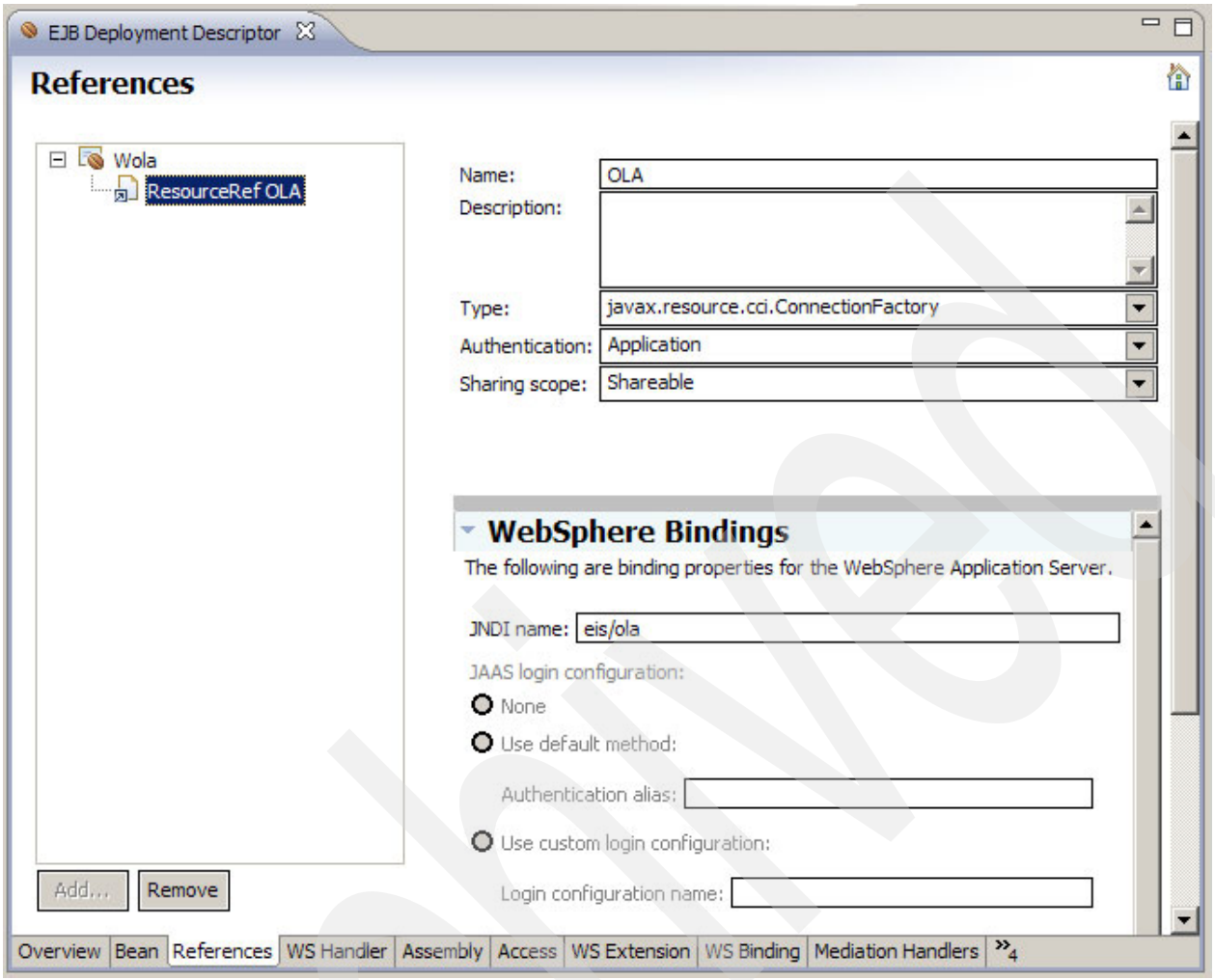



*Figure 5-7 Resource Reference for the indirect lookup of the OLA J2C Connection Factory*

### 5.4.3 Deploying the application

During the deployment of the application our resource reference for JNDI, eis/ola, was mapped to the J2C Connection factory added to the configuration when the olaRar.py Jython script was executed.

## 5.5 Configuring the WebSphere Optimized Local Adapter CICS link server

We activated the WebSphere Optimized Local Adapter CICS link server in the target CICS region so the sample applications could be accessed from our sample WebSphere application.

### 5.5.1 CICS region updates

Starting the WebSphere Optimized Local Adapter CICS link server was automated by adding program BBOACPL2 to the PLT list used at startup (see Example 5-6).

*Example 5-6 CICS Initial Program List Table (PLTPI-PI)*

```
         DFHPLT TYPE=INITIAL,SUFFIX=PI
*  THE FOLLOWING PROGRAMS ARE RUN IN THE FIRST PASS OF PLTPI
         DFHPLT TYPE=ENTRY,PROGRAM=DFHDELIM
*  THE FOLLOWING PROGRAMS ARE RUN IN THE SECOND PASS OF PLTPI
         DFHPLT TYPE=ENTRY,PROGRAM=BBOACPLT
         DFHPLT TYPE=ENTRY,PROGRAM=BBOACPL2
         DFHPLT TYPE=FINAL
         END
```

**Note:** Program BBOACPLT in the initial program list table automatically starts the WebSphere Optimized Local Adapter task-related user exit (TRUE) at CICS startup.

The INITPARM parameter, shown in Example 5-7, was added to the SYSIN data stream used to provide CICS SIT initialization option overrides. This INITPARM was passed to program BBOCPL2, which started the CICS WebSphere Optimized Local Adapter link server task. The link server registered with the WebSphere daemon group WPCELL (daemon group short name), on node WPNODEC (node short name) for server WPS01C (server short name) with a registration name of CSCV. Once registered, the CICS WebSphere Optimized Local Adapter link server task would now receive and process all requests for all WebSphere Optimized Local Adapter outbound requests (SVC=”*’) in WebSphere that specified a register name of CSCV.

*Example 5-7 CICS INITPARM SIT override*

```
INITPARM=(BBOACPL2='STA RGN=CSCV SVN=WPS01C DGN=WPCELL NDN=WPNODEC SVC=*')
```

**Note:** If the WebSphere daemon is not active when CICS is started or is recycled without restarting CICS, the WebSphere Optimized Local Adapter CICS link server must be stopped with transaction BBOC STOP_SRVR RGN=CSCV and restarted with transaction BBOC START_SRVR RGN=CSCV SVN=WPS01C DGN=WPCELL NDN=WPNODEC SVC=*

Only one INITPARM for program BBOACPl2 is allowed during CICS startup. The parameter string passed to program BBOACPL2 is limited to 60 characters and no continuation across multiple lines is possible. There are other link server parameters which could not be entered on an INITPARM request, for example the WebSphere Optimized Local Adapter trace level (TRC, which defaults to 0), an alternative for the link server transaction ID (STX, which defaults to BBO$), an alternative for the link transaction ID (LTX, which defaults to BBO#), and so on.

**Note:** More information about these and other parameters can be found at:

`http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/index.jsp?topic=/com.ibm.websphere.nd.multiplatform.doc/info/ae/ae/rdat_cics.html`

## 5.6 Running the sample application

The J2EE sample application (provided in the supplemental material) consists of two data entry JSP pages, two servlets, one EJB, and two response JSPs. One data entry JSP collects information required to invoke the COMMAREA CICS program (CSCVINQ)) and the other data entry JSP collects information required for the Container CICS program (CSCVINC).

As described earlier, these two programs (CSCVINQ and CSCVINC) perform the same program logic. Both will either retrieve a record (S), update a record (U), delete a record from (D), or insert a record (I) into the CICS-supplied sample FILEA VSAM file, depending on the Action field in the COMMAREA or Container that is passed to them. The default action is to select a record using the provided employee number and returning the details in the results. See *CICS/ESA Sample Application Guide,* SC33-1173-01 for additional information about the CICS sample applications and the FILEA VSAM file on which these programs were based.

### 5.6.1 The CICS COMMAREA application

To access the initial JSP for the COMMAREA CICS application we entered:

```
http://wtsc04.itso.ibm.com:12267/ItsoWolaWeb/CICSCommarea.jsp
```

This opened the browser panel shown in Figure 5-8 on page 98.

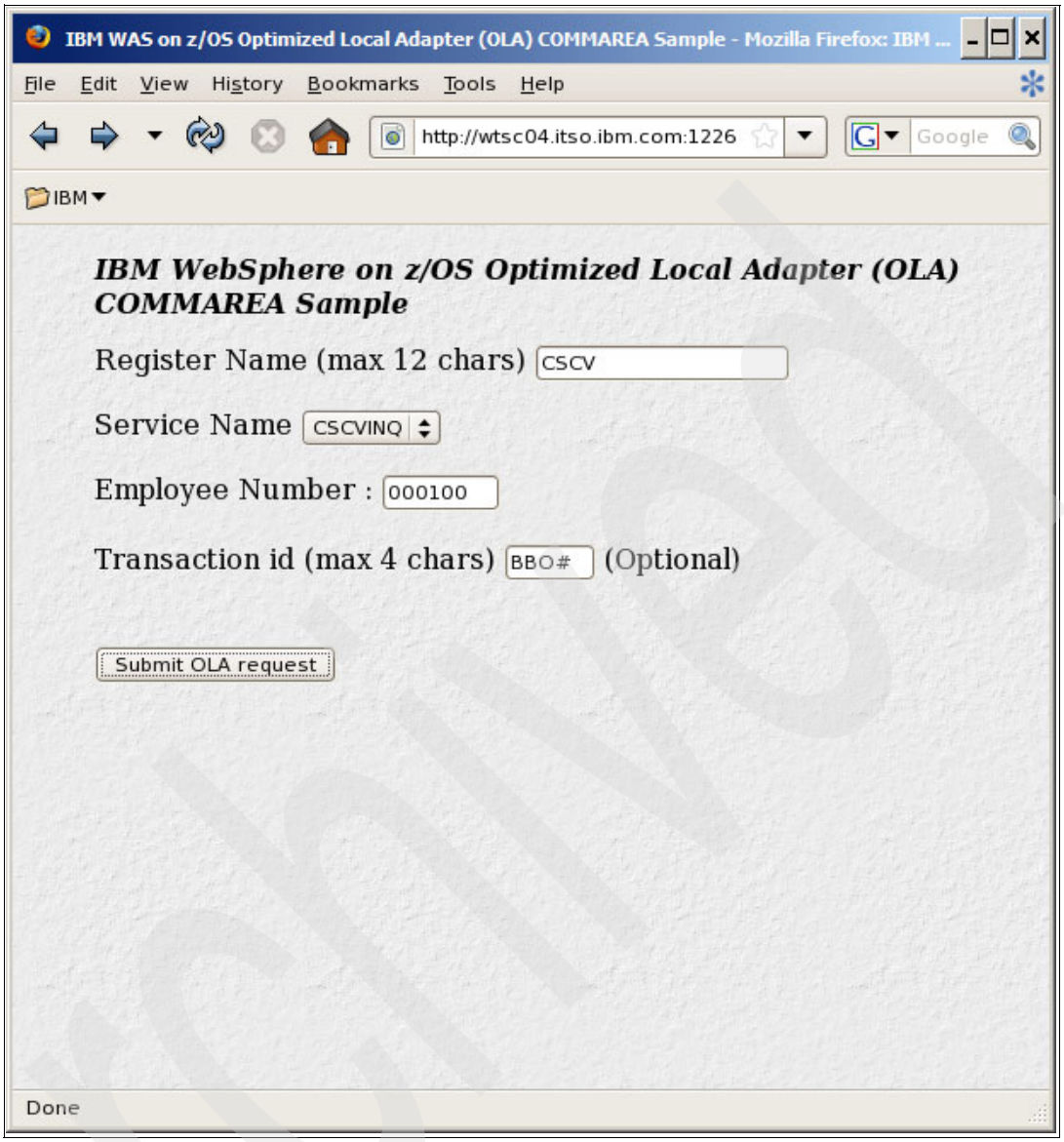



*Figure 5-8 CICS COMMAREA entry*

We selected the register name used when the CICS region registered with the daemon (see Example 5-7 on page 96) and specified the service name to be the name of the CICS program to be executed, for example CSCVINQ. After entering an employee number, we clicked **Submit OLA Request** and the WOLA CICS COMMAREA Results panel was displayed (see Figure 5-9 on page 99) with details that indicated that we had successfully accessed the CICS application and read the VSAM file to obtain the information about this employee.

**Note:** We used the CICS execution diagnostic facility transaction CEDX to monitor the flow of our requests in CICS. We started a 3270 session and entered transaction CEDX BBO# (the default LinkTranID for a WebSphere Optimized Local Adapter J2C connection).

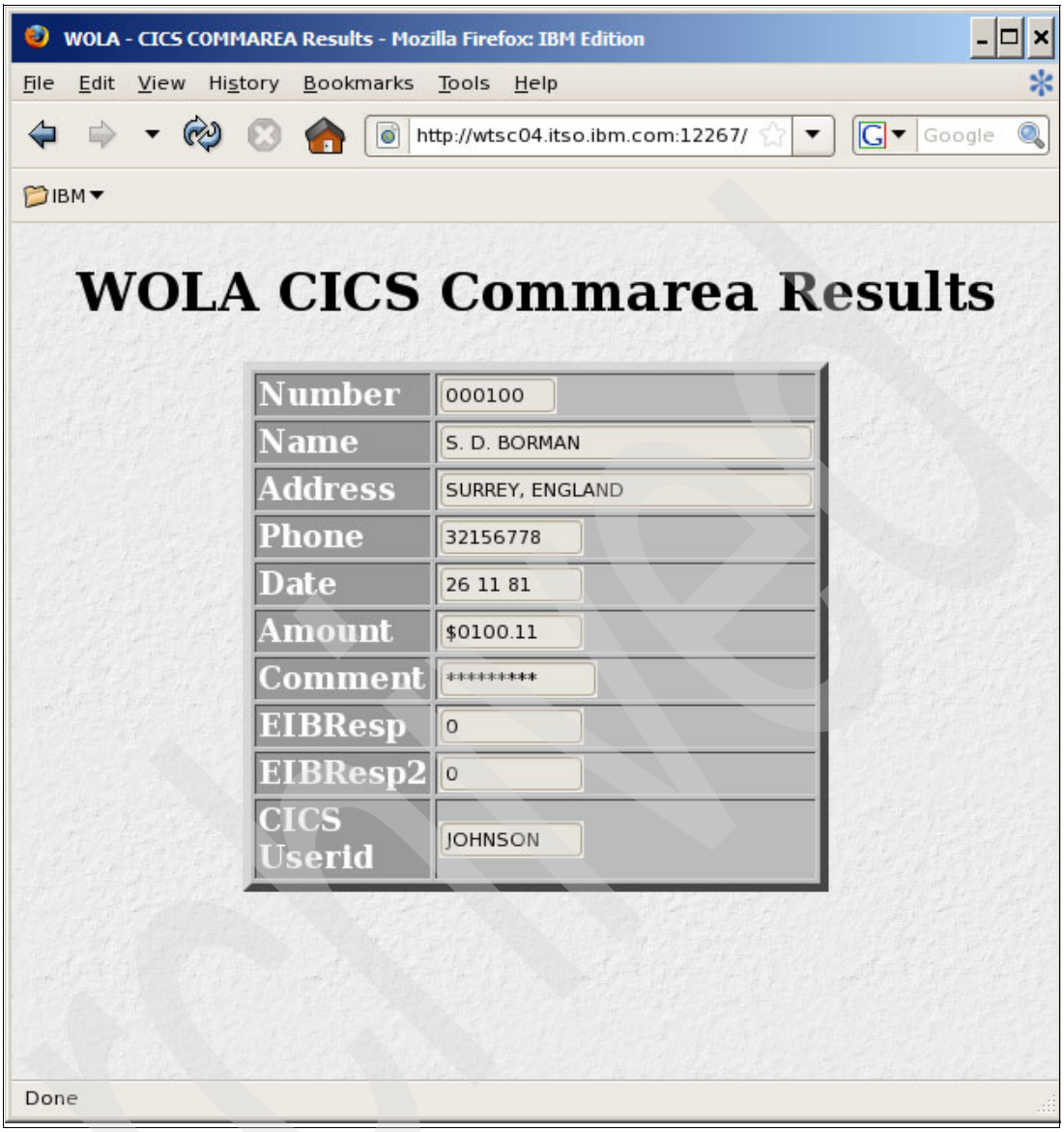



*Figure 5-9 CICS COMMAREA results panel*

**Note:** If the employee number had not been found in the VSAM file, the EIBResp field would have been 13 and EIBResp2 would have been 80. These EIB response codes indicate that a NOTFOUND condition had been raised on the CICS EXEC READ FILE command.

## 5.6.2 .The CICS Container application

To access the initial JSP for the COMMAREA CICS application, we entered:

```
http://wtsc04.itso.ibm.com:12267/ItsoWolaWeb/CICSContainer.jsp
```

This opened the browser panel shown in Figure 5-10 on page 100.

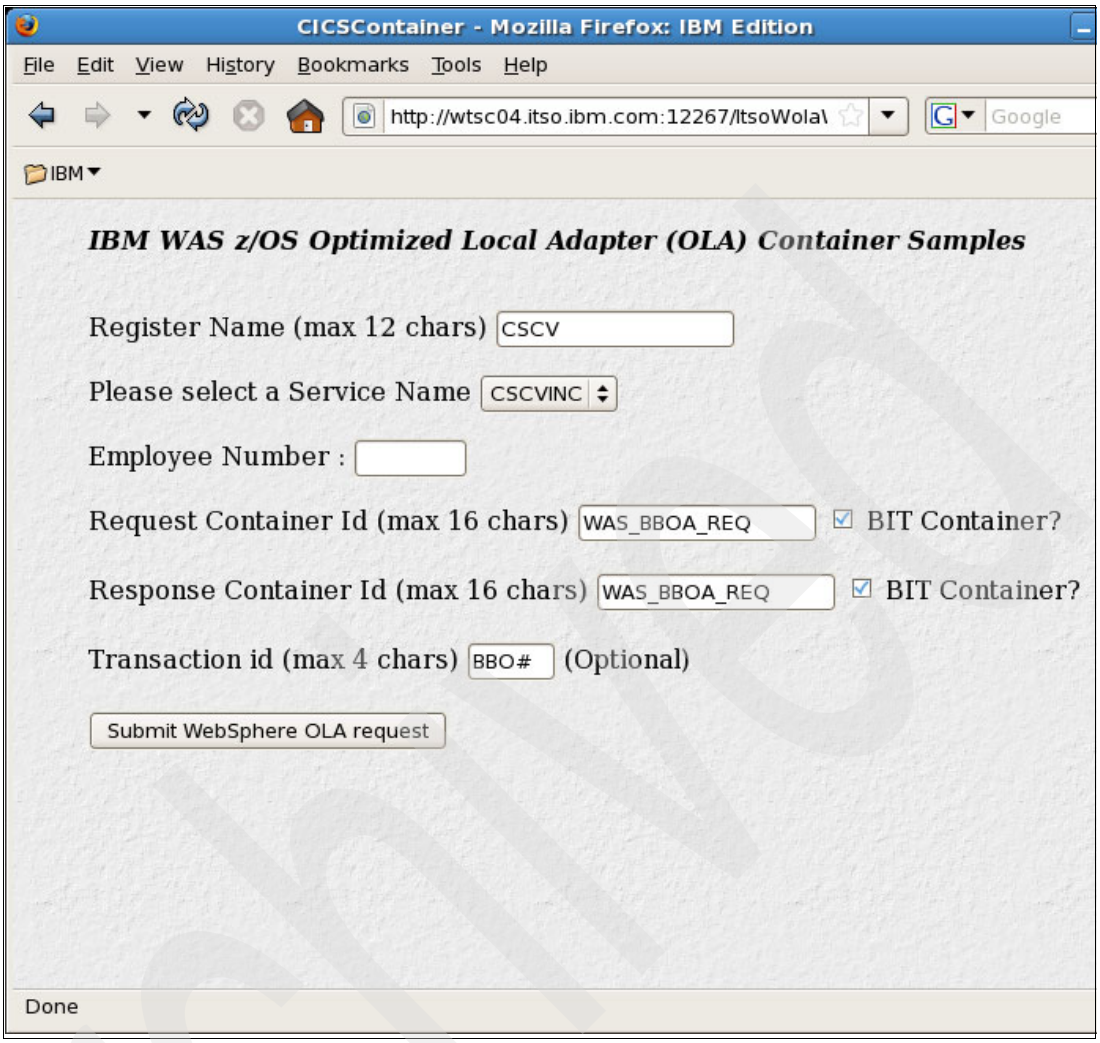



*Figure 5-10 CICS Container entry*

We selected the register name used by CICS when the CICS region registered with the daemon (see Example 5-11 on page 101) and specified the service name to be the CICS program to be executed, for example CSCVINC. We took the defaults for the request and response container names and the CICS link transaction. After entering an employee number we pressed **Submit OLA Request** and the WOLA CICS CONTAINER Results panel (see Figure 5-11 on page 101) was displayed with results that indicated that we had successfully accessed the CICS application and read a VSAM file to obtain the details about this employee.

**Note:** If we had not checked the BIT boxes WebSphere Optimized Local Adapter would have sent the containers in CHAR (or character data format) and CICS would have converted the contents from ASCII to EBCDIC. But since our J2C classes were performing the conversion for us already, CICS would actually be converting EBCDIC to EBCDIC and producing incorrect data. Sending the containers in BIT format prevents this unnecessary conversion.

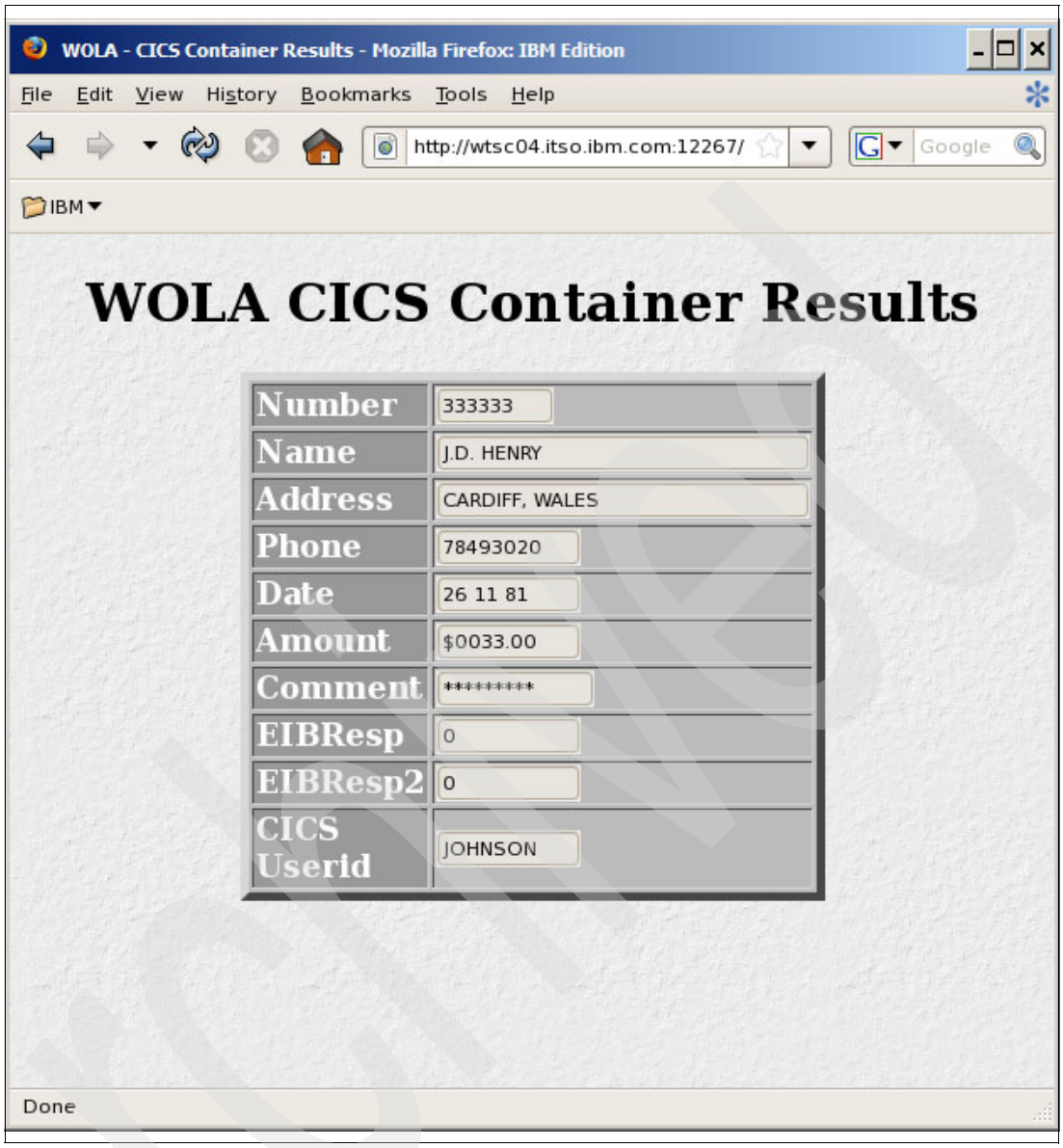



*Figure 5-11 CICS Container results*

A

# Additional material

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

## Locating the Web material

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

ftp://www.redbooks.ibm.com/redbooks/REDP4550

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 IBM Redpaper form number, REDP4550.

## Using the Web material

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

| *File name* | *Description* |
|---|---|
| **WolaEnabledTrade6.zip** | Zipped Code Samples described in Chapter 4 |
| **wola-ch4-add-Materials.zip** | Zipped Code Samples described in Chapter 4 |
| **wola-ch5-add-Materials.zip** | Zipped Code Samples described in Chapter 5 |

### How to use the Web material

Create a subdirectory (folder) on your workstation, and unzip the contents of the Web material zip file into this folder. Open the readme file. This file contains the description of the accompanying files and the instructions for their use.

# Related publications

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

## IBM Redbooks

For information about ordering these publications, see “How to get Redbooks” on page 105. Note that some of the documents referenced here may be available in softcopy only.

- *WebSphere for z/OS V6 Connectivity Handbook,* SG24-7064
- *Rational Application Developer V7.5 Programming Guide,* SG24-7672
- *Developing Connector Applications for CICS*, SG24-7714

## Online resources

These Web sites are also relevant as further information sources:

- A Brief Introduction to Optimized Local Adapters

  http://www-03.ibm.com/support/techdocs/atsmastr.nsf/WebIndex/WP101490
- WebSphere Application Server, Version 7.0 Information Center

  http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/index.jsp
- The future of SOA on z/OS built on a smarter foundation of CICS TS V4, WAS V7 and Rational Developer for System z

  ftp://ftp.boulder.ibm.com/software/systemz/pdf/telecon/Apr_8_Telecon_-_The_future_on_SOA_on_zOS_Built_on_a_Smarter_Foundation.pdf

## How to get Redbooks

You can search for, view, or download Redbooks, Redpapers, Technotes, draft publications and Additional materials, as well as order hardcopy Redbooks publications, at this Web site:

ibm.com/redbooks

## Help from IBM

IBM Support and downloads

ibm.com/support

IBM Global Services

ibm.com/services

# WebSphere on z/OS - Optimized Local Adapters


**Exploit high-performance bi-directional communications between WebSphere and CICS applications**

**Installation and configuration examples**

**Sample application scenarios**

This IBM Redpaper publication describes the steps involved in the installation, configuration and implementation of the new Optimized Local Adapters (OLA) support available with WebSphere Application Server.
A step-by-step approach is used to guide you through the OLA installation and configuration process.
The OLA bi-directional communications functions are presented in detail through the development, deployment and execution phases using three sample application scenarios:

- Modify the existing IBM benchmark application Trade6 to enable it to receive OLA inbound calls from an external application written in C.
- A CICS Cobol program modified to invoke an EJB within the WebSphere Application Server on z/OS.
- An EJB in the WebSphere Application Server on z/OS invoking a Cobol program in CICS.

**INTERNATIONAL TECHNICAL SUPPORT ORGANIZATION**

**BUILDING TECHNICAL INFORMATION BASED ON PRACTICAL EXPERIENCE**

IBM Redbooks are developed by the IBM International Technical Support Organization. Experts from IBM, Customers and Partners from around the world create timely technical information based on realistic scenarios. Specific recommendations are provided to help you implement IT solutions more effectively in your environment.

**For more information:**
**ibm.com**/redbooks

REDP-4550-00