BEA WebLogic Java Adapter for Mainframe

Scenarios Guide

BEA WebLogic Java Adapter for Mainframe 4.2
Document Edition 4.2
July 2001

Copyright

Copyright © 2001 BEA Systems, Inc. All Rights Reserved.

Restricted Rights Legend

This software and documentation is subject to and made available only pursuant to the terms of the BEA Systems License Agreement and may be used or copied only in accordance with the terms of that agreement. It is against the law to copy the software except as specifically allowed in the agreement. This document may not, in whole or in part, be copied photocopied, reproduced, translated, or reduced to any electronic medium or machine readable form without prior consent, in writing, from BEA Systems, Inc.

Use, duplication or disclosure by the U.S. Government is subject to restrictions set forth in the BEA Systems License Agreement and in subparagraph (c)(1) of the Commercial Computer Software-Restricted Rights Clause at FAR 52.227-19; subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013, subparagraph (d) of the Commercial Computer Software--Licensing clause at NASA FAR supplement 16-52.227-86; or their equivalent.

Information in this document is subject to change without notice and does not represent a commitment on the part of BEA Systems. THE SOFTWARE AND DOCUMENTATION ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND INCLUDING WITHOUT LIMITATION, ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. FURTHER, BEA Systems DOES NOT WARRANT, GUARANTEE, OR MAKE ANY REPRESENTATIONS REGARDING THE USE, OR THE RESULTS OF THE USE, OF THE SOFTWARE OR WRITTEN MATERIAL IN TERMS OF CORRECTNESS, ACCURACY, RELIABILITY, OR OTHERWISE.

Trademarks or Service Marks

BEA, WebLogic, Tuxedo, and Jolt are registered trademarks of BEA Systems, Inc. How Business Becomes E-Business, Operating System for the Internet, Liquid Data, BEA WebLogic E-Business Platform, BEA Builder, BEA Manager, BEA eLink, BEA Campaign Manager for WebLogic, BEA WebLogic Commerce Server, BEA WebLogic Personalization Server, BEA WebLogic Process Integrator, BEA WebLogic Collaborate, BEA WebLogic Enterprise, BEA WebLogic Server, and BEA WebLogic Integration are trademarks of BEA Systems, Inc.

All other trademarks are the property of their respective company.

BEA WebLogic Java Adapter for Mainframe Scenarios Guide

Document Edition	Part Number	Date	Software Version
4.2	N/A	July 2001	BEA WebLogic Java Adapter for Mainframe 4.2

Contents

4. Web-enabling an IBM 3270 Application

5. Using JAM in a Clustered Environment

About This Document

The BEA WebLogic Java Adapter for Mainframe product (hereafter referred to as JAM) is a gateway connectivity application that enables client/server transactions between Java applications and OS/390 or IMS programs.

This document describes the following scenarios for how you might use the JAM product:

- "Developing a Multi-Service Data Entry Servlet" provides a scenario that illustrates how to develop a multi-service application for WebLogic Server.
- "Enhancing an Existing Servlet to Originate a Mainframe Request" provides a scenario that illustrates how to enhance an existing servlet to originate a mainframe request using WebLogic Server.
- "Updating an Existing EJB to Service a Mainframe Request" provides a scenario that shows how to update and existing EJB to service a request from the mainframe.
- "Web-enabling an IBM 3270 Application" provides a scenario that shows how to develop a single service servlet-based application that invokes a CrossPlex script on the mainframe when you are using WebLogic Server.
- "Using JAM in a Clustered Environment" provides a scenario that extends the base EJB client sample to demonstrate a client requesting multiple employee actions against an EJB that is deployed in a clustered environment.

What You Need to Know

This document is intended for system administrators, application programmers, and business analysts who will use the BEA WebLogic Java Adapter for Mainframe application.

e-docs Web Site

BEA product documentation is available on the BEA corporate Web site. From the BEA Home page, click on Product Documentation or go directly to the "e-docs" Product Documentation page at **http://edocs.bea.com/**.

How to Print the Document

A PDF version of this document is available on the JAM documentation Home page on the e-docs Web site (and also on the installation CD). You can open the PDF in Adobe Acrobat Reader and print the entire document (or a portion of it) in book format. To access the PDFs, open the JAM documentation Home page, click the PDF files button, and select the document you want to print.

If you do not have the Adobe Acrobat Reader, you can get it for free from the Adobe Web site at **http://www.adobe.com/**.

Related Information

The following BEA publications are available for JAM 4.2:

- *BEA WebLogic Java Adapter for Mainframe Release Notes*

- *BEA WebLogic Java Adapter for Mainframe Introduction*
- *BEA WebLogic Java Adapter for Mainframe Installation Guide*
- *BEA WebLogic Java Adapter for Mainframe Configuration and Administration Guide*
- *BEA WebLogic Java Adapter for Mainframe Programming Guide*
- *BEA WebLogic Java Adapter for Mainframe Scenarios Guide*
- *BEA WebLogic Java Adapter for Mainframe Workflow Processing Guide*
- *BEA WebLogic Java Adapter for Mainframe Reference Guide*

Contact Us

Your feedback on the BEA WebLogic Java Adapter for Mainframe documentation is important to us. Send us e-mail at **docsupport@bea.com** if you have questions or comments. Your comments will be reviewed directly by the BEA professionals who create and update the JAM documentation.

In your e-mail message, please indicate that you are using the documentation for the BEA WebLogic Java Adapter for Mainframe 4.2 release.

If you have any questions about this version of JAM, or if you have problems installing and running JAM, contact BEA Customer Support through BEA WebSupport at **www.bea.com**. You can also contact Customer Support by using the contact information provided on the Customer Support Card that is included in the product package.

When contacting Customer Support, be prepared to provide the following information:

- Your name, e-mail address, phone number, and fax number
- Your company name and company address
- Your machine type and authorization codes
- The name and version of the product you are using
- A description of the problem and the content of pertinent error messages

Documentation Conventions

The following documentation conventions are used throughout this document.

Convention	Item
blue text	Indicates a hypertext link in PDF or HTML
italics	Indicates emphasis or book titles or variables.
`"string with quotes"`	Indicates a string entry that requires quote marks.
UPPERCASE TEXT	Indicates generic file names, device names, environment variables, and logical operators. *Examples*: LPT1 SIGNON OR

`monospace text`	Indicates code samples, commands and their options, data structures and their members, data types, directories, and file names and their extensions. Monospace text also indicates text that you must enter from the keyboard.
	Examples:
	`#include <iostream.h> void main ( ) the pointer psz`
	`chmod u+w *`
	`\tux\data\ap`
	`.doc`
	`tux.doc`
	`BITMAP`
	`float`

`monospace boldface text`	Identifies significant words in code. *Example*: `void` **`xa_commit`** `( )`
`{ }`	Indicates a set of choices in a syntax line. The braces themselves should never be typed.

Convention	Item

`[ ]`	Indicates optional items in a syntax line. The brackets themselves should never be typed.
	Example:
	`buildclient [-v] [-o name ] [-f file-list]...`
	`[-l file-list]...`

`|`	Separates mutually exclusive choices in a syntax line. The symbol itself should never be typed.

`...`	Indicates one of the following in a command line:
	■ That an argument can be repeated several times in a command line
	■ That the statement omits additional optional arguments
	■ That you can enter additional parameters, values, or other information
	The ellipsis itself should never be typed.
	Example:
	`buildclient [-v] [-o name ] [-f file-list]...`
	`[-l file-list]...`

. . .	Indicates the omission of items from a code example or from a syntax line. The vertical ellipsis itself should never be typed.

1 Developing a Multi-Service Data Entry Servlet

This section contains a scenario that shows how to develop a multi-service application for WebLogic Server. The concepts presented for the servlet-only application model described in the *BEA WebLogic Java Adapter for Mainframe Programming Guide* are used and extended for this scenario.

In this scenario, a new application is developed and existing applications are updated. WebLogic Server samples are used to illustrate any existing applications. All discussions are from the application developer's point of view.

Note: Although the sample code in this section represents typical applications, it is intended for example only and is not supported for actual use.

Action List

The following table lists the tasks that must be completed to develop a multi-service data entry servlet.

	Your action...	Refer to...
1	Verify that prerequisite tasks have been completed.	"Prerequisites"

	Your action...	Refer to...
2	Use eGen COBOL Code Generator to generate an application.	"Task 1: Use the eGen COBOL Code Generator to Generate an Application"
3	Create your custom application from the generated application.	"Task 2: Create Your Custom Application from the Generated Application"
4	Update the JAM configurations and update WebLogic Server configuration.	"Task 3: Update the JAM Configurations and Update BEA WebLogic Server web.xml File"
5	Deploy your application.	"Task 4: Deploy Your Application"
6	Use the application.	"Task 5: Use the Application"

The following example creates a servlet that invokes the sample COBOL programs described at the end of this chapter.

Prerequisites

Before you begin to develop a multi-service data entry servlet, ensure that the following prerequisites have been completed.

	Your action...	Refer to...
1	Verify that the required software has been properly installed: WebLogic Server, WebLogic Java Adapter for Mainframe.	BEA WebLogic Server documentation, *BEA WebLogic Java Adapter for Mainframe Installation Guide*
2	Verify that the environment and the software components have been properly configured.	BEA WebLogic Server documentation, *BEA WebLogic Java Adapter for Mainframe Configuration and Administration Guide*.
3	Verify the appropriate mainframe application is available.	Your mainframe system administrator.

	Your action...	Refer to...
4	Review the steps to develop a single service application.	*BEA WebLogic Java Adapter for Mainframe Programming Guide*

Task 1: Use the eGen COBOL Code Generator to Generate an Application

Identify the mainframe application and obtain its COBOL copybook, usually a CICS `DFHCOMAREA` or the user data portion of an IMS queue record layout. The copybook's name in this discussion is `emprec.cbl`, as shown in Listing 1-1.

Listing 1-1 Mainframe Application COBOL Copybook `emprec.cbl`

```
02  emp-record.
       05  emp-ssn                   pic 9(9) comp-3.
       05  emp-name.
             10  emp-name-last       pic x(15).
             10  emp-name-first      pic x(15).
             10  emp-name-mi      pic x.
     05  emp-addr.
             10  emp-addr-street  pic x(30).
             10  emp-addr-st      pic x(2).
              10  emp-addr-zip       pic x(9).
```

Step 1: Prepare eGen COBOL Script

The script shown in Listing 1-2 generates the `emprecData` DataView from the copybook named `emprec.cbl`.

Listing 1-2 Basic eGen script

```
view empRecData from emprec.cbl
```

Step 2: Add Service Entries

Add the single line service entries in Listing 1-3 for create, read, update, and delete operations. They all use `empRecData` as input and return `emprecData` as output. In this example, a single DataView is used; however, the input and output DataViews could be different.

Listing 1-3 Service Names Associated with Input and Output Views

```
service empRecCreate accepts empRecData returns empRecData
service empRecRead accepts empRecData returns empRecData
service empRecUpdate accepts empRecData returns empRecData
service empRecDelete accepts empRecData returns empRecData
```

Step 3: Add Page Declaration in eGen COBOL Script

Multiple pages can be chained together. Any service entries should match services defined elsewhere in the script. The page declarations shown in Listing 1-4 associate buttons on the HTML display with services declared in the previous step.

Listing 1-4 Page Declaration Associating Display Buttons with Services

```
page empRecPage "Employee Record”
{
  view empRecData
     buttons
     {
              "Create" service(empRecCreate) shows empRecPage
              "Read" service(empRecRead) shows empRecPage
              "Update" service(empRecUpdate) shows empRecPage
              "Delete" service(empRecDelete) shows empRecPage
     }
}
```

Step 4: Add Servlet Name

As shown in Listing 1-5, `empRecServlet` is the servlet name to be registered at a URL in the WebLogic Server `web.xml` file. (Every servlet requires a URL to be registered this way. Refer to WebLogic Server documentation about deploying servlets for more specific information.) Here, the `empRecPage` is to be displayed when the `empRecServlet` is invoked.

Listing 1-5 Add Servlet Name

```
servlet empRecServlet shows empRecPage
```

The script is saved as `emprec.egen`.

Step 5: Generate the Java Source Code

Use the eGen COBOL Code Generator to generate the application as shown in Listing 1-6. These classes will be extended in Task 2 to customize the servlet. The `empRecData.java` is the DataView object for `emprec.cbl`.

Warning: `CLASSPATH` should include the WebLogic Server `.jar` files and the `jam.jar` file; otherwise, the compile fails.

Note: You can create a script file containing the eGen COBOL command line, along with the `javac` command to make the invocation easier.

Listing 1-6 Generating the Java Source Code

```
$egencobol emprec.egen
$ls  emp*.java
     empRecData.java
     empRecServlet.java

$javac emp*.java
```

Step 6: Review the Java Source Code

Obtain a list of accessors for use later. Look at the `eGen COBOL` output to become familiar with each of the scenarios presented in this section.

The entire method of customizing the generated output is predicated on derivation from generated code. The application can be regenerated without destroying the custom code.

Note: Each COBOL group item has its own accessor. This is important because the group name represents a nested inner class that must be accessed in order to retrieve the members.

In the Listing 1-7, the output from the `grep` command shows the relationships in reverse order, for example:

```
getEmpRecord().getEmpAddr().getEmpAddrSt()
```

This relationship is illustrated in the actual code example shown in Listing 1-7.

Listing 1-7 Review the Java Source Code

```
$grep get emp*.java
  empRecData.java:    public BigDecimal   getEmpSsn()
  empRecData.java:    public String       getEmpNameLast()
  empRecData.java:    public String       getEmpNameFirst()
  empRecData.java:    public String       getEmpNameMi()
  empRecData.java:    public EmpNameV     getEmpName()
  empRecData.java:    public String       getEmpAddrStreet()
  empRecData.java:    public String       getEmpAddrSt()
  empRecData.java:    public String       getEmpAddrZip()
  empRecData.java:    public EmpAddrV     getEmpAddr()
  empRecData.java:    public EmpRecordV   getEmpRecord()
```

Task 2: Create Your Custom Application from the Generated Application

The preferred customization method is to derive a custom class from the generated application.

Step 1: Start with Imports

In Listing 1-8, `BigDecimal` supports `COMP-3` packed data. `HttpSession` is available for saving limited state. `DataView` is the base for `emprecData`. The `empRecData` and `empRecServlet` were generated from the COBOL copybook.

Listing 1-8 Using Imports to Start Creating the Custom Application

```
import java.math.BigDecimal;
import javax.servlet.http.HttpSession;
import bea.dmd.dataview.DataView;
import empRecData;
import empRecServlet;
```

Step 2: Declare the New Custom Class

Listing 1-9 shows how to extend the generated servlet. This method enables regeneration of the application without destroying customized code. Fields can be added to the copybook without disrupting the customized code.

Listing 1-9 Declaring the New Custom Class

```
public class customCrud
       extends empRecServlet
{
:
:
```

Step 3: Add Implementation for doGetSetup

Listing 1-10 demonstrates how to provide a new DataView and the `http` session. The `HttpSession(s)` can be used to hold a reference to the DataView, ensuring that you are actually in the first pass rather than a browser back arrow. The DataView provided (`dv`) is a fresh instance of the `empRecData` DataView. Refer to the *BEA WebLogic Java Adapter for Mainframe Programming Guide* for more information on `doGetSetup`.

Listing 1-10 Add Implementation for `doGetSetup`

```
public DataView doGetSetup(DataView dv, HttpSession s){
empRecData erd = (empRecData)s.getValue("customCrud");
if (erd == null)
   erd = (empRecData)dv; // use new dataview
```

In Listing 1-11, note the use of group level accessors to obtain fields. This code pre-fills fields with data entry hints as to which fields are required or how numeric values should be entered. You can fill form data in any manner required prior to displaying the fields.

Listing 1-11 Continue Implementation for `doGetSetup`

```
if(erd.getEmpRecord().getEmpSsn().compareTo(BigDecimal.valueOf(0L)) == 0)
     erd.getEmpRecord().setEmpSsn(BigDecimal.valueOf(123121234L));

if (erd.getEmpRecord().getEmpName().getEmpNameLast().length() == 0)
     erd.getEmpRecord().getEmpName().setEmpNameLast("Entry Required");

if (erd.getEmpRecord().getEmpName().getEmpNameFirst().trim().length() == 0)
     erd.getEmpRecord().getEmpName().setEmpNameFirst("Entry Required");

if (erd.getEmpRecord().getEmpAddr().getEmpAddrStreet().trim().length() == 0)
     erd.getEmpRecord().getEmpAddr().setEmpAddrStreet("Entry Required");

if (erd.getEmpRecord().getEmpAddr().getEmpAddrSt().trim().length() == 0)
     erd.getEmpRecord().getEmpAddr().setEmpAddrSt("TX");

if (erd.getEmpRecord().getEmpAddr().getEmpAddrZip().trim().length() == 0)
     erd.getEmpRecord().getEmpAddr().setEmpAddrZip("123451234");
```

In Listing 1-12, note the use of the `HttpSession` `putValue` to save a reference to the DataView. The `doGet()` processing continues on return. This data is be presented in the displayed form.

Listing 1-12 Finish Implementation for `doGetSetup`

```
s.putValue("customCrud",(Object)erd);
      return erd;
}
```

Step 4: Create Implementation for doPostSetup

In Listing 1-13, the DataView passed in contains values entered into the form by the application user. (The `HttpSession` is also available for use at this point, if required.) Refer to the *BEA WebLogic Java Adapter for Mainframe Programming Guide* for more information on `doPostSetup`.

Listing 1-13 Create Implementation for `doPostSetup`

```
public DataView doPostSetup(DataView dv, HttpSession s)
{
     empRecData erd = (empRecData)dv;
```

In Listing 1-14, note the use of group-level accessors to obtain fields. This code checks for original defaults, as well as missing data. `SocialSecurity` is a `BigDecimal` object. Validation can be simple or complex as required.

Listing 1-14 Continue implementation for `doPostSetup`

```
if(erd.getEmpRecord().getEmpSsn().compareTo(BigDecimal.valueOf(0L
)) == 0)
      throw new Error("Social Security Number Is Required");
if(erd.getEmpRecord().getEmpSsn().compareTo(BigDecimal.valueOf(12
3121234L)) == 0)
      throw new Error("Social Security Number Is Required");
if (erd.getEmpRecord().getEmpName().getEmpNameLast() == null)
      throw new Error("Last Name Is Required");
if
(erd.getEmpRecord().getEmpName().getEmpNameLast().trim().length()
== 0)
      throw new Error("Last Name Is Required");
```

```
if
(erd.getEmpRecord().getEmpName().getEmpNameLast().trim().compareT
o("EntryRequired") == 0)
      throw new Error("Last Name Is Required");
```

In Listing 1-15, note the use of group-level accessors to obtain fields. This code checks for original defaults, as well as missing data. (Validation routines could have been split out by field.)

Listing 1-15 Continue Implementation of `doPostSetup`

```
if (erd.getEmpRecord().getEmpName().getEmpNameFirst() == null)
    throw new Error("First Name Is Required");
if (erd.getEmpRecord().getEmpName().getEmpNameFirst().trim().length == 0)
    throw new Error("First Name Is Required");
if (erd.getEmpRecord().getEmpName().getEmpNameFirst().trim().compareTo("Entry
      Required") == 0)
    throw new Error("First Name Is Required");
if (erd.getEmpRecord().getEmpAddr().getEmpAddrStreet() == null)
    throw new Error("Street Address Is Required");
if (erd.getEmpRecord().getEmpAddr().getEmpAddrStreet().trim().length() == 0)
    throw new Error("Street Address Is Required");
if (erd.getEmpRecord().getEmpAddr().getEmpAddrStreet().trim().compareTo("Entry
       Required") == 0)
      throw new Error("Street Address Is Required");
```

In Listing 1-16, notice the use of group-level accessors to obtain fields. This code checks for original defaults, as well as missing data. Depending on the application, it may be more advantageous to develop validations as separate methods. This development process enables routines to be developed and tested with a servlet and easily re-used in an EJB.

Listing 1-16 Continue Implementation for `doPostSetup`

```
if (erd.getEmpRecord().getEmpAddr().getEmpAddrSt() == null)
    throw new Error("State Abreviation Is Required");

if (erd.getEmpRecord().getEmpAddr().getEmpAddrSt().trim().length() == 0)
    throw new Error("State Abreviation Is Required");
```

```
if (erd.getEmpRecord().getEmpAddr().getEmpAddrSt()
                                    .trim().compareTo("TX") != 0)
    throw new Error("Texas Employees ONLY");

if (erd.getEmpRecord().getEmpAddr().getEmpAddrZip() == null)
    throw new Error("ZipCode Is Required");

if (erd.getEmpRecord().getEmpAddr().getEmpAddrZip().trim().length() == 0)
    throw new Error("ZipCode Is Required");

if erd.getEmpRecord().getEmpAddr().getEmpAddrZip().trim().compareTo
    ("123451234") == 0)
    throw new Error("ZipCode Is Required");
```

In Listing 1-17, the `HttpSession` is used to remove a reference to the DataView. This method prevents re-posting the same data twice. The `doPost` processing continues on return. This data is now passed to the mainframe.

Listing 1-17 Finish Implementation for `doPostSetup`

```
else
     s.removeValue("customCrud");
       return erd;
}
```

Step 5: Create Implementation for doPostFinal

In Listing 1-18, the `doPostFinal` occurs after mainframe transmission, but prior to re-display in the browser. This example clears entered data after it is sent to the mainframe. This step completes the custom servlet.

Listing 1-18 Create Implementation for doPostFinal

```
public DataView doPostFinal(DataView dv, HttpSession s)
{
empRecData erd = (empRecData)dv;
erd.getEmpRecord().setEmpSsn(BigDecimal.valueOf(0L));
```

```
erd.getEmpRecord().getEmpName().setEmpNameLast("");
erd.getEmpRecord().getEmpName().setEmpNameFirst("");
erd.getEmpRecord().getEmpName().setEmpNameMi("");
erd.getEmpRecord().getEmpAddr().setEmpAddrStreet("");
erd.getEmpRecord().getEmpAddr().setEmpAddrSt("");
erd.getEmpRecord().getEmpAddr().setEmpAddrZip("");
return erd;
}
```

Step 6: Update the jcrmgw.cfg File with Service Entries

Listing 1-19 shows definitions of the entries that are used when the corresponding Create/Read/Update/Delete form buttons are pushed; for example, the Create button triggers `empRecCreate` which invokes `DPLDEMOC`. The gateway must be restarted for the new services to take effect.

Listing 1-19 Update jcrmgw.cfg File

```
empRecCreate          RDOM="CICS410"
                      RNAME="DPLDEMOC"
empRecRead            RDOM="CICS410"
                      RNAME="DPLDEMOR"
empRecUpdate          RDOM="CICS410"
                      RNAME="DPLDEMOU"
empRecDelete          RDOM="CICS410"
                      RNAME="DPLDEMOD"
```

Step 13: Create Basic Three-Part HTML Frame

In Listing 1-20, the primary frame (identified as “main” in the HTML code) displays the servlet, while an auxiliary frame provides links to HELP pages. The “Built on BEA WebLogic” logo is also displayed. A single line of Java script is used to ensure the window displays in the foreground.

Listing 1-20 Create Basic Three-Part HTML Frame

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<html>
```

```
  <head>
    <title>eGen</title>
  </head>
<script language="javascript">
<!--
if (window.focus) {self.focus();} // -->
</script>
  <FRAMESET cols="20%, 80%">
    <FRAMESET rows="20%, 80%">
      <FRAME src="bea_built_on_wl.gif" name="logo">
        <FRAME src="panel.html" name="aux">
    </FRAMESET>
   <FRAME src="http://machine.domain.com:7001/empRec" name="main">
  </FRAMESET>
</html>
```

Step 14: Create a Series of Links to HELP Pages

Listing 1-21 shows how the HTML can display as a sidebar frame. The `intro.html`, `emprec.html`, and `create.html` can display inside the "main" frame to provide basic HELP.

Listing 1-21 Creating a Series of HELP Page Links

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<html>
  <head> <title>eGen help</title>  </head>
<script language="javascript">
<!--
if (window.focus) {self.focus();} // -->
</script>
<body>
<TABLE summary="This table contains links to help pages.">
<TR> <TH>empRec Info</TH>
<TR> <TD><a href="intro.html" target="help">Introduction </a>
<TR> <TD><a href="emprec.html" target="help">EmpRec </a>
<TR> <TD><a href="create.html" target="help">Create </a>
<TR> <TD><a href="read.html" target="help">Read </a>
<TR> <TD><a href="update.html" target="help">Update </a>
<TR> <TD><a href="delete.html" target="help">Delete </a>
</TABLE>
</body>
</html>
```

Task 3: Update the JAM Configurations and Update BEA WebLogic Server web.xml File

Update the `jcrmgw.cfg` file with the remote service entries shown in Listing 1-22. The Java gateway must be restarted for new services. The entries are used when the corresponding form button is pushed. For example, the `Create` button triggers `empRecCreate`, which invokes `DPLDEMOC`. The service name must match values in the eGen script. In this example, the RNAME must match an actual CICS program name.

Listing 1-22 Remote Service Entries for Create/Read/Update/Delete

```
empRecCreate          RDOM="CICS410"
                      RNAME="DPLDEMOC"
empRecRead            RDOM="CICS410"
                      RNAME="DPLDEMOR"
empRecUpdate          RDOM="CICS410"
                      RNAME="DPLDEMOU"
empRecDelete          RDOM="CICS410"
                      RNAME="DPLDEMOD"
```

Update the WebLogic Server `web.xml` file with the entries shown in Listing 1-23. For more information, see the WebLogic Server documentation.

Listing 1-23 Update WebLogic Server web.xml File

```
<?xml version="1.0" ?>
<!DOCTYPE web-app (View Source for full doctype...)>
  <web-app>
  <servlet>
<servlet-name>customCrud</servlet-name>
<servlet-class>customCrud</servlet-class>
</servlet>
- <servlet-mapping>
<servlet-name>customCrud</servlet-name>
<url-pattern>customCrud</url-pattern>
```

```
</servlet-mapping>
</web-app>
```

Task 4: Deploy Your Application

At this point, you have created a basic form capable of receiving data entry, along with some static HTML code for display. For a complete description of how to deploy a servlet, refer to the WebLogic Server documentation. For evaluation purposes, refer to the *BEA WebLogic Server Quick Start Guide*.

Task 5: Use the Application

Figure 1-1 shows the default servlet with customized code displayed in an HTML frame. This type of servlet is useful for presentation, proof-of-concept, and as a test bed for development.

Figure 1-1 New Data Entry Servlet Display

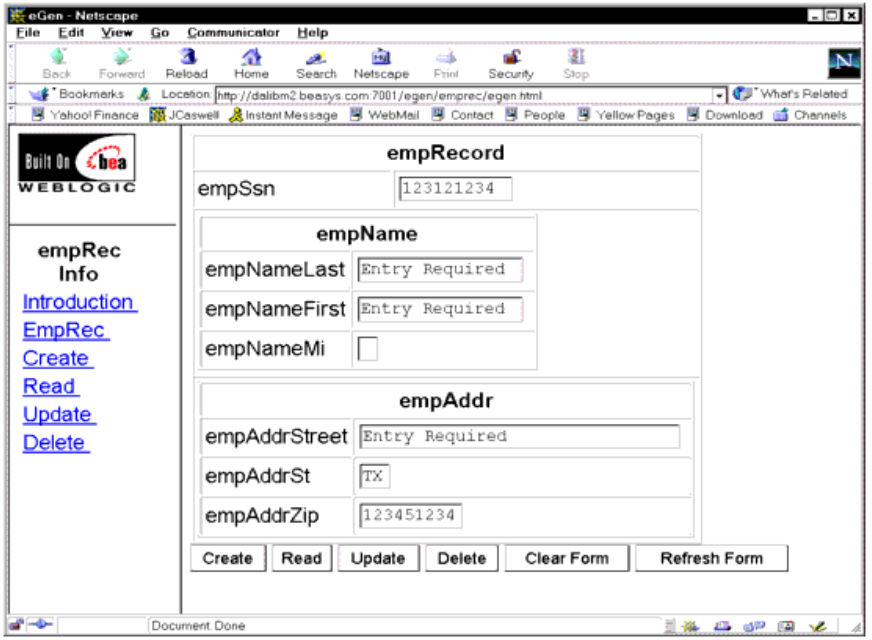


Figure 1-2 shows the servlet with the Create HELP page displayed in a new window on top of the application window.

Figure 1-2 Servlet with HELP Page Displayed

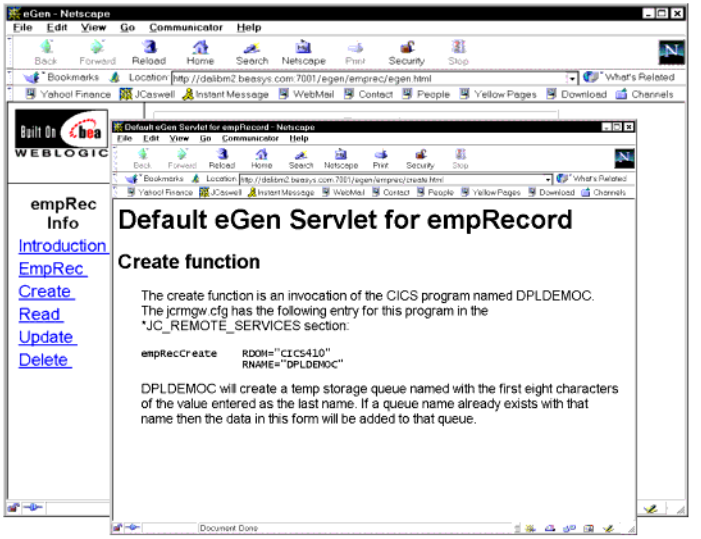
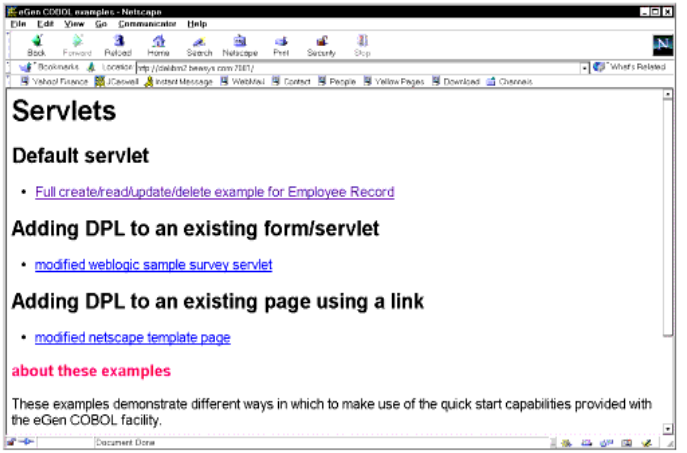


Figure 1-3 is an example of the page used for the front end of the new custom servlet.

Figure 1-3 New Data Entry Servlet Front End Page



Sample COBOL Programs Invoked by the Multi-Service Data Entry Servlet

The following section describe show COBOL programs for each of these button and service combinations:

- Create (`DPLDEMOC`)
- Read (`DPLDEMOR`)
- Update (`DPLDEMOU`)
- Delete (`DPLDEMOD`)

All of these programs make use of a CICS temporary storage queue for data. This simple technique is useful for testing and demonstrations.

Create

The simple program shown in Listing 1-24 writes a temporary storage queue using the first eight characters of the employee name as the `QID`.

Listing 1-24 COBOL Program for Create (`DPLDEMOC`)

```
IDENTIFICATION DIVISION.
       PROGRAM-ID.    DPLDEMOC.
       INSTALLATION.
       DATE-COMPILED.
       ENVIRONMENT DIVISION.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01  TSQ-DATA-LENGTH    PIC S9(4) COMP VALUE ZERO.
       01  TSQ-NAME.
              05 TSQ-ID       PIC X(8) VALUE SPACES.
              05 FILLER       PIC X(30) VALUE SPACES.
       LINKAGE SECTION.
       01  DFHCOMMAREA.
              COPY EMPREC.
```

```
       PROCEDURE DIVISION.
       MAINLINE SECTION.
              MOVE EMP-NAME TO TSQ-NAME
              MOVE LENGTH OF EMP-RECORD
              TO  TSQ-DATA-LENGTH
              EXEC CICS WRITEQ TS
                     QUEUE(TSQ-ID)
                     FROM(EMP-RECORD)
                     LENGTH(TSQ-DATA-LENGTH)
              END-EXEC.
              EXEC CICS RETURN
              END-EXEC.
               EXIT.
```

Read

The simple program shown in Listing 1-25 reads a temporary storage queue using the first eight characters of the employee name as the `QID`. If the read fails, the `COMMAREA` is reset so that residual data does not appear as the result of a read.

Listing 1-25 COBOL Program for Read (`DPLDEMOR`)

```
IDENTIFICATION DIVISION.
       PROGRAM-ID.    DPLDEMOR.
       INSTALLATION.
       DATE-COMPILED.
       ENVIRONMENT DIVISION.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01  TSQ-DATA-LENGTH   PIC S9(4) COMP VALUE ZERO.
       01  TSQ-RESP    PIC S9(4) COMP VALUE ZERO.
       01  TSQ-NAME.
             05 TSQ-ID   PIC X(8) VALUE SPACES.
             05 FILLER   PIC X(30) VALUE SPACES.
           LINKAGE SECTION.
       01  DFHCOMMAREA.
              COPY EMPREC.
       PROCEDURE DIVISION.
       MAINLINE SECTION.
              MOVE EMP-NAME TO TSQ-NAME
              MOVE LENGTH OF EMP-RECORD
              TO TSQ-DATA-LENGTH
```

```
                EXEC CICS READQ  TS
                       ITEM(1)
                       INTO(EMP-RECORD)
                       QUEUE(TSQ-ID)
                       LENGTH(TSQ-DATA-LENGTH)
                       RESP(TSQ-RESP)
                END-EXEC.
                IF TSQ-RESP NOT EQUAL ZERO
                       MOVE ZEROS  TO EMP-SSN
                       MOVE SPACES TO EMP-NAME-FIRST
                       MOVE SPACES TO EMP-NAME-MI
                       MOVE SPACES TO EMP-ADDR
                END-IF
                EXEC CICS RETURN
                END-EXEC.
```

Update

The simple program shown in Listing 1-26 deletes a temporary storage queue using the first eight characters of the employee name as the `QID`. It then creates a new queue with the `COMMAREA` provided.

Listing 1-26 COBOL Program for Update (`DPLDEMOU`)

```
IDENTIFICATION DIVISION.
       PROGRAM-ID.   DPLDEMOU.
       INSTALLATION.
       DATE-COMPILED.
       ENVIRONMENT DIVISION.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01  TSQ-DATA-LENGTH  PIC S9(4) COMP VALUE ZERO.
       01  TSQ-NAME.
              05 TSQ-ID     PIC X(8) VALUE SPACES.
              05 FILLER     PIC X(30) VALUE SPACES.
       LINKAGE SECTION.
       01  DFHCOMMAREA.
              COPY EMPREC.
       PROCEDURE DIVISION.
       MAINLINE SECTION.
              MOVE EMP-NAME TO TSQ-NAME
              MOVE LENGTH OF EMP-RECORD
```

```
                TO TSQ-DATA-LENGTH
                EXEC CICS DELETEQ TS
                       QUEUE(TSQ-ID)
                END-EXEC.
                EXEC CICS WRITEQ TS
                       QUEUE(TSQ-ID)
                       FROM(EMP-RECORD)
                       LENGTH(TSQ-DATA-LENGTH)
                END-EXEC.
                EXEC CICS RETURN
                END-EXEC.
                EXIT.
```

Delete

This simple program shown in Listing 1-27 deletes a temporary storage queue using the first eight characters of the employee name as the `QID`. The `COMMAREA` is reset so that residual data does not remain after the delete.

Listing 1-27 COBOL Program for Delete (`DPLDEMOD`)

```
IDENTIFICATION DIVISION.
       PROGRAM-ID.    DPLDEMOD.
       INSTALLATION.
       DATE-COMPILED.
       ENVIRONMENT DIVISION.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01  TSQ-DATA-LENGTH  PIC S9(4) COMP VALUE ZERO.
       01  TSQ-NAME.
              05 TSQ-ID     PIC X(8) VALUE SPACES.
              05 FILLER     PIC X(30) VALUE SPACES.
       LINKAGE SECTION.
       01  DFHCOMMAREA.
              COPY EMPREC.
       PROCEDURE DIVISION.
       MAINLINE SECTION.
              MOVE EMP-NAME TO TSQ-NAME
              MOVE LENGTH OF EMP-RECORD
              TO  TSQ-DATA-LENGTH
              EXEC CICS DELETEQ TS
                     QUEUE(TSQ-ID)
```

```
END-EXEC.
MOVE SPACES
TO DFHCOMMAREA
MOVE ZEROS  TO EMP-SSN
EXEC CICS RETURN
END-EXEC.
EXIT.
```

2 Enhancing an Existing Servlet to Originate a Mainframe Request

This scenario illustrates how to enhance an existing servlet to originate a mainframe request using WebLogic Server. In this scenario, a new application is developed and existing applications are updated. WebLogic Server samples are used to illustrate any existing applications. All discussions are from the application developer's point of view, presume a properly installed and configured environment, and presume an appropriate mainframe application is available.

Note: Although the sample code in this section represents typical applications, it is intended for example only and is not supported for actual use.

Action List

The following table lists the actions to develop a multi-service data entry servlet:

	Your action...	Refer to...
1	Verify that the prerequisite tasks have been completed.	"Prerequisites"
2	Create the `survey` servlet.	"Task 1: Obtain the survey Servlet"

	Your action...	Refer to...
3	Use eGen COBOL Code Generator to create an application.	"Task 2: Use eGen COBOL Code Generator to Generate a Base Class"
4	Update the `survey` servlet using the generated class.	"Task 3: Update the survey Servlet Using the Generated Class"
5	Update the JAM configurations and update the WebLogic Server configuration.	"Task 4: Update the JAM Configurations and Update WebLogic Server web.xml File"
6	Deploy your application.	"Task 5: Deploy Your Application"
7	Use the application.	"Task 6: Use the Application"

Prerequisites

Before you begin, ensure that the following prerequisites have been completed.

	Your action...	Refer to...
1	Verify that the required software has been properly installed.	BEA WebLogic Server documentation, *BEA WebLogic Java Adapter for Mainframe Installation Guide*
2	Verify that the environment and the software components have been properly configured.	BEA WebLogic Server documentation, *BEA WebLogic Java Adapter for Mainframe Configuration and Administration Guide*.
3	Verify the appropriate mainframe application is available.	Your mainframe system administrator.
4	Review the steps to develop a java application.	*BEA WebLogic Java Adapter for Mainframe Programming Guide*

Enhancing a Multi-Service Data Entry Servlet

To enhance a multi-service data entry servlet, complete the following tasks.

Task 1: Obtain the survey Servlet

Use the WebLogic Server `survey` servlet and add a mainframe request to the `post` routine. In future steps, you will add the code to the postprocessing routine to create a mainframe buffer and send it to CICS where an application writes the buffer to a temporary storage queue and returns.

Task 2: Use eGen COBOL Code Generator to Generate a Base Class

Identify the mainframe application and obtain its COBOL copybook, usually a CICS `DFHCOMAREA` or the user data portion of an IMS queue record layout. The copybook's name in this discussion is `survey.cbl`, shown in Listing 2-1.

Listing 2-1 Mainframe Application COBOL Copybook `survey.cbl`

```
02  survey-record.
       05  survey-ide       pic x(12).
       05  survey-emp       pic x(12).
       05  survey-cmt       pic x(256).
```

Step 1: Prepare eGen Script

In Listing 2-2, both the DataView `surveyData` and the client class `SurveyClient` are generated from the copybook `survey.cbl`.

Listing 2-2 Basic eGen script

```
view surveyData from survey.cbl
service doSurvey accepts surveyData returns surveyData
client class SurveyClient
{
       method doSurvey is service doSurvey
}
```

You are now finished creating the `survey.egen` script file and are ready to generate the source code.

Step 2: Generate the Java Source Code

As shown in Listing 2-3, invoke the eGen COBOL Code Generator to create the base class. This action makes java class files (`*java.class`) available for servlet customizing. The `surveyData.java` is the DataView object for `survey.cbl`.

Warning: `CLASSPATH` should have both the WebLogic Server subdirectories and the `jam.jar` file; otherwise, the compile fails.

Note: You could create a script file containing the eGen COBOL command line, along with the `javac` command to make the invocation easier.

Listing 2-3 Generating the Java Source Code

```
>egencobol survey.egen
>ls  *.java
SurveyServlet.java surveyData.java SurveyClient.java
>tasks
```

Step 3: Review the Java Source Code

Obtain a list of accessors for later use. Review the eGen COBOL output to become familiar with the information presented in this section.

Note: Each COBOL group item has its own accessor. The group name represents a nested inner class that must be accessed in order to retrieve the members.

In Listing 2-4, the output from the `grep` command shows the relationships in reverse order, for example:

```
getSurveyRecord().getSurveyIde()
```

This relationship is illustrated in the actual code example shown subsequently in this scenario.

Listing 2-4 Review the Java Source Code

```
grep get surveyData.java
       public String        getSurveyIde()
       public String        getSurveyEmp()
       public String        getSurveyCmt()
       public SurveyRecordV getSurveyRecord()
grep set surveyData.java
       public void          setSurveyIde(String value)
       public void          setSurveyEmp(String value)
       public void          setSurveyCmt(String value)
```

Task 3: Update the survey Servlet Using the Generated Class

The preferred customization method is to derive a custom class from the generated application. You are now ready to update the WebLogic Server example `survey` servlet.

Step 1: Start with Imports

In Listing 2-5, `bea.jam.egen` provides the `eGen COBOL` client and DataView base. The `surveyData` is the specific DataView generated from the COBOL copybook. `SurveyClient` is the generated client class.

Listing 2-5 Using Imports to Start Creating the Custom Application

```
import bea.jam.egen.*;
import surveyData;
import SurveyClient;
```

Step 2: Add New Data Members

In Listing 2-6, the code adds a private member for `SurveyClient`, which can be created in the `init()` function because there is no state for it. The `init()` is then updated for a new member. The `SurveyClient` obtains a connection factory when created. A single instance of `SurveyClient` can serve all requests.

Listing 2-6 Adding New Data Members

```
init ()

//Add private member for SurveyClient
private SurveyClient egc = null;
//Update init() for new member
egc = new SurveyClient();
```

Step 3: Update doPost with Mainframe Request

Listing 2-7 shows the local variables for form data and DataView in `doPost`. The DataView is the minimum requirement. The `values` entry has been declared previously.

Listing 2-7 Update `doPost` with Mainframe Request

```
values = req.getParameterNames();
surveyData        sd  = new surveyData();
```

Step 4: Continue Updating doPost by Extracting Form Data

In Listing 2-8, the code loops through the form using DataView accessors to set data. The `submit` field is skipped. The `surveyData` accessors are used to set values for `ide`, `employee`, and `comment`. The `surveyData` object represents the mainframe message buffer that ultimately is used to make the request. (The `surveyData` class was generated using the eGen COBOL Code Generator with the mainframe COBOL copybook.)

Listing 2-8 Continue Updating `doPost`

```
while(values.hasMoreElements())
{
       String name = (String)values.nextElement();
       String value = req.getParameterValues(name)[0];
              if(name.compareTo("submit") != 0)
              {
                if(name.compareTo("ide") == 0)
                     sd.getSurveyRecord().setSurveyIde(value);
                else if(name.compareTo("employee") == 0)
                     sd.getSurveyRecord().setSurveyEmp(value);
                else if(name.compareTo("comment") == 0)
                     sd.getSurveyRecord().setSurveyCmt(value);
              }
}
```

Step 5: Continue Updating doPost by Calling Mainframe Service

In Listing 2-9, the code shows how to make the mainframe request. The `doSurvey` command blocks until a response is provided. The call can throw either `IOException` or `snaException`. In this listing, `doSurvey` is in a try block that catches `IOException`. The `doSurvey` command returns a DataView that contains a response.

Listing 2-9 Continue Updating `doPost`

```
egc.doSurvey(sd);
```

The snaException is the base class for several exceptions, shown in Listing 2-10. A time-out is the most likely error an application would get.

Listing 2-10 Mainframe Exceptions

```
snaException
       jcrmConfigurationException
       snaCallFailureException
       snaLinkNotFoundException
       snaNoSessionAvailableException
       snaRequestTimeoutException
       snaServiceNotReadyException
```

Task 4: Update the JAM Configurations and Update WebLogic Server web.xml File

In Listing 2-11, update the jcrmgw.cfg file with the remote service name doSurvey. The Java gateway must be restarted for new services to take effect. The RNAME DPLSURVY is a CICS program that exists on the mainframe.

Listing 2-11 Update the jcrmgw.cfg File with Service Name

```
doSurvey        RDOM="CICS410"
                 RNAME="DPLSURVY"
```

Update the WebLogic Server web.xml file with the entries shown in Listing 2-12.

Listing 2-12 Update WebLogic Server web.xmlFile

```
<?xml version="1.0" ?>
<!DOCTYPE web-app (View Source for full doctype...)>
- <web-app>
- <servlet>
<servlet-name>survey</servlet-name>
```

```
<servlet-class>examples.servlets.SurveyServlet</servlet-class>
</servlet>
- <servlet-mapping>
<servlet-name>survey</servlet-name>
<url-pattern>survey</url-pattern>
</servlet-mapping>
</web-app>
```

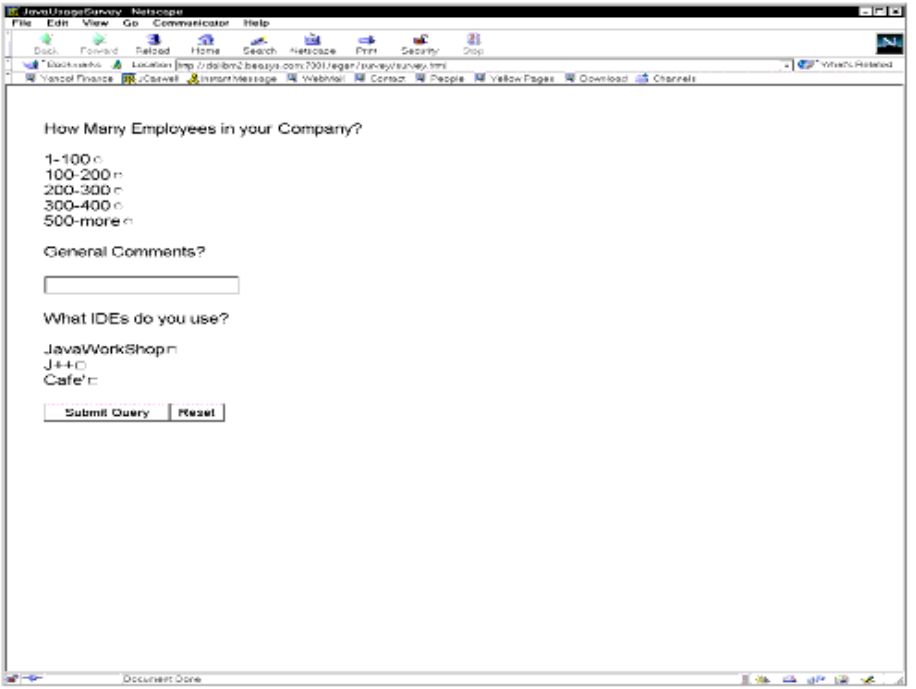
Task 5: Deploy Your Application

At this point, you have a basic form capable of making a maintenance request. For a complete description of how to deploy a servlet, refer to the WebLogic Server documentation. For evaluation purposes, refer to the *BEA WebLogic Server Quick Start Guide*.

Task 6: Use the Application

Figure 2-1 shows the HTML display of the enhanced application.

Figure 2-1 Enhanced Survey Servlet Display



Sample COBOL Program to Write to Temporary Storage Queue

The simple program shown in Listing 2-13 writes the contents of the `COMMAREA` to a temporary storage queue. This type of servlet is useful for testing, demonstrations, and new application development.

Listing 2-13 COBOL Program for `DPLSURVY`

```
IDENTIFICATION DIVISION.
       PROGRAM-ID.    DPLSURVY.
       INSTALLATION.
       DATE-COMPILED.
       ENVIRONMENT DIVISION.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01  TSQ-DATA-LENGTH         PIC S9(4) COMP VALUE ZERO.
       01  TSQ-ID                  PIC X(8) VALUE SPACES.
       LINKAGE SECTION.
       01  DFHCOMMAREA.
              COPY SURVEY.
       PROCEDURE DIVISION.
       MAINLINE SECTION.
              MOVE 'SURVEY' TO TSQ-NAME
              MOVE LENGTH OF SURVEY-RECORD
              TO  TSQ-DATA-LENGTH
              EXEC CICS WRITEQ TS
                     QUEUE(TSQ-ID)
                     FROM(SURVEY-RECORD)
                     LENGTH(TSQ-DATA-LENGTH)
              END-EXEC.
              EXEC CICS RETURN
              END-EXEC.
              EXIT.
```

Note: Some applications have extremely large `COMMAREA` copybooks. Distributed applications can be very sensitive to large amounts of data being transferred between components. If the Java application requires only a few fields from a large copybook, it would be advantageous to front-end the target application with a simpler program passing only the data required.

3 Updating an Existing EJB to Service a Mainframe Request

This section contains a scenario that shows how to update an existing EJB to service a request from the mainframe. Practical examples for using JAM tools are presented as tasks with step-by-step procedures. In this scenario, a new application is developed and existing applications are updated. WebLogic Server samples are used to illustrate any existing applications. All discussions are from the application developer's point of view, presume a properly installed and configured environment, and presume an appropriate mainframe application is available.

Note: Although the sample code in this section represents typical applications, it is intended for example only and is not supported for actual use.

Action List

The following table lists the actions to update an existing EJB to service a mainframe request:

	Your action...	Refer to...
1	Verify that prerequisite tasks have been completed.	"Prerequisites"

	Your action...	Refer to...
2	Use eGen COBOL to create a base java class.	"Task 1: Use eGen COBOL Code Generator to Generate a Base Class"
3	Update the trader interface using the generated java class.	"Task 2: Update the Trader Interface Using the Generated Class"
4	Update the JAM configurations.	"Task 3: Update the JAM Configurations"
5	Deploy your application.	"Task 4: Deploy Your Application"
6	Use the application.	"Task 5: Use the Application"

Prerequisites

Before you begin, ensure that the following prerequisites have been completed.

	Your action...	Refer to...
1	Verify that the required software has been properly installed.	BEA WebLogic Server documentation, *BEA WebLogic Java Adapter for Mainframe Installation Guide*
2	Verify that the environment and the software components have been properly configured.	*BEA WebLogic Java Adapter for Mainframe Configuration and Administration Guide*
3	Verify the appropriate mainframe application is available.	Your mainframe system administrator.
4	Review the steps to develop a java application.	*BEA WebLogic Java Adapter for Mainframe Programming Guide*
5	Create the `survey` servlet prior to attempting the enhancement discussed in this scenario.	*BEA WebLogic Java Adapter for Mainframe Programming Guide*

Update an Existing EJB to Service a Mainframe Request

To update an existing EJB to service a mainframe request, complete the following tasks.

Task 1: Use eGen COBOL Code Generator to Generate a Base Class

Use the WebLogic Server basic `statelessSession` `TraderBean` and update the interface to add a dispatch function that is given control upon receipt of an inbound request. The eGen COBOL client class code generation model is used. The `TraderBean` is designed to run from a stand-alone client and output a list of stock trades.

You should have successfully run the WebLogic Server basic `statelessSession` `TraderBean` prior to attempting the updates discussed in this scenario. You must then identify the mainframe application and obtain its COBOL copybook. This is typically a CICS `DFHCOMAREA` or the user data portion of an IMS queue record layout. The copybook's name in this discussion is `trader.cbl`, as shown in Listing 3-1.

Listing 3-1 Mainframe Application COBOL Copybook `trader.cbl`

```
02  TRADER-RECORD.
       05  CUSTOMER          PIC X(24).
       05  SYMBOL            PIC X(6).
       05  SHARES            PIC 9(7) COMP-3.
       05  PRICE             PIC 9(7) COMP-3.
```

Step 1: Prepare eGen COBOL Script

The single-line script in Listing 3-2 generates the DataView `traderData` from the copybook named `trader.cbl`. The script is then saved as `inboundEJB.egen`.

Listing 3-2 Basic eGen COBOL script

```
view traderData from trader.cbl
```

You are now finished creating the `inboundEJB.egen` script file and are ready to generate the source code.

Step 2: Generate the Java Source Code

In Listing 3-3, the eGen COBOL Code Generator is invoked to compile `trader.cbl` copybook and `inboundEJB.egen`. The `traderData.java` is the DataView object for `trader.cbl`.

Warning: `CLASSPATH` should have both the WebLogic Server subdirectories and the `jam.jar` file; otherwise, the compile fails.

Note: You could create a script file containing the eGen COBOL command line, along with the `javac` command to make the invocation easier.

Listing 3-3 Generating the Java Source Code

```
egencobol inboundEJB.egen
ls  traderDat*.java
traderData.java
javac traderData.java
```

Step 3: Review the Java Source Code

Obtain a list of accessors for use later. Look at the eGen COBOL output to become familiar with each of the scenarios presented in this section.

The entire method of customizing the generated output is based on deriving the output from generated code. The base application can be regenerated without destroying the custom code.

Note: Each COBOL group item has its own accessor. This is important because the group name represents a nested inner class that must be accessed in order to retrieve the members.

In Listing 3-4, the output from the `grep` command shows the relationships in reverse order, for example:

```
getTraderRecord().getPrice()
```

This relationship is illustrated in the actual code example shown in Listing 3-4.

Listing 3-4 Review the Java Source Code

```
grep get traderData.java
       public String            getCustomer()
       public String            getSymbol()
       public BigDecimal        getShares()
       public BigDecimal        getPrice()
       public TraderRecordV     getTraderRecord()
grep set traderData.java
       public void         setCustomer(String value)
       public void         setSymbol(String value)
       public void         setShares(BigDecimal value)
       public void         setPrice(BigDecimal value)
```

Task 2: Update the Trader Interface Using the Generated Class

Update the WebLogic Server `trader` example basic `statelessSession` bean.

Step 1: Start with Import

In Listing 3-5, the EJB interface is updated. In the `Trader` interface declaration, the `EJBobject` is replaced with `gwObject`. The `gwObject` extends `EJBObject` and provides the `dispatch` method that gets control on receipt of an incoming request.

Listing 3-5 Using Imports to Start Updating the EJB

```
import bea.sna.jcrmgw.gwObject;
.
.
.
public interface Trader extends gwObject {
.
.
.
```

Step 2: Continue with Imports

The code in Listing 3-6 performs four imports to update the EJB. The `bea.base.io.*` import provides the mainframe reader and writer. The `traderData` import is the specific DataView generated from the COBOL copybook. The `BigDecimal` class handles packed decimal `COMP-3` fields. The mainframe reader and writer can generate `IOExceptions`.

Listing 3-6 Continuing Imports

```
import bea.base.io.*;
import traderData;
import java.math.BigDecimal;
import java.io.IOException;
```

Step 3: Update EJB with dispatch

In Listing 3-7, the gateway invokes `dispatch` with a byte array of mainframe EBCDIC data. The code converts the mainframe byte array to a DataView using a `MainFrameReader`. The `traderData` is the generated DataView class.

Listing 3-7 Update EJB with `dispatch`

```
.
.
.
public byte[] dispatch(byte[] b)
       {
              traderData td = null;
              try {
                td = new traderData(new MainframeReader(b));
                catch(IOException ie) { return b; }
                      // error protocol required
```

Step 4: Continue Updating EJB with dispatch

In Listing 3-8, the code uses accessors to get input and set output. The mainframe `COMMAREA` is updated with the result. Note the use of an accessor to obtain the group-level class prior to accessing the member variable. An application-level error indicator in the data is used to convey the exception. Updating the DataView member results in updates to the mainframe application. Any application exception thrown from the `dispatch` routine results in an `abend` returned to the mainframe.

Listing 3-8 Continue Updating EJB with `dispatch`

```
try {
       TradeResult tr = buy(td.getTraderRecord().getCustomer()
              ,td.getTraderRecord().getSymbol()
              ,td.getTraderRecord().getShares().intValue());
       td.getTraderRecord().setShares(new
                     BigDecimal((long)tr.numberTraded));
       td.getTraderRecord().setPrice(new
                     BigDecimal((long)tr.priceSoldAt));
}catch(ProcessingErrorException pe)
       td.getTraderRecord().setSymbol("*ERROR");}
```

Step 5: Finish Updating EJB with dispatch

The code in Listing 3-9 converts the DataView back into a byte array to be returned to the mainframe using a `MainframeWriter`. The `MainframeWriter` and DataView handle conversions. Note that the `dispatch` function takes a byte array and returns a byte array. This process means when you set up an initial configuration, you can stub `dispatch` as an echo function.

Listing 3-9 Finish Updating EJB with `dispatch`

```
try {
                return td.toByteArray(new MainframeWriter());
    }           catch(IOException ie) {return b; }
                       // error protocol required
}
```

Task 3: Update the JAM Configurations

Update the `jcrmgw.cfg` file with the service name shown in Listing 3-10. The JAM gateway must be restarted for new services to take effect.

Listing 3-10 Update the `jcrmgw.cfg` File with Service Name

```
*JC_LOCAL_SERVICES
statelessSession.TraderHome     RNAME=”DPL1SVR”
```

Task 4: Deploy Your Application

Use the build function supplied with WebLogic Server to build the `statelessSession` example. The EJB is saved in `/myserver/ejb_basic_statelessSession.jar`. Deploy the EJB using the WebLogic Server Console.

To run the client, follow the instructions in the WebLogic Server documentation.

Warning: DataView classes are not included in the jar file using the default script. You must either add traderData*.class entries to the jar file or copy the entries to another location on the CLASSPATH. The EJB does not deploy if the `traderData` classes cannot be found.

Task 5: Use the Application

Listing 3-11 shows the inbound mainframe request for a "buy" transaction executed by the `traderBean`. If the previous tasks have been performed correctly, the result should look similar to this listing.

Listing 3-11 Inbound Mainframe Request

```
Thu Feb 17 15:31:10 CST 2000:<I> <EJB> EJB home interface:
'examples.ejb.basic.statelessSession.TraderHome' deployed bound to
the JNDI name: 'statelessSession.TraderHome'

Thu Feb 17 15:31:10 CST 2000:<I> <EJB> 0 EJBs were deployed using
.ser files.

Thu Feb 17 15:31:10 CST 2000:<I> <EJB> 1 EJBs were deployed using
.jar files.
.
.
.

**** Inbound Mainframe Request ****

buy (JEFF TESTER, WEBL, 150)

Executing stmt: insert into tradingorders (account, stockSymbol,
shares, price) VALUES ('JEFF TESTER','WEBL',150,10.0)
```

Sample COBOL Program to Write to Temporary Storage Queue

The simple program shown in Listing 3-12 writes the contents of the `COMMAREA` to a temporary storage queue. This type of simple mainframe program is useful for testing, demonstrations, and new application development.

Listing 3-12 COBOL Program for `DPL1CLT`

```
IDENTIFICATION DIVISION.
       PROGRAM-ID.    DPL1CLT.
       INSTALLATION.
       DATE-COMPILED.
       ENVIRONMENT DIVISION.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01  STUFF.
              COPY INBOUND.
       PROCEDURE DIVISION.
       MAINLINE SECTION.
              MOVE 'JEFF TESTER' TO CUSTOMER
              MOVE 'WEBL'        TO SYMBOL
              MOVE ZEROS         TO PRICE
              MOVE +150          TO SHARES
              EXEC CICS LINK
                     PROGRAM('DPL1SVR')
                     COMMAREA(STUFF)
              END-EXEC.
              EXEC CICS WRITEQ TS
                     QUEUE('TRADER')
                     FROM(STUFF)
              END-EXEC.
              EXEC CICS RETURN
              END-EXEC.
```

Note: Some applications have extremely large `COMMAREA` copybooks. Distributed applications can be very sensitive to large amounts of data being transferred between components. If the Java application requires only a few fields from a large copybook, it would be advantageous to preface the target application with a simpler program passing only the data required.

4 Web-enabling an IBM 3270 Application

This section contains a scenario that shows how to develop a single service servlet-based application that invokes a CrossPlex script on the mainframe when you are using WebLogic Server. Similar techniques may be used to interface to other third-party products. Because CrossPlex requires the use of a record header that should not be presented on a browser page, some DataView manipulation will be required.

This scenario is based on the general procedures presented in the *BEA WebLogic Java Adapter for Mainframe Programming Guide*. It gives practical examples for using JAM tools, presented as tasks with step-by-step procedures. In this scenario a new application is developed. All discussions are from the application developer's point of view, presume a properly installed and configured environment, and presume an appropriate mainframe application is available.

Note: Although the sample code in this section represents typical applications, it is intended for example only and is not supported for actual use.

Action List

The following table provides an action list for implementing JAM with CrossPlex:

	Your action...	Refer to...
1	Verify that all prerequisite activities have been completed.	"Prerequisites"

	Your action...	Refer to...
2	Create a CrossPlex script.	"Task 1: Create a CrossPlex Script"
3	Use eGen COBOL Code Generator to generate an application.	"Task 2: Use eGen COBOL to Create a Base Application"
4	Create your custom application from the starter application.	"Task 3: Create Your Custom Application from the Generated Application"
5	Update the JAM configurations and update WLS properties.	"Task 4: Update the JAM Configuration and WebLogic Server web.xml"
6	Deploy your application.	"Task 5: Deploy Your Application"
7	Use the application.	"Task 6: Use the Application"

Prerequisites

Before you begin, verify that the following prerequisites have been completed.

	Your action...	Refer to...
1	Verify that the required software has been properly installed.	BEA WebLogic product Installation Guides and SofTouch CrossPlex documentation
2	Verify that the environment and the software components have been properly configured.	BEA WebLogic product Installation Guides and SofTouch CrossPlex documentation
3	Verify the appropriate mainframe application is available.	Your mainframe system administrator
4	Review the steps to develop a java application.	*BEA WebLogic Java Adapter for Mainframe Programming Guide*

Implementing JAM with CrossPlex

To implement JAM with CrossPlex, complete the following tasks.

Task 1: Create a CrossPlex Script

A CrossPlex script provides the business logic to execute one or more 3270 transactions running on the mainframe. Transactions in any VTAM system, such as CICS or IMS, can be accessed. When a script executes in CrossPlex, it usually requires some input data, such as customer number and part number. This input data is passed from your application in a container called a record definition.

During execution, a script selects and optionally reformats data from the screen displays of the executed 3270 transactions. This selected data is returned to your application in a record definition.

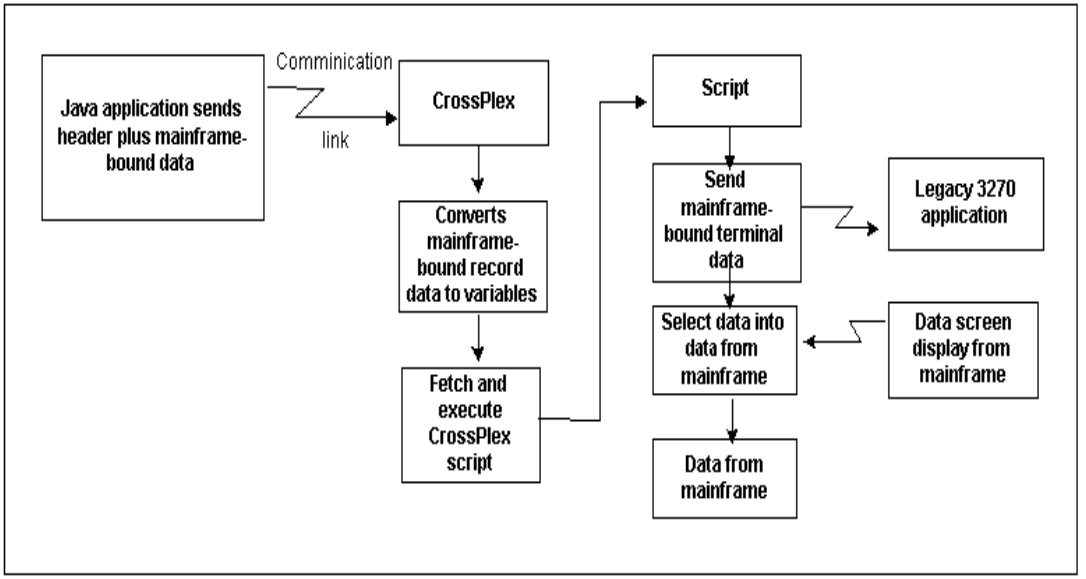
Note: Record definitions do not necessarily conform to any known data record in a file. A record definition is simply a description of a series of data fields being passed to and from a script.

Record definitions are created with the CrossPlex development system. An online editor is used to define each field in the record, along with its length and type (alpha, numeric, binary, packed). A single record definition may be used for data passing to and from the mainframe, or two definitions may be used.

Another of the CrossPlex development tools creates a COBOL copybook, using a record definition as input. The generated copybook is stored in a PDS member where it can be copied into your application program as needed.

Figure 4-1 illustrates the processing flow from the JAM front end to retrieve data from one or more mainframe transactions.

Figure 4-1 Processing Flow from JAM to Mainframe Transactions



Step 1: Prepare Record Definition for the Mainframe

Assign a record name and description, then define each data field to be passed to the CrossPlex script. The process of defining a record definition is described in detail in the *CrossPlex Middleware Programmer's Guide*.

To illustrate, assume the mainframe application is a simple name/address display that requires a customer number and company number as input. For this example, the record definition to and from the mainframe are different, though the same record definition can be used for both. Figure 4-2 shows how the record sent to the mainframe appears.

Figure 4-2 Illustration of a Record Sent to the Mainframe

```
____  Format  Sort  Delete  Exit(X)  Help                                EDRECORD
--------------------------------------------------------------------------------
                    CrossPlex Record Definition Edit
Record name INREC___
File name   ________
Description Sample_inbound_record_definition_____________________________

Cmd Fieldname                                Pos Length Type Occurs          Seq
*** CUSTNO_____________________________________ ___1  ___7   A    __1          __1
*** COMPANY____________________________________ ___8  ___3   N    __1            2
*** _____________________________________________ ___0  ___0   _    __0          ...
*** _____________________________________________ ___0  ___0   _    __0          ...
*** _____________________________________________ ___0  ___0   _    __0          ...
*** _____________________________________________ ___0  ___0   _    __0          ...
*** _____________________________________________ ___0  ___0   _    __0          ...
*** _____________________________________________ ___0  ___0   _    __0          ...
*** _____________________________________________ ___0  ___0   _    __0          ...
*** _____________________________________________ ___0  ___0   _    __0          ...
*** _____________________________________________ ___0  ___0   _    __0          ...
*** _____________________________________________ ___0  ___0   _    __0          ...
*** _____________________________________________ ___0  ___0   _    __0          ...
*** _____________________________________________ ___0  ___0   _    __0          ...

  Enter F1=Help F2=Keys F3=Exit F7=Bwd F8=Fwd F10=Actn
```

The data required by the mainframe transaction is `CUSTNO`, a seven-byte alphanumeric field beginning in position one of the record, and `COMPANY`, a three-byte numeric field beginning in position eight.

Step 2: Create a Copybook of the Record Definition Sent to the Mainframe

Store the generated copybook in a PDS member where you can easily copy it to your development system. For a complete description of the process of creating a COBOL copybook from a record definition, refer to the *CrossPlex Middleware Programmer's Guide*.

Continuing with the same example, a COBOL copybook generated from the previously illustrated record definition, `INREC`, appears as shown in Listing 4-1:

Listing 4-1 `INREC` Example

```
*****************************************************************
*       INREC     - Sample record definition sent to the m/f*
*****************************************************************
  01   INREC-START.
```

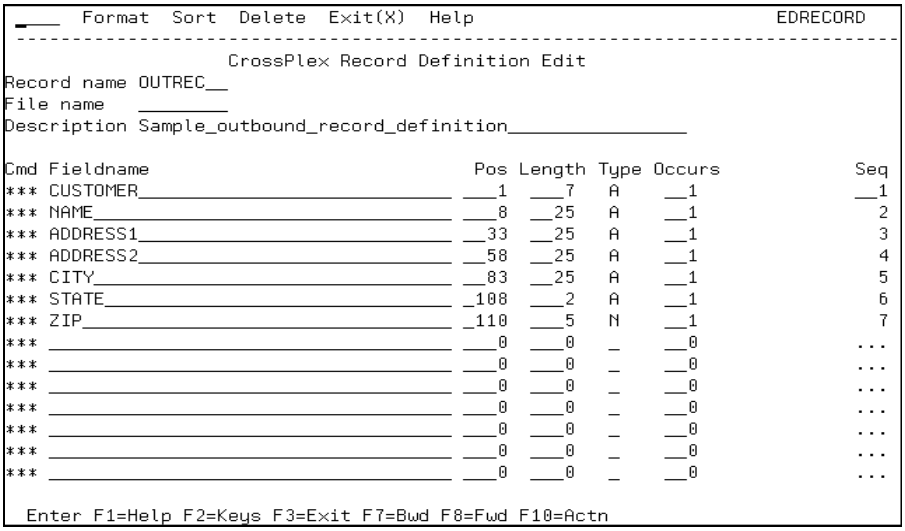
```
      05     INREC-CUSTNO     PIC X(007).
      05     INREC-COMPANY    PIC 9(003).
```

Step 3: Create a Record Definition and Copybook Sent From the Mainframe

If the data sent from the mainframe is to use a different record format from the data sent to the mainframe, repeat Steps 1 and 2 to prepare the record definition and copybook.

For this example, the record definition and copybook appears as in Figure 4-3.

Figure 4-3 Record Definition for Data Sent From the Mainframe



```
____  Format  Sort  Delete  Exit(X)  Help                                EDRECORD
------------------------------------------------------------------------------
                    CrossPlex Record Definition Edit
Record name OUTREC__
File name   ________
Description Sample_outbound_record_definition_________________

Cmd Fieldname                              Pos Length Type Occurs          Seq
*** CUSTOMER_____________________________ ___1  ___7   A    __1            __1
*** NAME_________________________________ ___8  __25   A    __1             2
*** ADDRESS1_____________________________ __33  __25   A    __1             3
*** ADDRESS2_____________________________ __58  __25   A    __1             4
*** CITY_________________________________ __83  __25   A    __1             5
*** STATE________________________________ _108  ___2   A    __1             6
*** ZIP__________________________________ _110  ___5   N    __1             7
***  _____________________________________ ___0  ___0   _    __0           ...
***  _____________________________________ ___0  ___0   _    __0           ...
***  _____________________________________ ___0  ___0   _    __0           ...
***  _____________________________________ ___0  ___0   _    __0           ...
***  _____________________________________ ___0  ___0   _    __0           ...
***  _____________________________________ ___0  ___0   _    __0           ...
***  _____________________________________ ___0  ___0   _    __0           ...

 Enter F1=Help F2=Keys F3=Exit F7=Bwd F8=Fwd F10=Actn
```

```
*****************************************************************
*      OUTREC - Sample record definition sent from the m/f      *
*****************************************************************
  01   OUTREC-START.
       05    OUTREC-CUSTOMER    PIC X(007).
       05    OUTREC-NAME        PIC X(025).
       05    OUTREC-ADDRESS1    PIC X(025).
       05    OUTREC-ADDRESS2    PIC X(025).
       05    OUTREC-CITY        PIC X(025).
       05    OUTREC-STATE       PIC X(002).
       05    OUTREC-ZIP        PIC 9(005).
```

Step 4: Prepare the CrossPlex Script

Scripts can be coded using the CrossPlex script editor, or they may be coded on any external editor and imported into the CrossPlex control file. The CrossPlex script language and the process of creating a script are described in the *CrossPlex Middleware Programmer's Guide*.

Note: In the CrossPlex documentation, scripts are also known as command streams and stream objects.

Prepare a script that navigates through a series of 3270 transactions in the same manner as a terminal operator. The script acts as a virtual operator, performing a log-on to the OLTP system, sending terminal data to the mainframe as if keyed on a keyboard, examining the returned screen display for correct execution, and selecting data from the screen if needed. Any number of transactions may be executed. The script language also provides a method of linking to a user program on the mainframe in order to perform direct retrieval of data that may not be available in a 3270 transaction display.

Continuing with the example of name/address data retrieval, the script might appear as Listing 4-2.

Listing 4-2 CrossPlex Script

```
        CALLCPX MSGAREA(NMAD)Initiate transaction NMAD.
    CALLCPX ROWCOL(05023) DATA(&CUSTNO)Send CUSTNO to row 5 col 23.
        IF ROWCOL(24021) EQ DATA(NOT ON FILE)-Verify customer
record found
            GOTO(NOTFOUND)
        SELECT RECORD(OUTREC) -Select data from mainframe
            ROWCOL(05023) RFIELD(CUSTNO) -screen into remaining
            ROWCOL(06023) RFIELD(NAME) -record fields.
            ROWCOL(07023) RFIELD(ADDR1) -
            ROWCOL(08023) RFIELD(ADDR2) -
            ROWCOL(09023) RFIELD(CITY) -
            ROWCOL(10023) RFIELD(STATE) -
            ROWCOL(11023) RFIELD(ZIP)
        GOTO(ENDJOB)Skip following error routine
NOTFOUND Enter if customer not found
        SELECT RECORD(OUTREC) -Move zeros to customer number
```

```
            DATA(0000000) RFIELD(CUSTNO)
ENDJOB Enter or fall through
        CALLCPX AID(PF3) Terminate NMAD transaction
```

Note: This example illustrates row/column addressing of screen data. CrossPlex also provides a method of assigning screen field names to avoid specific row/column references

Step 5: Test and Debug the Script

You can fully test and debug the script that executes on the mainframe without connecting it to your front-end application. CrossPlex provides a variety of execution and debugging tools to ensure the back-end portion of your application is operating properly.

When you are satisfied that the script is doing what you want and the returned data is correct, proceed to prepare the front-end of your application and connect the two together.

The process of testing and debugging a script is described in the *CrossPlex Middleware Programmer's Guide*.

Handling the Mainframe Sign-on

Most VTAM systems require the user to sign on in the target region when first connecting. You must also sign on when connecting to a target region with CrossPlex. This sign-on requirement can be handled in any one of the following ways:

- Interact with a user sign-on transaction in the script.

  The most common situation, especially for CICS, requires that your script handle the sign-on. Many users have CICS configured so that upon the first connection, the terminal is presented with a sign-on panel that may have been customized for the installation. If this is the case, the first CALLCPX command of the script returns the sign-on screen to the script and a subsequent CALLCPX must send a valid user ID and password. The mainframe sign-in is discussed in the *CrossPlex Middleware Programmer's Guide.*

- Let CrossPlex perform a short-form sign-on.

Supplying a valid user ID and password in the CrossPlex header will cause CrossPlex to perform a short-form sign-on before sending the first transaction data from the script.

Note: This case is valid for CICS systems only, and is installation dependent.

The short-form CICS sign-on may be disabled, depending on the user's CICS configuration. This case is discussed in the *CrossPlex Middleware Programmer's Guide*.

- Perform a mass log-on at CICS startup.

  With this technique, several FEPI virtual terminals are logged-on when CICS is first started and they remain active until CICS is recycled. If this is done, scripts do not need to be concerned with doing a sign-on at all. This topic is discussed in the *CrossPlex Web Enabling Guide*.

Task 2: Use eGen COBOL to Create a Base Application

Copy the CrossPlex COBOL copybooks to your development system. These copybooks include the copybook for the CrossPlex header (CSMF), the script invocation record definition (in this case INREC), and the script result record definition (in this case OUTREC). This scenario requires that you generate four DataView classes from these three copybooks, by merging them in the correct pattern. Table 4-1 lists the four DataView classes created from the three copybooks.

Table 4-1 Merge Pattern for DataView Classes

Purpose	Copybook(s) used	Combined Copybook Name
Initial form for presentation on browser	`INREC`	`INREC`
Record sent to mainframe	`CSMF + INREC`	`INREC-H`
Result returned from mainframe	`CSMF + OUTREC`	`OUTREC-H`
Result presented to user	`OUTREC`	`OUTREC`

When your application calls CrossPlex to retrieve data from the mainframe, it must pass a 256-byte header (CSMF), followed by the record area (INREC) to the mainframe. The data selected in the script will be returned in the record area (OUTREC) from the mainframe, which occupies the same memory address as the record to the mainframe, immediately following the header.

The CrossPlex header is described in the *CrossPlex Middleware Programmer's Guide*. Three copybooks are distributed to describe this area. A COBOL version called `XPLXCBL` is available, as well as a C version (`XPLXC`) and an Assembler version (`XPLXASM`).

In addition to the required fields listed in *Standardized Message Format*, two additional fields must be supplied by your application:

Table 4-2 Additional Standardized Message Format Fields

`XP-EXECUTING-SCRIPT`	The name of the CrossPlex script to execute.
`XP-INBOUND-RECORD`	The name of the record definition sent to the mainframe.
`XP-MODE`	Operating mode. Must contain `CMDR` to execute a script with a record definition as input.

The record definition from the mainframe is named in a `SELECT` statement within the script.

Listing 4-3 shows the COBOL version of the header copybook.

Listing 4-3 COBOL Version of Header Copybook

```
*****************************************************************
*                                                               *
*         XPLXCBL - CROSSPLEX STANDARDIZED MESSAGE FORMAT       *
*                       COBOL VERSION                          *
*                                                               *
*****************************************************************
 01  XP-COMMAREA.
     05  XP-COMMAND                PIC X(4).
     05  XP-RESPONSE               PIC S9(8).
     05  XP-EXCEP-DATA.
         10  XP-EXECP-ROWCOL       PIC S9(4) COMP.
         10  XP-EXCEP-LENGTH       PIC S9(4) COMP.
```

```
    10  XP-FLD-ERR              PIC S9(4) COMP.
    10  XP-EXCEP-MSG-FIELD     PIC S9(4) COMP.
    10  XP-EXCEP-FEPI          PIC X(4).
    10  XP-EXCEP-EIBRESP       PIC S9(8) COMP.
    10  XP-EXCEP-EIBRESP2      PIC S9(8) COMP.
05  XP-OPTIONAL-PARMLIST       PIC S9(8) COMP.
05  XP-TARGET                  PIC X(8).
05  XP-POOL                    PIC X(8).
05  XP-AIDBYTE                 PIC X(6).
05  XP-INSCREEN                PIC X(8).
05  XP-OUTSCREN                PIC X(8).
05  XP-CURSOR.
    10  XP-CURSOR-ROW          PIC S9(4).
    10  XP-CURSOR-COL          PIC S9(4).
05  XP-SIGNON-USERID           PIC X(8).
05  XP-SIGNON-PASSWORD         PIC X(8).
05  XP-NODENAME                PIC X(8).
05  XP-FEPI-CONVID             PIC X(8).
05  XP-DEBUG-QUEUE             PIC X(8).
05  XP-ASSOC-NAME              PIC X(8).
05  XP-MODE                    PIC X(4).
    88  XP-HTML                VALUE 'HTML'.
    88  XP-HTQS                VALUE 'HTQS'.
    88  XP-3270                VALUE '3270'.
    88  XP-CMDS                VALUE 'CMDS'.
    88  XP-CMDR                VALUE 'CMDR'.
05  XP-TRANSLATION-SCREEN      PIC X(8).
05  XP-IN-LENGTH               PIC S9(4).
05  XP-AREA-LENGTH             PIC S9(4).
05  XP-OUT-LENGTH              PIC S9(4).
05  XP-TERM-OPTION             PIC X(1).
    88  XP-NOTERM              VALUE 'N'.
05  XP-USD-OPTION             PIC X(1).
    88  UNSOLICITED-DATA-EXPECTED  VALUE 'N'.
05  XP-USD-WAIT-TIME          PIC S9(4) COMP.
05  FILLER                    PIC X(36).
05  XP-EXECUTING-SCRIPT       PIC X(8).
05  XP-FEPI-TIMEOUT           PIC S9(4) COMP.
05  FILLER                    PIC X(15).
05  XP-INBOUND-RECORD         PIC X(8).
05  FILLER                    PIC X(41).
05  XP-MESSAGE-AREA.
```

Step 1: Prepare eGen COBOL Script

In Listing 4-4, the DataViews are generated from the combined copybooks.

Listing 4-4 Basic eGen COBOL Script

```
view InrecRecord from INREC.cbl
view InrecHdrRecord from INREC-H.cbl
view OutrecRecord from OUTREC.cbl
view OutrecHdrRecord from OUTREC-H.cbl
```

Step 2: Add Service Entry

Add the single line service entry in Listing 4-5 for the CrossPlex operation. This entry specifies the DataView.

Listing 4-5 Service Names Associated with Input and Output Views

```
service DoIt accepts InrecHdrRecord returns OutrecHdrRecord
```

Step 3: Add Page Declarations in eGen COBOL Script

This application requires two pages: one to invoke the operation and another to present the results. Note that the full records (with header) are mentioned, even though these are not displayed. The custom code written later in the scenario specifies this display.

Listing 4-6 Page Declaration Associating Display Buttons with Services

```
page page1 "Invoke Operation" {
view InrecHdrRecord
       buttons {
               "doit" service(DoIt) shows resultPage
       }
}
page resultPage "Results of Operation" {
       view OutrecHdrRecord
       buttons {
             // No buttons on this page.
       }
}
```

Step 4: Add Servlet Name

As shown in Listing 4-7, `BaseServlet` is the servlet name to be registered as a URL in the WebLogic Server `web.xml` file. (Every servlet requires a URL to be registered this way. Refer to WebLogic Server documentation about deploying servlets for more specific information.) Here, the page "page1" is to be displayed when the servlet "BaseServlet" is invoked.

Listing 4-7 Add Servlet Name

```
servlet BaseServlet shows page1
```

The script is then saved as `crossplex.egen`.

Step 5: Generate the Java Source Code

In Listing 4-8, invoke the eGen COBOL Code Generator to create the application that is then compiled. This process makes class files (`*.class`) available for servlet customizing. `CLASSPATH` should include the WebLogic Server subdirectories and the `jam.jar` file; otherwise, the compile fails. You can create a script file containing the eGen COBOL command line, along with the `javac` command to make the invocation easier.

Listing 4-8 Generating the Java Source Code

```
egencobol emprec.egen
ls *.java
       BaseServlet.java
       InrecHdrRecord.java
       InrecRecord.java
       OutrecHdrRecord.java
       OutrecRecord.java
```

Task 3: Create Your Custom Application from the Generated Application

The preferred customizing method is to derive a custom class from the generated application. In this case, we will subclass the generated servlet code to both change record formats and manipulate CrossPlex header fields.

Step 1: Start with Imports

In Listing 4-9, `BigDecimal` supports `COMP-3` packed data. `HttpSession` is available for saving limited state. `DataView` is the base for all generated data records.

Listing 4-9 Using Imports to Start Creating the Custom Application

```
import java.util.Hashtable;
import javax.servlet.http.HttpSession;
import com.bea.dmd.dataview.DataView;
import InrecRecord;
import InrecHdrRecord;
import OutrecRecord
import OutrecHdrRecord;
import com.bea.dmd.dataview.HashtableLoader;
import com.bea.dmd.dataview.HashtableUnloader;
import com.bea.dmd.dataview.PrefixChanger;
```

Step 2: Declare the New Custom Class

Listing 4-10 shows how to extend the generated servlet. Extension of the generated servlet enables regeneration of the base application without destroying customized code. Fields can be added to the copybook without disrupting the customized code.

Listing 4-10 Declaring the New Custom Class

```
public class customServlet
     extends BaseServlet
```

```
{
:
```

Step 3: Add Implementation for doGetSetup

In Listing 4-11, the `doGetSetup ( )` function is used to ensure that the user is presented with a form reflecting the `INREC` record.

Listing 4-11 Add Implementation for doGetSetup

```
public DataView doGetSetup(DataView dv, HttpSession s){
return new InrecRecord ();
}
```

Step 4: Create Implementation for doPostSetup

The `doPostSetup` method performs operations after a button has been pressed on the form, prior to the mainframe call. In Listing 4-12, the DataView passed in contains values entered into the form by the application user. This code moves the specified data into an `InrecHdrRecord`; then sets the header fields for the operation you wish to perform.

Listing 4-12 Create Implementation for doPostSetup

```
public DataView doPostSetup(DataView dv, HttpSession s)
    {
         InrecHdrRecord bhr = new InrecHdrRecord();
         try
         {

           // Move the contents, by using a Hashtable as an
           intermediate holder.
           Hashtable h = new HashtableUnloader(new PrefixChanger
              (“mwdrecStart.”, “xpCommarea.”)).unload(dv);
              new HashtableLoader().load(h,(bhr);
           // Load header fields.
           bhr.getXpCommarea().setXpCommand("EXEC");
           bhr.getXpCommarea().setXpTarget("THISCICS");
```

```
            bhr.getXpCommarea().setXpPool("POOLM2");
            bhr.getXpCommarea().setXpFepiConvidBin(0L);
            bhr.getXpCommarea().setXpMode("CMDR");
            bhr.getXpCommarea().setXpInLength((short) 300);
            bhr.getXpCommarea().setXpAreaLingth((short) 1300);
            bhr.getXpCommarea().setXpExecutingScript("MYSCRIPT");
            bhr.getXpCommarea().setXpInboundRecord("INRECRECRD");

        }
        catch (Exception e)
        {
        }
        return bhr;
}
```

The meaning of each field in the CrossPlex header is described in the *CrossPlex Middleware Programmer's Guide*. For most executions, the following fields must contain meaningful data:

Table 4-3 CrossPlex Header Fields

COMMAND	Contains "EXEC" to execute a script, or "TERM" to terminate a session.
TARGET	Contains the FEPI target name of the VTAM region where transactions are to be executed.
POOL	Contains the FEPI pool name for this session.
ASSOC	Instead of TARGET and POOL, a CrossPlex Association can be named, which defines the target, pool and connection type (FEPI or BRIDGE).
MODE	Must contain "CMDR" if a record definition to the mainframe is used and a script is to be executed.
AREA-LENGTH	Contains the maximum length of MESSAGEAREA.
EXECUTING-SCRIPT	The name of the script to be executed.
INBOUND-RECORD	The name of the record definition sent to the mainframe.
MESSAGEAREA	Contains the record sent to the mainframe when CrossPlex is called and the record from the mainframe upon return.

USERID	To perform a short sign-on to the target region using FEPI, supply a valid user ID in this field.
PASSWORD	Valid password if USERID is present.
DEBUGQ	Name of a debug queue where execution trace records are to be written.
Upon return from CrossPlex, the following fields are supplied:	
NODENAME	The FEPI node name used by the mainframe session.
CONVID	The FEPI conversation ID assigned to the mainframe session.

On the first call to CrossPlex, all fields of the CSMF header must be completely initialized to their default values or filled with user data. The generated DataView code initializes with default values. Upon return, the header contains some fields provided by CrossPlex, such as the FEPI conversation ID. If subsequent calls to CrossPlex are made for the same session, these fields must not be re-initialized, since CrossPlex needs the FEPI conversation ID to continue the same session

Step 5: Create Implementation for doPostFinal

In Listing 4-13, the doPostFinal occurs after mainframe transmission, but prior to re-display in the browser. This example moves the result OutrecHdrRecord into an OutrecRecord prior to display.

Listing 4-13 Create Implementation for doPostFinal

```
public DataView doPostFinal(DataView dv, HttpSession s)
{

       OutrecHdrRecord qhr = (OutrecHdrRecord) dv;
       int resp=qhr.getXPCommarea().getXpCommarea().getXpResponse();
       if (resp != 0  &&  resp != 12)
             throw new Error("Bad xp-response: " + resp);

       OutrecRecord qr = new OutrecRecord();

       try
       {
           // Move the contents, by using a Hashtable as an
           intermediate holder.
```

```
            Hashtable h = new HashtableUnloader(new
            PrefixChanger("xpCommarea.", "mwdrecStart."))
                 .unload(dv);
            new HashtableLoader.load(h, qr);
            }
            catch (Exception e)
            {
            }

            return qr;
      }
```

Task 4: Update the JAM Configuration and WebLogic Server web.xml

Update the `jcrmgw.cfg` file with the remote service entries shown in Listing 4-14. The JAM gateway must be restarted for new services. The entries are used when the corresponding form button is pushed. The `doit` button triggers `DoIt`, which invokes `XPLXSBEA`. The service name must match values in the eGen COBOL script. In this example, the `RNAME` must match an actual CICS program name.

Listing 4-14 Remote Service Entries for Create/Read/Update/Delete

```
DoIt                  RDOM="CICS410"
                      RNAME="XPLXSBEA"
```

Update the WebLogic Server `web.xml` file with the entries shown in Listing 4-15.

Listing 4-15 Update WebLogic Server web.xml File

```
weblogic.httpd.register.crossplex=customServlet
```

Task 5: Deploy Your Application

At this point, you have a basic form capable of receiving data entry, along with some static HTML code for display. For a complete description of how to deploy a servlet, refer to the WebLogic Server documentation. For evaluation purposes, refer to the *BEA WebLogic Server Quick Start Guide*.

Task 6: Use the Application

Figure 4-4 shows the default servlet with customized code displayed in an HTML facade. This type of servlet is useful for presentation, proof-of-concept, and as a test bed for development.

Figure 4-4 New Data Entry Servlet Display

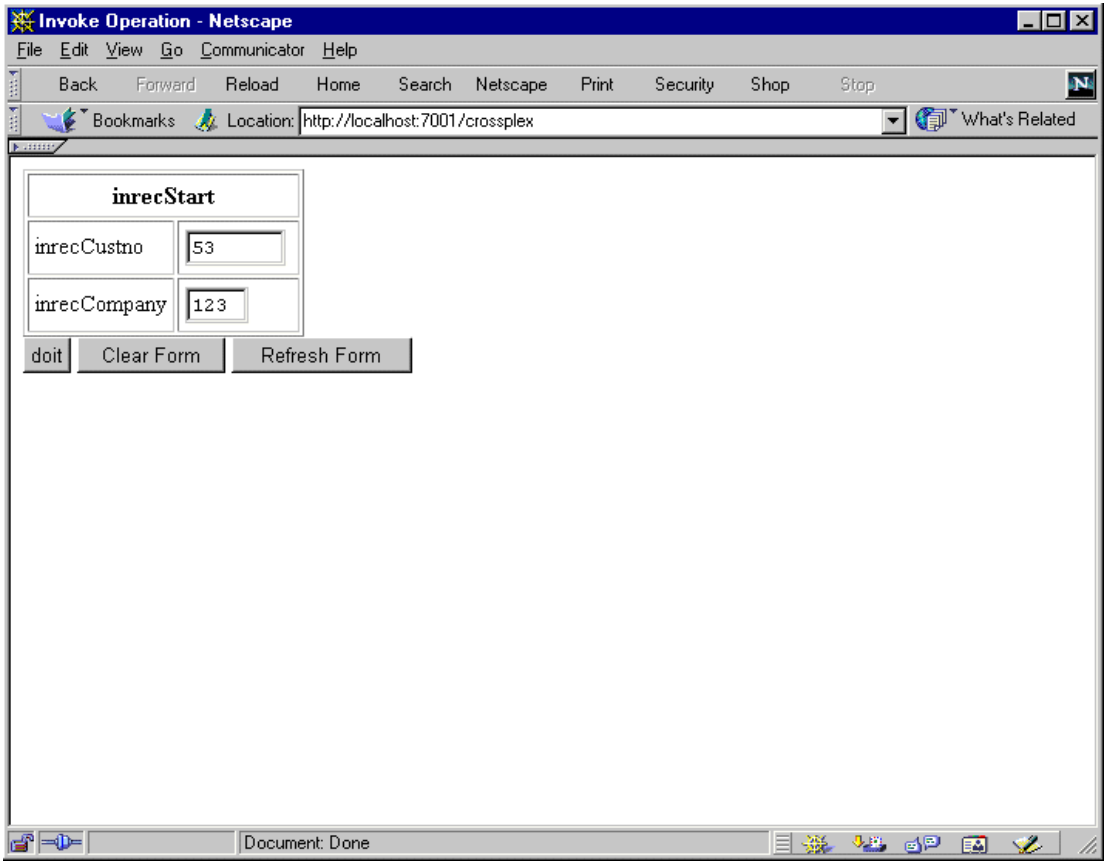
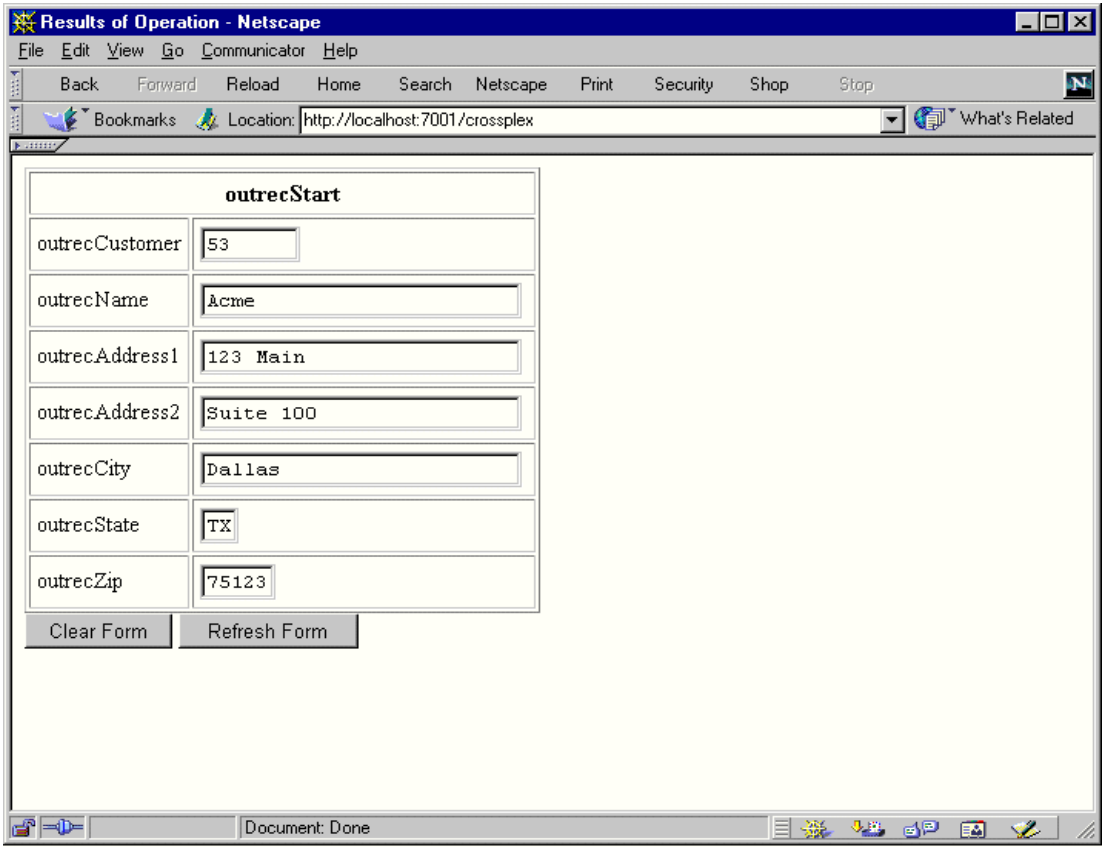


Figure 4-5 is an example of the page used for the front end of the new custom servlet.

Figure 4-5 New Data Entry Servlet Front End Page



5 Using JAM in a Clustered Environment

This scenario extends the EJB client model described in the *BEA WebLogic Java Adapter for Mainframe Programming Guide* to demonstrate a client requesting multiple employee actions against an EJB that is deployed in a cluster. The client holds a remote interface for each EJB on each WebLogic Server in the cluster on which it is deployed. A JAM gateway must be running on each WebLogic Server in the cluster. Each gateway is connected to a CRM running on the same machine or distributed to a different machine. The client is used to make multiple requests to the clustered EJB. The EJB writes a message to the WebLogic Server console, showing the distribution of the client requests.

Action List

To use JAM in a clustered environment, complete the following tasks.

	Your action...	Refer to...
1	Verify that prerequisite tasks have been completed.	"Prerequisites"
2	Prepare your system.	"Preparing Your System"
3	Run the sample.	"Running the Sample"

Prerequisites

Verify that the following prerequisite tasks have been completed.

	Your action...	Refer to...
1	Verify that the required software has been properly installed: WebLogic Server, WebLogic Java Adapter for Mainframe.	*BEA WebLogic Server Getting Started Guide, BEA WebLogic Java Adapter for Mainframe Installation Guide*
2	Verify that the environment and the software components have been properly configured.	*BEA WebLogic Server Administration Guide, BEA WebLogic Java Adapter for Mainframe Configuration and Adminstration Guide*
3	Verify the appropriate mainframe application is available.	Your mainframe system administrator

Preparing Your System

Complete the following steps to run the clustering scenario:

1. Add the WebLogic cluster information to the WebLogic Server domain where you will run your cluster samples.
2. Generate the EJB Client sample by following the steps described in the *BEA WebLogic Java Adapter for Mainframe Programming Guide*.
3. Add the services generated from the client sample to the JAM configuration file for each WebLogic Server in your clustered configuration.
4. Start the JAM gateway under each WebLogic Server. If the WebLogic Server is already started, or the JAM gateway is already running, use the admin servlet to perform the necessary steps. See the "Using JAM Administration Utilities" section of the *BEA WebLogic Java Adapter for Mainframe Programming Guide*.

Running the Sample

To run the clustering sample provided with the JAM product, complete the following steps.

1. Extend the base example by adding the `clusterSampleClientBean` to the sample container. The `clusterSampleClientBean` extends the functions of the `SampleClientBean methods readEmployee` and `newEmployee`. These methods will write a line to the WebLogic Server console.
2. The `clusterSampleClientBean` should be generated and packaged into the sample EJB.

Listing 5-1 clusterSampleClientBean.java

```
// ============================================================

// clusterSampleClientBean.java
// Example class that extends a generated JAM client EJB application.
//------------------------------------------------------------
  package sample;
// Imports
  import java.math.BigDecimal;
  import java.io.IOException;
  import com.bea.sna.jcrmgw.snaException;
// Local imports
  import sample.EmployeeRecord;
  import sample.EmployeeRecord.EmpRecord1V;
  import sample.SampleClientBean;
//*********************************************************

Extends the SampleClientBean EJB class, adding additional business logic.
*/
  public class clusterSampleClientBean
    extends SampleClientBean
{
// Public functions
*****************************************************************
    * Read an employee record.
    */
     public EmployeeRecord readEmployee(EmployeeRecord commarea)
         throws IOException, snaException
```

```
    {
        EmployeeRecord  erec = (EmployeeRecord) commarea;
        try {
        // Make the remote call.
           erec = super.readEmployee(commarea);
        // Log the results
           printEmployee("readEmployee : ", erec);
        } catch (Exception e) {
           log("Read Exception "  + e.toString() + " for "
              + erec.getEmpRecord().getEmpName().getEmpNameLast());
           throw new IOException();
        }
        //  Return the Employee Record
        return erec;
    }
/*******************************************************

    * Create a new employee record.
    */
    public EmployeeRecord newEmployee(EmployeeRecord commarea)
        throws IOException, snaException
    {
        EmployeeRecord  erec = (EmployeeRecord) commarea;
        try {
        // Make the remote call.
           erec = super.newEmployee(commarea);
        // Log the results
           printEmployee("newEmployee  : ", erec);
        } catch (Exception e) {
           log("Create Exception " + e.toString() + " for "
              + erec.getEmpRecord().getEmpName().getEmpNameLast());
           throw new IOException();
        }
        //  Return the Employee Record
        return erec;
    }
// Private Functions
/*********************************************************
        * Print the Employee Record
        */
        private void printEmployee(String title, EmployeeRecord emp)
        {
            EmpRecord1V empinfo = emp.getEmpRecord();
            log(title +
               empinfo.getEmpName().getEmpNameFirst() + " " +
               empinfo.getEmpName().getEmpNameMi() + " " +
               empinfo.getEmpName().getEmpNameLast() + ", " +
               empinfo.getEmpAddr().getEmpAddrStreet() + ", " +
               empinfo.getEmpAddr().getEmpAddrSt() + " " +
```

```
            empinfo.getEmpAddr().getEmpAddrZip());
     }
   private void log(String s) {
            System.out.println(s);
   }
}
```

3. Change the `ejb-jar.xml` file to use the extension classes. Change the `<ejb-class>` to reference `clusterSampleClientBean` as illustrated in the following example.

```
<enterprise-beans>
      <session>
        <ejb-name>SampleClient</ejb-name>
        <home>sample.SampleClientHome</home>
        <remote>sample.SampleClient</remote>
        <ejb-class>sample.clusterSampleClientBean</ejb-class>
        <session-type>Stateless</session-type>
        <transaction-type>Container</transaction-type>
      </session>
```

4. Deploy the sample to each of the machines in the cluster node by referencing each of the target machines on the EJB deployment list in the `WebLogic Server config.xml` as illustrated in the following example.

```
<Application Deployed="true" Name="SampleClient”
Path=.\config\mydomain\applications">
<EJBComponent Name="SampleClient" Targets="mach1,mach2,mach3"
URL="SampleClient.jar”"/>
</Application>
```

5. Perform a client request against the EJB that is deployed on the cluster. The client will look up the home interface for the clustered EJB and get the remote interface stub for each of the EJBs in the cluster. If you make multiple requests to the EJB, you should see the requests being clustered based on the cluster algorithm you selected under WebLogic Server.

In the example in Listing 5-2, the `ClientTest` sample is used to make the cluster requests.

Listing 5-2 ClientTest Sample

```
java sample.ClientTest –u “t3://cluster:7001” –c 10 –i 100
```

The options are in this example are defined in the following way:

`-u` option
: specifies the cluster alias and the port number the WebLogic cluster servers are listening on

`-c` option
: specifies 10 concurrent requests to be made, each one on its own thread

`-i` option
: specifies that 100 requests will be made on each of the concurrent requests

Index

S

T

U

W

X